Hjælpeguide til programmering i Unity

# Formål med guiden

At lære at programmere for første gange kan være udfordrende, og der er mange små ting at huske på, og som kan gå galt! Den her guide er lavet til at hjælpe med at introducere de vigtigste koncepter, og også virke som et lille opslagsværk for når man kommer i tvivl.

# Hvordan bruges guiden

Overskrifterne er skrevet, så du har det danske udtryk, efterfulgt af det engelske udtryk i parenteser. Alle engelske udtryk er skrevet i *kursiv*. Vigtige ord står med **fed** tekst. Er der en streg under, kan du klikke på den for at springe til hvor udtrykket er forklaret. Blå tekst med en streg under linker til en hjemmeside. Ex:

Tema (*theme*)

Tekst om temaet. På engelsk siger man *theme*. Her er et **vigtigt** ord. Du kan klikke på [**tema**](#Tema_link_eksempel) for at springe til overskriften for tema. Det her link leder til [Unity’s *tutorials*](https://unity3d.com/learn/tutorials).

Kasser med kodeksempler er skrevet, så de ser ud som hvis det var i Visual Studio med mørk tema. Ex:

public void hello()

{

string greeting = "Hello pirates! ";

int someNumber = 5;

Debug.Log(greeting + "This is a number: " + someNumber);

}

Indhold

[Formål med guiden 1](#_Toc535085197)

[Hvordan bruges guiden 1](#_Toc535085198)

[Programmering i korte træk 4](#_Toc535085199)

[Begreber i C# programmering 4](#_Toc535085200)

[Variabel (*variable*) 4](#_Toc535085201)

[Datatype (*data type*) 5](#_Toc535085202)

[Navn (*name*) 5](#_Toc535085203)

[Synlighed (*access modifier*) 6](#_Toc535085204)

[Række (*array*) 6](#_Toc535085205)

[Liste (*list*) 6](#_Toc535085206)

[Operatorer (*operators*) 6](#_Toc535085207)

[Matematiske operatorer (arithmetic operators) 6](#_Toc535085208)

[Relationelle operatorer (*relational operators*) 7](#_Toc535085209)

[Logiske operatorer (*logical operators*) 7](#_Toc535085210)

[Tildelings operatorer (*assignment operators*) 8](#_Toc535085211)

[Andre operatorer 8](#_Toc535085212)

[Funktion (*function*) 8](#_Toc535085213)

[Hvornår bruger vi funktioner? 9](#_Toc535085214)

[Returtype (*return type*) 9](#_Toc535085215)

[Parameter (*parameter*) 10](#_Toc535085216)

[Kontrolsætninger og flow (*control statements and flow*) 10](#_Toc535085217)

[Hvis-sætning (*if statement*) 10](#_Toc535085218)

[Skifte-sætning (*switch statement*) 10](#_Toc535085219)

[Løkker (*loops*) 10](#_Toc535085220)

[Kodeblok (*code block*) 11](#_Toc535085221)

[Omfang (*scope*) 11](#_Toc535085222)

[Klasse (*class*) 11](#_Toc535085223)

[Objekt (*object*) 11](#_Toc535085224)

[Nedarving (*inheritance*) 12](#_Toc535085225)

[Begreber i Unity 12](#_Toc535085226)

[Projekter 12](#_Toc535085227)

[Scener 12](#_Toc535085228)

[Klasser I Unity 12](#_Toc535085229)

[Specielle funktioner i Unity 12](#_Toc535085230)

# Programmering i korte træk

Når vi skriver kode, skriver vi egentlig en serie af små opgaver til vores computer. Når vi så kører vores kode, vil vores computer prøve på at lave de opgaver for os, i den rækkefølge som de står i.

Computeren forventer, at opgaverne vi giver den er skrevet i på en bestemt måde, i præcis den rækkefølge det skal bruges, og at vi ikke beder den om at gøre noget der er umuligt.

På en måde kan en computer sammenlignes med en ekstremt lydig og intelligent hund, som gør NØJAGTIGT hvad du beder den om. Den tænker ikke på hvad du MENER, kun på det du SIGER.

Forestil dig, at du har den her hund, og du vil lege kaste-pind med den. Du kaster en pind, og siger til hunden at den skal tage den. Hunden løber efter pinden, finder den, tager den i munden… Men den kommer ikke tilbage med den. Den står bare stille bagefter. Du sagde jo ikke noget om, at den skulle komme tilbage med pinden, selv om det var egentlig det du gerne ville!

