# Casino de Gouden Driehoek

# Inhoud

# Inleiding

Welkom bij Casino de Gouden Driehoek. Dit is een oefening bestaande uit acht losse opdrachten die aansluiten op de verschillende lessen die tijdens de cursus 'Java programmeren' worden gegeven.

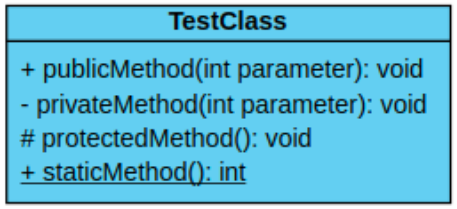
De casus is als volgt:
Casino de Gouden Driehoek is een casino met meerdere locaties in Nederland waar verschillende spellen gespeeld kunnen worden. Gezien de populariteit van internet casino's heeft de directie besloten om de digitale markt te gaan betreden. In deze reeks van opdrachten zul jij een eerste testversie van het online casino ontwikkelen.

De applicatie die je gaat ontwikkelen, zal een command line applicatie zijn waar met casino munten verschillende spellen gespeeld kunnen worden. Tijdens de opdrachten ga je een voor een de verschillende spellen bouwen om ze uiteindelijk te combineren tot één volledige casino applicatie.

Op github (link) vind je de startcode en de oplossingen van de verschillende opdrachten zodat je je eigen oplossing hiermee kunt vergelijken of kunt spieken als je er niet helemaal uit komt.

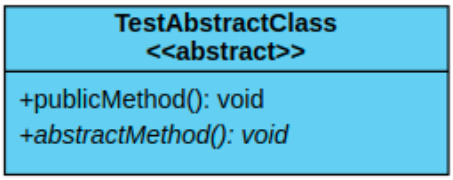
# Voorkennis

Bij de opdrachtbeschrijvingen vind je klassendiagrammen (UML) waarmee de structuur van de te bouwen applicatie wordt beschreven. Het is voor het lezen van deze diagrammen dus van belang dat je de basiselementen van een UML klassendiagram kent. Hieronder vind je de basiselementen die tijdens de opdrachten gebruikt worden.



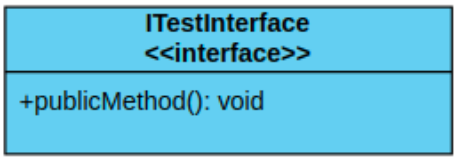

**Class**

Methods can be public (+), private (-) or protected (#). When a method is underlined it is static.



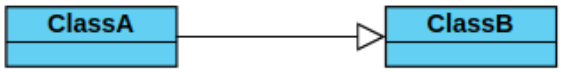

**Abstract class**

Abstract classed use the <<abstract>> header. Abstract methods are shown in *italic.*

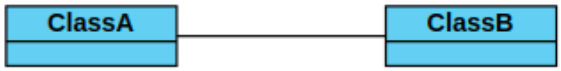



**Interface**

Interfaces use the <<interface>> header



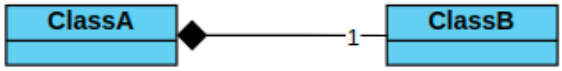

**Inheritance**
A inherits from B (extends)



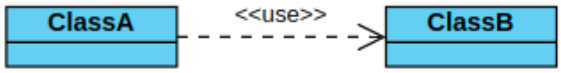

**Association**
A calls B or B calls A


**Aggregation**

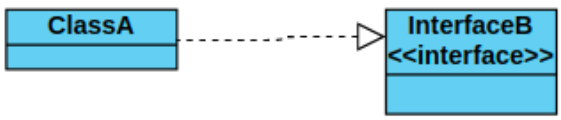
An instance of A has an instance of B, B can survive if A is deleted. The 0..* means a has 0 to infinite instances of B.




**Composition**
A has an instance of B, B can **not** survive if A is deleted. The 1 means A has exactly 1 instance of B.




**Usage**
A uses B but does not have an instance of B (ex. calls a static method).




**Implements**

A implements interface B

# De opdrachten

1. **Java Basis**
   Tijdens deze opdracht ga je met de basis Java concepten aan de slag en ga je het eerste casino spel bouwen: hoger of lager.

2. **Relaties**
   Je gaat het spel Blackjack implementeren in de applicatie. Hierin zul je zowel met Java in het algemeen en de verschillende relaties tussen objecten kunnen oefenen.

3. **Overerving & abstracte klassen**
   Om te oefenen met overerving en abstracte klassen ga je het Blackjack spel dat je in de vorige opdracht hebt opgezet uitbreiden. Het doel hierin is dat je verschillende varianten op Blackjack kunt toevoegen zonder enorm veel code te dupliceren.

4. **Scope, access modifiers & keywords**
   Tijdens deze oefening gaan we een nieuwe set van spellen toevoegen aan de applicatie: fruitmachines. Je zal hierin te maken krijgen met alle concepten uit de voorgaande oefening aangevuld met scope, access modifiers & keywords.

5. **Static methods & overloading**
   Je gaat het spel galgje toevoegen aan de applicatie, hierin zul je kunnen oefenen met complexere logica, statische methodes en method overloading.

6. **Interfaces**
   Alle spellen zitten nu in de applicatie, het doel van deze opdracht is om ze aan elkaar te knopen zodat je ze allemaal kunt spelen zonder steeds de applicatie te hoeven stoppen. Dat zal de uitdaging van deze oefening zijn.

7. **Maven**
   De applicatie is nu zo goed als af, maar hij kan wel wat kleurrijker. Om dit voor elkaar te krijgen gaan we Maven toevoegen. Met Maven kun je eenvoudig een library toevoegen waarmee je wat kleur aan de applicatie kunt geven.

