

Objekt-Orienteret Programmering

Funktioner

Aslak Johansen asjo@mmmi.sdu.dk
Peter Nellemann pmn@mmmi.sdu.dk

September 20, 2024

Part 0:
Funktioner i Matematik

Funktioner i Matematik ▷ Definition

En udregning der tager noget:

- ▶ ***Input:*** En række af navngivne parametre der tildeles værdier.
- ▶ ***Output:*** En enkelt værdi.

Hver værdi er et tal.

Funktioner i Matematik ▷ Definition

En udregning der tager noget:

- ▶ ***Input:*** En række af navngivne parametre der tildeles værdier.
- ▶ ***Output:*** En enkelt værdi.

Hver værdi er et tal.

Vi siger at funktionen *mapper* inputtet til outputtet, eller at funktionen repræsenterer en *mapping* fra input til output.

Funktioner i Matematik ▷ Gymnasiematematik ▷ Lineære Funktioner

Lineære funktioner er funktioner der tager formen:

$$f(x) = a \cdot x + b \quad (1)$$

Funktioner i Matematik ▷ Gymnasiematematik ▷ Lineære Funktioner

Lineære funktioner er funktioner der tager formen:

$$f(x) = a \cdot x + b \quad (1)$$

En konkret lineær funktion erstatter a og b i (1) med specifikke værdier:

$$f(x) = 2 \cdot x + 1$$

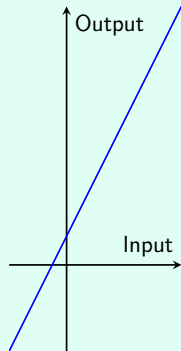
Funktioner i Matematik ▷ Gymnasiematematik ▷ Lineære Funktioner

Lineære funktioner er funktioner der tager formen:

$$f(x) = a \cdot x + b \quad (1)$$

En konkret lineær funktion erstatter a og b i (1) med specifikke værdier:

$$f(x) = 2 \cdot x + 1$$



Funktioner i Matematik ▷ Gymnasiematematik ▷ Lineære Funktioner

Lineære funktioner er funktioner der tager formen:

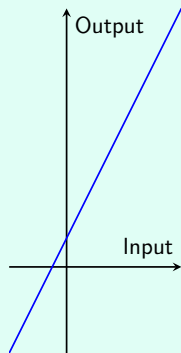
$$f(x) = a \cdot x + b \quad (1)$$

En konkret lineær funktion erstatter a og b i (1) med specifikke værdier:

$$f(x) = 2 \cdot x + 1$$

I C# er en ækvivalent:

```
double f (double x) {  
    return 2*x-1;  
}
```



Funktioner i Matematik ▷ Gymnasiematematik ▷ Lineære Funktioner

```
double f (double x) {  
    return 2*x-1;  
}  
  
for (double x=-3 ; x<=3 ; x+=0.5) {  
    double y = f(x);  
    Console.WriteLine("f("+x+") = "+y);  
}
```

Funktioner i Matematik ▷ Gymnasiematematik ▷ Lineære Funktioner

```
double f (double x) {  
    return 2*x-1;  
}  
  
for (double x=-3 ; x<=3 ; x+=0.5) {  
    double y = f(x);  
    Console.WriteLine("f("+x+") = "+y);  
}
```

