Java traineeship

De eerste basis

Leerdoelen

Let op: er komen veel onderwerpen aan bod die je nog niet meteen zult begrijpen. In latere hoofdstukken gaan we dieper op alle onderwerpen van dit eerste hoofdstuk in.

In dit hoofdstuk maken we daarom enkel kennis met de vele programmeerconcepten rondom **Java**

Leerdoelen

- Het herkennen van de structuur van een `.java` klasse.
- Het uitvoeren van je eerste Java applicatie vanaf de command line.
- Het importeren van Java packages
- Het toepassen van de juiste *access modifier*
- Kennis maken met abstracte klasses
- Kennis maken met het *static* keyword
- Kennis over *Java domain features* en *components*

Eigenschappen van Java

- Platform onafhankelijkheid
  - Java werkt met de visie: Write Once, Run Anywhere (WORA)
- Object georiënteerd
  - Java emuleert real-life objectdefinitie en gedrag. In het echte leven zijn toestand en gedrag aan een object gebonden.
- Abstractie
  - Met Java kun je objecten abstraheren en alleen de vereiste eigenschappen en gedragingen in uw code opnemen.
- Encapsulatie
  - De status of de velden van een klasse zijn beschermd tegen ongewenste toegang en manipulatie. Je kunt het toegangsniveau en de wijzigingen aan uw objecten beheren.
- overerving
  - Java stelt zijn klassen in staat om andere klassen te erven en interfaces te implementeren. Dit voorkomt dat je gemeenschappelijke code opnieuw moet definiëren.
- Polymorfisme
  - De letterlijke betekenis van polymorfisme is 'vele vormen'. Java stelt instanties van zijn klassen in staat om meerdere gedragingen te vertonen voor dezelfde methodeaanroepen
- Type safety
  - Je moet een variabele met zijn gegevenstype declareren *voordat* je deze kunt gebruiken.

Motivatie Java

Stel je voor dat je een nieuwe IT organisatie opzet, waar meerdere ontwikkelaars werken.

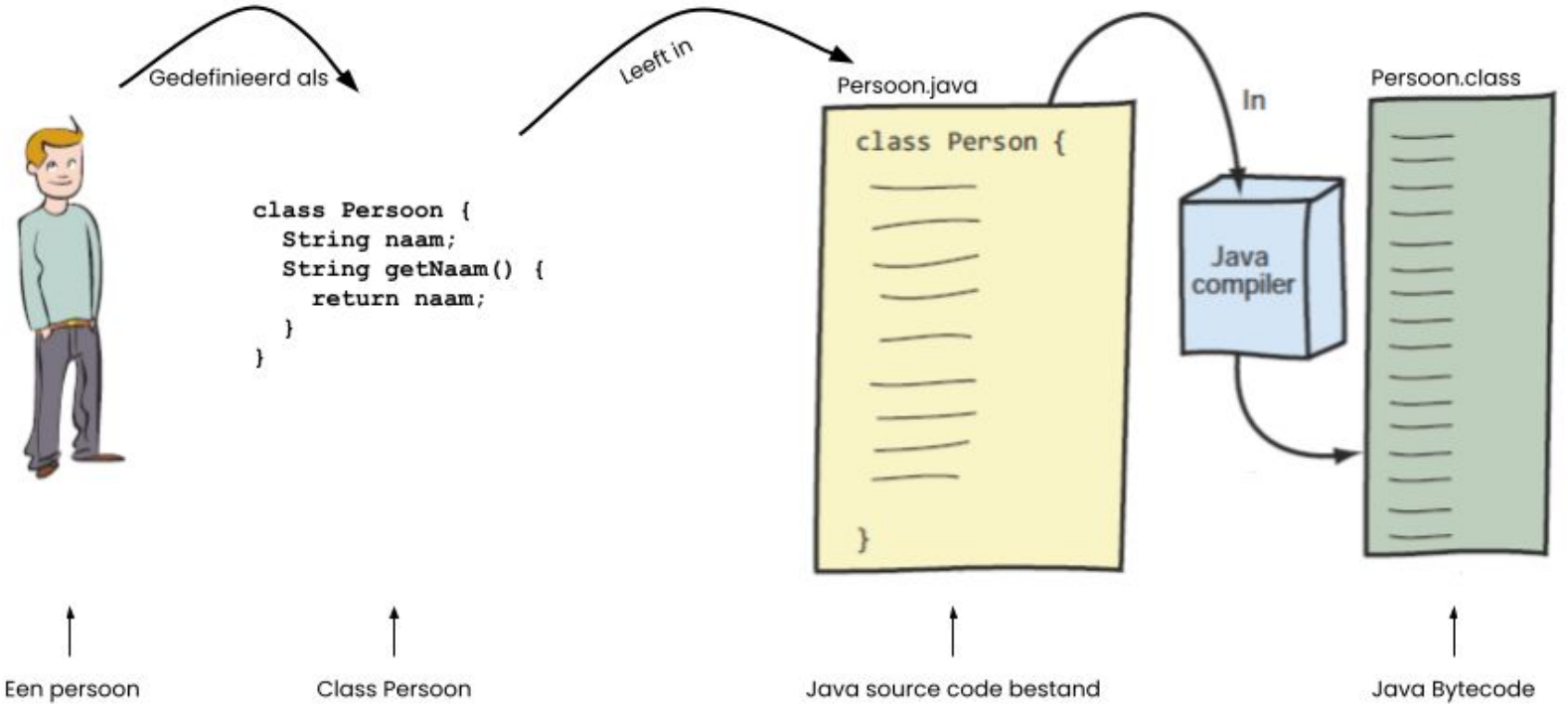
Om een vlotte en efficiënte werking te garanderen, deel je het bedrijf op in afdelingen en krijgt iedereen afzonderlijke verantwoordelijkheden.

