



# "COBOL 2000"

(Update as of 4/99)

Artur Reimann

**Fujitsu Software Corporation**

Development Tools Group

3055 Orchard Drive

San Jose, CA 95134-2022

Email: [COBOL@adtools.com](mailto:COBOL@adtools.com)

# Contents

<i>"COBOL 2000"?</i>	3
COBOL Standards	3
<i>General</i>	4
Main new feature: Object Orientation	4
<i>COBOL Format</i>	5
<i>Compiler Directing Statements</i>	6
<i>Program Structure and Execution</i>	8
<i>File Handling</i>	9
<i>Data Types and Representation</i>	10
<i>Boolean/Bit Support</i>	12
Usage BIT	12
<i>Table Handling</i>	13
<i>Exception Handling</i>	14
Exception Handling: Concepts	14
Exception-Names	15
Exception Declaratives	15
Exception Handling: New Statements	16
Exception Handling: Intrinsic Functions	16
Exception Handling: Compiler Directives	16
<i>Processing</i>	17
Functions and Prototypes	17
Enhanced Inter-Program Communication	17
Arithmetic Expressions as Subscripts	18
Enhanced Date Support	18
Procedural Statements	19

# "COBOL 2000"?

What is COBOL 2000?: It's the most powerful, most flexible, most state-of-the-art COBOL language you've ever seen. That's the good news. The bad news is: It should have been COBOL 97!

To make it clear: We're talking of a *language standard*, not of a *compiler*! And not about the *existing* COBOL language standard, which was established in 1985 and is therefore (informally) called the *COBOL '85 standard*, but of the future COBOL standard, which is now expected to be in its final shape by 2000, when the "Final Committee Draft" is to be published and reviewed. (The official ISO publication date may actually be in 2002.) The original target was 1997, but this was not achievable without sacrificing quality and usability of the new language.

Highlight of the new standard is certainly the support of *object-oriented programming*. But beyond this, the new COBOL standard will include a large number of other enhancements, like support of boolean and bit data, native binary and floating point data, generalized exception handling, an enhanced CALL mechanism, improved table handling, automatic and based data, user-defined functions, standardized compiler directives and many, many more.

Fujitsu actively supports the work on the future standard through membership in the COBOL standards committee J4. The Fujitsu representative is Artur Reimann, who was involved with COBOL as early as 1961, and who was already a member of the COBOL standards committee in the 80's, which created the current COBOL '85 standard.

This documentation gives a schematic overview of the new language elements. Many of the new syntactical elements are self-explanatory. If you are interested in details, contact the chairman of the J4 committee (doncobol@mediaone.net).

## **COBOL Standards**

COBOL standards have existed as early as 1968. This does not contradict to the fact that COBOL was first introduced in 1960, because the early publications did not have the status of a standard. After "COBOL 68", the COBOL standard was revised twice, as "COBOL 74" and as "COBOL 85".

- Existing Standards
  - International Standard: ISO/IEC 1989:1985
  - USA Standard: ANS X3.23-1985 (+ Intrinsic Functions and Corrections amendments)
  - National Standards
- Future Standard ("COBOL 2000")
  - Defined by US committee J4 for ANSI and ISO
  - Ad hoc group for Object Orientation
- International coordination by ISO SC22 Working Group 4
- Tentative milestones (as of 4/99):
  - 9/1996            1st Committee Draft (CD) (4 months international review)
  - 7/1998            10 weeks internal review in J4 and WG4
  - 6/2000            Final CD (4 months international review)
  - 9/2001            Draft International Standard (2 months international review)
  - 4/2002            International Standard to ISO

# General

One goal of the revision is to improve application portability by eliminating the large number of levels and modules, which had led to the situation, that every COBOL compiler could "pick and choose" a different subset of the standard. The new standard will not have any subsets, except for the Communication Module.

- No subdivision in levels and modules, except Communication Module
- Communication Module is
  - the only "optional module"
  - an "obsolete element"
- COBOL 85 "Obsolete Elements" are removed
- New Category "archaic":
  - Language element no longer recommended but kept in standard.

## ***Main new feature: Object Orientation***

The most significant feature of the new language is the explicit support of "Object Oriented Programming" (OOP). This new programming paradigm supplements the powerful capabilities provided by the "classical" COBOL. These are the typical elements of OOP, which are all supported by the new COBOL language:

- Classes, Class Objects ('Factories'), Methods
- Explicit and implicit invocation
- Inheritance: class inheritance and interface inheritance
- Multiple inheritance
- Properties
- Parameterized Classes
- Polymorphism
- COBOL Base Class

Details of the elements of Object Orientation in COBOL 2000 are given in a separate document.