$$f(-3.0) = -7.0$$

$$f(-2.5) = -6.0$$

$$f(-2.0) = -5.0$$

$$f(-1.5) = -4.0$$

$$f(-1.0) = -3.0$$

$$f(-0.5) = -2.0$$

$$f(0.0) = -1.0$$

$$f(0.5) = 0.0$$

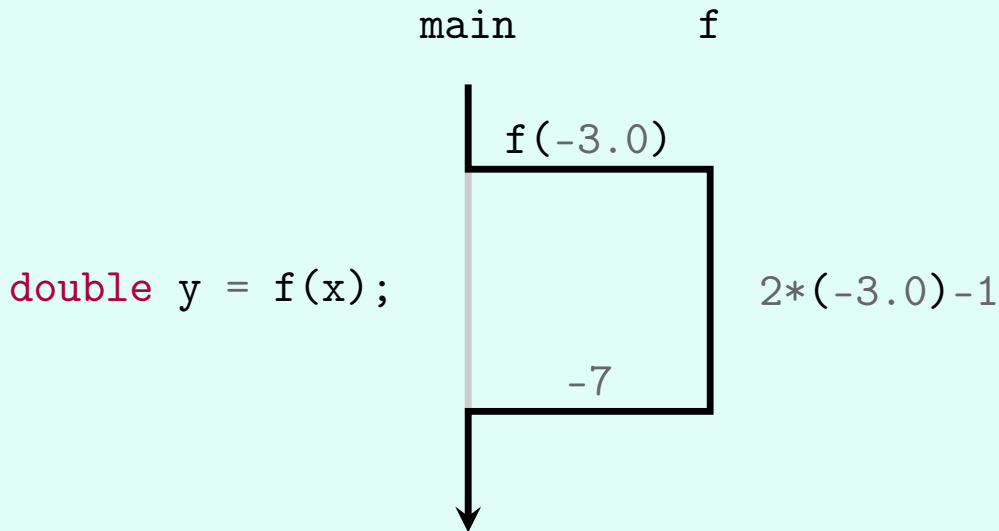
$$f(1.0) = 1.0$$

$$f(1.5) = 2.0$$

$$f(2.0) = 3.0$$

$$f(2.5) = 4.0$$

$$f(3.0) = 5.0$$



Funktioner i Matematik ▷ Gymnasiematematik ▷ Areal af Rektangel

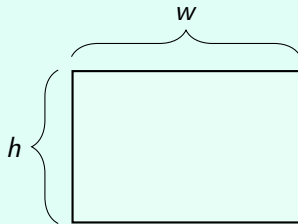
Arealet af et rektangel udregnes ud fra dets bredde og højde:

$$area(w, h) = w \cdot h$$

Funktioner i Matematik ▷ Gymnasiematematik ▷ Areal af Rektangel

Arealet af et rektangel udregnes ud fra dets bredde og højde:

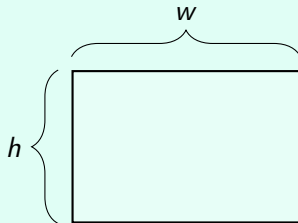
$$area(w, h) = w \cdot h$$



Funktioner i Matematik ▷ Gymnasiematematik ▷ Areal af Rektangel

Arealet af et rektangel udregnes ud fra dets bredde og højde:

$$area(w, h) = w \cdot h$$



I C# er en ækvivalent:

```
int area (int w, int h) {  
    return w*h;  
}
```

Funktioner i Matematik ▷ Gymnasiematematik ▷ Areal af Rektangel

```
int area (int w, int h) {  
    return w*h;  
}
```

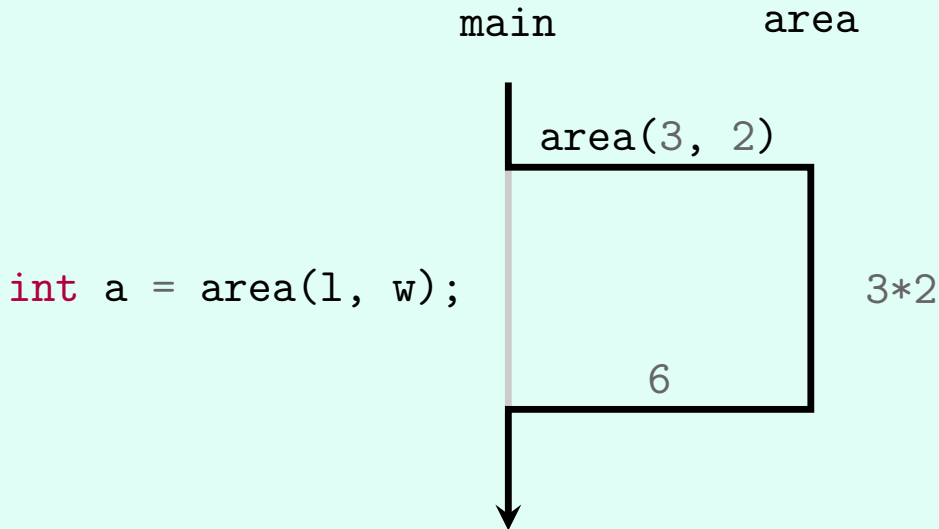
```
for (int h=0 ; h<=4 ; h+=1) {  
    for (int w=0 ; w<=8 ; w+=1) {  
        int a = area(w, h);  
        Console.Write(" {0,2}", a);  
    }  
    Console.WriteLine("");  
}
```