Afdelingen communiceren alleen maar met elkaar als dat nodig is. Hiernaast hoeft niet elke medewerker toegang te hebben tot alle bedrijfsgegevens.

Java geeft je alle handvaten om op dezelfde manier je code in te delen.

De structuur en componenten van de organisatie kunnen worden vergeleken met de klassenstructuur en componenten van Java, en de afdelingen van de organisatie kunnen worden vergeleken met Java-pakketten. Het beperken van de toegang tot sommige gegevens in de organisatie kan worden vergeleken met de toegangsmodifiers van Java.

Herhaling:
Java source files (.java) en Java class files (.class)



De structuur van een Java klasse

Java klasses hebben altijd dezelfde opbouw:

```
Package statement
Import statement(s)
Comments
Klasse declaratie {
    Variabelen
    Comments
    Constructors
    Methoden
}
```

Laten we de componenten eens een voor een doorlopen.

Package statement

Alle Java klasses zijn onderdeel van een *package*.

Het *Package statement* wordt gebruikt om expliciet te definiëren in welk pakket een klasse zich bevindt.

Als een klasse een pakketinstructie bevat, moet dit de eerste instructie in de klassendefinitie zijn:

```
package mijn_pakket_naam;
class Voorbeeld {

}
```

Het package statement moet het eerste statement van de java klasse zijn.

De rest van de code voor de klasse *Voorbeeld.*

Package statement (fout)

```
class Voorbeeld {

}
package mijn_pakket_naam;
```

Als je de package statement plaatst na de klasse definitie compileert je code niet.

```
class Voorbeeld {
  package mijn_pakket_naam;
}
```

Een package statement kan ook niet binnen een klasse definitie geplaatst worden.

```
package mijn_pakket_naam;
package nog_een_naam;
class Voorbeeld {
}
```

Je kan niet meer dan één package statement hebben in je code.

Opdracht: correcte package

Stap 1: creëer een nieuwe map met daarin twee submappen: *com* en *cert* (bijvoorbeeld: java_traineeship/com/cert)

Stap 2: Maak in de *cert* map een Java bestand genaamd *Opdracht_1.java*

Stap 3: Zet de juiste verwijzing naar de package in de code

Stap 4: Maak een Opdracht_1 klasse aan

Stap 5: Kijk of de code compileert.

