# Introduktion til PL1 og COBOL i Mainframe-miljøer

## Introduktion til mainframe-miljøer

Mainframes anvendes fortsat i store organisationer til drift af kritiske forretningssystemer, især i bank-, forsikrings- og offentlige sektorer. Programmering på mainframe adskiller sig fra moderne udviklingsmiljøer ved sin afhængighed af terminalbaseret adgang, datasætstrukturer og batchorienteret afvikling.

Udviklingsarbejdet foregår typisk på IBM z/OS via **TSO/ISPF** eller gennem **3270-emulatorer**, hvor filer og programmer håndteres i såkaldte *datasets*. Et dataset fungerer som en fil, men kan indeholde flere *members* (programmer eller komponenter).

Kendskab til kommandolinjen er centralt for at kunne kompilere, afvikle og fejlfinde programmer effektivt.

---

## Grundlæggende struktur i PL/I og COBOL

Både PL/I og COBOL er stærkt strukturerede sprog, men de adskiller sig i udtryksform og syntax. PL/I minder i struktur om C og Pascal, mens COBOL har et mere verbalt og deklarativt udtryk.

### PL/I – eksempel på et simpelt program:

```
/* Simple PL/I program */ HELLO: PROC OPTIONS(MAIN); PUT SKIP LIST('Hello, Mainframe world!'); END HELLO;
```

Et PL/I-program består typisk af en procedure med en `PROC`-deklaration, efterfulgt af programlogik og afsluttet med `END`.

### COBOL – tilsvarende eksempel:

```
    `IDENTIFICATION DIVISION.        PROGRAM-ID. HELLO.        PROCEDURE DIVISION.
  DISPLAY "Hello, Mainframe world!".            STOP RUN.`
```

COBOL er opdelt i divisioner, sektioner og paragrafstrukturer.
Programmeringslogikken ligger i **PROCEDURE DIVISION**, mens data deklareres i **DATA DIVISION**.

---

# Datahåndtering og filstrukturer

Mainframes arbejder typisk med sekventielle datafiler, VSAM-filer eller databaser som DB2. Begge sprog giver mulighed for eksplicit håndtering af record-baserede strukturer.

## COBOL – fildeklaration og læsning:

```
   `FILE SECTION.        FD  CUSTOMER-FILE.        01  CUSTOMER-RECORD.
05  CUST-ID        PIC 9(5).            05  CUST-NAME      PIC X(30).
PROCEDURE DIVISION.            OPEN INPUT CUSTOMER-FILE            READ CUSTOMER-
FILE            DISPLAY CUST-NAME            CLOSE CUSTOMER-FILE.`
```

Her defineres en recordstruktur og filadgang gennem FD- og WORKING-STORAGE-sektioner.

## PL/I – tilsvarende filhåndtering:

```
DCL CUST_FILE FILE INPUT; DCL 1 CUSTOMER_RECORD, 2 CUST_ID FIXED DEC(5), 2 CUST_NAME CHAR(30); OPEN FILE(CUST_FILE); READ FILE(CUST_FILE) INTO(CUSTOMER_RECORD); PUT SKIP LIST(CUST_NAME); CLOSE FILE(CUST_FILE);
```

Begge sprog kræver en tydelig forståelse af dataformater og recordlængder, især ved håndtering af EBCDIC-encoded data.

---

# Pointere og procedurer (PL/I)

PL/I tilbyder mere avancerede muligheder for dynamisk hukommelsesstyring end COBOL. Pointere bruges til at referere til datafelter og strukturer i hukommelsen, hvilket gør det muligt at opbygge dynamiske datastrukturer som lister eller tabeller.

**Eksempel – pointer i PL/I:**

```
DCL PTR POINTER; DCL 1 EMPLOYEE BASED(PTR), 2 NAME CHAR(20), 2 SALARY FIXED DEC(7,2); ALLOCATE EMPLOYEE; NAME = 'SMITH'; SALARY = 50000.00; PUT SKIP LIST(NAME, SALARY); FREE EMPLOYEE;
```

Pointere og dynamisk allokering kræver omhyggelig fejlhåndtering, da forkert frigivelse eller reference kan føre til fejl under kørsel.

---

# JCL og batchafvikling

JCL (Job Control Language) anvendes til at beskrive, hvordan et program skal afvikles på mainframe. Det styrer input, output og ressourceallokering.

Et typisk JCL-job består af tre sektioner: `JOB`, `EXEC` og `DD` (Data Definition).

**Eksempel – JCL-job til afvikling af et COBOL-program:**

```
//HELLOJOB JOB (ACCT),'COBOL RUN',CLASS=A,MSGCLASS=X //STEP1 EXEC PGM=HELLO
//SYSPRINT DD SYSOUT=* //SYSOUT DD SYSOUT=* //INPUT DD DSN=USER.DATA.INPUT,DISP=SHR
```

Her definerer `JOB`-kortet selve jobbet, `EXEC` angiver det program, der skal køres, og `DD`-kortene specificerer de datasets, programmet skal bruge.