Funktioner i Matematik ▷ Gymnasiematematik ▷ Areal af Rektangel

```
int area (int w, int h) {  
    return w*h;  
}
```

```
for (int h=0 ; h<=4 ; h+=1) {  
    for (int w=0 ; w<=8 ; w+=1) {  
        int a = area(w, h);  
        Console.WriteLine(" {0,2}", a);  
    }  
    Console.WriteLine("");  
}
```

0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8
0	2	4	6	8	10	12	14	16
0	3	6	9	12	15	18	21	24
0	4	8	12	16	20	24	28	32



Part 1: Funktioner i C#

Funktioner i C# ▷ Introduktion


Funktioner er en sproglig konstruktion der muliggør genbrug af kode:

- ▶ Lignende kodelumper med ens ansvar identificeres.
- ▶ En generel *parameteriseret* udgave defineres i form af en funktion.
- ▶ Denne udgave udgør en abstraktion.
- ▶ De identificerede kodelumper kan da hver erstattes af et kald til denne funktion.
- ▶ **Resultat:** Delt logik kan defineres ét sted i stedet for at være spredt rundt som kopier i ens kode.
 - ▶ Hvorfor er dét at spredte kopier et problem?

Funktioner i C# ▷ En Funktions Anatomi

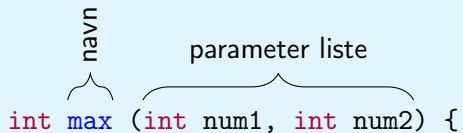
```
int max (int num1, int num2) {  
  
    int result;  
  
    if (num1 > num2) {  
        result = num1;  
    } else {  
        result = num2;  
    }  
  
    return result;  
}
```

Funktioner i C# ▷ En Funktions Anatomi



```
int max (int num1, int num2) {  
  
    int result;  
  
    if (num1 > num2) {  
        result = num1;  
    } else {  
        result = num2;  
    }  
  
    return result;  
}
```

Funktioner i C# ▷ En Funktions Anatomi



```
int max (int num1, int num2) {  
  
    int result;  
  
    if (num1 > num2) {  
        result = num1;  
    } else {  
        result = num2;  
    }  
  
    return result;  
}
```

Funktioner i C# ▷ En Funktions Anatomi

Diagram illustrating the components of a function signature:

```
int max (int num1, int num2) {
```

The components are labeled as follows:

- navn** (name): `max`
- parameter liste** (parameter list): `(int num1, int num2)`
- signatur** (signature): `int max (int num1, int num2)`
- return type**: `int`

The body of the function is shown below the signature:

```
    int result;
```

Funktioner i C# ▷ En Funktions Anatomi

return type navn parameter liste

`int` `max` (`int` num1, `int` num2) {

`int` result; signatur

```
    if (num1 > num2) {  
        result = num1;  
    } else {  
        result = num2;  
    }  
  
    return result;  
}
```


Funktioner i C# ▷ En Funktions Anatomi

return type navn parameter liste

`int` `max` (`int` num1, `int` num2) {

`int` result; signatur

```
if (num1 > num2) {  
    result = num1;  
} else {  
    result = num2;  
}
```

```
return result; } return statement  
}
```

Funktioner i C# ▷ En Funktions Anatomi

The diagram illustrates the components of a C# function definition. The function signature is `int max (int num1, int num2) {`. Brackets identify the parts: `int` is the **returtype** (return type), `max` is the **navn** (name), and `(int num1, int num2)` is the **parameter liste** (parameter list). The function body is enclosed in curly braces `{ ... }`. A bracket on the left labels the entire body as **funktionens krop** (function body). Inside the body, `int result;` is labeled as **signatur** (signature). The conditional logic `if (num1 > num2) { ... } else { ... }` is part of the body. The `return result;` statement is labeled as **return statement**.

```
int max (int num1, int num2) {  
    int result;  
  
    if (num1 > num2) {  
        result = num1;  
    } else {  
        result = num2;  
    }  
  
    return result;  
}
```

Funktioner i C# ▷ Anatomi ▷ Parametre

Kaldes ofte for *argumenter*.

