7
Kommandoer, Udtryk og Input Output i Eiffel.

Typesammenlignelighed

Kommandoer
- Den tomme kommando.
- Assignment.
- Procedurekald.
- Den betingede kommando.
- Den iterative kommando.

Udtryk
- Operatorudtryk.
- Konstanter.

Input output

PS1 -- Kommandoer, udtryk og IO i Eiffel

Noter

Denne forelæsning har ikke ret meget at gøre med objekt-orienteret programmering. Der fokuseres på kommandoer (sætninger) og udtryk i Eiffel. Kommandoer og udtryk bliver anvendt til at definere detaljerne af operationer i klasser.

Inden vi er i stand til at beskrive assignments og operationsaktivering er det nødvendigt at beskrive hvornår to typer er sammenlignelige, således at objekter af disse typer kan assignes og overføres som parameter på en typesikker måde.

Den første del af denne forelæsning vil derfor handle om typer og typesammenlignelighed.

Oversigt over typer i Eiffel.

- Typer baseret på klasser.
  - Reference typer: typer definerede af uekspanderede klasser.
  - Ekspanderede typer.
    - Typer på formen “expanded T,” hvor T er defineret af en uekspanderet type.
    - Typer der er defineret af en klasse på formen “expanded class ...”.
  - Generisk afledte typer: Typer som er fremkommet ved at give aktuelle typeparametre til en generisk (typeparametriseret) klasse.
    - Kan være referencetyper eller ekspanderede typer.
  - Basale typer: typer der afspejler tal, boolske værdier, tegn og bits.
    - Specielle eksempler på ekspanderede typer.
  - Forankrede typer: typer der udtrykker “samme type som attribut/formel parameter/current.”

Noter

Alle typer i Eiffel falder tilbage på klasser.

På denne slide giver vi en oversigt over typer, og deres sammenhæng til forskellige former for klasser i Eiffel.

I dette kursus gør vi ikke ret meget ud af generiske afledte typer (og dermed generiske klasser). Hvis man ikke går alt for dybt ned i emnet om generiske klasser er disse lette at lave, og vigtist af alt, at anvende.

Ekspanderede typer er mindre væsentlige end reference typer. Vi studerer referencer typer grundigt. Ekspanderede typer studeres mere sporadisk. Vi kan ikke ignorere emnet, idet de basale typer (tal, boolean og tegn) er ekspanderede typer.

Forankrede typer vil ikke blive behandlet på kurset, men man kan ikke undgå at blive konfronteret med disse hvis man anvender klasser fra Eiffel bibliotektet.

I “Eiffel the Language” giver kapitel 12 en oversigt over typerne.

Lidt om de basale typer:

BOOLEAN er sandhedsværdierne **true** og **false** (nøgleord).

CHARACTER er de såkaldte ASCII tegn, (tegn i et standardiseret og meget anvendt, ordnet sæt af tegn. Vi vender tilbage til ASCII tegnsættet i forbindelse med forelæsningen om strenge.) Character_bits angiver antallet af bits i objekter af typen CHARACTER. Denne konstant findes i klassen PLATFORM.

INTEGER er heltal. (Integer_bits er en konstant i klassen PLATFORM, som angiver antallet af bits som benyttes i heltal).

REAL er 32 bit "floating point" tal (mere præcist enkelt præcissions relle tal af højst Real_bits længde, hvor Real_bits er en konstant i klassen PLATFORM).

DOUBLE er 64 bit "floating point" tal (mere præcist dobbelt præcissions relle tal af højst Double_bits længde, hvor Double_bits er en konstant i klassen PLATFORM)

BITS *M* er bit-strenge af længde *M*.

De basale klasser er beskrevet i kapitel 32 af “Eiffel the Language”.

Typesammenlignelighed.

Typesammenlignelighed er en relation mellem to typer.

Relationen er asymmetrisk, refleksiv og transitiv.

