

JAVA DEEL 2

H7 ARRAYS

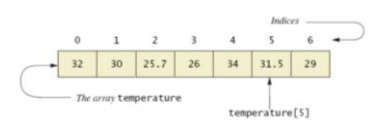
HH

CREËREN EN BENADEREN ARRAYS

- Een array is een speciaal soort object
- Beschouw het als een collectie van variabelen van hetzelfde type
- Creëren van een array van 7 variabelen van het type double

```
double[] temperature = new double[7];
```

- Om een element te benaderen gebruik je
 - de naam van de array
 - Een index tussen vierkante haken
- Array indices beginnen bij 0
- Een veelgebruikte manier om een array te visualiseren



- Syntax voor declaratie van een array met **new**

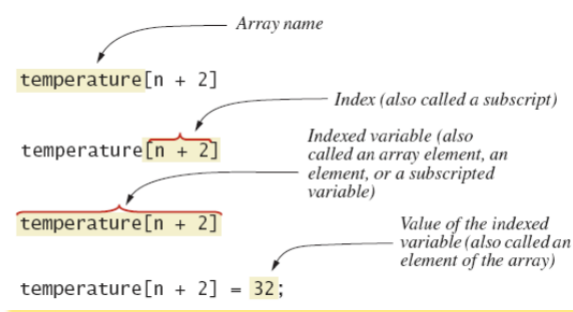
```
Base_Type[] Array_Name = new Base_Type[Length];
```

- Het aantal elementen van een array is de lengte van de array
- Het type van de array elementen is de base type van de array
- Bij een data type tijdens het declareren van een array

```
int[] pressure;
```
- Voor het afbakenen van een integer expressie dat de lengte van de array bepaalt

```
pressure = new int[100];
```
- Om een element van een array te benaderen aan de hand van de index

```
pressure[3] = keyboard.nextInt();
```



DE INSTANTIEVARIABLE `length`

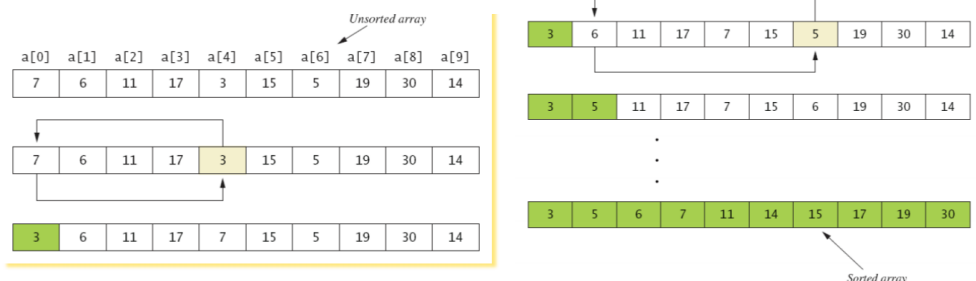
- Een array object heeft één enkele publieke instantievariabele
 - Variabele `length`
 - Bevat het aantal elementen van een array
 - Variabele is een constante en kan dus niet gewijzigd worden

SORTING AND SEARCHING ARRAYS

SELECTION SORT

We zetten de waarden van de indexen in een andere volgorde zodat:

$A[0] < A[1] < \dots < A[a.length-1]$



For (index= 0; index < a.length; index++)

place the (index +1)th smallest index in a[index] deze 'swap' noemen we interchange sorting algorithm.

Het grootste getal in een array vinden gebeurt op dezelfde manier als het kleinste getal vinden.

In deze array hebben we 2 delen. Een deel dat nog niet gesorteerd is en een deel die wel al gesorteerd is. De gesorteerde groeit met elke repetitie van de loop en de niet-gesorteerde neemt af. De loop eindigt bij $a[a.length-2]$ want de laatste is sowieso juist gesorteerd.

Selection sort is eenvoudig

– Maar heel inefficiënt voor grote arrays

Java Class Library bevat sorteer algoritmen

– Bevat een klasse `Arrays`

– Klasse heeft meerdere static sorteer methodes

– Zie later

`Arrays.sort(anArray, first, last)`

ZOEK ALGORITMEN

- Method gebruikt in `EenvoudigeLijst` is sequentieel zoeken
 - Zoeken start bij eerste tot en gaat tot en met laatste
 - Goed voor niet gesorteerde arrays
- Search stopt wanneer Item gevonden is of einde van de lijst is bereikt
- Indien lijst gesorteerd, gebruik meer efficiënte search algoritmes

In engels is de methode `isOnList` van de klasse `OneWayNoRepeatsList`

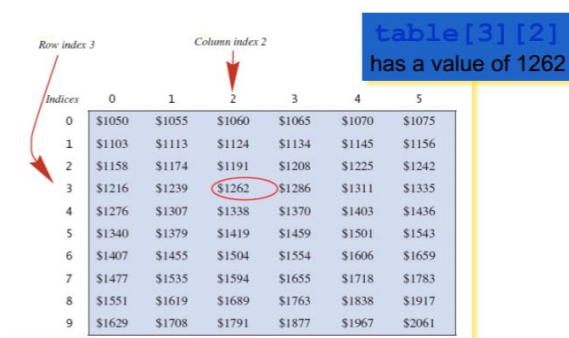
Dit is een vb van een sequential search. Het doorloopt alle elementen en kijkt of het gezochte item ertussen zit.

MULTIDIMENSIONAL ARRAYS

2-dimensies:

Rij en kolomindices voor een array met als naam

`table`



Row index 3

Column index 2

`table[3][2]`
has a value of 1262

Indices	0	1	2	3	4	5
0	\$1050	\$1055	\$1060	\$1065	\$1070	\$1075
1	\$1103	\$1113	\$1124	\$1134	\$1145	\$1156
2	\$1158	\$1174	\$1191	\$1208	\$1225	\$1242
3	\$1216	\$1239	\$1262	\$1286	\$1311	\$1335
4	\$1276	\$1307	\$1338	\$1370	\$1403	\$1436
5	\$1340	\$1379	\$1419	\$1459	\$1501	\$1543
6	\$1407	\$1455	\$1504	\$1554	\$1606	\$1659
7	\$1477	\$1535	\$1594	\$1655	\$1718	\$1783
8	\$1551	\$1619	\$1689	\$1763	\$1838	\$1917
9	\$1629	\$1708	\$1791	\$1877	\$1967	\$2061

`Array_Name[Row_index][Column_index]`

we zetten er dus een een extra paar haakjes bij. Voor meer dan 2 indices, moeten er gewoon haakjes aan worden toegevoegd.

- Elementen van de tabel kunnen benaderd worden via een geneste loop
- Voorbeeld:

```
int[][] tabel = new int[ROWS][COLUMNS];
for(int row = 0; row < ROWS; row++){
    for(int column = 0; column < COLUMNS; column++){
        tabel[row][column] = getBedrag(1000.00, row + 1, (5+ 0.5 * column));
    }
}
```

Een parameter of returntype van een methode kunnen een volledige multidimensionale array zijn. De syntax voor de methodes heading is gelijk aan die voor een ééndimensionale array parameters of return types maar je gebruikt meer haakjes:

SYNTAX: `public static Return_Type Method_Name (Base_Type[...][...] Parameter_Name)`
of

`public static Base_Type[...][...] Method_Name (Parameter_List)`

je kan andere modifiers gebruiken dan public en static

```
VB) public static int getOneElement(char[][] a, int row, int column)
    public void readArray (int[][] an Array)
    public static char[][] copy(char[][] array)
    public int[][] getArray()
```

Multidimensional arrays are arrays of arrays.

JAVA REPRESENTATION MULTI-ARRAY

- Multidimensionale arrays worden opgeslagen als meerder één dimensionale arrays
- Gegeven
`int [][] table = new int [10][6];`
- Array table komt overeen met een 1 dimensionale array met als type `int[]`
 - Array van Arrays
 - Array-gebaseerde datastructuur

H12: DYNAMIC DATA STRUCTURES AND GENERICS

ARRAY-BASED DATA STRUCTURES

THE CLASS ARRAYLIST

Bevindt zich in the package `java.util` uit de Java Class Library.