Funktioner i C# ▷ Anatomi ▷ Parametre

Kaldes ofte for *argumenter*.

En funktion kan defineres til at modtage en vilkårligt antal parametre og der er ikke begrænsninger på hvilke typer disse kan erklæres som.

Funktioner i C# ▷ Anatomi ▷ Parametre

Kaldes ofte for *argumenter*.

En funktion kan defineres til at modtage en vilkårligt antal parametre og der er ikke begrænsninger på hvilke typer disse kan erklæres som.

Parametrene (med deres angivne rækkefølge og type) definerer sammen med funktionens navn funktionens *signatur* (som vi var inde på under funktionens anatomi).

Funktioner i C# ▷ Anatomi ▷ Signatur

Kombinationen af funktionens navn og rækkefølgen af parametertyper kaldes for funktionens *signatur*.

Funktioner i C# ▷ Anatomi ▷ Signatur

Kombinationen af funktionens navn og rækkefølgen af parametertyper kaldes for funktionens *signatur*.

```
int max (int num1, int num2) {  
    if (num1 > num2) {  
        return num1;  
    } else {  
        return num1;  
    }  
}
```

Funktioner i C# ▷ Anatomi ▷ Signatur

Kombinationen af funktionens navn og rækkefølgen af parametertyper kaldes for funktionens *signatur*.

```
int max (int num1, int num2) {  
    if (num1 > num2) {  
        return num1;  
    } else {  
        return num1;  
    }  
}
```

Signatur: max × (int, int)

Funktioner i C# ▷ Funktion Overloading

Man kan have flere funktioner med samme navn så længe deres signaturer er unikke:

- ▶ **Typer af parametre**
- ▶ Antal af parametre

Funktioner i C# ▷ Funktion Overloading

Man kan have flere funktioner med samme navn så længe deres signaturer er unikke:

- ▶ **Typer af parametre**
- ▶ **Antal af parametre**

```
// signatur: max * (int, int)  
int max (int num1,  
        int num2) {  
    return (num1 > num2 ? num1 : num2);  
}
```

```
// signatur: max * (double, double)  
double max (double num1,  
           double num2) {  
    return (num1 > num2 ? num1 : num2);  
}
```

```
int t1 = 1;  
int t2 = 0;  
double n1 = 1.0;  
double n2 = 3.0;
```

```
Console.WriteLine(max(t1, t2));  
Console.WriteLine(max(n1, n2));
```

Funktioner i C# ▷ Funktion Overloading

Man kan have flere funktioner med samme navn så længe deres signaturer er unikke:

- ▶ Typer af parametre
- ▶ **Antal af parametre**

Funktioner i C# ▷ Funktion Overloading

Man kan have flere funktioner med samme navn så længe deres signaturer er unikke:

- ▶ Typer af parametre
- ▶ **Antal af parametre**

```
// signatur: max * (int, int)
int max (int num1,
        int num2) {
    return (num1 > num2 ? num1 : num2);
}
```

```
// signatur: max * (int, int, int)
int max (int num1,
        int num2,
        int num3) {
    return max((num1 > num2 ? num1 : num2),
              num3);
}
```

```
int t1 = 1;
int t2 = 0;
int t3 = 4;
double n1 = 1.0;
double n2 = 3.0;
```

```
Console.WriteLine(max(t1, t2));
Console.WriteLine(max(t1, t2, t3));
```

Funktioner i C# ▷ Anatomi ▷ Krop

En funktions krop er en block der indeholder en sekvens af statements.

Funktioner i C# ▷ Anatomi ▷ Krop

En funktions krop er en block der indeholder en sekvens af statements.

En funktion ...

- ▶ har adgang til de værdier som funktionens parametre er bundet til under kaldet.
- ▶ kan producere en værdi som den returnerer.
- ▶ kan ændre på variable udenfor funktionen.