# COBOL Format

When COBOL was "invented" at the end of the fifties, punch cards were the typical input medium of the computers of that time. Consequently, the COBOL input format was punch card oriented, one of the reasons why COBOL has the image of an old-fashioned, archaic language.

Support of "multi-octet character sets" (see below) made it necessary to introduce a new free format not bound to the 80-character limit. Of course, this new free format is also much more appropriate for modern input devices, editors, preprocessors etc., and consistent with most other programming languages.

- Free Format
  - Option for 'column' independent format
  - Fixed format (as in current standard) is the default
  - No sequence number area
  - No separate area for comment and continuation characters
  - Literal continuation: "- at end of line to be continued  
"first part of literal... "-  
"continuation of literal"
  - Comments start with \*> (comment line or in-line)
- Inline comments ( \*> ... ) in free format as well as in fixed format
- Concatenation of literals with & operator

# Compiler Directing Statements

Compiler directing statements are not new in COBOL. The COPY statement is as old as the COBOL language itself, and still has a potential for improvement. New is the inclusion of "compiler directives", which in the past were considered in the responsibility of the implementor. Some of these directives are, however, common to most implementations, and making them part of the standard will enhance portability and compatibility of applications.

- Compiler Directives

- General Format

>>compiler-directive

- Convention for CALL mechanism

>> CALL-CONVENTION { COBOL  
[ *call-convention-name1* ] }

- Flagging incompatibilities with Standard '85

>> FLAG-85 { ALL  
[ DE-EDITING  
DIVIDE  
FUNCTION-ARGUMENT  
MOVE  
NOT-SIZE  
NUMVAL  
SET  
STANDARD-1  
STANDARD-2  
ZERO-LENGTH ] } { ON }  
[ OFF ] }

- Activating/deactivating compiler listing

>> LISTING { ON }  
[ OFF ] }

- New page

>> PAGE [ *comment-text* ]

- Propagating exception conditions to calling program

>> PROPAGATE { ON }  
[ OFF ] }

- Specifying COBOL format

>> SOURCE FORMAT IS { FIXED }  
[ FREE ] }

- Activating/deactivating exception handling

```
>>TURN exception-name [ file-name ] ... CHECKING { ON [ WITH LOCATION ] }
{ OFF }
```

– Implementor defined

```
>>IMP ...
```

- Conditional Compilation

– Defining compilation variables

```
>>DEFINE compilation-variable-name AS { expression }
{ literal } { OVERRIDE }
{ PARAMETER }
{ OFF }
```

– IF directive

```
>>IF ...
```

```
    source-lines
```

```
[ >>ELSE ...
```

```
    source-lines ]
```

```
>>END-IF
```

– EVALUATE directive

```
>>EVALUATE [ TRUE ] ...
```

```
{ >>WHEN expression-2 { THROUGH } expression-3 }
```

```
    source-lines ...
```

```
[ >>WHEN OTHER
```

```
    source-lines ]
```

```
>>END-EVALUATE
```

- COPY/REPLACE extensions

– Nested COPY (without REPLACING only)

```
* library text may contain COPY
```

– Multiple REPLACE

```
REPLACE ALSO ...
```

```
REPLACE [ LAST ] OFF
```

– COPY REPLACING and REPLACE for prefix/suffix

```
COPY ... REPLACING { LEADING } ==partial-word-1== BY ==partial-word-2==
```

```
REPLACE ALSO ...
```

```
REPLACE [ LAST ] OFF
```



# File Handling

Although data base technologies have largely reduced the importance of "classical" COBOL file handling, there have been still requirements for extensions and improvements, which are met by the revision.

- Dynamic allocation of file name

SELECT ... ASSIGN ... USING *data-name*

– Contents of *data-name* at OPEN time defines external file name

- Reverse processing:

READ { NEXT  
PREVIOUS } *RECORD etc.*

START *file-name* { FIRST  
KEY relational data-name *etc.*  
LAST }

- 'File Sharing and Record Locking' mechanism:

SELECT ...

