```java
int x = 6;

if (x < 7)
    if (x < 6)
        System.out.println("A");
else
    System.out.println("B");
```

“A” geen output “B”

Wat is de uitvoer van deze code?

```java
void printer(int[] p, int index) {

    if (index >= p.length)
        System.out.println(Arrays.toString(p));

    for (int b = 0; b < 2; b++) {
        p[index] = b;
        printer(p, index + 1);
    }
}

// aanroep
printer(new int[5], 0);
```

geen idee

ik denk
dat ik het weet

Wat doet deze methode?

Recursie

```java
void recursive(int i)
{
    if (...) { // base case
        ...
        return;
    }

    // recursive case
    ...
    recursive(i + 1);
}
```

Dit is hoe recursieve methoden er doorgaans uitzien in java. Er is een duidelijke splitsing in een base case, code die uitgevoerd wordt als de recursie ophoudt, en de recursive case, de code die bepaalde acties uitvoert en de method opnieuw aanroept, met een ander argument.

Zonder de base case zou de methode oneindig vaak worden aangeroepen.

De swap functie (pseudocode)

```
a = "a"
b = "b"

print a + " " + b + " "

swap(a, b)

print a + “ “ + b + “ “
```

resultaat

```
a b
b a
```

Hier hebben we een klassieke functie beschreven in pseudo-code (dwz. niet in een specifieke taal, maar in grote lijnen, zodat het idee duidelijk is).

De swap functie neemt twee input variabelen en verwisselt hun inhoud. Kunnen we deze functie in java implementeren?

```
void swap(int a, int b) {...}
```

```
void swap(Integer a, Integer b) {...}
```

- geen
- de eerste
- de tweede
- beide

Welke van deze functies kunnen we implementeren in Java?

```java
void swap(String[] a, String[] b) {...}
```

```java
void swap(Thing a, Thing b) {
    Thing temporary = new Thing();

    temporary.x = a.x;
    a.x = b.x;
    b.x = a.x;
}
```

```java
class Thing {

    public Object x;

}
```

Dit zijn de twee swap functies die we kunnen implementeren in Java. We kunnen de inhoud van een array swappen en de inhoud van een object. We kunnen niet de objecten zelf swappen. Als we dus een object hebben (zoals **`String`**), waarvan we de inhoud niet kunnen veranderen, kunnen we er ook geen swap functie voor schrijven.

Het is belangrijk om het principe hierachter goed te begrijpen. Op het moment dat een methode aangeroepen wordt, worden de argumenten van die methode gekopieerd en wordt de methode uitgevoerd op de kopieën. Bij primitives (**`int`**, **`double`**, etc.) kan de methode dus alleen de kopieën manipuleren.

Als de methode objecten als argumenten heeft geeft java ook kopieën door, maar niet kopieën van het hele object. De referentie naar het object op de heap wordt gekopieerd. Zowel de methode als de aanroeper werken dus met hetzelfde object, maar ze hebben ieder hun eigen referentie. Je kunt dus de inhoud van het object swappen, maar de referenties niet.

Waar zit de fout?

```
class Thing {
    int num() {
        return n;
    }

    static int n = 3;
}
```

```
class Thing {
    int n = 3;

    static int num() {
        n;
    }
}
```

```
class Thing {}

static print() {
    Thing t;
    t = new Thing();
    print(t);
}
```

Welke van deze drie stukken code compileert niet? We gebruiken print() hier even als afkorting voor System.out.println().

Waar zit de fout?

Een waarschuwing voor we verder gaan. De slides bevatten vaak lelijke code. De acces modifiers (public, private etc) ontbreken en variabele namen zijn te kort. Dit doen we omdat het anders moeilijk te lezen is op slides. Soms gebruiken we lelijke code om het verschil aan te geven tussen lelijke en illegale code.

Dit betekent dat je de code op de slides niet zonder meer over kunt nemen. Houd je aan de stijlconventies van Java en van het vak als je zelf code schrijft.

Hoe schrijf ik SIMPELE code?

We kennen nu de basisprincipes van Java: loops, ifs, methodes en variabelen. Met deze bouwstenen kun je alles maken wat er op een computer te maken valt (de taal is *Turing-compleet*). Maar het is niet altijd even gemakkelijk. Naarmate een project groeit wordt het steeds lastiger om in de gaten te houden wat alles betekent, waar alles staat, en hoe de structuren gebruikt moeten worden.

De structuren die we in de laatste 4 colleges uitleggen zijn er niet om meer dingen mogelijk te maken, maar om de complexiteit van grote softwareprojecten te lijf te gaan. De eerste van deze principes hebben jullie al gezien. Door data en code bij elkaar te bundelen in een object kun je je code organiseren en zorgen dat data op de juiste manier gemanipuleerd wordt.

the software crisis

Computers begonnen in de jaren 40 en 50 als peperdure en specialistische machines. Meestal werden ze afgeleverd met een vast programma, om één enkele taak uit te voeren. Naarmate hardware sneller en goedkoper werd, begonnen gebruikers computers te gebruiken voor meerdere taken, en begon de scheiding tussen software en hardware ook bij de gebruikers door te dringen: software werd gedeeld, aangepast en gedraaid op verschillende platformen.

De software crisis is de naam die gegeven werd aan het probleem dat al snel duidelijk werd: software is ontzettend complex. Met een set van twee instructies heb je al een Turing complete taal, die alles kan beschrijven wat je computer kan doen, maar om een ingewikkeld computerprogramma te snappen, en zeker te weten dat het doet wat je wilt is en andere kwestie.

bron foto's: **The 1968/69 NATO Software Engineering Reports**
`http://homepages.cs.ncl.ac.uk/brian.randell/NATO/NATOReports/index.html`

De laatste 4 colleges

hoe schrijf ik:

simpele code

- inheritance
- polymorphisme
- collections

veilige code

- generics
- exceptions
- mutability

bruikbare code

- API design
- commenting

Laatste college: vragen, extra uitleg, revisie, voorbereiding tentamen

In de laatste drie colleges bekijken we de ge-avanceerdere features van Java. Deze features zijn er met name op gericht je code simpeler, veiliger, en gebruiksvriendelijker te maken. We gebruiken deze drie thema's om de cursus af te sluiten.

Het laatste college gebruiken we om laatste onderwerpen aan te stippen, vragen te beantwoorden, het tentamen voor te bereiden en te kijken wat je als kersverse programmeur nog te leren hebt.

Een belangrijke vraag die we hopen te beantwoorden in deze laatste colleges is waarom we Java geven als eerste taal, in plaats van bijvoorbeeld Python of C, en waarom Java werkt zoals het werkt. Veel van de eigenschappen van Java die aanvankelijk vervelend lijken zijn met een goede reden zo ontworpen. Maar om dat te begrijpen, moet je eerst weten welke doelen een programmeertaal als Java heeft.