ARRAYLIST

- Beperkingen van Java arrays
 - Array lengte is niet dynamisch aanpasbaar
 - Het is mogelijk om een nieuwe grotere array te creëren en elementen te kopiëren – maar dit is niet echt handig
- Elegante oplossing door gebruik te maken van een instantie van `ArrayList`
 - Lengte is aanpasbaar at run time

Er zijn 2 nadelen bij het gebruik van `ArrayList`:

- Minder efficiënt dan gebruik van een array
- Kan alleen objecten opslaan–
- Dus geen primitieve data types

De implementatie van ArrayList gebruikt arrays. Om de capaciteit van zijn onderliggende array uit te breiden, gebruikt het dezelfde techniek als dat we gebruikten om onze array order uit te breiden. Een arraylist gebruiken ipv een array zal meer computertijd inpalmen. Je moet een keuze maken nadat je de situatie grondig geanalyseerd hebt. Volgens het tweede nadeel zullen we gebruik moeten maken van de wrapper classes bvb Integer voor int.

ARRAYLIST

- Klasse ArrayList is een implementatie van het **Abstract Data Type (ADT) lijst**
- Elementen kunnen toegevoegd worden aan
 - Het einde
 - Het begin
 - Tussen twee elementen
- Mogelijk om elementen van de lijst te bewerken, te verwijderen, op te halen en te tellen

CREËREN INSTANTIE ARRAYLIST

- Package importeren

```
import java.util.ArrayList;
```
- Creatie van instantie

```
ArrayList<String> list =  
    new ArrayList<String>(20);
```
- Deze list zal
 - **String** objecten bevatten
 - Initieel bestaan uit 20 elementen

GEBRUIK MAKEN VAN DE METHODES VAN ARRAYLIST

Objecten van ArrayList zijn hetzelfde geïndexeerd als een array → eerste index is nul.

Vbn)

- `anArray[index]="Hi mom!", → aList.set(index, "Hi mom!")`
- `String temp=anArray[index] → String temp = aList.get(index)`

We maken dus gebruik van set en get bij ArrayLists (het gaat dan ook om objecten. Maar de methode 'set' is bedoeld om een value te veranderen maar niet om het voor de eerste keer in te stellen. Om een element voor de eerste keer te setten, gebruiken we add als methode. Deze plaatst elementen in index met positie 0,1,2,3,.. en zo verder. ArrayLists moeten altijd in die volgorde gevuld zijn. Vb index 6 vullen voor index 5 gevuld is, is illigaal. Je kan add wel gebruiken om een bepaald element tussen 2 bestaande elementen te plaatsen. De bestaande elementen maken dan plaats voor het nieuwe element. Om te vinden hoeveel elementen er al aan een lijst zijn toegevoegd: `aList.size()`.

Enkele methodes:

<code>public ArrayList<Base_Type>(int initialCapacity)</code>
Creates an empty list with the specified <i>Base_Type</i> and initial capacity. The <i>Base_Type</i> must be a class type; it cannot be a primitive type such as <code>int</code> or <code>double</code> . When the list needs to increase its capacity, the capacity doubles.
<code>public ArrayList<Base_Type>()</code>
Behaves like the previous constructor, but the initial capacity is ten.
<code>public boolean add(Base_Type newElement)</code>
Adds the specified element to the end of this list and increases the list's size by 1. The capacity of the list is increased if that is required. Returns true if the addition is successful.
<code>public void add(int index, Base_Type newElement)</code>
Inserts the specified element at the specified index position of this list. Shifts elements at subsequent positions to make room for the new entry by increasing their indices by 1. Increases the list's size by 1. The capacity of the list is increased if that is required. Throws <code>IndexOutOfBoundsException</code> if <code>index < 0</code> or <code>index > size()</code> .

<code>public Base_Type get(int index)</code>
Returns the element at the position specified by <code>index</code> . Throws <code>IndexOutOfBoundsException</code> if <code>index < 0</code> or <code>index ≥ size()</code> .
<code>public Base_Type set(int index, Base_Type element)</code>
Replaces the element at the position specified by <code>index</code> with the given element. Returns the element that was replaced. Throws <code>IndexOutOfBoundsException</code> if <code>index < 0</code> or <code>index ≥ size()</code> .
<code>public Base_Type remove(int index)</code>
Removes and returns the element at the specified index. Shifts elements at subsequent positions toward position <code>index</code> by decreasing their indices by 1. Decreases the list's size by 1. Throws <code>IndexOutOfBoundsException</code> if <code>index < 0</code> or <code>index ≥ size()</code> .
<code>public boolean remove(Object element)</code>
Removes the first occurrence of <code>element</code> in this list, and shifts elements at subsequent positions toward the removed element by decreasing their indices by 1. Decreases the list's size by 1. Returns true if <code>element</code> was removed; otherwise returns false and does not alter the list.

- Object van een `ArrayList` wordt gebruikt zoals een array
 - Maar men maakt gebruik van methode en dus geen vierkante haken
- Gegeven


```
ArrayList<String> aList =
    new ArrayList<String>(20);
```

 - Een waarde toekennen


```
aList.add(index, "Hi Mom");
aList.set(index, "Yo Dad");
```

FOR-EACH LOOP & ARRAYLIST

- Case


```
for(String element: toDoList)
    System.out.println(element);
```
- Snelle manier om door een lijst te lopen
- Algemeen:

```
for(Base_Type element: ArrayList<Base_Type>)
    Statements
```

KLASSEN MET PARAMETERS

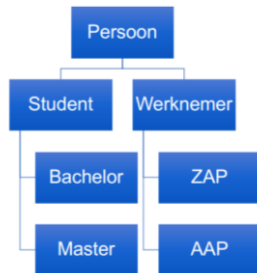
- Klasse `ArrayList` is een klasse met een type als parameter
- Later meer
- Momenteel enkel kunnen gebruiken bij initialiseren van een `ArrayList`

H8: INHERITANCE, POLYMORPHISM AND INTERFACES

INHERITANCE BASICS

Inheritance of overerving: geeft je de mogelijkheid om een heel algemene klas te definiëren en daarna deze klasse op te splitsen in meer gespecialiseerde klassen die meer details hebben dan de general class. De nieuwe klasse erft alle eigenschappen van de overkoepelende klasse over.

GESPECIALISEERDE KLASSEN



Student is a derived class of een subklasse van de klasse persoon. De derived class extends de bestaande klasse of superklasse.

Heading: Public class Student extends Person

Een object van de klasse student krijgt alle methodes van de klasse person bijgevoegd bij alle methoden die special zijn van de klasse student. We noemen deze relatie een is-a relationship. Overerving is enkel mogelijk als er een is-a relationship bestaat tussen de superklasse en de subklasse.

OVERRIDING METHOD DEFINITION

Zowel de klasse student als de klasse persoon hebben een methode writeOutput zonder parameters. Java heeft een regel om dit probleem te vermijden: als een derived class een methode definieert met dezelfde naam, dezelfde aantal parameters van hetzelfde type en dezelfde return type als de methode in de base class, overtreft de methode in de derived class deze van de baseclass (override) Je kan zeggen dat als de ene methode de andere override, moeten ze dezelfde signature en return type hebben.

OVERRIDING VERSUS OVERLOADING

Als de methode in de derived class dezelfde naam en hetzelfde returntype hebben maar een verschillend aantal parameters van een verschillend type van de methode van de base class, dan is de naam van de methode overloaded.

Overloading placed an additional load on a method name by using it for another method whereas overriding replaces a method's definition.

THE 'FINAL' MODIFIER

Als je niet wil dat een specifieke methode kan worden overriden door een nieuwe definitie van een derived class, kan je 'final' toevoegen.

