HO
GENT

H3 Klasse en object

Table of Contents

1. Doelstellingen . . . 1
2. Inleiding: een aantal sleutelconcepten . . . 1
   2.1. Object . . . 1
   2.2. Abstractie . . . 2
   2.3. Klasse . . . 3
3. Klassen en objecten: in de diepte . . . 4
   3.1. Wanneer is iets een object? . . . 4
   3.2. Kenmerken van een object . . . 5
      3.2.1. Toestand . . . 5
      3.2.2. Gedrag . . . 7
      3.2.3. Identiteit . . . 11
   3.3. Verantwoordelijkheid van een object . . . 12
   3.4. Relaties tussen objecten . . . 12
   3.5. Objecten van eenzelfde soort . . . 12
4. Voorstelling klasse in UML . . . 13
   4.1. Inleiding . . . 13
   4.2. Klassenaam . . . 14
   4.3. Attributen . . . 15
   4.4. Operaties (=methodes) . . . 16
   4.5. Constructoren . . . 18
   4.6. Een extraatje: overloading . . . 19
   4.7. Een extraatje: weergave getters en setters in UML . . . 20
5. Implementatie klasse, gedefinieerd in UML . . . 21
   5.1. De klasse . . . 22
   5.2. De attributen . . . 22
   5.3. De getters en setters . . . 23
   5.4. De constructoren . . . 28
   5.5. Andere methodes . . . 30
   5.6. Voorbeeld UML met pijltjes bij attributen en bijhorende code . . . 31
6. Bouwen van een applicatie m.b.v. die klasse . . . 32
   6.1. Functionaliteit van de applicatie . . . 32
   6.2. UML van de applicatie . . . 33
   6.3. Implementatie van de domeinklasse KauwgomAutomaat . . . 34
   6.4. Implementatie van de applicatieklasse KauwgomAutomaatApplicatie . . . 35
      6.4.1. Referentievariabele . . . 35
      6.4.2. Applicatieklasse . . . 37
7. Testen van de domeinklasse . . . 39

1. Doelstellingen

- kan volgende **OO concepten** begrijpen en omschrijven: klasse, object, toestand, eigenschappen, attributen, gedrag, operatie, constructor, getter, setter, actie , identiteit, abstractie, inkapseling
- kan volgende **UML concepten** begrijpen en omschrijven: klassendiagram, attributen, operaties, constructor, visibiliteit
- kan een klasse weergeven in **UML**
- kan een klasse voorgesteld in UML vertalen naar **java code**
- kan die javaklasse **testen** m.b.v. een gegeven testklasse (JUnit)
- kan een **applicatie bouwen** die werkt met objecten van een eigen gemaakte klasse

2. Inleiding: een aantal sleutelconcepten

De naam van dit opleidingsonderdeel is *Object oriented software development*. De bedoeling is om software te ontwikkelen met een object georiënteerde programmeertaal. Het is dus belangrijk om eerst even na te gaan wat een *object* is.

Er bestaan drie sleutelconcepten:

- object
- abstractie
- klasse

2.1. Object

Bij het ontwikkelen van software, heeft de klant een stuk software nodig om bepaalde problemen op te lossen, om efficiënter te kunnen werken, … De klant beschrijft de reële situatie waarvoor de te ontwikkelen software een oplossing moet zijn. Object georiënteerd werken sluit nauw aan bij de werkelijkheid, die reële situatie beschreven door de klant.

Met **real-world objecten** zijn we vertrouwd. Als we even rondom ons kijken, zien we heel wat van die objecten. Deze objecten kunnen we beschrijven:

- Waarvoor dienen ze?
- Welke eigenschappen/karakteristieken hebben ze?
- Wat je ermee kan doen?

Voorbeeld real-world object: een pennenzak

Beschrijving

Een pennenzak werd oorspronkelijk uit leer vervaardigd, maar is tegenwoordig in allerlei materialen verkrijgbaar. De pennenzak is voorzien van een afsluitmechanisme, bijvoorbeeld een rits. Het heeft een handzaam formaat en is eenvoudig mee te nemen. Pennenzakken zijn beschikbaar in verschillende kleuren en worden soms ook van plaatjes voorzien.

Eigenschappen

kleur, lengte, hoogte, breedte, materiaal

Wat kan je ermee doen?

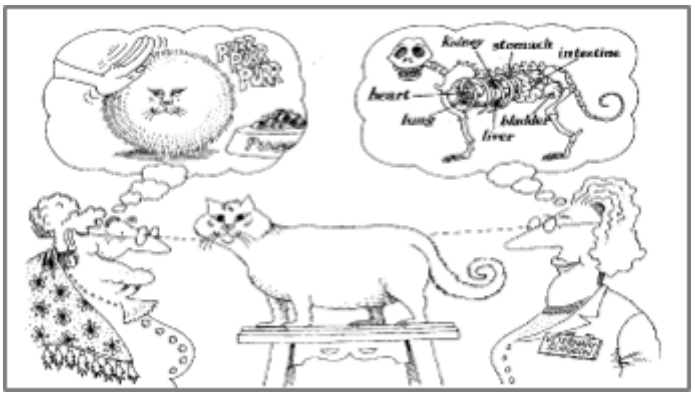
openen, sluiten, gooien, wassen, leegmaken, vullen

Deze real-world objecten

- kunnen **onderscheiden** worden van elkaar
- hebben **eigenschappen** die benoemd kunnen worden en die een waarde kunnen hebben
- hebben een **gedrag**. Dit omvat wat je kan doen met het object. Dit gedrag kan eventueel eigenschappen veranderen van waarde.

Software objecten zijn een weerspiegeling van deze real-world objecten: ze kunnen **onderscheiden** worden, ze hebben **eigenschappen** en ze hebben **gedrag**

2.2. Abstractie



Bij het ontwikkelen van software is abstractie een heel belangrijk concept. Abstractie verwijst

eigenlijk naar het enkel opnemen van de noodzakelijke eigenschappen en/of gedrag en tegelijk het verbergen van irrelevante details. Zo reduceren we de complexiteit van een applicatie en bevorderen we de efficiëntie ervan.

Een software object kan in de ene context andere eigenschappen en gedrag vertonen dan in een andere context.

Voorbeeld: pennenzak in 2 verschillende contexten

- **Applicatie 1: fabrikant van pennenzakken**

Belangrijk: Alle productiedetails en alle verkoopdetails zijn uitermate belangrijk. Van welk materiaal is de pennenzak gemaakt, hoeveel kan er in die pennenzak, hoeveel stof is er nodig om de pennenzak te vervaardigen, …

- **Applicatie 2: grafische, schematische voorstelling van een klas**

De details van een pennenzak zijn hier in deze context helemaal niet belangrijk. Hier is het belangrijk om te weten of de student al dan niet een pennenzak heeft en of die al dan niet op de tafel gelegd wordt.

2.3. Klasse

Via abstractie ontstaan soorten, types van objecten.

Soortgelijke objecten hebben

- dezelfde eigenschappen
- hetzelfde gedrag

Een **klasse** bevat de omschrijving van de **eigenschappen** en het **gedrag** van **soortgelijke objecten**

Klasse Pennenzak
Eigenschappen • kleur • open/dicht • lengte
Gedrag • geefKleur • maakOpen • vulMetPotlood

De klasse is een abstracte omschrijving van een pennenzak. Een object is een concrete pennenzak, conform de abstracte omschrijving.

Van die klasse kunnen we nu oneindige veel **instanties** maken. Een instantie is een synoniem voor een object. Bij het instantiëren van een klasse, maken we dus een object waarbij al zijn eigenschappen een concrete waarde krijgen. Eens het object gemaakt is, kan het dan van alles doen. Dit is dan het gedrag van dit object.

Tijdens **software ontwikkeling**

- wordt nagedacht over objecten uit de werkelijkheid, hun gemeenschappelijke eigenschappen en gedrag
- worden klassen ontworpen
- worden klassen geïmplementeerd

Tijdens **de uitvoering van software**

- komen objecten tot leven
- worden objecten gecreëerd, geconsulteerd, ze voeren operaties uit, werken samen, veranderen, voeren taken uit, worden vernietigd…

3. Klassen en objecten: in de diepte

3.1. Wanneer is iets een object?

In de OO-wereld is alles een object, een object wordt beschreven door een klasse, een object is een instantie van een klasse.

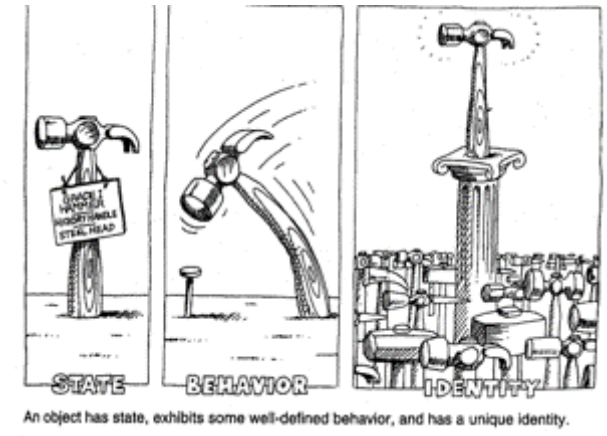
- iets **tastbaar**
  - een auto, een pennenzak, …
- een **concept** of **abstract ding**
  - een bankrekening, een klas, …

- een **organisatorische eenheid**
  - een departement, een manager, …
- een **interactie** die moet onthouden worden
  - een aankoop, …

Besluit: Voor om het even wat je wil voorstellen in je OO-applicatie heb je een object nodig.

3.2. Kenmerken van een object

Elk object heeft een **toestand**, een **gedrag** en een **identiteit**.



An object has state, exhibits some well-defined behavior, and has a unique identity.

We bekijken deze drie kenmerken aan de hand van een voorbeeld van een kauwgomautomaat. We proberen ook deze drie kenmerken zo goed mogelijk te beschrijven.

3.2.1. Toestand

We bekijken onderstaande objecten.

- Som een aantal mogelijke eigenschappen op.
- Wat is de huidige waarde van die eigenschap?
- Kan die waarde evolueren in de tijd?

- Kan die waarde van object tot object verschillen?

- Een aantal mogelijke eigenschappen
  - de kleur
  - de vorm
  - de hoogte, breedte, lengte
  - aantal kauwgomballetjes in de pot
- Huidige waarde van de eigenschappen
  - Kauwgomautomaat 1 heeft een rode kleur.
  - Kauwgomautomaat 3 is 1m hoog.
  - Kauwgomautomaat 2 bevat 63 kauwgomballetjes.
- Evolutie van de waarde van een eigenschap
  - De kleur kan veranderen, bv verbleken in de zon of kan herspoten worden.
  - Het aantal kauwgomballetjes verandert bij het vullen van de automaat. Het wijzigt ook wanneer iemand voldoende centen in de automaat stopt en draait aan de knop.
  - De hoogte, breedte en lengte blijven altijd dezelfde.
- Verschilt de waarde van een eigenschap van object tot object?
  - Kauwgomautomaat 3 is veel hoger dan kauwgomautomaat 1.
  - Kauwgomautomaat 2 bevat 63 kauwgomballetjes, terwijl kauwgomautomaat 1 leeg is.