Problemet er, at hvis vi ikke fortæller computeren præcist den hvad den skal på den rigtige måde, så gør den højst sandsynligt ikke det vi forventer. I værste fald vil den prøve på at gøre noget, som den egentlig ikke kan, og så crasher vores kode.

Så der er to store udfordringer når vi programmerer. Den ene er at forstå hvordan en computer tænker, og den anden er at lære dens sprog. Begge dele tager tid, men er langt fra umuligt. At finde frem til svarene kan netop være det sjove i at programmere.

C# er en af mange mulige programmeringssprog. Unity som spilmotor er teknisk set skrevet i et andet, mere krævende sprog som hedder C++, men er bygget til at du kan styre den med C#, som er mere brugervenligt.

# Begreber i C# programmering

## Variabel (*variable*)

En variabel er et sted, hvor vi husker på information. Hvad vi husker på, kan være praktisk talt hvad som helst. Ofte skal vi huske på simple informationer. Engang imellem skal vi huske på hele objekter i spillet.

En variabel har fire ting: en [**synlighed**](#_Synlighed_(access_modifier)), en [**data type**](#_Data_type_(data), et [**navn**](#_Navn_(name)) og en **værdi**. Vi kan sagtens lave en variabel, som vi ikke giver en speciel værdi fra starten af, men så bliver den automatisk sat til at være en standard værdi. Det er dog en god idé at være eksplicit og skrive det fra starten hvis vi kan, så det står klart i koden.

Hvis variablen er af en [**primitiv**](#_Data_type_(data) type, og ikke får noget at vide, er det normalt ikke et problem. Tal bliver sat til at være ”0”, og tekst bliver tomme. 0 er dog stadig et tal, og et tomt stykke tekst husker stadig på at den er tom, så de kan stadig bruges.

public int nyHeltal = 10;

private string nyHemmeligTekst = "Hej pirat!";

public bool erPirat = true;

public int standardTal; // Den her vil blive til 0 automatisk

public string standardTekst; // Den her vil blive til ”” automatisk

public bool standardBool; // Den her vil blive til false automatisk

Er variablen af en [**sammensat**](#_Data_type_(data) type, så som en [**klasse**](#_Klasse_(class)_1), og får ingen værdi, bliver den sat til at være ”*Null*”. *Null* er et specielt udtryk, som betyder at der slet ikke er nogen information, ikke engang et 0. Den er simpelthen tom. Det kan forvirre vores spil hvis vi ikke passer på.   
Problemet er, at variabler peger på et sted i din computers hukommelse, og hvis en variabel er sat til *Null*, så peger den på ingenting! Prøver vi så at bruge den variabel alligevel, så får vi en bestemt fejl i vores spil, kendt som *NullReferenceException* – vi referer til noget, som ikke findes.

Vector3 test1 = new Vector3(); // Laver en Vector3, hvor alle tal er sat til 0

Vector3 test2 = new Vector3(5, 10); // Laver en Vector3, hvor x er 5, y er 10, og z er sat til 0

Vector3 test3 = new Vector3(5, 10, 20); // Laver en Vector3, hvor x er 5, y er 10, og z er 20

Vector3 test4; // Ups! Den her er tom, og er derfor null!

Vi kan dog bruge de [**relationelle operationer**](#_Relationelle_operatorer_(relational) ”==” og ”!=” for at se om noget er *null*, inden at noget går galt. Og nogle gange giver det mening at bruge *null* med vilje – bare man er forsigtig.

## Datatype (*data type*)

Der er mange former for informationer at holde styr på, så som tal, objekter, og farver. Alle de her typer af informationer kaldes for **datatyper**, og vi har ofte brug for at skrive hvilken datatype der er brug for.

De fleste informationer vi kommer til at bruge, er kaldt for **primitive** datatyper. Det er information som heltal, kommatal, ja/nej sætninger, bogstaver og ord.   
Her er en liste af meget almindelige, primitive data typer:

|  |  |  |  |
| --- | --- | --- | --- |
| Kode | *Engelsk navn* | Hvad er det | Eksempler |
| int | *integer* | Heltal | 1, 7, -13 |
| float | *floating points* | Kommatal, decimaler | 3.14f, -0.2f |
| char | *character* | Karakter, bogstav, tekstsymbol | ’a’, ’b’, ’c’, ’1’, ’2’, ’3’, ’!’, ’?’ |
| string | *string* | Tekst streng, serie af karakterer | ”Ord”, ”En sætning” |
| bool | *boolean* | Boolsk værdi,  sandt eller falsk | true, false |

Der findes flere, men for det meste er de her nok at kunne.