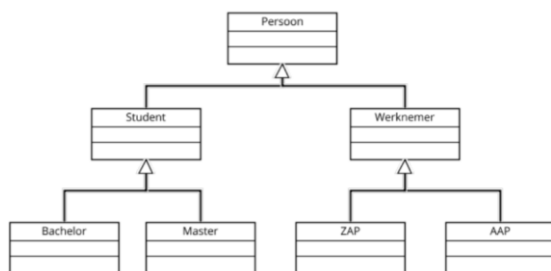
```
Public final void specialMethod();
```

Een hele klasse kan final worden gedefinieerd, in dat geval kunnen er geen derived classes voortkomen uit deze base class die dan eigenlijk geen base class is maar een gewone klasse.

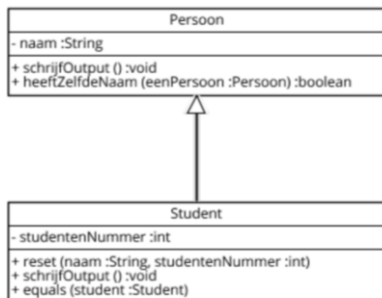
PRIVATE INSTANCE VARIABLES AND PRIVATE METHODS OF A BASE CLASS

- Beschouw private instantievariabelen in de basisklasse
 - Worden niet overgeërfd door subklasse
 - Kunnen alleen gemanipuleerd worden via publieke accessor en modifier methodes
- Gelijkaardig, private methodes in een superklassen worden niet overgeërfd door subklasse

UML INHERITANCE DIAGRAMS



De pijltjes tonen de richting aan van een derived class naar een base class. Deze pijlen tonen dus de is-a relationship. Ze helpen ook om een locating method thuis te brengen (van een subklasse moet je gewoon de pijltjes volgen om alle mogelijke methoden te vinden).



Deze laatste afbeelding toont meer details.

PROGRAMMING WITH INHERITANCE

CONSTRUCTORS IN DERIVED CLASSES

Bevat enkel zijn eigen constructor en gene van de base class. De base class heeft ook zijn eigen constructor. Constructors van een derived class kunnen wel constructors van een base klasse invoken hiervoor gebruiken ze het woord super als methode. Je gebruikt niet de naam van de constructor zelf! Person(initialName) is ILLIGAL!

Om super te gebruiken moet dit altijd de eerste actie zijn in een constructordefinitie.

```
Public Student ()
{
    super();
}
```

```

        studentNumber=0;
    }

```

(Als we super() hier zouden weglaten blijft de uitkomst gelijk)

- Een gespecialiseerde klasse erft de constructor van zijn basis klasse niet over
 - Constructor in de subklasse moet de constructor van de basisklasse oproepen
- Gebruik hiervoor keyword **super**
- Moet eerste statement zijn in constructor

```

public Student(int studentenNummer, String naam) {
    super(naam);
    this.studentenNummer = studentenNummer;
}

```

THE 'THIS' METHOD-AGAIN

Wordt gebruikt om een andere constructor op te roepen in dezelfde klasse. Je kan this en super op dezelfde manier gebruiken.

Ook hier moet alle uitspraken met this in het begin van de constructor gebeuren. Een constructor kan dus niet zowel super als this gebruiken. Als je ze toch beide wil gebruiken, kan je this gebruiken om een constructor te callen dat super als eerste actie heeft.

- Het is ook mogelijk om keyword **this** te gebruiken
 - Kan men gebruiken om eender welke constructor in klasse op te roepen

```

public Person()
{
    this("No name yet");
}

```

- Wanneer men this gebruikt in een constructor, roept men constructor op in dezelfde klasse.
 - **super** roept constructor op van de basisklasse

CALLING AN OVERRIDDEN METHOD

Een methode van een derived class dat overrides een methode van een base class kan super gebruiken om de overridden method te callen (maar toch op een beetje een verschillende manier).

- **Super** kan men ook gebruiken voor oproepen van methode in gespecialiseerde methode

```

public void schrijfOutput(){
    super.schrijfOutput();
    System.out.println("Studentnummer: " + this.studentenNummer);
}

```

- Roept methode van de basisklasse op met dezelfde naam

TYPE COMPATIBILITY (COMPTABILITEIT)

Als we de klasse undergraduate bekijken zien we dat dit een derived class is van de klasse student. Dus elk object van de klasse undergraduate kan zowel methoden gebruiken van de klasse student als de klasse

undergraduate. Hier is geen sprake van type casting, het is gewoon een feit dat elke bachelor een deel uitmaakt van de studenten

- De klasse hiërarchie
 - Elke **Bachelor** is ook een **Student**
 - Elke **Student** is een **Persoon**
- An object van een gespecialiseerde klasse kan ook dienst doen als object van de basisklasse
 - Dit is geen type casting
- Een variabele van het superklasse type kan verwijzen naar een object van een subklasse

Zo zien we ook dat een object meer dan 1 type kan hebben. Undergraduate is een object van de klasse student en dus een object van het type student. De klasse student is een object van de klasse persoon en dus een object van het type persoon → undergraduate is zowel van het type student als het type persoon.

Dus het is perfect mogelijk dat als bvb someUndergrad een object is van het type undergraduate en in een bepaalde methode hebben ze een parameter nodig van het type persoon, dat deze someUndergrad geldig is in deze positie.

- Overerving implementeert een "is-a" relatie
 - Een **Student** is een **Person**
- Een ander soort relatie is de "has-a" relatie
 - Een klasse kan een object als instantievariabele bevatten dat van een ander klasse type is
 - Bvb Person heeft een geboortedatum instantievariabele van het type **Date**

EEN STUDENT IS EEN PERSOON, MAAR EEN PERSOON IS NIET NOODZAKELIJK EEN STUDENT.

Has-a relationship: vb we hebben een klasse 'date' dat records a date. We kunnen dit toepassen door een date of enrolment toe te kennen aan de klasse student. We zeggen dan: een student "has a" date.

THE CLASS OBJECT

Java heeft een "Eve"-klasse dat is een voorvader van alle andere klassen. In java is elke klasse afkomstig van de klasse object. Zelfs klassen die je zelf definieert zijn klassen afkomstig van de klasse object. Deze klasse heeft natuurlijk ook methodes die door elke andere klasse kunnen worden gebruikt: equals and toString. Deze methodes worden dus veel overriden. De inherited method toString heeft geen argumenten. Deze methode zet alles om van dit object in een string. Dit is eigenlijk wel meestal useless want het zal nooit een correcte string representatie geven van de data.

OBJECT KLASSE

- Java heeft een klasse dat de ultieme ouder is van elke klasse
 - Klasse **Object**
- Het is mogelijk om een methode te definiëren met een parameter van het type **Object**
 - Actuele parameter kan object zijn van eender welk type
- Vb: methode **println(Object theObject)**

OBJECT KLASSE

- Object klasse heeft een aantal methodes die door elke klasse worden overgeërfd
- Voorbeelden
 - Methode **equals**
 - Methode **toString**
- Methode **toString** wordt gebruikt wanneer methode **println(theObject)** wordt opgeroepen
 - Beter je eigen **toString** definiëren

A BETTER EQUALS METHOD

- Klasse specialiseert best de methode equals van **Object** Klasse
- Zie code van Student klasse

```
public boolean equals(Object object){
    boolean isEqual = false;
    if((object != null) && (object instanceof Student)){
        Student eenStudent = (Student) object;
        isEqual = this.heeftZelfdeNaam(eenStudent) &&
            (this.studentenNummer == eenStudent.studentenNummer);
    }
    return isEqual;
}
```



Meestal wordt de Object's definition van equals overridden en wordt er een eigen versie van gemaakt zoals hierboven is gebeurd. Zo kunnen equals ook verschillende parametertypes hebben. In dat geval is equals enkel overloaded.