8. **Unittesten met JUnit**
   Om te garanderen dat de applicatie blijft werken zoals bedoeld, zul je ook unit tests moeten toevoegen. Dat is wat je tijdens deze opdracht zult doen voor een van de spellen.

# Java basis

Zoals beschreven in de inleiding ga je een command line casino applicatie ontwikkelen, tijdens deze opdracht ga je daarvoor de eerste stappen zetten. Je gaat het spel 'hoger of lager' bouwen zodat je dit aan het einde van de opdracht kunt spelen.
Tijdens deze eerste opdracht ga je een begin maken aan de applicatie door het eerste spel (hoger of lager) te implementeren.

Doorloop de voorbereidingsstappen die je hier direct onder vindt en lees de rest van de opdrachtbeschrijving goed door. Maak daarna de applicatie af zodat alle stappen die in 'verloop van het spel' beschreven staan correct doorlopen worden en het spel speelbaar is.

## Voorbereiding

1. Maak een lege map aan en plaats daar de code in die je in de GitHub repository (link) in de map 'start' vindt. Dit is het startpunt vanaf waar je de applicatie gaat opbouwen.
2. Importeer deze map als een nieuw project in Intellij:
    a) Klik in de menubalk van Intellij op File -> New -> Project from existing sources...
    b) Kies de map die je zojuist hebt aangemaakt
    c) Kies bij de volgende stap de optie 'create project from existing sources' en klik op 'next'.
    d) Geef het project een naam en klik vervolgens bij alle daaropvolgende stappen op 'next' zonder iets aan te passen.
    e) Het project wordt nu voor je geopend. Je kunt de applicatie draaien door de 'Main' klasse te openen in de novi map en naast de code op een van de twee groene start icoontjes te klikken.

## De start applicatie

Als je het project hebt geopend in Intellij zul je zien dat er al wat code in staat, namelijk een 'Main' klasse met daarin de main methode en een HigherLowerGame klasse met een playGame methode. De logica die nodig is om het 'higherLower' spel te laten werken zul je in de HigherLowerGame klasse moeten gaan toevoegen.

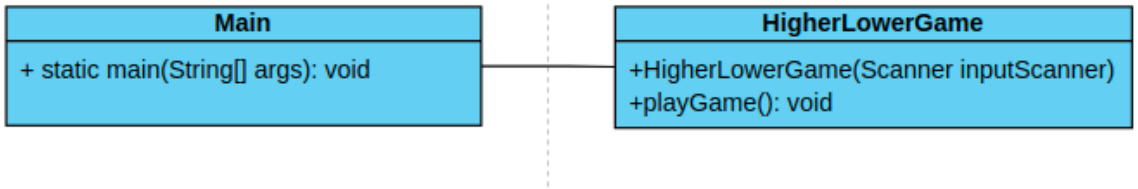
Voor het verwerken van input van de speler ga je gebruik maken van de 'Scanner' klasse. Hiervan wordt al een object voor je aangemaakt. Iedere keer dat je een methode op het scanner object aanroept (bijv. nextLine(), nextInt(), nextLong() etc.) zal je applicatie wachten met het uitvoeren van de rest van de code tot de gebruiker iets heeft ingetypt in de console en op 'enter' heeft gedrukt. Daarmee kun je dus de input van de gebruiker verwerken. Hier vind je iets meer informatie over de Scanner klasse.

In het start project is dit mechanisme al (deels) voor je geïmplementeerd.

## Spelverloop

1. De speler start het spel (je start de applicatie door het starten van de ‘main’ methode)
2. Het spel kiest een willekeurig nummer tussen 0 en 100 dat geraden moet worden.
3. Het spel zet het aantal gespeelde beurten op 0
4. Het spel print de regel ‘Make a guess’
5. De speler raadt een nummer
6. Het spel hoogt het aantal gespeelde beurten op met 1
7. Was het nummer te hoog? Het spel print de regel `That number is too high!`
   (terug naar stap 4)
8. Was het nummer te laag? Het spel print de regel `That number is too low!`
   (terug naar stap 4)
9. Komt het nummer overeen? Het spel print de regel `Correct! You guessed the number in **x** turns` waarbij **x** het aantal gespeelde beurten is.
   (het spel wordt gestopt)

## Klassendiagram



# Relaties

Tijdens deze oefening ga je het spel Blackjack toevoegen aan de applicatie. Hierin zul je aanzienlijk meer klassen nodig hebben die verschillende relaties met elkaar hebben. Kijk goed naar de stappen in het spelverloop en het klassendiagram hieronder en maak de implementatie af.

## Voorbereiding

1. Heb je de vorige opdracht afgerond? Dan is dat ook meteen het startpunt van deze opdracht. Wanneer dit niet zo is kun je de code in de map 'solution-1' gebruiken als startpunt.
2. Maak een nieuwe map 'blackjack' aan in de novi map en maak daarin alvast de klasse 'BlackjackGame' aan. Geef deze klasse dezelfde implementatie als de `HigherLower` start klasse waar je in opdracht 1 mee begonnen bent (zoals hij in de 'start' map te vinden is). Dit is een goed beginpunt.
3. Pas de 'main' methode in de Main klasse aan zodat er een instantie van 'BlackjackGame' wordt aangemaakt in plaats van 'HigherLowerGame'.