We noteren deze eigenschappen nu op een meer gestructureerde manier.

- Eigenschappen worden voorgesteld door een **attribuut**.
- Elk attribuut heeft een **naam** en een **type**. De **naam** van een attribuut is in 1 woord geschreven (geen spaties) en begint altijd met een kleine letter. Het **type** bepaalt de mogelijke waarde voor die eigenschap.
  - boolean – ja/nee
  - int – gehele getallen
  - String – karakterstrings
  - double - kommagetallen
  - …

Attributen	
naam	**type**
kleur	String
vorm	String
hoogte	double

aantalKauwgomballen	int

SAMENVATTING

- Eigenschappen van objecten - **attributen** - worden omschreven in hun **klasse** met behulp van een naam én een type.
- Objecten zijn **instanties** van een **klasse** en hebben een **toestand**.
  - de eigenschappen van de klasse met een actuele waarde
  - het type van de actuele waarde moet overeenkomen met het type beschreven in de klasse
- Merk op dat wanneer software wordt uitgevoerd
  - de toestand van een object mogelijks evolueert met de tijd
  - de toestand mogelijks verschilt van object tot object

3.2.2. Gedrag

Een kenmerk van objecten is dat ze **gedrag** vertonen. Daaronder begrijpen we wat objecten **kunnen doen**. Dat gedrag zijn eigenlijk **diensten** die een object aanbiedt.

Objecten sturen boodschappen naar objecten

- Objecten vragen aan andere objecten om iets te doen.
- Zo doen objecten beroep op de dienst van andere objecten.

Objecten ontvangen boodschappen van objecten

- Objecten reageren op een vraag om iets te doen.
- De reactie bestaat uit het uitvoeren van het gepaste gedrag.

Het sturen en ontvangen van boodschappen noemen we communicatie. We bekijken onderstaande communicatie.

Een object vraagt aan de kauwgomautomaat-object om de hendel te draaien

- Formuleer de reactie van de kauwgomautomaat.
- Is de reactie afhankelijk van de toestand van het object?
- Heeft de reactie invloed op de toestand van het object?

- Reactie van de kauwgomautomaat

Als een ander object een boodschap "draai hendel" stuurt naar het kauwgomautomaat-object, dan reageert dit als volgt:

- Als er voldoende geld gegeven werd
- en als er nog kauwgomballetjes aanwezig zijn
- dan laat het automaat 1 kauwgomballetje uit zijn voorraad rollen.
    - Is de reactie afhankelijk van de toestand van het object?

Ja, het aantal kauwgomballetjes maakt deel uit van de toestand. Als er geen voorraad meer is, zal er geen kauwgombal uit het automaat rollen, anders wel. Dus de reactie is zeker afhankelijk van de toestand.

- Heeft de reactie invloed op de toestand van het object?

Ja, dat kan. Maar het hoeft niet altijd. Het kan zijn dat door de reactie het aantal kauwgomballetjes met 1 vermindert (= bij voldoende voorraad). Het kan ook zijn dat de toestand gelijk blijft, als de voorraad momenteel op 0 staat.

Dit voorbeeld illustreert wat we bedoelen met gedrag, met communiceren, met een dienst. "Draai hendel" is een voorbeeld van 1 dienst die het object kauwgomautomaat kan leveren. Net als bij de attributen proberen we nu een gestructureerde manier te vinden om dat gedrag netjes te noteren.

- Elke dienst die een object aanbiedt wordt omschreven via een **methode**, **operatie**
- Een methode beschrijft een **welomlijnde, duidelijk afgebakende taak**
- De **klasse** bevat de opsomming van alle methodes die objecten van die klasse aanbieden

Een methode bestaat uit een **signatuur** en een stuk **logica**

De signatuur van de methode

- stelt objecten in staat om op een correcte wijze boodschappen te versturen
- de signatuur bestaat uit
    - een naam
    - eventuele parameter(s)
    - eventueel returntype

Het stuk logica van een methode

- stelt objecten in staat om op een correcte wijze te reageren op boodschappen
- voorlopig gaan we niet verder inzoomen op deze logica, dit komt later aan bod

We zoomen hieronder in op de 3 onderdelen van de signatuur van een methode.

- een naam
    - Een naamkeuze is uitermate belangrijk tijdens het gehele software ontwikkelingsproces! Uit de naam die je bepaalt voor deze methode moet je perfect kunnen afleiden wat precies die

ene taak is waarvoor deze methode dient.
    - De naam moet een geldige identifier zijn (zie H1)
    - Aangezien een methode een actie voorstelt, zorgen we ervoor dat de naam start met de stam van een werkwoord. Zo benoem je die actie.
- eventuele parameters
    - Parameters laten toe extra informatie met een boodschap te versturen. De methode heeft die parameters nodig om zijn taak te kunnen uitvoeren. Die parameters worden dan gebruikt in het stuk logica van de methode (zie later).
    - Elke parameter heeft een naam en een type. We zorgen voor een betekenisvolle naam van de parameter. Zo weet je exact aan de naam welke informatie de parameter bevat. Het type legt de soort informatie vast, bv. een kommagetal, een stukje tekst, …
- een returntype
    - Een methode kan op het eind van de reactie een resultaat, een antwoord teruggeven.
    - Het returntype bepaalt de soort van de returnwaarde, bv. een kommagetal, een stukje tekst, …
    - Niet elke methode geeft iets terug aan het eind van de reactie. Indien de methode niets teruggeeft, is het returntype **void**. Dit betekent per definitie dat er geen antwoord terugkomt.

Gedrag		
naam	**parameters**	**returntype**
draaiHendel	geen	void

Er zijn twee soorten operaties/methoden die heel typisch zijn en vaak zullen voorkomen in klassen.

- operaties om de waarde van een welbepaald attribuut **op te vragen**, we noemen deze methoden de **getters**
- operaties om de waarde van een welbepaald attribuut **te wijzigen**, we noemen deze methoden de **setters**

De **GETTERS** laten ons toe de waarde van een welbepaald attribuut van een object **op te vragen**

- een getter retourneert steeds een waarde
- een getter zal de toestand van een object nooit veranderen
- conventie: de naam van dergelijke methodes start met get, samengesteld met de naam van het attribuut. In die samenstelling wordt de naam van het attribuut gestart met een hoofdletter (camelCasing).
  Kan ook starten met is, indien het attribuut van het type boolean is (bv. isVergrendeld()).

De **SETTERS** laten ons toe de waarde van een welbepaald attribuut van een object **te wijzigen**

- een setter retourneert nooit een waarde
- een setter zal de toestand van een object mogelijks veranderen

- de nieuwe waarde voor het attribuut wordt aangeleverd via een parameter
- conventie: de naam van dergelijke methodes start met set, samengesteld met de naam van het attribuut. In die samenstelling wordt de naam van het attribuut gestart met een hoofdletter (camelCasing).

Gedrag		
naam	**parameters**	**returntype**
draaiHendel	geen	void
getKleur	geen	String
setKleur	kleur van type String	void

Behalve getters en setters bevat een klasse meestal nog enkele **specifieke acties** die door de objecten van deze klasse moeten kunnen uitgevoerd worden. Deze acties kunnen

- de toestand van het object raadplegen en/of
- de toestand van het object veranderen en/of
- aanleiding geven tot creatie van nieuwe objecten
- communiceren met andere objecten via getters, setters en/of ander acties

Merk op hoe we door een zinvolle naamgeving reeds een goed beeld hebben van de reactie die bij deze methode hoort, zonder een omschrijving van de logica te zien.

Gedrag		
naam	**parameters**	**returntype**
draaiHendel	geen	void
getKleur	geen	String
setKleur	kleur van het type String	void
vulBij	aantal van het type int	void

SAMENVATTING

- het **gedrag** is een verzameling van diensten die een object aanbiedt
- de diensten zijn omschreven als **methodes** in een klasse
- objecten interageren met elkaar door **boodschappen** te versturen; we noemen dit ook het aanroepen van methodes
- wanneer een object een boodschap ontvangt wordt de **logica** van methode uitgevoerd
  - de logica maakt gebruik van de actuele toestand van het object
  - de logica kan eventueel op zijn beurt leiden tot het versturen van boodschappen waardoor een complexe interactie tussen vele objecten kan ontstaan
- door uitvoering van de logica kan **de toestand van het object wijzigen**
- de uitvoering van de logica **kan resulteren in een antwoord**, het object die de boodschap

verstuurde ontvangt dan dit antwoord

3.2.3. Identiteit

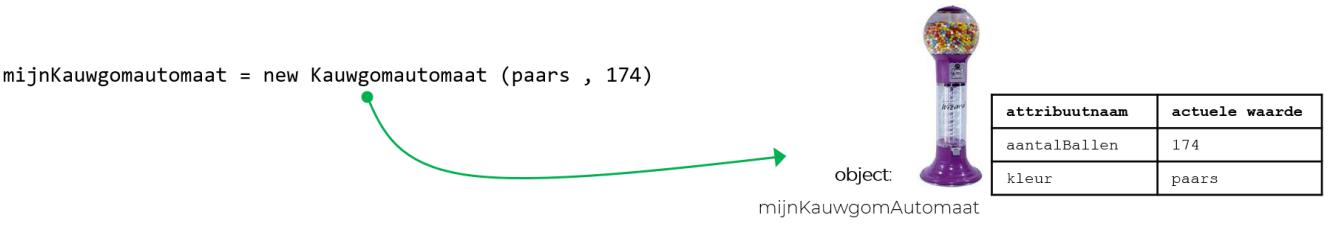
Ieder object heeft een unieke identiteit. Dit maakt het mogelijk om objecten van elkaar te onderscheiden. Twee objecten van eenzelfde klasse, met exact dezelfde waarden voor hun attributen, blijven via hun verschillende identiteit ook twee unieke objecten.

Een object verkrijgt zijn identiteit wanneer het geïnstantieerd wordt.

CONSTRUCTOREN laten toe objecten te instantiëren en te initialiseren.

- Net zoals een methode heeft een constructor een **signatuur**
  - **naam**: Deze kan je niet zelf kiezen, je moet de naam van de klasse gebruiken.
  - **parameters**: Deze kunnen helpen om een initieel geldige toestand te verzekeren.
  - Merk op dat de signatuur van een constructor **GEEN returntype** heeft.
- Net zoals een methode heeft een constructor een stuk **logica** Deze logica zorgt dat het nieuw geïnstantieerde object een geldige initiële toestand krijgt.