Als we een methode hebben met een parameter van het type Object en een parameter van het type Student. De body van deze methode bevat: studentParam.equals(objectParam) Als je zowel een argument van het type Student voor de parameter objectParam als een argument voor van het type student voor studentParam, zal Java de definitie gebruiken van equals van de klasse Object en niet van de klasse student. Hierdoor zal equals soms het verkeerde antwoord geven. Om dit probleem op te lossen moeten we het type van de parameter veranderen voor de equal methode in de Student class van Student naar Object. Hierbij moeten we dus typecasten.

Het nadeel hieraan is dat eerder welk object kan worden gebruikt. Dus wat als er een argument wordt gebruikt die niet van het type Student is. Een run-time error zal plaatsvinden als de type cast naar Student plaatsvindt. We doen het dus werken voor alle objecten en stellen de code zo in dat wanneer het object niet van type Student is, returnen we false.

We kunnen gebruik maken van instanceof om te checken of een object van het type student is:

Object instanceof Class_Name

POLYMORPHISM

= geeft je de mogelijkheid om veranderingen in de methode definitie te maken voor de subclasses en zorgen ervoor dat deze veranderingen apply voor methodes in de base class.

- Overerving laat toe om een basisklasse te definiëren en op basis hiervan gespecialiseerde klassen te definiëren
- Polymorfisme maakt het mogelijk om wijzigingen aan te brengen aan de methode definities van de gespecialiseerde klassen en deze wijzigingen zullen het gedrag van methodes in de basisklasse beïnvloeden

DYNAMIC BINDING AND INHERITANCE

Vb) we willen een comité opzetten die bestaat uit 4 personen die ofwel student of werknemer zijn. Als we gebruik maken van een Array, maken we een Array van het type Person zodat het objecten van elke derived class kunnen gebruiken.

- Beschouw een Array van **Persoon**

```
Persoon[] personen= new Persoon[4];
```

- Omdat **Student** en **Bachelor** van het type **Persoon** zijn, kunnen we objecten hiervan toewijzen aan een **Person** variabele

```
personen[0] = new Student("DeBanque, Robin", 8812);  
personen[1] = new Bachelor("Cotty, Manny", 8812, 1);
```

Wanneer we personen[0].writeOutput invoken, zien we dat dit van het type persoon is, dus dat het ook de methode writeOutput van de klasse Persoon is die wordt toegepast. Maar dit is niet wat gebeurt!! Java ziet dat het object dat hierin gestockeerd zit van het type student is, zal het de methode gekoppeld aan deze klasse uitvoeren. =**dynamic binding or late binding**

- Wanneer een gespecialiseerde methode wordt opgeroepen
 - Wordt de methode gebruikt die gedefinieerd is in de klasse dat gebruikt werd voor het instantiëren van het object met **new**
 - Wordt niet bepaald door type van de variabele
- Variabele van een supertype kan verwijzing bevatten naar object van een van de subklasse

INTERFACES AND ABSTRACT CLASSES

CLASS INTERFACES

- Beschouw een set van acties die horen bij een huisdier
 - Een naam geven
 - Eten
 - Reageren op bevelen
- We kunnen methode headers specificeren voor deze acties
- Deze methode headers vormen een klasse interface
- Beschouw nu meerdere klassen die deze interface implementeren
 - Ze beschikken allemaal over het hetzelfde gedrag
 - Aard van het gedrag zal verschillend zijn
- Elke klasse implementeert het gedrag / de methodes verschillend.

JAVA INTERFACES

= een component van een programma dat de headings voor aantal publieke methodes omvat. Sommige interfaces beschrijven alle public methods in een klasse en sommige beschrijven maar enkele. Een interface kan ook enkel publiek gedefinieerde constanten definiëren. De Java Class Library bezit interfaces die al geschreven zijn speciaal voor jou om te gebruiken maar je kan ook je eigen interface definiëren.

Public interface Interface_Name

Deze interface kan verschillende public methods bevatten die van elkaar gescheiden worden dmv ‘;’. De naam van een interface begint met een hoofdletter. Een interface wordt opgeslaan in zijn eigen file eindigend op .java. Een interface definieert geen constructor voor een klasse. Methodes in een interface moeten public zijn.

- Een programma component dat enkel de headers bevat van een aantal publieke methodes
 - Bevat ook commentaar die de methode beschrijft
- Interface kan ook publieke constanten definiëren.

IMPLEMENTATIE VAN EEN INTERFACE

Als je een klasse schrijft met een methode declareert in een interface, dan zeggen we dat de klasse the interface implementeert.

Een klasse dat een interface implementeert moet een body definiëren voor elke methode dat de interface specificeert. Een interface moet niet elke methode in een klasse definiëren.

Om een interface te implementeren, a class must do 2 things:

- 1) Schrijf de zin: implements Interface_Name vb) implements MyInterface, YourInterface
- 2) Define each method declared in the interfaces.
 - Implementatie
 - Voeg dit toe aan klasse

```
implements Interface_name
```
 - Definieer elke methode van de interface
 - Zie
 - `class Rechthoek implements Meetbaar`
 - `class Cirkel implements Meetbaar`

Verschillende klassen kunnen dus dezelfde interface implementeren.

AN INTERFACE AS A TYPE

Een interface is een reference type. Dus een methode met een parameter kan gebruik maken van een parameter van type Meetbaar.

Vb) public static void display(Measurable figure)

- ➔ Deze methode kan altijd invoked worden door een object van eender welke klasse dat de interface implementeert.
- ➔ Zie boek p 665

EXTENDING AN INTERFACE

Eens je een interface hebt, kan je een andere interface definiëren dat er op bouwt of ‘extends’ de eerste door een soort overerving te gebruiken. Je creëert dus een interface die bestaat uit de oude interface en sommige nieuwe methoden. Vb)

```
Public interface Nameable
{
}
```

```
Public interface Callable extends Nameable
{
}
```

ZIE CASE STUDIES!!!!

ABSTRACT CLASSES

Klasse **ShapeBasics** is ontworpen om een basisklasse te zijn van een andere klassen

- Methode **tekenHier** zal gedefinieerd worden voor elke subklasse
- Deze methode moet *abstract* gedefinieerd worden – een methode zonder body

Dit maakt de klasse abstract

Men kan geen object aanmaken van een abstracte klasse

Het is niet de bedoeling dat we objecten van de klasse ShapeBasics maken, het is enkel een overkoepelende klasse voor de verschillende klassen die genaamd zijn naar vormen zoals rectangle, circle,...

Dit doen we als volgt: ShapeBasics shapeVariable = new ShapeBasics ();

We geven de klasse ShapeBasics de methode DrawHere zodat dit kan gebruikt worden voor alle subklassen. We zijn echter wel van plan om deze te overriden aangezien niet alle vormen op dezelfde manier kunnen worden getekend. Het is dus eigenlijk gewoon een placeholder. We kunnen de methode dan als abstract definiëren:

```
Public abstract void drawHere ();
```

Een abstracte methode moet Overriden worden door een nonabstract derived class en moet gedefinieerd worden in deze klasse. Java zegt dat als een klasse minstens 1 abstracte methode heeft, dan moet de klasse ook als abstract worden gedefinieerd. Dit doe je door ook the keyword abstract in je class definition te gebruiken:

Public abstract class ShapeBase → dit noemen we dan een abstracte klasse. Je kan dan geen objecten van deze klasse maken. Het is niet noodzakelijk dat alle methodes in de abstracte klasse abstract zijn.

We gebruiken abstracte klassen omdat deze het denken vergemakkelijken.

GRAPHICS SUPPLEMENT

THE CLASS JAPPLET

```
Public class LabelDemo extends JApplet
```

Deze klasse heft methodes genaamd init en paint. Dus applets met deze methodes overriden ze dus. Dit gebeurt meestal automatisch voor jou bij het aanmaken van aan JApplet.

