# Objectgericht programmeren: samenvatting

## 1 Introductie

- Objectgericht programmeren: Programmeerstijl waarin programma bestaat uit klassen, met elk eigen velden en methodes die de toestand en het gedrag van het object definiëren. Overerving en encapsulatie worden gebruikt om modulariteit te verkrijgen.
- Modulair programmeren: Software ontbinden in kleinere modules die onafhankelijk ontwikkeld, begrepen, geverifiëerd en getest kunnen worden.
    - Een module in interactie met een andere module wordt een klantmodule genoemd. Om onafhankelijke development te garanderen, is het belangrijk dat er API over syntax en semantiek beschikbaar is voor de klant.
        - De correctheid van een module is afhankelijk van deze API specificaties, niet van de implementatie

## 2 Single-object abstractions

### 2.1 First steps in modular programming

- Encapsulatie: De velden van een object mogen nooit direct toegankelijk zijn voor de klant, maar informatie van een object moet verkregen worden via methodes.
- Instance methods: i.p.v. static method: Interval.getWidth(Interval interval), instance method: interval.getwidth(), interval wordt de receiver van de methode genoemd.
- Overloading: meerdere methodes/constructoren met dezelfde naam zijn mogelijk, maar ze moeten dan wel verschillend aantal parameters of verschillende parametertypes hebben om ze te kunnen onderscheiden.
- Documentatie:
    - Informele documentatie: is documentatie in woorden.
    - Formele documentatie: is documentatie in programmeertaal.
    - Postcondities(@pre): Toestandsvoorwaarden die *true* moeten zijn na het uitvoeren van een methode, gegeven dat de precondities *true* waren voor het uitvoeren.
    - Precondities(@post): Toestandsvoorwaarden die *true* moeten zijn voor het uitvoeren van een methode, indien dit niet zo is, moeten postcondities ook niet *true* zijn na het uitvoeren.
    - Private invarianten(@invar): (=representation invarianten) staan boven private velden van een klasse en zijn toestandsvoorwaarden die *true* moeten zijn voor een object in geldige toestand.
    - Publieke invarianten(@invar): staan in documentatie van klasse zelf. =toestandsvoorwaarden voor de returnwaarden van getters van een object dat *true* moeten zijn op het moment dat er geen methode of constructor wordt uitgevoerd.
- Unit test guidelines:
    - Test belangrijkste inputs
    - AssertEquals(verwachte waarde, eigenlijke waarde)
    - Test alle methodes apart → makkelijker om te zien waar het foutloopt

### 2.2 Managing complexity through modularity and abstraction

Modulariteit = taak opsplitsen in kleinere problemen die makkelijker op te lossen zijn, onafhankelijk begrepen, geverifiëerd, geïmplementeerd en getest kunnen worden, en die samen de volledige taak

oplossen. De verschillende delen kunnen dus ook in parallel ontwikkeld en geüpdate worden, zonder problemen, zolang er rekening gehouden wordt met de API van de andere modules.

→ meest effectieve manier om dit te doen is via abstracties: procedurale en data abstracties.

- Procedurale abstracties: identificeer operaties waarvoor de taak makkelijker zou zijn, moesten deze operaties in Java ingebouwd zijn en implementeer deze operatie in een aparte module.
  Bv. Stel dat applicatie sqrt nodig heeft → procedurale abstractie zegt:
  - Maak applicatie in java++ = java uitgebreid met sqrt operatie
  - Maak sqrt module die enkel sqrt operatie implementeert (deze sqrt is dus een voorbeeld van een procedurale abstactie)

  Dit is makkelijker want:

  - Klant (= developper applicatie) moet zich geen zorgen maken over implementatie sqrt
  - Sqrt developper moet zich geen zorgen maken over applicatie

  Dit is enkel mogelijk als de abstractie voldoende precies en abstract is (implementatie moet verborgen zijn en API moet zo simpel mogelijk zijn).

- Data abstracties: = datatypes die het oplossen van de taak makkelijker zouden maken, moesten ze al in Java ingebouwd zijn.
  Voordelen zijn idem aan procedurale abstracties.
- Mutable en immutable classes:
  - Immutable(@immutable in Javadoc): klant kan de abstracte toestand van de klasse niet aanpassen.
    - Heeft vaak *of(input)* methode ipv. Constructor.
    - Het is aangeraden om de velden als final markeren
  - Mutable(@mutable niet per se nodig): klant kan wel toestand aanpassen. Heeft als voordeel dat er veel minder objecten gemaakt moeten worden → betere performance. Nadeel: bij een immutable object moet er geen onderscheid gemaakt worden tussen het object en zijn abstracte toestand, het object IS de toestand. Terwijl een mutable object enkel die toestand opslaat op een bepaald moment.

## 2.3 Representation objects and representation exposure

Elke instantie van een data abstractie heeft een bepaalde abstracte toestand, gerepresenteerd door de velden van het object. Echter, in de meeste gevallen, zijn de velden van een object niet genoeg om de abstracte toestand op te slaan en moeten we gebruik maken van hulpobjecten, die representation objects (bv. een int[] of een andere data abstractie) genoemd worden.

- Representation exposure: = Het lekken van een representation object waardoor de klant deze ongewenst kan aanpassen. Bij een immutable object bv. zou de klant de immutability kunnen breken. Bij een mutable object zou het de representation invarianten kunnen breken.
  Oplossing: kopie van representation object meegeven bij getters() → @creates | result toevoegen bij Javadoc v.d. getter.
  Een andere manier van representation exposure is wanneer een klasse een al-bestaand object, dat zichtbaar is voor de klant, als representation object gebruikt. Wanneer de klant nl. het al-bestaande object aanpast, verandert het representation object ook, wat we niet willen.
  Oplossing: kopie van het object gebruiken als representation object.
  Enkel mutable objects moeten gelabeld worden als representation object aangezien velden uiteraard niet ongewild gewijzigd zouden kunnen worden als ze immutable zijn.

