11: Static, final en abstract

Static

Wanneer men van een welbepaald attribuut slechts één enkele kopie wil bewaren, onafhankelijk van het aantal objecten dat er van die klasse is aangemaakt.

Heeft dus als doel om iets per klasse te laten bestaan en niet per aangemaakt object. (De klassenaam moet dus worden gebruikt om dergelijk lid te benaderen)

Indien we *static* toepassen op een methode, betekent het dat we te maken hebben met klassenmethode. Activering => naam van de klasse
(net zoals main, ook maar éénmaal mogelijk)

Final

= **CONSTANTEN.** Indien er nog eens *static* bijstaat, wordt er voor de klasse slechts één kopie van bijgehouden en moeten er niet noodzakelijk objecten zijn om ze te kunnen gebruiken.

- *Finale* **attributen** (**niet static**) kunnen ook bij constructie een waarde krijgen. Ze kunnen in andere methodes wel geen andere waarde meer krijgen.
- *Finale* **Klassen** zijn klassen die niet kunnen worden uitgebreid. (geen specialisatie, deelklassen). Alle methoden van een finale klassen zijn per definitie final. Vb: klasse **String**
- *Finale* **methoden** in een niet-final klasse kunnen **niet** worden overschreven. Ze worden in-line gecompileerd.

Redenen voor gebruik Final:

- De compiler kent de code volledig en dat er bij het uitvoeren van het programma niet moet worden uitgezocht of de methode ev. Is overschreven en of er moet rekening worden gehouden met dynamische binding
- Nadeel: extra plaats die in de code in beslag neemt
- Voordeel: optimalisatie uitvoeringstijd
- Veiligheid. Bv klasse String: We gebruiken Strings als padnamen. Indien deze objecten van deze klasse mutable zou zijn, zouden er veiligheidslekken kunnen ontstaan tussen de controle op toegangsrechten van de file en het gebruik van het bestand.

In-line compilatie: De compiler voorziet bij functies die in-line gecompileerd zijn niet in een echte functieaanroep (zie p105), maar zal hij de opdrachten zelf genereren die nodig zijn om de functie haar werk te laten doen.

- ⇨ Voor elke functieaanroep zijn telkens alle opdrachten voor deze functie expliciet in de eigenlijke bytecode geïntegreerd.

Abstract

- Verplicht overriding. “Het ding is onvolledig en moet worden gespecialiseerd door afleiding om het te kunnen gebruiken”.
- Tegengesteld aan final. Komen nooit samen voor
- Abstracte methode => geen body. Vereist dat ook de klasse waar ze deel van uitmaakt ook abstract is
- Van een abstracte klasse kunnen geen objecten worden aangemaakt!
- Kan weldegelijk over een constructor bezitten (om code te sparen in afgeleide klassen
- Afgeleide klassen die een abstracte methode uit de bovenklasse niet zouden implementeren, moeten zelf ook abstract zijn.

Een klasse wordt abstract gemaakt als volgende **drie voorwaarden** voldaan zijn:
- ⇨ Er meerdere afgeleide klassen zullen zijn
- ⇨ Je wil alle objecten van de deelklassen ook kunnen behandelen als object van de abstracte bovenklasse
- ⇨ Een object van de abstracte bovenklasse heeft geen betekenis

12: Interfaces

Meervoudige overerving is zinvol wanneer de nieuwe klasse meerdere contracten wenst te combineren en sommige of alle implementaties van die contracten.
- ⇨ Twistpunt in OOP-wereld. Concept bestaat wel in C++, niet in Java.

Interface(s)

- definieert een type op een abstracte manier als een verzameling methoden of andere types die het contract vormen voor dat type, een bepaald gedrag voorschrijven.
- Bevatten geen implementatie
- Kan geen objecten van interfaces aanmaken
- Zorgt voor interoperabiliteit: Een “bril” waar mee je naar objecten kijkt.

- Kan bekeken worden als een klasse met uitsluiten abstracte methoden en dus abstract gedrag. Kan enkel *final* attributen hebben.
- **Let op:** Geen interface met enkel constanten definiëren => slechte programmeergewoonte!!
- Klasse kan verschillende interfaces implementeren
- Omdat een interface ook een type is, kunnen er wel referenties naar een interface worden gedeclareerd; hierdoor wordt **dynamische binding** mogelijk.
- Alle methoden van interfaces zijn per definitie abstract

Interface versus abstracte klasse

Abstracte klasse: klasse die onvolledig is, vereist verdere specialisatie.

Interface: slechts een specificatie of een voorschrift van een gedrag. Moet niet noodzakelijk zorgen voor specialisatie zoals overerving.

Men kan slechts van één enkele bovenklasse erven, maar wel meerdere interfaces implementeren.
Interfaces voorzien ook in de mogelijkheid om referenties naar functies te voorzien die bij het schrijven van de code nog niet gekend zijn. **Niet mogelijk bij overerving.**

13: Wikkelklassen en opdrachtlijn

Primitieve types: Eenvoudig type. Waarde hiervan kan niet meer opgedeeld worden in eenvoudigere waarden.
In java zijn dit:

- boolean
- int
- double
- char
- byte
- short
- long
- float

Soms is het handige om primitieve waarden te behandelen als objecten, om bv. In collections op te slaan.

