



# MANUAL TRANSMITTAL

Department of the Treasury  
Internal Revenue Service

2.5.3

JULY 10, 2020

## EFFECTIVE DATE

(07-10-2020)

## PURPOSE

- (1) This transmits revised IRM 2.5.3, Systems Development, Programming and Source Code Standards.

## MATERIAL CHANGES

- (1) Changed from Chief Information Officer (CIO) Gina Garza to Acting Chief Information Officer (ACIO), Nancy Sieger
- (2) Manual Transmittal - Added Background section
- (3) 2.5.3.1, Added section 1, Program Scope and Objective
- (4) 2.5.3.1.1, Added subsection 1 - Background
- (5) 2.5.3.1.2, Added subsection 1 - Authority
- (6) 2.5.3.1.3, Added subsection Roles and Responsibilities (1 - 10)
- (7) 2.5.3.1.4, Added Application Development Responsibilities
- (8) 2.5.3.1.5, Added subsection 1 - Program Management and Review (1) - (3)
- (9) 2.5.3.1.6, Added subsection 1 - Program Controls
- (10) 2.5.3.1.7, Added subsection 1 - Acronyms and Terms
- (11) 2.5.3.2, Added subsection 1 - Related Resources
- (12) 2.5.3.4, Removed subsection1 Purpose from under Section 1 Introduction and moved to section 1 Program Scope and Objective
- (13) 2.5.3.4.2, Definitions - Removed title Definitions, and moved Exhibit information under it
- (14) 2.5.3.5, Federal Government Applications Standards and Guidance
- (15) 2.5.3.5, Introduction - Added line (8) The subsection, **Assembler Programming** addresses topics specific to Assembler Programming, removed the definition, and changed grammar
- (16) 2.5.3.5.1, Added Application Security Control Frameworks
- (17) 2.5.3.5.1.1, Added Application Security Controls
- (18) 2.5.3.5.4 (1), IRM 2.5.2, Software System Testing is obsolete - Changed to IRM 2.127.2 , Information Technology Testing Process and Procedures
- (19) 2.5.3.5.4 (1), Changed last bullet from Java Programming Language, Sun Microsystems, Inc. to Oracle Corporation
- (20) 2.5.3.5.5, Waivers - Added reference IRM 2.5.14.3, System Development Quality Assurance - Waiver Justification Procedure
- (21) 2.5.3.6(1), General Programming - Changed sentence according to IRS Writing Style guide

- (22) 2.5.3.6, Basic Principles - Added (5) Always code static database software DB2 cursors in the working storage section of your program, Do not code them in procedure division of the program.
- (23) 2.5.3.6, Basic Principles Added (6) For Customer Information Control System (CICS) use XCTL (transfer control) instead of LINK to switch between programs, if you need to return to the calling program otherwise; use LINK.
- (24) 2.5.3.6.4.1 (1), Documenting Code - Removed repeated sentences
- (25) 2.5.4.6.4.5, Corrected grammar for last three bullets
- (26) 2.5.3.6.7.1.2 (7), Defining Data Standards - Removed paragraph 7
- (27) 2.5.3.6.7.2, Tape Interface - Modified grammar for third bullet
- (28) 2.5.3.7.3, Changed title from Structured Programming to COBOL Structured Programming
- (29) 2.5.3.7.3, COBOL Structured Programming - Converted table to meet 508 compliance guidelines
- (30) 2.5.3.7.4, Added COBOL Compile Run-Time Warning Messages
- (31) 2.5.3.7, Changed title from General Programming to COBOL Programming Standards
- (32) 2.5.3.7.4(3) a), Added Identification Division
  - 1. Program Name
  - 2. Author Name
  - 3. Installation
  - 4. Date Written
  - 5. Date Completed
  - 6. Security Information
- (33) 2.5.3.7.3, Modified paragraphs 2 and 8, and removed 5
- (34) 2.5.3.7.3, Structured Programming - Removed paragraphs 5 - 9
- (35) 2.5.3.7.4(3)(b), COBOL Programming Standards - Added Environment Division and sub-list :
  - 1. Configuration Section: a Source Computer, b. Object Computer
  - 2. Input/Output section: a) File-Control, b) I/O Control
- (36) 2.5.3.7.4 (3) ( c), COBOL Programming Standards - Added Data Division
- (37) 2.5.3.7.4 (3) (d), COBOL Programming Standards, Procedure Division modified 14- 11
- (38) 2.5.3.8, Added Assemble Programming Language
- (39) 2.5.3.8.1, Added ALC overview
- (40) 2.5.3.8.2, Added ALC Basic Principles
- (41) 2.5.3.8.3, Added ALC Program Comments and Documentation
- (42) 2.5.3.8.4, Added IRS Assembler Language Coding Conventions
- (43) 2.5.3.8.4.1, Added ALC Defining Constants and Storage
- (44) 2.5.3.8.5, Added IRS Standard Assembler Macros
- (45) 2.5.3.9.2.1, C++ - Converted table to meet 508 compliance guidelines

- (46) 2.5.3.9.6, C++ File Prologs -converted table to meet 508 compliance guidelines
- (47) 2.5.3.6.9.7, Converted table header to meet 508 compliance guidelines
- (48) 2.5.3.10.5, Added IRS Standard Assembler Macros
- (49) 2.5.3.9, Combined all Java Programming content into one section
- (50) 2.5.3.9.1, Java Programming Overview
- (51) 2.5.3.9.2, Added Java Programming Programs
- (52) 2.5.3.9.2.1, Added Java Programming Source File Structure
- (53) 2.5.3.9.2.1.1, Added Java Programming Beginning Comments
- (54) 2.5.3.9.2.1.2, Added Java Programming Package and Import Statements
- (55) 2.5.3.9.2.2, Added Java Programming Naming Conventions
- (56) 2.5.3.9.2.2.1, Added Java Programming Capitalization Conventions
- (57) 2.5.3.9.2.2.2, Added Java Programming Type Member Names
- (58) 2.5.3.9.3, Added Java Programming Layout Conventions
- (59) 2.5.3.9.3.1, Added Java Programming Wrapping Lines
- (60) 2.5.3.9.4, Added Java Programming Commenting Conventions
- (61) 2.5.3.9.4.1, Added Java Programming Single Line Comments
- (62) 2.5.3.9.4.2, Added Java Programming Block Comments
- (63) 2.5.3.9.5, Added Java Programming Class Design
- (64) 2.5.3.9.5.1, Added Java Programming Packages
- (65) 2.5.3.9.5.2, Added Java Programming Interfaces
- (66) 2.5.3.9.5.3, Added Java Programming Classes
- (67) 2.5.3.9.5.3.1, Added Java Programming Abstract Classes
- (68) 2.5.3.9.5.3.2, Added Java Programming Sealed Classes
- (69) 2.5.3.9.5.3.3, Added Java Programming Inner Classes
- (70) 2.5.3.9.5.3.4, Added Java Programming Immutable Classes
- (71) 2.5.3.9.5.3.5, Added Java Programming Objects
- (72) 2.5.3.9.5.3.6, Added Java Programming Class Access Modifiers
- (73) 2.5.3.9.5.3.7, Added Java Programming Fields
- (74) 2.5.3.9.5.3.8, Added Java Programming Types
- (75) 2.5.3.9.5.3.8.1, Added Java Programming Autoboxing and Unboxing Types
- (76) 2.5.3.9.5.3.8.2, Added Java Programming Enumeration Types