Het instantiëren van een klasse gebeurt door gebruik te maken van de **constructor** in combinatie met een speciale operator, meestal noemt dit de **new** operator. Als resultaat wordt een nieuw object, een nieuwe instantie, gecreëerd. De attributen van dit nieuw object kunnen worden ingesteld via de logica van de constructor.



Aangezien de signatuur van een constructor vrij gelijkaardig is aan die van methoden, kunnen we ook de constructor netjes noteren in onze tabel.

Methoden en constructoren		
naam	**parameters**	**returntype**
KauwgomAutomaat	kleur van het type String en aantalBallen van het type int	-

draaiHendel	geen	void
getKleur	geen	String
setKleur	kleur van het type String	void
vulBij	aantal van het type int	void

3.3. Verantwoordelijkheid van een object

Een **object** heeft een **duidelijke, wel afgebakende verantwoordelijkheid**, deze wordt gerealiseerd via

- de toestand, de eigenschappen van het object
- het gedrag, de diensten die het object aanbiedt.

Software heeft een **duidelijke, wel afgebakende verantwoordelijkheid**, het biedt een wel gedefinieerde functionaliteit aan, deze wordt gerealiseerd via

- de creatie van objecten
- een complexe interactie tussen de objecten.

Elke object draagt een deel van de verantwoordelijkheid voor het geheel

Eén van de grote uitdagingen bij OO-ontwerpen is het toekennen van de juiste **verantwoordelijkheden** aan de juiste soort objecten

3.4. Relaties tussen objecten

Tussen objecten bestaan relaties. Deze ontstaan wanneer één of meerdere attributen van een object, verwijzen naar een ander object. Dit is mogelijk door in een klasse eigenschappen op te nemen die als type een andere klasse hebben. We hebben tot nu toe gewerkt met primitieve types: int, double, boolean, … In volgende hoofdstukken zullen we dieper ingaan op relaties tussen objecten. De toestand van een object is op deze manier bepaald door de toestand van andere objecten. Objecten moeten elkaar kennen om beroep te kunnen doen op elkaar. Dit "kennen" kunnen we realiseren door relaties tussen objecten te definiëren.

Eén van de grote uitdagingen bij OO-ontwerpen is het definiëren van de juiste **relaties** tussen de juiste objecten

3.5. Objecten van eenzelfde soort

We zagen reeds het volgende:

Een **klasse** bevat de omschrijving van de **eigenschappen** en het **gedrag** van **soortgelijke objecten**

Dus de opgemaakte tabellen met de beschrijving van de eigenschappen en het gedrag, zijn eigenlijk ook de beschrijving van de klasse. Alles wat gedefinieerd wordt in een klasse, vinden we nu terug in de tabel.

Voorbeeld: KauwgomAutomaat

Klasse KauwgomAutomaat		
Attributen		
naam	**type**	
kleur	String	
vorm	String	
hoogte	double	
aantalKauwgomballen	int	
Methoden en constructoren		
naam	**parameters**	**returntype**
KauwgomAutomaat	kleur van het type String en aantalBallen van het type int	-
draaiHendel	geen	void
getKleur	geen	String
setKleur	kleur van het type String	void
vulBij	aantal van het type int	void

In het volgende deel bekijken we een meer professionele wijze om een klasse schematisch weer te geven.

4. Voorstelling klasse in UML

4.1. Inleiding

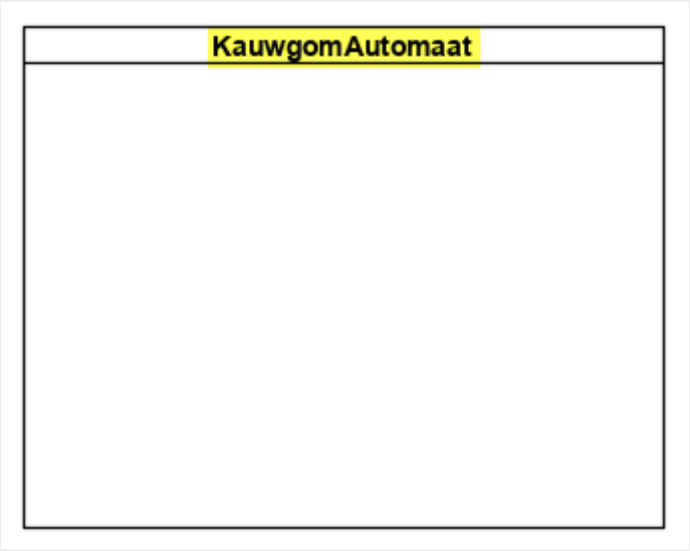
- Tijdens OO ontwerpen gaan we klassen vorm geven, gaan we **klassen definiëren**. Deze klassen zullen bepalen welk soort van objecten er kunnen gemaakt worden, welke verantwoordelijkheid ze dragen, en hoe ze gerelateerd zijn. We kunnen dit relatief taalonafhankelijk doen. Dit wil zeggen dat het "model" dat we bouwen in om het even welke OO programmeertaal kan geïmplementeerd worden.
- UML: Unified Modeling Language Dit is een **modelleertaal** die toelaat OO analyses en ontwerpen te maken. Het is een **grafische weergave** van bepaalde kenmerken van een OO systeem. We kunnen deze grafische voorstelling op papier tekenen of met behulp van specifieke software, zoals Visual Paradigm.
- In dit onderdeel bouwen we 1 klasse op in UML en deze klasse plaatsen we op een klassendiagram. In volgende hoofdstukken zal dit dan nog verder uitgroeien met meerdere klassen en onderlinge relaties tussen deze klassen.
- Een klasse wordt in UML voorgesteld door een rechthoek met maximaal 3 compartimenten. Er is altijd een klassenaam, in theorie kunnen de 2 andere delen eventueel ontbreken.

We overlopen hieronder elk "compartiment".

4.2. Klassenaam

- in het midden van het bovenste vakje van de rechthoek
- moet **betekenisvol** zijn, de naam vertelt ons reeds heel veel over de klasse!
- gebruik steeds een **zelfstandig naamwoord**
- gebruik steeds **enkelvoud**, nooit meervoud
- de naam van een klasse **start** steeds **met een hoofdletter**, en staat verder in **camel case**

Voorbeeld



4.3. Attributen

Voor elk attribuut definieer je onderstaande zaken:

visibiliteit

Hiermee geven we aan of andere objecten rechtstreeks dit attribuut kunnen raadplegen en/of wijzigen. Een attribuut mag nooit rechtstreeks wijzigbaar zijn, we schermen dit altijd af. De visibiliteit van een attribuut is **private**. Later wordt dit nog duidelijker!

naam

Altijd **startend** met een **kleine letter**, gevolgd door camel case. Het is essentieel om **betekenisvolle** namen te gebruiken die ondubbelzinnig verduidelijken wat het attribuut voorstelt

datatype

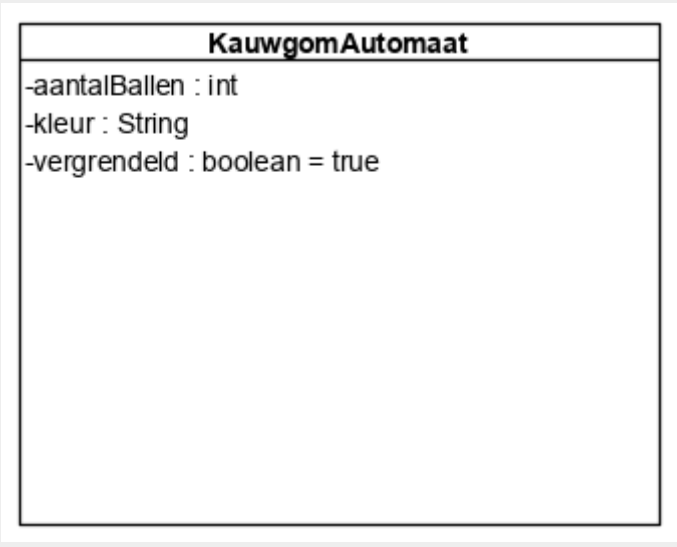
Hiermee geven we aan welk soort gegevens aan dit attribuut kunnen worden toegekend

standaardwaarde

De waarde die het object automatisch krijgt bij instantiatie, dit is optioneel te vermelden

Voorbeeld

- Elk kauwgomautomaat-object heeft een eigenschap genaamd **aantalBallen**.
- Dit is een **private** attribuut (-) wat betekent dat andere objecten dit attribuut niet kunnen raadplegen of wijzigen
- in dit attribuut kunnen we **een geheel getal** (integer) plaatsen (int)
- voor dit attribuut is niet expliciet een standaardwaarde gedefinieerd, de impliciete standaardwaarde voor gehele getallen is 0
- Het attribuut vergrendeld kreeg wel een **standaardwaarde**, namelijk true.



4.4. Operaties (=methodes)

Voor elke operatie leg je de signatuur als volgt vast:

visibiliteit

Hiermee geven we aan of andere objecten beroep kunnen doen op dit gedrag.

naam

- **startend** met een **kleine letter**, gevold door camel case
- start steeds met de **stam van een werkwoord**
- het is essentieel om **betekenisvolle namen** te gebruiken die de intentie van de operatie verduidelijken

parameters

Hiermee geven we aan welk soort gegevens moeten aangeleverd worden om de operatie zinvol te kunnen uitvoeren. Een methode kan parameters hebben, maar dat is niet verplicht.

datatype terugkeerwaarde

Hiermee wordt aangegeven welk soort gegevens de operatie kan retourneren. Wanneer een operatie wordt aangeroepen dan kan deze als antwoord een resultaat naar de aanroeper retourneren.

We leerden twee vaak voorkomende methodes kennen: de **getters** en de **setters**.