[ SHARING WITH { ALL OTHER  
NO OTHER  
READ ONLY } ]

[ LOCK MODE IS { MANUAL  
AUTOMATIC } [ WITH LOCK ON [ MULTIPLE ] { RECORD  
RECORDS } ] ]

OPEN { INPUT  
OUTPUT  
I-O  
EXTEND } [ SHARING WITH { ALL OTHER  
NO OTHER  
READ ONLY } ] [ RETRY { *exp TIMES*  
*exp SECONDS*  
FOREVER } ]

READ *file-name etc.*

[ ADVANCING ON LOCK  
IGNORING LOCK  
*retry-phrase* ]

[ WITH [ NO ] LOCK ]

[ AT END *etc.* ]

WRITE | REWRITE ... [ *retry-phrase* ] [ WITH [ NO ] LOCK ]

UNLOCK { *file-name* [ RECORDS ] } ...

- Improved Report Writer (no longer optional)
- Screen Section and Accept/Display for screen files

Details: See draft standard.

# Data Types and Representation

Though the COBOL language can be considered a "pioneer" for a modern, problem-oriented description of data (e. g. the level-number concept), it still lacks a number of important elements, especially for interaction with data base systems and other languages. Binary and floating-point data, pointers, automatic and based data, boolean and bit data (see below), and user-defined types will improve language usability.

- Maximum for numeric data is 31 digits (instead 18)
- True binary data types (no picture needed)

$$[\text{USAGE IS}] \left\{ \begin{array}{l} \text{BINARY-CHAR} \\ \text{BINARY-SHORT} \\ \text{BINARY-LONG} \\ \text{BINARY-DOUBLE} \end{array} \right\} \left[ \begin{array}{l} \text{SIGNED} \\ \text{UNSIGNED} \end{array} \right]$$

- Floating point data

$$[\text{USAGE IS}] \left\{ \begin{array}{l} \text{FLOAT-SHORT} \\ \text{FLOAT-LONG} \\ \text{FLOAT-EXTENDED} \end{array} \right\}$$

– External presentation, e.g. PIC 9(8)E999

- Data items of any length (alphanumeric, national, boolean items in linkage section only)

ANY LENGTH clause

- Based items, pointers, and program pointers

```
01 record BASED [ON pointer-name ]
[USAGE IS] POINTER [TO type-name ]
SET pointer-name TO ADDRESS OF record
SET ADDRESS OF based-item TO pointer-name
SET pointer-name { UP | DOWN } BY arithm-expression
[USAGE IS] PROGRAM-POINTER [ TO program-prototype-name ]
```

- Constants

$$01 \textit{constant-name} \textit{CONSTANT} [\textit{IS GLOBAL}] \left\{ \begin{array}{l} \textit{AS} \left\{ \begin{array}{l} \textit{expression} \\ \textit{literal} \\ \textit{LENGTH OF data-name} \end{array} \right\} \\ \textit{FROM compilation-variable-name} \end{array} \right\}$$

- LOCAL-STORAGE SECTION

– Data items initialized with each program invocation ('automatic' data)

- SAME AS clause

```
01 record-name-2 SAME AS record-name-1
record-name-2 has same structure as record-name-1
```

- User defined types, strong typing

– Defined by clause '01 *type-name* ... IS TYPDEF [ STRONG ]'

- Referenced by 'TYPE *type-name*'
- Improved currency symbol support
  - Currency symbols consisting of multiple characters
  - Arbitrary characters allowedDefined (in Special-Names) by  

```
[CURRENCY SIGN IS literal-1 [WITH PICTURE SYMBOL literal-2]]
```
- Data validation ('Validate')
  - Specification using picture characters and additional clauses:
    - ALLOW clause
    - CLASS clause
    - DEFAULT clause
    - DESTINATION clause
    - ERROR clause
    - INVALID WHEN clause
    - PRESENT WHEN clause
    - VARYING clause
  - Activated by `VALIDATE` statement

# Boolean/Bit Support

There is probably no other example of misuse of the COBOL language like the various, sometimes ridiculous, attempts to handle bit strings and control blocks including bit data with existing COBOL. None of these "solutions" were portable, and, of course, the "clean" solution to use an assembler-language subroutine was in no way more portable. COBOL 2000 will provide the necessary support.