⇨ **Wrapperclass**: Omhult de primitieve waarde in een object van de corresponderende wikkelklasse. (Wikkelklasse naam = primitief type naam met hoofdletter)

Wrapperclasses:

- Zijn final en hun objecten zijn immutable
- beschikken over 2 constructoren:
  - constr die als argument de waarde krijgt van het primitief type en die er een object van de corresponderende wikkelklasse van maakt
  - constr met een string parameter die deze converteert naa de initiële waarde van het object; indien het niet gaat om een valabele stringvoorstelling, gooit de constructor een NumberFormatException op
- voorzien een aantal methoden die toelaten de primitieve waarde om te zetten van en naar het wikkelobject en van en naar een Stringobject.

Boxing: converteert automatisch de primitieve waarde naar het corresponderend wikkelobject.
Unboxing: converteert automatisch het wikkelobject naar de corresponderende primitieve waarde. (**Vanaf Java 5**)
Mechanisme van automatische omzettingen => **auto boxing**

Auto boxing:

- Werkt deels bij logische uitdrukkingen die relationele operatoren gebruiken, maar het gedrag is implementatieafhankelijk. Men gebruikt dus best de overschreven methode *equals()*.
- Zorgt voor extra overhead en dus vertraging bij het uitvoeren
- De primitieve waarde wordt niet steeds omgezet naar een object.
- Opletten dat de referentie niet null is!!

Argumenten op de commandline doorgeven

Bij public static void main(String[] args), is het via args dat je argumenten vanuit de commandline kan doorgeven. De argumenten worden opgevangen in de Stringtabel. Er is geen beperking op de lengte van de tabel en er is geen nood om het aantal elementen door te geven: dit kan je makkelijk achterhalen mbv .length (in tegenstelling tot C++)
Nog een verschil met C(++): Het nulde element is niet de naam van het commando of het programma. Deze is immers reeds gekend in het prog: Het is de naam van de klasse waarvan de methode main deel van uitmaakt.

14: Exception Handling

Kan op 2 manieren:

- als onderdeel van de programmeertaal worden geïmplementeerd
- met behulp van de taal

We gebruiken foutafhandeling in de taal.

Traditionele foutafhandeling:

- ⇨ **Via de return waarde van een functie**
  - **Voordeel**: Elke functie wordt verlaten na de return opdracht. Alle lokale variabelen worden netjes opgekuist.
  - **Nadeel:** In de hele aanroephiërarchie moet telkens opnieuw de teruggeefwaarde van een functie worden gecontroleerd. Functies die niets met de fout te maken hebben, worden ahw vervuild met code die niet relevant is voor de betreffende functie.
- ⇨ **Foutnummering** : bv Unix system-calls die -1 teruggeven indien de opdracht niet kan uitgevoerd worden. (alle andere waarden = ok). Nagaan wat de reden is van de fout: globale variabele errno (slechts zinvolle waarde na een niet geslaagde systeemaanroep)
  - **Voordeel:** Aanvulling van de return, waarbij het niet de teruggeefwaarde zelf is die de soort fout bepaalt, maar wel een globale foutindicatievariabele.
  - **Nadeel:** Zelfde als bij return.

⇨ **Assert, abort, exit**

- Assert:
  - Ingebouwde procedure.
  - Argument: voorwaarde (test waar het om draait)
  - Indien onwaar = foutmelding op scherm + programma wordt afgebroken.
  - Na compilatie: Uitvoeren met –enableassertions of –ea. Dit laat toe om enkel in ontwikkelfase het te gebruiken.
  - **Noodmaatregel: stoppen van programma. Mag NOOIT in definitief programma voorkomen.**
- Abort
  - Zelfde familie als assert: ook acute noodmaatregel, breek programma af
  - Heeft geen parameters
  - Foutmelding: “Abnormal program termination”
- Exit:
  - Zelfde familie als voorgaande
  - Het programma wordt onmiddellijk onderbroken en er wordt een zelf gekozen exit-code teruggegeven naar het proces dat het programma heeft opgestart

Error-handlers:

- Foutbehandelende functies.
- Worden geactiveerd bij constatatie van een fout.
- Programmeur kan zelf zo’n functie ontwerpen
- Taak: Fout oplossen en terugkeren naar de plaats in het prog waar de fout werd geconstateerd en de functie werd opgeroepen.

Nadeel traditionele foutafhandeling:

- Programmeur moet met veel discipline te werk gaan
- Laat geen recovery of herstel toe

Exception Handling:

- Wanneer in een functie een overwachte gebeurtenis optreedt en die functie weet er zelf geen raad mee, dan kan die een *exception throw*en. De *catch* functie vangt de fout op en handelt de situatie af.

Vb:

⇨ indexfout in tabel
⇨ geheugen is opgebruikt
⇨ overflow bij berekening
⇨ ….

Exceptions kosten wel wat **overhead.**
Exceptions zijn van de basisklasse *Exception* die zelf is afgeleid van de klasse *Throwable.*

Objecten van de klasse *Error* worden ook opgeworpen, maar kunnen niet worden opgevangen. => Fouten binnen de run-time omgeving van Java: zeldzaam en bijna altijd **fataal.**

2 Groepen:
1. **Runtime-uitzonderingen:** kunnen worden opgevangen, **niet noodzakelijk.** Indien ze niet worden opgevangen, stopt het prog en wordt de uitzondering gemeld door de Java-virtuele machine.
(bv OutofBounds, NullPointerException, NumberFormatException, …)

2.**Alle andere uitzonderingen:** bv I/O fouten, zelfgemaakte uitzonderingen. Deze **moeten** in het programma opgevangen worden, anders genereert de compiler een fouterror. Indien men de fout niet opvangt binnen de functie, moet de hoofding throw …. Bevatten!!

