5
Basal Objekt-orienteret Programmering II.

Historik og sprogoversigt.

Programbeskrivelse kontra programudførelse.

Referencer og værdier.

Skabelse af objekter i Eiffel.

Features og deres klassificering i Eiffel.

Routiner i Eiffel.

Current object.

Konstanter af simple typer og klassetyper.

PS1 -- Basal objekt-orienteret programmering II

Noter

I dette kapitel (og videre i noterne) vil vi antage følgende:

> Når vi introducerer en klasse vil vi med mindre andet fremgår eksplicit indskrænke os til at studere situationen, hvor objekter af klassen er dynamisk allokerede. Med andre ord arbejder vi som hovedregel altid med referencer til objekter, og vi ignorerer typisk at man rent faktisk i Eiffel godt kan have klasser, hvis objekter er statisk allokerede.

I Eiffel-teknisk forstand ser vi altså ofte bort fra klasser, der er expanderede, og variabelerklæringer på formen

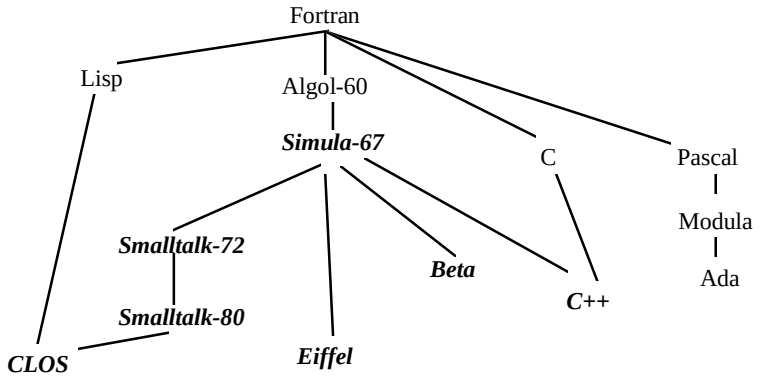
> var: **expanded** C

hvor C er en klasse.

Det vil komplicere disse noter for meget at fortælle hele sandheden om Eiffel, og I mange praktiske sammenhænge er det ikke nødvendigt at kende hele sandheden om sproget.

Hvis du ikke forstår ovenstående bemærkning fuldt ud gør det ikke så meget. Du skal blot forstå, at dette kapitel (og generelt disse noter) ikke er en fuldstændig beskrivelse af mulighederne i Eiffel. Ofte vil vi henvise til “Eiffel the language” for detaljer.

Den historiske udvikling af objekt-orienterede programmeringssprog.



PS1 -- Basal objekt-orienteret programmering II

Noter

På denne slide er det illustreret, hvordan nogle interessante objekt-orienterede sprog er i familie med hinanden, og hvordan de er i familie med ikke-objekt-orienterede sprog. Kun sprog skrevet med ***fede, kursiverede typer*** kan betegnes som objekt-orienterede.

Den vertikale dimension afspejler en tidsakse med de nyeste sprog forneden og de ældste øverst. Det kan være vanskeligt nøjagtigt at tidsfæste nogle sprog, så tag dette med et gran af salt.

Bemærk, at alle objekt-orienterede sprog har rod i Simula-67, der som navnet antyder er udviklet midt i 60'erne. Simula er udviklet i Norge af bl.a. Kristen Nygaard. Kristen Nygaard blev i sommeren 91 æresdoktor på AUC.

De familiemæssige sammenhænge mellem sprog danner et generaliserings/specialiseringshierarki, som i et vist omfang er analogt til de hierarkier, vi har set på i forelæsningen om abstrakte datatyper. Ovenstående hierarki er naturligvis groft simplificeret, og ukomplet i forhold til virkelighedens.

Beskrivelse kontra udførelse af et objekt-orienteret program.

Statisk programbeskrivelse som det fremgår af programteksten:

- *Klasser* , med variable (attributter) og operationer (routiner)

Dynamisk programudførelse som det forekommer i maskinen, når man kører programmet:

- *Objekter* med felter der svarer til variable.
  Felter:
  - Værdier af simple typer.
  - *Refencer* til objekter.

Objekter og referencer udgør et "netværk", som udvikler sig i takt med at programudførelsen skrider frem.