Funktioner i C# ▷ Anatomi ▷ Krop

En funktions krop er en block der indeholder en sekvens af statements.

En funktion ...

- ▶ har adgang til de værdier som funktionens parametre er bundet til under kaldet.
- ▶ kan producere en værdi som den returnerer.
- ▶ kan ændre på variable udenfor funktionen.

```
int[] fill (int[] array, int value) {  
    for (int i=0 ; i<array.Length ; i++) {  
        array[i] = value;  
    }  
    return array;  
}
```

Funktioner i C# ▷ Anatomi ▷ Krop

En funktions krop er en block der indeholder en sekvens af statements.

En funktion ...

- ▶ har adgang til de værdier som funktionens parametre er bundet til under kaldet.
- ▶ kan producere en værdi som den returnerer.
- ▶ kan ændre på variable udenfor funktionen.

```
int[] fill (int[] array, int value) {  
    for (int i=0 ; i<array.Length ; i++) {  
        array[i] = value;  
    }  
    return array;  
}
```


Funktioner i C# ▷ Anatomi ▷ Krop

En funktions krop er en block der indeholder en sekvens af statements.

En funktion ...

- ▶ har adgang til de værdier som funktionens parametre er bundet til under kaldet.
- ▶ kan producere en værdi som den returnerer.
- ▶ kan ændre på variable udenfor funktionen.

```
void fill (int[] array, int value) {  
    for (int i=0 ; i<array.Length ; i++) {  
        array[i] = value;  
    }  
}
```

Funktioner i C# ▷ Anatomi ▷ Return Statement

Udførelsen af en funktion kan afsluttes på et vilkårligt sted med et `return` statement.

Funktioner i C# ▷ Anatomi ▷ Return Statement

Udførelsen af en funktion kan afsluttes på et vilkårligt sted med et `return` statement.

Typeangivelsen `void` bruges til at indikere, at der *ikke skal* returneres en værdi.

Hvis en funktion har en retur type anderledes end `void` så

- ▶ skal alle mulige veje igennem funktionen ende i et `return` statement.
- ▶ skal alle `return` statements have et expression der evaluerer til en værdi der kan implicit castes til retur typen.

Funktioner i C# ▷ Anatomi ▷ Return Statement

Udførelsen af en funktion kan afsluttes på et vilkårligt sted med et `return` statement.

Typeangivelsen `void` bruges til at indikere, at der *ikke skal* returneres en værdi.

Hvis en funktion har en retur type anderledes end `void` så

- ▶ skal alle mulige veje igennem funktionen ende i et `return` statement.
- ▶ skal alle `return` statements have et expression der evaluerer til en værdi der kan implicit castes til retur typen.

Dette er den typiske måde at overføre en værdi fra den kaldte til den kaldende funktion.

Part 2: Funktionskald

Funktionskald ▷ Parameterbinding

Vi *kalder* en funktion ved at skrive navnet på den efterfulgt af et set parenteser.

- ▶ Tomme parenteser hvis funktionen ikke har nogen parameter.
- ▶ Parenteser med kommaseparerede udtryk hvis funktionen har parametre.
 - ▶ Rigtigt antal.
 - ▶ Riktig(e) type(r).

Funktionskald ▷ Parameterbinding

Vi *kalder* en funktion ved at skrive navnet på den efterfulgt af et set parenteser.

- ▶ Tomme parenteser hvis funktionen ikke har nogen parameter.
- ▶ Parenteser med kommaseparerede udtryk hvis funktionen har parametre.
 - ▶ Rigtigt antal.
 - ▶ Riktig(e) type(r). Hvor rigtige?

Funktionskald ▷ Parameterbinding

Vi *kalder* en funktion ved at skrive navnet på den efterfulgt af et set parenteser.

- ▶ Tomme parenteser hvis funktionen ikke har nogen parameter.
- ▶ Parenteser med kommaseparerede udtryk hvis funktionen har parametre.
 - ▶ Rigtigt antal.
 - ▶ Rigtig(e) type(r). Hvor rigtige?

```
int[] intArr = new int[12];  
int[] oneArr = fill(intArr, 1);
```

```
int[] fill (int[] array, int value) {  
    for (int i=0 ; i<array.Length ; i++) {  
        array[i] = value;  
    }  
  
    return array;  
}
```


Funktionskald ▷ Parameterbinding


Vi *kalder* en funktion ved at skrive navnet på den efterfulgt af et set parenteser.