- Boolean data:
  - Defined with picture character '1'
  - Representation
    - \* as character: `USAGE DISPLAY` (Default)
    - \* as bits: `USAGE BIT` (see below)
- Boolean literals:
  - e. g. `B"01101000111001"`
- Boolean Expressions:
  - Operators `B-NOT`, `B-AND`, `B-OR`, `B-XOR`
  - Parentheses permitted
- Comparison operations:
  - equal/not equal only
- Class condition 'Boolean' (for `USAGE DISPLAY` only)
- Boolean condition
  - Boolean expression of length 1 used as condition
  - Permitted in:  
`COMPUTE`, `EVALUATE`, `INITIALIZE`, `INSPECT`, `MOVE`
- Conversion functions `INTEGER-OF-BOOLEAN` and `BOOLEAN-OF-INTEGER`

## *Usage BIT*

- Format:  
`USAGE IS BIT [ ALIGNED ]`
- Must be defined as boolean data item, i. e. with picture character 1.
- Specifies that boolean positions are represented by bits.
- Justification of bit item:
  - On first bit position within "natural character boundary" (byte):
    - \* the first data element of a group item
    - \* the first data element within a record following a non-bit data element
    - \* data element defined with `ALIGNED`
  - Otherwise in first bit position following the preceding bit data element

# Table Handling

The new language will provide an easier way for table initialization and a straight-forward table sort using the SORT verb.

- Direct table initialization

– sequence of VALUE specifications

```
{VALUE} [IS] {{literal} ... FROM ( {subscript-1} ... )  
{VALUES} [ARE] [ TO ( {subscript-2} ... ) ] } ...
```

- Table sort using SORT verb

```
SORT data-name-1 { ON {ASCENDING}  
{DESCENDING} } KEY [data-name-2] ... } ...  
[WITH DUPLICATES IN ORDER]  
[collating-sequence phrase]
```

# Exception Handling

Handling of exception and error conditions has been done in COBOL in a very inconsistent and incomplete way. The requirement to provide a consistent and comprehensive exception handling capability has been around for a long time. As a result, the revision includes a "generalized exception handling" feature, which covers all parts of the language.

- Generalized Declaratives
- Functions provide information on reason and exception location
- Mechanism for activating and deactivating (`TURN` compiler directive)

## ***Exception Handling: Concepts***

- Exception Condition
  - Error or exception condition during program execution
- Exception-Name
  - name of a specific type of exception
  - three levels:
    - \* level 1: `EC-ALL`; includes all exception conditions
    - \* level 2: main groups of exception conditions
    - \* level 3: names of individual exception conditions
- Exception Status Indicator
  - a conceptual entity defined for each exception-name
  - has one of two possible states: 'set' or 'cleared'

## Exception-Names

- level 1: EC-ALL
- level 2:

EC-ARGUMENT	argument error
EC-BOUND	boundary violation
EC-DATA	data exception
EC-FLOW	execution control flow violation
EC-I-O	input-output exception
EC-IMP	implementor-defined exception
EC-LOCALE	locale related exception
EC-OO	OO exception
EC-ORDER	ordering exception
EC-OVERFLOW	STRING/UNSTRING overflow condition
EC-PROGRAM	inter-program communication exception
EC-RAISING	EXIT ... RAISING exception
EC-RANGE	range exception
EC-REPORT	Report Writer exception
EC-SCREEN	screen handling exception
EC-SIZE	size error exception
EC-SORT-MERGE	Sort or Merge exception
EC-STORAGE	storage allocation exception
EC-VALIDATE	VALIDATE exception
EC-USER	user-defined exception condition
- level 3 example:

EC-BOUND-SUBSCRIPT	subscript out of bounds
...	...

## Exception Declaratives

- New format of USE statement

```
USE AFTER EXCEPTION { exception-name-1 } ...
```

  - gets control when condition occurs and condition handling is enabled (see >>TURN directive)
- exception-name-1
  - one of the defined exception-names
- Old-style USE statement has precedence
- Level 3 has precedence over level 2, level 2 over level 1:
  - e. g. when USE statements for EC-I-O and EC-I-O-LOGIC-ERROR exist, the latter wins
- Type of condition governs processing at end of USE procedure:
  - "fatal" - program is aborted, except when RESUME statement is specified
  - "non-fatal" - dependent on statement that raised the exception



# Processing

If COBOL wants to become a language of choice on PC's and workstations, it needs to be more flexible in communicating with other languages and subsystems. As a result, an improved CALL mechanism, use of prototypes, and user-defined functions will be added to the language.

Improved flexibility, as well as more consistency, is also a motivation for permitting arbitrary arithmetic expressions as subscripts.

## Functions and Prototypes

- Program prototypes

PROGRAM-ID *program-prototype-name* IS PROTOTYPE ...