Vi kan også komme til at arbejde med **sammensatte** data typer, som er lavet af flere stykke information på én gang. En *Vector3* indeholder en sammensætning af 3 kommatal, en for x, y, og z-aksen i verdenen. En *Transform* er også sammensat af flere *Vector3* data-stykker. Og vi kan også selv lave dem som objekter gennem vores [**klasser**](#_Klasse_(class)).

## Navn (*name*)

Alle [**funktioner**](#_Funktion_(function))**,** [**variabler**](#_Variabel_(variable)) og [**klasser**](#_Klasse_(class)_1) skal have et unikt navn. Et godt navn beskriver også hvad en [**variabel**](#_Variabel_(variable)) indeholder, hvad en [**funktion**](#_Funktion_(function)) gør, eller hvad en **klasse** står for, uden at fylde for meget. Når du skal bruge dem senere, kalder du dens navn.

|  |  |
| --- | --- |
| Eksempel | Hvor godt er navnet? |
| int t; | Ikke særlig godt, fordi ”t” kan stå for hvad som helst! Kan misforstås som mange ting, der starter med t. |
| int tal; | Ikke særlig godt. Vi kan se at det må være et tal, men vi ved ikke hvad tallet betyder. |
| int bolcher; | Godt. Det er nemt at regne ud at det er antallet af bolcher. |
| int bolcherISlikskålen | Lidt for specifikt. Hvis variablen ligger inde i en klasse for slikskåle, behøver vi ikke skrive det i navnet. Du kommer også til at bliver træt af at skrive så langt et navn når du skal bruge variablen. |

Et navn må skrives med tal og bogstaver. Det må bare ikke starte med et tal.

Måden vi skriver navnet på kan hjælpe os med at holde styr på hvad det repræsenterer. En god tommelfingerregel er:

* Funktioner, klasser og objekter skal skrives med ”*PascalCase*”
  + alle ord i navnet skal starte med stort bogstav.
* Variabler og værdier skal skrives med ”*camelCase*”
  + alle ord i navnet, bortset for det første, skal starte med stort bogstav.

## Synlighed (*access modifier*)

Når vi skriver en ny variabel, funktion eller klasse, er det første vi skriver dens **synlighed** for resten af programmet.

Der er flere typer af synlighed, men de vigtigste som vi bruger, er ***public*** (offentlig), og ***private*** (privat).

|  |  |  |  |
| --- | --- | --- | --- |
| *Public* (offentlig) | *Internal* (intern) | *Protected* (beskyttet) | *Private* (privat) |
| Gør noget synligt for alt.  Er en variabel *public*, kan den ses og redigeres ude i Unity’s editor!  Er en funktion *public*, kan andre kodestykker kalde på den! | Gør at noget kun er synlig for den samling af kode den ligger i.  Skriver du ingen *access modifier* når der bør stå en, bliver *internal* brugt automatisk. | Gør at noget kun er synligt for den klasse den ligger i, og alt der arver fra den klasse.  Lige som med *private*, så hjælper det med at skjule detaljer for andre dele af koden, men holder det synligt for nedarvede klasser! | Skjuler noget for alt andet end præcis den klasse, som det står i.  Fungerer godt, hvis dit program skal huske noget selv, uden at nogen eller noget andet kan pille i det. |

## Rækker og lister (*array**s* *and* *lists*)

Der vil komme et tidspunkt, hvor du har brug for huske på mange ting af samme type, og måske er du ikke engang sikker på præcis hvor mange. De almindelige variabler kan dog kun huske på én ting af gangen, så hvad gør vi?

Der er et par løsninger til problemet, nemlig ***arrays*** og ***lists***. Begge er strukturer til at huske på store mængder af information på ét sted. De har hver deres styrker og svagheder, så hvilken en man skal bruge kommer an på formålet.

int[] talArray = { 0, 1, 2, 3 };

List<int> talListe = new List<int>() { 0, 1, 2, 3, 4 };

|  |  |  |
| --- | --- | --- |
|  | *Array* | *List* |
| At lave en ny variabel af typen | Nemt | Lidt sværere |
| At lave en ny variabel der arbejder i flere dimensioner | Nemt | Sværere |
| At ændre størrelsen på hvor meget den kan indeholde | Svært, langsomt for computeren | Nemt, hurtigt for computeren |
| At finde en værdi baseret på index | Hurtigt for computeren | Langsomt for computeren |
| At finde en bestemt værdi i samlingen | Langsomt for computeren | Hurtigt for computeren |

### Række (*array*)

*Arrays* er nok den nemmeste måde at lave en variabel, som kan huske på mange ting. Når du laver en *array*, ligger forskellen i at vi nu også skriver to firkantede parenteser, [ og ], bagefter datatypen, og at den skal have at vide hvor meget plads der skal reserveres. Det kan gøres enten ved at fortælle den præcist hvor meget plads der skal være, eller ved at give den et sæt tal på forhånd som den skal huske, som den så tæller. Får den 4 tal, laver den så 4 pladser.