Het **uitzonderingsmechanisme** komt in werking dmv het woord *throw*. Dan zijn er 2 opties.

- Ofwel vangt de programmeur de fout in zijn eigen functie op, dmv *try* en *catch*.
- Ofwel vangt hij de uitzondering **niet** op en worden alle lokale klasse-objecten netjes opgeruimd en wordt de functie ter plaatse beëindigd, alsof het een return is.

Optie 2 kan zo een tijdje doorgaan, tot een functie wel aangeeft dat ze geïnteresseerd is in de fout en deze wil behandelen. Alles wat op de stack staat wordt keurig opgeruimd tot op het punt waar de uitzondering wordt opgevangen.

Als er geen enkele functie geïnteresseerd is in de fout, dan komt het uiteindelijk in de main terecht. Indien deze de fout ook niet kan/wil afhandelen wordt ze verder geworpen naar JVM, die na het plaatsen v.e. foutboodschap het prog zal beëindigen.

Java maakt gebruik van het **termination-model:** De functie waar de fout zich voordeed, wordt beëindigd. Het programma gaat door met de opdracht na de *catch*-opdracht die de fout heeft afgehandeld.

Catch: Heeft een parameterlijst, vglkbaar met die van een methode: bevat een uitzonderingtype gevolgd door de naam van de formele parameter (free to choose).
Finally: Komt na de catch. Dit gedeelte wordt **altijd** uitgevoerd.

Zelfgemaakte uitzonderingsklasse

- Moet een deelklasse zijn van Exception
- Kunnen een message meegeven aan constructor via super(message)
- Kunnen ook attributen meegeven!!

Uitzonderingen doorgeven

- De methode/constructor moet in de hoofding vermelden dat hij een fout kan werpen (*throw …*)
- In het corresponderende *catch*- blok wordt na het schrijven van de foutmelding een *throw*-opdracht uitgevoerd.

Uitzondering al dan niet verplicht opvangen

- Indien je wil verplichten een niet-verplichte uitzondering op te vangen, dan moet je zelf een uitzonderingsklasse definiëren, die door zijn naam ook duidelijker de fout aangeeft.

Catch(Exception e): Handler die alles mogelijke excepties opvangt. Staat als laatste in de lijst.

15: Objecten klonen

Als men variabelen (**niet-primitief**) aan elkaar toewijzen, dan wordt de referentie overgenomen en wijzen ze allebei naar hetzelfde object in het geheugen.
Vaak aangewezen om volledige kopie te nemen. (privacy leak!)

.clone(): Geeft een **nieuw** object terug van het type Object. Moet dus gecast worden naar de juiste verm.
Let wel: Voorziet standaard enkel in kopiëren van de attributen van een primitief type, referenties naar objecten worden ook gekopieerd, maar **niet** de objecten zelf! Gevolg: soms moet je de .clone() methode overschrijven en aanvullen om objecten zelf te kopiëren.

Tweede factor: interface **Cloneable.** Lege interface. Wanneer men hem implimenteert: .clone() mag alle attributen van een object van die klasse kopiëren.

Factor 3: (indien niet geïmplementeerd: uitzonderingsfout CloneNotSupportedException)

Houding van een klasse tov klonen

- ⇨ Ondersteunen van de methode clone(). Klasse implementeert Cloneable en declareert de clone-methode op zo'n manier dat ze geen uitzondering opwerpt.
- ⇨ Voorwaardelijk ondersteunen van clone(). Objecten kunnen slechts worden gekloond als de volledige inhoud kan worden gekloond. **Hoe?** Interface Cloneable implementeren, clone() moet elke CloneNotSupportedException verder opwerpen.
- ⇨ Alleen deelkassen clone() te laten ondersteunen en dus niet publiek toe te laten. **Hoe?** Interface niet implementeren, maar als de standaard manier om velden te kopiëren niet volstaat, voorziet ze wel in een protected clone-implementatie die haar velden correct kloont.

- ⇨ Klonen **expliciet verbieden. Hoe?** Interface niet implementeren en voorzien in een final clone-methode die steeds CloneNotSupported Exception opwerpt.

Soms is het eenvoudige om kopies te maken op een alternatieve manier, zoals bvb een goed werkende copyconstructor.

16: Invoer en uitvoer met *STREAMS*

Overzicht van gegevenscoderingen

- ⇨ ASCII (interne voorstellingscode) en ISO 88959-1 (gebruikt in vele system, met ASCII als subset + 128 meest gangbare tekens in West-Europa)
- ⇨ Unicode: Vooral door Amerikaanse computerbouwers gespecifieerd. Doel: Allerlei tekens van over de hele wereld te kunnen voorstellen in meertalige software. Java 5.0 is compatibel met Unicode 4.0
- ⇨ UCS: **U**niversal multiple-octet coded **C**haracter **S**et : roepnaam van ISO/IEC 10646 standaard. 3 Niveau's. Niveau 1 zonder tekens uit Zuid-Aziatische talen, Niveau 3 laat toe om tekens te coderen die in voorstelling een combinatie zijn van tekens.
- ⇨ UTF: **U**CS **T**ransformation **C**ode. Transformatiealgo dat Unicode tekens omzet naar blokken van 2 bytes(UTF-16) of 1 of meerdere bytes (UTF-8).
- ⇨ Binair Formaat: Getallen die anders worden opgeslagen dan ze worden voorgesteld. Worden voorgesteld zoals ze normaal in het geheugen gecodeerd worden.
- ⇨ Objectenformaat: Java. Objecten kunnen in hun geheel op stream worden geplaatst of afgehaald: **Serialization** en **deserialization.**

Wat zijn streams?