## 2.4 How to properly document single-object abstractions

**Concepten:**

- Een data abstractie is gedefinieerd door zijn publieke aspecten:
  - Naam van de klasse
  - De namen, parametertypes, en returntypes van de publieke methoden van de klasse
  - De parametertypes van de publieke constructoren
  - De documentatie van de klasse, de publieke methodes en publieke constructoren
- De implementatie van een data abstractie is gedefinieerd door:
  - De private velden
  - De bodies van de publieke methoden en constructoren
  - Niet-publieke methodes en constructoren
  - Documentatie van de niet-publieke velden
- De abstracte toestand van een instantie van een data abstractie is gegeven door de return waardes van de getters van de instantie.
- Geldige abstracte toestand: is een toestand waarvoor alle publieke invarianten *true* zijn.
- Representatie van een data abstractie: is de manier waarop de implementatie de abstracte toestand van een instantie opslaat. Bv. lower bound en width bij een interval i.p.v. lower bound en upper bound.
- Concrete toestand van een instantie van een data abstractie: is gegeven door de waarden van de velden van de instantie.
- Geldige concrete toestanden: bv.(lowerbound: 5, width: -4) is geen geldige concrete toestand van een interval aangezien de invariant width >= 0 niet behouden is.
- Verband tussen concrete en abstracte toestanden: De getter's geven altijd een geldige abstracte toestand als de concrete toestand geldig is.

**Documenteren:**

- Documentatie van een klasse:
  - @immutable als een instantie van een klasse niet gemodifiëerd kan worden.
  - Als niet elke mogelijke abstracte toestand geldig is, beschrijf je de geldigheid van een abstracte toestand in publieke klasse invarianten. Dit kan bovenaan de klasse m.b.v. @invar of boven elke getter m.b.v. @post.
- Documentatie van de velden:
  - Alle velden van elke klasse moeten als private gemarkeerd worden
  - Als niet elke mogelijke concrete toestand geldig is, beschrijf je de geldigheid van een toestand in private klasse invarianten (= representation invarianten). Dit doe je boven de velden m.b.v. @invar.
- Documentatie van constructoren en methoden:
  - Als niet alle mogelijke waarden van een argument van een constructor of een methode geldige waarden zijn, moet je dit defensief of contractueel duidelijk maken.
    - Contractueel: Implementeer de methode onder de assumptie dat de argumenten geldig zijn, maar controleer wel of ze voldoen aan de opgelegde voorwaarden m.b.v. precondities boven de methode of constructor.
    - Defensief: Schrijf code in het begin van de body om the controleren of de input argumenten geldig zijn. Zo niet, gooi een IllegalArgumentException. Zet ook 1 of meerdere @throws clausules in de Javadoc van de constructor of methode om te specifiëren onder welke voorwaarden zulke exceptions gegooid zullen worden.
    - Defensief programmeren is veiliger dan contractueel omdat het errors in 1 module weerhoud om problemen te veroorzaken in een andere module. Errors worden ook eerder opgemerkt.
  - Het resultaat en neveneffecten van een methode specifiëren doe je m.b.v. postcondities die zeggen welke voorwaarden moeten gelden nadat de methode uitgevoerd is. Als de methode of constructor een object muteert, moet je de volledige

nieuwe abstracte toestand van het object specifiëren (wat er veranderd is, maar dus ook wat hetzelfde is gebleven).
    - Je moet altijd een constructor toevoegen aan een klasse anders maakt Java er automatisch één die je onmogelijk kan documenteren.
- Toegelaten uitdrukking in formele Javadoc:
    - Elke boolean uitdrukking zonder neveneffect, dus ook methodes (en constructoren) van de klasse die gedocumenteerd wordt.
    - In Javadoc voor een publieke methode mag je niet refereren naar een niet-publieke klasse, methode of constructor in dezelfde package.
    - Evaluatie van formele documentatie mag nooit crashen.
- @Mutates clausules:
    - Als een object niet in de @mutates clausule van een methode vermeld is en het object wordt niet gebruikt om de toestand van een ander object, dat wel vermeld is in @mutates, te representeren, dan mag de methode het object niet aanpassen.
    - Constructoren moeten niet @mutates this vermelden aangezien this nog niet bestaat voor het oproepen.
- @Inspects clausules:
    - Zelfde principe als @mutates, maar nu met het inspecteren van een object i.p.v. het aanpassen.
- Defaults van @mutates en @inspects:
    - Als er geen @mutates of @inspects vermeld is bij een methode of constructor dan mag deze elk object inspecteren en elk object aanpassen dat niet immutable is.
        - Uitzondering: als een methode start met get of is, dan mag deze by default enkel this inspecteren en niets muteren.
    - Constructor mag this muteren.
- @Creates clausules:
    - @creates | O betekent:
        - O is null of een immutable object of
        - O was aangemaakt tijdens het uitvoeren van de methode en O verschilt van alle objecten direct vermeld in @inspects of @mutates clausules en ook van alle representation objects van deze objecten.

# 3 Inheritance

## 3.1 Polymorphismen

- Uitleg m.b.v. voorbeeld: Shapes Circle en Polygon
    - Circle en polygon zijn beide shapes dus Shape is de superklasse van Circle en Polygon
    - → Circle extends Shape, idem Polygon
    - Circle en Polygon zijn subklasses van Shape en erven over van Shape
    - Shape is een abstracte klasse: het is niet mogelijk om een instantie van Shape aan te maken.
    - Shape is polymorphic: het kan refereren naar een instantie van Circle of naar een instantie van Polygon.
- Static type checking: Java controleert voor het runnen van een programma of het programma niet probeert om een methode M op te roepen op een object wiens klasse geen methode M definiëert en dat het niet probeert om een veld F van een object op te vragen wiens klasse geen veld F definiëert. Java doet dit door een static type aan elke uitdrukking van het programma toe te kennen:
    - De static type van een variabele is het gedeclareerde type van de variabele
        - Bv. static type van shape1 in de lijn *Shape shape1* is Shape
    - Static type van een referentie naar een veld is het gedeclareerde type van dat veld, voor een methode is dat het return type van de methode.

