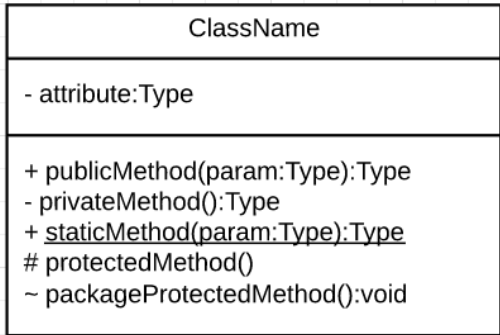


UML Klassendiagrammen

UML klassendiagrammen kunnen gebruikt worden om een applicatie te ontwerpen. Vervolgens wordt het ontwerp, dus het klassendiagram gebruikt om de implementatie in code te maken. Een klassendiagram geeft richting aan een programmeur om de implementatie te maken. Zelden zal een ontwerp/klassendiagram als een geheel compleet en strikte bouwhandleiding voor de programmeur dienen. Hoe meer details je specificeert in het UML diagram, hoe strikter de implementatie dient plaats te vinden. In onderstaande voorbeelden hebben we een strikte relatie tussen UML en code gemaakt om de betekenis van de diverse symbolen te benadrukken. Als gevraagd wordt een ontwerp te maken, zorg dan voor genoeg 'implementatievrijheid' voor de programmeur. Het is onmogelijk voor de ontwerper om een hele applicatie strikt te specificeren, zodat de code uiteindelijk 100% overeenkomt met het ontwerp. Concreet betekent dit dat niet ieder attribuut, getter, setter, methode of concrete relatie gespecificeerd hoeft te worden in het ontwerp. Bedenk als ontwerper wat echt belangrijk is. Een ontwerp dient de intentie van de ontwerper weer te geven, de applicatieprogrammeur gebruikt deze intentie om zijn code te schrijven.

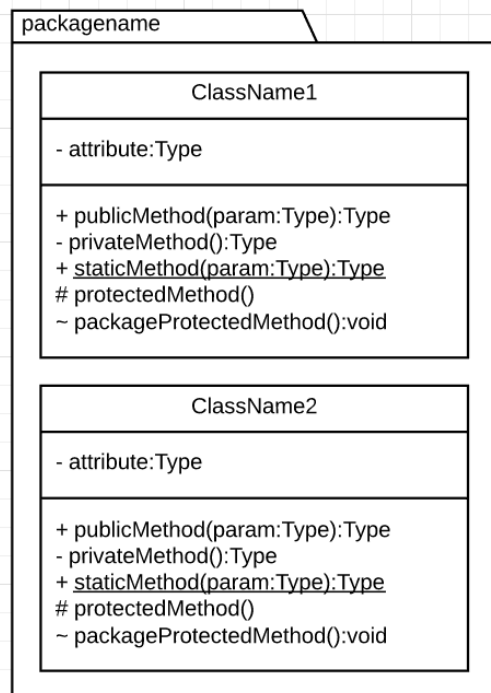
Class											
 <p>The diagram shows a class box with three compartments. The top compartment contains 'ClassName'. The middle compartment contains '- attribute:Type'. The bottom compartment contains '+ publicMethod(param:Type):Type', '- privateMethod():Type', '+ <u>staticMethod(param:Type):Type</u>', '# protectedMethod()', and '~ packageProtectedMethod():void'.</p>	<p>Dit is het symbool voor een klasse. Het bestaat uit 3 vakjes. Bovenaan de naam van de klasse, het middelste vak beschrijft de attributen en het onderste vak beschrijft de methoden van deze klasse.</p> <p>Als de klassenaam of methodenaam <i>schuingedrukt</i> is betreft het een abstracte klasse of methode.</p> <p>Onderstreept betekent dat de methode of attribuut static is.</p>										
<p>Access-modifiers:</p> <p>De access-modifiers bij de attributen en methoden worden aangegeven door de volgende symbolen:</p> <table border="1"><thead><tr><th>Symbool</th><th>Betekenis</th></tr></thead><tbody><tr><td>+</td><td>public</td></tr><tr><td>#</td><td>protected</td></tr><tr><td>~</td><td>package-private</td></tr><tr><td>-</td><td>private</td></tr></tbody></table>		Symbool	Betekenis	+	public	#	protected	~	package-private	-	private
Symbool	Betekenis										
+	public										
#	protected										
~	package-private										
-	private										
<p>Type-declaration:</p> <p>De type-declaratie wordt in tegenstelling tot de taal Java achter het attribuut, parameter of methode weergegeven. Het type en attribuut, parameter of methode worden gescheiden door een '·'.</p> <p>Voorbeelden:</p> <pre>+ publicMethod(param1:String):boolean => public boolean publicMethod(String param1) - attribute:int => private int attribute;</pre>											