Een **stream** is een object dat een geordende stroom van gegevens voorstelt waarvan de lengte niet doozakelijk op voorhand kan worden bepaald. Wordt dus sequentieel verwerkt, van voor naar achter.

Invoerstreams: bv bestand, toetsenbord,netwerkverbinding
Uitvoerstreams: bv het scherm, een bestand, een andere computer op het netwerk

Twee klassenhiërarchieën:

- character streams: Unicode lettertekens
- byte streams: gegevens als digitale stroom van 8 bits.

Alle I/O maakt gebruik van streams.

- ⇨ Er bestaan streams voor invoer (*InputStream, Reader*) als uitvoer (*OutputStream, Writer,…)*
- ⇨ Streams behandelen gegevens als bytereeksen(*InputStream, OutputStream*) of als UNICODE – lettertekens (*Reader, Writer)*
- ⇨ Streams kunnen gekoppeld zijn aan bestanden (*FileReader, FileWriter, …*), aan System.in, System.out en System.err (= standaard I/O kanalen), aan tabellen in geheugen (*StringReader, StringWriter*), …
    - Lezen geeft steeds IOException
    - ArrayIndexOutOfBoundsException: wanneer er geen arg op de commandline is meegegeve
    - FileNotFoundException: opgegeven bestand kan niet worden gelezen.

Overzicht van streamklassen

Op basis van opzet:

- ⇨ **Data Sink streams:** lezen van of schrijven naar speciale gegevensverzamelingen zoals strings, bestanden of pipes. Overzicht zie p 158
- ⇨ **Gegevensverwerkende streams:** verzorgen tijdens het lezen of schrijven een of andere operatie zoals buffering of karaktercodering. Overzicht zie p 159
- ⇨ **Character streams:** Reader en Writer zijn hiervan de astracte bovenklassen. De afgeleide ervan implementeren gespecialiseerde streams en worden verdeeld in **lezen/schrijven van/naar dataverzamelingen en gegevensverwerkend streams.**
- ⇨ **Bytestreams:** InputStream en OutputStream zijn basisklassen voor de bytestream. De deelklassen implementeren gespecialiseerde streams en worden verdeeld in **lezen/schrijven van/naar dataverzamelingen** (zie p161)

Basisstreamklassen

- Reader en Writer
  - Abstracte klassen
  - Wel degelijk zin om in deze klassen methoden te def.: Kunnen gebruikt worden door de objecten van de afgeleide klassen.
  - Belangrijkste methode: read() met eventueel int off als index van tabel en int len = #lettertekens. Zie p162 e.v
  - Belangrijkste methode : write() met ook off en len als opties
  - Streams zijn vaak gebuffered. (= interne buffer). Flush verplicht hem onmiddellijk uit te schrijven. *close* doet flush en afsluiten.

- FileReader en FileWriter
  - I/O routine van FileWriter zet als laatste stap interne Unicode om naar code van lokale bedrijfssysteem. (omgekeerd bij FileReader)
- InputStream en OutputStream
  - Abstracte basisklassen bytestreams
  - Sequentieel verwerken van binaire bestanden, zoals GIF of MIDI
- PrintWriter
  - Rechtstreeks afgeleid van Writer
  - Eenvoudigste manier voor uitvoer naar tekstbestand
  - Constructor met arg . Writer w en eventueel boolean autoflush (indien true = elke keer flush na println.)

Andere streams

- Gebufferde streams
  - Zodra we letterteken nodig hebben van een invoerstroom = efficiënter om meteen groter aantal in te lezen en tijdelijk int geheugen te plaatsen. => kan sneller geraadpleegd worden
  - BufferedReader, BufferedWriter, BufferedInputStream, ….
  - Readline()

- LineNumberReader
  - Hetzelfde gedrag als BufferedReader (is een afgeleide ervan)

    - Houdt volgnummers ingelezen lijnen bij.

- PushBackReader
    - Zelfde mogelijkheid als putback bij C++
    - Unread() = letterteken lezen ongedaan maken

Serialisering van objecten

Serialization is de mogelijkheid om objecten te bewaren op een byte-stream die kan getransfereerd worden over een netwerk, die kan bewaard worden in een bestand op een harde schijf of in een gegevensbank. Ook het omgekeerde is mogelijk = **deserialization.**

Er wordt telkens een object van de algemene bovenklasse **Object** teruggegeven.

De klasse serialiseerbaar maken

1. Zijn klasse implementeert de interface Serializable
2. Waarom zijn niet alle klassen auto serialiseerbaar? => Safety Rozijnen

Standaarmethode: elke veld van het object dat niet-static is, te serialiseren. Primitieve types worden omgezet volgens hun interne coderingen in het geheugen, referenties naar andere objecten moeten verwijzen naar objecten die ook deze interface implementeren en dus serialiseerbaar zijn.
Vereist ook dat ofwel de bovenklasse van het gerialiseerd object een constructor bij verstek bezit die kan worden geactiveerd bij het deserialiseren, ofwel dat die bovenklasse zelf de interface Serializable implementeert.