- (77) 2.5.3.9.4, Added Java Programming Commenting Conventions
- (78) 2.5.3.9.4.1, Added Java Programming Single Line Comments
- (79) 2.5.3.9.4.2, Added Java Programming Block Comments
- (80) 2.5.3.9.5, Added Java Programming Class Design
- (81) 2.5.3.9.5.1, Added Java Programming Packages
- (82) 2.5.3.9.5.2, Added Java Programming Interfaces
- (83) 2.5.3.9.5.3, Added Java Programming Classes
- (84) 2.5.3.9.5.3.1, Added Java Programming **Abstract** Classes
- (85) 2.5.3.9.5.3.2, Added Java Programming **Sealed** Classes
- (86) 2.5.3.9.5.3.3, Added Java Programming **Inner** Classes
- (87) 2.5.3.9.5.3.4, Added Java Programming **Immutable** Classes
- (88) 2.5.3.9.5.3.5, Added Java Programming Objects
- (89) 2.5.3.9.5.3.6, Added Java Programming Class Access Modifiers
- (90) 2.5.3.9.5.3.7, Added Java Programming Fields
- (91) 2.5.3.9.5.3.8, Added Java Programming Types
- (92) 2.5.3.9.5.3.8.1, Added Java Programming Autoboxing and Unboxing Types
- (93) 2.5.3.9.5.3.8.2, Added Java Programming Enumerations Types
- (94) 2.5.3.9.5.3.8.3, Added Java Programming Nullable Types
- (95) 2.5.3.9.5.3.8.4, Added Java Programming Nested Classes
- (96) 2.5.3.9.5.3.8.5, Added Java Programming Numeric Types
- (97) 2.5.3.9.5.3.8.6, Added Java Programming, Generics
- (98) 2.5.3.9.6, Added Java Programming Statements
- (99) 2.5.3.9.6.1, Added Java Programming Variable Declaration
- (100) 2.5.3.9.6.2, Added Java Programming Expressions
- (101) 2.5.3.9.6.3, Added Java Programming Conditional Statements
- (102) 2.5.3.9.6.4, Added Java Programming Iteration Statement
- (103) 2.5.3.9.6.5, Added Java Programming Empty Statement
- (104) 2.5.3.9.6.6, Added Java Programming Assertion Statement
- (105) 2.5.3.9.7, Added Java Programming Expression
- (106) 2.5.3.9.7.1, Lambda Expression
- (107) 2.5.3.9.8, Added Java Programming Operators

- (108) 2.5.3.9.9, Added Java Programming Member Design
- (109) 2.5.3.9.9.1, Added Java Programming Member Overloading
- (110) 2.5.3.9.9.2, Added Java Programming Constructor Design
- (111) 2.5.3.9.9.3, Added Java Programming Finalizer Design
- (112) 2.5.3.9.9.4, Added Java Programming Field Design
- (113) 2.5.3.9.9.5, Added Java Programming Property Design
- (114) 2.5.3.9.9.5.1, Added Java Programming Abstract Properties
- (115) 2.5.3.9.9.5.2, Added Java Programming Constants
- (116) 2.5.3.9.9.6, Added Java Programming Parameter Design
- (117) 2.5.3.9.9.6.1, Added Java Programming Variable Length Parameter
- (118) 2.5.3.9.9.6.2, Added Java Programming Event Design
- (119) 2.5.3.9.9.7, Added Java Programming Methods
- (120) 2.5.3.9.9.8, Added Java Programming Language Guidelines
- (121) 2.5.3.9.9.8.1, Added Java Programming Arrays
- (122) 2.5.3.9.9.9, Added Java Programming Exceptions
- (123) 2.5.3.9.9.9.1, Added Java Programming Catching and Handling Exceptions
- (124) 2.5.3.9.9.9.2, Added Java Programming Throwing Exceptions
- (125) 2.5.3.9.9.9.3, Added Java Programming Unchecked Exceptions Best Practices
- (126) 2.5.3.9.9.10, Added Java Programming Concurrency
- (127) 2.5.3.9.9.10.1, Added Java Programming Threads
- (128) 2.5.3.9.9.10.2, Added Java Programming High-Level Concurrency
- (129) 2.5.3.9.9.11, Added Java Programming Native Code Interoperability
- (130) 2.5.3.9.9.12, Added Java Programming Design for Extensibility
- (131) 2.5.3.9.9.12.1, Added Java Programming Unsealed Classes
- (132) 2.5.3.9.9.12.2, Added Java Programming Protected Members
- (133) 2.5.3.9.9.12.3, Added Java Programming Events and Callbacks
- (134) 2.5.3.9.9.12.4, Added Java Programming Virtual Members
- (135) 2.5.3.9.9.12.5, Added Java Programming Abstractions
- (136) 2.5.3.9.9.12.6, Added Java Programming Base Classes for Implementing Abstractions
- (137) 2.5.3.9.9.12.7, Added Java Programming Sealing
- (138) 2.5.3.9.9.13, Added Java Programming Secure Coding Guidelines