- Java's static type checker is incompleet: laat niet toe om sommige programma's te runnen die in geen enkel geval zouden crashen.
  bv.
  ```
  Shape myShape = new Circle(5, 10, 5);
  Circle myCircle = myShape;
  int radius = myCircle.getRadius();
  ```
  omdat myCircle Shape heeft als static type en Shape heeft geen methode getRadius().

- Typecasts: incompleetheid van de Type checker oplossen door (Circle) toe te voegen voor myShape waardoor de static type van myShape nu Circle is:
  ```
  Shape myShape = new Circle(5, 10, 5);
  Circle myCircle = (Circle)myShape;
  int radius = myCircle.getRadius();
  ```
- Pattern matching: "shape instanceof Circle circle" kijkt eerst of shape een instantie is van Circle, zo niet returnt het false, zo wel returnt het true en doet Circle circle = (Circle)shape;
- De "Object" klasse: = built-in klasse, die automatisch de (directe of indirecte) superklasse is van elke klasse in een Java programma.
  - Java's primitieve types: boolean, byte, short, char, int, long, float en double zijn geen klasses en hun waarden zijn dus geen Objects.
    - Toch werkt: *Object[] values = {true, 42, "A", 3.14}* want Java doet automatisch *Object[] values = {Boolean.valueOf(true), Integer.valueOf(42), ...}* waar *Boolean.valueOf(true)* een instantie van klasse Boolean retourneert met waarde *true*. Idem concept andere waardes. Deze klassen (Boolean, Integer, …) worden wrapper classes genoemd en deze feature wordt autoboxing genoemd.

## 3.2 Dynamic binding

Als beide Circle en Polygon een methode .toSVG() hebben, op verschillende manieren geïmplementeerd, en je wil shape.toSVG() oproepen zonder op voorhand te weten van welke Shape shape een instantie is (Circle of Polygon), moet je eerst een abstracte methode toSVG() toevoegen aan de Shape klasse. Hierdoor zal de computer shape.toSVG() uitvoeren op basis van het dynamische type van het receiving object. Als shape dus refereert naar een instantie van Circle zal toSVG() in de Circle klasse uitgevoerd worden en wanneer shape refereert naar een instantie van Polygon zal toSVG() in de Polygon klasse uitgevoerd worden. (Dit noemt men het dynamisch binden van methode oproepen.)
De toSVG() methodes in Circle en Polygon overschrijven dan de methode met dezelfde naam en zelfde aantal en types van parameters van hun superklasse.

- Methodes: equals(), hashCode() en toString():
  - equals(): checkt of 2 objecten hetzelfde voorstellen (bv. 2 punten zijn hetzelfde als ze dezelfde coördinaten hebben)
    Standaardimplementatie: return other == this
    - arrays erven deze equals() methode dus ook over en array1.equals(array2) is dus equivalent met array1 == array2 wat enkel hun identiteit vergelijkt en niet hun inhoud. Om inhoud te vergelijken, gebruik: Arrays.equals(array1, array2)
  - hashCode(): returnt een int die gebruikt kan worden als hash code als key voor het object in een hash table.
    Standaardimplementatie: returnt een int gebaseerd op de identiteit van het object. Verschillende objecten hebben (niet altijd) een verschillende hashCode
  - toString(): returnt een textuele representatie van het object
  - Deze 3 methodes kan je overschrijven.
- Voeg @Override toe aan de Javadoc van een methode van een klasse die een methode van zijn superklasse overschrijft.
- Record classes: een klasse gedefinieerd door "public record Point(int x, int y) {} is klasse met:
  - Private final veld voor elke parameter met dezelfde naam en type → record classes zijn immutable (hier *private final int* x en *private final int y*)
  - Public getters voor elke parameter, met zelfde naam en type (1)hier: *.x()* en *.y()*

  - Een constructor met dezelfde zichtbaarheid als de klasse zelf (hier *public*)
  - Een equals() methode: returnt *true* als *other instanceof record* class (hier record Point) en de componenten van other en this *equal* zijn.
  - hashCode() methode: returnt int berekend a.d.h.v. waarden (als componenten primitief) of hash codes (als componenten objecten zijn) van de componenten.
  - toString() methode: returnt naam van record class + naam van stringvoorstellingen van de componenten.
  - Enkele restricties:
    - Ze zijn final. Een finale klasse kan niet extended worden door andere klassen.