- ▶ Tomme parenteser hvis funktionen ikke har nogen parameter.
- ▶ Parenteser med kommaseparerede udtryk hvis funktionen har parametre.
 - ▶ Rigtigt antal.
 - ▶ Richtig(e) type(r). Hvor rigtige?

```
int[] intArr = new int[12];
int[] oneArr = fill(intArr, 1);

int[] fill (int[] array, int value) {
    for (int i=0 ; i<array.Length ; i++) {
        array[i] = value;
    }

    return array;
}
```



Funktionskald ▷ Parameterbinding

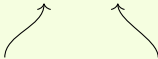
Vi *kalder* en funktion ved at skrive navnet på den efterfulgt af et set parenteser.

- ▶ Tomme parenteser hvis funktionen ikke har nogen parameter.
- ▶ Parenteser med kommaseparerede udtryk hvis funktionen har parametre.
 - ▶ Rigtigt antal.
 - ▶ Rigtig(e) type(r). Hvor rigtige?

```
int[] intArr = new int[12];
int[] oneArr = fill(intArr, 1);

int[] fill (int[] array, int value) {
    for (int i=0 ; i<array.Length ; i++) {
        array[i] = value;
    }

    return array;
}
```



Funktionskald ► Parameteroverførsel

Vi kan overføre værdier fra den kaldende funktion til den kaldte funktion.

- Kaldes for *value passing*.

Funktionskald ▷ Parameteroverførsel

Vi kan overføre værdier fra den kaldende funktion til den kaldte funktion.

- ▶ Kaldes for *value passing*.

Når en værdi overføres fra den kaldende funktion (på engelsk “the caller”) til den kaldte funktion (på engelsk “the callee”) gælder følgende regler:

- ▶ Ved variable af primitive typer kopieres værdien.
- ▶ Ved variable af komplekse typer kopieres værdien, **men i dette tilfælde er værdien den adresse i hukommelsen hvor den komplekse værdi er lagret.**

Funktionskald ▷ Parameteroverførsel

Vi kan overføre værdier fra den kaldende funktion til den kaldte funktion.

- ▶ Kaldes for *value passing*.

Når en værdi overføres fra den kaldende funktion (på engelsk “the caller”) til den kaldte funktion (på engelsk “the callee”) gælder følgende regler:

- ▶ Ved variable af primitive typer kopieres værdien.
- ▶ Ved variable af komplekse typer kopieres værdien, **men i dette tilfælde er værdien den adresse i hukommelsen hvor den komplekse værdi er lagret.**

```
int[] intArr = new int[12];
```

```
int i = -1;
```

```
fill(intArr, i);
```

↑
reference†

↑
værdi

Funktionskald ▷ Funktionskald som Expression

```
int max (int[] array) {  
    int max = -1;  
    for (int i=0 ; i<array.Length ; i++) {  
        if (array[i]>max) {  
            max = array[i];  
        }  
    }  
    return max;  
}
```

```
int[] months = {31,28,31,30,31,30,31,31,30,31,30,31};  
int longer = max(months) + 1;  
Console.WriteLine("No month is "+longer+" days long");
```

Funktionskald ▷ Funktionskald som Expression

```
int max (int[] array) {  
    int max = -1;  
    for (int i=0 ; i<array.Length ; i++) {  
        if (array[i]>max) {  
            max = array[i];  
        }  
    }  
    return max;  
}
```

```
int[] months = {31,28,31,30,31,30,31,31,30,31,30,31};  
int longer = max(months) + 1;  
Console.WriteLine("No month is "+longer+" days long");
```

Output ved kørsel: No month is 32 days long

Funktionskald ► Funktionskald som Statement

```
void prettyPrintArray (int[] array) {  
    Console.Write("[");  
    for (int i=0 ; i<array.Length ; i++) {  
        Console.Write((i==0 ? "" : ",")+array[i]);  
    }  
    Console.WriteLine("]");  
}
```

```
int[] months = {31,28,31,30,31,30,31,31,30,31,30,31};  
prettyPrintArray(months);
```