Import statement

Klasses en interfaces in dezelfde package kunnen elkaar gebruiken zonder de *prefix* van de package naam er voor te hoeven zetten.

Dit is wel het geval wanneer je iets gebruikt van een andere package. Dan gebruik je de volledige naam, namelijk:

```
packageNaam.SubpackageNaam.Klassenaam
```

Een voorbeeld hiervan vind je terug in het importeren van de *String* klasse in Java:

```
java.lang.String
```

Import: naamgevingsconventies

Een Java package volgt altijd een bepaalde structuur. Deze gaan van meest generiek (links) naar meest specifiek (rechts).

Neem bijvoorbeeld de package naam: `com.oracle.javacert.associate`

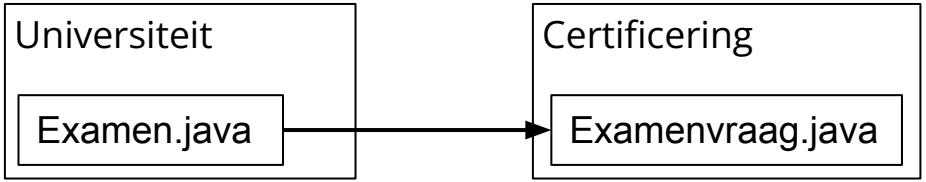
Package of subpackage naam	Betekenis
com	Commercieel. Je hebt ook nog andere afkortingen zoals `gov` en `edu` voor government en education.
oracle	Naam van de organisatie.
javacert	Een categorie of een bepaald project binnen de organisatie.
associate	Een verdere subcategorie waar deze code in valt.

Import: Instellen van je mappen

Elk pakket of subpakket bevind zich in de naam van de aangegeven package. Als we weer hetzelfde voorbeeld (`com.oracle.javacert.associate`) nemen staat de code van dit pakket dus hier:

Import statement: Voorbeeld

Stel we hebben de volgende situatie:

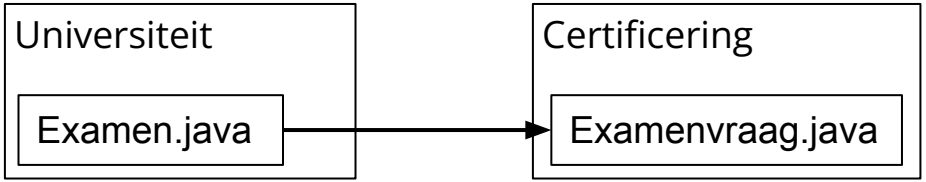


De universiteit package heeft informatie nodig van de Certificering package. Het *Examen.java* bestand kan dan de inhoud van de *Examenvraag.java* importeren om hier gebruik van te kunnen maken.

```
package universiteit;
import certificering.Examenvraag;
class Examen {
  Examenvraag ev;
}
```

Opdracht: Importeren oefenen

Programmeer de volgende situatie:



Kijk of je de code kan laten compileren.

Denk goed na hoe de mappenstructuur moet worden gebouwd.

Comments

Weet je nog?
Je kan ook opmerkingen toevoegen aan je Java code. Dit wordt voornamelijk gebruikt om de code die je hebt geschreven uit te leggen.

Het is namelijk makkelijker om Nederlands/Engels te lezen dan honderden regels Java code.

```
class Test {
 /*
   Ik kan meerdere regels
   comments schrijven.
 */
}
```

en:

```
class Test {
 // ik gebruik deze manier voor één zin
}
```

Comments

Over het algemeen is het gebruikelijk om comments boven stukken code te zetten. Nooit er naast.

```
class Persoon {
    // Dit is de naam van de persoon.
    String naam;

    // Dit is de id van de persoon.
    String id;
}
```

```
class Persoon {
    String naam; // Dit is de naam van de persoon.
    String id; // Dit is de id van de persoon.
}
```

Comments

Alles na de // of /* wordt gezien als een comment. Deze code zal niet worden uitgevoerd.

```
class Persoon {
    // Dit is de naam van de persoon.
    // String naam;

    // Dit is de id van de persoon.
    // String id;
}
```

Veel editors geven dit ook aan door de kleur van je code te veranderen.