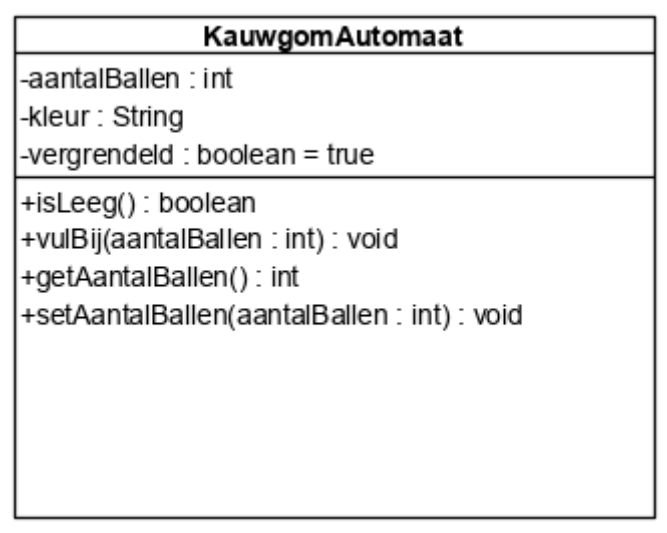
De getter

- Via een getter ga je de waarde van een welbepaald attribuut opvragen
- De signatuur van een getter leid je automatisch af uit het te bevragen attribuut
  - visibiliteit: meestal + (public), je maakt een getter zodat een ander object iets kan opvragen.
  - naam: get<NaamBevraagdeAttribuut>
  - parameters: geen
  - datatype terugkeerwaarde: datatype bevraagde attribuut

De setter

- Via een setter ga je een waarde toekennen aan een welbepaald attribuut
- De signatuur van een setter leid je automatisch af uit het attribuut waaraan je een waarde wil toekennen
  - visibiliteit: als het puur voor intern gebruik in de klasse is, schermen we de methode af (-), als een ander object een nieuwe waarde moet kunnen doorgeven om een attribuut mee in te stellen zorgen we dat deze setter zichtbaar is (+). Of je nu kiest voor - of +, daar komen we later op terug.
  - naam: set<NaamBevraagdeAttribuut>
  - parameters: de waarde die je aan het attribuut wenst te geven, exact 1 parameter dus.
  - datatype terugkeerwaarde: void

Voorbeelden



- **Voorbeeld 1**: operatie isLeeg
  - Aan elke kauwgomautomaat-object kan je vragen of die leeg is via de operatie genaamd isLeeg
  - Dit is een publieke operatie (+) wat betekent dat andere objecten deze operatie mogen gebruiken
  - Als je deze operatie gebruikt (~aanroept) hoef je geen extra informatie door te geven, de parameterlijst van de methode is leeg (). Je ziet enkel een rond haakje open en dicht staan.
  - Als je deze operatie aanroept krijg je een antwoord van het type boolean
- **Voorbeeld 2**: operatie vulBij
  - Aan elke kauwgomautomaat-object kan je vragen om zich bij te vullen via de operatie genaamd vulBij
  - Dit is een publieke operatie (+) wat betekent dat andere objecten deze operatie mogen gebruiken
  - Als je deze operatie gebruikt (~aanroept) moet je extra informatie doorgeven, namelijk het aantal ballen waarmee het kauwgomautomaat-object wordt bijgevuld. De parameterlijst bevat 1 parameter genaamd aantalBallen en deze is van het datatype int (aantalBallen: int)
  - als je deze operatie aanroept krijg je geen antwoord, dit geven we aan met void
- **Voorbeeld 3**: een getter
  - De getter waarmee we aan een kauwgomautomaat-object kunnen vragen wat de waarde is van zijn attribuut genaamd aantalBallen noemt getAantalBallen
  - Een getter heeft altijd een lege parameterlijst: (). Een getter heeft nooit nood aan extra informatie, alle informatie om de operatie uit te voeren is immers aanwezig in het object!
  - De getter retourneert een int. Een getter zal per definitie steeds een waarde retourneren, het datatype van deze returnwaarde is altijd gelijk aan het datatype van

het bevraagde attribuut

Via een getter verkrijg je informatie over een object en kan de toestand van een object niet wijzigen.

- **Voorbeeld 4**: een setter
  - De setter waarmee we een waarde kunnen toekennen aan het attribuut genaamd aantalBallen van een kauwgomautomaat-object noemt setAantalBallen
  - Een setter heeft altijd een parameterlijst met 1 parameter: (aantalBallen: int). Het datatype van deze parameter is steeds gelijk aan het datatype van het attribuut waaraan we een waarde willen toekennen. Ook de naam van de parameter komt overeen met de naam van het attribuut waaraan we een waarde willen toekennen.
  - Een setter heeft nooit returnwaarde: void

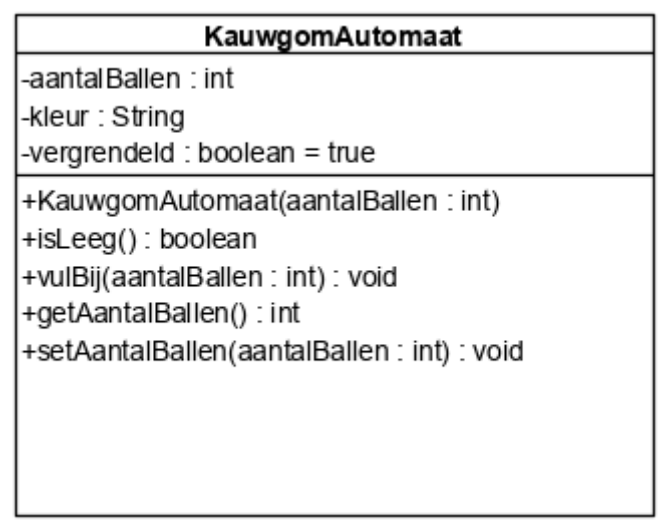
Via een setter wijzig je de toestand van een object en verkrijg je geen informatie over het object.

4.5. Constructoren

- De constructoren worden eveneens in het vakje met operaties vermeld, al behoren constructors niet tot het gedrag van een object.
- Ze zijn als volgt opgebouwd
  - naam: is identiek aan de naam van de klasse, inclusief de hoofdletter. Merk op dat in het vak met operaties alleen de constructoren met een hoofdletter beginnen.
  - parameters: hiermee kunnen we waarden aanleveren die kunnen helpen om het geïnstantieerde object te initialiseren in een geldige toestand.

Voorbeeld

- Via de constructor genaamd Kauwgomautomaat kunnen we kauwgomautomaat-objecten instantiëren.
- De constructor heeft 1 parameter (aantalBallen: int). Via deze parameter kunnen we aangeven hoeveel balletjes er in het nieuw Kauwgomautomaat-object zullen zitten.



4.6. Een extraatje: overloading

Overloading is een OO concept. Je kan dit hebben bij methodes én bij constructoren: **meerdere methodes/constructoren** in eenzelfde klasse kunnen **dezelfde naam** hebben **maar** dan moet de **lijst van parameters** wel **verschillen**.

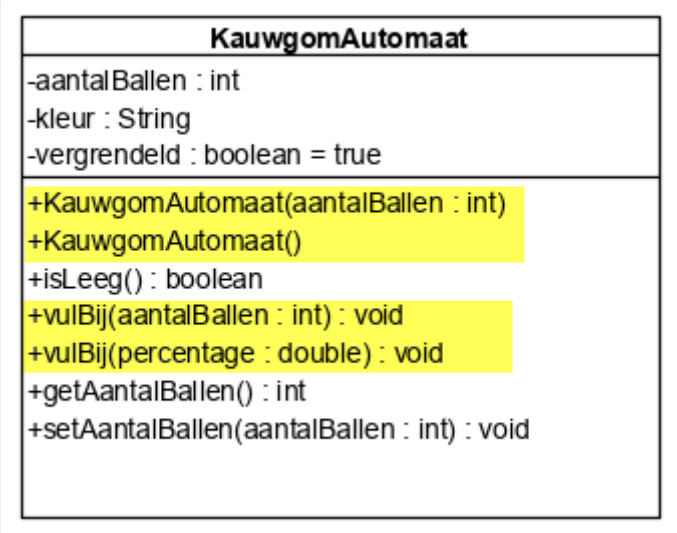
Een lijst van parameters kan verschillend zijn op volgende manieren:

- verschillend aantal parameters
- verschillende datatypes van parameters
- verschillende volgorde van datatypes van parameters

Voorbeeld

2 constructoren in een klasse

In de klasse KauwgomAutomaat voorzien we 2 constructoren. Aangezien een constructor dezelfde naam heeft als de klasse, hebben die 2 constructoren dus dezelfde naam. Doordat er 2 constructoren zijn, kan je op 2 verschillende manieren een object aanmaken en initialiseren. Bij de constructor met 1 parameter geef je het aantal balletjes mee, het attribuut aantalBallen zal dus geïnitialiseerd worden op deze waarde. De constructor zonder parameters, die vaak de naam default constructor krijgt, maakt een object waarbij aantalBallen automatisch op 0 wordt geïnitialiseerd. Deze automaat is dus leeg als het gecreëerd wordt.



2 methodes met de naam vulBij

Een kauwgomautomaat-object kan op 2 verschillende manieren bijgevuld worden. Als de methode vulBij aangeroepen wordt met een geheel getal als parameter (bv waarde 50), dan komen er 50 extra balletjes bij de huidige vooorraad. Als de methode vulBij aangeroepen wordt een een kommagetal als parameter (bv 0.50), dan wordt de automaat tot de helft gevuld (voor 50%).

EXTRA: gebruik van deze constructoren en methodes (zie ook 6.4)

```
KauwgomAutomaat ka = new KauwgomAutomaat(100);
KauwgomAutomaat ka2 = new KauwgomAutomaat();
```

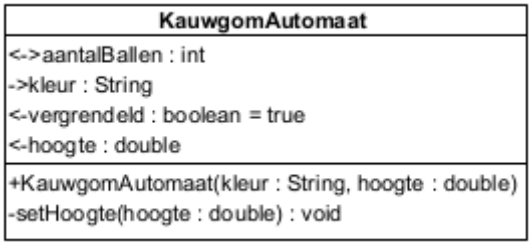
Met de operator new gevolgd door de naam van de constructor maken we een nieuw object aan. We kennen dit toe aan een referentievariabele om er dan later mee te kunnen communiceren. De ene keer geven we een waarde mee tussen de ronde haakjes van de constructor, de andere keer niet. Volgens het aantal waarden dat je meegeeft, wordt nu beslist welke constructor uitgevoerd zal worden.

```
ka2.vulBij(20);
ka2.vulBij(0.50);
```

Via de referentievariabele kunen we communiceren met het object. Een voorbeeld van communicatie is het bijvullen van het toestel door de methode vulBij aan te roepen. In dit voorbeeld zijn er 2 dergelijke methodes: afhankelijk van het datatype van de concrete waarde die je doorgeeft, wordt de ene of de andere methode uitgevoerd.

4.7. Een extraatje: weergave getters en setters in UML

Hoe een getter en setter eruit zien, ligt per definitie vast. Daarom is het niet altijd zinvol om deze methodes weer te geven in UML, ze nemen nogal veel plaats in. Maar het is wel belangrijk om te weten welke getters en setters precies aanwezig zijn in de code. Niet elk attribuut moet een getter en setter hebben.



In Visual Paradigm kan je door middel van pijltjes bij het attribuut aangeven of er een public getter en/of een public setter is.