- (139) 2.5.3.9.9.13.1, Added Java Programming Fundamentals
- (140) 2.5.3.9.9.13.2, Added Java Programming Denial of Service
- (141) 2.5.3.9.9.13.3, Added Java Programming Confidential Information
- (142) 2.5.3.9.9.13.4, Added Java Programming Input Validation and Data Sanitization
- (143) 2.5.3.9.9.13.5, Added Java Programming Injection and Inclusion
- (144) 2.5.3.9.9.13.6, Added Java Programming Accessibility and Extensibility
- (145) 2.5.3.9.9.13.7, Added Java Programming Serialization and Deserialization
- (146) 2.5.3.9.9.13.8, Added Java Programming Access Control
- (147) 2.5.3.9.9.13.9, Added Java Programming Defensive User of the Java Native Interface (JNI)
- (148) 2.5.3.10, Introduction Added line (g) Assembler Language Coding (ALC)
- (149) 2.5.3.13.6.3 (1), Conventional Constants - Added missing attribute to table
- (150) Changed shall to must for requirements throughout this IRM
- (151) All Tables throughout IRM , Included a Title Heading
- (152) Exhibit 2.5.3.-1, Added Java Programming Example of Wrapping Lines
- (153) Exhibit 2.5.3-2, Added Java Programming Example of Objects
- (154) Exhibit 2.5.3-3, Added Java Programming Example Object Test Results
- (155) Exhibit 2.5.3-4, Added Java Programming Example of “instanceof”
- (156) Exhibit 2.5.3-5, Added Java Programming Example of Subclasses
- (157) Exhibit 2.5.3-6, Added Java Programming Example of Enumeration Type
- (158) Exhibit 2.5.3-7, Added Java Programming Nested Classes
- (159) Exhibit 2.5.3-8, Added Java Programming Switch Example
- (160) Exhibit 2.5.3-9, Added Java Programming Design Example
- (161) Exhibit 2.5.3-10, Added Java Programming Constructor Example
- (162) Exhibit 2.5.3.9, Added Java Programming Abstract Example
- (163) Exhibit 2.5.3-12, Added Java Programming Event Design
- (164) Exhibit 2.5.3-13, Added Java Programming Thread Example
- (165) Exhibit 2.5.3-14, Added Java Programming Examples of Single Words used for Capitalization Purposes
- (166) Exhibit 2.5.3-15, Added COBOL Examples
- (167) Exhibit 2.5.3.15, Added C Language Source Code Template
- (168) Exhibit 2.5.3-16, Added C Language Header File Template

- (169) Exhibit 2.5.3-17, Added Acronyms and Terms
- (170) Exhibit 2.5.3-18, Added Terms and Descriptions
- (171) Exhibit 2.5.3-19, Assembler Language Code (ALC) Standards and References
- (172) References, Added IBM High Level Assembler for z/OS Language Reference V1R6
- (173) References, Added IBM Assembler System Standards, Chapter 1- 8 (IRS-defined)
- (174) References, Added Java Assembly Package title and hyperlink
- (175) References, Added Java Programming Package and Import Statements title and hyperlink
- (176) References, Added Java Programming Input Validation and Data Sanitation title and hyperlink
- (177) References, Added Catching and Handling Exceptions
- (178) Testing and Debugging (1), Removed obsolete IRM 2.5.2 and replaced with IRM 2.127.2 , Information Technology Testing Process and Procedures
- (179) References, Added Government Accountability Office (GAO) report, June 2018, GAO-18-298 Investments' Performance and Risks
- (180) References, Added IRM 10.8.1 Information Technology (IT) Security, Policy Guidance
- (181) References, Added IRM 10.8.6 Information Technology (IT) Security, Application Security and Development
- (182) 2.5.3.9.3.1 (3), General Programming - Changed line Begin to insert comment lines in these specified areas to Begin to insert comment lines, statements, and code in these areas
- (183) 2.5.3.9.3.1 (3), General Programming - Reorganize section (a -e) to the following:
  - a. a. IDENTIFICATION DIVISION - Added new sub-bullet list
  - b. b. DATA DIVISION: - Moved to line c, and replaced with ENVIRONMENT DIVISION with subsection Configuration Section
  - c. c. WORKING STORAGE SECTION - Replaced with. DATA DIVISION, and moved WORKING-STORAGE SECTION under as a bullet
  - d. d. LINKAGE SECTION - Moved under c. DATA DIVISION
  - e. e. PROCEDURE DIVISION - Moved to line d.
  - f. Removed line f.- Any section within the PROCEDURE DIVISION -
  - g. Removed line g. - Any paragraph /section that represents a Structure Chart module with the PROCEDURE DIVISION

#### **EFFECT ON OTHER DOCUMENTS**

IRM 2.5.3, dated 3-1-2007, is superseded.

## **AUDIENCE**

The audience intended for this transmittal is personnel responsible for engineering, developing, or maintaining Agency software systems identified in the Enterprise Architecture. This engineering, development, and maintenance includes work performed by IRS management, Information Technology government employees and contractors.