Comments: Voorbeeld

```java
/**
 * @author JBrandsen
 * @version 1.0
 *
 * Klasse die een persoon representeert.
 */
package universiteit;
class Persoon {
    // Hoe oud iemand is.
    int leeftijd;
    String naam;
}
// Nog een comment.
```

Probeer zelf eens een mooie header comment te maken. Welke informatie kan je er nog meer allemaal inzetten?

Klasses

Een klasse declaratie markeert de start van een *class*.
Deze heb je nu al enkele keren gezien:

```
public class Persoon {

}
```

Een klasse is een ontwerp dat wordt gebruikt om de kenmerken en het gedrag van een object te specificeren.

De **attributen** van een object worden geïmplementeerd door **variabelen** en het **gedrag** wordt geïmplementeerd door **methoden**.

Klasses

Een bestand mag meer dan één klasse bevatten, maar er mag er maar één **public** zijn. Deze moet dezelfde naam hebben als het *.java* bestand die het programma bevat.

In dit voorbeeld moet de klasse **Persoon** dus worden opgeslagen in **Persoon.java**

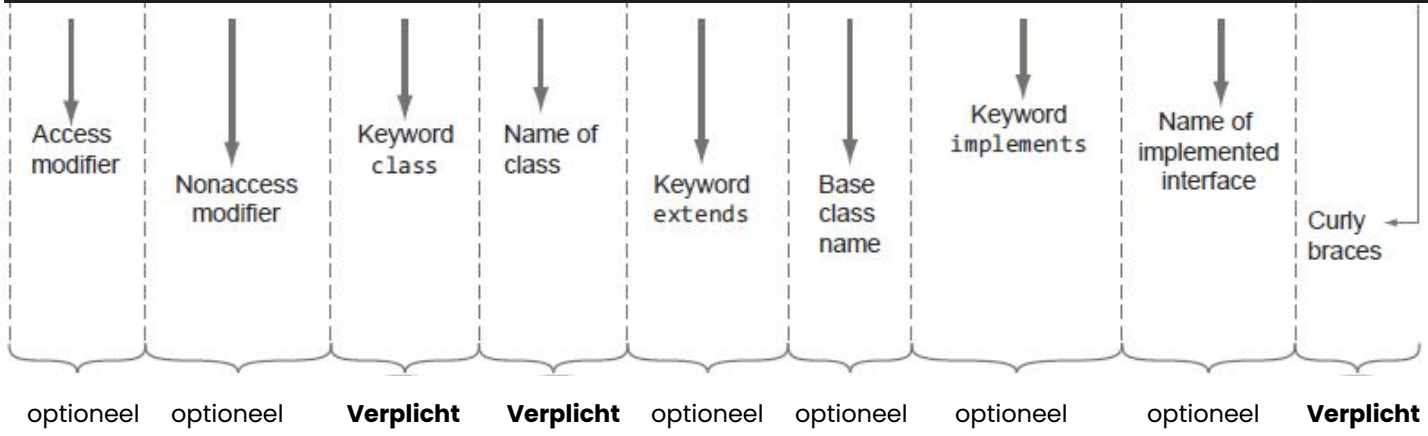
```
public class Persoon {

}
```

Klasses

De declaratie van een klasse kan nog verder worden uitgebreid. Wat dat allemaal betekent, komt in latere lessen nog aan bod.

```
public final class Hardloper extends Persoon implements Atleet {}
```



Variabelen

Variabelen verwijzen naar de staat van een bepaald object. Zo kan de staat van een ‘persoon’ de leeftijd 24 zijn. Echter is niet elke persoon 24, dus kunnen andere objecten een andere leeftijd-staat hebben.

Dus, elk object heeft zijn eigen kopie van de instantievariabelen. Als je de waarde van een instantievariabele voor een object wijzigt, verandert de waarde voor dezelfde genoemde instantievariabele niet voor een ander object.

Methoden

Methoden worden voornamelijk gebruikt om de staat van de variabelen te beïnvloeden. Een voorbeeld hiervan is een methode `rennen()` Elke persoon is in staat om te rennen. Wanneer deze methode wordt aangeroepen zal de `snelheid` van deze persoon aangepast worden.