For så at bruge vores *arrays*, skal vi fortælle hvilken plads vi vil have en værdi fra. Det gør vi ved at bruge et såkaldt **index**, som er et tal. Her skal vi holde tungen lige i munden, for computere tæller fra 0! Så skal vi have den første værdi i en array, skal vi bruge index 0! Så vi skriver *array* variablens navn, efterfulgt af firkantede parenteser, [ og ]. Inde imellem de firkantede parenteser skriver vi så vores index, så der til sidst står noget i retning af ”navn[0]”.

int[] tomArray; // Laver en variabel til et array, men den kan ikke bruges endnu. Den tæller som null

int[] nyArray = new int[4]; // Laver et array, der kan holde 4 tal, alle sat til 0

int[] fyldtArray = { 1, 2, 4, 8 }; // Som før, men pladsen er baseret på 4 tal der bliver gemt

int[,] array2D = new int[2, 2]; // Laver et 2-dimensionelt array, som har plads til 2 sæt af 2 tal

int[,] fyldtArray2D = { { 1, 2 }, { 4, 8 } }; // Laver et 2-dimensionelt array, som er fyldt med tal

int resultat;

resultat = fyldtArray[3]; // Variablen resultat får tallet fra index 3 i fyldtArray, som er 8

resultat = fyldtArray2D[0, 1]; // Variablen resultat får tallet fra index (0, 1) i fyldtArray2D, som er 2

fyldtArray[0] = 50; // Sætter tallet i index 0, der før var 1, til at være 50

fyldtArray2D[1, 1] = 50; // Sætter tallet i index (1, 1), der før var 1, til at være 50

Arrays kan også arbejde i flere dimensioner. Det betyder, at hvis du forestiller dig din normale *array* som én række af værdier, så er en 2-dimensionel *array* en stabel af rækker. For at bruge en flerdimensionel *array*, så sætter du bare kommaer inde imellem firkantparenteserne, og følger ellers sammen logik.

Et bibliotek er en god abstraktion for flerdimensionelle *arrays*. En enkelt boghylde kan ses som en 1-dimensionel *array*, der holder bøger. En bogreol har flere hylder, så den virker som en 2-dimensionel *array* af bøger. Har vi mange bogreoler, har vi en 3-dimensionel *array* af bøger. Og hvis vi tæller alle biblioteker i Danmark med, kommer vi op på 4 dimensioner!

Et problem der dog er med *arrays* er, at hvis du kommer til at have en værdi mere end der er plads til på dit eksisterende *array*, så bliver det svært at gøre den større midt i at alting kører. Kort fortalt prøver computeren på at reservere en række af plads på dens hukommelse, som er lige så stor som du bad den om at være da du lavede dit *array*. De reserverede pladser ligger tæt sammen, og er ikke lige til at skubbe rundt.   
Så den eneste måde for din computer at kunne ændre på størrelsen af dit *array* er at lave den helt forfra et nyt sted i hukommelsen med en ny størrelse, hvor den kopierer alting i den gamle *array* over i den nye. Det er en træls opgave, der for computeren kan tage lang tid!

### Liste (*list*)

## Operatorer (*operators*)

En **operator** er en kommando til din kode, som får den til at udregne simple logiske opgaver eller matematik. Du har højst sandsynligt set et par stykker allerede, uden at vide det!

De næste par sektioner har de mest almindelige operatorer, men der er flere.

### Matematiske operatorer (arithmetic operators)

På dansk er det egentlig ”aritmatiske operatorer”. Aritmetik er en gren af simpel matematik.