Nancy Sieger  
Acting Chief Information Officer



2.5.3

Programming and Source Code Standards

## Table of Contents

2.5.3.1 Program Scope and Objectives

2.5.3.1.1 Background

2.5.3.1.2 Authority

2.5.3.1.3 Roles and Responsibilities

2.5.3.1.4 Program Management and Review

2.5.3.1.5 Program Controls

2.5.3.1.6 Acronyms and Terms

2.5.3.1.7 Related Resources

2.5.3.2 Federal Government Application Standards Guidance

2.5.3.2.1 Application Security Control Frameworks

2.5.3.2.1.1 Application Security Controls

2.5.3.2.2 AD Waivers

2.5.3.3 General Programming

2.5.3.3.1 Goals

2.5.3.3.2 Basic Principles

2.5.3.3.3 Design Specifications

2.5.3.3.4 Documenting, Testing, and Debugging Source Code

2.5.3.3.4.1 Documenting Code

2.5.3.3.4.2 Testing and Debugging Code

2.5.3.3.5 Selecting Programming Languages

2.5.3.3.6 Data Controls

2.5.3.3.6.1 Basic Principles of Data Controls

2.5.3.3.6.2 Programming Considerations for Data Controls

2.5.3.3.6.3 Internal Controls

2.5.3.3.6.3.1 Input Controls

2.5.3.3.6.3.2 Processing Controls

2.5.3.3.6.3.3 Output Controls

2.5.3.3.6.4 External Data Controls

2.5.3.3.6.4.1 Control Totals

2.5.3.3.6.4.2 Intra-Run Controls

2.5.3.3.6.5 Including Data Controls

2.5.3.3.7 File Design and Cartridge Interface Formats

2.5.3.3.7.1 File Design Formats

2.5.3.3.7.1.1 Record Format Design

2.5.3.3.7.1.2 Defining Data Fields

- 
- 2.5.3.3.7.1.3 File Design
  - 2.5.3.3.7.2 Tape Interface
  - 2.5.3.3.8 Date Fields
    - 2.5.3.3.8.1 Year
    - 2.5.3.3.8.2 Date
    - 2.5.3.3.8.3 Gregorian Dates
    - 2.5.3.3.8.4 Exceptions
  - 2.5.3.4 COBOL Programming
    - 2.5.3.4.1 COBOL Overview
    - 2.5.3.4.2 COBOL Basic Principles
    - 2.5.3.4.3 COBOL Structured Programming
      - 2.5.3.4.3.1 COBOL Programming Standards
      - 2.5.3.4.3.2 COBOL Identification Division
      - 2.5.3.4.3.3 COBOL Environment Division
      - 2.5.3.4.3.4 COBOL Data Division
      - 2.5.3.4.3.5 COBOL Procedure Division
    - 2.5.3.4.4 COBOL Compile Run-Time Warning Messages
  - 2.5.3.5 C Programming
    - 2.5.3.5.1 C File Naming
    - 2.5.3.5.2 C Source Code Files
      - 2.5.3.5.2.1 C Prologue
      - 2.5.3.5.2.2 C Includes
        - 2.5.3.5.2.2.1 C Header File Organization
        - 2.5.3.5.2.2.2 C Header File Inclusion in the File that defines the Function
        - 2.5.3.5.2.2.3 C Nested Header Files
        - 2.5.3.5.2.2.4 C Header File Names
      - 2.5.3.5.2.3 C Defines and Typedefs
      - 2.5.3.5.2.4 C Global Definitions
      - 2.5.3.5.2.5 C Function Placement
    - 2.5.3.5.3 C Other Files
    - 2.5.3.5.4 C Global Variable Declarations
      - 2.5.3.5.4.1 C Global Variables
      - 2.5.3.5.4.2 C Structure Declaration
      - 2.5.3.5.4.3 C Typedef Declaration
    - 2.5.3.5.5 C Local Variable Declarations
      - 2.5.3.5.5.1 C Local Variable Names
      - 2.5.3.5.5.2 C Typedef Declaration
      - 2.5.3.5.5.3 C Abbreviations for Common Variable
    - 2.5.3.5.6 C Constants

- 
- 2.5.3.5.6.1 C Defining Constants
  - 2.5.3.5.6.2 C Consistency of Constant Definitions
  - 2.5.3.5.6.3 C Conventional Constants
  - 2.5.3.5.6.4 C Enumeration Data
  - 2.5.3.5.6.5 C Symbolic Constants - #define
  - 2.5.3.5.7 C Functions
    - 2.5.3.5.7.1 C Return Values
    - 2.5.3.5.7.2 C Parameter Lists
    - 2.5.3.5.7.3 C Function Body
    - 2.5.3.5.7.4 C Function Prototype
    - 2.5.3.5.7.5 C Function Naming
  - 2.5.3.5.8 C Comments
    - 2.5.3.5.8.1 C Template for File and Header
    - 2.5.3.5.8.2 Function Comments
  - 2.5.3.5.9 C Statements
    - 2.5.3.5.9.1 C Statements per Line
    - 2.5.3.5.9.2 C Single Statement Blocks
    - 2.5.3.5.9.3 C Multiple Statement Blocks
    - 2.5.3.5.9.4 C Levels of Control Structure Nesting
    - 2.5.3.5.9.5 C Goto Statement
    - 2.5.3.5.9.6 C Break Statement
    - 2.5.3.5.9.7 C Null Statement
    - 2.5.3.5.9.8 Conditional Statement
    - 2.5.3.5.9.9 C Exit Statement
    - 2.5.3.5.9.10 C Default Truth Value
    - 2.5.3.5.9.11 C - Added Statements for Debugging
  - 2.5.3.5.10 Operators
  - 2.5.3.5.11 ESQL/C
    - 2.5.3.5.11.1 ESQL/C Database Error Checks
    - 2.5.3.5.11.2 ESQL/C Operations
    - 2.5.3.5.11.3 ESQL/C Performance
    - 2.5.3.5.11.4 SQL Statements
  - 2.5.3.5.12 Whitespace
    - 2.5.3.5.12.1 Vertical Spacing of Conditional Operators on Separate Lines
    - 2.5.3.5.12.2 C Spacing for Parentheses
  - 2.5.3.5.13 C Portability
    - 2.5.3.5.13.1 C Machine-Dependent Code Placement
    - 2.5.3.5.13.2 C Machine-Dependent Code Usage
  - 2.5.3.6 C++ Programming Overview

- 
- 2.5.3.6.1 C++ Scope
  - 2.5.3.6.2 C++ Classes
    - 2.5.3.6.2.1 C++ Class Declaration
    - 2.5.3.6.2.2 C++ Constructors and Destructors
    - 2.5.3.6.2.3 C++ Class Data Initialization
    - 2.5.3.6.2.4 C++ Class Execution
    - 2.5.3.6.2.5 C++ Inheritance
    - 2.5.3.6.2.6 C++ Initialization
      - 2.5.3.6.2.6.1 C++ Initialization of Variables
      - 2.5.3.6.2.6.2 C++ Initialization of Classes
  - 2.5.3.6.3 Variables Scope
  - 2.5.3.6.4 Data Types
  - 2.5.3.6.5 Conditional Constructs
  - 2.5.3.6.6 File Prologs
    - 2.5.3.6.6.1 File Size and Structure
      - 2.5.3.6.6.1.1 File Size
      - 2.5.3.6.6.1.2 File Structure
    - 2.5.3.6.6.2 C++ Name Conventions
      - 2.5.3.6.6.2.1 C++ General Naming Conventions
        - 2.5.3.6.6.2.1.1 C++ Identifiers
        - 2.5.3.6.6.2.1.2 C++ Functions and Parameter
        - 2.5.3.6.6.2.1.3 C++ Constants
  - 2.5.3.6.7 C++ Formatting
    - 2.5.3.6.7.1 C++ Indentation
    - 2.5.3.6.7.2 C++ Spacing
    - 2.5.3.6.7.3 Grouping
    - 2.5.3.6.7.4 Includes
  - 2.5.3.6.8 Functions
    - 2.5.3.6.8.1 Declarations
    - 2.5.3.6.8.2 Function Parameters
    - 2.5.3.6.8.3 Function Invocation, Execution, and Return
  - 2.5.3.6.9 Error Handling
    - 2.5.3.6.9.1 General Error Handling
    - 2.5.3.6.9.2 Throwing Exceptions
    - 2.5.3.6.9.3 Handling Exceptions
  - 2.5.3.6.10 Expressions
    - 2.5.3.6.10.1 Expression Arithmetic
    - 2.5.3.6.10.2 Type Conversions
    - 2.5.3.6.10.3 Pointers in Expressions