Praktisk anvendelse af typesammenlignelighed:

v := e	Antag at v er erklæret som v: T. Assignmentet er lovligt i typemæssig forstand, hvis typen af e er *typesammenlignelig* med T.
f(e)	Antag at f er en procedure eller funtion defineret som f(p: T). Kaldet er lovligt i typemæssig forstand, hvis typen af e er *typesammenlignelig* med T.

Relationen *typesammenlignelighed* er den refleksive transitive lukning af relationen *direkte typesammenlignelighed:*

- **Refleksivitet**: En type er altid sammenlignelig med sig selv.
- **Transitivitet**: T1 er sammenlignelig med T2, T2 er sammenlignelig med T3 medfører, at T1 er sammenlignelig med T3.

Noter

Det er vigtigt at forstå, hvad det betyder, at relationen typesammenlignelighed er asymmetrisk: Hvis T er sammenlignelig med S, er S ikke nødvendigvis sammenlignelig med T.

Strategien i definitionerne på denne slide er følgende: Vi definerer relationen *type sammenlignelighed* ved først at definere en simplere relation, *direkte typesammenlignelighed.* Direkte typesammenlignelighed defineres på næste slide.

I "Eiffel the Language" og OOSC svarer det, vi har valgt på dansk at kalde typesammenlignelighed, til *type conformance.* I "Eiffel the Language" beskriver kapitel 13 dette emne i stor detalje.

Direkte typesammenlignelighed.

1. INTEGER er direkte typesammenlignelig med REAL.

2. REAL er direkte typesammenlignelig med DOUBLE.

3. BITS*N* er direkte typesammenlignelig med BITS*N+1*.

4. For enhver type T (som ikke er af en BIT type) eksisterer der et positivt heltal Bit_size_T så
   T er direkte typesammenlignelig med BIT Bit_size_T

Noter

Med vort nuværende kendskab til objekt-orienterede begreber og til Eiffel er direkte typesammenlignelighed helt triviel at definere og at forstå. Det eneste, vi behøver at definere, er, hvordan nogle få, simple typer forholder sig til hinanden parvis.

Hvad angår typesammenlignelighed (som er defineret ud fra direkte typesammenlignelighed) gælder jo altid, at en type er sammenlignelig med sig selv. Det vil sige, at vi kan assigne værdier til variable, og overføre værdier som parametre, forudsat værdier og variable/parametre har identiske typer. De eksplicit givne regler for direkte typesammenlignelighed giver os lidt flere muligheder. Vi kan f.eks. tilskrive et heltal til en variabel af typen DOUBLE. Dette er lovligt, idet typesammenlignelighed er en transitiv lukning af direkte typesammenlignelighed (INTEGER er direkte typesammenlignelig med REAL, som igen er direkte typesammenlignelig med DOUBLE. Ergo er INTEGER typesammenlignelig med DOUBLE). Næste slide introducerer DOUBLE, INTEGER m.m. i forhold til typer, der er defineret ud fra klasser.

Regel 4 gør det muligt at konvertere en vilkårlig type til bits, *men absolut ikke omvendt.* Dette tillader os at tilgå den binære repræsentation af alle vore data. Dette er kun nødvendigt når vi skal lave grænseflader til lavniveau komponenter. Man skal være klar over, at tilgang til bit-repræsentationen af data er maskinafhængig, og som sådan "gift" for højniveau programmer der skal kunne køre på tværs af forskellige maksintyper.

Når vi senere introducerer nedarvning mellem klasser, introducerer vi samtidig subtyper. Derved kommer der mere kød på relationen *direkte typesammenlignelighed.* (Se forelæsningen "Nedarvning II").

Hvis man også betragter generiske klasser, og dermed typer der har typer som parametre, bliver der føjet endu et par regler til *direkte typesammenlignelighed* . Vi vil dog ikke i dette kursus gøre ret meget ud af generiske klasser og parametriserede typer.

Kommandoer i Eiffel.

Udførelse af en kommando ændrer programmets tilstand.

Basale kommandoer:

Assignment.

Procedurekald på objekt

Den tomme kommando.

Kontrolstrukturer:

Sekvens: **;**

Selektion:
if, **inspect**.