Multipliciteit:

Bij de klassen kan worden vermeld hoeveel instanties er van voorkomen. Deze aanduiding wordt geplaatst bij de relaties tussen de klassen aan de zijde van de klasse waar de multipliciteit betrekking tot heeft:

Aanduiding	Betekenis
1	Er is 1 instantie van deze klasse
0..1	Er is 0 of 1 instantie van deze klasse
1..5	Er zijn minimaal 1 tot maximaal 5 instanties van deze klasse
0..*	Er zijn 0 of meer instanties van deze klasse
*	Er zijn 0 of meer instanties van deze klasse
5..*	Er zijn minimaal 5 of meer instanties van deze klasse
enzovoorts	

Package



Een package is een samenhangende groep van klassen. Je groepeer je klassen in packages op basis van het SRP (Single Responsibility Principle). In het label bovenaan het symbool vermeld je de packagenaam, in het packagesymbool geef je de klassen weer die in het package zitten.

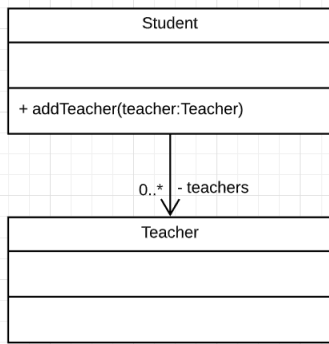
Je geeft in de klasse aan tot welke package deze behoort op de volgende manier:

```
package packagename;
```

Gebruik je een klasse uit deze package in een andere package, dan moet je de klasse importeren:

```
import packagename.ClassName;
```

Relaties



Een relatie tussen 2 klassen heet een association (associatie in het Nederlands) als een referentie naar de andere klasse(n) in een attribuut wordt opgeslagen. De relatie kan door symbolen aan de uiteinden extra worden gespecificeerd:

Symbol	Betekenis
—	Er is geen richting gedefinieerd. Beide richtingen zijn mogelijk.
➔	Navigatie is verplicht naar de aangewezen klasse.
✕	Er is expliciet <u>geen</u> navigatie mogelijk naar de klasse met het kruis.

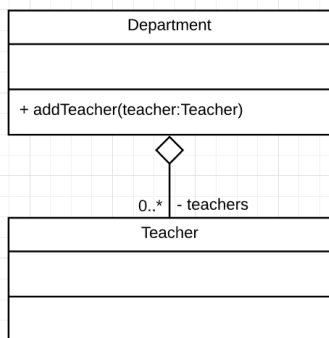
```

// association
import java.util.ArrayList;
import java.util.List;

public class Student {
    ...
    private List<Teacher> teachers = new ArrayList<>();
    ...
    public void addTeacher(Teacher teacher) {
        teachers.add(teacher);
    }
}
  
```

Eigenschappen:

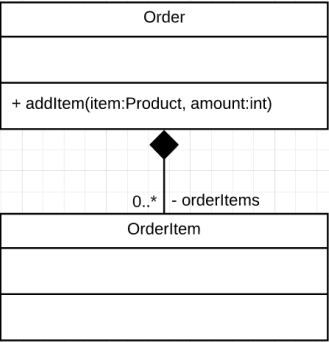
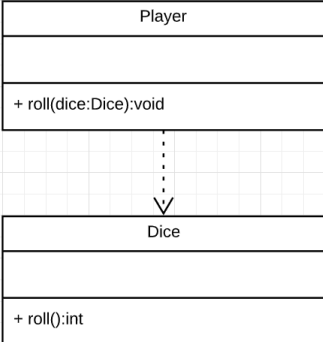
- Geen eigenaarschap
- Referentie wordt opgeslagen in een attribuut
- Onafhankelijke levensduur van objecten

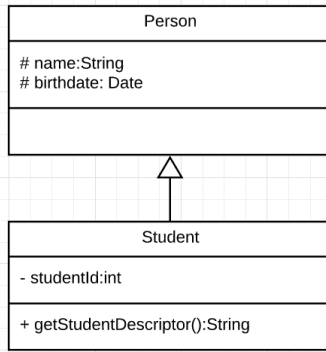


Een relatie tussen 2 klassen heet een aggregation (aggregatie in het Nederlands) als er een 'heeft-een' relatie tussen 2 klassen is, waarbij de 2 klassen onafhankelijk van elkaar bestaansrecht hebben. Een aggregatie wordt alleen op design niveau gebruikt om dit uit te drukken, in code is er geen verschil tussen een normale association of een aggregation.