- Een pijl naar rechts (wijst naar het attribuut) : je kan het attribuut een waarde geven, er is een public setter
- Een pijl naar links (wijst weg van het attribuut) : je kan de waarde van het attribuut opvragen, er is een public getter
- Ook de combinatie van een pijl naar links én naar rechts is mogelijk: er is dan zowel een public getter als een public setter
- Zelfs een attribuut zonder pijltjes is theoretisch mogelijk: er is geen public setter en geen public getter

Op de figuur hierboven zie je de verschillende mogelijkheden:

- Attribuut aantalBallen: je ziet voor de naam van het attribuut zowel een pijltje naar links als naar rechts, dus dit attribuut heeft zowel een public getter als public setter
- Attribuut kleur: je ziet enkel een pijl naar rechts, er is enkel een public setter terug te vinden.
- Attributen hoogte en vergrendeld: je ziet enkel een pijltje naar links, er is dus een public getter voor beide attributen. MAAR het attribuut hoogte heeft ook een private setter. Dat kan je niet weergeven met een pijltje, de private setter zie je dus als volledige methode verschijnen in de UML. Het attribuut vergrendeld heeft in dit voorbeeld geen setter.
- Extra: een private getter heeft geen zin. In de klasse zelf heb je rechtstreeks toegang tot het attribuut, daarvoor hoef je geen getter te maken. Als van buitenaf geen toegang nodig is tot een getter, dan heb je gewoon GEEN getter.

5. Implementatie klasse, gedefinieerd in UML

Eens een klasse gedefinieerd is in UML kunnen we die nu gaan omzetten in javacode. We vertrekken van onderstaande UML. Van elk onderdeel bespreken we hoe dit omgezet wordt in javacode.

5.1. De klasse

In UML tekenen we de klasse met behulp van 1 rechthoek. Deze rechthoek vertalen we naar 1 tekstbestand met als naam KauwgomAutomaat.java, in dat tekstbestand vinden we onderstaande code:

```
public class KauwgomAutomaat {

}
```

public

Deze klasse is zichtbaar voor alle andere klassen in de JVM.

class

Gereserveerd woord in java om een klasse te definiëren.

KauwgomAutomaat

Naam van de klasse, begint ALTIJD met een hoofdletter.

⇒ Deze drie woorden samen vormen de **signatuur** (=definitie) van de klasse.

{}

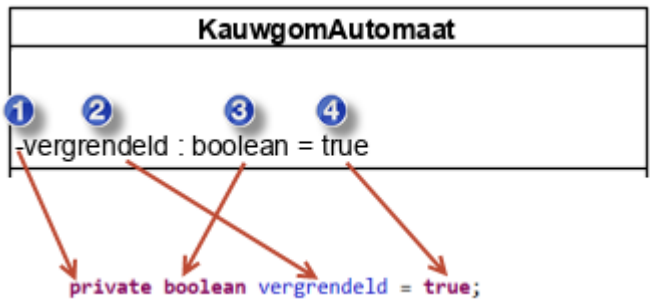
Alles wat verder in de klasse gedefinieerd wordt, zoals de attributen, constructoren en operaties, komt tussen de openingsaccolade en eindaccolade. Dit noemen we de **body** van de klasse.

5.2. De attributen

Hieronder zie je aan de hand van een voorbeeld hoe je vertrekkend vanuit de UML de vertaalslag naar Java kunt doen. Voor elk attribuut volg je hetzelfde stramien.

UML		Java	
1	- (visibiliteit)	private	1
2	naam	vergrendeld	3

	:	niet in code	
3	type	boolean	2
4	= defaultwaarde	= true	4




```
        private int aantalBallen;
        private String kleur;
        private boolean vergrendeld = true;
```

Als je meerdere attributen van **hetzelfde datatype** hebt, kan je een **verkorte schrijfwijze** gebruiken. In de javacode noteer je eerst de toegankelijkheid, het datatype en daarna ga je telkens met komma's ertussen de namen van de attributen opsommen.

Een attribuut krijgt soms ook de naam **instantievariabele**. Letterlijk betekent dit eigenlijk: een variabele gekend in de volledige instantie. Elke methode die je hebt in de klasse heeft toegang tot zo'n variabele, de variabele is gekend in elke methode van die klasse.

Voorbeeld:

```
private String kleur;
private String model;
```

kan je noteren als

```
private String kleur,model;
```

5.3. De getters en setters

Getters en setters zijn methodes met een specifieke verantwoordelijkheid, die heel vaak voorkomen. Om de vertaalslag naar javacode te maken, volgen we eigenlijk de algemene regels om een signatuur van een methode om te zetten naar javacode.

1. De signatuur van een methode omzetten naar java (geldig voor zowel getter als setter)

Algemene structuur van de UML

toegankelijkheid methodenaam (parameter:type, …) : returntype
Vertaling in java

toegankelijkheid returntype methodenaam (type parameternaam, …) {
 	
}

Aandachtspunten bij het vertalen van de signatuur naar javacode:

- In de UML staat returntype helemaal achteraan, in javacode komt dit onmiddellijk na de toegankelijkheid.
- Bij de definitie van een parameter in de parameterlijst is de volgorde van type en naam in javacode omgekeerd aan de volgorde in de UML. Het dubbelpunt zie je niet terugkeren in de javacode.
- De toegankelijkheid van een methode bepaalt wie met de methode aan de slag kan.
  - Toegankelijkheid in UML wordt aangegeven door een minteken (-): deze methode kan enkel intern in de klasse gebruikt worden. Het minteken wordt vertaald door het woord **private** in javacode.
  - Toegankelijkheid in UML wordt aangegeven door een plusteken (+): deze methode kan door elk object aangeroepen worden. Het plusteken wordt vertaald door het woord **public** in javacode.
- In de UML vind je enkel de signatuur van een methode. In de javacode vind je een vertaalslag van die signatuur maar ook de implementatie van de methode zelf. Dit noemen we de body van de methode en die staat altijd tussen accoladetekens ({ }). Voor elke accolade-open ({) moet je een een accolade-dicht terugvinden. (})

Voorbeeld van een getter en een setter:

```
    public int getAantalBallen(){

    }

    public void setAantalBallen(int aantalBallen){

    }
```

2. De methode zelf implementeren

Bij het implementeren van een methode moet je goed weten wat de functionaliteit is van deze methode. Wat moet er gebeuren als deze methode aangeroepen wordt? Wat is de reactie die je verwacht van deze methode? Die reactie moet in java instructies omgezet worden.

- Getter

Functionaliteit: een getter geeft de huidige waarde van één attribuut van het object terug.

De attributen zijn reeds gedefinieerd in de klasse. We pikken er een voorbeeld uit: attribuut aantalBallen. In de klasse is dit attribuut overal beschikbaar. Dat wil zeggen dat als je in de body van een methode - zoals hier in de getter - wil weten hoeveel balletjes aanwezig zijn in het object dat je dit attribuut kan aanspreken via de naam aantalBallen. De waarde van aantalBallen moet teruggegeven worden in deze methode. Om iets terug te geven (als antwoord), gebruiken we in java het woord **return**. Als we de instructie return tegenkomen, verlaten we de methode onmiddellijk, de waarde die na return komt, stuur je als antwoord terug.

De volledige instructie ziet eruit als volgt:

```
    return aantalBallen;
```

Dan wordt de volledige getter (signatuur + implementatie) als volgt uitgeschreven:

```
    public int getAantalBallen(){
        return aantalBallen;
    }
```

- Setter

Functionaliteit: een setter geeft - indien mogelijk - een nieuwe waarde aan één attribuut van het object.

De parameter van de methode bevat de nieuwe waarde die we willen toekennen aan het overeenkomstige attribuut.

Een toekenning in java ziet er algemeen uit als volgt:

```
naam van variabele waar je nieuwe waarde wil aan toekennen = nieuwe waarde;
```

We proberen dit toe te passen:

We willen een nieuwe waarde toekennen aan ons attribuut aantalBallen.

```
aantalBallen =
```

Na het gelijkteken komt de nieuwe waarde die we willen toekennen, dat is de waarde die doorgegeven wordt via de parameter aantalBallen.

```
aantalBallen = aantalBallen;
```

Dit is nu een vreemde instructie. Voor het gelijk-aan-teken staat het **attribuut** aantalBallen, na het gelijk-aan-teken staat de **parameter** aantalBallen. Dit zijn twee verschillende variabelen, dit zullen we duidelijk moeten maken in deze instructie.

Als een parameter én een attribuut een gelijke naam hebben - wat je heel vaak ziet bij een setter - dan maken we een onderscheid tussen deze twee variabelen door voor het attribuut **this.** te gebruiken. De betekenis van **this.** kan je als volgt interpreteren: we nemen de variabele aantalBallen **van het object zelf**, en dat is dan het attribuut. Dit is de eigenschap van het object. Een parameter is maar een waarde die doorgegeven wordt, dit is geen onderdeeltje van het object zelf.

De instructie wordt dus

```
this.aantalBallen = aantalBallen;
```

Dan wordt de volledige setter (signatuur + implementatie) als volgt uitgeschreven:

```
    public void setAantalBallen(int aantalBallen){
        this.aantalBallen = aantalBallen;
    }
```

3. Controles

De verantwoordelijkheid van een setter is een nieuwe waarde toekennen aan één attribuut, MAAR alleen maar als dit een toegelaten waarde is.

Een voorbeeld: aantalBallen is een attribuut van het KauwgomAutomaat-object. Dit attribuut is van het type int. Dit stelt een geheel getal voor. De range van dit geheel getal is [-2147483648,2147483647]. Alle waarden uit dit interval kan je doorgeven aan een parameter van het type int. Maar dat zijn niet allemaal goede waarden voor het attribuut aantalBallen. Een negatieve waarde voor dit attribuut, heeft helemaal geen betekenis. In het kauwgomautmaat kunnen maximaal 149 balletjes, dan is het volledig gevuld. Dus alle waarden die groter zijn, kunnen ook niet.

In de setter moeten we dus testen of de nieuwe waarde die binnenkomt via de parameter, wel een geldige waarde is. Enkel en alleen als het een geldige waarde is zal die waarde toegekend worden aan het attribuut.

Om die controles te schrijven kan je alle controlestructuren gebruiken die je leerde kennen in hoofdstuk 2.

```
        private void setAantalBallen(int aantalBallen) {
            if (aantalBallen >= 0 && aantalBallen < 150)
                this.aantalBallen = aantalBallen;
        }
```

4. Opgelet met public setter

Een setter kan afhankelijk van de situatie public of private zijn. Als een setter private is - waar we standaard vanuit gaan - dan kan die setter alleen maar vanuit de klasse zelf aangeroepen worden. Dit betekent bijvoorbeeld, dat de constructor van de klasse gebruik kan maken van de setter.

Soms is er functionaliteit nodig dat een ander object - van buiten af dus, die setter moet kunnen aanroepen. Dan kunnen we die setter niet afschermen, maar moeten we die toegankelijk maken voor het andere object. Dan wordt de setter bijvoorbeeld public.