Iteration:
loop.

Specielle kommandoer:
check **debug**

PS1 -- Kommandoer, udtryk og IO i Eiffel

Noter

Denne slide giver et en oversigt over kommandoer i Eiffel.

Bemærk den ofte oversete kontrolstruktur: sekvens, der noteres med et semikolon. Kommandoer i sekvens udfører først den førstangivne kommando, dernæst rekursivt sekvensen af de øvrige kommandoer. I Eiffel3 er det frivilligt om man medtager semikolon mellem to kommandoer i sekvens. Det anbefales dog at have et semikolon mellem kommandoer i sekvens, for at understrege og eksplicitere den omtalte kontrolform.

Assignments i Eiffel.

```
Variabel: Type;
....
Variabel := udtryk
```

Variabel:

- Attribut i en klasse.
- Lokal variabel i en procedure eller funktion.
- *Result* (resultatet af en funktion).

Begrænsning: typen af udtrykket er *typesammenlignelig* med 'Type'

Semantik:

'Type' er en reference-type:

- 'Variabel' tilskrives en reference til det objekt, som er et resultat af beregningen af 'udtryk'.

'Type' er en ekspanderet type (integer, boolean, character, real, ekspanderet klasse):

- 'Variabel' tilskrives objektet, som er resultatet af beregningen af 'udtryk'.

PS1 -- Kommandoer, udtryk og IO i Eiffel

Noter

Som man kan se, er meningen med et assignment afhængig af variablens type-erklæring.

Inden programudførelsen kan det lade sig gøre at bestemme et udtryks type. (Dette kan lade sig gøre ved at konsultere indgående variables erklærede typer og funktioners resultattyper.) Dette er statisk 'typing'. Lad os kalde udtrykkets type for 'udtryk-type'. Som det ses, er det et krav, at 'udtryk-type' er sammenlignelig med 'type'.

Hvis typen af variablen er en reference type (dvs. at værdierne vi arbejder med er referencer til objekter, og dermed ikke objekterne selv) skal udtrykket resultere i en instans af en klasse (et objekt). I dette tilfælde tildeles variablen en *reference* til dette objekt. Hvis udtryk returnerer **void**, tildeles variabel værdien **void.**

I kapitlet “Basal objekt-orienteret programmering II” har vi allerede set på assignment af blandede reference typer og ekspanderede typer. Der henvise til sliden med titlen “reference og værdisemantik af assignment”.

Hvis typen af variablen er en såkaldt ekspanderet type, typisk (men ikke nødvendigvis) een af de prædefinerede, simple typer, tildeles variablen et objekt (et heltal, en real etc.) som værdi. Man skal bemærke, at hvis udtryk returnerer **void** i dette tilfælde, opstår der en fejl.

Procedurekald (I).

-- i betydningen kald af procedure på et objekt.

Et kald af en procedure P på et objekt kan antage følgende former:

```
P
P(x, y, ...)
```
ukvalificerede kald.

```
Expr.P
Expr.P(x, y, ....)
```
kvalificerede kald, enkelt dot.

```
Expr1. Expr2. ... Exprn.P
Expr1. Expr2. ... Exprn.P(x, y, ...)
```
kvalificerede kald, multi dot.

```
P(x, y, ...)  ~ Current.P(x, y, ...)

Expr1. Expr2. ... Exprn.P(x, y, ...)  ~
   temp1 := Expr1. Expr2;
   temp2 := temp1. Expr3;
   ....
   tempn-1.P(x, y, ...)
```

PS1 -- Kommandoer, udtryk og IO i Eiffel

Noter

Procedurekald kan være med eller uden parametre.

Foruden parameter-karakteristika, kan formen af et procedurekald enten være et ukvalificeret, et kvalificeret enkelt-dot eller et kvalificeret multi-dot. "Dotning" forekommer, når vi vil angive, at et kald foregår på et eksplicit bestemt objekt. "Multi-dotning" forekommer, hvis det ønskede procedurekald skal foregå på et objekt, der nås ved at "gå igennem" et antal objekter.