## 3.3 Behavioral subtyping: modular reasoning about programs that use dynamic binding

- Niet-modulair redeneren: redeneren over het volledige programma, een programma is correct of niet, een methode heeft geen notie van correctheid in deze vorm van redeneren.
- Modulair redeneren: Door specificatie toe te voegen aan elke methode, heeft een methode een correcte werking. Een programma is correct als elke methode overeenkomt met zijn specificatie.
- Method call resolution: de Java compiler checkt voor elke methode oproep of er een corresponderende methode bestaat in het programma en zoekt naar deze methode door naar de static type van de receiver te kijken. De methode die zo gevonden wordt is de resolved method.
  - Statisch gebonden oproep: at run time is de opgeroepen methode exact de resolved methode.
  - Dynamisch gebonden oproep: at run time is de opgeroepen methode oftewel de resolved methode of een methode die deze overschrijft.
- Modulair redeneren en dynamische binding:
  - Een methode is correct enkel en alleen als zijn gedrag wordt toegestaan door zijn eigen specificatie en door de specificaties van alle methoden die hij overschrijft, ervan uitgaande dat het gedrag van elke oproep die hij uitvoert voldoet aan de specificatie van de resolved methode van de oproep.
  - Als alle methoden van een programma correct zijn en de specificatie van de hoofdmethode van het programma het toegestane gedrag van het programma als geheel uitdrukt, dan volgt daaruit de correctheid van het programma als geheel.
- Versterken van specificatie:
  - Elke methode moet overeenstemmen met zijn eigen specificatie en elke specificatie van een overschrijvende methode moet de specificatie van de overschreven methode versterken.
    - Specificatie S versterkt S' als
      - @pre van S verzwakt @pre van S'
      - @post van S versterkt @ post van S'
- Behavioral subtyping: Als we een specificatie toekennen aan elke methode van een klasse C, dan definiëren we daarmee een gedragstype. We zeggen dat een object O van het gedragstype C is als, voor elke methode M van C, het gedrag van een oproep van M op O voldoet aan de specificatie van M in C.
  - Het gedragstype, gedefinieerd door een klasse, is enkel afhankelijk van de documentatie en niet van implementatie.
  - Een gedragstype D is een behavioral subtype van gedragstype C als elk object van gedragstype D, ook van gedragstype C is.
    - Als de specificatie van de methodes van D die methodes van C overschrijven de specificaties versterken, dan is D een behavioral subtype van C.
  - Een programma respecteert behavioral subtyping als voor elke klasse D extends C geldt dat D een behavioral subtype is van C
  - Met behulp van deze definities volgt dat een programma correct is wanneer:

    - Elke methode van een programma voldoet aan zijn specificatie, er vanuit gaand dat elk object waarmee het interageert van het gedragstype is dat gegeven wordt door zijn statische type,
    - en het programma behavioral subtyping respecteert,
    - en de specificatie van de hoofdmethode de correctheid van het hele programma beschrijft.
- Overerven van specificatie: In dit vak enkel als een methode M helemaal geen specificatieclausules bevat (d.w.z. geen @pre, @post, @inspects, …), en M een methode M' overschrijft, dan definiëren we de specificatie van M = specificatie van M'.

## 3.4 Interfaces

- Singleton Pattern: Het is niet logisch om verschillende instanties van BooleanType te hebben, dus in plaats van een constructor te definiëren, heeft de klasse een statisch veld INSTANCE dat refereert naar de enigste instantie van BooleanType dat ooit zal bestaan.

```
public class BooleanType extends Type {
    private BooleanType() {}
    public static final BooleanType INSTANCE = new BooleanType();
```

  Een statisch veld verschilt van een normaal veld omdat het een eigenschap van de klasse zelf is en niet van een instantie van de klasse.

Java laat een klasse niet toe om over te erven van meerdere superklassen (=multiple inheritance). Maar, een klasse mag wel extenden van 1 superklasse en 0 of meer interfaces. Een interface, verschilt van een klasse omdat het geen non-static velden kan declareren en ook geen constructoren. Voor de rest is het hetzelfde als een klasse.

- Interfaces zijn altijd abstract (by default)
- Methodes van interfaces zijn altijd public en abstract (by default)
- Een klasse kan meerdere interfaces implementeren, als de klasse niet abstract is, moet hij alle methodes van de interface implementeren (overschrijven)
- Instanceof en typecasting werkt hetzelfde als bij (super)klassen

## 3.5 Implementation inheritance

- Een subklasse erft de instance fields van zijn superklasse over
- Een constructor van een subklasse MOET EERST de velden van de superklasse initialiseren. De superklasse constructor moet dus opgeroepen worden m.b.v. super(…).
- Een methode van de superklasse kan opgeroepen worden m.b.v. super.M()
- Alle methodes van de superklasse worden overgeërfd
- Velden zijn verborgen NIET overschreven: wanneer een klasse D een veld declareert met dezelfde naam F als het veld dat D overerfde van zijn superklasse C, dan is het overgeërfde veld verborgen, niet overschreven. D heeft dus twee velden met naam F, D.F roept het veld van D op, C.F en D.super.F dat van C.

## 3.6 Closed types

In het algemeen zijn klassen en interfaces open-ended: een klant kan aan de ene kant willekeurig veel instanties van de klasse aanmaken en aan de andere kant kan de klant een open-ended aantal van directe subtypes (subklassen/implementaties van interface) aanmaken. Soms is het echter handig om een closed type te hebben, om oftewel een klant niet toe te laten om nieuwe instanties aan te maken, of om de klant niet toe te laten om nieuwe subtypes te definiëren, of beide.

- Types met een gesloten aantal instanties:
  - Bv. score, wiens instanties de mogelijke scores voorstellen:

```
public class Score {
    public static final Score LOVE = new Score(0, "LOVE", 0);
    public static final Score FIFTEEN = new Score(1, "FIFTEEN", 15);
    public static final Score THIRTY = new Score(2, "THIRTY", 30);
    public static final Score FORTY = new Score(3, "FORTY", 40) {
        @Override
        public Score next() { throw new UnsupportedOperationException(); }
    };
    private static final Score[] values = {LOVE, FIFTEEN, THIRTY, FORTY};

    public static Score[] values() { return values.clone(); }

    private final int ordinal;
    private final String name;
    private final int value;

    public int ordinal() { return ordinal; }
    public String name() { return name; }
    public int value() { return value; }
    public Score next() { return values[ordinal + 1]; }

    private Score(int ordinal, String name, int value) {
        this.ordinal = ordinal;
        this.name = name;
        this.value = value;
```

    De constructor is private zodat de klant geen nieuwe instanties kan aanmaken.
  - In Java kan dit makkelijker m.b.v. Enum classes:

```
public enum Score {
    LOVE(0),
    FIFTEEN(15),
    THIRTY(30),
    FORTY(40) {
        @Override
        public Score next() { throw new UnsupportedOperationException(); }
    };

    private final int value;

    public int value() { return value; }
    public Score next() { return values()[ordinal() + 1]; }

    private Score(int value) { this.value = value; }
```