Objecten

fields ->

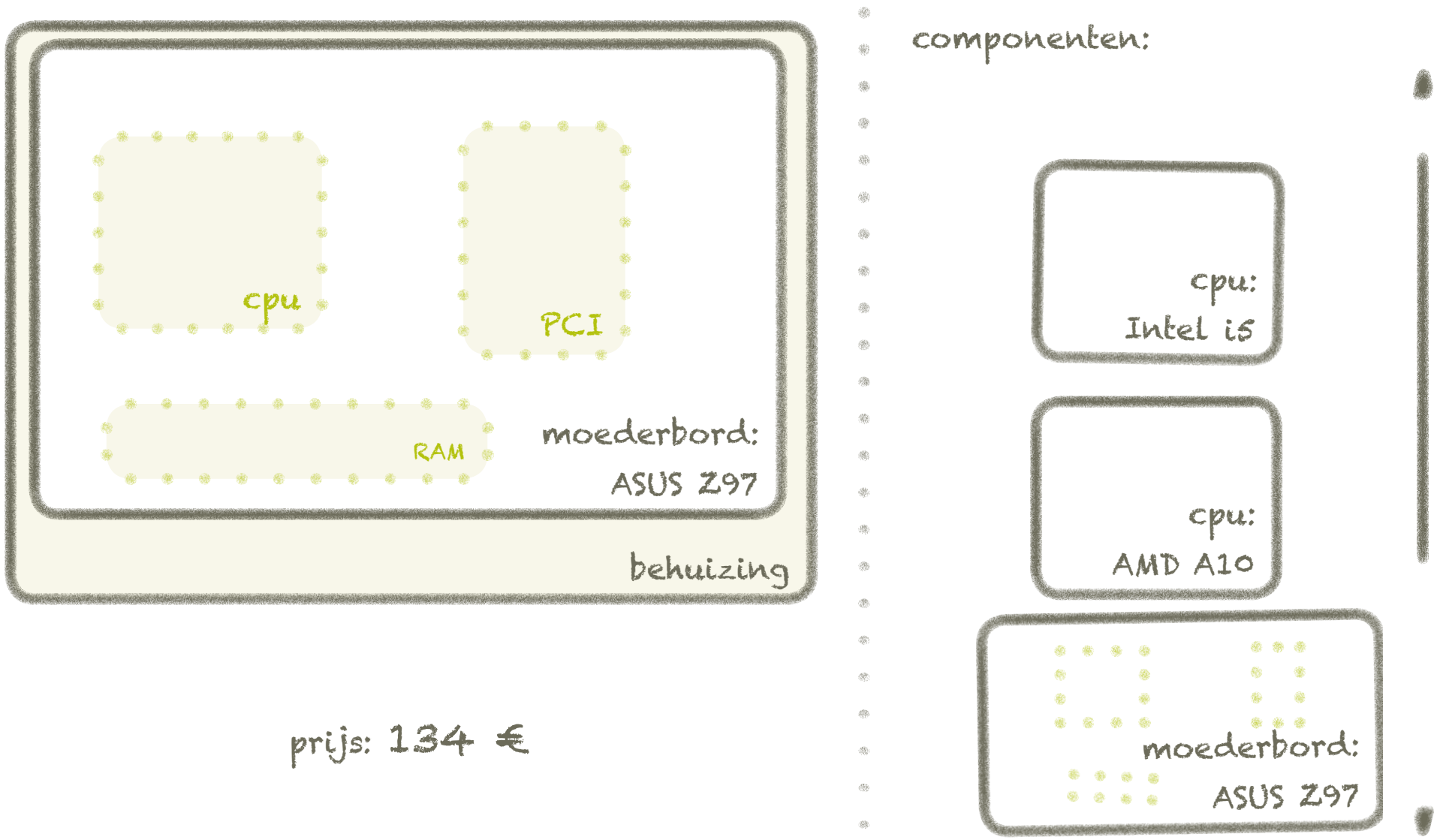
constructor ->

method ->

static field ->

```
public class Person {

    private int age;
    private Date birthDay;

    public Person(int age, Date bDay)
    {...}

    public void incrementAge() {
        age++;
    }

    public static final in MAX_AGE = 95;
}
```

Voorbeeld: Studentenopdracht



Om de onderwerpen van vandaag te illustreren, beginnen we met een voorbeeld. Twee studenten informatica krijgen de opdracht om een app te schrijven om het bouwen van een PC te simuleren. De app toont een lege behuizing, waarin de gebruiker componenten kan plaatsen. Er zijn verschillende beperking op de componenten (het moederbord moet de juiste socket hebben voor de CPU, etc.) De gebruiker sleept een component naar een open plek. Als de component past klinkt een belletje, als het niet past een toeter. In het laatste geval geeft de app ook een uitleg van wat er fout is.

De app toont ook de prijs van de tot nu toe samengestelde componenten.

Voorbeeld: Studentenopdracht

Student A

```
int[] choices = new int[4];

boolean isCorrect(int[] choices)
{...}

if (isCorrect(choices))
    playBell();
else
    playNoise();
```

Student B

```
class Component {...}
class CPU extends Component {...}
class MotherBoard extends Component
{...}

Component[] = new Component[4];

boolean isCorrect(int[] choices)
{...}

if (isCorrect(choices))
    playBell();
else
    playNoise();
```

Student A wil snel klaar zijn, en schrijft de app puur procedureel. Hij heeft geen zin om allemaal objecten te gaan maken, dus hij kent gewoon ieder object een integer toe, en houdt in een array bij welke componenten in de behuizing geplaatst zijn. Alle moederborden hebben drie onderdelen, dus een array van vier integers is altijd genoeg.

Student B gaat er rustiger voor zitten, en maakt voor iedere component een eigen klasse. Daarnaast maakt hij een klasse `Component` waar de andere klassen van *overerven*. We zien hier het eerste voordeel van overerving. Omdat alle componenten overerven van Component, kun je een array van Componenten aanmaken en daar zowel een CPU als een Moederbord in stoppen.

Hie zie je wat stukjes code van de beide richtingen.

Voorbeeld: Studentenopdracht

Opdracht 2: Nieuwe componenten, inclusief een videokaart met vervangbaar RAM en een moederbord met twee CPU-sockets.

```
int[] choices = new int[16];
int totalComponents;

int numSlots(int compID){...}
```

```
class Component {
    Component[] SubComponents;

    boolean add(Component c, int slot)
    {...}
    int numSlots() {...}
}

class Housing extends Component {}
```

Beide studenten leveren de opdracht succesvol in. De week daarop maakt de docent de tweede opdracht beschikbaar: breid de app uit met nieuwe componenten. Een videokaart met vervangbaar geheugen en een moederbord met twee CPU sockets.