`Snelheid` is in dit geval dan een **variabele** en `rennen()` de **methode**.

Constructoren

Constructoren worden gebruikt om **objecten** mee te initialiseren.

Een voorbeeld hiervoor is dat elke persoon een *naam* en een *leeftijd* heeft. Wanneer dit object wordt aangemaakt zullen deze waardes vooraf aangegeven moeten worden.

Uitvoerbare Java applicaties

De main methode

De eerste vereiste om een Java applicatie te maken is het creëren van een `main()` methode.

```
public class Voorbeeld {
    public static void main(String[] args) {
        System.out.println("Hallo wereld!");
    }
}
```

De main methode **moet** bestaan uit de volgende elementen:
> De methode moet *public* zijn
> De methode moet *static* zijn
> De naam van de methode moet *main* zijn.
> Het return type moet *void* zijn.
> De methode moet een *String array* als argument accepteren.

Geen probleem als je nog niet weet wat dit allemaal is!

Het uitvoeren van je code

Om je applicatie te laten runnen moet je eerst je code compileren met **javac**. Dit zet je code om naar uitvoerbare bytecode. De code kan dan uitgevoerd worden door het **java** argument te gebruiken en de naam van je klasse te geven:

```
public class Voorbeeld {
    public static void main(String[] args) {
        System.out.println("Hallo wereld!");
    }
}
```

```
> Javac Voorbeeld.java
> java Voorbeeld
Hallo wereld!
```

Het uitvoeren van je code: args[]

Ik heb gezegd dat de hoofdmethode een array van String accepteert als de methodeparameter. Maar hoe en waar geef je de array door aan de hoofdmethode? Laten we de vorige code wijzigen om waarden van deze array te openen en uit te voeren:

```java
public class Voorbeeld {
    public static void main(String[] args) {
        System.out.println(args[0]);
        System.out.println(args[1]);
    }
}
```

```
> Javac Voorbeeld.java
> java Voorbeeld 1 2
1
2
```

De parameters die je meegeeft aan de variabele *args* worden opgeslagen en geprint naar de terminal

Java access modifiers

Access modifiers

Modifier	Class	Package	Subclass	World
`public`	+	+	+	+
`protected`	+	+	+	-
geen modifier	+	+	-	-
`private`	+	-	-	-

Java kent verschillende **access modifiers**. Dit zijn een soort labels die je aan je variabelen en je methodes kan meegeven om aan te geven wie toegang heeft.

Denk nog maar eens terug aan het voorbeeld van de IT-organisatie. Alleen de mensen die zich bezighouden met geldzaken moeten toegang hebben tot het *rekeningnummer*. Een programmeur of schoonmaker niet.

Java heeft de volgende access modifiers: *Public*, *Protected* en *Private*. Deze staan op volgorde van minst restrictief naar meest restrictief.

Access modifiers: public

De **public** modifier is de minst restrictieve modifier. Methodes en variabelen met deze modifier kunnen door elk ander package of programma aangeroepen worden.

```
package bibliotheek;
public class Boek {
    public String isbn;
    public void printBoek() {}
}
```