Public setters die ook aangeroepen worden vanuit de constructor kunnen later voor problemen zorgen bij overerving (zie hoofdstuk 7). Om deze problemen te voorkomen maken we een public setter altijd **final**.

```
    public final void setVergrendeld(boolean vergrendeld) {
        this.vergrendeld = vergrendeld;
    }
```

5. Voorbeelden

```
        public int getAantalBallen() {
            return aantalBallen;
        }

        private void setAantalBallen(int aantalBallen) {
            if (aantalBallen >= 0 && aantalBallen < 150)
                this.aantalBallen = aantalBallen;
        }

        public String getKleur() {
            return kleur;
        }

        private void setKleur(String kleur) {
            this.kleur = kleur;
        }

        public boolean isVergrendeld() {
            return vergrendeld;
        }

        public final void setVergrendeld(boolean vergrendeld) {
            this.vergrendeld = vergrendeld;
        }
```

Opmerking:

In de voorbeelden hierboven zie je de methode **isVergrendeld**. De implementatie ziet eruit als de implementatie van een getter, de waarde van het attribuut vergrendeld wordt teruggegeven. De naam van een getter wordt altijd samengesteld door get + naam van het attribuut. Op deze regel is één uitzondering, nl. als het attribuut van het datatype boolean is. Dan wordt in plaats van **get** begonnen met **is**.

5.4. De constructoren

Een constructor bestaat net zoals methodes uit een signatuur, vastgelegd in de UML, én een implementatie.

De implementatie zorgt ervoor dat er een nieuw object wordt aangemaakt in een geldige toestand. Een geldige toestand betekent dat elk attribuut een geldige, toegelaten waarde moet krijgen.

De parameters van een constructor dienen om attributen correct te kunnen initialiseren.

In de UML zien we 2 constructoren: 1 met 1 parameter - aantalBallen - én 1 zonder parameters.

We starten met de **constructor met 1 parameter**. De parameter aantalBallen dient om het attribuut aantalBallen correct in te stellen. Hierboven bij de implementatie van de setter zagen we dat er een controle moet gebeuren op die waarde, nl. aantalBallen mag niet negatief zijn en mag maximaal 149 zijn. Ook bij de constructor moet dat dus gebeuren, maar een zelfde controle implementeren we liefst maar 1 keer, we willen geen dubbele code in de klasse. De code die we schreven in de setter kunnen we hergebruiken in de constructor door die setter aan te roepen vanuit de implementatie van de constructor.

```
        public KauwgomAutomaat(int aantalBallen) {
            setAantalBallen(aantalBallen);
            setKleur("rood");
        }
```

Een methode die in de klasse zelf staat kan je gewoon aanroepen vanuit een constructor of een andere methode, door de naam te noteren en de juiste waarden door te geven voor de gedefinieerde parameters. In hoofdstuk 6 komen we hier nog uitgebreid op terug.

Wat als de parameter aantalBallen negatief is of groter dan 149?

We roepen de constructor aan met als waarde 160 voor de parameter aantalBallen. Met die waarde 160 wordt de setter aangeroepen. Dat wil zeggen dat we de implementatie van de constructor verlaten en springen naar de implementatie van de setter. Daar wordt gedetecteerd dat 160 groter is dan 149, dus de waarde van de parameter wordt NIET toegekend aan het attribuut. Als alle code van de setter is uitgevoerd, springen we terug naar de implementatie van de constructor en doen daar verder met de rest van de implementatie. Het attribuut aantalBallen kreeg geen nieuwe waarde in de setter. Dat wil zeggen dat het attribuut op een **defaultwaarde** zal ingesteld worden. De defaultwaarde wordt bepaald door het datatype van het attribuut.

datatype	**defaultwaarde attribuut**
boolean	false
byte	0
short	0
int	0
long	0

char	0
float	0.0
double	0.0

Heel concreet betekent het nu dat als we de constructor met 1 parameter aanroepen en waarde 160 doorgeven dat een object wordt aangemaakt in volgende toestand:

attribuut	**waarde**	**verklaring**
aantalBallen	0	De waarde die doorgegeven wordt, nl. 160, is geen toegelaten waarde, het attribuut blijft zijn defaultwaarde 0 - voor type int - behouden.
kleur	"rood"	Als er geen kleur wordt doorgegeven als parameter, wat hier het geval is, dan wordt de kleur altijd op rood ingesteld. Hierop zit geen extra controle in de setter, dus dat lukt altijd.
vergrendeld	true	De defaultwaarde voor een boolean is false, maar bij de declaratie van het attribuut werd vastgelegd dat dit attribuut altijd de waarde true krijgt. Aangezien de implementatie van de constructor niets anders meer doet voor dit attribuut, zal de waarde dus true zijn.

```
        public KauwgomAutomaat(int aantalBallen) {
            setAantalBallen(aantalBallen);
            setKleur("rood");
        }

        public KauwgomAutomaat() {
            this(0);
        }
```

Volgens de code maakt de defaultconstructor hier telkens rode, lege kauwgomautomaten die vergrendeld zijn.

Wat als er **geen constructor gedefinieerd** wordt in de UML van de klasse? Kan je dan geen objecten maken van die klasse?

```
Als je zelf geen constructor definieert, is er sowieso een defaultconstructor
beschikbaar. Dit wil zeggen dat je de constructor zonder parameters kan aanroepen. Die
zal alle attributen instellen op hun defaultwaarde volgens datatype of de initiële
waarde die gedefinieerd staat bij de declaratie van het attribuut. Onthoud dat van
zodra er een constructor gedefinieerd wordt, die "automatische" defaultconstructor
verdwijnt.
```

Opgelet:

Als je de implementatie van de code van de 2 constructoren die we nu hebben, zie je bv. dat het instellen van de kleur op rood bij allebei aanwezig is. Als er meerdere constructoren zijn in 1

klasse, heb je vaak dezelfde code die in meerdere constructoren voorkomt. Eigenlijk willen we dat maar 1 keer schrijven.

Met behulp van het woord **this(...)** zullen we de code, geschreven in een andere constructor kunnen hergebruiken.

We kijken naar een voorbeeld:

```
        public KauwgomAutomaat(int aantalBallen) {
            setAantalBallen(aantalBallen);
            setKleur("rood");
        }

        public KauwgomAutomaat() {
            this(0);    ①
        }
```

① Met het woord this onmiddellijk gevolgd door ronde haakjes - al dan niet met iets tussen - roepen we een constructor aan die in dezelfde klasse staat. Hier staat 1 waarde tussen de ronde haakjes, nl. een geheel getal. We roepen dus de constructor aan met 1 parameter van het type int. Met die 0 drukken we uit dat er een lege kauwgomautomaat gemaakt wordt, en verder zal die kauwgomautomaat een rode kleur krijgen.

Het is "good practice" om zo weinig mogelijk gedupliceerde code te hebben. Probeer code die je al geschreven hebt, te herbruiken waar mogelijk.

5.5. Andere methodes

Elke methode heeft een signatuur en een implementatie.

De algemene regel om de signatuur, gedefinieerd in UML, te vertalen naar javacode wordt uitgelegd bij het onderdeel "getters en setters". Diezelfde regel kan je voor elke andere methode volledig volgen: zie getters en setters

Om de methode te implementeren moet je heel goed de functionaliteit van de methode kunnen verwoorden. Daarna probeer je die om te zetten naar javacode en daarvoor kan je alle bouwstenen die je zag in hoofdstuk 2 gebruiken.

Twee concrete voorbeelden:

1. De methode **isLeeg** moet antwoord geven op de vraag of dit KauwgomAutomaat-object leeg is. Als die vraag binnenkomt bij het object, dan moet het object eigenlijk alleen maar controleren of zijn attribuut aantalBallen gelijk is aan 0 of niet. Het resultaat van deze vergelijking is meteen ook het antwoord dat de methode moet terugsturen.

```
        public boolean isLeeg() {
                return aantalBallen == 0;
        }
```

2. De methode **vulBij** krijgt een aantalBallen door via de parameter. Dit is het aantalBallen dat we willen toevoegen aan het object. Het bijvullen kan alleen maar als het kauwgomautomaat NIET vergrendeld is. Bijvullen betekent dan dat het doorgegeven aantal bovenop het huidige aantal komt. Die som wordt het nieuwe aantalBallen in het automaat, tenzij we het maximum overschrijden. Als het maximum overschreven zou worden, verandert er niets. Het object wordt niet bijgevuld. Deze functionaliteit bekomen we door de setter aan te roepen - deze controleert al op die maximumgrens - en we geven de som van het huidige aantalBallen met de extra ballen die toegevoegd zullen worden door.

```
        public void vulBij(int aantalBallen) {
                if (!vergrendeld && aantalBallen >=0)
                        setAantalBallen(this.aantalBallen + aantalBallen);
        }
```

5.6. Voorbeeld UML met pijltjes bij attributen en bijhorende code

Als je de UML van de klasse krijgt, dan kan je de implementatie (=javacode) volledig uitwerken. Als je de code van de klasse krijgt, dan kan je perfect de voorstelling in UML tekenen. Deze1-op-1 match werkt in beide richtingen.

6. Bouwen van een applicatie m.b.v. die klasse

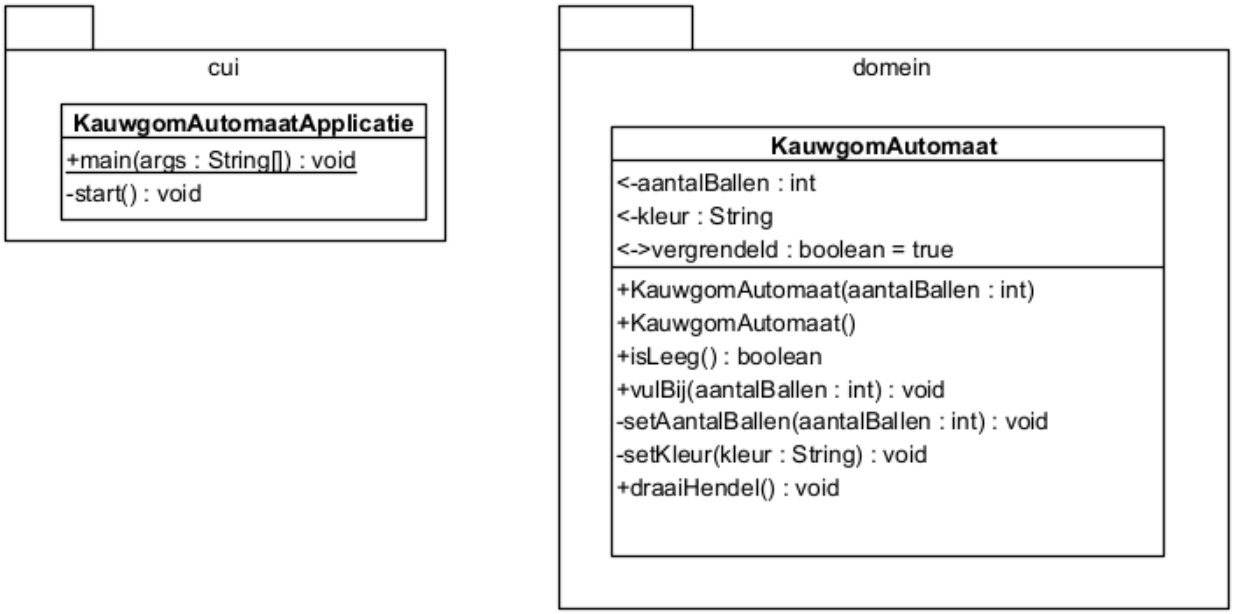
Een applicatie bouwen met behulp van die klasse betekent eigenlijk dat we een uitvoerbaar programma (=applicatie) zullen bouwen waarbij we gebruik zullen maken van objecten van de klasse KauwgomAutomaat.