Student A moet nu hard nadenken. Zijn array van vier element is niet genoeg meer. Hij bedenkt een ingenieuze oplossing: hij beschouwt de gebouwde computer als een boom, en slaat eerst voor een gegeven niveau alle componenten op en daarna de kinderen van die component, en zo voorts. Hij voegt de methode numSlots toe die voor een gegeven component teruggeeft hoeveel subcomponenten het heeft.

Student B moet ook veel aanpassen. Hij besluit de behuizing ook te zien als een component. Daarnaast slaat hij in ieder component de subcomponenten op. De boom van componenten zit zo vanzelf in de objecten opgeslagen. We zien hier het tweede voordeel van inheritance: Student B geeft Component een field en een methode. Alle klassen die Component extenden hebben deze methoden nu.

Voorbeeld: Studentenopdracht

Opdracht 3: Als je een Blu-ray speler succesvol toevoegt, speelt de app het geluid van een draaiende Blu-ray in plaats van een belletje.

```
if (isCorrect(choices))
{
    if (lastComponent == BLU_RAY)
        playDrive();
    else
        playBell();
} else
{
    playNoise();
}
```

```
class Component {
    Sound getSound() {
        // return bell sound
    }
}

class BluRay extends Component
{
    Sound getSound() {
        // return drive sound
    }
}
```

Student A was veel tijd kwijt met de vorige opdracht. Al zijn code moest aangepast, en hij moest allerlei nieuwe bugs zien te vinden. Student B was dit keer sneller klaar. Hij hoefde alleen de main methode aan te passen en dingen toe te voegen. De bestaande code, die hij al getest had bleef grotendeels onveranderd.

Opdracht 3 is om een ander geluidje te spelen voor een nieuw component: een Blu-ray speler.

Student A voegt een nieuwe methode toe en past zijn if-else statement aan. Hij moet nu het laatst toegevoegde component bijhouden en controleren of dat een Blu-ray speler was. Het werkt, maar de student hoopt dat niet iedere component straks zijn eigen geluidje krijgt, want het wordt er niet mooier op.

Student B besluit iedere component zelf de verantwoordelijkheid te geven over welk geluidje het speelt. Componenten die niks aangeven erven het belletje, en de BluRay definieert een nieuw geluidje. De BluRay klasse *override* de getSound methode.

Voorbeeld: Studentenopdracht

Opdracht 4: Maak een delete-knop in de app van een medestudent.

```
delete(int index, int[] choices)
{
    ...
}
```

```
class Component
{
    void delete() {
        // delete all subcomponents
        for (Component c : subComps)
            c.delete();
        // set slot in parent comp
        // to null
        ...
    }
}
```

De sadistische docent heeft weer iets nieuws bedacht. De studenten moeten nu elkaars code aanpassen. Er moet een klein knopje komen om een component te verwijderen.

De arme student C, die de code van student A aan moet passen is gedwongen om de structuur van het choices array te doorgronden. Hij moet niet alleen het component verwijderen (en de rest van de componenten opschuiven), hij moet ook de kinderen opzoeken en de kinderen daarvan. Als hij een fout maakt blijft hij over met een array waar niks van klopt en een CPU zomaar in het luchtledige kan zweven.

Student D krijgt de code van Student B en heeft het makkelijk. Hij voegt een recursieve delete methode toe. Deze zorgt automatisch dat alle kinderen van het component zichzelf verwijderen en verwijdert dan zichzelf uit het component waar hij in zit.

Het doel van OOP

- Organisatie van code en data
- Geen herhaling van code
- Information hiding (encapsulation)

Uit dit verhaal blijken de belangrijkste voordelen van object-georienteerd programmeren. Het kan altijd zonder, maar als je code groeit en de eisen blijven veranderen, kun je met slim gebruik van objecten en inheritance het hoofd boven water houden.

Objecten laten je methoden en data bij elkaar opslaan.

Inheritance geeft je de mogelijkheid om herhaling van code te voorkomen: Student B hoefde niet voor iedere component dezelfde `getSound()` methode te schrijven; hij kon de overeenkomsten tussen de componenten abstraheren naar een superklasse.

Objecten laten je de ruwe data verbergen, zodat de gebruiker van het object alleen die dingen kan doen die correct zijn met de data. Zo zou student A bijvoorbeeld zijn `choices` array hadden kunnen verbergen in een object, zodat niet iedereen zomaar componenten had kunnen verwijderen zonder de subcomponenten weg te halen.

Voorbeeld: Date

```
class Date {
    public int day,
               month,
               year;

    Date add(int n){...}
}
```

```
class Date {
  private int day,
              month,
              year;

  public Date(
    int d, int m, int y)
  {...}

  int day(){...}
  int month(){...}
  int year(){...}

  Date add(int n){...}
}
```

```
class Date {
    // since 1 Jan 1970
    private long days;

    int day(){...}
    int month(){...}
    int year(){...}

    Date add(int n){...}
}
```

Information hiding is een belangrijk principe, dus daar geven we nog een voorbeeld van. Een datum klasse kun je op twee manieren maken: met drie velden voor de dag, maand en het jaar, of met een enkel veld waarin het aantal dagen vanaf een bepaalde vaste dag opgeslagen is.

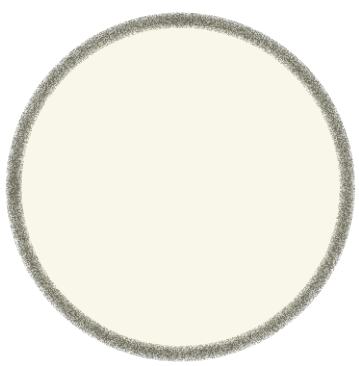
In het eerste geval is een methode zoals add() wat moeilijker te schrijven en in het tweede geval vereisen de methoden day(), month() en year() wat meer rekenwerk.

Het belangrijkste punt om op te merken is dat als we de meest linker optie gebruiken, met publieke velden voor de dag, maand en het jaar, we een object kunnen hebben dat een illegale datum voorstelt (bijvoorbeeld -56 Maart in het jaar 0). Door de velden private te maken en goed op te letten in de methoden kunnen we garanderen dat als een Datum object bestaat, het altijd een geldige datum representeert.

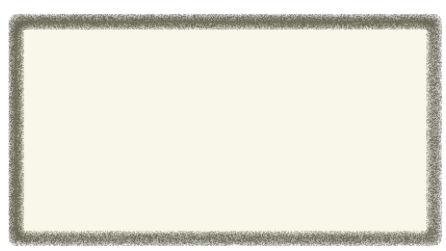
Inheritance, de details

```
class Shape
{}
```

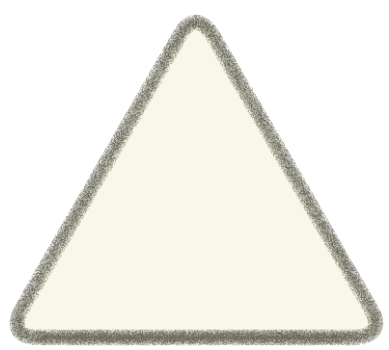
```
class Circle
    extends Shape
{}
```



```
class Rectangle
    extends Shape
{}
```



```
class Triangle
    extends Shape
{}
```



Je geeft aan dat een klasse een van een andere klasse overerft door middel van het `extends` keyword.