  - Switch statements en switch expressions om gevalsanalyse te doen bij een instantie van een enum klasse:

```
public String getScoreInFrench(Score score) {
    switch (score) {
        case LOVE -> { return "zéro"; }
        case FIFTEEN -> { return "quinze"; }
        case THIRTY -> { return "trente"; }
        default -> { return "quarante"; }
```

```
public String getScoreInFrench(Score score) {
    return switch (score) {
        case LOVE -> "zéro";
        case FIFTEEN -> "quinze";
        case THIRTY -> "trente";
        case FORTY -> "quarante";
```

- Types met een gesloten aantal directe subtypes:
  - Door sealed toe te voegen laat je niet toe om subtypes aan te maken die niet vermeld zijn na de permits clausule.

```
public sealed interface GameState
    permits GameState.Regular, GameState.Advantage, GameState.Won
{ /* ... */ }
```

  - Switchen over een sealed type:

```
public String toString(GameState state) {
    return switch (state) {
        case GameState.Regular(var serverScore, var receiverScore) ->
                serverScore.value() + "-" + receiverScore.value();
        case GameState.Advantage(var server) ->
                "advantage " + (server ? "server" : "receiver");
        case GameState.Won(var server) ->
                "won by the " + (server ? "server" : "receiver");
    };
}
```

## 3.7 Lists, sets and maps

In dit hoofdstuk bekijken we 3 belangrijke voorbeelden van het gebruik van overereving om te generalizeren over verschillende implementaties van een abstract datatype. Deze voorbeelden tonen hoe dat éénzelfde API op verschillende manieren geïmplementeerd kunnen worden.

- Lists: om een sequentie van waardes op te slaan, hiervoor kunnen we ook een array gebruiken. Maar, het is niet mogelijk om waardes toe te voegen aan of te verwijderen uit arrays. Hierom definiëren we dus een list abstractie die dit toelaat. We doen dit op 2 manieren, met elk hun eigen voordelen:
  - ArrayList: waarden opslagen in een array
    - Voordelen:
      - Element opzoeken op bepaalde index in constante tijd
      - Element aan het einde toevoegen in constante tijd
      - Laatste element verwijderen in constante tijd
    - Nadelen:
      - Element toevoegen aan het begin of het eerste element verwijderen neemt tijd in beslag die evenredig is met de lengte van de lijst.
  - LinkedList: waarden opslagen in een dubbel gelinkte lijst structuur
    - Voordelen:
      - Element aan het begin of einde toevoegen of verwijderen kan in constante tijd.
    - Nadelen:
      - Het element op bepaalde index opzoeken kost tijd evenredig met de lengte van de lijst.
- Sets: Als het in een applicatie belangrijker is om te checken of een sequentie een bepaalde waarde bevat of om een bepaalde waarde te verwijderen, is een hash table beter dan Lists. Als er dus een goede hash function gebruikt is, dan gebeurt het checken van voorkomen van een element en het toevoegen van of verwijderen van een element allemaal in constante tijd.
- Maps: Applicaties moeten vaak een set van key-value paren opslaan (waarbij verschillende values verschillende keys hebben) en efficiënt een paar kunnen toevoegen en de value kunnen ophalen die bij een bepaalde key hoort. In Java worden dergelijke verzamelingen maps genoemd en een key-value paar wordt een map entry genoemd.

LEES ZEKER OOK HET DEEL VAN DE CURSUS, HIERIN STAAT VEEL MEER INFO EN CODE

# 4 Multi-object abstractions (= entity-relationship abstractions)

Twee types van data abstracties:

- Single-object abstracties: elke instantie van een klasse representeert (immutable) of slaat (mutable) een abstracte waarde op.
- Multi-object abstracties: groepen van objecten van dezelfde klasse (single-class multi-object abstractions) of van verschillende klassen (multi-class multi-object abstractions) worden samen gebruikt om entiteitsgrafen op te slaan.

## 4.1 Single-class entity-relationship abstractions

- Algemeen principe: Definiëer een Java klasse voor elk type entiteit, een getter in klasse E voor elke attribuut van entiteit type E en getters in klasse E en F voor elke relatie tussen entiteiten van type E en F.
  Om vervolgens een entiteitsgraaf op te slaan, maken we een aparte instantie van klasse E aan voor elke entiteit van type E in de graaf. Wanneer een relatie van de graaf entiteit e1 en e2 met elkaar linkt, slaan we in e2 een referentie op naar e1 en ook omgekeerd. (Ook slaan we in object e de waarden van de attributen op die toegekend zijn aan entiteit e door de graaf.)
- Consistentie van bidirectionele associatie: als een object e1 een relatie heeft met e2 dan MOET e2 ook een relatie hebben met e1, wanneer 1 van de 2 deze relatie verwijderd, moet de andere dit dus ook doen.
  - Dit wordt gereflecteerd in de representatie invarianten van de objecten. Als bv. s1.teammate = s2, dan moeten de invarianten ook beschrijven dat de teammate van de teammate van s1, s1 zelf is (in dit vb. kan een persoon dus maar 1 teammate hebben).
- Peer groups:
  - Om te vermijden dat we deze invarianten breken moeten we regels opstellen:
    - Een representation invariant van een object X mag enkel een niet-final veld of niet-immutable eigenschap van een object Y vermelden als:
      - X = Y, of
      - Y is een representation object van X, of
      - Y is een peer object van X.
    - Als X een peer object is van Y, dan is Y een peer object van X
  - Een peer group is een set van objecten die elk direct of indirect peer objects zijn van elkaar. Deze objecten vormen samen een entity-relationship abstraction. Een enkel object van een peer group kan niet beschouwd worden als een aparte abstractie; de noties van geldigheid van representatie en abstracte toestand zijn alleen zinvol op de peer group als geheel.
  - Voeg @peerObject toe voor velden die verwijzen naar peer objects en voor getters met peer objects als result.
  - Als @mutates | *this* vermeld is bij een object van een peer group, betekent dit dat elk object van de peer group gemuteerd kan worden.