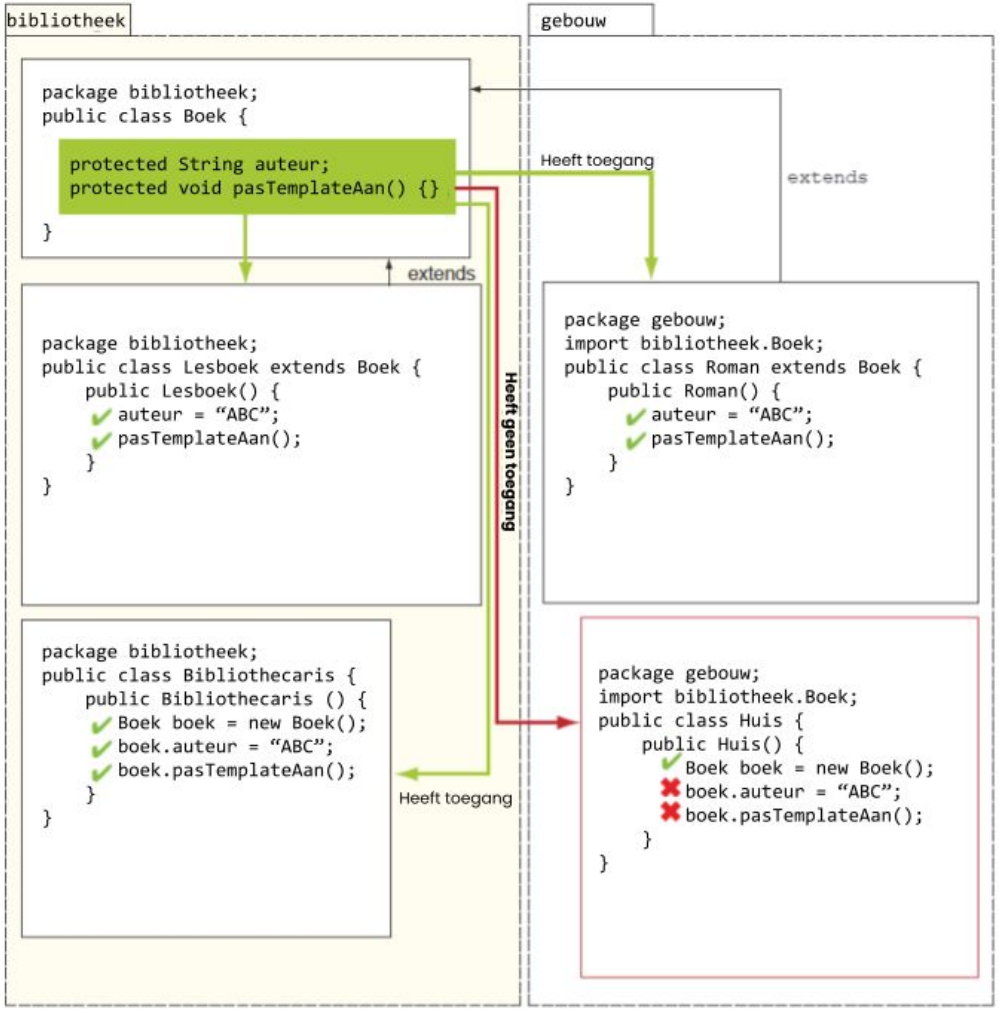
```
package gebouw;
import bibliotheek.Boek;
public class Huis {
    Huis() {
        Boek boek = new Boek();
        String isbn = boek.isbn;
        boek.printBoek();
    }
}
```

De onderstaande code kan gewoon uitgevoerd worden, want de methoden en variabelen van de *Boek* klasse zijn zichtbaar voor de *Huis* klasse.

Access modifiers: protected

Leden van de klasse gedefinieerd onder de **protected** modifier hebben toegang tot:

- Klasses en interfaces van dezelfde package.
- Alle samengestelde klasses, ook al zijn ze gedefinieerd in andere packages.


bibliotheek
gebouw
package bibliotheek;
public class Boek {
protected String auteur;
protected void pasTemplateAan() {}
}
Heeft toegang
extends
extends
package bibliotheek;
public class Lesboek extends Boek {
public Lesboek() {
✔ auteur = "ABC";
✔ pasTemplateAan();
}
}
package gebouw;
import bibliotheek.Boek;
public class Roman extends Boek {
public Roman() {
✔ auteur = "ABC";
✔ pasTemplateAan();
}
}
Heeft geen toegang
package bibliotheek;
public class Bibliothecaris {
public Bibliothecaris () {
✔ Boek boek = new Boek();
✔ boek.auteur = "ABC";
✔ boek.pasTemplateAan();
}
}
Heeft toegang
package gebouw;
import bibliotheek.Boek;
public class Huis {
public Huis() {
✔ Boek boek = new Boek();
✖ boek.auteur = "ABC";
✖ boek.pasTemplateAan();
}
}

Access modifiers: geen modifier

Leden van een klasse die gedefinieerd zijn **zonder modifier** hebben een zogenaamde *package accessibility*. Dit wilt zeggen dat alleen leden die onderdeel zijn van dezelfde package toegang hebben tot de variabelen of methodes van die klasse.