Inheritance, de details

```
class Shape
{
    protected Point center;

    public void move(double x, double y)
    {...}
}
```

```
class Circle
    extends Shape
{}
```

```
class Rectangle
    extends Shape
{}
```

```
class Triangle
    extends Shape
{}
```

In de superklasse definieer je de velden en de methoden die alle kinderen gemeen hebben. Alle vormen (Shapes) hebben een middelpunt, en je kunt ze bewegen. Met de acces modifier protected geef je aan dat een methode of veld alleen toegankelijk is voor kinderen van de huidige klasse.

Inheritance, de details

```
class Shape
{
    double circumference(){...}
}
```

```
class Circle extends Shape
{
    private double radius;

    double circumference(){...}
    double radius(){...}
}
```

```
class Rectangle extends Shape
{
    private double width, height;

    double circumference(){...}
    double width(){...}
    double height(){...}
}
```

In de kinderen (subclasses) geef je de methoden aan die alleen voor de kinderen gelden, en override je de methoden van de parent als het gedrag van het kind speciaal is.

Inheritance, **super**

```
class Shape {
    Point center;
    ...
}


class Triangle extends Shape {
    /**
     * Flips this triangle upside down
     */
    public void flip()
    {
        // get center
        Point c = super.center
        // flip vertically around c
        ...
    }
}
```

Als je de velden of methoden van de superklasse wilt benaderen kun je het `super` keyword gebruiken. Dit werkt net als `this`. Je kunt super hier ook weglaten. Als het kind geen veld genaamd center heeft kun je ook gewoon `center` aanroepen. Voor de leesbaarheid is het vaak wel handig om er `super` voor te zetten,

Inheritance, `super`

```
class SuperClass {
    void method()
    {
    }
}

class SubClass extends SuperClass {
    void method()
    {
        // do the standard stuff
        super.method();
        // do my extra stuff
        ...
    }
}
```

Dit is een patroon wat je vaak ziet. De kindklasse *override* de methode van de parent, maar wil de methode niet helemaal definieren. De kindklasse voert eerst de methode van de parentklasse uit en voert dan zijn eigen toevoegingen uit.

In dit geval is het `super` keyword wel nodig. Als we het weglaten krijgen we een oneindige recursie.

Abstract class

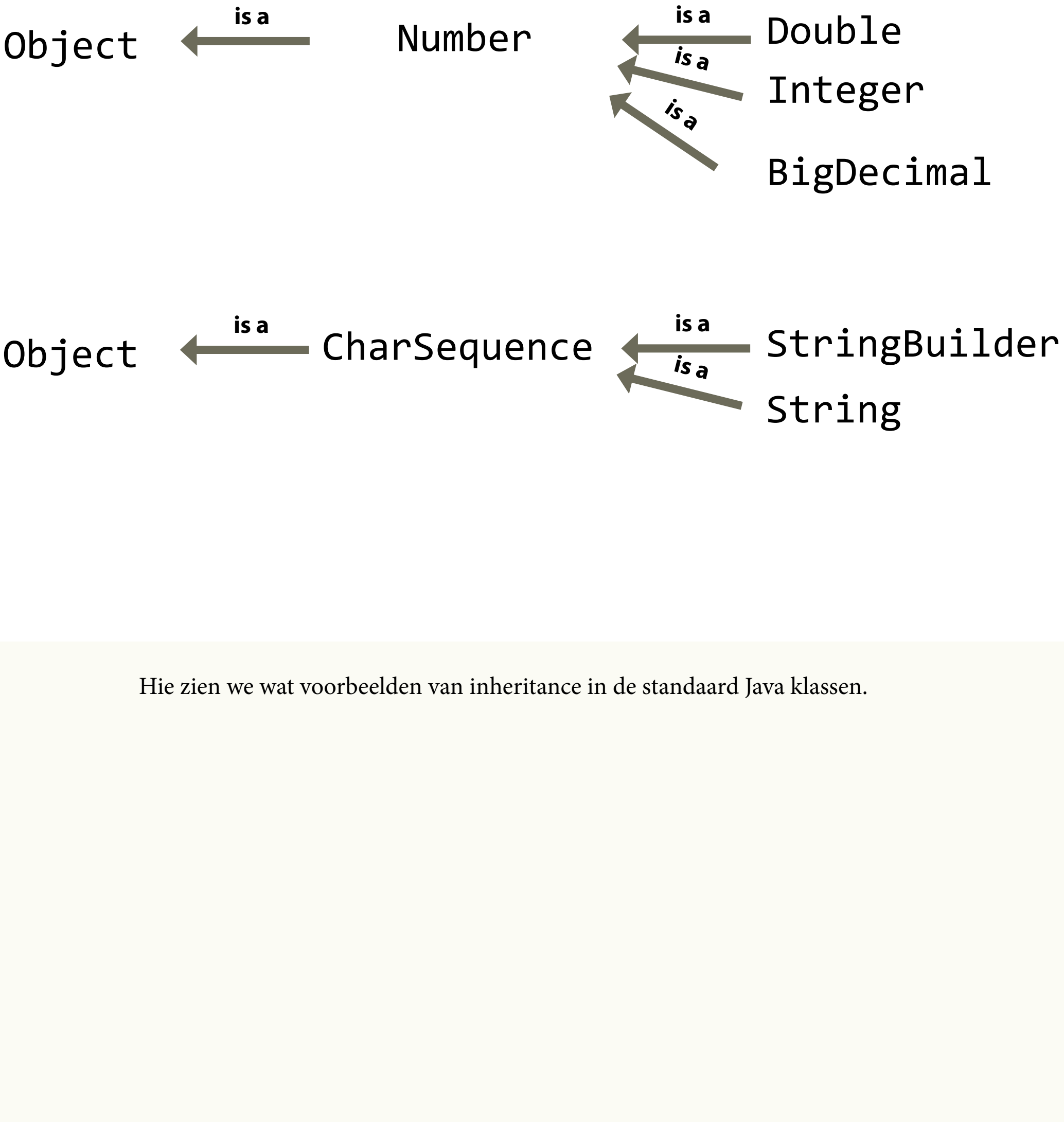
```
abstract class Shape
{
    Point center;
    abstract double circumference();
}
```

Soms is een superklasse heel nuttig om veel voorkomende code te abstraheren, maar heeft het weinig nut om de superklasse zelf te instantieren. In dit geval is het beter als een Shape niet zelf als object kan bestaan. Een shape heeft bijvoorbeeld zelf geen omtrek, maar moet wel vastleggen dat al zijn kinderen een omtrek hebben. In dat geval kun je een abstracte klasse maken.