GUI'S & OVERVERVING; JAPPLET

```
//
public class HappyFace extends JApplet {

    public void paint(Graphics canvas){
        super.paint(canvas);
        canvas.drawOval(100, 50, 200, 200);
        canvas.fillOval(155,100,10,20);
        canvas.fillOval(230, 100, 10, 20);
        canvas.drawArc(150, 160, 100, 50, 180, 180);
    }

    public void init() {
        // TODO start asynchronous download of heavy resources
    }

    // TODO overwrite start(), stop() and destroy() methods
}
```

GUI'S & OVERVERVING: JFRAME

```
public class HappyFaceJFrame extends JFrame {

    public void paint(Graphics canvas){
        super.paint(canvas);
        canvas.drawOval(X_GEZICHT, Y_GEZICHT, DIAMETER_GEZICHT, DIAMETER_GEZICHT);
        canvas.fillOval(X_RECHTEROOG,Y_RECHTEROOG,BREEDTE_OOG,LENGTE_OOG);
        canvas.fillOval(X_LINKEROOG,Y_LINKEROOG,BREEDTE_OOG,LENGTE_OOG);
        canvas.drawArc(X_MOND, Y_MOND, BREEDTE_MOND, LENGTE_MOND, START_HOEK_MOND,
    }

    public HappyFaceJFrame(){
        this.setSize(600,400);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}
```

WINDOW EVENTS AND WINDOW LISTENERS

In H2 zagen we methoden om een window te sluiten. We kunnen ook ons eigen programma schrijven voor als het programma gesloten is. De close-window button fires an event zoals veel buttons van JButton. De window event wordt onder de hoede van de window Listener genomen. De klasse WindowAdapter is een window listener en dus elke klas afkomstig van deze klasse is een window listener. Een window listener moet geregistreerd worden voor een JFrame GUI. Dit doen we door addWindowListener te callen. (dit is analoog aan addActionListener voor JButton)

Zie p 690,691 voor vb

H9: EXCEPTION HANDLING

BASIC EXCEPTION HANDLING

We maken een onderscheid tussen een normal case en een exceptional case. Een exception is een object dat een signaal geeft wanneer een ongewone gebeurtenis plaatsvindt tijdens de uitvoering van een programma. Het proces van dit object aanmaken noemen we "throwing an exception". De code dat detecteert en omgaat met de exception handelt de exception.

EXCEPTIONS IN JAVA

- **Try** blok
 - Bevat code waar mogelijks iets fout kan gaan
 - Indien iets fout gaat, wordt een exception gegooid.
- **catch** blok
 - Indien een exception is gegooid, zal uitvoering van **catch** blok beginnen
 - Analooq aan oproepen methode met parameter
 - Parameter is het gegooide exception object

Vb) een exception is thrown wanneer men wil delen door 0 (herinner donut en melk vb p 707).

Als de throw statement is executed, creëert het een nieuw object van de klasse Exception:

New Exception (“exception: No milk!”) → “Exception: No milk” is een argument van de constructor. De exception object dat hier gecreëerd is, wordt in een instance variable gestoken zodat ze later kan gebruikt worden. Als een exception is thrown, de code die daarop volgt wordt niet meer invoked. We gaan direct door naar het catch block. Executing het catch block noemt men catching the exception. Een catchblock lijkt op een method definition but it isn’t.

catch(Exception e) → e lijkt op een parameter en gedraagt zich ook veelal als een parameter. We noemen e een catch-block-parameter. We kunnen e zien als de naam voor the exception object that was thrown. Elk exception object heeft een methode getMessage en deze ontvangt de string die het exception object bij de constructor heeft meegekregen in de trybox. Dus hier “Exception: No Milk”.

De naam van de klasse die na the catch-block parameter komt, hoort meer te specificeren wat voor soort exception het is.

Verschillende catch-blocks kunnen volgen na een try-block maar alleen de catchblock dat correspondeert met het specifieke type van exception wordt uitgevoerd.

Wanneer de catchblock is uitgevoerd, keert het niet terug naar de try-block. Meestal wordt het programma hier beëindigd met system.exit(0).

PREDEFINED EXCEPTION CLASSES

- Java klasse bibliotheek bevat voorgedefinieerde exception klassen
 - Worden opgevangen in **try** blok
 - Gevolgd door een **catch** blok voor dit type exception
- Voorbeelden Exception klassen
 - **BadStringOperationException**
 - **ClassNotFoundException**
 - **IOException**
 - **NoSuchMethodException**

Voorbeeld code

```
SampleClass object = new SampleClass();
try
{
    <Possibly some code>
    object.doStuff(); //may throw IOException
    <Possibly some more code>
}
catch(IOException e)
{
    <Code to deal with the exception, probably including the following:>
    System.out.println(e.getMessage());
}
```

DEFINING YOUR OWN EXCEPTION CLASSES

Je kan zelf ook je exception klassen definiëren, maar ze moeten derived classes zijn van al eerder gedefinieerde exception classes. Wanneer we een exception class definiëren, zijn de constructors de belangrijkste en meestal ook enigste methodes naast deze dat ze meekrijgen van the base class zoals getMessage. Vb) de exception class DivideByZeroException. In de default constructor word dan super() gebruikt vb) super("Dividing by Zero!").

- Moeten afgeleid zijn van een voorgedefinieerde exception klasse
 - Zie `class DelenDoorNulException extends Exception`
- Methode `getMessage` gedefinieerd in exception klasse
 - Retourneert string die werd meegegeven als argument aan constructor van de exception
 - Indien geen argument werd meegegeven, wordt de standaard message geretourneerd
- Het type van een object is de naam van de exception klasse
- Richtlijnen
 - Definieer ten minste 2 constructors:
 - Default, geen parameters
 - Met `String` parameter
 - Start constructor definitie met oproep constructor van de basis klasse door gebruik te maken van `super`
 - Overschrijf de methode `getMessage` niet

MORE ABOUT EXCEPTION CLASSES

Techniques for using exceptions

DECLARING EXCEPTIONS (PASSING THE BUCK)

Soms is het logisch om de handling van een exception uit te stellen. Zo kan je een methode hebben die een exception throws maar je wil de exception nog niet catchen in die methode. Als een methode een exception niet caught, moet het de programmeurs wel waarschuwen dat een del van de methode een exception kan throwen. Deze 'warning' noemen we een throws clause.

Vb) `public void sampleMethod() throws DivideByZeroException`

– Syntax throws clause

```
public Type Method_Name(Parameter_List) throws List_Of_Exceptions  
Body_Of_Method
```

- Opgelet verschil tussen
 - Keyword **throw** voor het gooien van een exception
 - Keyword **throws** gebruikt in methode header voor declareren van exception
- Indien een methode
 - een exception gooit en
 - Exception wordt niet opgevangen binnen de methode zal de methode onmiddellijk stoppen na het gooien van de exception
- Een throws clause in de overgeërfde methode
 - Kan minder exceptions declareren dan gedeclareerd door de super methode

Deze techniek is een sort van Passing the buck vb) MethodA heft een throws clause; als method een invocation van MethodA bevat, dan moet methodB met de exception afrekenen. MethodA geeft dus de verantwoordelijkheid door aan eender welke methode dat MethodA invoked. Natuurlijk kan ook MethodB de verantwoordelijkheid doorgeven door dezelfde throws clause te gebruiken in zijn heading. Maar elke exception dat geworpen wordt, moet uiteindelijk gecatched worden door een catchblock door een methode dat de verantwoordelijkheid niet doorgeeft. Wanneer er zich meerdere exceptiontypes voordoen, worden deze allemaal vermeld en gescheiden van elkaar dmv komma's.

Vb) `public int myMethod() throws IOException, DivideByZeroException`

A throws clause in an overriding method can declare fewer, but not more, exceptions than the overridden method declared.