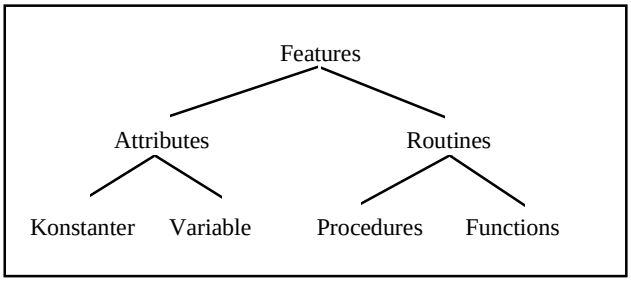
Noter

Det er meget vigtig at holde programbeskrivelsen (klasserne) adskilt fra programudførelsen (objekter, med indbyrdes referencer).

I nogle sprog er klasser også objekter, som kan manipuleres af et sæt af operationer. Smalltalk og CLOS er eksempler på sådanne sprog. Når klasser også er objekter, kan man spørge sig selv, hvilke klasser klasse-objekterne er instanser af? Normalt kalder man klasser, hvis instanser repræsenterer klasser, for *metaklasser*. Operationerne i disse metaklasser tillader, at man undersøger og påvirker aspekter af programmet selv. Denne "selv-optagethed" kaldes med et fint ord for *refleksion.* På dette kursus vil vi imidlertid ikke komme ind på disse ting, men metaklasser, og hvad deraf følger, udgør en ganske spændende verden; specielt hvis man er interesseret i at lave værktøj og programmeringsomgivelser.

Features i Eiffel.

"Features" er en fællesbetegnelse for variable/konstanter og procedure/funktioner i Eiffel.

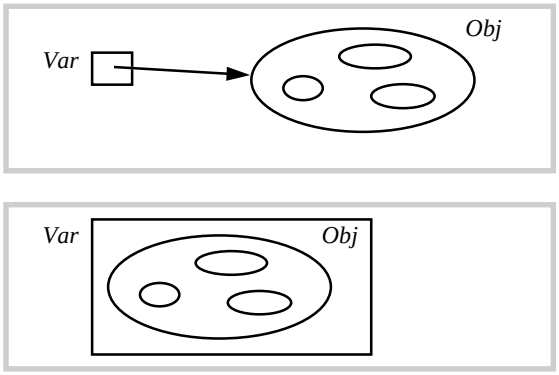


Noter

Denne slide viser den logiske klassificering af features i Eiffel. Lidt senere i kapitlet vender vi tilbage til denne klasseficering, set fra et andet perspektiv.

Referencer og værdier.

Lad *Var* være en variabel, som er initialiseret til et objekt *Obj*:



Var indeholder en reference til objektet *Obj*

Var indeholder en værdi, som *er* objektet *Obj*

Noter

I det øverste tilfælde vil objektet være dynamisk instantieret, og i det nederste tilfælde statisk instantieret.

Som omtalt i forrige forelæsning, er de simple prædefinerede typer i Eiffel altid statisk instantierede. Vi vil som regel være interesseret i dynamisk instantiering af klassetyper. Hvis man imidlertid erklærer en attribut 'a' som

a: **expanded** C

bliver objektet statisk instantieret. I Eiffel er der også mulighed for at definere selve klassen som en expanded class; denne mulighed er attraktiv, hvis ethvert objekt af den pågældende klasse skal instantieres statisk.

En variabel, som er tiltænkt at indeholde en reference, har værdien **void,** inden den dynamiske instantiering finder sted.

I Eiffel kan man afgøre om en reference i en variabel *var* er void ved følgende udtryk:

var = Void

Skabelse af et objekt i Eiffel: Instantiering.

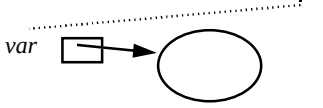
Et objekt kan skabes ud fra en klasse gennem instantiering.
Instantiering er i Eiffel tæt koblet med efterfølgende initialisering af det nye objekts felter.

Instantiering af klassen C**:**

```
var: C; -- var er void
...
do

  !!var.Creation-procedure;

  ...
end
```



Instantierings- og initialiseringstrin i Eiffel:

1. Der allokeres lager til objektet.
2. Alle felter initialiseres til defaultværdier.
3. Den angivne creation procedure bliver kaldt, for at videre-initialisere objektet.