In sommige gevallen mag een serialiserbare klasse welbepaalde objecten niet serializeren. Elke poging om zo'n object te serialiseren werpt een NotSerializableException op.

Indien serialisering niet correct zou werken voor een klasse (standaardmeth), dan kunnen er binnen de klasse twee private methodes writeObject en readObject worden gedeclareerd, die dan serialiserend werken.

Versiebeheer

Samen met de klassedefenitie wordt een vingerafdruk (hascode) mee opgeslagen op de stream; uniek voor elke klassedefinitie. Indien ze niet overeenkomen = deserialiseringsproces kan niet doorgaan.

Mogelijkheid is wel dat een klasse van zichzelf aanduidt dat ze compatibel is met één van haar vorige versies. Om dit te verwezenlijken moet de vingerafdruk van de vorige klassedefinitie opgenomen worden in de nieuwe versie van de klasse.
(mbv *serialver*)

17: OGP in C++

- ⇨ Alle attributen en methoden zijn bij verstek privaat
- ⇨ Methodes in C++ worden zelden in de klassendeclaratie volledig gedeclareerd. Reden is dat derg. methodes inline worden gecompileerd, want enkel wenselijk is voor korte methodes.
- ⇨ In klassedeclaratie meestal alleen **methodedeclaratie:** enkel methodehoofding gevolgd door ;
- ⇨ C++ kan ook constructor overloading hebben
- ⇨ Objecten worden niet aangemaakt met new, maar door bij declaratie van het object tussen haken de argumenten van de constructor te vermelden achter de naam van het object. En objecten zijn **geen pointers!**
- ⇨ In C++ bestaat er een destructor. De code die zich hier bevindt wordt uitgevoerd telkens er een object van de klasse wordt verwijderd. In C++ heb je geen garbage collection!!

Separate of afzonderlijke compilatie: alle modules die in het prog worden gebruikt, worden apart gecompileerd. De verschillende objectfiles worden nadien aan elkaar gelinkt tot één uitvoerbare run-file.

- Bij werken aan het project met meerdere personen
- Indien een softwarebib wordt aangekocht zijn alleen de objectfiles beschikbaar, niet de bronbestanden.

De **copy-constructor** wordt in drie gevallen gebruikt:

- bij een declaratie van de vorm “Student student3(student1)”
- Bij het teruggeven van een object als functiewaarde
- Indien in een functiehoofding een klassenobject als formele waardeparameter vermeld staat, zal bij de functieaanroep de actuele parameter (het argument) ook gekopieerd worden naar de formele parameter met behulp van de copy-constructor.

Deze constructor is **altijd standaard aanwezig.**

Operator overloading

Men kan zelf operatoren herdefiniëren voor klassenobjecten => **operator overloading.**

Ongeveer alle operatoren komen hiervoor in aanmerking, behalve . en de scope :: . Wil je een operator met één operand overloaden, dan maak je een functie zonder parameter.

Toekenningsoperator herdefiniëren

- iets complexer
- resultaat van de toekenningsoperator: linker operand, waarvan de waarde gelijk is geworden aan die van de rechter operand.
- Als het “=” na de +, -, …. Komt
- Return *this: wijst naar object waarvoor lidfunctie wordt geactiveerd.

- Ipv void komt er nu het type van de klasse
- De functiewaarde moet dus gebeuren via return by reference. Voor de naam van de functie moet een & worden geplaatst.
- Return by reference is ook noodzakelijk bij het herdefiniëren van de indexoperator []

Bevriende operatoren
- Operatoren die bewerking doen met objecten van verschillende klassen
- Zo'n operator is géén lid van de klasse, maar is wel mee bevriend: krijgt toegang tot de private leden.
- Kan resulteren in een obejct van die klasse

Overloading van >> en <<
- implementatie als friend-functie
- ostream en istream
- ook een & voor "operator"
- return os of is

Overerving

Protected: In een afgeleide klasse kunnen de lidfuncties de beschermde leden van de basisklasse ook gebruiken en wijzigen.

Een klasse wordt als afgeleid aangeduid door de vermelding **afgeleide klasse : public basisklasse.**

Het is altijd eerst de constructor bij verstek van de basisklasse die wordt geactiveerd. Daarna wordt een eventuele aanwezige constructor in de afgeleide klasse geactiveerd.

Ipv super bij Java, is het nu onder de vorm van een *initializer* (constructor bovenklasse erachter met :)
Het systeem van *initializer* kan ook gebruikt worden om attributen van een klasse te initialiseren.

Overriding bestaat ook C++. Wil men om een of andere strange rozijn voor een afgeleid object toch nog de functie van de basisklasse gebruiken, dan kan dat door vermelding van de scope:operator samen met de aanduiding van de basisklasse.