KINDS OF EXCEPTIONS

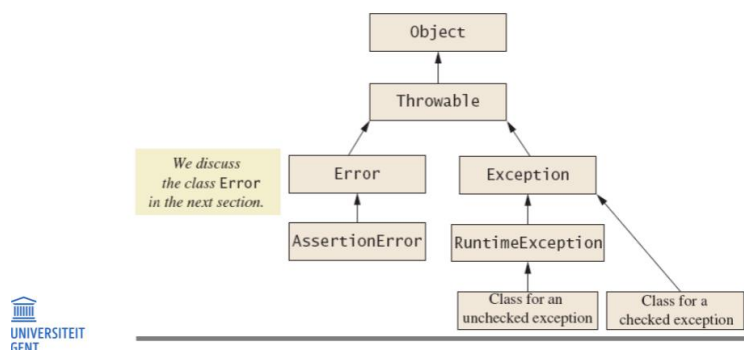
Dus of een exception moet gecatched worden door een catchblock of gedeclareerd zijn in een throwsclause in een method's heading. But there is an exception to this rule (uizondering op een uitzondering 😊)

Alle Java exceptions zijn in een hokje gestopt met de naar checked of unchecked.

- Checked exception: moeten gecatched zijn of gedeclareerd zijn in a throws clause. Alle exceptions van de Java Class Library zijn checked.
- Een unchecked of runtime exception: tonen meestal aan dat er iets fout is met je code en dat je er iets moet aandoen. Je moet dan je code aanpassen ipv een catchblock in te voeren. Een uncaught run-time exception terminates program execution.

Hoe weet je nu wanneer een exception is checked of unchecked? → je kan de documentatie van de Java Class Library raadplegen. Er bestaat ook een soort hiërarchie in:

– Hierarchy voorgedefinieerde exception klassen



Classes of unchecked exceptions are derived from the class RuntimeException.

ERRORS

Een Error is een object van de klasse Error. Deze klasse is derived van Throwable en dit is ook de base Class van Exception. Error en al de klassen die hieruit voortkomen zijn dus geen deel van de exception class. Ze zijn wel gelijkaardig aan unchecked exceptions. Errors zijn min of meer niet onder controle te houden. Vb) een OutOfMemoryError komt voor als je programma geen geheugen meer heeft. Dan zal je je programma moeten herbekijken om meer memory over te houden of door meer memory te kopen voor jouw computer.

Wanneer een programma eindigt met een error message, gebeurt eigenlijk een AssertionError. De klasse AssertionError komt voort uit de klasse Error.

MULTIPLE THROWS AND CATCHES

- Een try blok kan meerdere exceptions gooien van verschillende types
- Elk catch blok kan maar een exception opvangen van één bepaald type
 - Volgorde catch blok is belangrijk
- Exceptions kan men gebruiken voor afhandelen van ongeldige gebruikersinput
- Gebruik van **throw** statement alleen in uitzonderlijke omstandigheden
- Geneste try-catch zijn zelden zinvol

De catchblocks worden onderzocht volgens appearance. De eerste block dat het type exception matched, wordt uitgevoerd.

THE FINALLY BLOCK

Na een rij van catchblocks, is het mogelijk om een finally block toe te voegen. De code hierin wordt sowieso uitgevoerd of een exception nu thrown is of niet. Dit geeft je de mogelijkheid om losse eindjes te verdoezelen.

Volgende mogelijke uitkomsten kunnen voorkomen wanneer de code in een catch-finally block is run:

- Try block wordt helemaal uitgevoerd (geen exception is thrown). De finally block komt dus na de try
- An exception is thrown in een try block en wordt in een catchblock opgevangen. de finally block wordt dan na de catchblock uitgevoerd.
- Een exception is thrown maar er is geen matching catchblock. De finally block wordt dan uitgevoerd ipv een matching catchblock.

GRAPHICS SUPPLEMENT

An uncaught exception in een non-GUI Java application eindigt met een error message. In een JFrame GUI of een applet eindigt het programma echter niet. Het is dus nog belangrijker om hier ervoor te zorgen dat alle checked exceptions dat geworpen zijn, opgevangen worden.

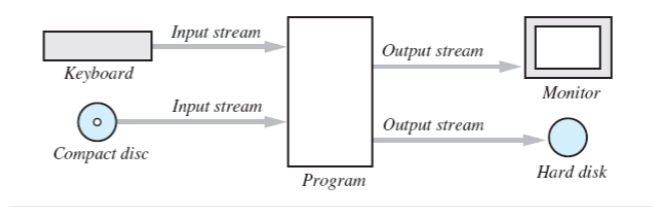
Zie p756 en 757 voor vb

H10 STREAMS, FILE I/O, AND NETWORKING

AN OVERVIEW OF STREAMS AND FILE I/O

THE CONCEPT OF A STREAM

Een stream is een flow van data. Deze data kunnen zowel characters, cijfers of bytes zijn. Als data in je programma flows spreken we over een inputstream, als het uit je programma stroomt spreken we over een outputstream. Vb) keyboard is inputstream. In Java zijn streams geïmplementeerd als objecten van speciale stream klassen. Objecten van de klasse Scanner zijn inputstreams, terwijl object System.out een output stream is.



WHY USE FILES FOR I/O?

Door files kunnen we data permanent bewaren. De inhoud van de file blijft bewaard tot er weer iemand iets aan verandert. Een input file kan door verschillende programma's telkens opnieuw gebruikt worden.

TEKST FILES AND BINARY FILES

Alle data in eender welke file is opgeslagen als binary digits. Maar in de meeste files zien we de inhoud niet als een reeks binaire code maar als karakters die voor ons makkelijk te begrijpen zijn.

Files die gezien worden als een reeks karakters noemen we tekst files. Alle andere noemen we binary files. Java programma's zijn opgeslagen als text files, terwijl muziek en afbeeldingen opgeslaan zijn als binary files. De meeste code in binary files zijn gestandaardiseerd zodat ze door verschillende apparaten kunnen gelezen worden. Java programma's kunnen zowel tekst als binary files lezen. Over welke file het gaat heeft enkel inspraak op welke klasse we gebruiken om input en uitput te doen ontstaan. Het grote voordeel bij text files is dat je het kan maken, het bekijken en weer bewerken met een tekst editor. Terwijl binary files gelezen en geschreven moeten worden door een speciaal programma. Soms is het zelfs zo dat enkel dezelfde soort

computer die een binary file gemaakt heeft, deze ook weer kan gebruiken.

- Alle data in bestanden worden opgeslagen als binaire digits
- Bestanden met sequentie van karakters, noemt men txt files:
 - Java programma source code
 - Kan men openen en aanpassen met een txt editor
- Alle andere bestanden zijn binaire bestanden
 - Movie, muziek
 - Vereist gespecialiseerd programma

TEXT FILE I/O

CREATING A TEKST FILE

De klasse `PrintWriter` wordt gebruikt om een tekst file te creëren en te schrijven. Dit bevindt zich in de package `java.io`. we moeten ons programma dan wel starten met een import statement. Voor we een tekst file kunnen schrijven, moeten we eerst connecteren met een output stream → we openen de file. Om dit te doen hebben we de naam van een file nodig als string. We moeten ook een variabele declareren dat we zullen gebruiken om te verwijzen naar de stream die geassocieerd wordt met de actual file. Deze variabele wordt de stream variabele genoemd met data type `PrintWriter`. We openen een textfile door een `PrintWriter`'s constructor te invoken en de naam van de file te gebruiken als argument. Deze actie kan een exception throwen → try block.

```
String fileName="out.txt";
PrintWriter outputStream= null;

Try
{
    outputStream= new PrintWriter (fileName);
}

Catch(FileNotFoundException e)
{
    System.out.println("Error opening the file "+ fileName);
    System.exit(0);
}
```

De klasse `FileNotFoundException` moet ook geïmporteerd worden uit de package `java.io`.