Een abstracte klasse mag abstracte methoden definiëren. Deze methoden hebben geen body, en worden met een puntkomma afgesloten. Iedere klasse die Shape extend moet nu de methode circumference overriden. Doet de klasse dit niet, dan geeft de compiler een foutmelding.

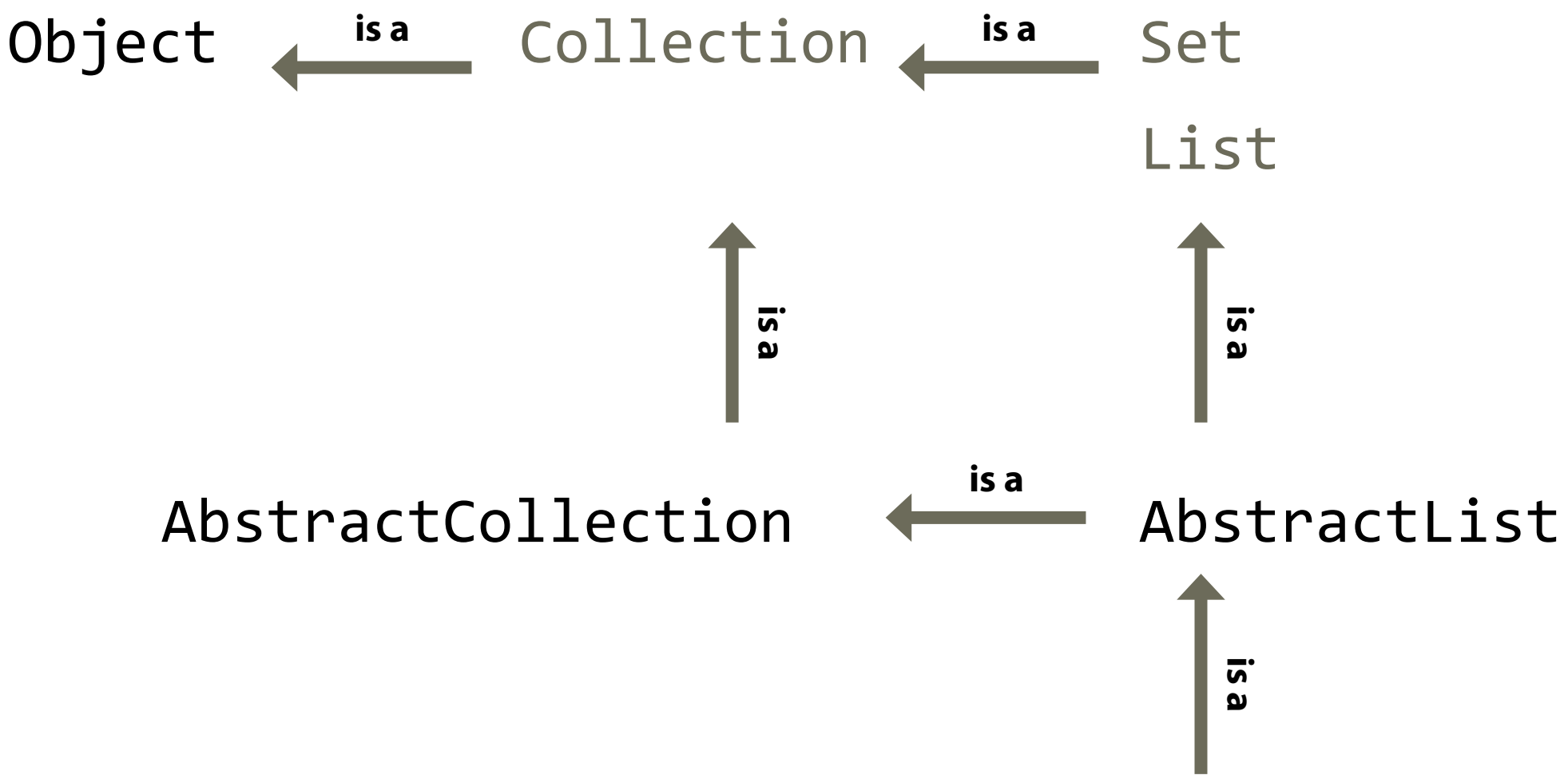
vragen?

Inheritance in de SDK



Hie zien we wat voorbeelden van inheritance in de standaard Java klassen.

Inheritance in de SDK



De java Collection SDK bevat klassen die je helpen om object op te slaan. Daar heb je tot nu toe steeds arrays voor gebruikt, maar de Collections hebben verschillende voordelen. Dit is de hierarchie. De bruingekleurde element zijn eigenlijk geen klassen, maar interfaces. Interface zijn een soort abstracte klassen waarin alle methoden abstract zijn.

Interfaces

- Soort abstracte klasse, met alleen maar abstracte methoden
- Een klasse mag maar 1 andere klasse extenden, maar meerdere interfaces
- Volgend college meer

```
public class MyClass
    extends Super
    implements OneInterface, AnotherInterface
{
    ...
}
```

Inheritance in de SDK

Collection

```
Collection c = ...;
// Grootte
int s = c.size();
// Itereren
Iterator it =
           c.iterator();
while(it.hasNext()) {
    it.next()
}
```

List

```
// Random access
List l = ...;

l.get(0);
l.set(23, "x");
l.indexOf("x");
```

ArrayList

```
// Capacity
ArrayList l = ...;

l.trimToSize();
```

```
// Voordeel
List list = new ArrayList(...);
Collection col = new ArrayList(...);
```

Hier zie je hoe de verschillende functionaliteiten verdeeld zijn: een Collection is iets waar je dingen in kunt stoppen en weer uit kunt halen. Dat uithalen kan alleen één voor één. Als je willekeurige elementen direct wilt kunnen bereiken gebruik je een List.

Een lijst kan je op verschillende manieren maken. Het makkelijkst is door in een object een array te verbergen en daar de elementen in te stoppen. Je moet dan wel zorgen dat als het array te klein is, je een nieuw array aanmaakt en de element overkopieert. Dit heet in java een ArrayList. Als je denkt dat je geen elementen meer gaat toevoegen kun je trimToSize aanroepen, zodat de extra ruimte in het array weg wordt gegooid, en je geen geheugenruimte verspilt.

Een groot voordeel is dat je een referentie naar een ArrayList in een List variable `list` kunt stoppen. Je kunt dan all je code schrijven met de methoden van List, en als je later bedenkt dat een LinkedList toch een betere keuze was geweest, kun je het zonder probleem veranderen.

Als je een Collection variabele gebruikt kun je later zelfs je keuze veranderen naar een Set een Queue of een Stack.