Default feltværdier:

BOOLEAN	false
CHARACTER	null tegn
INTEGER	0
REAL	0.0
DOUBLE	0.0
Class Type	void

PS1 -- Basal objekt-orienteret programmering II

Noter

Et objekt kan skabes på to forskellige måder: (1) Ved at skabe objektet med klassen som forskrift, eller (2) ved at skabe objektet med et andet og eksisterende objekt som forskrift. På denne og næste slide ser vi på mulighed 1. Derefter studerer vi mulighed 2.

En variabel der er erklæret af en klassetype, indeholder værdien **void** lige efter dets erklæring.

Den faste syntaks for instantiering er anvendelse af to udråbstegn:

```
!!target.creation_procedure(parametre).
```

Faktisk indeholder mekanismen endnu en mulighed, nemlig en eksplicit angiven creation type:

```
! creation-type!target.creation_procedure(parametre).
```

Semantikken er at der skabes et objekt af creation-type, denne initialiseres gennem trin 2 og 3 ovenfor, og sluttelig etableres en reference fra target til det ny-skabte objekt. Vi får dog først brug for denne udvidede form når vi begynder at interessere os for nedarvning.

Referencen mellem en variabel V og et objekt kan brydes ved assignmentet:

```
var := void
```

Dette medfører ikke nødvendigvis, at objektet bliver nedlagt, men kun at referencen mellem variablen og objektet bliver brudt. Som omtalt i det første kapitel i disse noter har et kørende Eiffel system en såkaldt *garbage collector*, der har til ansvar at deallokere objekter, som ikke har nogen chance for at påvirke det kørende program. (Dette vil i praksis sige objekter, der ikke kan nåes fra et sæt af globale variable og andre såkaldte "rødder").

Creation procedurer i Eiffel.

Formålet med en creation procedure er primært at initialisere nye objekter lige efter overstået instantiering.

Sekundært kan creation procedurer anvendes lige som alle andre procedurer i en klasse.

```
a, b: C              -- her erklæres variable
t1, t2: T

...

!!a.create;          -- her instantieres og
!!b.make(t1, t2);    -- initialiseres

...

a.make(t1,t2);       -- her illustreres den
b.create             -- den sekundære anvendelse
                     -- af creation procedurer
```

```
class C
  ...
creation
  make, create, ...
feature


  create is
    do ... end

  make(x, y: T) is
    do ... end;
  ...

end -- class C
```


Noter

På denne slide illustreres anvendelse af creation procedurer i forhold til omridset af en klassedefinition.

Det ses, at man kan have mere end én creation procedure. Dette tillader, at man kan programmere mere end én form for initialisering af en klasse, hvilket en gang imellem er nyttigt. Typisk (men ikke nødvendigvis) vil de forskellige creation procedurer adskille sig ved at have forskellige parametre, herunder forskellig antal parametre.

Hvis man ikke angiver creation procedurer (rettere, hvis man slet ikke har afsnittet der starter med creation i klassen), kan man oprette en instans af en klasse på følgende måde:

```
!!x
```

Hermed udføres kun de to første trin i instantiering og initialisering, som vist på forrige slide.

Nederst på sliden (til venstre) vises hvordan creation procedurerne kan anvendes som ganske almindelige procedurer (operationer) i objekt-orienteret programmering. I praktiske sammenhænge kan dette anvendes hvis man vil re-initialisere et objekt (resetting). For at illustrere at operationer på objekter er uafhængige af måden, hvorpå objekterne er skabt, har jeg her valgt at re-initialisere objektet der referes af variablen a (skabt af create) med make, og tilsvarende med objektet refereret af b.

Det er muligt for en klasse af have mere end én creation del, og det er muligt at angive en begrænsning på en creation del:

```
class C
  ...
creation
  make, create, ...
creation{D, E}
  create_after_troubles
feature ...
```

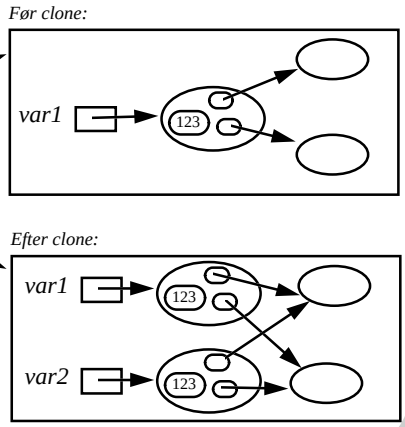
I Ovenstående kan create_after_troubels kun anvendes i klienter D eller E.