Wanneer je een file aan een output Stream koppelt op die manier, zal je altijd starten met een lege file. Als de file `out.txt` al bestaat, zal de oude inhoud verloren gaan. Doordat de file nog niet bestaat, betekent de exception niet persé dat de file niet gevonden is maar hier dan eerder dat de file niet kan aangemaakt worden omdat er al een file bestaat met dezelfde naam in een bepaalde folder.

`PrintWriter` heeft een methode `print` die gebruikt wordt op dezelfde manier als `System.out.print` met het enige verschil dat de output naar een txt file gaat. Nu de file open is, refereren we enkel nog naar file via de stream variabele.

```
outputStream.println("this is line1");
outputStream.println("this is line2");
```

deze uitdrukkingen worden allemaal verzamelt en pas later in één groot packet door PrintWriter naar de file gestuurd. Het wordt opgeslaan op een bepaald geheugen dat we een buffer noemen. Als de buffer te vol is, schrijven we de inhoud over naar de file. We sluiten de stream die gekoppeld is aan de file door:

```
outputStream.close();
```

Als je je stream niet sluit, zal Java dit voor jou doen, maar dit gebeurt dan niet altijd op de juiste manier. Het is makkelijker om de methode close te gebruiken (van PrintWriter). De calls naar println en close moeten gebeuren in de try block.

APPENDING TO A TEXTFILE

We moeten dus altijd beginnen met een lege file. Maar als de naam van de file al bestaat, zal de oude inhoud verloren gaan. Soms willen we eigenlijk gewoon de oude inhoud bewerken of verder uitwerken. Dit doen we via:

```
outputStream=new PrintWriter (new FileOutputStream(fileName, true));
```

Omdat PrintWriter geen constructor heeft voor deze opgave, maken we gebruik van de klasse FileOutputStream. Deze moeten we importeren van de package java.io. de 'true' bevestigt dat we data willen toevoegen aan de al bestaande file.

READING FROM A TEKST FILE

- Statement voor het openen van het bestand

```
Scanner stream_naam =  
    new Scanner(new File([bestandsnaam]))
```

- Boolean statement voor het lezen van tekst en beëindigen van de "lees" loop

```
while(inputStream.hasNextLine()){  
    String lijn = inputStream.nextLine();  
    System.out.println(lijn);  
}
```

Hoewel Scanner een constructor heeft met String als argument, toch zou dit niet gelden als de naam van de file. Scanner heeft wel een constructor die een klasse als argument neemt → File. File heeft een constructor waar we een file name als argument kunnen nemen.

- Bijkomende methodes van de klasse `Scanner`

- `Scanner_object.hasNext()`
- `Scanner_object.hasNextDouble()`
- `Scanner_object.hasNextInt()`
- `Scanner_object.hasNextLine()`

TECHNIQUES FOR ANY FILE

Techniques that we can use with both txt files and binary files.

THE CLASS FILE

Deze klasse File geeft een manier om file names weer te geven in een general way. Een String mag dan wel een file name zijn, java ziet dit niet als een file name. Natuurlijk kunnen we altijd een String gebruiken als file name,

want dit object weet dan dat het eigenlijk de file name is. Sommige Stream klassen kunnen enkel een File object als argument nemen.

- De klasse `File`
 - Gebruik maken van padnamen
 - Methodes van de `File` Klasse
 - Een methode definiëren voor het openen van een Stream
 - Een csv-bestand lezen
-
- Klasse maakt het mogelijk om bestandsnamen voor te stellen op een algemene manier
 - Een `File` object is een systeemafhankelijke abstractie van de bestandsnaam
 - Het object
`new File ("treasure.txt")`
is geen eenvoudige string
 - Het is een object dat weet dat het moet overeenkomen met de naam van een bestand

USING PATH NAMES

Zoals we hiervoor weergegeven hebben, gaan we er telkens vanuit dat de file altijd in dezelfde directory folder zit als waarin het programma zich afspeelt. Maar wat als dit niet het geval is? Dan maken we gebruik van path names ipv enkel de naam van de file. Een **volledige path** name geeft het volledige afgelegde pad startend van the root directory. Een **relative path name** geeft het pad weer vanaf de directory die het programma bevat.

- UNIX path: /user/smith/homework1/data.txt
- Windows: gebruikt \ ipv / vb) C:\homework\hw1\data.txt meestal wordt hierbij \\ gebruikt ipv \ want anders zou het geïnterpreteerd kunnen worden als een backslash met een ander karakter vb) \h en dan is het een escape karakter. Om deze escape karakters te vermijden kan je ook altijd UNIX gebruiken. Meestal werken ze wel beiden op een computer.

METHODS OF THE CLASS FILE

Methodes kunnen gebruikt worden om properties van een file te onderzoeken.

- Je wil testen of een file al bestaat: `fileObject.exists()` (boolean dus meestal in een if else systeem)
 - Kijken of the operating system de file die je wil openen kan lezen: `fileObject.canRead()` (ook boolean)
 - De methode `canWrite` is ongeveer gelijk aan `canRead` maar kijkt dan of the operating system je zal toestaan om verder te schrijven aan de file.
- `File` klasse heeft methodes die men kan gebruiken voor het ophalen informatie over het pad en het bestand
 - `public boolean canRead()`
 - `public boolean canWrite()`
 - `public boolean delete()`
 - `public boolean exists()`
 - `public String getName()`
 - `public String getPath()`
 - `public long length()`

DEFINING METHOD TO OPEN A STREAM

The following methodes creëren een output stream geconecteerd aan de naam van de textfile maar returns de stream. Zie p 795

- Methode heeft een `String` als parameter
 - De bestandsnaam
- Methode zal het stream object retourneren
- Methode zal exception gooien indien
 - Het bestand niet gevonden is
 - Bij andere I/O problemen
- Moet worden opgeroepen binnen een `try` blok en heeft aangepast `catch` blok

BASIC BINARY- FILE I/O

We gebruiken de stream klassen `ObjectInputStream` en `ObjectOutputStream` om een binary file te schrijven en lezen. Elke klasse heeft methodes om data one byte at a time te lezen. Deze streams converteren ook nummers en andere karakters naar bytes. `ObjectOutputStream` wordt gebruikt om de binary file te schrijven en `ObjectInputStream` wordt gebruikt om van de file te lezen.

– code

```
public static PrintWriter openOutputTextFile(String fileName)
    throws FileNotFoundException, IOException
{
    PrintWriter toFile = new PrintWriter(fileName);
    return toFile;
}
```

– Oproep

```
PrintWriter outputStream = null;
try
{
    outputStream = openOutputTextFile("data.txt");
}
< appropriate catch block(s) >
```



CREATING A BINARY FILE

Als je een binary file wil maken om waarden in te steken van het primitieve type, strings en andere objecten. Dit doen we door `ObjectOutputStream` te gebruiken. Dit moet plaatsvinden in een try-block want elke code die een binaire code heeft van I/O kan een `IOException` throwen.

Opening een binary file: `ObjectOutputStream outputStream= new ObjectOutputStream (new FileOutputStream ("number.dat"));`

`Number.dat` bestaat dus nog niet en is een lege file. Stel dat het wel al zou bestaan, zal de bestaande content verwijderd worden zodat de file leeg start.

De constructor van een `ObjectOutputStream` kan geen string als argument gebruiken maar de constructor van `FileOutputStream` kan dat wel. `ObjectOutputStream` heeft wel een constructor die `FileOutputStream` als argument gebruikt.