Object

```java
/**
 * String representatie van het object
 * -- standaard:
 */
public String toString() {...}

/**
 * True als de objecten functioneel equivalent zijn
 * -- standaard: alleen letterlijk hetzelfde object
 */
public boolean equals(){...}

/**
 * Hash: een getal dat altijd het zelfde is als twee objecten equal zijn,
 * en zo min mogelijk hetzelfde is voor verschillende objecten
 * -- standaard: het object id.
 */
public int hashCode(){...}
```

Ieder object in Java inherit van de klasse **`Object`** (ook als je niks expliciet extend). Dit betekent dat je de publieke methoden van **`Object`** aan kunt roepen op ieder object dat je tegenkomt. Dit zijn de drie belangrijkste. Als je zelf een klasse maakt, moet je dus goed nadenken over of je de wilt overriden.

Als je equals override is het essentieel dat je ook hashcode override en vice versa. De contracten van beide methoden verwijzen naar elkaar.

toString()

```
Object                MyObject@119c082

Integer               4, -41

List of Integers      [4, 5, 6, 8, 9, 41]
```

Tip: primitives hebben geen toString. Om een array te printen gebruik je `Arrays.toString(myArray);`

Hier zien we hoe de toString method werkt. Als je hem niet override, dan geeft de methode de klasse-naam en de hashcode terug (in hexadecimale notatie).

De Integer klasse geeft de integer terug, zoals we gezend zijn hem te zien.

De List klasse geeft een komm-gescheiden lijst terug van wat er in de lijst zit tussen rechte haken. Voor ieder element roept de lijst de toString() methode van het element aan. Dit is een krachtig principe waardoor je meestal niet erg je best hoeft te doen voor een mooie toString() methode: je kunt gewoon de toStrings van je field mooi verpakken.

Om dit principe in stand te houden geeft de toString methode meestal een string zonder regeleinden terug. Voor meer informatie, over meerdere regels, kun je beter een aparte methode schrijven.

De 3 principes van inheritance

1. Alle methoden en velden die niet `private` zijn, inheriten
2. Als C `extends` P, dan mag C worden toegekend aan een referentie van P: **`P thing = new C();`**
3. Als C `extends` P, dan mogen P's methoden overridden worden in C.

1) Alle methoden en fields inheriten

```
abstract class Food {
    ...
    protected int calories;
    ...
}

class Carrot extends Food {
    public double recommended()
    {
        return 3.1 * calories + 15;
    }
}
```

```
abstract class Food {
    protected int calories()
    {
        ...
    }
}

class Carrot extends Food {
    public double recommended()
    {
        return 3.1 * calories() + 15;
    }
}
```

Alle methoden en field van een object worden door een subklasse ge-orven. Als ze private zijn, kun je er vanuit de subklasse niet bij.

De protected access modifier is speciaal voor inheritance. Dingen die protected zijn kunnen allen vanuit de klasse zelf of vanuit kinderen van de klasse aangeroepen worden.

`private` methoden en fields zijn niet accessible vanuit de subklasse. Of ze daarmee niet inheriten of alleen niet accessible zijn, is een beetje een filosofische vraag. Zie bijvoorbeeld: `http://stackoverflow.com/questions/4716040/does-subclasses-inherit-private-fields`

2) Superklasse referenties

```
class Person {}
class Woman extends Person {}

// assignment
Person p = new Woman();

Woman w = new Person(); // dit mag niet

// method calling
void personMethod(Person p) {}

Woman woman = new Woman();
personMethod(new Woman());
```

2) Superklasse referenties

```
Shape[] shapes = new Shape[1];

shapes[0] = new Circle(...);

// This works (all shapes have an area)
System.out.println(shapes[0].area());

// This doesn't. The object is a circle, but we're
// referencing it as a shape, which doesn't have a
// radius
System.out.println(shapes[0].radius());

// This works, but it's ugly and dangerous.
System.out.println( ((Circle) shapes[0]).radius());
```

3) Overriden

```
class Polygon {

    double area()
    {
        // complicated calculation
    }
}

class Triangle extends Polygon{

    @Override
    double area()
    {
        // simpler calculation
    }
}
```

3) Overriden

```
class City {

    public void destroy() {
        // remove city
        // lower player income
    }
}

class Capital extends City{

    @Override
    public void destroy() {
        super.destroy();
        // remove player from game
        ...
    }
}
```

Nog een voorbeeld. Stel je maakt een strategiespel waarbij spelers elkaars steden kunnen veroveren en eventueel kunnen verwoesten. Als je hoofdstad verwoest wordt heb je verloren.

In deze situatie kun je voor steden in het algemeen de standaardstappen implementeren en alleen voor hoofdsteden (een subklasse) een extra methode maken. Deze roept eerst de destroy methode van de superklasse aan en voegt dan zijn eigen stappen toe.

Constructors

- Constructors erven niet over.
- De superklasse moet eerst geconstruct worden

```
class Polygon {
    // geen default
    // constructor
    public Polygon(
            int numSides)
    {...}
}
```

```
class Triangle
    extends  Polygon {

    public Triangle()
    {
        // eerst super
        // constructen
        super(3);
        // dan de rest
        ...
    }
}
```

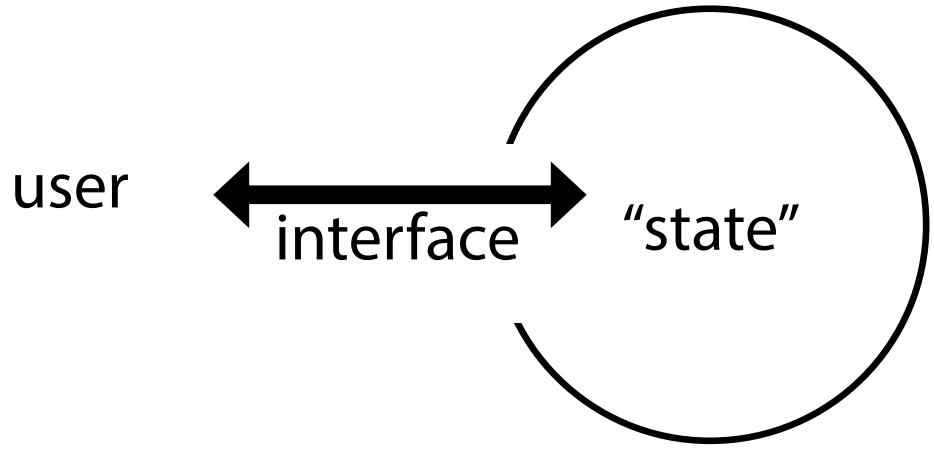
```
class IsoTriangle
    Triangle extends {

    public IsoTriangle()
    {
    // gebruikt automa-
    // tisch lege con-
    // structor
    }
}
```

In tegenstelling tot methoden worden constructors niet ge-orven.