Skabelse af et objekt i Eiffel: Cloning.

Funktionen Clone returnerer et nyt objekt som er en kopi af parameteren. Clone skaber kun ét ny objekt, dvs. det kopierer kun “et niveau”.

Cloning:

```
var1, var2: C; --var er void
...
do
  var2 := clone(var1);
end
```



Variationer af cloning:

- **Deep-clone**: Dublerer et helt objektnetværk ved rekursivt at skabe en isomorf kopi.
- **Copy**: x.copy(y) kopierer indholdet af et objekt over i et *eksisterende objekt* x.

PS1 -- Basal objekt-orienteret programmering II

Noter

Ved kopiering med operationen clone overtager det nyoprettede objekt originalens værdier, men der kopieres kun dette ene objekt. Evt. referencer til andre objekter i originalen kopieres. De refererede objekter kopieres ikke (rekursivt). Dette illustreres ved eksemplet til højre ovenfor.

Når vi siger, at “der kun oprettes ét objekt” ved cloning tæller vi ikke statisk instantierede objekter med. Tal, tegn og andre statisk instantierede objekter kopieres altid af clone. Referencer kopieres, hvilket vil sige at der i ovenstående figurer oprettes en “ny pil”, men ikke et nyt objekt i enden af pilen.

Clone foretager både instantiering og initialisering. Der findes en operation, copy, som blot initialiserer indholdet af et *eksisterende* objekt ud fra et andet objekt. I vores implementation af Eiffel er clone rent faktisk implementeret ved en instantiering af en klasse efterfulgt af en copy operation.

Alle operationer som er omtalt på denne slide (og en hel del mere) er definerede i klassen ANY, hvorfra det viser sig at alle klasse arver.

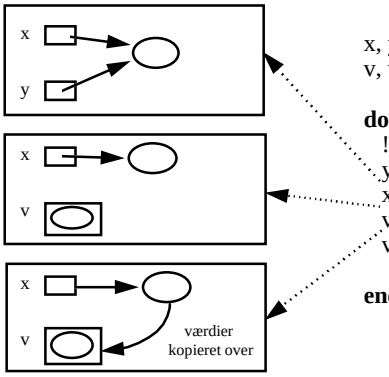
For helt præcises detaljer henvises til kapitel 19 i “Eiffel the Language”, som hedder “Duplicating and comparing objects”.

Reference- og værdi-semantik af assignment.

variabel := udtryk

Et assignment har **reference semantik** når både variabel og udtryk er af (uekspanderede) klassetyper.

I de øvrige tilfælde har et assignment **værdi semantik** (enten clone eller copy).




```
x, y: C;
v, w: expanded C;

do
  !!x.Create;
  y := x;  -- reference semantik
  x := v;  -- v clones og ref. assignes til x
  v := x;  -- ækvivalent med v.copy(x)
  v := w  -- ækvivalent med v.copy(w)
   ...
end
```


Noter

Vi taler om reference semantik når det udelukkende er referencer der undersøges eller manipuleres. Og vi taler om værdi semantik når vi på en eller anden måde undersøger eller manipulerer selve objekterne.

Figurerne til venstre viser situationen efter hvert af de første tre assignments i programfragmentet til højre.

I tilfælde af assignment har vi kun reference semantik hvis både variablen (venstresiden af assignmentet) og udtrykket (højresiden) er af uekspanderede klassetyper. Denne situation er helt typisk for “normale” klasser i Eiffel.

Vi har værdi semantik når både venstre- og højresiden af assignmentet er af ekspanderede typer. Flg. er et meget typisk eksempel (husk på, at integer er defineret som en expanded class, jf. kapitel 32 i “Eiffel the Language”)

```
n, m: integer;

n := m
```

Sidstnævnte assignment er altså ækvivalent med

```
n.copy(m)
```

hvilket afspejler vores forventning: Efter assignmentet er værdien af n en kopi af m.

Ovenfor behandler vi for fuldstændigheds skyld endvidere de to blandede tilfælde, som begge er meget mindre typiske.

Reference- og værdi-semantik af ligheds-operationer.

Når det giver mening (når to referencer sammenlignes) har "=" referencesemantik.