## 4.2Multi-class entity-relationship abstractions


- Probleem:

```java
package bigteams;

public class ProjectCourseStudent {

    private Team team;

    public Team getTeam() { return team; }

    public ProjectCourseStudent() {}

    public void join(Team team) {
        if (this.team != null)
            throw new IllegalStateException("this student is already in a team");

        this.team = team;
        team.members.add(this);
    }

    public void leaveTeam() {
        if (this.team == null)
            throw new IllegalStateException("this student is not in a team");

        team.members.remove(this);
        team = null;
    }
}
```

  Student probeert een privaat veld, members, van zijn team te bekijken, dit gaat uiteraard niet. We kunnen members niet publiek maken want dit zou encapsulatie breken. Echter kunnen we wel Student en Team in dezelfde package plaatsen en members package-accesible maken. Een veld is by default package-accesible als er geen accesibility modifier zoals *private* of *public* voor vermeld staat.
- team is @peerObject van Student
- members zijn @peerObjects van Team
- Voeg ook @peerObject(s) toe voor de getters getTeam() en getMembers()
- De representation invariant *team == null || team.members.contains(this)* van Student is niet volledig als *team.members == null*. In klasse Team is er een representation invariant *members != null*. Deze invariant kunnen we gebruiken om de invariant van Student te beschermen door de volgore van het checken van representation invariants te definiëren: de Fase 1 representation invariant van elk peer object wordt gecontroleerd, de Fase 2 invariant wordt gecontroleerd, en zo verder tot alle representation invarianten gecontroleerd zijn. Fase 1 invariant = de eerste representation invariant die gedefinieerd is in de klasse, en zo verder. Hierom voegen we een dummy representation invariant *true* toe in Student, zodat de invariant die steunt op het niet null zijn van team.members een Fase 2 invariant is.
- @mutates  properties | this.getTeam(), team.getMembers() voor joinTeam(team) in Student zegt dat joinTeam enkel en alleen this.getTeam() en team.getMembers() muteert en alle andere eigenschappen van beide peer groups onveranderd blijven.

**Advanced: Nesting class-encapsulated and package-encapsulated abstractions**

## 4.3 How to properly document multi-object abstractions

Deze instructies complementeren de instructies van sectie 2.4.

- Documentatie van velden:
  - Als een @invar clausule van een object O een veld van een object O' vermeld, dan moet O' een representation of peer object zijn van O.
  - Een @invar clausule moet goed gedefinieerd zijn, mag dus niet crashen en niet oneindig loopen, voor willekeurige (concrete) toestanden van de betrokken objecten, behalve bij de volgende gevallen:
    - Een @invar clausule wordt enkel geëvalueerd als de voorgaande @invar clausules *true* gaven. Bv. als een eerste clausule zegt dat een veld niet *null* is dan mag de volgende een methode van het object van dat veld oproepen.
    - De N'de @invar clausule van een object O dat een lid is van een peer group wordt enkel geëvalueerd als, voor elk lid O' van dezelfde groep, en elke I < N, de I'de @invar clausule van O' *true* gaf.
- **Advanced topics:**
  - @mutates: Een methode mag enkel en alleen een object O muteren als:
    - O in de methode zelf aangemaakt wordt
    - Een lid van O's peer group vermeld is in de @mutates clausule
    - O een lid is van de peer group van een representation object van een lid van de peer group van een object dat vermeld is in de @mutates clausule
    - O een lid is van een peer group van een representation object van een lid van de peer group van een representation object van een lid van de peer group van een object vermeld in de @mutates clausule
    - Enzovoort
  - Een @mutates_properties clausule kan gebruikt worden als de resultaten van de meeste getters van objecten in een peer group onveranderd blijven na de uitvoer van de methode. Het vervangt @mutates en 1 @post (die beschrijft dat andere, niet vermelde objecten onveranderd blijven).
  - @inspects: Een methode mag nooit de toestand van een al-bestaand mutable object inspecteren dat geen lid is van een object vermeld in de @mutates of @inspects clausules, en dat ook niet gebruikt wordt om een toestand van één van deze vermelde objecten te beschrijven.
  - Defaults van @mutates en @inspects: Als er geen @mutates of @inspects vermeld is bij een methode of constructor dan mag deze elk object inspecteren en elk object aanpassen dat niet immutable is.
    - Uitzondering: als een methode start met get of is, dan mag deze by default (zonder vermelden van clausules) this inspecteren, maar geen enkel object muteren.
  - @creates: Als een object O vermeld is in de @creates clausule van een methode dan betekent dit dat elk lid van O's peer group aangemaakt wordt in deze methode en dat deze peer group niet verbonden is aan een direct of indirect representation object van een peer group vermeld in de @inspects of @mutates clausules
  - Evaluaties van clausules: @mutates, @mutates_properties en @inspects worden voor de uitvoering van de methode of constructor geëvalueerd en betekenen dus dat ze de "old" waardes van de vermelde objecten mogen aanpassen/inspecteren.
    - Het is dus niet logisch om *@mutates | this* of *@mutates_properties | this* te vermelden bij een constructor.
  - Specifying collections of objects: Als een methode een hele collectie (bv. list) van objecten inspecteert of muteert, dan kan dit in de clausules gespecifiëerd worden m.b.v. de *…collection* syntax, bv.:

```
 * @inspects | lists, strings
 * @mutates | ...lists
 */
static void allAddAll(StringList[] lists, String[] strings) {
    for (StringList list : lists)
        list.addAll(strings);
```

# 5 Advanced topics

## 5.1 Iterators

Een iterator heeft methodes om:

- Te testen of de iterator het einde van de datastructuur bereikt heeft (hasNext())
- Het huidige element te verkrijgen (next())
- De iterator te muteren zodat het naar het volgende element verwijst (next())