De her operatorer laver en matematisk behandling mellem to værdier. De virker primært på værdier, der kan håndteres som tal.

|  |  |  |
| --- | --- | --- |
| Operator | Beskrivelse | Eksempler (A = 9, B = 20, C = 9) |
| + | Lægger to værdier sammen | A + B = 29  A + C = 18 |
| - | Trækker den anden værdi fra den første | A – B = -11  A - C = 0 |
| \* | Ganger de to værdier sammen | A \* B = 180  A \* C = 81 |
| / | Dividerer den første værdi med den anden | A / B = 0,45  A / C = 1 |
| % | Modulo – laver en heltals-division, og giver dig resten. | A % B = 9  B % C = 2 |
| ++ | Forøg en værdi med 1.  Kan bruges på to måder:  ++X lægger 1 til X først, og giver dig X bagefter. X++ giver dig X først, og lægger 1 til X bagefter. | A++ = 10 (men giver dig 9 først)  ++B = 21 |
| -- | Formindsk en værdi med 1  Kan bruges på to måder:  --X trækker 1 fra X først, og giver dig X bagefter. X-- giver dig X først, og trækker 1 fra X bagefter. | A-- = 8 (men giver dig 9 først)  --B = 19 |

NOTE: Bruger du + mellem tekststrenge, lægger du dem efter hinanden til én tekststreng. Enkelte bogstaver kan også lægges på en tekststreng på den måde, men to bogstaver lagt sammen virker anderledes. Hver bogstav er, et sted i baggrunden, håndteret som tal, så lægger du bogstav nummer 9 med bogstav nummer 20, får du bogstav nummer 29!

### Relationelle operatorer (*relational operators*)

De her operatorer ser på to værdier i relation af hinanden, og giver dig et ja/nej svar.

|  |  |  |
| --- | --- | --- |
| Operator | Beskrivelse | Eksempler (A = 9, B = 20, C = 9) |
| == | Ser på om de to værdier er lig med hinanden | (A == B) giver false  (A == C) giver true |
| != | Ser på om de to værdier ikke er lig med hinanden | (A != B) giver true  (A != C) giver false |
| < | Ser på om den første værdi er mindre end den anden | (A < B) giver true  (A < C) giver false |
| > | Ser på om den første værdi er større end den anden | (A > B) giver false  (A > C) giver true |
| <= | Ser på om den første værdi er mindre end eller lig den anden | (A <= B) giver true  (A <= C) giver true |
| >= | Ser på om den første værdi er større end eller lig den anden | (A >= B) giver false  (A >= C) giver true |

NOTE: Bruger du == eller != på objekter, vil de ikke give dig svaret på om de er identiske eller ej, men om det er præcist det samme objekt eller ej!

### Logiske operatorer (*logical operators*)

De her operatorer arbejder med ja/nej værdier – boolske værdier – og giver et ja/nej som svar.

|  |  |  |
| --- | --- | --- |
| Operator | Beskrivelse | Eksempler  (A = true, B = false, C = true) |
| && | OG (*AND*) operator – ser om begge værdier er true | (A && B) giver false  (A && C) giver true |
| || | ELLER (*OR*) operator – ser om én af værdierne er true | (A || B) giver true  (A || C) giver true |
| ! | IKKE (*NOT*) operator – giver det modsatte af værdien | (!B) giver true  (!A) giver false  (!(A && C)) giver false |

### Tildelings operatorer (*assignment operators*)

De her operatorer arbejder med at tildele værdier fra højre mod venstre.

|  |  |  |
| --- | --- | --- |
| Operator | Beskrivelse | Eksempler (A = 9, B = 20, C = 9) |
| = | Simpel tildeling (*simple assignment*).  Sætter værdien til venstre til at være det samme som den til højre. | (A = B) giver A værdien 20  (B = C) giver B værdien 9 |
| += | Sammenlæg og tildel (*Add and assign*) Lægger de to værdier sammen, og gemmer resultatet i værdien til venstre. | (A += B) giver A værdien 29  (B += C) giver B værdien 29 |
| -= | Træk fra og tildel (*Subtract and assign*)  Trækker værdien til højre fra den til venstre, og gemmer resultatet i værdien til venstre. | (A -= B) giver A værdien -11  (B -= C) giver B værdien 11 |
| \*= | Gang og tildel (*Multiply and assign*) Gange de to værdier sammen, og gemmer resultatet i værdien til venstre. | (A \*= B) giver A værdien 180  (A \*= C) giver A værdien 81 |
| /= | Divider og tildel (*Divide and assign*) Divider værdien til venstre med værdien til højre, og gem resultatet i værdien til venstre. | (A /= B) giver A værdien 0,45  (A /= C) giver A værdien 1 |
| %= | Modulo og tildel (*Modulo and assign*) Laver en heltals-division af værdien til venstre med værdien til højre, og gemmer resten i værdien til venstre. | (A %= B) giver A værdien 9  (B %= C) giver B værdien 2 |