I øvrige tilfælde har "=" værdisemantik, realiseret af equal.

x = y	equal(x, y)	deep_equal(x, y)

- **Reference equality**.
  Når både x og y udtrykkene har referencer som værdier, returner x = y hvorvidt begge referencer peger på det samme objekt.
- **Shallow equality**.
  Operationen equal sammenligner om de to objekter, som x og y refererer til, strukturelt er ens. Kun ét objekt-niveau sammenlignes.
- **Deep equality**.
  Operationen deep_equal sammenligner om de to objekt-netværk, som x og y refererer til, strukturelt og rekursivt er ens. Tilbundsgående strukturel lighed.

x = y ==> equal(x,y) => deep_equal(x,y)

PS1 -- Basal objekt-orienteret programmering II

Noter

Hvis værdien af x er en reference og værdien af y er et objekt (eller hvis både x og y som værdi har objekter), giver det ikke mening at lade = sammenligne referencer. Derfor benyttes semantikken af equal i dette tilfælde.

Hvorfor anvendes formen

```
equal(x,y)
```

i stedet for dot-notationen

```
x.equal(y)   eller  y.equal(x)  ?
```

Førstnævnte notation er symmetrisk i behandlingen af de to parametre, hvorimod sidstnævnte kan siges at være mere på linie med objekt-orienteret operations-anvendelse (hvor jo en operation anvendes på netop ét objekt (mål-objektet, eller på engelsk target). Den mere kontante begrundelse for at anvende formen equal(x,y) fremfor dot-notation er, at x.equal(y) ikke kan bruges hvis x er void. Generelt kræves at "target" i

```
target.op(parametre)
```

er ikke-void. Med andre ord, kan man ikke sammenligne to void referencer med x.equal(y), men det kan man med equal(x,y).

I vores implementation af Eiffel systemet findes en operation is_equal, som netop kaldes på formen:

```
x.is_equal(y).
```

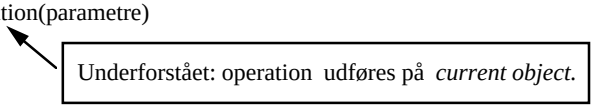
Internt implementeres equal ved brug af is_equal. I forbindelse med nedarvning og redefinition viser det sig at være en stor fordel at håndtere lighedsoperationen ligesom alle andre operationer.

Det nuværende objekt.

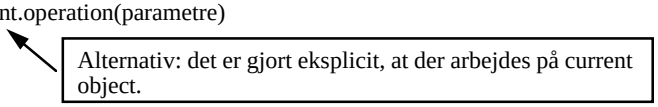
Et objekt kan både manipuleres "udefra" og "indefra" relativt til en klasse.

Manipulation udefra:
objekt.operation(parametre)

Manipulation indefra:
operation(parametre)



current.operation(parametre)



Der kan være behov for at referere til det nuværende objekt som helhed, imodsætning til at referere til dets bestanddele.

- Returnere det nuværende objekt fra en funktion.
- Overføre det nuværende objekt som parameter til en operation.

Noter

De fleste objekt-orienterede sprog har en notation, som tillader at man inde fra et objekt kan referere til objektet som helhed. I Eiffel benytter man **current**, i C++ **this**, i Smalltalk **self**.

I Eiffel er **current** et udtryk. Det medfører, at man ikke kan ændre på værdien af denne "variabel".

Nogle sprog insisterer på, at man indefra et objekt skriver current.feature for at referere til feature. Andre sprog, bl.a. Eiffel og C++, tillader en kortere (og mindre objekt-orienteret) notation, som på overfladen ser ud som procedurekald.

Det forekommer ofte, at man har behov for at returnere hele det nuværende objekt fra en operation. Det er måske mindre typisk, at man overfører det nuværende objekt som parameter til en operation. Dette skyldes, at et operationskald altid foregår på et objekt, og hvis ikke andet er nævnt, sker operationskaldet på det nuværende objekt. Derfor bliver det nuværende objekt en implicit parameter til alle operationer, hvorpå operationen kaldes.

Tidligere i dette kapitel så vi på cloning og kopieringsoperationer, som rent faktisk blot er operationer i en generel klasse ANY, som alle klasser arver fra. Når vi i et program skriver

```
x := clone(y)
```

mener vi altså

```
x := current.clone(y).
```

Ville det ikke have været “mere objekt-orienteret” i stedet for at skrive

```
x := y.clone;
```

altså at anvende det eksisterende objekt (der ønskes kopieret) som det objekt (target) hvorpå vi udfører operationen? Svaret er “jo”, men det samme forhold som ved equal gør sig gældende her.