- In Repository paragraph:

PROGRAM *program-prototype-name* [ AS *literal-1* ]

- Invocation through new format of CALL statement:

CALL [ { *literal*  
*identifier* } AS ] *program-prototype-name* ... [ USING ... ]

- Prototypes allow compile-time testing and coercion of arguments

- User defined functions

FUNCTION-ID. *function-name*

PROCEDURE DIVISION ... USING ... RETURNING

- Function prototypes

FUNCTION-ID *function-prototype-name* IS PROTOTYPE ...

- In Repository paragraph:

FUNCTION *function-prototype-name* [ AS *literal-1* ]

- Invocation:

*function-prototype-name* ( *argument* ... )

- Intrinsic functions without keyword FUNCTION

- In Repository paragraph:

FUNCTION { *function-name* | ALL } INTRINSIC

## Enhanced Inter-Program Communication

- Returning a result:

RETURNING phrase in procedure division header and in CALL statement

- Enhancements with new prototype-format of CALL statement (also for functions and methods)

- Permitted as arguments, when passed BY CONTENT:

- \* arithmetic and boolean expressions
- \* literals
- \* functions

- \* inline method invocations (see OO documentation)
- \* addresses
- \* any other predefined identifiers
- Passing the value of an argument:  
`CALL ... USING ... BY VALUE ...`
- Permitted as arguments, when passed BY VALUE:
  - \* numeric identifiers
  - \* object references
  - \* pointers
  - \* numeric functions
- Optional parameters:  
`PROCEDURE DIVISION USING ... [ OPTIONAL ] ...`  
`CALL ... USING ... OMITTED ...`
  - \* Class test for OMITTED
- Call convention to be applied when this program is called:  
`OPTIONS. CALL-CONVENTION IS call-convention-name`
- Call convention to be applied for calling other programs:  

$$\gg \text{CALL-CONVENTION } \left\{ \begin{array}{l} \text{COBOL} \\ \text{call-convention-name} \end{array} \right\}$$
- AS phrase in PROGRAM-ID, FUNCTION-ID, EXTERNAL, CLASS-ID, INTERFACE-ID, METHOD-ID allows defining externalized names.

## ***Arithmetic Expressions as Subscripts***

In current COBOL, you can use arithmetic expressions in reference modification, but not in subscripting. This is a strange inconsistency, which can only be explained historically. This restriction will be removed.

- Arithmetic expressions as subscript

## ***Enhanced Date Support***

The "year-2000 problem", although not a primary objective of the COBOL revision - the new standard simply is not available in time to help with this problem - will not be ignored. There will be new formats of the ACCEPT statement providing a 4-digit year, and there will be also new intrinsic functions for conversion between dates containing 2-digit years and dates containing 4-digit years.

- ACCEPT enhancements

$$\text{ACCEPT } \textit{identifier} \text{ FROM } \left\{ \begin{array}{l} \text{DATE } [\text{YYYYMMDD}] \\ \text{DAY } [\text{YYYYDDD}] \end{array} \right\}$$

- Date functions

`FUNCTION YEAR-TO-YYYY ( arg-1 [ arg-2 ] )`
`FUNCTION DATE-TO-YYYYMMDD ( arg-1 [ arg-2 ] )`

FUNCTION DAY-TO-YYYYDDD ( *arg-1* [ *arg-2* ] )

## ***Procedural Statements***

Beyond the new facilities described above, there will be several simplifications and enhancements of well-known COBOL statements, as well as new statements for dynamically acquiring and releasing storage areas.

- INITIALIZE enhancements

INITIALIZE ... [ WITH FILLER ] ...

- FILLER items are initialized

INITIALIZE ... [ TO VALUE ]

- Initialization according to VALUE specification

- STRING simplification

- DELIMITED BY SIZE may be omitted

- SET TO FALSE

88 *condition-name* VALUE ... WHEN SET TO FALSE VALUE ...

SET *condition-name* TO FALSE

- ALLOCATE and FREE

ALLOCATE { *expr* CHARACTERS } [ INITIALIZED ] [ RETURNING *pointer-name* ]  
          { *based-item* }

FREE { *pointer-name* } ...

- EXIT options

EXIT PERFORM [ CYCLE ]

- \* branch to end or beginning of loop

EXIT { PARAGRAPH }  
          { SECTION }

- STOP RUN extension, GOBACK

STOP RUN WITH { ERROR } STATUS ...  
                  { NORMAL }

GOBACK