Funktionskald ▷ Funktionskald som Statement

```
void prettyPrintArray (int[] array) {  
    Console.Write("[");  
    for (int i=0 ; i<array.Length ; i++) {  
        Console.Write((i==0 ? "" : ",")+array[i]);  
    }  
    Console.WriteLine("]");  
}
```

```
int[] months = {31,28,31,30,31,30,31,31,30,31,30,31};  
prettyPrintArray(months);
```

Output ved kørsel: [31,28,31,30,31,30,31,31,30,31,30,31]

Funktionskald ▷ Funktionskald med Sideeffekt

```
int checkedIn = 0;

void checkin () {
    checkedIn++;
}

for (int i=0 ; i<100 ; i++) {
    checkin();
}
Console.WriteLine(checkedIn+" people has checked in");
```

Funktionskald ► Funktionskald med Sideeffekt

```
int checkedIn = 0;

void checkin () {
    checkedIn++;
}

for (int i=0 ; i<100 ; i++) {
    checkin();
}
Console.WriteLine(checkedIn+" people has checked in");
```

Output ved kørsel: 100 people has checked in

Part 3:

Modularisering med Funktioner

Modularisering med Funktioner

Funktioner kan gøre kode genbrugelig, uden de restriktioner på flow som kendes fra branches og loops.

Modularisering med Funktioner

Funktioner kan gøre kode genbrugelig, uden de restriktioner på flow som kendes fra branches og loops.

Kombinationen af signatur og returværdi repræsenterer en kontrakt mellem de som skriver funktionen og de som bruger den.

Modularisering med Funktioner

Funktioner kan gøre kode genbrugelig, uden de restriktioner på flow som kendes fra branches og loops.

Kombinationen af signatur og returværdi repræsenterer en kontrakt mellem de som skriver funktionen og de som bruger den.

Denne adskillelse isolerer de to sider fra hinanden (så længe ingen sideeffekter modificerer delt tilstand).

Modularisering med Funktioner

Funktioner kan gøre kode genbrugelig, uden de restriktioner på flow som kendes fra branches og loops.

Kombinationen af signatur og returværdi repræsenterer en kontrakt mellem de som skriver funktionen og de som bruger den.

Denne adskillelse isolerer de to sider fra hinanden (så længe ingen sideeffekter modificerer delt tilstand).

Nogle klasser af fejl begrænses herved til funktionen.

Part 4:

Funktioner der Kalder Funktioner

Modularisering med Funktioner ▷ Funktion til Funktion Kald

```
int[] data = {1,3,5,8};  
int maxD = maxDiff(data);  
Console.WriteLine("The maximum difference is "+maxD);
```

```
int maxDiff (int[] array) {  
    int[] diffs = new int[array.Length-1];  
    for (int i=0 ; i<diffs.Length ; i++) {  
        diffs[i] = array[i+1] - array[i];  
    }  
    return max(diffs);  
}
```

```
int max (int[] array) {  
    int max = -1;  
    for (int i=0 ; i<array.Length ; i++) {  
        if (array[i]>max) {  
            max = array[i];  
        }  
    }  
    return max;  
}
```

Part 5: Rekursion

Rekursion ▷ Definition

En funktion der virker ved at kalde sig selv én eller flere gange kaldes en rekursiv funktion.

Rekursion ▷ Definition

En funktion der virker ved at kalde sig selv én eller flere gange kaldes en rekursiv funktion.

De fleste rekursive funktioner er struktureret omkring en branch, der adskiller ét eller flere basistilfælde fra hvad der kaldes et rekursionstrin.

Rekursion ▷ Definition

En funktion der virker ved at kalde sig selv én eller flere gange kaldes en rekursiv funktion.

De fleste rekursive funktioner er struktureret omkring en branch, der adskiller ét eller flere basistilfælde fra hvad der kaldes et rekursionstrin.

Basistilfælde er ofte trivielle.

Rekursion ▷ Definition

En funktion der virker ved at kalde sig selv én eller flere gange kaldes en rekursiv funktion.

De fleste rekursive funktioner er struktureret omkring en branch, der adskiller ét eller flere basistilfælde fra hvad der kaldes et rekursionstrin.

