# 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** for at springe til overskriften for tema. Det her link leder til Unity's *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

# 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**, en **data type**, et **navn** 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** 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** type, så som en **klasse**, 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.

```
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!
```

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.

Vi kan dog bruge de **relationelle operationer** ”==” 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 former for informationer kaldes for **datatyper**.

De fleste informationer vi kommer til at bruge, er kaldt for **primitive** data typer. 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 | ’a’, ’b’, ’c’ |
| `string` | *string* | Tekst streng | ”Alfabet”, ”ord” |
| `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**.

## 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<br>A + C = 18 |
| `-` | Trækker den anden værdi fra den første | A – B = -11<br>A - C = 0 |
| `*` | Ganger de to værdier sammen | A * B = 180<br>A * C = 81 |
| `/` | Dividerer den første værdi med den anden | A / B = 0,45<br>A / C = 1 |
| `%` | Modulo – laver en heltals-division, og giver dig resten. | A % B = 9<br>B % C = 2 |
| `++` | Forøg en værdi med 1.<br>Kan bruges på to måder:<br>++X lægger 1 til X først, og giver dig X bagefter.<br>X++ giver dig X først, og lægger 1 til X bagefter. | A++ = 10 (men giver dig 9 først)<br>++B = 21 |
| `--` | Formindsk en værdi med 1<br>Kan bruges på to måder:<br>--X trækker 1 fra X først, og giver dig X bagefter.<br>X-- giver dig X først, og trækker 1 fra X bagefter. | A-- = 8 (men giver dig 9 først)<br>--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`<br>(A == C) giver `true` |
| `!=` | Ser på om de to værdier ikke er lig med hinanden | (A != B) giver `true`<br>(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`<br>(A > C) giver `true` |
| `<=` | Ser på om den første værdi er mindre end eller lig den anden | (A <= B) giver `true`<br>(A <= C) giver `true` |
| `>=` | Ser på om den første værdi er større end eller lig den anden | (A >= B) giver `false`<br>(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**<br>**(A = `true`, B = `false`, C = `true`)** |
|---|---|---|
| `&&` | OG (*AND*) operator – ser om begge værdier er `true` | (A && B) giver `false`<br>(A && C) giver `true` |
| `\|\|` | ELLER (*OR*) operator – ser om én af værdierne er `true` | (A \|\| B) giver `true`<br>(A \|\| C) giver `true` |
| `!` | IKKE (*NOT*) operator – giver det modsatte af værdien | (!B) giver `true`<br>(!A) giver `false`<br>(!(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*).<br>Sætter værdien til venstre til at være det samme som den til højre. | (A = B) giver A værdien 20<br>(B = C) giver B værdien 9 |
| `+=` | Sammenlæg og tildel (*Add and assign*)<br>Lægger de to værdier sammen, og gemmer resultatet i værdien til venstre. | (A += B) giver A værdien 29<br>(B += C) giver B værdien 29 |
| `-=` | Træk fra og tildel (*Subtract and assign*)<br>Trækker værdien til højre fra den til venstre, og gemmer resultatet i værdien til venstre. | (A -= B) giver A værdien -11<br>(B -= C) giver B værdien 11 |
| `*=` | Gang og tildel (*Multiply and assign*)<br>Gange de to værdier sammen, og gemmer resultatet i værdien til venstre. | (A *= B) giver A værdien 180<br>(A *= C) giver A værdien 81 |
| `/=` | Divider og tildel (*Divide and assign*)<br>Divider værdien til venstre med værdien til højre, og gem resultatet i værdien til venstre. | (A /= B) giver A værdien 0,45<br>(A /= C) giver A værdien 1 |
| `%=` | Modulo og tildel (*Modulo and assign*)<br>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<br>(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 |
|---|---|---|
| `.`<br>`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;`<br>`Time.deltaTime;` |
| `new` | *Type instantation*. Fortæller at der skal skabes et helt nyt objekt. | `Vector3 v = new Vector3();` |
| `? :`<br>`t?x:y` | *Conditional expression*. En anden måde at skrive en *if-statement*.<br>Hvis testen t giver `true`, bruger den x, ellers bruger den y. | (false ? ”ja” : ”nej”) giver ”nej”<br><br>(4 < 5 ? 4 : 5) giver 4, fordi (4 < 5) giver `true` |

## Nedarving (*inheritance*)

## Objekt (*object*)

Hvis en **klasse** 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**. Så lige så snart du lægger et *script* med en **klasse** 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.

### Navn (*name*)

Alle **funktioner, variabler** og **klasser** skal have et unikt navn. Et godt navn beskriver også hvad en **variabel** indeholder, hvad en **funktion** 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 eller funktion, 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* | *Internal* | *Protected* | *Private* |
|---|---|---|---|
| Gør noget synligt for alt. Er en variabel *public*, kan den ses og redigeres ude i Unity's editor!<br><br>Er en funktion *public*, kan andre kodestykker kalde på den! | Gør at noget kun er synlig for den samling af kode den ligger i.<br><br>Skriver du ingen *access modifier* til en variabel eller funktion, har den *internal* automatisk. | Gør at noget kun er synligt for den klasse den ligger i, og alt der arver fra den klasse.<br><br>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.<br><br>Fungerer godt, hvis dit program skal huske noget selv, uden at nogen eller noget andet kan pille i det. |

## 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**, en **retur type**, et **navn**, to parenteser, og en **kode blok**, hvor dens opgave står i. Nogle funktioner bruger også **parametre**, 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**, 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*. Den skal også give en sodavand tilbage, så dens returtype er ”Sodavand”, som er en klasse 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 *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*, 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;
}
```

### 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** 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*)

### Kode blok (*code block*)

### Krølleparentes (*curly bracket*)

## 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** – som antal ben på et dyr
- Hvad den kan gøre – **funktioner** – dyret kan gå
- Hvordan den gør det – koden i en **funktion** – 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** for fart (*speed)*, og en **funktion** 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 **kode blok**.

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

//HUSK AT SKRIVE OM FILNAVN

# Begreber i Unity

## Projekter

## Scener

## 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, 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.<br>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.<br>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.<br>Alle kollisioner bliver håndteret her. | Skal der ske noget fysisk, fx du skal skubbe en bold? Gør det her!<br>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.<br>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. |