Information hiding i Eiffel.

I Eiffel defineres interfacet til klienter ved en kategorisering af features i en klasse.

Muligheder:

- Klassens egenskaber er synlige fra alle klasser.
- Klassens egenskaber er kun synlige i nærmere definerede klasser.
  - Klassens egenskaber er kun synlige i sig selv.
- Klassens egenskaber er private.

```
class C
creation ...
feature
  -- features her er tilgængelige
  -- for alle klienter af C
feature{A, B}
  -- features her er kun tilgængelige
  -- for A og B
feature{C}
  -- features her er kun tilgængelige
  -- i C (dot-notation)
feature {NONE}
  -- features her er private til C
end -- class C
```


Noter

Denne slide konkretiserer "Information hiding" for Eiffel. Der mindes om, at vi i forrige forelæsning studerede information hinding mere generelt.

Hvad er forskellen på

(1) feature{C}

og

(2) feature{NONE}

i det ovenstående?

Lad os for at besvare dette spørgsmål studere en routine kaldet op i C (vi vil også antage, at C-op er en egenskab (feature) i klassen C)

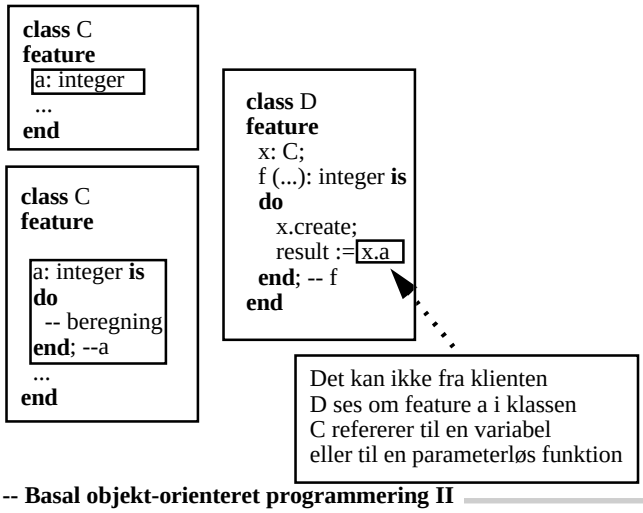
```
--operation i klassen C
op(...) is
local another: C
do
  C-op;  -- lovligt i både tilfælde  (1) og (2)
         -- C-op kaldes på det nuværende objekt.
  another.C-op  -- kun lovligt i tilfælde (1)
end
```

Vi ser, at eksport til sig selv (tilfælde 1) er nødvendig når en operation i en klasse *via dotnotation* påkalder sig en af sine egne operationer.

En bemærkning om de features, der er listet i creation delen, og eksportstatus af features, som angivet af deres kategorisering under features. Der er ingen kobling mellem creationstatus og eksportstatus. En feature kan udnævnes til creator-procedure under creation, og helt uafhængig af dette kan denne være en del af klient-interfacet, eller holdes skjult for klienter.

Det uniforme tilgangs-princip.

Det er transparent for klienter, om en feature er lagret eller beregnet.



Noter

På denne slide er der vist to udgaver af klassen C.

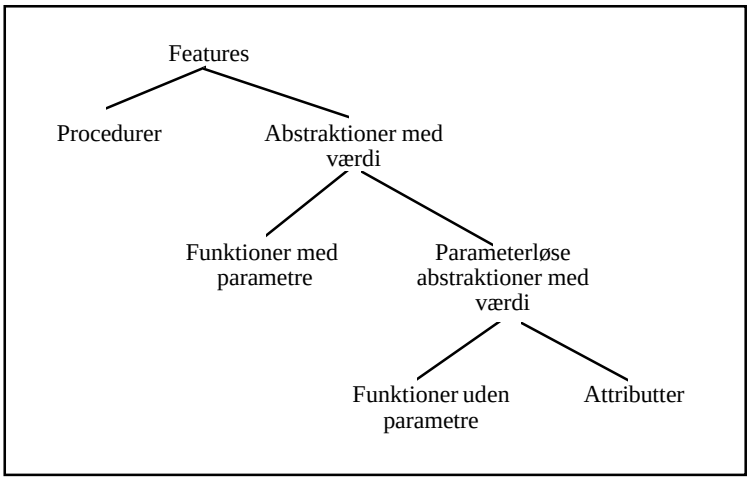
I den øverste er feature a en variabel. a er altså *lagret.*

I den nederste er feature a en funktion. a er altså beregnet, og der optages ikke lager i instanser af C til lagring af feature a.