- 2.5.3.6.11 Comments
- 2.5.3.6.12 Memory Management
  - 2.5.3.6.12.1 Heap and Stack Memories
  - 2.5.3.6.12.2 Memory Leaks
  - 2.5.3.6.12.3 Buffers Overflows
- 2.5.3.7 Assembler Language Code (ALC) Programming
  - 2.5.3.7.1 Assembler Language Code (ALC) Overview
  - 2.5.3.7.2 Assembler Language Code (ALC) Basic Principles
  - 2.5.3.7.3 Assembler Language Code (ALC) Program Comments and Documentation
  - 2.5.3.7.4 Assembler Language Coding Conventions (ALC)
    - 2.5.3.7.4.1 Assembler Language Code (ALC) Defining Constants and Storage
  - 2.5.3.7.5 Assembler Language Code (ALC) Standard Macros
- 2.5.3.8 Java Programming Language
  - 2.5.3.8.1 Java Programming Overview
  - 2.5.3.8.2 Program Objectives
    - 2.5.3.8.2.1 Source File Structure
      - 2.5.3.8.2.1.1 Beginning Comments
      - 2.5.3.8.2.1.2 Package and Import Statements
    - 2.5.3.8.2.2 Naming Conventions
      - 2.5.3.8.2.2.1 Capitalization Conventions
      - 2.5.3.8.2.2.2 Type Member Names
      - 2.5.3.8.2.2.3 General Names
      - 2.5.3.8.2.2.4 Assembly Names
      - 2.5.3.8.2.2.5 Package Names
      - 2.5.3.8.2.2.6 Resource Names
  - 2.5.3.8.3 Layout Conventions
    - 2.5.3.8.3.1 Java Programming Example - Wrapping Lines
  - 2.5.3.8.4 Java Programming Commenting Conventions
    - 2.5.3.8.4.1 Java Programming Single Line Comments
    - 2.5.3.8.4.2 Java Programming Block Comments
  - 2.5.3.8.5 Class Design
    - 2.5.3.8.5.1 Packages
    - 2.5.3.8.5.2 Interfaces
    - 2.5.3.8.5.3 Classes
      - 2.5.3.8.5.3.1 Abstract Classes
      - 2.5.3.8.5.3.2 Sealed Classes
      - 2.5.3.8.5.3.3 Static Classes
      - 2.5.3.8.5.3.4 Inner Classes
      - 2.5.3.8.5.3.5 Immutable Classes

- 
- 2.5.3.8.5.3.6 Objects
  - 2.5.3.8.5.3.7 Class Access Modifiers
  - 2.5.3.8.5.3.8 Fields
  - 2.5.3.8.5.3.9 Types
    - 2.5.3.8.5.3.9.1 Autoboxing and Unboxing Types
    - 2.5.3.8.5.3.9.2 Enumeration Types
    - 2.5.3.8.5.3.9.3 Nullable Types
    - 2.5.3.8.5.3.9.4 Nested Classes
    - 2.5.3.8.5.3.9.5 Numeric Types
    - 2.5.3.8.5.3.9.6 Generics
  - 2.5.3.8.6 Statements
    - 2.5.3.8.6.1 Variable Declaration
    - 2.5.3.8.6.2 Expressions
    - 2.5.3.8.6.3 Conditional Statements
    - 2.5.3.8.6.4 Iteration Statement
    - 2.5.3.8.6.5 Empty Statement
    - 2.5.3.8.6.6 Assertion Statement
  - 2.5.3.8.7 Expressions
    - 2.5.3.8.7.1 Lambda Expressions
  - 2.5.3.8.8 Operators
  - 2.5.3.8.9 Member Design
    - 2.5.3.8.9.1 Member Overloading
    - 2.5.3.8.9.2 Constructor Design
    - 2.5.3.8.9.3 Finalizer Design
    - 2.5.3.8.9.4 Field Design
    - 2.5.3.8.9.5 Property Design
      - 2.5.3.8.9.5.1 Abstract Properties
      - 2.5.3.8.9.5.2 Constants
    - 2.5.3.8.9.6 Parameter Design
      - 2.5.3.8.9.6.1 Variable Length Parameter
      - 2.5.3.8.9.6.2 Event Design
    - 2.5.3.8.9.7 Methods
    - 2.5.3.8.9.8 Language Guidelines
      - 2.5.3.8.9.8.1 Arrays
    - 2.5.3.8.9.9 Exceptions
      - 2.5.3.8.9.9.1 Catching and Handling Exceptions
      - 2.5.3.8.9.9.2 Throwing Exceptions
      - 2.5.3.8.9.9.3 Unchecked Exception Best Practices
    - 2.5.3.8.9.10 Concurrency

- 2.5.3.8.9.10.1 Threads
- 2.5.3.8.9.10.2 High-Level Concurrency
- 2.5.3.8.9.11 Native Code Interoperability
- 2.5.3.8.9.12 Design for Extensibility
  - 2.5.3.8.9.12.1 Unsealed Classes
  - 2.5.3.8.9.12.2 Protected Members
  - 2.5.3.8.9.12.3 Events and Callbacks
  - 2.5.3.8.9.12.4 Virtual Members
  - 2.5.3.8.9.12.5 Abstractions
  - 2.5.3.8.9.12.6 Base Classes for Implementing Abstractions
  - 2.5.3.8.9.12.7 Sealing
- 2.5.3.8.9.13 Secure Coding Guidelines
  - 2.5.3.8.9.13.1 Fundamentals
  - 2.5.3.8.9.13.2 Denial of Service
  - 2.5.3.8.9.13.3 Confidential Information
  - 2.5.3.8.9.13.4 Input Validation and Data Sanitization
  - 2.5.3.8.9.13.5 Injection and Inclusion
  - 2.5.3.8.9.13.6 Accessibility and Extensibility
  - 2.5.3.8.9.13.7 Serialization and Deserialization
  - 2.5.3.8.9.13.8 Access Control
  - 2.5.3.8.9.13.9 Defensive Use of the Java Native Interface (JNI)