## Spelregels

- Het spel begint met een set kaarten (deck), in deze variant spelen we zonder azen en zonder jokers. Dan zijn dus de kaarten 2, 3, 4, 5, 6, 7, 8, 9, 10, boer, vrouw en de koning.
- Iedere kaart heeft een waarde. Bij numerieke kaarten is dit de numerieke waarde, de boer, vrouw en koning zijn alle drie 10 punten waarden.
- Het spel wordt gespeeld tussen een 'dealer' (de applicatie) en een 'player' (de speler).
- Het doel is om zo dicht mogelijk bij (of exact op) 21 uit te komen in de waarde van de kaarten die de speler in zijn hand heeft. Gaat hij er overheen dan verliest hij.
- Bij aanvang van het spel krijgt de speler 2 kaarten, de dealer krijgt er 1.
- De speler kan dan kiezen voor een 'stay', hij trekt geen nieuwe kaart en hoopt dus dat de dealer over de 21 heen gaat (bust). Of de speler trekt een kaart (hit).
- De dealer doet daarna hetzelfde.
- Het spel eindigt wanneer de speler of de dealer over de 21 heen is gegaan of wanneer de speler of dealer heeft gekozen voor een 'stay' en de ander een hogere kaartwaarde (onder of gelijk aan 21) heeft bereikt.
- Bij een gelijke kaartwaarde in de eindsituatie wint de dealer altijd.

## Spelverloop

1. De speler start het spel
2. Het spel maakt een deck aan met daarin alle benodigde kaarten.
3. Het spel maakt de player en dealer objecten aan, beide met een lege ‘hand’
4. Het spel schudt de kaarten
5. Het spel geeft de player 2 kaarten van de stapel
6. Het spel geeft de dealer 1 kaart van de stapel
7. Keuze aan de speler: Hit or stay?
8. Indien hit: geef de speler een kaart van de stapel
9. Is de speler ‘bust’?

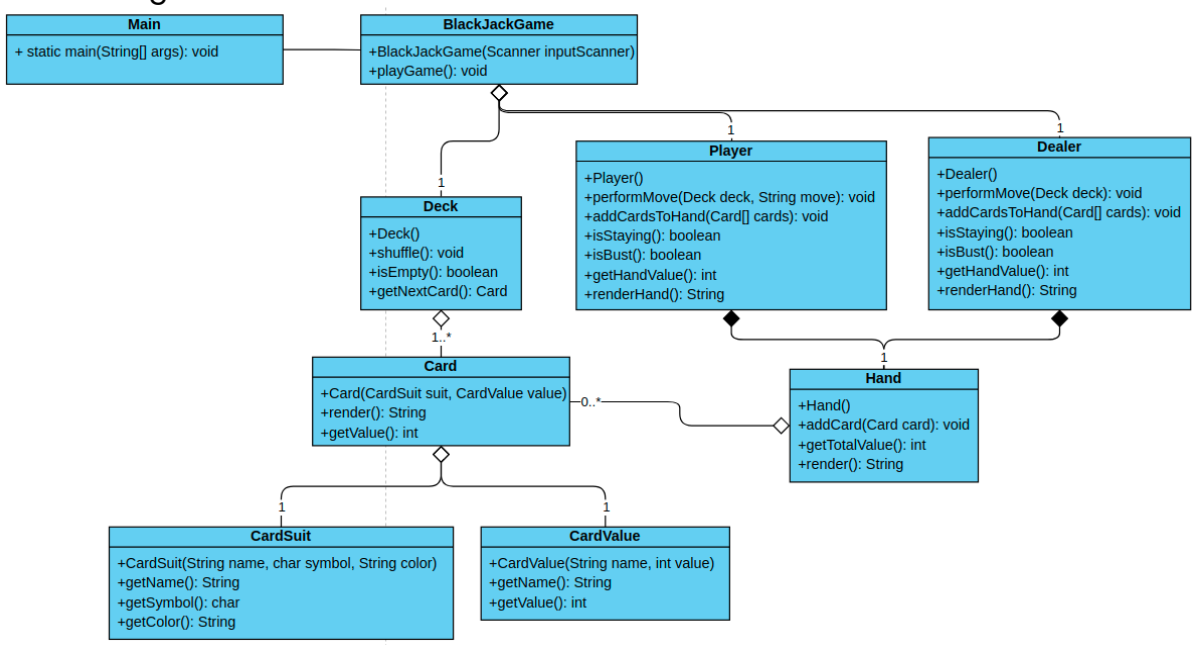
   *De speler heeft verloren*
10. Keuze aan de dealer: Hit or stay?
11. Indien hit: geef de dealer een kaart van de stapel
12. Is de dealer ‘bust’?

    *De speler heeft gewonnen*
13. Heeft de dealer gekozen voor een ‘stay’ en is de waarde van zijn hand < de waarde van de hand van de speler?

    *De speler heeft gewonnen*
14. Heeft de speler gekozen voor een ‘stay’ en is de waarde van zijn hand <= de waarde van de hand van de dealer?

    *De dealer heeft gewonnen*
15. Terug naar stap 6-12 (rekening houdende met de ‘stay’ status van de player en de dealer)

## Klassendiagram



## Hints

- De dealer zal gespeeld worden door de applicatie en zal dus zelf moeten beslissen of hij voor een ‘hit’ of een ‘stay’ kiest. Voor de eenvoud kun je er voor kiezen om de dealer altijd voor een ‘hit’ te laten kiezen wanneer de waarde van zijn kaarten < 17 is. In alle andere gevallen kiest hij dan voor een ‘stay’.
- In het klassendiagram is af te lezen dat de `CardSuit` klasse een ‘symbol’ property heeft van het type char. In de unicode standaard zitten de vier verschillende iconen (ruiten, harten, klavers & schoppen) die wij hiervoor kunnen gebruiken. Hieronder vind je een voorbeeld waarin dit voor de vier ‘suits’ voor je is ingevuld.

```
CardSuit[] suits = new CardSuit[] {
    new CardSuit("diamonds", '\u2666', "red"),
    new CardSuit("spades", '\u2660', "black"),
    new CardSuit("hearts", '\u2665', "red"),
    new CardSuit("clubs", '\u2663', "black")
};
```