Det skal bemærkes, at ved multi-dotning kan Expri være et udtryk (såsom et funktionskald), der returnerer et objekt. Eksempelvis kan vi se på

```
x(5).y(6,x).z(7,8)
```

i hvilken x og y er funktioner af hhv. 1 og 2 parametre.

Den nederste del af denne slide viser, hvordan man skal forstå ukvalificerede kald og kvalificerede multi-dotning i termer er enkelt-dotning.

Hvis vi ser bort fra skelnen mellem ingen eller flere parametre, kan vi altså reducere problemstillingen til kun at betragte tilfældet Expri.P(x, y, ...), hvor Expri er et udtryk uden dot.

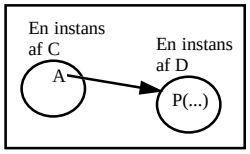
På næste slide vil vi beskrive begrænsninger og semantik af dette tilfælde.

Procedurekald (II).

```
Expr.P(x, y, ....)
```

```
class C export
  ....
feature

  operation is
    do
      Expr.P(x, y, ....)
    end
end
```



Antagelser og begrænsninger:

- Ovenstående forekommer i en operation i klassen C.
- Expr er en attribut A eller et kald af en funktion fra klassen C.
- Typen af en aktuel parameter skal være *typesammenlignelig* med typen af den tilsvarende formelle parameter.
- Værdien af Expr skal være et objekt (ikke **void**), lad os sige af klassen D.
- P er synlig i klassen C.

Semantik:

- Initialiser P's formelle parametre til værdierne af x, y osv. Initialiser P's lokale variable til default værdier.
- Udfør kroppen af P på det objekt, der er værdien af Expr (instansen af D).

Noter

På denne slide beskrives dels forudsætningerne for at kunne udføre et procedurekald på et objekt, dels semantikken af kaldet.

Det er vigtigt at få fat i, at enhver operation forekommer i en sammenhæng. I programbeskrivelsen er denne sammenhæng en bestemt klasse (og i en bestemt operation) (her 'operation' i klassen C). I programudførelsen er sammenhængen et bestemt objekt.

Når værdien af et aktuelt parameterudtryk A bindes til et formelt parameternavn N, skal de samme typeforhold gælde, som hvis vi assignede på følgende måde: N := A.

I forelæsningen vil vi studere et antal mere konkrete og praktiske eksempler på ovenstående.

Den tomme kommando.

Noter

Den tomme kommando (null command) noteres som den tomme streng (ingenting). Den tomme kommando efterlader programmets tilstand uforandret.

De betingede kommandoer i Eiffel.

```
if boolean_expression_1 then
        command_sequence_1
{elseif boolean_expression_2 then
        command_sequence_2}
[else
        command_sequence_3]
end
```

Semantik:

De boolske udtryk testes i rækkefølge. Det første udtryk, som returnerer *true*, foranlediger at den efterfølgende kommando-sekvens udføres. Herefter terminerer if kommandoen.

```
inspect expression
{when choises then
  command_sequence}
[else
  command_sequence]
end
```

Semantik:

Expression beregnes til en værdi. Dette skal være af typen integer eller character. På basis af værdien udføres den kommando-sekvens, hvis 'choises' indeholder værdien. Hvis intet 'choise' indeholder værdien, udføres else kommando-sekvensen.

PS1 -- Kommandoer, udtryk og IO i Eiffel

Noter

I syntaksen for if-kommandoen forekommer to forskellige former for syntaksnotation: {x} betyder her 0, 1 eller flere forekomste af x. [y] betyder 0 eller 1 forekomst af y; Vi siger også, at y er "optional".

Bemærk, at notationen {x} er lidt forskellig fra bogens. Forskellen består i, at bogen i forbindelse med 0, 1 eller flere forekomster af en konstruktion x også beskriver et separatorsymbol mellem disse. Sammenlign med bogens syntax-notation for if kommandoen, hvor nøgleordet **elseif** spiller rollen af en sådan separator.