Je moet wel, als eerste regel in de constructor van je subklasse zorgen dat de superklasse geconstruct wordt. Je doet dit door super aan te roepen met tussen haakjes de argument van de constructor die wilt hebben. Als er een default constructor is en je doet dit niet, dan wordt de default constructor gebruikt. Als er geen default constructor is en je doet dit niet compileert je code niet.

Encapsulation

- Een object is een ‘pakketje’ met een binnenkant en een buitenkant



Vertel de gebruiker wat je object doet, maar niet hoe het werkt.

- Maak fields altijd private (of heel soms protected), **nooit public**
- Pas goed op met inheritance

getters (en setters)

```java
public class Book {
    private String title;
    private Person author;

    public Book(String author, String title) {
        ...
    }

    public Person getAuthor() {
        return author;
    }

    public void setAuthor(Person author) {
        this.author = author;
    }
}
```

Dit is een patroon dat je vaak ziet. Een private field, dat door een getter en setter methode aangeroepen wordt. Dit lijkt misschien veel werk maar het heeft veel voordelen.

getters (en setters)

```java
public class Date {
    private long daysSince1Jan1970;

    public int getDay(){...}
    public int getMonth(){...}
    public int getYear(){...}

    public void setDay(int d){...}
    public void setMonth(int m){...}
    public void setYear(int y){...}
}
```

```java
public class Date {
    private int day;
    private int month;
    private int year;

    public int getDay(){...}
    public int getMonth(){...}
    public int getYear(){...}

    public void setDay(int d){...}
    public void setMonth(int m){...}
    public void setYear(int y){...}
}
```

Hier is bijvoorbeeld de `Date` klasse nog een keer. De linker heeft gegarandeerd altijd een correcte state. Voor de buitenwereld lijkt het alsof een date uit drie ints bestaat, maar het is onmogelijk om de maand op 20 te zetten of de dag op -10. Het kost wat rekenwerk, maar dat verschil is op moderne hardware niet merkbaar.

De rechter implementatie kan in theorie een illegale state hebben, maar omdat we getters en setters gebruiker, kunnen we in de stter controleren of het argument in de juiste range valt. Omdat de fields private zijn, weten we zeker dat we alle code die toegang tot het veld heeft zelf onder controle hebben.

Als we ooit besluiten dat we van de rechter implementatie willen overstappen naar de linker, dan hoeven we ons geen zorgen te maken dat we allerlei code kapot maken die one Date klasse gebruiken. De **interface** blijft hetzelfde, we passen alleen de **implementatie** aan. Dit is een heel belangrijk principe waar we nog een aantal keer op terugkomen.

Polymorphisme

```
class Math {
    public int max(int a, int b) {...}

    public double max(double a, double b) {...}
}

System.out.println(max(1, 2));
```

Dit heet ook wel *method overloading*.

Polymorphisme

Overloading is nuttig, maar wees er voorzichtig mee

```
class Test {
    static void m(int a, int b) {System.out.println("ints!");}
    static void m(double a, double b) {System.out.println("doubles!");}
    static void m(double a, int b) {System.out.println("mixed!");}

    public static void main(String[] args)
    {
        m(1, 0.1);
    }
}
```



mixed! compileert niet doubles!

Wat is de uitvoer van deze code?

Polymorphisme

```
class Test {
    static void m(Object n) {System.out.println("Object!");}
    static void m(Double s) {System.out.println("Double!");}
    static void m(String s) {System.out.println("String!");}

    public static void main(String[] args) {
        Object[] os = new Object[3];
        os[0] = new Object(); os[1] = new Double(1.0); os[2] = "";

        for(Object o : os)
            m(o);
    }
}
```

Object!, Double!, String!

Object!, Object!, Object!

Compileert niet

Wat is de uitvoer van deze code?

Overriding

```
class A { void m(){System.out.println("A!");}}
class B extends A { void m(){System.out.println("B!");}}
class C extends B { void m(){System.out.println("C!");}}

class Test {
    public static void main(String[] args) {
        A[] as = new A[3];
        as[0] = new A(); as[1] = new B(); as[2] = new C();

        for(A a : as)
            a.m();
    }
}
```

A!, B!, C!

A!, A!, A!

Compileert niet

Wat is de uitvoer van deze code?

Casting, instanceof

```
List stuff = ...;

stuff.add("x");
String l =
    stuff.get(0).toLowerCase();
// compiler fout
```

```
List stuff = ...;

stuff.add("x");

String s = (String) stuff.get(0);
String l = s.toLowerCase();
```

```
List stuff = ...;

stuff.add("x");

if(stuff.get(0) instanceof String)
{
    String s = (String) stuff.get(0)
    String l = s.toLowerCase();
} else
    ...
```

Als we een list op de ouderwetse manier gebruiken, stoppen we er bijvoorbeeld een String in, maar krijgen we er een Object uit. We kunnen dus niet direct de methoden aanroepen van de String klasse.

Omdat we in dit geval zeker weten dat het object dat we terug krijgen een String is, kunnen we het *casten* naar een String. We veranderen daarmee het type van de referentie. Zodra we een String referentie hebben kunnen we de String methoden aanroepen.

Casten op objecten is gevaarlijk. Als ons object toch geen string blijkt te zijn, merken we dat pas tijdens het draaien van het programma (we krijgen een ClassCastException). In sommige gevallen kan die situatie zo zeldzaam zijn dat het pas fout gaat als het programma al in productie draait en er veel geld of levens van afhankelijk zijn.

Als je het zeker wilt weten kun je met instanceof controleren of een bepaald object naar een bepaalde Klasse of Interface gecast kan worden.

Sneek preview: generics

```
List<String> stuff = ...;

stuff.add("x");

// geen probleem
stuff.get(0).toLowerCase();

// dit mag niet meer
stuff.add(new Double(1.0));
```