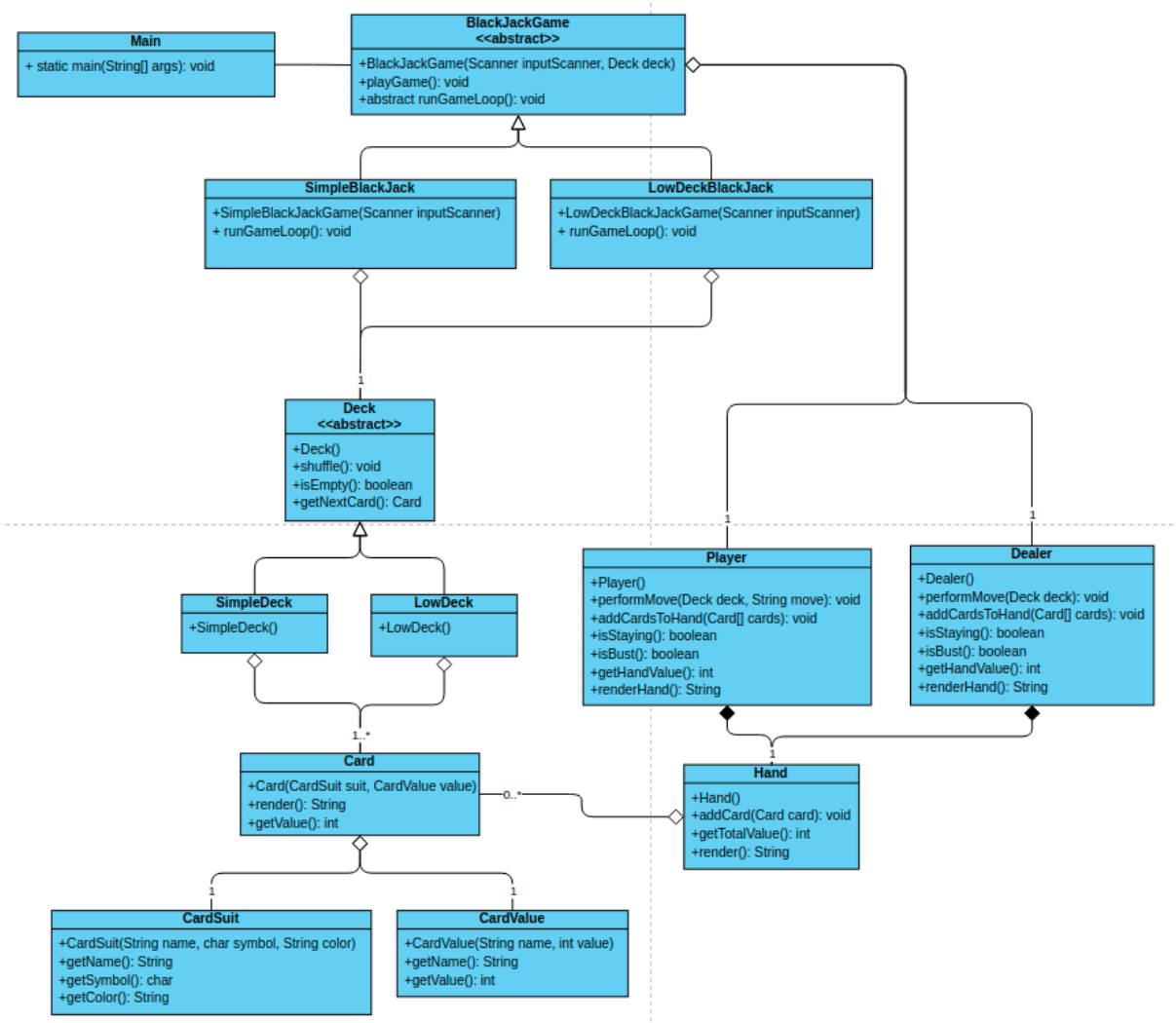
# Overerving & abstracte klassen

Tijdens deze oefening ga je het spel blackjack dat je in de vorige opdracht hebt gebouwd uitbreiden. Het doel is om het spelen van verschillende varianten op blackjack mogelijk te maken zonder code te hoeven dupliceren. Om dat voor elkaar te krijgen ga je gebruik maken van overerving en abstracte klassen.

Het spel moet twee varianten gaan ondersteunen: 'SimpleBlackjack' met kaarten en regels zoals je ze in de vorige opdracht hebt opgezet, en een variant die we 'LowDeckBlackjack' noemen waarbij wordt gespeeld met een kaartendeck bestaande uit alle kaarten met waardes tussen 2 en 6 uit drie losse decks.

Kijk goed naar het klassendiagram om de bestaande blackjack applicatie zo aan te passen dat beide spellen speelbaar zijn (na aanpassen van de 'main' methode).

## Klassendiagram



# Scope, access modifiers & keywords

Deze opdracht staat in het teken van scope, access modifiers en keywords, maar eigenlijk komen ook alle concepten uit de vorige opdracht nog eens terug. Je gaat wederom een spel toevoegen aan de applicatie, dit keer een casino klassieker: fruitautomaten.

Het spel zal bestaan uit drie verschillende fruitautomaten met verschillende instellingen. Bij het starten van het spel krijgt de speler een hoeveelheid munten, die kan hij gebruiken om bij één van de drie automaten te spelen.

In deze applicatie zul je zowel private, public als protected methodes gaan schrijven. Probeer zelf ook gebruik te maken van de verschillende keywords die Java kent.