6.1. Functionaliteit van de applicatie

- Maak 2 kauwgomautomaten aan: de ene automaat is een rode kauwgomautomaat die momenteel nog leeg is. De andere kauwgomautomaat is ook rood maar is al onmiddellijk gevuld met 50 balletjes.
- Druk van beide kauwgomautomaten de kleur én het aantal balletjes af.
- Ontgrendel de eerste kauwgomautomaat.
- Vul die bij met 20 balletjes.
- Vergrendel de kauwgomautomaat opnieuw.

- Druk van beide kauwgomautomaten de kleur én het aantal balletjes af.
- Vul diezelfde kauwgomautomaat nog eens bij met 75 balletjes (in vergrendelde toestand).
- Druk van beide kauwgomautomaten de kleur én het aantal balletjes af. Wat stel je vast?

6.2. UML van de applicatie



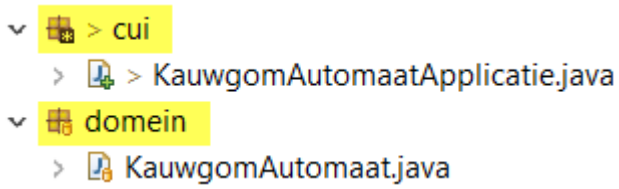
In deze applicatie hebben we nu 2 klassen: uiteraard de klasse KauwgomAutomaat maar aangezien we ook een uitvoerbare applicatie nodig hebben, hebben we ook een klasse met een main-methode nodig.

De klasse KauwgomAutomaatApplicatie bevat de flow van ons programma, zal de nodige objecten aanmaken en gegevens afdrukken op het scherm zodat we iets te zien krijgen.

De klasse KauwgomAutomaat bevat enkel de functionaliteit van zo'n kauwgomautomaat, niets meer, niets minder.

De taken van deze klassen zijn totaal verschillend. Om structuur in onze applicatie te stoppen gaan ze elke klasse in een aparte package plaatsen. Een **package** is een groepering van klassen met gelijkaardige functionaliteiten. De naam van een package wordt in java altijd met kleine letters geschreven.

De twee packages die hier gebruikt worden zijn **cui** en **domein**.



- **cui** staat voor **console user interface**. Deze package verzamelt klassen die "communiceren" met de gebruiker. In ons concrete geval hier bestaat deze communicatie enkel uit gegevens

afdrukken op het scherm. Wat op het scherm afgedrukt wordt, komt in de console (cfr schermonderdeel in Eclipse). Naast een console user interface heb je bv. ook een grafische user interface.

- De package **domein** zal de **business logica** van onze applicatie bevatten. Later komen we daar zeker nog op terug, maar hier hoort dus onze klasse KauwgomAutomaat thuis.

6.3. Implementatie van de domeinklasse KauwgomAutomaat

De code van deze klasse werd al volledig besproken in punt 5 van dit hoofdstuk. Maar doordat de klasse nu in de package domein geplaatst wordt, vind je nog 1 extra instructie in deze klasse.

```
    package domein;
```

Deze lijn code is de allereerste instructie die je terugvindt in de code, nog vóór de definitie van de klasse. Let er ook op dat ook deze instructie afgesloten wordt met een puntkomma.

De volledige code van de klasse KauwgomAutomaat ziet er dus uit als volgt:

```
package domein;

public class KauwgomAutomaat {

        private int aantalBallen;
        private String kleur;
        private boolean vergrendeld = true;

        public KauwgomAutomaat(int aantalBallen) {
            setAantalBallen(aantalBallen);
            setKleur("rood");
        }

        public KauwgomAutomaat() {
            this(0);
        }

        public int getAantalBallen() {
            return aantalBallen;
        }

        private void setAantalBallen(int aantalBallen) {
            if (aantalBallen >= 0 && aantalBallen < 150)
                this.aantalBallen = aantalBallen;
        }

        public String getKleur() {
            return kleur;
```

```
        }

        private void setKleur(String kleur) {
            this.kleur = kleur;
        }

        public boolean isVergrendeld() {
            return vergrendeld;
        }

        public final void setVergrendeld(boolean vergrendeld) {
            this.vergrendeld = vergrendeld;
        }

        public boolean isLeeg() {
                return aantalBallen == 0;
        }

        public void vulBij(int aantalBallen) {
                if (!vergrendeld && aantalBallen >=0)
                        setAantalBallen(this.aantalBallen + aantalBallen);
        }

        public void draaiHendel()
        {
            if(!isVergrendeld() && !isLeeg())
                aantalBallen--;
        }
}
```

6.4. Implementatie van de applicatieklasse KauwgomAutomaatApplicatie

6.4.1. Referentievariabele

Om goed te begrijpen wat een referentievariabele eigenlijk is, kijken we eerst nog eens naar het begrip **variabele**.

Declaratie van een variabele = het reserveren van een plekje in het geheugen, je geeft naam aan het plekje en in java leg je vast welk soort data je kan bewaren met dat plekje.

Voorbeeld 1: variabele van een primitief datatype

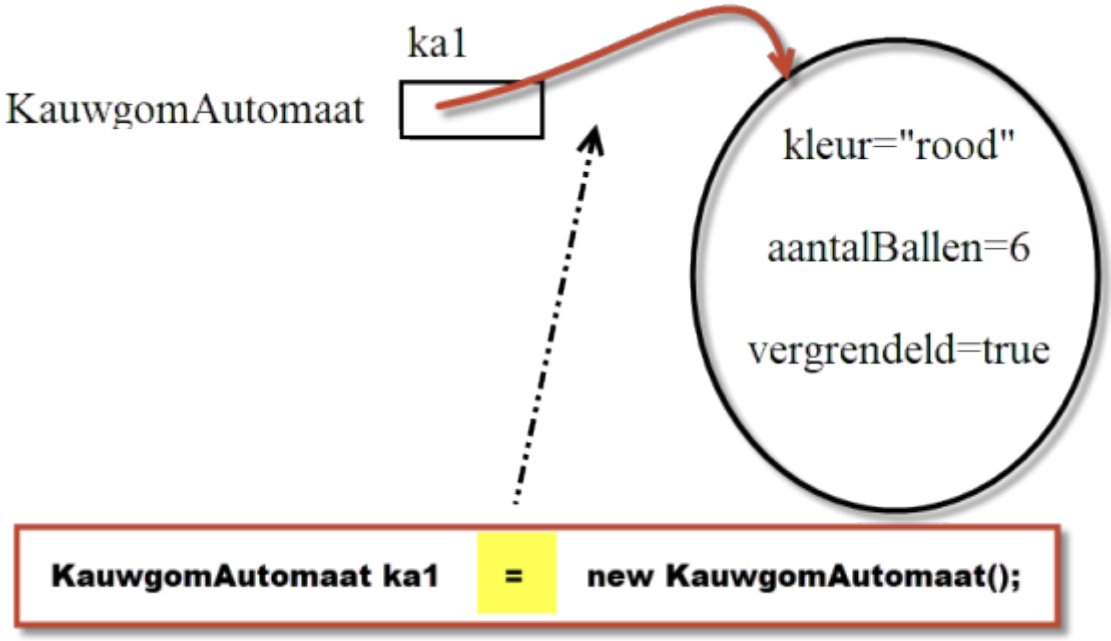
Eens we dat plekje hebben in het geheugen, kunnen we nu iets bewaren in deze variabele.

Via deze variabele kunnen we nu "werken" met deze waarde. In het geval van een getal, kan je bijvoorbeeld een berekening uitvoeren.

```
getal *= 2;
```

Voorbeeld 1: variabele voor een object (=referentievariabele)

We hebben nu dus ook een plekje in het geheugen waarmee we iets kunnen bewaren. Dit plekje kan nu verwijzen naar een KauwgomAutomaat-object.



Aan de rechterzijde van de toekenning zie je dat de constructor aangeroepen wordt voorafgegaan door het woord new. Nu wordt dus een object aangemaakt. In dat object (voorgesteld door de ovaal) zitten alle attributen met hun waarde.

De toekenning zorgt ervoor dat ons gedeclareerde plekje in het geheugen verwijst/refereert naar het object, ergens anders in het geheugen. Vandaar dat we spreken over een referentievariabele.

Via deze referentievariabele kunnen we nu praten/communiceren met het object. We noteren dat met de puntnotatie.

```
    boolean status = ka1.isVergrendeld();
```

6.4.2. Applicatieklasse

```
package cui;

import domein.KauwgomAutomaat;

public class KauwgomAutomaatApplicatie {

    public static void main(String[] args) { ①

        KauwgomAutomaatApplicatie kaa = new KauwgomAutomaatApplicatie();
        kaa.start();

    }

    private void start() {

        //maken van 2 objecten KauwgomAutomaat
        KauwgomAutomaat ka1 = new KauwgomAutomaat(); ②
        KauwgomAutomaat ka2 = new KauwgomAutomaat(50); ③

        //afdrukken van kleur en aantal balletjes ④
        System.out.printf("De eerste kauwgomautomaat is %s van kleur en bevat %d
balletjes.%n", ka1.getKleur(), ka1.getAantalBallen());
        System.out.printf("De tweede kauwgomautomaat is %s van kleur en bevat %d
balletjes.%n", ka2.getKleur(), ka2.getAantalBallen());

        //ontgrendelen
        ka1.setVergrendeld(false); ⑤

        //vullen
        ka1.vulBij(20); ⑥

        //vergrendelen
        ka1.setVergrendeld(true); ⑦

        //afdrukken van kleur en aantal balletjes
        System.out.printf("De eerste kauwgomautomaat is %s van kleur en bevat %d
balletjes.%n", ka1.getKleur(), ka1.getAantalBallen());
        System.out.printf("De tweede kauwgomautomaat is %s van kleur en bevat %d
balletjes.%n", ka2.getKleur(), ka2.getAantalBallen());

        //vullen
        ka1.vulBij(75); ⑧

        //afdrukken van kleur en aantal balletjes
        System.out.printf("De eerste kauwgomautomaat is %s van kleur en bevat %d
```

```
balletjes.%n", ka1.getKleur(), ka1.getAantalBallen()); ⑨
        System.out.printf("De tweede kauwgomautomaat is %s van kleur en bevat %d
balletjes.%n", ka2.getKleur(), ka2.getAantalBallen());

    }
}
```

① Bij het opstarten van een applicatie wordt altijd gezocht naar die speciale **main-methode**. Alle functionaliteit die we willen uitwerken komt dus in die main-methode te staan.