WRITING PRIMITIVE VALUES TO A BINARY FILE

Geeft geen methode zoals `println` maar wel een methode genaamd `writeln` dat één enkele int value kan toevoegen aan een binary file. Deze methode kan een `IOException` throwen. Een binary file heeft geen lijnen of

andere mogelijkheden om de data items van elkaar te scheiden. Ze komen allemaal direct achter elkaar. je kan een stream van de klasse ObjectOutputStream gebruiken om een value van eender welke primitive type te gebruiken.

```
- public ObjectOutputStream(  
    new FileOutputStream([Bestandsnaam]))  
    public ObjectOutputStream(  
        new FileOutputStream(new  
            File([Bestandsnaam])))  
- public void close()  
    throws IOException  
  
- public void writeInt(int n)  
    throws IOException  
- public void writeLong(long n)  
    throws IOException  
- public void writeDouble(double x)  
    throws IOException  
- public void writeFloat(float x)  
    throws IOException  
  
- public void writeChar(int c)  
    throws IOException  
- public void writeBoolean(boolean b) throws  
    IOException  
- public void writeUTF(String  
    aString) throws IOException  
  
- public void writeObject(Object anObject)  
    throws IOException,  
    NotSerializableException,  
    InvalidClassException
```

! als we outputStream.writeChar('A') hebben verwachten ze eigenlijk een int value daarom gebeurt er eigenlijk een type casting maar dit wordt niet expliciet geschreven.

WRITING STRINGS TO A BINARY FILE

Hiervoor gebruiken we de methode writeUTF. Vb) outputStream.writeUTF("Hi mom");

Wat als je output hebt die een combinatie is van int, double en String? → Dan hebben we expliciet nodig in welke volgorde deze verschillende types voorkomen want we gebruiken een verschillende methode om deze verschillende types te lezen.

SOME DETAIL ABOUT WRITEUTF

Alle primitive types hebben op voorhand al een vast aantal bytes dat het zal nodig hebben, bij String is dit niet het geval. Langere Strings hebben meer bytes nodig dan korte Strings. Daarom schrijven we een beetje extra informatie aan het begin van je string om duidelijk te maken hoeveel bytes de readUTF zal moeten lezen. Er is wel een verschil tussen hoeveel bytes het moet lezen en hoeveel characters het moet lezen. Gelukkig zijn alle ASCII karakters gestockeerd in byte. Dus als je ASCII gebruikt, mag dit geen probleem vormen.

READING FROM A BINARY FILE

Om een binary file te lezen, kan je gebruik maken van `ObjectInputStream`. Elke `ObjectOutputStream` heeft een `ObjectInputStream`. Een methode is bijvoorbeeld `readInt`.

Een binary file openen om het te lezen gebeurt op dezelfde manier maar dan met `ObjectInputStream`:

```
ObjectInputStream inputStream= new ObjectInputStream (new FileInputStream (fileName));
```

Ook hier heeft `ObjectInputStream` geen constructor dat een `String` als argument gebruikt. Maar wel de klasse `FileInputStream`. Deze kan een `FileNotFoundException` throwen wat een soort `IOException` is.

`ObjectInputStream` geeft je de mogelijkheid om verschillende types te lezen van dezelfde file. Ze moeten natuurlijk ook in de juiste volgorde staan, als de volgende data niet matched met het type, zal het resultaat hoogstwaarschijnlijk niet kloppen.

OPENEN EN SLUITEN BINAIR BESTAND VOOR INPUT

```
- public ObjectOutputStream(  
    new FileInputStream([Bestandsnaam]))  
    public ObjectInputStream(  
        new FileInputStream(new  
            File([Bestandsnaam])))  
- public void close()  
    throws IOException  
  
- public int readInt() throws EOFException,  
    IOException  
- public int readLong() throws  
    EOFException, IOException  
- public int readDouble() throws  
    EOFException, IOException  
- public int readFloat() throws  
    EOFException, IOException  
  
- public Object readObject() throws  
    ClassNotFoundException,  
    InvalidClassException,  
    OptionalDataException, IOException  
  
    - public int readChar() throws  
        EOFException, IOException  
    - public int readBoolean() throws  
        EOFException, IOException  
    - public int readUTF() throws IOException  
        UTFDataFormatException
```

DE KLASSE EOFEXCEPTION

Veel methodes die gebruikt worden om te lezen zullen dit soort exception throwen wanneer ze proberen te lezen voorbij het einde van een file. Deze exception kan testen of het einde van de file al aangekomen is. Dit doormiddel van een whileloop wiens boolean expression true is, het is eigenlijk een soort infinite loop maar het

komt wel tot een einde wanneer het einde van de file is aangebroken, dan is een exception thrown zodat het overgaat naar de catch block ipv de try block.

```
import java.util.logging.Logger;

/**
 *
 * @author fgailly
 */
public class BinaireInputDemo {

    public static void main(String[] args) {
        String bestandsNaam = "getallen.dat";
        try {
            ObjectInputStream inputStream =
                new ObjectInputStream(new FileInputStream(bestandsNaam));
            System.out.println("Lezen van de positieve getallen");
            int eenGetal = inputStream.readInt();
            while(eenGetal > 0){
                System.out.println(eenGetal);
                eenGetal = inputStream.readInt();
            }
            System.out.println("Alle getallen gelezen !!");
            inputStream.close();
        } catch (FileNotFoundException ex) {
            System.out.println("Probleem bij het openen van het bestand");
        } catch (IOException ex) {
            System.out.println("Probleem bij het lezen van het bestand");
        }
    }
}
```

soms wordt er een andere manier gebruikt worden vb) het einde van de file is aangebroken als er een negatief nummer wordt getypt. Hiervoor mag je dan geen negatieve nummers nodig hebben in je file. Deze exception kan eender welke date stockeren.

BINARY-FILE I/O WITH OBJECTS AND ARRAYS

Hoe lezen we objecten die geen Strings zijn? → object Serialization: kan een object voorstellen als een rij bytes dat kunnen geschreven worden als een binary file.

Public class Species implements Serializable

Serializable is een interface in de package java.io. Deze interface is leeg dus we hebben additionele methoden om te implementeren. Hoewel dit vrij nutteloos lijkt, vertelt deze klas Java om een klasse serializable te maken

```
Import java.io.Serializable;
```

- Interface **Serializable** is een lege interface
- Men moet geen methodes implementeren
- Zorgt ervoor dat de klasse serialiseerbaar is
 - objecten van deze klassen wegschrijven naar een binair bestand met de methode **writeObject**
 - objecten lezen met de methode **readObject()**
 - Object moet wel nog gecast worden naar juiste type

De objecten moeten eerst nog getypecast worden. inputStream is niet van het type Species maar zou het wel moeten zijn:

```
readOne= (Species)inputStream.readObject();
```

inputStream is van het type ObjectInputStream dan geconecteerd is aan een file. readOne is van het type Species. readObject geeft de waarde van een object van het type Object. Als we willen dat het van het type Species is → type casten.

SOME DETAILS OF SERIALIZATION

- Wanneer is klasse serialiseerbaar:
 - Implementatie interface `Serializable`
 - All instantie variabelen van klasse zijn ook objecten van een serialiseerbare klasse
 - De directe superklassen van de klasse zijn serialiseerbaar of definiëren een default constructor

Wat is het effect van een klasse serializable maken? Er is geen direct effect op de klasse, maar wel op hoe Java een I/O file aanpakt. Als een klasse serializable is, geeft java een serial number aan elk object van die klasse dat het schrijft naar een stream van het type `ObjectOutputStream`. Als bijvoorbeeld een object meer dan ene keer gebruikt wordt, gebruikt Java in de plaats het serial number. Dit zorgt ervoor dat de grootte van de file kleiner wordt. Niet alle klassen zijn serializable gemaakt voor security reasons.

ARRAY OBJECTS IN BINARY FILES

Java behandelt Arrays op dezelfde manier als een object. Dus we kunnen `writeObject` gebruiken om een hele Array te schrijven in een binary file. De Array heeft een base type en deze klasse moet serializable zijn.

```
toFile.writeObject(group);    (creëren van de file)
```

```
Species[] myArray= (Species[])fromFile.readObject();
```

GRAPHICS SUPPLEMENT

Gebruiken ook interfaces of GUI's. ! op p833 vind je een programma om een JFrame te maken om een file te verwijderen. Als je een file wil verwijderen type je de naam van de file en druk je op de remove file button.

```
fileNameField.setText("enter file name here");
```

zie voor de rest cursus!