Den beskrevne semantik for if kommandoen kan gøres mere præcis, hvis man bruger en rekursiv formulering. Det overlades til læserne som en øvelse at gøre dette.

Inspect kommandoen i Eiffel svarer til case-sætningen i Pascal.

I inspect kommandoen skal værdien af udtrykket være at finde i netop ét "choice". I denne sammenhæng betragtes else-delen som et default valg. Hvis værdien ikke findes i et "choice", og hvis der ikke er nogen else-del, opstår der en fejl.

Bemærk at rækkefølgen af “choices” og tilhørende kommandoer er ligegyldig i inspect, men at rækkefølgende af boolske udtryk og kommandoer i if kontrolstrukturen er af stor betydning.

Der henvises til "Eiffel the language", afsnit 14.5, for en mere fyldestgørende og mere præcis beskrivelse af inspect kommandoen.

Den iterative kommando i Eiffel.

```
from initialization_commands
until boolean_expression
loop command_sequence
end
```

Semantikken af loop kommandoen kan defineres ved følgende ækvivalens til Pascal´s while-kommando:

```
initialization_commands;
while not boolean_expression
do begin
      command_sequence
   end
```


Noter

Ovenstående loop kommando er den eneste iterative kommando i Eiffel.

Som anskueliggjort minder semanikken af loop konstruktionen om en Pascal while sætning, bortset fra at initialiserings-kommandoerne ikke er en del af en while sætning, og bortset fra at det boolske udtryk i loop er en "exit-betingelse" og ikke en "fortsættelses-betingelse".

Der findes ingen repeat-agtig eller for-agtig iterationskommando i Eiffel.

Syntaksen af loop konstruktionen er, desværre, meget forskellig fra mere konventionelle (læs: Pascal-agtige) iterations-kommandoer.

Der er to andre bestanddele af en loop-konstruktion. Det drejer sig om en invariant-del og en variant-del. Ingen af disse er påkrævede. Disse diskuteres på næste slide. Invarianter og varianter diskuteres også i 7.8.1 i OOSC.

Invariant og variant i Eiffel's iterations-kommando.

```
from initialization_commands
invariant assertion
variant integer_expression
until boolean_expression
loop command_sequence
end
```

Løkke-invariant:

- Et assertion er en sekvens af formelle eller uformelle boolske udtryk.
- Et uformelt boolsk udtryk er en kommentar.
- Invarianten skal være opfyldt
  - efter initialiserings-delen
  - efter hver iteration (specielt efter den sidste iteration).

Løkke-variant:

- Et heltalsudtryk.
- Skal være ikke-negativ efter initialiserings-delen og efter hver iteration.
- Skal tælles én ned for hver iteration.

PS1 -- Kommandoer, udtryk og IO i Eiffel

Noter

Løkke-invarianten og varianten udtrykker tilsammen korrektheds-kriterier for den iterative kommando.

Invarianten udtrykker hvilken betingelse der skal holdes konstant og uforandret før, mellem og efter iterationen. Betingelsen er, at alle de boolske udtryk er opfyldt på de nævnte steder.

Invarianten er af central betydning, når man skal udtale sig om, hvilken effekt løkken har haft på programmets tilstand.

Efter loop konstruktionen gælder invarianten **and** boolean_expression, hvor boolean_expression er det boolske udtryk efter **until.**

Varianten udtrykker, at loop kommandoen terminerer efter et endeligt antal skridt. Antag f.eks. at varianten er et udtryk, hvis værdi i starten er 100. Hvis variantens værdi skal tælles ned for hver iteration kan løkken maksimalt gennemløbes 100 gange, uden at varianten bliver negativ.

Varianten er således af betydning, når man vil sikre sig, at loop-kommandoen terminerer.

Både invarianten og varianten involverer udtryk, hvori der indgår variable, som ændres i løkken.

I Eiffel programmeringsomgivelsen kan man bede om, at variant og invariant betingelserne testes under udførelsen af løkken, og at programudførelsen afbrydes, hvis reglerne for disse brydes. Der henvises til kapitlet om Eiffel omgivelsen for detaljer.