Public overerving:
- ⇨ is een-relatie
- ⇨ objecten van de afgeleide klasse:
  - kunnen aan de lidfuncties van de bovenklasse
  - een sub-object van de bovenklasse bevatten
  - kunnen niet aan de private leden van de bovenklasse: mutatoren en/of inspectoren
- ⇨ meest gebruikte vorm van overerving en meest aangewezen

Private overerving

- ⇨ geen echte overerving
- ⇨ object van de afgeleide klasse:
  - kan geen gebruik van de methodes van de bovenklasse
  - weet niet dat hij er van afgeleid is
  - de is relatie bestaat niet
  - het is alsof object van de bovenklasse deel uitmaakt van de afgeleide klasse
- ⇨ bovenklasse is een privaat deelobject van de afgeleide klasse

Verschil private – public:
Public:

- Interface wordt overgeërfd: publieke deel basisklasse ook beschikbaar als publieke interface in de afgeleide klasse
- Gebruiker mag met afgeleid klassenobj doen wat hij met een basisklassenobject ook kan.

Private:

- implementatie wordt overgeërfd: de programmeur van de afgeleide klasse kan gebruik maken van de functies van de basisklasse, maar deze zijn ontoegankelijk voor de gebruikers van de afgeleide klasse
- interface wordt NIET geërfd, de IS EEN relatie is doorbroken.

Polymorfisme en dynamische binding

Men mag aan een pointer naar de basisklasse het adres toekennen van een afgeleid object zonder dat er typeconversie moet gebeuren (typecast).
Nut:

- verzamelingen aanmaken met ongelijksoortige objecten indien ze pointers bevatten naar elementen van de basisklasse
- er gaan geen eigenschappen verloren van de afgeleide klasseobjecten
- object waarnaar verwezen wordt verandert niet door de adresmanipulatie
- Pointer speelt essentiële rol

⇨ **IMPLEMENTATIE VAN POLYMORFISME**

In C++ werkt polymorfisme alleen bij het gebruik van pointers en referenties EN uitsluiten bij publieke overerving.

Let wel: vermits het pointers naar de basisklasse zijn, worden de functies van de basisklasse geactiveerd, niet die van de afgeleide. Oplossing: zie verder.

Soorten Lidfuncties:

- ⇨ Functies die identiek zijn voor alle klassen uit de hiërarchie
- ⇨ Functies die uitstluitend voor één enkele soort klasse beschikbaar zijn
- ⇨ Functies waarvan de functionaliteit voor alle klassen dezelfde is, maar waarvan de implementatie verschilt.

Focus op laatste punt. Als bij het schrijven van de code bekend is over welke soort objecten het gaat, dan resulteert dit in **vroege binding** of **compile-time binding:** op het moment van compilatie kan compiler bepalen welke functie moet worden geactiveerd. Dit maakt dat het programma snel is: Tijdens uitvoeren niet meer zoeken naar welk type variabele de pointer verwijst.

Virtuele functies

Dynamische binding:

- ➔ kost beetje uitvoeringstijd en extra geheugenplaats
- ➔ lidfuncties in basisklasse, waarvan men wil dat at-runtime bepaald wordt voor welk soort object ze moeten worden geactiveerd: moeten in de basisklasse **virtueel** worden gemaakt
- ➔ exact dezelfde signatuur in alle klassen (basis + afgeleiden)
- ➔ basisklasse moet de functie hebben, ook al is die leeg!

Dynamische binding is essentieel voor OO talen. Polymorfisme heeft er ook mee te maken. Het feit dat pointers naar objecten van de basisklasse ook kunnen verwijzen naar objecten van de afgeleide klassen is essentieel.

Kenmerken virtuele functies:

- signatuur is in alle klassen dezelfde
- return waarde moet dezelfde zijn
- indien een virtuele functie in de basisklasse heeft in een andere signatuur en toch dezelfde naam heeft in de afgeleide klasse, dan spreekt men van **function-hiding**, dynamische binding werkt dan niet.

Pure virtuele functie: Wanneer v.f. in basisklasse geen functiebody bezit. Wordt aangeduid bij declaratie dmv = 0 .
Er kunnen geen objecten van deze klasse worden aangemaakt => **abstracte klasse.**

Nut: In basisklasse kan worden vastgelegd dat de afgeleide klassen deze functie moeten implementeren.

Virtuele destructor

- ➔ bij basisklasse
- ➔ zorgt ervoor dat eerst de aanroep van de destructor van de afgeleide klasse gebeurt en dan pas die van de basisklasse.
- ➔ Indien deze niet virtueel zou zijn, dan wordt de afgeleide destructor niet geactiveerd !!

Templates in C++

- ➔ Bied de mogelijkheid om van een functie of een klasse een omschrijving te geven, zonder ze in detail vast te leggen.

Templates voor functies

Een functie-template is eigenlijk geen functie, maar een beschrijving van een functie waarin het type van een of meer argumenten niet worden gespecifieerd.

Template <Typename T>
Voor de compiler betekent dit : “Nu komt er een def van een functie of klasse, waarin gewerkt wordt met type T; om welk type het gaat, zal later worden bepaald.

Matching algoritme

Het genereren van nieuwe versies heeft voorrang op het matching-algoritme. Voor elk soort argument zal dus een nieuwe versie worden aangemaakt, ook al is dit niet echt noodzakelijk en kan er via standaardomzettingen worden gewerkt.

Het algoritme om overeenkomsten te bepalen in combinaties met templates, verloopt als volgt:

1. Zoek naar een gewone functie die precies overeenkomt met de benodigde versie.
2. Zoek naar een template die de functie exact definieert
3. Gebruik de normale regels voor overloaded functies en conversies
4. Indien dit alles niet lukt, genereert de compiler een fout.