Beschouw een package als je huis, klassen als kamers en dingen in kamers als variabelen met *package accessibility*. Deze dingen zijn niet beperkt tot één kamer - ze zijn toegankelijk in alle kamers in je huis.

Maar ze zijn nog steeds privé voor je huis - je zou niet willen dat ze buiten je huis toegankelijk zijn.

Access modifiers: private

De modifier voor **privétoegang** is de meest beperkende toegangsmodifier. De leden van een klasse die zijn gedefinieerd met behulp van de modifier voor privétoegang, zijn alleen voor henzelf toegankelijk.

```
class A {
    private int data = 40;
    private void bericht() {
        System.out.println("Hello java");
    }
}

public class B {
    public static void main(String[] args) {
        A obj = new A();
        System.out.println(obj.data); //Compile Time Error
        obj.bericht(); //Compile Time Error
    }
}
```

Java nonaccess modifiers

Nonaccess modifiers

Access modifiers bepalen de toegankelijkheid van je klasse en zijn leden buiten de klasse en het pakket. **Nonaccess-modifiers** veranderen het standaardgedrag van een Java-klasse en zijn leden.

Een voorbeeld hiervan is het keyword **abstract**. Als je deze toevoegt aan je klasse, kan deze niet meer worden geïnstantieerd. Dat is de magie van de nonaccess-modifiers.

Naast **abstract** behandelen we ook nog de **final** en **static** keywords.

Nonaccess modifiers: Abstract class

Wanneer het **abstract** keyword is toegevoegd aan de definitie van een klasse, interface of methode, verandert de abstracte modifier zijn standaardgedrag.

De volgende code is een geldig voorbeeld van een abstracte klasse:

```
abstract class Persoon {
    private String naam;
    public void printNaam() {}
}
```

```
class Universiteit {
    Persoon p = new Persoon();
}
```

Als we nu een nieuw object willen instantiëren van de Persoon klasse, krijgen we een error.

Je kan een abstracte klasse daarom het beste zien als een blauwdruk. In verdere hoofdstukken behandelen we dit verder.

Nonaccess modifiers: Abstract interface

Voor modulaire software is het vaak fijn bepaalde garanties over een object te hebben.

Bijvoorbeeld: “Object A heeft een click-methode die aangeroepen kan worden zodra iemand klikt op [..].”

Een garantie dat een gegeven object of klasse bepaalde methoden en constanten bevat, kan in veel programmeertalen worden gegeven met behulp van een **interface**.

Een interface is een soort van contract waar een klasse zich aan moet houden en bevat een lijst van methoden en constanten die een klasse moet implementeren.

Interfaces zijn per definitie ook abstract.

Nonaccess modifiers: Abstract method

Ten slotte hebben we nog **abstract methods**. De kenmerken van zo een methode zijn vergelijkbaar aan die van een *abtracte klasse.*

Abstracte methodes hebben geen *body* dat wilt zeggen dat er geen inhoud is tussen de { en de }.

Een abstracte methode kan gebruikt worden als je wel wilt verplichten dat een object gebruikt maakt van de methode, maar het je niet interesseert wat er specifiek in de methode staat.

Nonaccess modifiers: Final variable

Men gebruikt het **final** keyword bij een variabel wanneer men wilt dat de waarde niet overschreven kan worden.

```
class Persoon {
    final int MAX_AGE = 99;
}
```

```
class Persoon {
    final int MAX_AGE = 99;
    MAX_AGE = 100; // error!
}
```

Dit kan erg handig zijn wanneer je te maken hebt met constante variabelen in je programma.

Nonaccess modifiers: Final method

Methodes kunnen ook gedefinieerd worden als **final**, in dat geval kunnen ze net als variabelen niet overschreven worden.