Det illustreres, at i klassen D, som er en klient af klassen C, kan man ikke se forskel på de to situationer.

Som en væsentlig konsekvens af ovenstående kan man i klassen C ændre strategi fra lagring til beregning (eller omvendt) uden at klienterne af C skal ændres af denne årsag. Dette er et aspekt af *repræsentationsuafhængighed.*

Features set fra klient klasser (2).



Noter

I forhold til den tidligere viste klassificering af features i Eiffel viser denne slide features set fra klientklassens synsvinkel. Pointen er, at funktioner uden parametre og attributter ikke kan skelnes fra et klient-synspunkt. Funktioner udtrykker en feature, hvis værdi *beregnes* , når funktionen kaldes. Attributter udtrykker features, hvis værdi *lagres* i instanser af klassen.

Routiner i Eiffel.

```
proc(x: T) is
 local
   ...
 do
   ...
 end
```

```
func(x: T): S is
 local
  ....
 do
   ...
  result := ...
 end
```

Parameter-mekanisme:

Objekter, som er instanser af (uekspanderede) klasser overføres ved deres reference.

Værdier af simple typer overføres som værdi parametre (call-by-value).

Ændring af formelle parameter bindinger:

Værdien bundet til en formel-parameter er "read only".

Lokale routiner:

Routiner i Eiffel kan *ikke* have indlejrede (lokale) routiner.

Noter

I ovenstående eksempler er der vist hhv. en procedure med navn proc og en funktion med navn func.

Ved kaldet proc(y) udvirker parameteroverførslen den samme effekt som assignmentet x:= y, set fra proc's indre synsvinkel.

Man kan læse om parameter-mekanismen i Eiffel i 8.4 i OOSC. I "Eiffel the language" sammenlignes Eiffel parameteroverførsel med mere konventionel parameteroverførsel i afsnit 20.7.

At det formelle parameternavn x er read-only betyder, at hverken et assignment til x eller !!x.create er lovlig. Vær dog opmærksom på, at man uden problemer kan ændre på felterne i det objekt x peger på (naturligvis gennem de operationer, der er tilgængelige på dette objekt).

Navne på lokale variable i en Eiffel routine skal være forskellige fra feature navnene i den omkringliggende klasse. Dette er et ukonventionelt designvalg i Eiffel. Normalt tillader man i programmeringssprog navnesammenfald mellem det lokale niveau og det (mere) globale niveau med den mening, at lokale navne overskygger (mere) globale navn.

Det er interessant at konstatere, at routiner ikke kan indlejres statisk i hinanden i Eiffel program-tekst. Dette kan opfattes som et tilbageskridt i forhold til sprog som Pascal og Modula, hvor dette er et vigtigt sprog-element. Hvis man overvejer, hvorfor Pascal og Modula understøtter indlejrede procedurer, hænger dette givetvis sammen med top-down udviklingsteknikken (som beskrevet tidligere) og trinvis forfining. Når en procedure forfines, er det nærliggende af forfiningerne defineres som lokale procedurer eller funktioner. Vi har argumenteret for, at objekt-orienteret programmering er mere bottom-up agtig i natur, og centreret omkring data frem for funktion. Så ét incitament for lokale procedurer er faldet bort. (Dette er ikke det samme som at sige, at der ikke i Eiffel kan opstå situationer, hvor en routine er så omfattende, at man burde definere en mængde af lokale hjælpe-procedure. Men det kan man altså ikke).

Iøvrigt kan man heller ikke indlejre klasser i hinanden. I det hele taget er såkaldt blok-struktur (indlejrede navne-rum) dårligt understøttet i Eiffel.

Scoperegler i Eiffel.

Scope af ...

Lokal variabel:	routinen hvori variablen er defineret.
Formel parameter:	routinen hvori parameteren forekommer.
Attribut og routine:	klasse, hvori disse er defineret.
Klasse:	klasser, som indbefatter klassen i en klynge i universet.

Noter

Scopereglerne i Eiffel hvad angår lokale variable og parametre er simple, idet routiner ikke kan indlejres i hinanden.

Der vil dog kunne forekomme navnesammenfald mellem lokale variable og attributter i den omkringliggende klasse. Eiffel forbyder dette! Eiffel er således meget konservativ med hensyn til traditionel blokstruktur og "huller i scope".