Basistilfælde er ofte trivielle.

I det rekursive trin løser funktionen en delmængde af problemet ved at kalde sig selv med denne delmængde som parameter, og så integrerer den resultatet af dette med løsningen til resten af problemet for at få den fulde løsning.



Rekursion ▷ Fibonacci Sekvensen ▷ Definition

Fibonacci sekvensen er defineret som:

$$fib(n) = \begin{cases} 0, & \text{for } n = 0 \\ n, & \text{for } n = 1 \\ fib(n-1) + fib(n-2), & \text{for } n > 1 \end{cases}$$

Rekursion ▷ Fibonacci Sekvensen ▷ Definition

Fibonacci sekvensen er defineret som:

$$fib(n) = \begin{cases} 0, & \text{for } n = 0 \\ n, & \text{for } n = 1 \\ fib(n-1) + fib(n-2), & \text{for } n > 1 \end{cases}$$

Dette giver: 0 1 1 2 3 5 8 13 21 34 55 ...

Rekursion ▷ Fibonacci Sekvensen ▷ Definition

Fibonacci sekvensen er defineret som:

$$fib(n) = \begin{cases} 0, & \text{for } n = 0 \\ n, & \text{for } n = 1 \\ fib(n-1) + fib(n-2), & \text{for } n > 1 \end{cases}$$

Dette giver: 0 1 1 2 3 5 8 13 21 34 55 ...

Hvordan kan vi implementere en funktion der returnerer den n'te Fibonacci tal?

Rekursion ▷ Fibonacci Sekvensen ▷ Definition

Fibonacci sekvensen er defineret som:

$$fib(n) = \begin{cases} 0, & \text{for } n = 0 \\ n, & \text{for } n = 1 \\ fib(n-1) + fib(n-2), & \text{for } n > 1 \end{cases}$$

Dette giver: 0 1 1 2 3 5 8 13 21 34 55 ...

Hvordan kan vi implementere en funktion der returnerer den n'te Fibonacci tal?

Grænsefladen til denne funktion må være:

```
int fib (int n)
```

Rekursion ▷ Fibonacci Sekvensen ▷ Løsning

```
int fib (int n) {  
    if (n==0) {  
        return 0;  
    } else if (n==1) {  
        return 1;  
    } else {  
        return fib(n-1) + fib(n-2);  
    }  
}  
  
for (int n=0 ; n<10 ; n++) {  
    Console.WriteLine(n+": "+fib(n));  
}
```

Rekursion ▷ Fibonacci Sekvensen ▷ Løsning

```
int fib (int n) {  
    if (n==0) {  
        return 0;  
    } else if (n==1) {  
        return 1;  
    } else {  
        return fib(n-1) + fib(n-2);  
    }  
}  
  
for (int n=0 ; n<10 ; n++) {  
    Console.WriteLine(n+": "+fib(n));  
}
```

0: 0
1: 1
2: 1
3: 2
4: 3
5: 5
6: 8
7: 13
8: 21
9: 34

Rekursion ▷ Fibonacci Sekvensen ▷ Løsning

```
int fib (int n) {  
    if (n==0) {  
        return 0;  
    } else if (n==1) {  
        return 1;  
    } else {  
        return fib(n-1) + fib(n-2);  
    }  
}  
  
for (int n=0 ; n<10 ; n++) {  
    Console.WriteLine(n+": "+fib(n));  
}
```

0: 0
1: 1
2: 1
3: 2
4: 3
5: 5
6: 8
7: 13
8: 21
9: 34

Hvorfor er dette en god løsning?

Rekursion ▷ Fibonacci Sekvensen ▷ Løsning

```
int fib (int n) {  
    if (n==0) {  
        return 0;  
    } else if (n==1) {  
        return 1;  
    } else {  
        return fib(n-1) + fib(n-2);  
    }  
}  
  
for (int n=0 ; n<10 ; n++) {  
    Console.WriteLine(n+": "+fib(n));  
}
```

0: 0
1: 1
2: 1
3: 2
4: 3
5: 5
6: 8
7: 13
8: 21
9: 34

Hvorfor er dette en dårlig løsning?

Questions?