## Voorbereiding

1. Heb je de vorige opdracht afgerond? Dan is dat ook meteen het startpunt van deze opdracht. Wanneer dit niet zo is kun je de code in de map 'solution-3' gebruiken als startpunt.
2. Maak een nieuwe map slotmachines aan in de novi map en maak daarin alvast de klasse SlotMachineGame aan. Geef deze klasse dezelfde implementatie als de `HigherLower` start klasse waar je in opdracht 1 mee begonnen bent (zoals hij in de 'start' map te vinden is). Dit is een goed beginpunt.
3. Pas de 'main' methode in de Main klasse aan zodat er een instantie van SlotMachineGame wordt aangemaakt en gestart.
4. De wielen van de fruitautomaten moeten uiteraard iconen hebben, hiervoor kun je (net als bij Blackjack) unicode iconen gebruiken. In de 'examples' map van het Github project staat een voorgedefinieerde klasse Symbol met daarin een aantal standaard iconen die je kunt gebruiken. Je kan uiteraard ook zelf wat Unicode iconen opzoeken (let er dan wel op dat ze niet allemaal in iedere terminal ondersteund worden).

## De machines

Het spel zal 3 machines gaan ondersteunen:

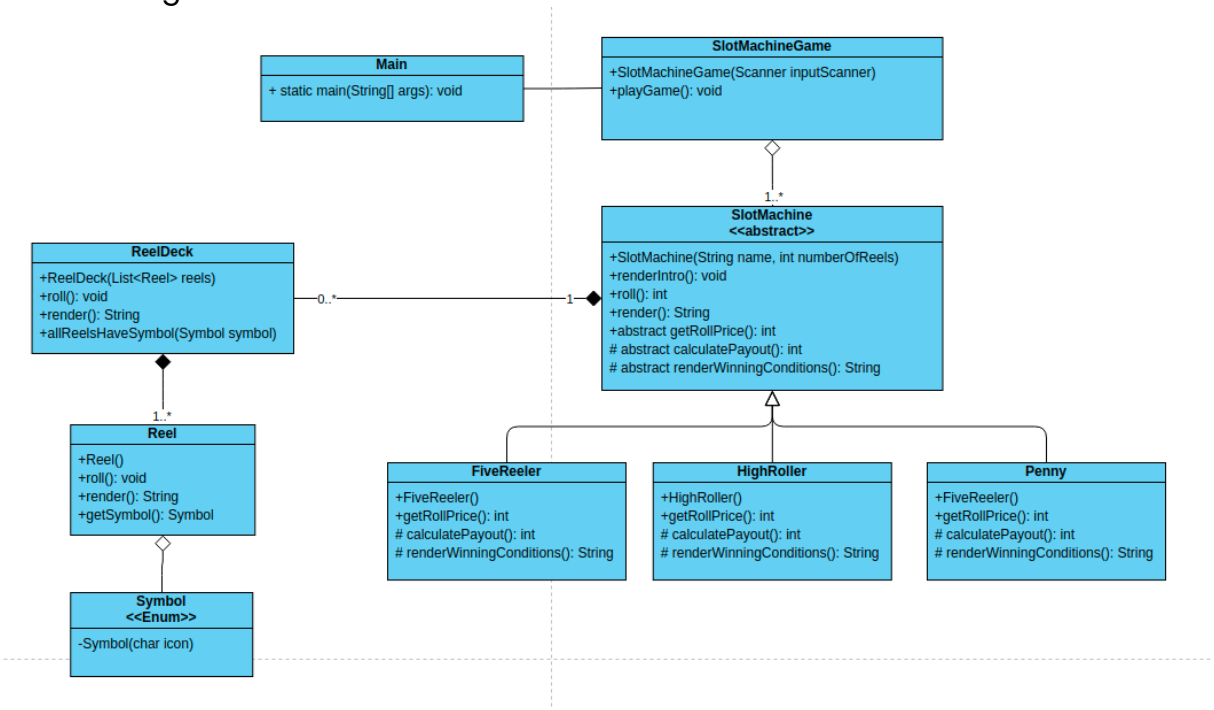
- De 'Penny Machine' heeft 3 wielen en ieder spel kost 1 munt. De speler wint enkel als alle wielen hetzelfde icoon tonen waarbij de winst afhankelijk is van het icoon. Je mag hier zelf de te winnen munten bij bedenken.
- De 'High Roller machine' is identiek aan de 'Penny Machine' alleen kost ieder spel 10 munten. De te winnen prijzen liggen uiteraard ook een stuk hoger, ook die mag je zelf bepalen.
- De '5-reeler' machine heeft, zoals de naam al aangeeft, 5 wielen in plaats van 3. Een spel op de machine kost 5 munten. De prijzen mag je wederom zelf bedenken.

## Spelverloop

1. De speler start het spel
2. De speler krijgt 100 munten om mee te spelen
3. Vraag aan de speler: op welke machine wil je spelen?
   Penny machine, High Roller machine of 5-reeler machine?
4. Heeft de speler genoeg munten om op de machine te spelen?
   a) Zo ja: ga naar stap 5
   b) Zo nee: ga terug naar stap 3

5. Start het spel op de machine
6. Trek aan de digitale hendel van de machine
7. Bereken de behaalde winst - de inleg en tel dit op bij het aantal munten van de speler
8. Heeft de speler genoeg munten om nog een keer te spelen?
    a) Zo ja: ga naar stap 9
    b) Zo nee: spel eindigt
9. Vraag aan de speler: wil je nog een keer spelen?
    a) Zo ja: ga terug naar stap 5
    b) Zo nee: spel eindigt

## Klassendiagram



# Static methods & overloading

In deze oefening ga je het spel 'galgje' toevoegen aan de applicatie. Hierbij zul je gebruik maken van een klasse met een statische methode voor het tonen van de juiste afbeeldingen.