Navnesammenfald mellem formelle parametre og lokale variable er, som i de fleste andre sprog, forbudt.

Attributter og routiner har effekt i den klasse, hvori disse er defineret. Gennem eksport kan man også sige at attributter og routiner også har effekt i klient klasser; og gennem nedarvning har attributter og routiner effekt i subklasser. Vi vil dog ikke lade eksport til klienter og nedarvning til subklasser indfluere på vores opfattelse af scopereglerne.

Scopet af en klasse er ikke bestemt af sproglige faktorer men af det univers, der defineres af sekvensen af klynger i en Eiffel system beskrivelse (ACE).

I Eiffel skal navne ikke nødvendigvis defineres før brug. Dette er endnu en forskellighed i forhold til konventionerne i Pascal.

Konstanter af simple typer i Eiffel.

En konstant er et symbolsk navn på en værdi.
Værdien af en konstant kan ikke ændres under programudførelsen.

Simple typer T**:**

INTEGER, BOOLEAN, REAL, CHARACTER

Const: T is *konstantudtryk*

Eksempler:

Manifeste konstanter

```
Big_int: integer is  99999
Ok: boolean is true
```

Konstanter optager *ikke* lagerplads på programmets udførelsestidspunkt på samme måde som variable.

Noter

Brugen af konstanter som symbolske navne for faste værdier i programmer kan kraftigt anbefales. Hvis f.eks. værdien 1001 bruges mange gange i et program er det fornuftigt at have et symbolsk navn på dette tal, f.eks. *antal_nætter*. Dels øger det læsbarheden af programmet, dels bliver det muligt statisk at ændre værdien, ved at editere konstant definitionen.

En konstant siges at være *manifest* i programmet, hvis vi har en synlig, direkte notation for konstantens værdi, som vi kan skrive i vores program.

Konstantudtryk behandles nærmere i kapitlet "Kommandoer, Udtryk og IO i Eiffel".

Vi taler ovenfor (og på dansk) om “simple typer”. I den engelske Eiffel literatur tales om “basic types”. De to betegnelser er synonyme.

Konstanter af klassetyper i Eiffel.

Problem: Der findes ikke nogen notation i sproget, som gør det muligt at "nævne" et objekt.

Der findes ikke manifeste konstanter af klassetyper i Eiffel. (Undtagelse: STRING og ARRAY).

Løsning:

Lad C være en klassetype.

```
C_konstant: C is
once
  result.create(...)
end
```

Ved første udførelse af en "once funktion" F udføres kroppen af F, og den returnerede værdi V huskes.

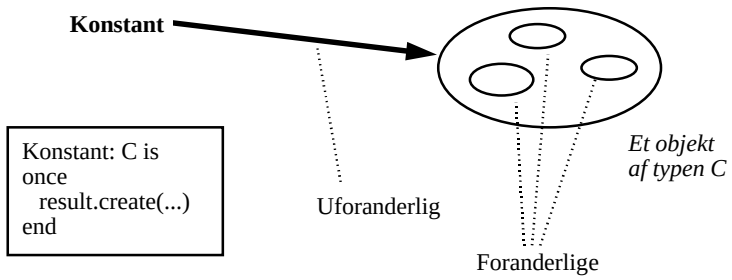
Ved efterfølgende udførelser af F returneres værdien af V umiddelbart, uden at kroppen af F udføres.

Evt. parametre til F har kun effekt ved det første kald af F.

Noter

Det er vigtigt at få fat i, at kroppen af en once-funktion F højst kan udføres én gang i hele programmets levetid. Dette er uanset, om der findes mange objekter, hvori F er en operation. Eiffel-systemet husker på en eller anden måde returværdien fra funktionens første kald.

Delning af objekter af klassetyper i Eiffel.



Alle referencer til 'konstant' (alle kald af "once funktionen") vil referere til det selvsamme objekt, som derfor bliver delt.

Objektet er derimod mutérbart (ikke-konstant).

Noter

Når man er i stand til at definere konstanter af såvel simple typer som klasse-typer vil man også være interesseret i at stille disse tilrådighed for udvalgte dele af et program, evt. hele programmet. Nedarvning kan anvendes til dette. I kapitlet "Nedarvning III" ser vi nærmere på dette. Se også evt. afsnit 13.2 i OOSC.