### Andre operatorer

De her operatorer falder lidt uden for kategori, men er meget vigtige at kende.

|  |  |  |
| --- | --- | --- |
| Operator | Beskrivelse | Eksempler |
| .  x.y | *Member access*. At skrive punktum efter x giver dig adgang til y - værdier og funktioner - inde i en klasse eller objekt. | Transform.Position;  Time.deltaTime; |
| new | *Type instantation*. Fortæller at der skal skabes et helt nyt objekt. | Vector3 v = new Vector3(); |
| ? :  t?x:y | *Conditional expression*. En anden måde at skrive en *if-statement*.  Hvis testen t giver true, bruger den x, ellers bruger den y. | (false ? ”ja” : ”nej”) giver ”nej”  (4 < 5 ? 4 : 5) giver 4, fordi (4 < 5) giver true |

## Funktion (*function*)

Funktioner kan ses lidt som maskiner, som laver bestemte opgaver for os. Nogle maskiner gør kun én ting, nogle gør mange ting samtidigt, nogle skal ”fodres” med noget for at virke, nogle giver dig noget tilbage, eller måske en blanding.

En funktion i programmering virker på samme måde. Når vi **kalder på en funktion** i vores spil, laver den en bestemt opgave for os. En funktion kan også kalde på andre funktioner. En stor opgave kan jo indeholde mange små opgaver der skal løses!

En funktion der er skrevet rigtigt har et par ting: en [**synlighed**](#_Synlighed_(access_modifier)), en [**retur type**](#_Retur_type_(return), et [**navn**](#_Navn_(name)), to parenteser, og en [**kode blok**](#_Kode_blok_(code), hvor dens opgave står i. Nogle funktioner bruger også [**parametre**](#_Parameter_(parameter)), som er ekstra information. De skal stå inde i parenteserne.

Ser vi på sodavands automaten igen, kan vi skrive hvad den kan med funktioner. Automaten selv kan beskrives med en [**klasse**](#_Klasse_(class)_1), hvor dens funktioner og indhold er beskrevet indeni.

At købe en sodavand er den primære funktion, og det giver mening at den hedder ”KøbSodavand”. Automaten kan bruges af alle, så funktionen kan ses som [*public*](#_Synlighed_(access_modifier)). Den skal også give en sodavand tilbage, så dens returtype er ”Sodavand”, som er en [klasse](#_Klasse_(class)_1) der er beskrevet et andet sted.  
For at købe en sodavand, skal den have to parametre: penge og nummeret på den sodavand du vil have.   
Maskinen spørger så sig selv med en funktion, om der er flere sodavand tilbage af den type. Vi kan gætte, at den har [returtypen](#_Retur_type_(return) *boolean*, fordi svaret er enten ja eller nej.  
Hvis automaten er løbet tør for den sodavand, siger den ”Løbet tør” på skærmen.   
Har du givet den for lidt penge, siger den ”Ikke nok penge”.   
Men er der nok af begge, trækker den prisen fra, og kalder på en anden funktion, der finder den rigtige sodavand, og giver dig den.   
I alle tre tilfælde, hvis der er penge til overs, bruger den en funktion der hedder GivPengeTilbage til at betale dem tilbage. Den funktion er højst sandsynligt sat til at være [*private*](#_Synlighed_(access_modifier)), for at folk ikke bare kan tage penge ud af automaten.

public Sodavand KøbSodavand(float penge, int sodavandsNummer)

{

Sodavand resultat = null;

if (SodavandTilbage(sodavandsNummer))

{

Debug.Log("Løbet tør.")

}

else if(penge < 20.0f)

{

Debug.Log("Ikke nok penge.");

}

else

{

penge -= 20.0f;

resultat = GivSodavand(sodavandsNummer);

}

if(penge > 0)

{

GivPengeTilbage(penge);

}

return resultat;

}

### Hvornår bruger vi funktioner?

Som vores spil og programmer bliver store, så kommer vi til at skrive meget kode. Funktioner er en god måde at indramme bestemte opgaver, så vi undgår at lave en kæmpe, uforståelig ”megakode”. Det er nemmere at læse navnene på en serie af funktioner efter hinanden, end koden indeni funktionerne! I programmering er der faktisk et udtryk der hedder ”spaghettikode”, som betyder at koden er lang, rodet sammen, og svær at rede ud.