#### Exhibits

- 2.5.3-1 Java Programming- Example of Wrapping Lines
- 2.5.3-2 Java Programming Example of Objects
- 2.5.3-3 Java Programming - Object Test Results
- 2.5.3-4 Java Programming Example of “instanceof”
- 2.5.3-5 Java Programming Example -Subclass
- 2.5.3-6 Java Programming Example - Enumeration Type
- 2.5.3-7 Java Programming Example - Nested Classes
- 2.5.3-8 Java Programming Class Switch Example
- 2.5.3-9 Java Programming Design Example
- 2.5.3-10 Java Programming Constructor Example
- 2.5.3-11 Java Programming Abstract Properties
- 2.5.3-12 Java Programming Example of Event Design
- 2.5.3-13 Java Programming Thread example
- 2.5.3-14 Java Programming Examples of Single Words used for Capitalization Purposes
- 2.5.3-15 C Programming Source Code Template
- 2.5.3-16 C Language Header File Template

- 
- 2.5.3-17 Acronyms and Terms
  - 2.5.3-18 Terms and Definitions
  - 2.5.3-19 Language Code (ALC) Standards and References



2.5.3.1  
(07-10-2020)  
**Program Scope and Objectives**

- (1) **Purpose:** This Internal Revenue Manual (IRM) establishes standards and guidelines to promote the development of maintainable, portable, reliable software applications in all Service used and approved programming languages as outlined in this IRM.
- (2) **Audience:** This guidance applies to all IRS Senior Leadership, Information technology (IT) managers at all levels. Also included are personnel responsible for: engineering, developing, or maintaining Agency software systems identified in the Enterprise Architecture. This engineering, development, and maintenance include services performed by both government employees and contracts.
- (3) **Policy Owner:** The current policy owner is the Associate Chief Information Officer (ACIO), Application Development.
- (4) **Program Owner:** The current program owner is the Director, Technical Integration Organization (TIO).
- (5) Primary stakeholders:
  - a. Application Development (AD) - AD staff, management, and contractual employees
  - b. Developers - government and contractual employees
  - c. Engineers - government and contractual employees
  - d. IRS IT managers
  - e. Quality Assurance (QA) - IRS QA staff, managers, and contractual employees

2.5.3.1.1  
(07-10-2020)  
**Background**

- (1) In response to Government Accountability Office (**GAO-18-298**) **Information Technology (IT): IRS Needs to Address Significant Risks to Tax Processing- Investments Performance and Risks**, June 2018 report provided to Congressional Committees. IRS IT recognized the importance of continuously improving the performance of IRS Major IT Investments. IT IRS organizations' Mainframe systems, using legacy programming languages: Common Business Oriented Language (COBOL), Assembler Language Code (ALC), Java programming; etc., IRS Application Development (AD) has established Risk Management strategies and guidance to identify, analyze, mitigate, and monitor risks and issues for system development standards

2.5.3.1.2  
(07-10-2020)  
**Authority**

- (1) IRM 2.5.1 System Development, establishes the System Development program for the IRS
- (2) This IRM 2.5.3 is consistent with the President's Executive Order 13800, Strengthening the Cybersecurity of Federal Networks and Critical Infrastructure
- (3) Government Accountability Office, (GAO)
- (4) Treasury Inspector General Tax Administration (TIGTA)
- (5) Presidential American Technology Council, 2017
- (6) Administrator of the General Services Administration (GSA)
- (7) Federal Information Security Modernization Act (FISMA) of 2014
- (8) Director of Office of Management and Budget (OMB)

- (9) Secretary of the Department of Homeland Security (DHS)
- (10) Secretary of Commerce for modernization of Federal IT
- (11) Federal Information Processing Standards (FIPS) Pub 73, Guidelines for Security of Computer Applications
- (12) 21st Century Integrated Digital Experience Act (IDEA), December 2018

2.5.3.1.3  
(07-10-2020)  
**Roles and  
Responsibilities**

- (1) **Information Technology (IT), Cybersecurity:** Cybersecurity manages the IRS IT Security program in accordance with the Federal Information Security Management Act with the goal of delivering effective and professional customer service to business units and support functions within the IRS. These procedures are done as the following:
  - a. Provide valid risk mitigated solutions to security inquisitions.
  - b. Respond to incidents quickly, and effectively in order to eliminate risks/threats.
  - c. Ensure all IT security policies and procedures are actively developed, and updated.
  - d. Provide security advice to IRS constituents, and proactively monitor IRS robust security program for any required modifications or enhancements.
- (2) Application Development's chain of command and responsibilities include:
  - a. **Commissioner:** Oversees and provides overall strategic direction for the IRS. The Commissioner's and Deputy Commissioner's main focus is for the IRS's services programs, enforcement, operations support, and organizations. Additionally, the Commissioner's vision is to enhance services for the nation's taxpayers, balancing appropriate enforcement of the nation's tax laws while respecting taxpayers' rights.
  - b. **Deputy Commissioner, Operation Support (DCOS):** Oversees the operations of Agency-Wide Shared Services: Chief Financial Officer, Human Capital Office, Information Technology, Planning Programming and Audit Oversight and Privacy, and Governmental Liaison and Disclosure.
  - c. **Chief Information Officer (CIO):** The CIO leads Information Technology, and advises the Commissioner on Information Technology matters, manages all IRS IT resources, and is responsible for delivering and maintaining modernized information systems throughout the IRS. Assisting the Chief Technology Officer (CTO) is the Deputy Chief Information Officer for Operations.
  - d. **Application Development (AD), Associate Chief Information Officer (ACIO):** The AD ACIO reports directly to the CIO; oversees and ensures the quality of: building, unit testing, delivering, and maintaining integrated enterprise-wide applications systems to support modernized and legacy systems in the production environment to achieve the mission of the service.
  - e. **Deputy AD Associate CIO (ACIO):** The Deputy AD ACIO reports directly to the AD ACIO, and is responsible for:
    - Leading all strategic priorities to enable the AD Vision, IT Technology Roadmap and the IRS future state
    - Executive planning, and management of the development organization which ensures all filing season programs are developed, tested, and delivered on-time and within budget