Forståelse af JCL er afgørende for at kunne teste og eksekvere programmer i batchmiljøer.

---

# Fejlhåndtering og debugging

Fejlhåndtering i PL/I og COBOL adskiller sig fra moderne sprog. PL/I anvender *ON conditions* til struktureret fejlbehandling, mens COBOL benytter filstatuskoder og return codes.

**PL/I – eksempel på fejlhåndtering:**

```
ON ENDFILE(INPUT) PUT SKIP LIST('End of file reached.');
```

**COBOL – eksempel med filstatus:**

```
    `IF FILE-STATUS NOT = "00"            DISPLAY "Error reading file"
STOP RUN.`
```

Fejl identificeres ofte gennem joblogs og systemgenererede dumps. Værktøjer som **IBM Debug Tool** og **ISPF-browse** anvendes til analyse af output og fejlfinding.

---

# Databaser, SQL og DB2-integration

Databaser udgør en central del af mainframe-applikationer. I IBM-miljøer er **DB2 for z/OS** den dominerende relationelle databaseplatform.
Både COBOL og PL/I kan kommunikere med DB2 gennem **embedded SQL**, hvor SQL-kommandoer skrives direkte i kildekoden og oversættes af præprocessoren før kompilering.

## Grundlæggende principper for embedded SQL

Embedded SQL muliggør integration mellem applikationslogik og databaseoperationer uden behov for et separat database-API.
Programmet kan udføre SELECT-, INSERT-, UPDATE- og DELETE-kommandoer mod DB2-tabeller ved hjælp af såkaldte **host variables** — dvs. variabler i COBOL- eller PL/I-programmet, som refereres til i SQL-sætninger.

Strukturen for embedded SQL er den samme i begge sprog:

- SQL-kommandoer skrives mellem `EXEC SQL` og `END-EXEC` (i COBOL) eller `EXEC SQL` / `END-EXEC;` (i PL/I).
- Data udveksles via host-variabler, som skal deklareres før brug.
- Forbindelsen til DB2 sker via en *plan* eller *package*, som defineres i kompilations- og deployments fasen.

---

## DB2 og programmeringsworkflow

Et typisk workflow for et COBOL- eller PL/I-program, der bruger DB2, består af følgende trin:

1. **Kildekode med embedded SQL** skrives og gemmes.
2. **DB2 precompiler** oversætter SQL-kommandoer til databasekald og genererer en modificeret programfil samt en *DBRM* (Database Request Module).
3. **Kompilering** af det prekompilerede program udføres.
4. **Binding** af DBRM til en DB2-plan eller -package, hvilket giver databasen kendskab til, hvilke SQL-kommandoer programmet kan udføre.
5. **Afvikling** sker via JCL, hvor programmet forbindes til DB2-runtime.

---

## COBOL og DB2 – eksempel på embedded SQL

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. EMPLOYEE-READ.
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01  WS-ID         PIC 9(5).
    01  WS-NAME       PIC X(30).
    01  SQLCODE       PIC S9(9) COMP.

    PROCEDURE DIVISION.
        MOVE 10001 TO WS-ID.
        EXEC SQL
            SELECT NAME
```

```
            INTO :WS-NAME
            FROM EMPLOYEE
           WHERE ID = :WS-ID
    END-EXEC.

    IF SQLCODE = 0
        DISPLAY "EMPLOYEE NAME: " WS-NAME
    ELSE
        DISPLAY "ERROR: SQLCODE=" SQLCODE
    END-IF.
    STOP RUN.
```

**Forklaring:**

- `WS-ID` og `WS-NAME` fungerer som **host variables**, der binder COBOL-data til SQL-spørgsmålet.
- `SQLCODE` indeholder returneringskoder fra DB2 (0 = succes, positive = advarsler, negative = fejl).
- DB2 stiller altid `SQLCA` (SQL Communication Area) til rådighed som en standardstruktur for fejlhåndtering og diagnostik.

---

# PL/I og DB2 – tilsvarende eksempel

```
EMPLOYEE_READ: PROC OPTIONS(MAIN);
DCL EMP_ID FIXED DEC(5) INIT(10001);
DCL EMP_NAME CHAR(30);
DCL SQLCODE FIXED BIN(31);

EXEC SQL
SELECT NAME INTO :EMP_NAME
FROM EMPLOYEE
WHERE ID = :EMP_ID
END-EXEC;