Het overschrijven van methoden wordt gedaan door klasses die zijn afgeleid van een hoofdklasse:

```
class Persoon {
    final void zing() {
        System.out.println("la..la..la..");
    }
}

class Professor extends Persoon {
     // Compileerfout: final methode mag niet overschreven worden
    void zing() {
        System.out.println("Alpha.. beta.. gamma");
    }
}
```

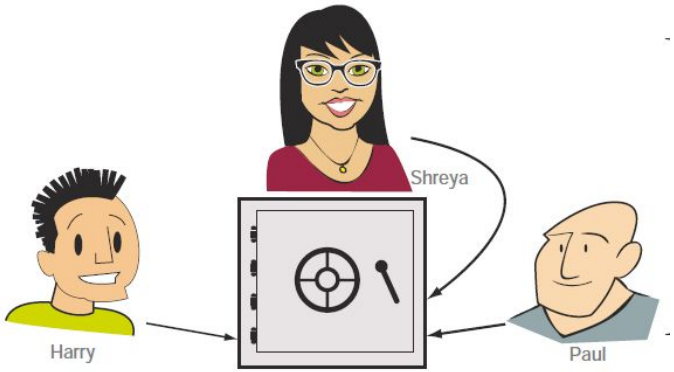
Nonaccess modifiers: static variables

Static variabelen behoren tot een klasse. Ze zijn gemeenschappelijk voor alle instanties van een klasse en zijn niet uniek voor een instantie van een klasse.

statische attributen bestaan onafhankelijk van instanties van een klasse en zijn toegankelijk, zelfs als er geen instanties van de klasse zijn gemaakt.

Je kunt een statische variabele vergelijken met een gedeelde variabele. Een statische variabele wordt gedeeld door alle objecten van een klasse.

Zie een **static variabele** als een gemeenschappelijke bankkluis die wordt gedeeld door de medewerkers van een organisatie. Elk van de werknemers heeft toegang tot dezelfde bankkluis, dus elke wijziging die door een werknemer wordt aangebracht, is zichtbaar voor alle andere werknemers.



Nonaccess modifiers: static variables

Voorbeeld:

```
class Medewerker {
    String naam;
    static int kluiscode;
}

class TestMedewerker {
    public static void main(String[] args) {
        Medewerker med1 = new Medewerker();
        Medewerker med2 = new Medewerker();
        med1.kluiscode = 12345;
        med2.kluiscode = 54321;
        System.out.println(med1.kluiscode); // print 54321, want de code is aangepast door med2
        System.out.println(med2.kluiscode); // print 54321
        System.out.println(Medewerker.kluiscode); // print 54321
    }
}
```

Nonaccess modifiers: static methods

In Java is een **statische methode** een methode die bij een klasse hoort in plaats van bij een instantie van een klasse. De methode is toegankelijk voor elke instantie van een klasse, maar methoden die in een instantie zijn gedefinieerd, zijn alleen toegankelijk voor dat object van een klasse.

Een statische methode maakt geen deel uit van de objecten die hij maakt, maar maakt deel uit van een klassendefinitie. In tegenstelling tot instantiemethoden, wordt naar een statische methode verwezen door de klassenaam en kan deze worden aangeroepen zonder een klasseobject te maken.

Simpel gezegd: zijn het methoden die bestaan, zelfs als er nog geen object is geconstrueerd en waarvoor geen aanroepobject nodig is.

Eindopdracht hoofdstuk 1

Eindopdracht

Schrijf een programma dat het volgende kan:

- onderdeel van een package
- importeert een andere package
- bevat meerdere comments
- bevat een class
  - Heeft variabelen met verschillende (non) access variables
  - Heeft methoden met verschillende (non) access variables
- compileert

Leerdoelen

- ✓ Het herkennen van de structuur van een `.java` klasse.
- ✓ Het uitvoeren van je eerste Java applicatie vanaf de command line.
- ✓ Het importeren van Java packages
- ✓ Het toepassen van de juiste *access modifier*
- ✓ Kennis maken met abstracte klasses
- ✓ Kennis maken met het *static* keyword
- ✓ Kennis over *Java domain features* en *components*

Vragen?

- E-mail mij op voornaam.achternaam@code-cafe.nl!

- Join de Code-Café community op discord!