I forelæsningen vil vi se på konkrete og praktiske eksempler på brug af variant og invariant.

I en følgende forelæsning (“Specifikation med logiske udtryk”) kommer vi tilbage til brug af assertions, nemlig som en måde at udtrykke en programspecifikation. I denne kommende forelæsning vil vi på en grundig måde studere en beslægtet form for invariant, som kaldes en klasseinvariant.

Eksempel på anvendelse af løkkeinvariant.

```
div (x, y: INTEGER): DIVISION is
require x > 0; y > 0;
local rest, quotient: INTEGER
do
  from rest := x; quotient := 0
  invariant rest + quotient * y =  x
  variant x - result
  until rest < y
  loop
    rest := rest - y;
    quotient := quotient + 1
  end;
  result.create(y, quotient, rest);
ensure rest < y;  rest >= 0;  x = result.dividend
end;
```


Noter

Denne slide viser en funktion, som beregner x div y og x mod y, og som returnerer begge dele i et objekt af typen DIVISION. Klassen DIVISION er skitseret på næste slide.

I ovenstående er x divided, y er divisor, og i resultatet vil vi (bl.a.) registrere resten og kvotienten.

Klassen DIVISION.

```
Class DIVISION
creation create
feature
  quotient, rest, divisor: INTEGER;

  create(d, q, r: INTEGER) is
  do
    divisor := d; quotient := q; rest := r
  end;

  dividend: INTEGER is
  do
    result := quotient * divisor + rest
  end;
invariant rest < divisor; rest >= 0 ;
              -- 'quotient' og 'rest' er hhv. kvotient og rest ved divison af dividend
              -- med divisor.
end -- class DIVISION
```


Noter

Denne klasse anvendes i eksemplet på forrige slide.

Specielle kommandoer i Eiffel.
-- Check og Debug

```
check
  assertion_sequence
end
```

Semantik:

Udsagnene i sekvensen af assertions skal alle være opfyldte i programmets nuværende tilstand.

```
debug
  command_sequence
end
```

Semantik:

Når programmet er oversat med "debug" compiler operation, udføres sekvensen af kommandoer. Uden "debug" option er debug-kommandoen ækvivalent med den tomme kommando.

Noter

Efter nøgldeordet “debug” kan der anføres en manifest streng. Debugging kan bl.a. aktiveres, når denne streng nævnes i Ace system beskrivelsesfilen i en debug option. For nærmere oplysninger om dette konsulter 14.8 og appendix D.11 i “Eiffel the Language”.

Check kommandoen er beskrevet i afsnit 9.13 i “Eiffel the Language”. Assertions_sequence beregnes når assertion checkning angives til et bestemt niveau. Se “Eiffel the Language” afsnit 9.17 for detaljer, samt sliden om compiler options i kapitet “Eiffel programmeringsomgivelsen” i disse noter.

Udtryk i Eiffel.

Et udtryk er en konstruktion, som kan beregnes, og som dermed returnerer en værdi til omgivelserne.

I sprog med 'statisk typing', såsom Eiffel, kan typen af alle udtryk bestemmes, før programmets udførelse.

Et udtryk efterlader programmets tilstand uforandret.

Forskellige former for udtryk:

- Funktionskald.
- Variabelnavn.
- Operatorudtryk.
- Konstantudtryk.
- *Current.*

PS1 -- Kommandoer, udtryk og IO i Eiffel

Noter

Semantikken af funktionskald svarer nøje til semantikken af procedurekald. Dog returnerer en funktion en værdi. Inde i funktionen sker dette ved at tildele variablen *result* en værdi. Funktionens resultat er værdien af *result* på det tidspunkt, funktionen returnerer. Bemærk at tilskrivning af en værdi til *result* ikke i sig selv medfører terminering af funktionen.

Et funktionskald kan naturligvis kun indgå i en sammenhæng, hvor "der skal bruges en værdi". Således kan et funktionskald ikke bruges som en kommando.