## Voorbereiding

1. Heb je de vorige opdracht afgerond? Dan is dat ook meteen het startpunt van deze opdracht. Wanneer dit niet zo is kun je de code in de map 'solution-4' gebruiken als startpunt.
2. Maak een nieuwe map hangman aan in de novi map en maak daarin alvast de klasse HangmanGame aan. Geef deze klasse dezelfde implementatie als de `HigherLower` start klasse waar je in opdracht 1 mee begonnen bent (zoals hij in de 'start' map te vinden is). Dit is een goed beginpunt.
3. Pas de 'main' methode in de Main klasse aan zodat er een instantie van HangmanGame wordt aangemaakt en gestart.

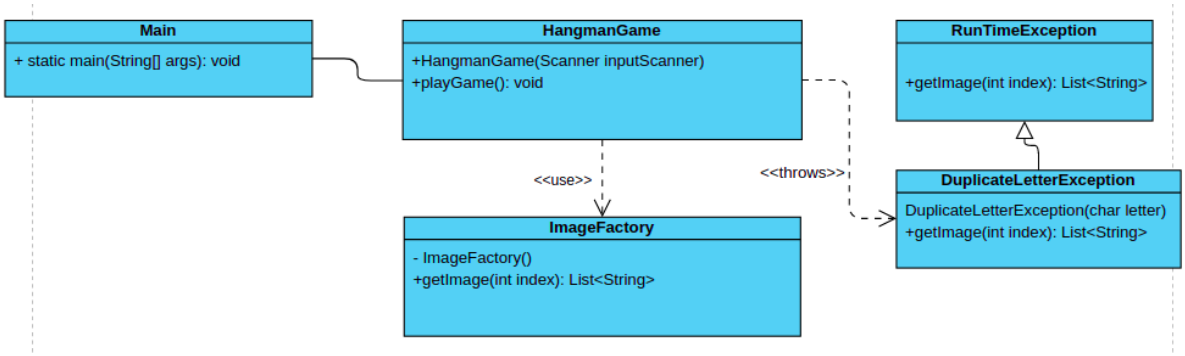
## Spelverloop

1. De speler start het spel
2. Het spel kiest een willekeurig woord uit een statische lijst van woorden.
3. Het spel toont het woord in * tekens om het aantal letters te laten zien
4. Het spel toont de tekst: 'Vul een letter in'
5. De gebruiker vult een letter in
6. Heeft de speler de letter al gebruikt?
   Zo nee: ga naar stap 7
   Zo ja: Gooi een 'DuplicateLetterException' en print het bericht, terug naar stap 4
7. Voeg de letter toe aan de letters die al gebruikt zijn
8. Zit de letter in het te raden woord?
   Zo ja: toon het woord in * tekens waarin je de geraden letters wel laat zien
   Zo nee: hoog het aantal foutieve beurten op met 1 en toon de juiste afbeelding van de galg
9. Zijn alle letters geraden? De speler heeft gewonnen
10. Heeft de speler 8 foutieve beurten? De speler heeft verloren
11. In alle andere gevallen: terug naar stap 4

## De ImageFactory klasse

1. Maak een klasse 'ImageFactory' aan in de hangman map. Deze klasse zal de 8 'hangman' afbeeldingen bevatten (een leeg canvas, de voet, de voet en de paal etc.)
2. Geef de klasse een statische variabele IMAGES van het type String[][] en vul deze met de 8 'hangman' afbeeldingen (een voorbeeld van de afbeeldingen is te vinden in de 'examples' map. Iedere afbeelding bestaat uit meerdere strings die (wanneer je ze regel voor regel print in de terminal) de afbeelding vormen.
3. De klasse heeft een statische methode 'getImage' met als parameter een int (het aantal foutieve beurten). Geef daarmee de juiste afbeelding terug uit de array van afbeeldingen.

## Klassendiagram



# Interfaces

In deze oefening, die om interfaces draait, ga je alle spellen samenvoegen zodat je ze allemaal kunt spelen zonder dat je steeds je code hoeft aan te passen. Je gaat daarnaast ook een muntensysteem aan alle spellen toevoegen zodat het wat meer op een casino lijkt.

## Voorbereiding

1. Heb je de vorige opdracht afgerond? Dan is dat ook meteen het startpunt van deze opdracht. Wanneer dit niet zo is kun je de code in de map 'solution-5' gebruiken als startpunt.