**Externe Iteratoren:**

We kunnen dus een externe iterator voor een ArrayList implementeren als volgt:

```
public interface Iterator {
    boolean hasNext();
    /**
     * Mutates the iterator to point to the next element and returns the current
     * element.
     */
    Object next();
```

```
public class ArrayListIterator implements Iterator {

    public ArrayList arrayList;
    public int index;

    public ArrayListIterator(ArrayList arrayList) {
        this.arrayList = arrayList;
    }

    public boolean hasNext() { return index < arrayList.elements.length; }

    public Object next() { return arrayList.elements[index++]; }

}
```

We kunnen het ook anders doen a.d.h.v. Iterables; een interface dat geïmplementeerd moet worden door elke collectie dat iteratie ondersteund.

```
public interface Iterable {
    /** Returns a new iterat
    Iterator iterator();
}
```

```
public class ArrayList implements Iterable {

    public Object[] elements;

    public Iterator iterator() { return new ArrayListIterator(this); }

}
```

Omdat ArrayListIterator en ArrayList geen onafhankelijke objecten zijn, is het logischer om ArrayListIterator te definiëren als een geneste klasse in de klasse ArayList:

```
public class ArrayList implements Iterable {

    private Object[] elements;

    private static class IteratorImpl implements Iterator {

        private ArrayList arrayList;
        private int index;

        private IteratorImpl(ArrayList arrayList) { this.arrayList = arrayList; }

        public boolean hasNext() { return index < arrayList.elements.length; }

        public Object next() { return arrayList.elements[index++]; }

    }

    public Iterator iterator() { return new IteratorImpl(this); }

    // Constructors and mutators not shown
```

Als we van de Iterator nu een inner class maken, die dus non-static is, kan het aan de velden van zijn Enclosing Class zonder dat deze expliciet in een veld opgeslagen moeten worden. Ook de constructor wordt impliciet gegenereerd en kan dus weggelaten worden:

```
public class ArrayList implements Iterable {

    private Object[] elements;

    private class IteratorImpl implements Iterator {

        private int index;

        public boolean hasNext() { return index < elements.length; }

        public Object next() { return elements[index++]; }

    }

    public Iterator iterator() { return new IteratorImpl(); }

    // Constructors and mutators not shown
```

Aangezien IteratorImpl enkel vermeld wordt in de iterator() methode, kunnen we dit nog simpeler maken door het als een lokale klasse te definiëren in de iterator() methode:

```
public class ArrayList implements Iterable {

    private Object[] elements;

    public Iterator iterator() {

        class IteratorImpl implements Iterator {

            private int index;

            public boolean hasNext() { return index < elements.length; }

            public Object next() { return elements[index++]; }

        }

        return new IteratorImpl();
    }

    // Constructors and mutators not shown
```

(Opmerking: lokale klassen zijn niet zichtbaar buiten de omsluitende methode, dus het zou geen zin hebben om het private keywoord toe te voegen, dit is in feite zelfs niet toegestaan.)

De simpelste manier om een externe Iterator te definiëren (en eigenlijk ook de enigste die je hoeft te onthouden) is via een anonieme klasse:

```
public class ArrayList implements Iterable {

    private Object[] elements;

    public Iterator iterator() {

        return new Iterator() {

            private int index;

            public boolean hasNext() { return index < elements.length; }

            public Object next() { return elements[index++]; }

        };
```

In al deze bovenstaande gevallen kan de klant de arrayList doorlopen als volgt (iterable is hier de ArrayList):

```
public void printAll(Iterable iterable) {
    for (Iterator iterator = iterable.iterator(); iterator.hasNext(); )
        System.out.println(iterator.next());
}
```

**Interne Iteratoren:**

De klant geeft een Consumer object aan de collectie. De collectie geeft elk element van de verzameling op zijn beurt mee als argument aan de accept methode van de consumer:

```
public interface Iterable {
    void forEach(Consumer consumer);
}
```

```
public interface Consumer {
    void accept(Object value);
}
```

Voor een ArrayList is de implementatie dan als volgt:

```
public class ArrayList implements Iterable {

    public Object[] elements;

    public void forEach(Consumer consumer) {
        for (int i = 0; i < elements.length; i++)
            consumer.accept(elements[i]);
    }
```

In het algemeen zal de for loop er iets anders uitzien:

```
for (Iterator i = iterator(); i.hasNext(); )
    consumer.accept(i.next());
```

De klant kan deze interne operator dan als volgt gebruiken:

```
public void printAll(Iterable iterable) {
    iterable.forEach(new Consumer() {

        public void accept(Object value) {
            System.out.println(value);
        }
```

Of via een lambda uitdrukking:

```
public void printAll(Iterable iterable) {
    iterable.forEach((Object value) -> { System.out.println(value); });
}
```

Of zelfs

```
iterable.forEach(value -> System.out.println(value));
```

## 5.2 Streams

https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/stream/package-summary.html

## 5.3 Generics

Stel bv. in klasse University wil je een LinkedList van Students en een LinkedList van StaffMembers bijhouden. Je zou dus 2 LinkedList klassen kunnen definiëren: LinkedListOfStudent en LinkedListOfStaff, of een algemene LinkedList klasse definiëren die alle objecten van type Object kan bevatten. Beide opties hebben hun nadelen, een beter optie is Generic types gebruiken: definieer een klasse LinkedList<T> die dus enkel objecten van klasse T kan bevatten.
We kunnen dus een klasse equivalent aan LinkedListOfStudent verkrijgen door een generische klasse LinkedList te instantiëren met het argument type Student. Zo verkrijgen we een geparametriseerd type LinkedList<Student>.
De velden van de University klasse zien er dan zo uit:

```
class University {

    private LinkedList<Student> students = new LinkedList<Student>();
    private LinkedList<Staff> staffMembers = new LinkedList<Staff>();
```

Opmerking: bij *new LinkedList<Student>()* is het door de context al duidelijk welk argumenttype er nodig is om de LinkedList te instantiëren dus kan deze weggelaten worden: *LinkedList<Student>* wordt dan *new LinkedList<>()*, dit wordt de diamond notatie genoemd.