I sammenhænge, hvor der forventes en værdi, kan man også blot referere til en variabel. Dette tilfælde kan, som tidligere påpeget, ikke skelnes fra et kald af en parameterløs funktion.

Current er allerede diskuteret i en tidligere forelæsning.

Tilbage er der blot at gøre rede for operator-udtryk og konstantudtryk. Dette vil ske på de kommende slides.

Operatorudtryk i Eiffel.

Unære	**not**	+	-		
Binære	=	/=	//	\\	^
	<	>	<=	>=	
Multiære	+	-	*	/	
	and	**and then**	**or**	**or else**	
	xor	**implies**			

x **and then** y: Hvis x er falsk, er udtrykket falsk, *og udtrykket y beregnes ikke.*

x **or else** y: Hvis x er sand, er udtrykket sand, *og udtrykket y beregnes ikke.*

Noter

Sammensatte operator-udtryk beregnes med hensyntagen til associations-regler og operator dominans (precedence) efter sædvanlige principper. Der henvises til "Eiffel the language" afsnit 23.5 for en konkret diskussion.

Det kan lade sig gøre at definere alle operatorerne på nær = og /= på objekter af egne klasser. Vi vil ikke i dette kursus komme nærmere ind på, hvordan dette kan gøres, men det er relativt simpelt. Denne facilitet kan benyttes til at definere naturlig notation på objekter af egne typer. F.eks. hvis vore egne objekter er naturligt ordnede, kan vi definere den binære operation < og skrive x < y, i stedet for f.eks. x.less_than(y). Se "Eiffel the language" afsnit 5.13.

Bemærk at // betyder heltalsdivision (ofte betegnet “div” i andre sprog). \\ betyder resten ved heltalsdivision (ofte betegnet “mod”).

Konstantudtryk i Eiffel.

	Eksempler	Ekspanderet/uekspanderet
Integer	0 -5 +6	Ekspanderet
Real	5. .5 5.5 +5.5 -5.5 5.3e3	Ekspanderet
Boolean	**true false**	Ekspanderet
Character	'a' 'b' 'c'	Ekspanderet
String	"Dette er en streng"	Uekspanderet

Noter

I mange bøger om programmeringssprog kaldes konstantudtryk også for *literals*.

Tegn-konstanter: Det er ikke umiddelbart muligt ved hjælp af quote notationen at notere tegn såsom tabulator og "back space". Derfor findes der en række special notationer for disse, f.eks. %T og %B. Se "Eiffel the language" i afsnit 23.17 og afsnit 25.15 for detaljer.

Typen String er ikke en ekspanderet type. En streng såsom "pip" er derfor et objekt af klassen STRING. Denne klasse understøtter imidlertid en speciel instantierings- og initialiseringsnotation, nemlig "...". Se endvidere forelæsningen om strenge samt 13.4 i OOSC (og 23.18 i “Eiffel the Language”).

Det skal bemærkes, at også for arrays findes der en konstant udtryksform. Denne vil blive omtalt i kapitlet “Arrays og lister samt Stakke og Køer”.

Input-Output i Eiffel.

Input-output operationer er ikke en del af sproget Eiffel.

Input-output håndteres via fil-operationer.

Input-output operationer (og fil-operationer i det hele taget) stilles til rådighed af en klasse, der hedder FILE.

I Eiffel findes der desuden en klasse STD_FILES, som er en klient af FILE.

- SDT_FILES erklærer tre filer
  input, output, error: FILE
- input, output og error spiller rollen af standard input, standard output og standard error i operativsystemet UNIX.
- input, output, og error åbnes ved den første anvendelse af dem.

I enhver klasse findes der en variabel io af typen STD_FILES.