- (3) **Application Development:** Responsible for building, testing, delivering, and maintaining integrated information applications systems, e.g. software solutions, to support modernized systems and production environment to achieve the mission and objectives of the service. Additionally, AD does the following:
  - a. Work in partnership with customers to improve the quality of the IRS information systems, products and services.
  - b. Maintains the effectiveness and enhance the integration of IRS installed base production systems and infrastructure while modernizing core business systems and infrastructure.
  - c. Establishes and maintains rigorous contract and fiscal management, oversight, quality assurance, and program risk management processes to ensure that strategic plans and priorities are being met.
  - d. Provides quality assessment/assurance of deliverables and processes.
  - e. Creates oversight support of enterprise modernization goals in coordination with Information Technology HR initiatives and policy.
  - f. Responsible for delivering filing season projects, and implementing Economic Stimulus changes.
  - g. AD has the following Domains:
    - Compliance
    - Corporate Data (CD)
    - Customer Service (CS)
    - Data Delivery Service (DDS)
    - Delivery Management; Quality Assurance (DMQA)
    - Identity & Access Management (IAM)
    - Internal Management (IA)
    - Submission Processing (SP)
    - Technical Integration Organization (TIO)
- (4) **Director, Compliance:** Provides executive direction for a wide suite of Compliance domain focused applications and oversee the IT Software Development organization to ensure the quality of production ready applications.
  - a. Directs and oversees an unified cross-divisional approach to compliance strategies needing collaboration pertaining for the following:
    - Abusive tax avoidance transactions needing a coordinated response
    - Cross-divisional technical issues
    - Emerging issues
    - Service-wide operational procedures
- (5) **Director, AD Corporate Data:** Directs and oversees the provisioning of authoritative databases, refund identification, notice generation, and reporting.
- (6) **Director, Customer Service:** Directs and oversees Customer Service Support for the IT Enterprise Service Desk ensuring quality customer to employee relationship.
- (7) **Director, Data Delivery Services:** Oversees and ensures the quality of data with repeatable processes in a scalable environment. The Enterprise Data Strategy is to transform DDS into a data centric organization dedicated to deliver Data as a Service (DaaS) through:
  - Innovation - new methods, discoveries
  - Renovation - streamline or automate
  - Motivate - incent and enable individuals

(8) **Director, Delivery Management & Quality Assurance (DMQA):**

- Executes the mission of DMQA by ensuring AD has a coordinated, cross-domain, and cross-organizational approach to delivering AD systems and software applications
- Reports to the AD ACIO, and chairs the AD Risk Review Board.
- Chairperson, Configuration Control Board, see IRM 2.5.1.2.2.2
- Government Sponsor, Configuration Control Board, see IRM 2.5.1.2.2.2

(9) **Director, Identity & Access Management (IAM) Organization:** Provides oversight and direction for continual secure online interaction by verification and establishing an individual's identity before providing access to taxpayer information "identity proofing" while staying compliant within federal security requirements.(10) **Director, Internal Management:** Provides oversight for the builds, tests, deliveries, refund identification, notice generation, and reporting.(11) **Director, Submission Processing:** Provides oversight to an organization of over 17000 employees, comprised of: a headquarters staff responsible for developing program policies and procedures, five W&I processing centers, and seven commercially operated lockbox banks. Responsible for the processing of more than 202 million individual and business tax returns.(12) **Director, Technical Integration Office:** Provides strategic technical organization oversight ensuring applicable guidance, collaboration, and consolidation of technical integration issues and quality assurance for the Applications Development portfolio.

2.5.3.1.4  
(07-10-2020)  
**Program Management  
and Review**

- (1) The Enterprise Program Management Office (EPMO) is responsible for the delivery of integrated solutions for several of the IRS's large scaled programs. EPMO plays a key role in establishing change, configuration, release plans; and implementing new information system functional capabilities.
- (2) The EPMO is the primary partner with the business for programs under their purview, and collaborates with IT delivery partners (AD, ES, EOPS, and the other ACIO areas) to deliver required capabilities. This structure positions each organization to maintain a strong core function to optimize their operations.

2.5.3.1.5  
(07-10-2020)  
**Program Controls**

- (1) The Enterprise Program Controls (EPC) Office is the lead for EPMO Information Technology enterprise-wide program management functions, and assist with the Applications Development (AD) organization in cross domain support for a variety of program management disciplines. The EPC office is comprised of seven sections: Business Operations, Program Support Services, Investment & Contract Management, Program Oversight & Reporting, Communications & Organization Readiness, Enterprise Transition Management Office, and Technical Integration.
- (2) The controls established in this Internal Revenue Manual (IRM) apply to Service personnel responsible for developing or maintaining the Service's application systems or software applications, identified in the IRS Enterprise Architecture. Service personnel who contract for development or maintenance of these systems/software applications must ensure contracts comply with these controls.

2.5.3.1.6  
(07-10-2020)  
**Acronyms and Terms**

- (1) See Exhibit 2.5.3-17 for Acronyms and Terms
- (2) See Exhibit 2.5.3-18 for Terms and Descriptions

2.5.3.1.7  
(07-10-2020)  
**Related Resources**

- (1) The following are supplement references on the development of maintainable, portable, reliable, and secure software applications.
  - The Elements of Programming Style, ISBN: 0070342075, Brian W. Kernighan and P. J. Plauger
  - IBM High Level Assembler for z/OS Language Reference V1R6
  - IRM 2.5.12 - Design Techniques and Deliverables
  - IRM 2.127.2 , Information Technology Testing Process and Procedures
  - Assembler Language Programming, ISBN: 0-471-88657-2, Nancy Stern, Alden Sager and Robert A. Stein
  - Structured COBOL Programming, ISBN 0-471-29987-1, Nancy Stern and Robert A Stern
  - The Elements of C Programming Style, ISBN 0070512787, Jay Ranade and Alan Nash
  - IBM Assembler System Standards, Chapter 1- 8 ( IRS-defined )
  - IRS Document 12384, C++ Programming Standards
  - Java Programming Language, Oracle Technology Network/Java
  - Java Assembly Package, <https://docs.oracle.com/javase/tutorial/deployment/jar/index.html>
  - Java Programming Package and Import Statements, <https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html>
  - Java Programming Input Validation and Data Sanitation <http://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeIOException.html>
  - How to write documentation comments and available tags <https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>
  - Java primitives - <https://cs.fit.edu/~ryan/java/language/java-data.html> and <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>.
  - IRM 10.8.1 - Security, Privacy and Assurance, IT Security, Policy and Guidance
  - IRM 10.8.6 - Security, Privacy and Assurance, IT Security, Application Security and Development
  - The Open Web Application Security Project (OWASP) <https://www.owasp.org>
  - For additional Assembler Standards and References, see Exhibit 2.5.3-19