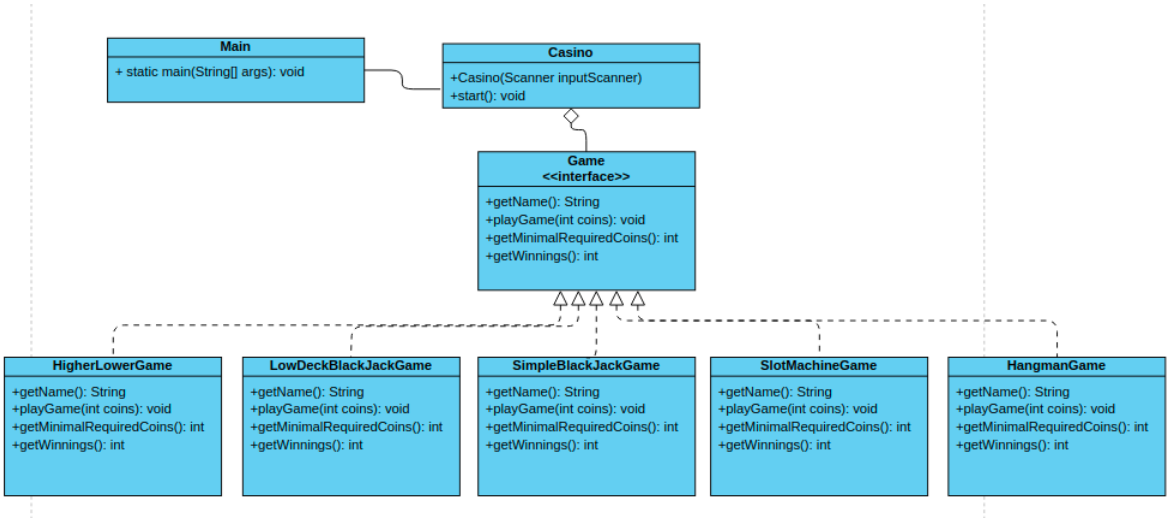
## Stappenplan

1. Maak een interface 'Game' aan in de novi map en definieer daarin de methodes die je in het klassendiagram terugziet.
2. Zorg er voor dat alle 'game' klassen die je in de vorige opdrachten hebt aangemaakt deze interface op de juiste manier implementeren.
3. Maak een 'Casino' klasse aan in de novi map en pas de main methode aan zodat er een object van deze klasse wordt aangemaakt en de 'start' methode wordt aangeroepen.
4. Zorg er voor dat de 'casino' klasse de hieronder beschreven logica bevat.

## De Casino klasse

1. Wanneer de speler de applicatie start ziet hij een welkomstbericht
2. De speler krijgt 1000 munten om mee te spelen
3. De speler kan kiezen om een spel te spelen (door de letter 'p' in te typen) of om het spel af te sluiten met de letter 'q'.
4. Kiest hij voor spelen dan krijgt hij een overzicht van de spellen die hij kan spelen. Dit zijn enkel de spellen waar hij op dat moment nog genoeg munten voor heeft.
5. De speler kiest een spel
6. De speler speelt het spel
7. Na afloop wordt de winst (of verlies) van de speler opgehaald bij het spel, deze wordt opgeteld bij het totaal aantal munten van de speler. Het resultaat en de huidige muntenvoorraad wordt aan de speler getoond.
8. Terug naar stap 3

## Klassendiagram



# Maven

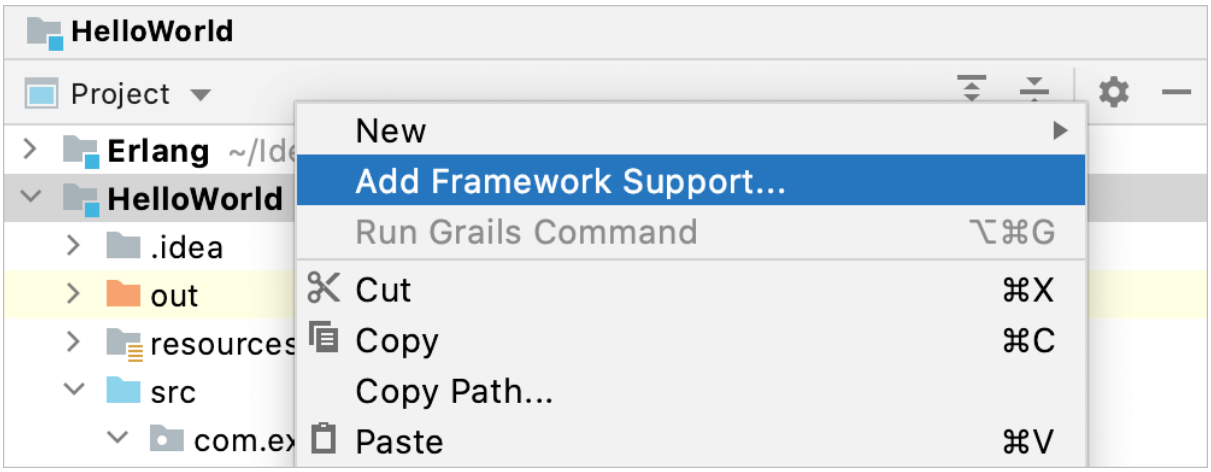
Functioneel is de applicatie nu zo goed als af, we kunnen hem enkel nog wat opleuken. In deze opdracht ga je dat doen door wat kleur aan de applicatie toe te voegen. Daarvoor ga je een externe library gebruiken die je via Maven gaat toevoegen aan de applicatie.

## Voorbereiding

1. Heb je de vorige opdracht afgerond? Dan is dat ook meteen het startpunt van deze opdracht. Wanneer dit niet zo is kun je de code in de map 'solution-6' gebruiken als startpunt.

## Maven installeren & configureren

1. Indien je maven nog niet op je computer geïnstalleerd hebt, dien je dit eerst te doen. Gebruik je Intellij voor het draaien van je project dan wordt de standaard Maven versie gebruikt die met Intellij wordt meegeleverd. In andere gevallen dien je Maven zelf te installeren (link).
2. Je kunt Maven vrij eenvoudig toevoegen aan een project.
    a) Open je project in Intellij
    b) Druk met de rechtermuisknop op de map van je project (links bovenin)
    c) Klik op 'Add framework support...'
    d) Vink 'Maven' aan en klik op OK
    e) Er wordt nu automatisch een pom.xml aan je project toegevoegd. Je dient hier nog wel het groupId en artifactId in te vullen.



## De library

Voor het toevoegen van kleur aan de applicatie ga je de library 'JColor' gebruiken. Voeg de library toe aan je pom.xml, de exacte dependency vind je hier. Zorg er daarna voor dat Maven de dependency ophaalt. Intellij toont hiervoor in je scherm een 'herlaad' icoontje wanneer je iets aan je pom hebt aangepast, dit haalt ook alle dependencies op.

Gebruik daarna de library om:

1. De introtekst bij het opstarten van je applicatie (Welkom bij het casino!) wat kleur te geven
2. De melding dat een speler iets gewonnen heeft groen te maken
3. De melding dat een speler iets verloren heeft rood te maken

Hoe dit moet vind je in de handleiding van de library op Github (link).