En anden god grund er, at vi ofte kommer til at skulle gøre den samme opgave mange gange, og nogle gange kan vi ikke være sikker på præcis hvor mange gange vi skal gøre det. Hvis vi skriver opgaven som en funktion, behøver vi kun skrive opgaven én gang, og så bare kalde den funktion.

### Returtype (*return type*)

En funktions returtype bestemmer hvad funktionen skal give tilbage når den er færdig med at køre. Returtypen kan være mange forskellige ting, afhængig af hvad vi har brug for, og det er lige før at der ingen grænser er! Du kan nemlig bruge en hvilken som helst [**datatype**](#_Data_type_(data) som returtype!

Der er to specielle ting man skal huske om returtyper.

Den første er, at returtypen også kan være ”*void*”, som i den her kontekst betyder ”ingenting”. Void bruger du hvis at funktionen bare skal gøre noget, uden at give noget tilbage.

Den anden ting er, at hvis du bruger en anden returtype end *void*, så skal du huske at skrive i funktionen hvorfra og hvornår den skal give resultatet. Det gør man ved at bruge kommandoen ”*return*”, efterfulgt af hvad du vil give som resultat. *Return* har også den ekstra effekt, at den stopper funktionen med det samme.

### Parameter (*parameter*)

Når vi skriver og bruger funktioner, skal vi ofte tænke på hvad funktionen skal bruge af informationer for at virke. Informationer som vi giver direkte til en funktion kaldes for **parametre**. Ikke alle funktioner har dog brug for parametre.

Når vi skriver en funktion, skriver vi parametre inde i de to parenteser. Måden de skrives på, er lidt ligesom at skrive almindelige variabler, men hvor de er delt op med komma. Hver parameter skal have en [**datatype**](#_Datatype_(data_type)) og et [**navn**](#_Variabel_(variable)). Rækkefølgen som parametrene bliver skrevet i er vigtig at holde styr på, for den samme rækkefølge skal man bruge når man kalder på funktionen senere.   
Når parametrene er definerede for funktionen, sker der noget spændende. Vi kan nu bruge de parametre som normale variabler inden for funktionens [**kodeblok**](#_Kodeblok_(code_block))! Det er også et eksempel af [***scope***](#_Omfang_(scope)).

Når vi kalder en funktion med parametre, skal vi give den de værdier som parametrene kræver, og i den samme rækkefølge som de er skrevet i.

## Kontrolsætninger og flow (*control statements and flow*)

### Hvis-sætning (*if statement*)

### Skifte-sætning (*switch statement*)

### Løkker (*loops*)

En løkke i programmering er en kontrolsætning, som gentager den samme opgave flere gange efter hinanden, indtil den får at vide at den skal stoppe.

Løkker er praktiske til mange ting. De kan bruges til at tælle, at lede efter noget specielt i en samling af ting, at få alle ting i en samling til at gøre noget specielt osv.

Unity’s *Update()*-funktion bliver kaldt inde i en løkke automatisk af Unity.

Man skal dog være forsigtig, for man kan godt skrive løkker som ikke kan stoppe igen, og får vores spil til at gå i stå, fordi den venter på at løkken bliver færdig!

#### For-løkke (for loop)

#### For-hver-løkke (for each loop)

#### Imens-løkke (while loop)

#### Gør-imens-løkke (do while loop)

## Kodeblok (*code block*)

En kodeblok er en sektion af kode, som hænger sammen, og er indrammet af to krølleparenteser, { og }.

Funktioner, klasser og kontrol flow bruger kodeblokke til at separere hvilken kode høre til den.

### Omfang (*scope*)

## Klasse (*class*)

En klasse kan ses som en beskrivelse af en ting i dit spil. Den har information omkring:

* Hvad tingen indeholder – [**variabler**](#_Variabel_(variable)) – som antal ben på et dyr
* Hvad den kan gøre – [**funktioner**](#_Funktion_(function)) – dyret kan gå
* Hvordan den gør det – koden i en [**funktion**](#_Funktion_(function)) – hvordan den bevæger benene efter hinanden.

Klasser kan være:

* Abstrakte – klassen beskriver ikke et bestemt dyr
* Specifikke – klassen beskriver en hund
* Præcise – klassen beskriver en helt unik schæfer-hund, som hedder Rufus

Alle kan være brugbare, afhængig af hvad du skal bruge. Vi kan sagtens lave mange hunde og katte i et spil, men bruge den samme generelle kode for et dyr med ben, og nogle gange skal vi kun bruge en kode til en enkelt, speciel hund, ligesom Rufus.