PS1 -- Kommandoer, udtryk og IO i Eiffel

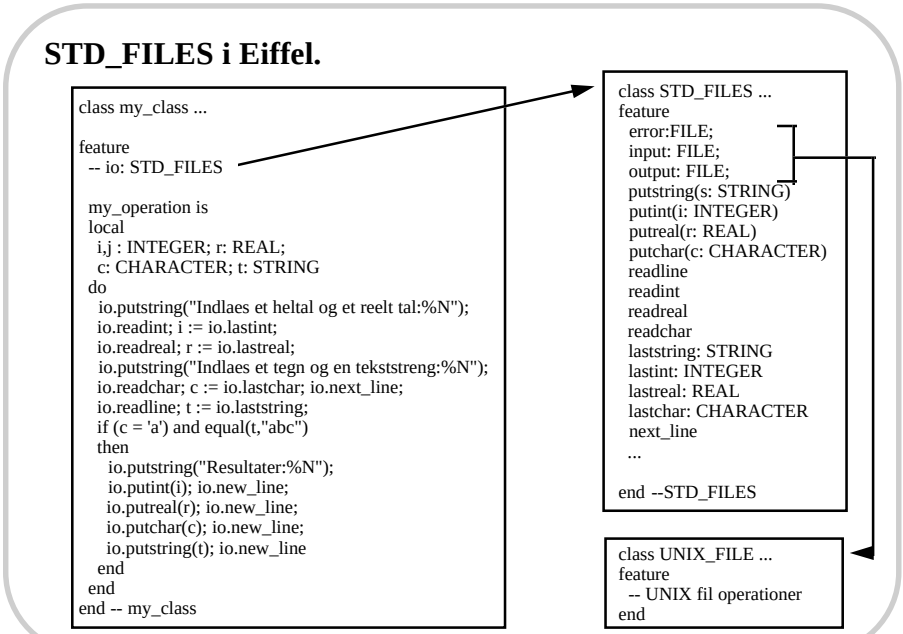
Noter

I moderne programmeringssprog er det meget typisk, at input output ikke er en del af selve programmeringssproget.

Det er ikke rent trylleri eller "snyd", der gør variablen io tilgængelig i enhver klasse. Alle klasser arver fra klassen som hedder ANY, og man kan tænke på io som en variabel i ANY. (Reelt er io en once-funktion, som ved første aktivering returnerer en instans af typen STANDARD_FILES). Nedarvning mellem klasser er en vigtig facilitet i objekt-orienterede programmeringssprog, og vi vender tilbage til dette lidt senere i kurset.

Det er ikke intensionen i denne forelæsning at dække detaljer om input-output. Der vil blive gjort rede for den overordnede filosofi og organisering. Der henvises til OOSC appendix E, og kapitel 31 i "Eiffel the Language". Kapitel 13.5 i "Eiffel the Libraries" handler også om input-output og filer.

STD_FILES i Eiffel.



PS1 -- Kommandoer, udtryk og IO i Eiffel

Noter

Operationen my_operation i den viste klasse, my_class, viser et eksempel på, hvordan man kan foretage simpel input-output i Eiffel. (Det eneste der mangler er en creation procedure, der kalder my_operation). I eksemplet indlæses først et heltal og et reelt tal, og dernæst et tegn og en streng. Hvis tegnet er 'a' og strengen er “abc” udskrives de indlæste data.

Variablen io er tilgængelig via implicit nedarvning fra klassen ANY, og variablen er af typen STD_FILES. Det betyder i praksis at IO altid er “ved hånden” i de klasser vi skriver.

Klassen FILE er en abstrakt klasse. Klassen UNIX_FILE er en konkret klasse, hvis operationer afspejler filbegrebet som det kendes fra UNIX.

Bemærk kaldet af next_line ovenfor. Next_line positionerer os på næste inputline på standard input. I det viste eksempel er det kun nødvendigt at bruge next_line efter læsning af tegnet.

Hvis man vil lege videre med dette eksempel kan man hente filen fra /user/normark/public/p1/my_class.e. I forhold til ovenstående indeholder filen også en creation procedure.

Ud over output-mulighederne via STANDARD_FILES kan man på ethvert objekt blot kalde print, som arves fra klassen ANY. Se “Eiffel the Language” afsnit 27.2 for detaljer.