② Allereerst willen we een object maken dat rood is en leeg. Hiervoor hebben een **default constructor** (=constructor zonder parameters) gemaakt in onze domeinklasse. Dus daar willen we nu gebruik van maken. Via de defaultconstructor maken we een nieuw object aan en we kennen dit toe aan de referentievariabele ka1.

③ We maken nog een tweede object van KauwgomAutomaat. Hier is de vraag om ervoor te zorgen dat er direct 50 balletjes voorradig zijn, daarom maken we gebruik van de constructor met 1 parameter. De waarde 50 wordt **doorgegeven als concrete waarde voor de parameter**, de kleur wordt terug automatisch ingesteld op "rood". Dit tweede object houden we bij met behulp van de referentievariabele ka2.

④ De kleur en het aantal balletjes van elk object drukken we af op het scherm. De instructies in java om af te drukken werden vroeger al besproken. We kiezen hier voor de printf-methode. De waarden van het attribuut kleur en het attribuut aantalBallen zitten in de twee objecten. Om aan die waarden te geraken zullen we moeten **communiceren met** deze **objecten**. Dat gebeurt via de **referentievariabele**. Met de **puntnotatie** kunnen we toegankelijke methodes via de referentievariabele aanroepen. Aangezien we de waarde van attributen willen kennen, zullen we de bijhorende getters gebruiken. Bijvoorbeeld: **ka1.getKleur()**. Deze getter heeft als returntype een String. Dus je krijgt een stuk tekst terug als antwoord op de vraag. Dat stukje tekst wordt dan via de printf-methode netjes op het scherm afgedrukt.

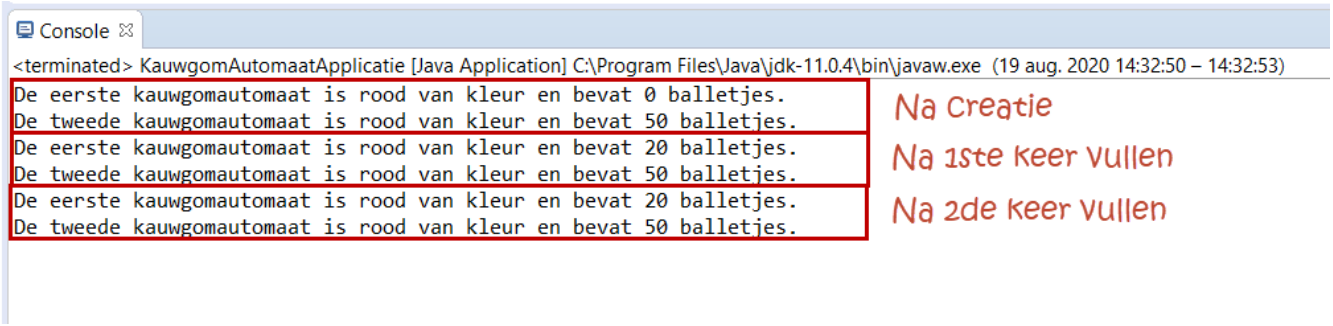
⑤ Om een kauwgomautomaat te ontgrendelen moeten we een nieuwe waarde kunnen geven aan het attribuut vergrendeld dat in het object aanwezig is. Hiervoor is in de domeinklasse een setter voorzien. **Via de parameter van die setter kunnen we een nieuwe waarde doorgeven.** Om die setter aan te roepen werken we weer via de referentievariabele en de puntnotatie. Op deze manier kunnen we communiceren met het object.

⑥ Om een kauwgomautomaat bij te vullen vinden we ook een methode in de klasse KauwgomAutomaat. Als we een object hebben van de klasse kunnen we dan ook die methode aanroepen. Dus vanuit de applicatie roepen we via de referentievariabele de methode vulBij aan. Deze methode heeft 1 parameter. Als we de methode aanroepen, moeten we een waarde doorgeven voor die parameter (een getal). Als deze instructie uitgevoerd wordt, doorlopen we de implementatie van vulBij in de klasse KauwgomAutomaat.

⑦ Via de referentievariabele ka1 roepen we de methode setVergrendeld aan en we geven als waarde voor de parameter *true* door. Op deze manier wordt de kauwgomautomaat terug vergrendeld. Als we daarna terug de kleur en het aantalBallen afdrukken zien we inderdaad dat bij de eerste kauwgomautomaat balletjes bijgekomen zijn, de tweede kauwgomautomaat is niet gewijzigd.

⑧ Op dezelfde manier proberen we nu nog eens extra balletjes toe te voegen aan de kauwgomautomaat. Maar momenteel is deze vergrendeld. Dus er zou niets mogen bijkomen…

⑨ Bij het afdrukken van het aantalBallen van beide kauwgomautomaten zien we inderdaad dat er geen balletjes toegevoegd werden bij de laatste keer bijvullen.



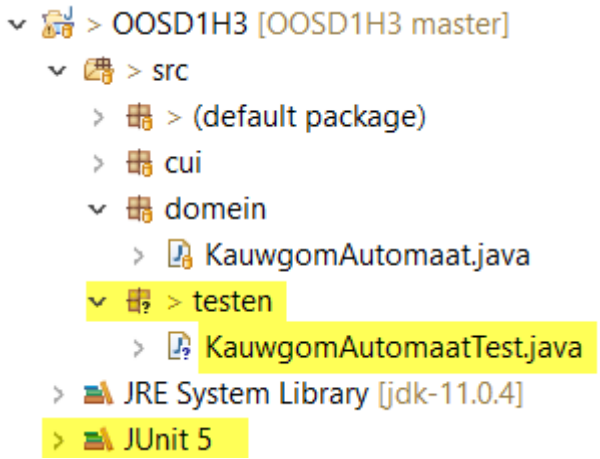
7. Testen van de domeinklasse

Bij het implementeren van de applicatie in punt 6 werken we eerst de domeinklasse uit en daarna de applicatieklasse. Is dit logisch? Als we in de applicatieklasse gebruik willen maken van de domeinklasse (objecten aanmaken, methodes aanroepen), dan is het inderdaad logisch dat die domeinklasse eerst volledig uitgewerkt wordt.

MAAR... eens je de code van die domeinklasse afgewerkt hebt, wil je uiteraard graag weten of die code werkt, of je nergens foutjes gemaakt hebt bij het implementeren. Aangezien we deze klasse niet kunnen uitvoeren - er is GEEN mainmethode - kunnen we niet zien of alles naar behoren werkt.

Om zeker te zijn dat alle functionaliteit in de domeinklasse correct werkt, zullen we leren hoe je deze klasse "test".

JUnit 5 is een testframework voor java-ontwikkelaars. Daar maken we gebruik van.



Per domeinklasse die getest moet worden heb je een testklasse. Deze worden allemaal verzameld in de package testen. Om gebruik te kunnen maken van dit testframework voegen we ook een extra bibliotheek (JUnit5) toe aan het project.

Een testklasse, zoals KauwgomAutomaatTest, test eigenlijk of alle uitgewerkte functionaliteit in de domeinklasse naar behoren werkt.

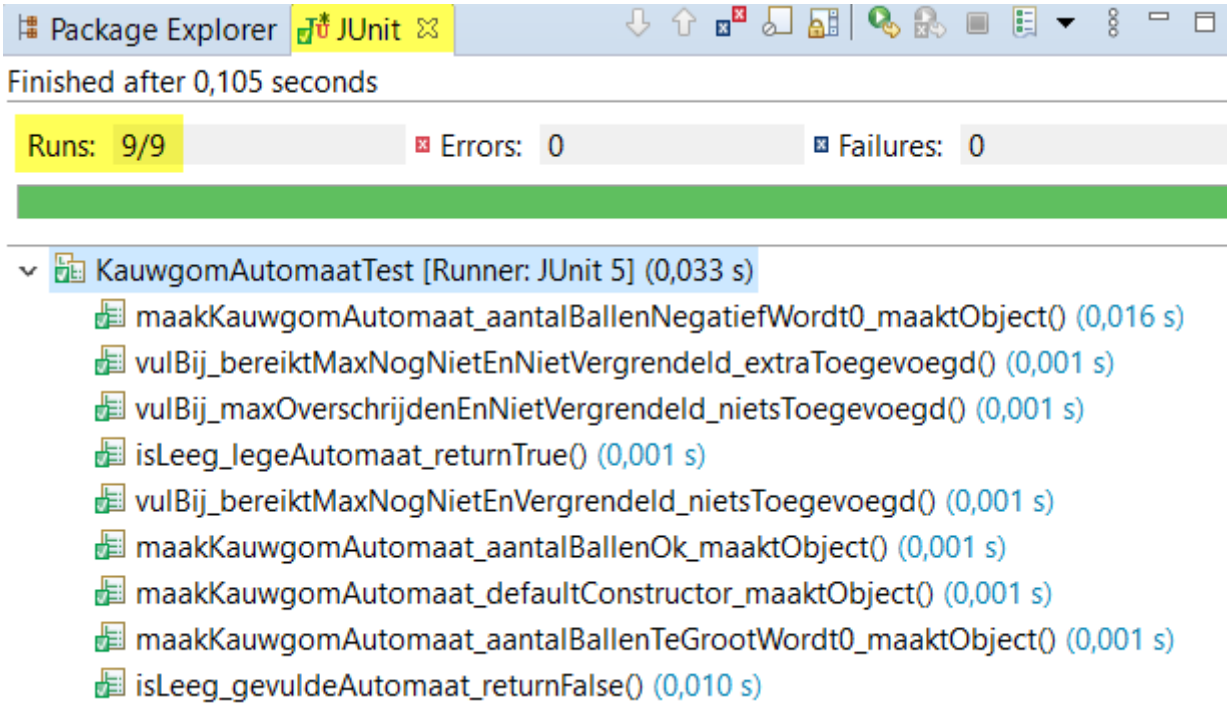
Bijvoorbeeld:

De methode isLeeg werkt goed als de methode true teruggeeft als het aantalBallen gelijk is aan 0. De methode werkt ook goed als de methode false teruggeeft wanneer het aantalBallen net niet gelijk is aan 0. Beide gevallen worden getest in de testklasse.

De constructor werkt goed als een object correct aangemaakt werd en alle attributen correct geïnitialiseerd zijn.

Zo moet dus alle functionaliteit (lees publieke methodes) getest worden op goede werking.

Bij het uitvoeren van de testen krijg je een overzicht. Hier worden bv 9 verschillende situaties getest. Ze worden opgesomd. Als alles groen kleurt, slagen alle testen. Dat is het streefdoel!



Als alle testen groen kleuren, weet je dat de domeinklasse volledig werkt. Dan kan je starten aan de applicatieklasse.