Tænk på hvordan vi styrer vores bold i Roll-A-Ball. Vi har et *script*, som indeholder en klasse, der hedder *PlayerController*. Den klasse er en beskrivelse af kontrol-systemet på spillerens bold, og indeholder for eksempel en [**variabel**](#_Variabel_(variable)) for fart (*speed)*, og en [**funktion**](#_Funktion_(function)) for hvad den skal gøre med jævne mellemrum hele tiden, *FixedUpdate()*.

For at din kode virker rigtigt i dine spil, skal de tilhøre en klasse. Altså, skal det hele være inde for klassens [**kodeblok**](#_Kode_blok_(code).

Der er også en anden vigtig ting når vi arbejder i Unity

//HUSK AT SKRIVE OM FILNAVN

## Objekt (*object*)

Hvis en [**klasse**](#_Klasse_(class)) er en beskrivelse af en ting, så er et **objekt** en ting der eksisterer i vores spil, baseret på vores beskrivelse! Så i stedet for at vi bare snakker om hvordan et dyr ser ud og hvad den kan, så er der et **objekt** i spillet, som ER dyr, som gør ting ud fra vores kode.

Alle ting, som vi skaber inde i spillet, er **objekter**, som er baseret på beskrivelser fra [**klasser**](#_Klasse_(class)). Så lige så snart du lægger et *script* med en [**klasse**](#_Klasse_(class)) på en ting i dit spil, så kobler du en beskrivelse af den type på. Én ting kan dog godt havde flere typer og komponenter på samme tid i Unity.

## Nedarving (*inheritance*)

# Begreber i Unity

## Projekter

## Scener

## Klasser I Unity

## Specielle funktioner i Unity

Unity laver en masse arbejde i baggrunden for os, og har nogle funktions-navne for os, som er gode at kende. De sker også i en helt bestemt rækkefølge. Du kan læse mere om dem på [Unity’s egen hjemmeside](https://docs.unity3d.com/Manual/ExecutionOrder.html), men her er en kort beskrivelse af de vigtigste, og hvornår Unity selv kalder på dem.

|  |  |  |
| --- | --- | --- |
| Funktionsnavn | Hvornår | Formål |
| **Når en scene bliver hentet…** | | |
| Awake() | Når en scene starter, og efter et objekt bliver skabt.  Er objektet inaktivt når spillet startet, kører det først når det bliver aktiveret. | Brug når et objekt skal gøre noget når det først kommer ind i spillet.  Vigtig detalje! Når spillet starter, så kører Awake() før Start()! |
| OnEnable() | Lige efter et objekt er aktiveret, og når en scene starter, hvis objektet er aktivt. | Brug når et objekt skal gøre noget specielt når den bliver aktiveret. |
| **Før første frame…** | | |
| Start() | Når spillet først starter.  Kører IKKE hvis objektet er skabt senere inde i spillet. | Brug til at gøre nogle ting når spillet starter, fx gemme hvor forskellige ting var da spillet startede. |
| **Loops…** | | |
| FixedUpdate() | Kører med et fast interval, sammen med alle fysiske beregninger.  Alle kollisioner bliver håndteret her. | Skal der ske noget fysisk, fx du skal skubbe en bold? Gør det her!  Vigtig detalje! FixedUpdate() bliver tjekket før den normale Update(), og kan blive kaldt flere gange før den anden, hvis din FPS er lav. Men har du høj FPS, kan den godt springe det over. |
| Update() | Kører lige så hurtig som din computer kan gøre det, altså din FPS. | Det meste af alt du laver, som skal gentage sig, kan du bruge inde i Update(). |
| LateUpdate() | Kører efter den normale Update(). | Vil du have at noget reagerer på alt der lige er sket i spillet, fx et kamera skal følge efter spilleren, *efter* de har bevæget sig, så gør det her. |
| **Ved slutningen af et spil-loop…** | | |
| OnDisable() | Når et objekt bliver deaktiveret.  Kører også når et objekt skal destrueres. | Brug når et objekt skal gøre noget specielt når det bliver deaktiveret. Fx et kannon-skud skal lave en eksplosion lige inden skuddet forsvinder. |
| OnDestroy() | Når et objekt bliver destrueret og fjernet i løbet af spillet. | Er der lige noget information vi vil justere inden at vi fjerner noget fra spillet? Brug den her. |