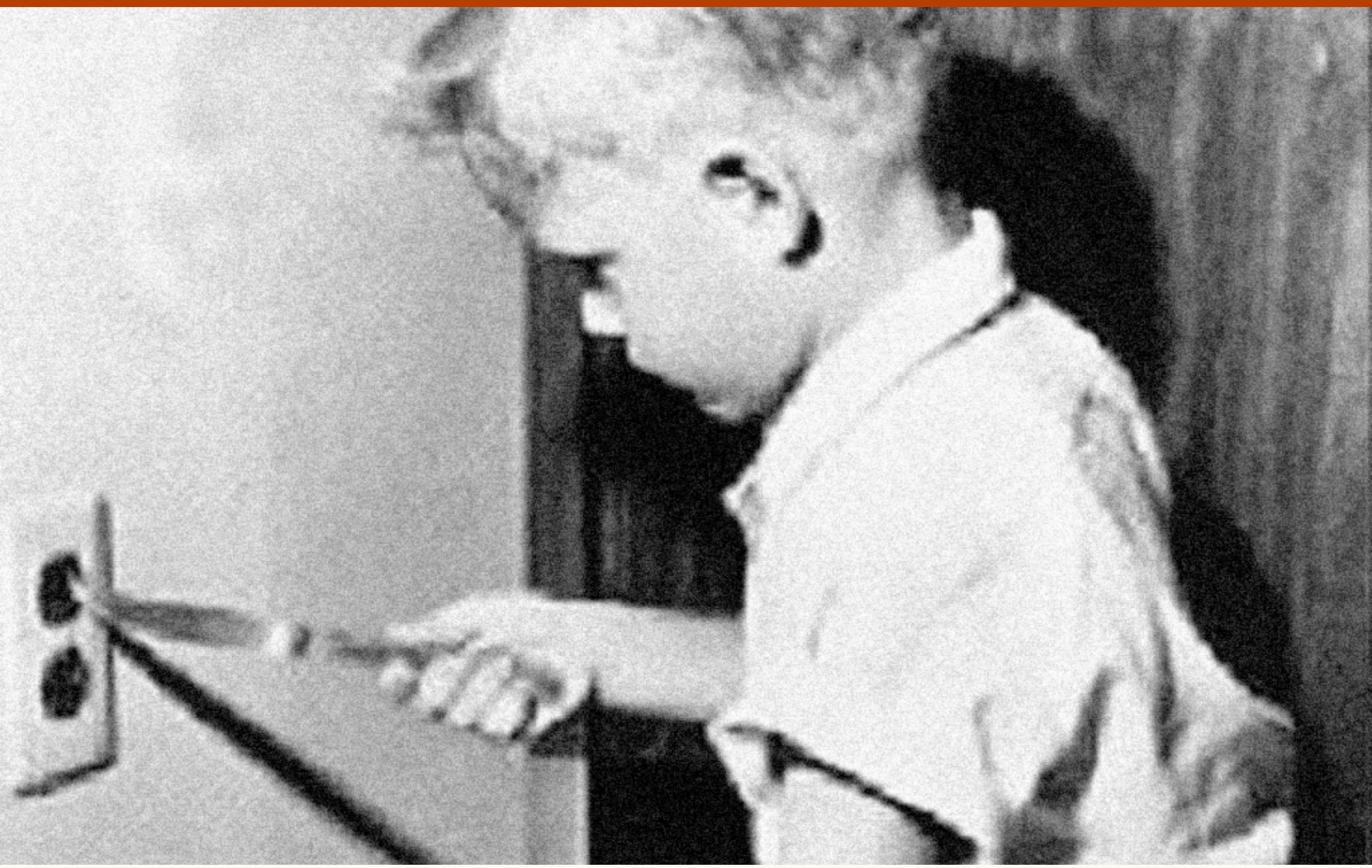
volgend college meer

Omdat deze situatie runtime errors en lelijke code in de hand werkte werden in Java 1.5 generics geintroduceerd. Met generics kun je (onder andere) aangeven wat voor type objecten je in een Collection stopt. Dit betekent dat als je goed programmeert, je bijna nooit meer hoeft te casten. Het kan nog steeds voorkomen, maar meestal in uitzonderlijke situaties.

Met Generics kun je overigens veel meer dan alleen maar zeggen wat er in een Lijst zit. Volgend college leggen we generics uitgebreid uit.

vragen?

things to try



De opdrachten die we jullie geven zijn noodzakelijkerwijs een beetje droog. Ze moeten snel te maken zijn, en we moeten kunnen zien of jullie alle principes begrijpen. Het nadeel is dat je daarmee niet ervaart waarom programmeren leuk is. Om de echte verslaving van programmeren te ontdekken, en om er goed in te worden is het belangrijk om zelf iets te bedenken, the programmeren en er mee te spelen.

Je hebt misschien het idee dat je nog niet echt kunt programmeren, maar hier zijn wat ideeën voor programma's die je nu al moet kunnen schrijven.

1) games

```
Welcome to Zork (originally Dungeon). This version crea
There are 5 users playing Zork. Of those, 5 have not lo
There are 44990 registered adventurers.

You are in an open field west of a big white house with
front door.
There is a small mailbox here.

> open mailbox

Opening the mailbox reveals:
A leaflet.

>
```

Java heeft hele mooie libraries voor grafische dingen en interactie, maar ze zijn nogal ingewikkeld om te leren. Gelukkig kun je met alleen de Scanner klasse ook al een eind komen. Voor een Zork-style text-adventure heb je niets mee nodig dan wat string manipulaties, wat if statements en een enkele loop.

Als je geen zin hebt in Zork, kun je ook denken aan zeeslag, schaak, of een zelf bedacht spel.

2) bestaan Lychrel nummers?

```
59
59 + 95 = 154
154 + 451 = 605
605 + 506 = 1111
```

```
32
32 + 23 = 55
```

```
196
196 + 691 = 887
887 + 788 = 1675
1675 + 5761 = 7436
7436 + 6347 = 13783
13783 + 38731 = 52514
52514 + 41525 = 94039
94039 + 93049 = 187088
187088 + 880781 = 1067
1067869 + 9687601 = 10
10755470 + 07455701 =
18211171 + 17111281 =
35322452 + 25422353 =
60744805 + 115985111 =
111589511 + 115985111
227574622 + 226475722
454050344 + 443050454
897100798 + 897001798
1794102596 + 695201497
```

Een Lychrel proces bestaat uit de iteratie van een nummer omdraaien, en het bij zichzelf optellen. Bijna alle getallen vormen na verloop van tijd een palindroom. Sommige, zoals 89, en 1091 pas na een groot aantal iteraties.

Sommige getallen, zoals 196 en 295, lijken nooit een palindroom te bereiken, maar een bewijs is nog nooit gevonden.

3) het Monty Hall probleem

switch stay maakt niet uit

Het Monty Hall probleem: een finalist in een quizshow mag als eindprijs één van drie deuren kiezen. Achter één van de deuren is een auto. Achter de twee andere staan geiten (die de finalist niet wil winnen).

Nadat de finalist een keuze heeft gemaakt. Doet de quizmaster (Monty Hall) één van de andere deuren open. De quizmaster weet waar de auto zit, dus die deur doet hij nooit open. Er zijn nu nog twee deuren over. De deur die de finalist had gekozen, of de andere deur. De quizmaster biedt de finalist nu aan om haar keuze te veranderen. Kan de finalist beter voor de andere deur kiezen (switch), bij haar keuze blijven (stay) of maakt het niet uit?

We geven in dit college niet het antwoord. De belangrijkste vraag, als je het met elkaar oneens bent, is hoe overtuig je iemand? Dit is een probleem waar Statistiek professoren zich de vingers aan hebben gebrand. De kans is groot dat je er met argumentatie niet uitkomt. Om iemand echt te overtuigen, moet je het uitproberen. En daar heb je nu de middelen voor.

Tip, gebruik: `Random rand = new Random();`

`int choice = rand.nextInt(3);` voor willekeurige getallen