**Bounded type parameters:**

Stel dat we studenten met elkaar willen vergelijken, bv. op basis van hun aantal studiepunten,

```
interface Comparable<T> {

    /**
     * Returns a negative number, zero, or a positive number if this objec
     * compares as less than, equal to, or greater than {@code other}.
     */
    int compareTo(T other);

}
```

en we willen ze op basis hiervan sorteren in de LinkedList. Dit kan door een klasse SortedLinkedList aan te maken:

```
class SortedLinkedList<T extends Comparable<T>> extends LinkedList<T> {

    @Override
    void addFirst(T element) {
        Node<T> sentinel = new Node<T>(null, firstNode);
        Node<T> node = sentinel;
        while (node.next != null && node.next.element.compareTo(element) < 0)
            node = node.next;
        node.next = new Node<T>(element, node.next);
        firstNode = sentinel.next;
    }
```

Door de upper bound *extends Comparable<T>* toe te voegen aan het parametertype T, laat de static type checker toe dat we element.compareTo() oproepen. Dit mag wel enkel als T werkelijk een subklasse is van Comparable<T> dus moeten we nog Student Comparable<Student> laten implementen:

```
class Student implements Comparable<Student> {

    int nbCredits;

    public int compareTo(Student other) { return nbCredits - other.nbCredits; }

}
```

**Generische types zijn invariant:**

Definiëer:

```
class Member {}

class Student extends Member implements Comparable<Student> { ... }

class Staff extends Member implements Comparable<Staff> { ... }
```

- LinkedList<T> is niet covariant in T: LinkedList<Student> is geen subtype van LinkedList<Member.
- LinkedList<T> is niet contravariant in T: LinkedList<Member> is geen subtype van LinkedList<Student> (of LinkedList<Staff>.

→ LinkedList<T> (en in het algemeen generische types) zijn invariant.

**Wildcards**

Omwille van invariantie kunnen we dus geen LinkedList<Student> meegeven aan LinkedList<Member>.addAll()

```
class LinkedList<T> implements Iterable<T> {

   // ...

   void addAll(LinkedList<T> other) {
       for (Iterator<T> iterator = other.iterator(); iterator.hasNext(); )
              addFirst(iterator.next());
   }
```

addAll() mag echter wel een LinkedList<U> als argument opnemen als U een subtype is van T, als we dit uitdrukken m.b.v. upper-bounded wildcards:

```
void addAll(LinkedList<? extends T> other) {
    for (Iterator<? extends T> iterator = other.iterator(); iterator.hasNext(); )
            addFirst(iterator.next());
```

Weeral omwille van invariantie kan er geen LinkedList<Member> als argument meegegeven worden aan LinkedList<Student>.copyInto():

```
void copyInto(LinkedList<T> other) {
    for (Iterator<T> iterator = this.iterator(); iterator.hasNext(); )
            other.addFirst(iterator.next());
```

copyInto() mag echter wel een LinkedList<U> als argument opnemen als U een supertype is van T, als we dit uitdrukken m.b.v. lower-bounded wildcards:

```
void copyInto(LinkedList<? super T> other) {
    for (Iterator<T> iterator = this.iterator(); iterator.hasNext(); )
            other.addFirst(iterator.next());
}
```

We kunnen deze 2 ook combineren op verschillende equivalente manieren:

```
static <T> void copyInto(LinkedList<T> from, LinkedList<? super T> to) {
static <T> void copyInto2(LinkedList<? extends T> from, LinkedList<T> to) {
static <T> void copyInto3(LinkedList<? extends T> from, LinkedList<? super T> to) {
```

**Generics erasure in Java:**

Samenvatting door chatGPT:

1. **Erasure Process:**
   - **Type Parameters Replacement**: In generic type declarations, each type parameter is replaced by its upper bound. If there's no explicit upper bound, it's replaced by `Object`.
   - **Type Arguments Removal**: Type arguments are removed from the program.
   - **Insertion of Typecasts**: Typecasts are inserted where necessary to ensure the program remains well-typed after erasure.

2. **Example:**

- **Before Erasure (Approach 3):**

```java
List<String> list = new ArrayList<>();
Iterator<String> it = list.iterator();
String str = it.next();
```

- **After Erasure (Approach 2):**

```java
List list = new ArrayList();
Iterator it = list.iterator();
String str = (String) it.next();
```

**Implications of Erasure:**

1. **Type Argument Restrictions:**
   - **Subtypes of Object Only:** Type arguments must be reference types (subtypes of `Object`). Primitive types like `int` cannot be used directly.
   - **Autoboxing:** To store primitives in a generic collection, they must be boxed into their wrapper classes (e.g., `ArrayList<Integer>` instead of `ArrayList<int>`). Autoboxing converts between primitives and their corresponding wrapper objects.
2. **Run-time Type Information Limitations:**
   - **Unavailable Type Arguments:** Type arguments are not available at runtime. This means certain operations are not allowed:
     - `new T()`, `new T[]`, `T.class`, and `instanceof T` are not permissible if `T` is a type parameter.
   - **Allowed with Type Erasure:** Operations like `new LinkedList<T>()` are allowed because the type argument is erased.
3. **Unchecked Cast Warnings:**
   - **Casts to Non-Erasure Types:** Casting to types that differ from their erasure (e.g., `T` or `LinkedList<T>`) is allowed but generates unchecked cast warnings. These casts may not be fully checkable at runtime.
   - **Example and Potential Issue:**

```java
Object object = ...;
LinkedList<Student> studentList = (LinkedList<Student>) object;  // Generates an u
Student student = studentList.iterator().next();  // May throw ClassCastException
```

If `object` is actually a `LinkedList<Staff>`, the cast to `LinkedList<Student>` will compile with a warning but could fail at runtime when elements are accessed.