# Unittesten met JUnit

De applicatie is nu echt af. Het enige wat nog mist, is een goede set aan unit tests waarmee we kunnen garanderen dat de applicatie blijft werken zoals we verwachten. In deze oefening ga je daar een begin mee maken door het toevoegen van unit tests voor de 'HigherLowerGame' klasse.

## Voorbereiding

1. Heb je de vorige opdracht afgerond? Dan is dat ook meteen het startpunt van deze opdracht. Wanneer dit niet zo is kun je de code in de map 'solution-7' gebruiken als startpunt.
2. Voeg alvast de Maven dependency 'junit-jupiter-engine' (link) toe aan je pom.xml en herlaad maven. Met deze library ga je unit tests schrijven. Raadpleeg opdracht 7 als je niet meer precies weet hoe dit moet.

## Unit tests

Wanneer we het hebben over de 'unit' in unit tests bedoelen we daar het kleinst mogelijke stukje van de applicatie mee. Wat het kleinste stukje is kan verschillen maar zal in veel gevallen een enkele klasse zijn. We willen testen of alle publieke methodes van de klasse, in alle mogelijke scenario's het juiste resultaat teruggeven. Daarmee kunnen we garanderen dat de klasse onder alle omstandigheden werkt zoals verwacht.

Als we kijken naar de 'HigherLowerGame' klasse zie we dat die vier publieke methodes heeft: getName, getMinimalRequiredCoins, playGame en getWinnings. Wanneer we deze klassen willen unit testen zullen we dus tests moeten schrijven die het resultaat van het aanroepen van deze vier methodes in alle mogelijke scenario's valideert.

## getName & getMinimalRequiredCoins

De publieke methodes 'getName' en 'getMinimalRequiredCoins' bevatten slechts één codepad, ze geven altijd direct een resultaat terug. Dat betekent dat we dus slechts één test per methode nodig hebben om ze compleet te kunnen testen.

1. Java unit tests plaats je over het algemeen in dezelfde map als de klasse die je wilt testen, maar dan niet in de 'src' map maar in de 'test' map. Maak dus een klasse 'HigherLowerGameTest' aan in de 'test/java/novi/higherlower' map.
2. Definieer hierin een klasse 'HigherLowerGameTest'
3. Voeg nu twee tests (methodes) toe aan deze klasse, in de naam neem je op wat je precies wilt testen:
    a) getNameShouldReturnHigerLower
    b) getMinimalRequiredCoinsShouldReturn5
4. Implementeer deze twee tests zodat de publieke methode die je wilt testen, wordt aangeroepen en de response waarde wordt gevalideerd met behulp van Junit.

## playGame & getWinnings

De publieke methodes playGame en getWinnings zijn iets complexer om te testen. Wat we willen testen, is dat het raden van het getal binnen een bepaald aantal beurten een prijs oplevert en je in andere gevallen niets wint. Gezien het getal dat geraden moet worden willekeurig bepaald wordt, is het nu onmogelijk om die twee flows in tests te vangen, je weet immers niet wat het nummer moet zijn.

Om dit probleem op te lossen en er daarmee voor te zorgen dat de 'HigherLowerGame' klasse een stukje testbaarder wordt, zul je ervoor moeten zorgen dat je in je test zelf het willekeurige nummer kunt bepalen. Dit doen we door gebruik te maken van een 'mock'.

In het klassendiagram zie je dat de HigherLowerGame klasse een tweede constructor parameter heeft gekregen: randomGenerator. Deze randomGenerator is een object die de interface IRandomGenerator implementeert met daarop één methode 'randomInt'. Van deze interface zijn twee varianten: de RandomGenerator klasse en de RandomGeneratorMock klasse. De eerste wordt gebruikt wanneer de applicatie draait en geeft een willekeurige waarde terug, de tweede variant kun je gebruiken bij het testen van de klasse en geeft een vaste waarde terug (bijv. 50) waarmee je eenvoudig kunt testen.

Voeg de RandomGenerator interface en klassen toe aan het project. Bij het aanmaken van een HigherLowerGame object in je main methode geef je een instantie van de RandomGenerator klasse terug, in je tests gebruik je de RandomGeneratorMock klasse.

Wanneer je dit hebt geïmplementeerd, is de willekeurige waarde dus niet langer willekeurig wanneer jouw unit test de 'playGame' methode aanroept. Nu moet je er alleen ook nog voor zorgen dat de 'inputScanner' die mee wordt gegeven aan het HigherLowerGame object in je test input geeft. We kunnen de test niet laten typen dus moeten we daar een ander mechanisme voor gebruiken.

Wanneer we een Scanner aanmaken geven we daar als parameter 'System.in' aan mee. System.in is een zogenaamde 'InputStream'. Standaard luistert hij naar de input die je intypt maar dat is te overschrijven. Met onderstaande code kunnen we 'System.in' een statische reeks aan waarden als input laten geven (de waardes zijn gescheiden door de '\n' tekens). Daarmee kunnen we de input van een gebruiker nabootsen.

```
System.setIn(new ByteArrayInputStream("4\n5\n50\n".getBytes()));
Scanner scanner = new Scanner(System.in);
```

Gebruik de 'RandomGeneratorMock' en het mechanisme voor het vullen van 'System.in' dat hierboven is beschreven om ook de playGame/getWinnings methodes te testen:

- Wanneer het getal binnen het gedefinieerde aantal pogingen wordt geraden dient de speler iets te winnen
- Wanneer het getal niet binnen het gedefinieerde aantal pogingen wordt geraden dient de speler niets te winnen.

Klassendiagram