IF SQLCODE = 0 THEN
PUT SKIP LIST('EMPLOYEE NAME: ', EMP_NAME);
ELSE
PUT SKIP LIST('SQL ERROR: ', SQLCODE);
END;
END EMPLOYEE_READ;
```

**Forklaring:**

- PL/I anvender også **host variables** markeret med kolon ( `:` ).
- `SQLCODE` er et centralt kontrolpunkt for fejlstatus.
- Programmet udføres typisk gennem et JCL-script, der specificerer DB2-plan og runtime-miljø.

---

# Fejlhåndtering og SQLCODE

Ved alle DB2-operationer skal programmer kontrollere `SQLCODE` eller `SQLSTATE`.
De mest almindelige værdier er:

| SQLCODE | Betydning |
|---|---|
| 0 | Operation udført korrekt |
| +100 | Ingen rækker fundet (end of data) |
| -803 | Dobbelt nøgle (unikhedsfejl) |
| -911 | Deadlock eller timeout |
| -904 | Ressource utilgængelig |

# Best Practices ved brug af DB2

1. **Valider altid SQLCODE efter hvert databasekald.**
2. **Brug COMMIT/ROLLBACK aktivt** for at sikre dataintegritet i batchprogrammer.
3. **Definér eksplicitte datatyper**, så host-variabler matcher kolonnedefinitioner.
4. **Undgå unødvendige FETCH- og UPDATE-operationer**, der belaster I/O.
5. **Udnyt static SQL** hvor performance er kritisk, og brug dynamic SQL kun hvor nødvendigt.

# Moderne udviklingsmiljøer, kompilering og run environments

Traditionelt udvikles PL/I- og COBOL-programmer direkte på mainframe via **TSO/ISPF**, hvor kode redigeres, kompileres og afvikles i terminalbaserede miljøer. Denne arbejdsform er fortsat central i mange organisationer, men moderne run environments og værktøjer har gjort det muligt at udvikle, teste og kompilere mainframekode på mere fleksible måder – ofte som en del af integrerede DevOps-pipelines.

## Moderne udviklingsværktøjer

Flere moderne værktøjer udvider de klassiske udviklingsmiljøer:

- **IBM Developer for z/OS (IDz)** – et Eclipse-baseret IDE, der giver syntaksfremhævning, fejlfinding og direkte adgang til z/OS.
  IDz kan både oprette, kompilere og afvikle programmer på mainframe, men tillader samtidig lokal redigering og Git-integration.
- **Visual Studio Code med Zowe CLI** – muliggør lokal udvikling i et moderne miljø med terminaladgang til z/OS.
  Ved hjælp af Zowe CLI kan udviklere interagere med datasets, afvikle JCL-jobs og hente build-logs direkte i editoren.

---

# Kompilering af programmer

Kompilering på mainframe foregår som regel via batch-jobs defineret i JCL, hvor kompileren og efterfølgende link-edit specificeres som trin i jobben.
Programmet oversættes, og der genereres et *load module* i et **load library** dataset.

**Eksempel – JCL til COBOL-kompilering:**

```
//COBOLCMP JOB (ACCT),'COMPILE COBOL',CLASS=A,MSGCLASS=X
//STEP1 EXEC PGM=IGYCRCTL
//SYSIN DD DSN=USER.PROJECT.SOURCE(HELLO),DISP=SHR
//SYSLIN DD DSN=&&LOADSET,UNIT=SYSDA,SPACE=(CYL,(1,1)),DISP=(MOD,PASS)
//SYSLIB DD DSN=USER.COPYLIB,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLMOD DD DSN=USER.PROJECT.LOAD(HELLO),DISP=SHR
```

Dette eksempel kompilerer et COBOL-program fra `SOURCE`-biblioteket, genererer et midlertidigt objektmodul ( `&&LOADSET` ) og gemmer det færdige load-module i `USER.PROJECT.LOAD`.

Tilsvarende gælder for PL/I, hvor compileren **IBM Enterprise PL/I** anvendes, ofte via **PGM=IBMZPLI** i JCL.

# Eksempel på datasetoprettelse

Datasæt bruges fortsat som grundlæggende lagringsstruktur for kildekode og build-artifakter. De oprettes via TSO-kommandoer eller automatiseres gennem scripts og pipelines.

**Eksempel – oprettelse af partitioneret dataset (PDS) til kildekode:**

```
ALLOC DSN('USER.PROJECT.SOURCE') NEW CATALOG SPACE(5,5) TRACKS
RECFM(F,B) LRECL(80) BLKSIZE(800) DSORG(PO)
```

Denne kommando opretter et partitioneret dataset (PDS) til PL/I- eller COBOL-kildekode. Parametrene definerer datasetstruktur, blokstørrelse og optimeret pladsallokering.

---

## Run environments

Mainframeapplikationer kan køres i forskellige miljøer afhængigt af deres funktion og kontekst:

- **Batch-miljøer:** Programmer afvikles sekventielt via JCL og behandler store datamængder.
- **CICS-miljøer:** COBOL- og PL/I-programmer integreres i online transaktionssystemer.
- **DB2-miljøer:** Programmer afvikles som databaseklienter via static eller dynamic SQL.
- **TSO-miljøer:** Muliggør interaktive test og debugging direkte i terminalen.

Moderne run environments understøtter desuden containerlignende strukturer, hvor **z/OS Container Extensions (zCX)** tillader, at applikationer afvikles som Linux-containere på mainframehardware.

Dette muliggør, at klassiske programmer kan interagere med moderne mikrotjenester eller API'er.