Een complicatie

Als de datatypes niet overeenkomen (bv int en long), dan zal de compiler de code niet accepteren. Er is geen exacte overeenkomst en er kan gaan standaardomzetting plaatsgrijpen.

3 Mogelijke oplossingen:

⇒ Functie-template met verschillende parameters
- Werkende oplossing
- Leidt tot zeer veel code (veel mogelijkheden!)

⇒ Specificatie van het type
- Betere oplossing
- Geeft expliciet aan voor welk type de templatefuncite moet worden gebruikt

⇒ Schrijven van een niet-template functie
- Extra functie schrijven, niet –template
- Standaardconversies worden daar wel toegepast
- Niet goed: moeten 2 keer functie schrijven.

Specialisatie

Dit is wanneer we voor specifieke gevallen een eigen versie van de functie willen maken. Voor de standaardgevallen wordt gebruik gemaakt van de service van de compiler, maar voor sommige uitzonderingen (zoals hier bij strings) bepalen we zelf wat er moet gebeuren.
Bij een template-specialisatie moeten de gegevenstypes van de argumenten altijd precies overeenkomen met die van de formele parameters.

Als we template-lidfuncties apart definiëren, moet voor elke lidfunctie de templateaanduiding worden herhaald en de scope-operator bevat eveneens de klasse-type aanduiding.

Klasse-templates kunnen we op twee manieren klassen van afleiden:

- overerving zonder behoud van generieke mogelijkheden
  - ontwikkelen van een klasse die zelf geen sjabloon is

- met behoud van generieke mogelijkheden
  - afgeleide klasse is zelf een template

18: Generics, iterators, collections

Generics

Er kan bij de definitie van klassen en methodes een parameter worden opgegeven die een aanduiding is van een generisch, willekeurig type.
Gevolg => Indien dergelijke klasse/methode wordt gebruikt, moet er een echt type worden opgegeven, zodat de compiler xtra controles kan uitvoeren.

Voordelen:

- Sommige gevallen geen typecasting meer nodig
- At-runtime moet er geen controle meer gebeuren of de types compatibel zijn

Raw type: gebruik van typeaanduiding is niet steeds vereist. Let wel: vanaf Java 5 geeft dit waarschuwingen. Het laat je immers toe dat je in een lijst objecten van verschillende types opslaat = slechte programmeergewoonte.

Controle gebeurt enkel op de add, niet op de get.

Hiërarchie van types met parameter is opgebouwd op basis van het basistype en NIET op het type van de parameter !!

! Om compatibel te zijn met oudere versies, kan bv List <X> omgezet worden naar List. Gevolg: lek in de type-veiligheid !!

Er is :

- ⇨ Compile-time safety
- ⇨ Run-Time safety, als alles wordt gecompileerd en uitgevoerd in een Javaversie vanaf 5.0 !!

Indien er toch oude code moet worden gecompileerd en uitgevoerd, dan is er toch controle mogelijk bij adding objecten aan de lijst.
De lijst moet dan omwwikkeld worden in een lijst die voor deze controle zorgt.

Foutieve opdrachten(add) genereren dan een ClasCastException.

Wildcards in de parametertypes

Opzet: de methode kan om het even welke soort lijst verwerken.
Oplossing: Wildcard! We gebruiken ? bvb List <?> list.

List is zo een lijst van onbekende dingen.
Wildcard kan enkel gebruikt worden in een parameterlijst, in een constructor fzo kan dit niet gebruikt worden (compiler MOET immers weten over welke soort objecten het gaat.
List<?> is enkel **read-only**. We kunnen enkel getten, niet adden.

Begrensde wildcards

Oftewel bounded wildcard. Met een begrenzing bvb dat het van een Number of afgeleide moet zijn: List <? Extends Number > list
Gevolg: geen typeomzeting meer nodig is bij eigenlijke sommering. (zie p212)

We kunnen nog steeds geen objecten toevoegen. We kunnen ook **ondergrens** geven, dmv super ipv extends.

Iterator

Een **iterator** is een object dat gebruikt wordt om te voorzien in de sequentiële toegang tot een verzameling gegevens.
Dit betekent dat een iterator een volgorde in de elementen veronderstelt, ook al betreft het een verzameling die zelf geen volgorde oplegt aan de elementen die ze bevat.

2 Interfaces implementeren:

- ⇨ Iterator <E>
  - o hasNext(), next, remove
  - o E : type van de verzameling gegevens waarover moet geïtereerd worden.
- ⇨ Iterable <T>
  - o Iterator <T> iterator()
  - o Returnt iterator over set of elements van Type T
  - o Associatie met de verzameling gegevens die moet doorlopen worden
- ⇨ Al de verzamelingentypes uit collections-framework zo itereerbaar
- ⇨ Ook bij een zelfgemaakte klasse toepasbaar

Collections

Houders van informatie. Slaan objecten op en organiseren ze zo dat ze op een efficiënte manier kunnen worden opgezocht.

Lijst van verschillende Collections: zie p 215

19: Threads

Proces: lopend programma. Niet alle processen moeten tegelijkertijd actief zijn: ze kunnen bvb wachten op hun time slice of wachten op invoer.

Thread: een uitvoerbare deelproces/taak, waarvan er meerdere kunnen zijn in een enkelvoudig proces. Onafhankelijk van de andere deeltaken. Een draad kan communiceren met andere draden binnen hetzelfde proces, maar opletten als dit gebeurt door de waarde te veranderen van gedeelde variabelen.