Eigenschappen:

- Geen eigenaarschap
- Referentie wordt opgeslagen in een attribuut
- Onafhankelijke levensduur van objecten

 <pre> classDiagram class Order { +addItem(item:Product, amount:int) } class OrderItem { } Order "1" *-- "0..*" OrderItem : - orderItems </pre>	<p>Een relatie tussen 2 klassen heet een composition of composite-aggregation (compositie in het Nederlands) als er een ‘heeft-een’ relatie tussen 2 klassen is, waarbij de ene klasse <u>geen</u> bestaansrecht heeft zonder de andere klasse en waarbij de ‘hoofdklasse’ verantwoordelijk is voor het aanmaken van objecten van het andere type.</p> <pre>//composition import java.util.ArrayList; import java.util.List; public class Order { private List<OrderItem> orderItems = new ArrayList<>(); ... public void addItem(Product product, int amount) { this.orderItems.add(new OrderItem(product, amount)); } ... }</pre> <p>Eigenschappen:</p> <ul style="list-style-type: none"> • Wel eigenaarschap • Child object wordt door parent aangemaakt en ‘leeft’ zolang de parent leeft. • Levensduur van child objecten is afhankelijk van de parent.
 <pre> classDiagram class Player { +roll(dice:Dice):void } class Dice { +roll():int } Player ..> Dice </pre>	<p>Een relatie tussen 2 klassen heeft altijd een dependency (afhankelijkheid in het Nederlands). Als de referentie wordt opgeslagen in een attribuut is het een association, aggregation of een composition. Tekenend we zo’n relatie, dan laten we de ‘stippelpijl’ achterwege. Deze dit al impliciet in de andere relatie. We tekenen wel de ‘stippelpijl’ als de relatie tijdelijk geleend wordt, bijv. tijdens uitvoer van een methode. De referentie naar de andere klasse wordt dan niet vastgehouden.</p> <pre>//dependency public class Player { public void rollDice(Dice dice) { int steps = dice.roll(); ... } }</pre> <p>Eigenschappen:</p> <ul style="list-style-type: none"> • Geen eigenaarschap, het object wordt geleend • Onafhankelijke levensduur van objecten



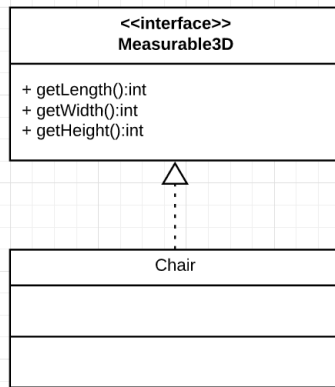
Het gedrag van een andere klasse uitbreiden doen we door middel van inheritance (overerving in het Nederlands). Alle eigenschappen in de superclass worden overgenomen in de subclass en het gedrag kan naar behoeven worden uitgebreid of veranderd. De relatie kan worden gezien als een 'is-een' relatie. In code geven we dit aan door het keyword 'extends'. Een relatie zoals deze noemen we ook wel een generalisation (generalisatie in het Nederlands).

```
public class Student extends Person {
    private int studentId;
    ...

    public String getStudentDescriptor() {
        return super.name + " (" + this.studentId + ")";
    }
}
```

Eigenschappen:

- Een superclass wordt in de subclass gerefereerd door `super`.
- Alle eigenschappen en gedrag wordt overgenomen door de subclass



Een contract afspreken voor een klasse doen we met behulp van een interface. Een interface beschrijft alle publieke methoden die de klasse moet implementeren, er wordt geen implementatie gegeven. Een interface vormt de abstractie van de concrete implementatie, dit wil zeggen: de interface omschrijft wat je er mee kunt; de implementatie bevat de code over hoe dat dat gedrag wordt uitgevoerd.

```
public interface Measurable3D {
    int getLength();
    int getWidth();
    int getHeight();
}

public class Chair implements Measurable3D {
    @Override
    public int getLength() {
        return 0;
    }

    @Override
    public int getWidth() {
        return 0;
    }

    @Override
    public int getHeight() {
        return 0;
    }
}
```

Eigenschappen:

- Een klasse kan een interface implementeren
- Alle methoden uit een interface moeten geïmplementeerd worden
- Als niet alle methoden geïmplementeerd kunnen worden, dan de interface splitsen in meerdere interfaces (interface segregation principle)
- Een interface is een Type. Dus je kunt het ook als parameter meegeven, retourneren uit een methode, opslaan als verzameling in een lijst (polymorfisme).
- Je kan een interface niet instantiëren omdat deze impliciet abstract is.