Tijd moet ook verdeeld worden tussen threads. Ofwel:
- elke draag krijgt een stukje tijd. Als dit voorbij is, wordt de draad onderbroken
- Draad wordt slechts onderbroken indien hij informatie van ergens moet krijgen
    - Nadeel: op die manier kan een draad alle processorcapaciteit opeisen

Veel softwareproblemen hebben oplossing dmv controledraden (**threads of control**) (bvb Gui met display)
Systemen met één draad simuleren dit meerdradig gdrag dmv interrupts of door **pollen.**

Pollen: mixt het display-deel en het invoer-deel van een toepassing.
- ⇨ Geeft aanleiding tot somanevoegen van progdelen die functioneel eigenlijk niets met elkaar te maken hebben
- ⇨ Code stuk minder onderhoudbaar

Voor de gebruiker van een systeem is het transparant of het multi-threaded is of niet.

Opzetten van threads

Er wordt een Thread object aangemaakt, of een object waarvan de klasse afgeleid is van Thread.
Na de creatie kunnen ook zijn eigenschappen worden ingesteld, zoals beginprioriteit en naam.

Start(): Wanneer de draad klaar is om uitgevoerd te worden, wordt deze methode aangeroepen. JVM start een nieuwe draad op dan, gebaseerd op de gegevens in het draadobject. Nadien gaat de methode(draad) die de startmethode heeft geactiveerd verder. Start kan slechts éénmaal per draad worden opgeroepen.

Run(): wordt door JVM uitgevoerd na start(), is de run van het draadobject, activeert zo die draad.

Standaardmethode run doet niets.
2 mogelijkheden om wél iets te krijgen;

- ⇨ Afgeleide klasse van thread
  - o Wachten doet men via sleep = kan InterruptedException veroorzaken => moet worden gevangen!
  - o Overschrijven van run()
- ⇨ implementatie van Runnable
  - o interface implementen
  - o ook sleep en Exception opvangen
  - o opstarten wél totaal anders: anonieme Thread-objecten, mbv constructor met als argument het Runnable-object van de zelf gemaakte klasse

Thread krijgt een naam en die kan worden opgevraagd met getName()
(naamattribuut in Thread, kan dus met super ingevuld worden.)

Levenscyclus van een draad: zie p 228 !!!

Status *not runnable*
Er zijn drie manieren waarop een draad van een draaiende in een niet-draaiende toestand kan terechtkomen:

- ⇨ zijn sleep()- methode werd aangeroepen
- ⇨ de draad roept zelf de *wait()*-methode aan die als doel heeft te wachten tot een bepaalde voorwaarde is voldaan
- ⇨ de draad blokkeert op IO

Om weer runnable te maken:

- indien een draad in slaapstand werd gebracht, dan moet het voorziene aantal milliseconden verstrijken.
- Als een draad aan het wachten is op het voorkomen van een bepaalde voorwaarde, dan moet een ander object hem “wekken” bij een conditieverandering, dit gebeurt door *notify()* of *notifyAll*

Een draad stoppen
Kan enkel maar als zijn run()-methode afloopt. Of we kunnen een while-voorwaarde implementen, waarbij de run() doorgaat zolang deze while true geeft. Indien de voorwaarde niet meer voldaan is, sterft de draad een natuurlijke dood.

Methode isAlive()

Geeft waarde true indien draad is gestart en nog niet gestopt is.
Geeft waarde false weet men enkel dat de draad in toestand *New Thread* of *Dead* is.
Er is geen mogelijkheid om te weten of de draad draaien of niet-draaiend is.

Synchroniseren van draden

Er zijn veel situaties waar verschillende tegelijkertijd draaiende draden gemeenschappelijke gegevens moeten delen en zich bovendien bewust moeten zijn van de activiteit van één of meerdere draden.

(vb een draad die bestand schrijft en andere draad die bestand leest)
In dergelijke situaties delen de draden **gemeenschappelijke systeembronnen** en moeten ze op één of andere manier worden **gesynchroniseerd.**

Synchronisatie moet op 2 manieren gebeuren:

- Twee draden mogen niet tegelijkertijd toegang krijgen tot een object
  - Door middel van locking
  - Draad 2 blijft geblocked tot object opnieuw toegankelijk is
  - Met methoden *synchronised* te declarerern

- Vorm van coördinatie tussen de twee draden
  - Thread-methoden *wait()* en *notifyAll().*

Draden wachten en verwittigen

➔ wait() zorgt voor het wachten tot conditie voldaan is
➔ NotifyAll() is allen verwittigen. Gaan dan na of hun conditie om verder te gaan is voldaan
  - Beide methoden dragen de modificator final en kunnen dus door geen enkele klasse meer worden overschreven.

Opmerking

- Alles wordt uitgevoerd als zijnde gesynchroniseerd
- Een belangrijk aspect van de wait()-methode is dat wanneer ze de draad doet pauzeren, ze automatisch het afsluiten van het object opheft. Indien dit niet zo zou zijn, zou er bvb een verwittiging kunnen plaatsgrijpen nadat het slot werd weggenomen maar voordat de uitvoering van de draad werd onderbroken.
- Het testen van de voorwaarde moet **altijd** in een herhaling gebeuren. Je mag er niet vanuit gaan dat als de draad opnieuw wakker wordt gemaakt via notifyAll(), dat de voorwaarde waarop gewacht werd voldaan is.