2.5.3.2  
(07-10-2020)  
**Federal Government  
Application Standards  
Guidance**

- (1) The Federal Information Security Modernization Act of 2014 (FISMA) was passed for providing a framework with better information security controls over information resources, supporting Federal Government operations and assets. IRS applications must be compliant with federal standards, e.g. NIST SP 800.53A Revision 5 “Assessing Security and Privacy Controls in Federal Information Systems and Organizations”; some key focus areas are:
  - Insider threats
  - Software application security (including web applications)
  - Cross domain solutions
  - Advanced persistent threats
  - Industrial / process control systems
  - Privacy

2.5.3.2.1  
(07-10-2020)

**Application Security  
Control Frameworks**

- (2) For information on IRS Information Technology Cybersecurity controls see IRM 10.8.1 Security, Privacy and Assurance, IT Security, Policy and Guidance and IRM 10.8.6 - Security, Privacy and Assurance, IT Security, Application Security and Development

- (1) The Security Control Framework assist with the organization's legal and regulatory security compliance efforts.
- (2) Application security is the use of software, hardware, and procedural methods to prevent security flaws in applications, and protect them from external threats. Security is a critical objective during development as application become more accessible over networks, are more vulnerable to vast variety of threats. Hence, security measures must be built into application to mitigate unauthorized code manipulation of applications to access, steal, modify, or delete sensitive data. As IRS software developers create robust code for effective IRS applications, to achieve secure software they must embrace and practice a wide variety of secure coding techniques. All tiers of an application: user interface, business logic, controller, database, etc. must be created with security controls in mind. Hence, developers must ensure they follow best practices and guidelines from these primary industries when applicable:
- a. Common Weakness Enumeration (CWE) - Targeted toward developers and security practitioners as a community initiative, and a formal list of software weaknesses types is created and updated for::
    - Providing a common baseline standard for: weakness, identification, mitigation, and prevention
    - Assisting with describing software security weaknesses in architecture, design, or code
  - b. For more information see, <https://cwe.mitre.org/index.html>
- (3) **OWASP Top 10 Proactive Controls 2018** : This is a list of security techniques that must be included in every software development project when applicable. The following control numbers are listed in order of importance, with control number 1 as the most important:
- a. Control 1 - Define Security Requirements
  - b. Control 2 - Leverage Security Frameworks and Libraries
  - c. Control 3 - Secure Database Access
  - d. Control 4 - Encode and Escape Data
  - e. Control 5 - Validate All Inputs
  - f. Control 6 - Implement Digital Identity
  - g. Control 7 - Enforce Access Controls
  - h. Control 8 - Protect Data Everywhere
  - i. Control 9 - Implement Security Logging and Monitoring
  - j. Control 10 - Handle all Errors and Exceptions
- (4) For more information see, [https://www.owasp.org/index.php/OWASP\\_Proactive\\_Controls](https://www.owasp.org/index.php/OWASP_Proactive_Controls)
- (5) The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC), ISO/ IEC 27034:2011+ , Information Technology, Security Techniques, Application Security - Provides guidance on information security for IT Managers, developers and auditors to ensure computer applications have the necessary level of security.



2.5.3.2.1.1  
(07-10-2020)  
**Application Security Controls**

- (6) ISO/ IEC 27034:2011+ - Key purpose is to assist organizations with information security controls through a set of processes integrated throughout the Systems Development Life Cycle (SDLC). The standard is a SDLC-method-agnostic; i.e., it does not mandate one or more specific development methods or approach; therefore, it complements other systems development standards and methods without confliction with them.
- (7) For more information see, <https://iso27001security.com/html/27034.html>

- (1) Application controls are a form of security that blocks or restricts applications from executing in ways that put data at risks, and also designed to improve the quality of data that is input into databases.
- (2) The key purpose is to reduce the risks and threats associated with applications by ensuring the confidentiality, integrity , and availability of data transmitted between applications. Examples of applications controls include the following:
  - a. **Completeness checks:** Controls ensure records are processed from initiation to completion
  - b. **Validity checks:** Controls ensure only valid data is input or processed
  - c. **Identification:** Controls ensure unique, unquestionable identification of all users
  - d. **Authentication :** Controls ensure access to the application system by approved business user only
  - e. **Input Controls:** Controls ensure data integrity feeds into the application system from valid sources
  - f. **Defense-in-Depth:** Is a security implementation which has layers of security implemented to protect an asset from unauthorized access or modifications. The objective is about layering defense tools in order to minimize the number of vulnerabilities in applications that would allow the occurrence of different attacks. For example, if one security layer fails the next security layer will catch the breach-of-attack at the next security layer, i.e. Client, Server, Application, and Database protection.

2.5.3.2.2  
(07-10-2020)  
**AD Waivers**

- (1) IRM 2.5.14 Systems Development, Quality Assurance documents the Application Development (AD) Delivery Management & Quality Assurance (DMQA) waiver process for tracking any project team's noncompliance of accepted IRS standards.

2.5.3.3  
(07-10-2020)  
**General Programming**

- (1) This section of the IRM is based on specific enterprise platforms and languages - These IRS standards and guidelines pertain to application program development and documentation efforts.
- (2) The objective of this section is to promote the development of programs that are reliable, modular, easily maintainable, and as portable as possible.
- (3) New software tools for application development and decision support may supplement and/or replace traditional design and programming techniques. Commercially acquired software packages may reduce development time by eliminating "detailed" design and programming activities. Off-the-shelf software packages should be carefully considered before the decision is made to develop software.

- (4) The scope of this directive is service-wide. This includes software developed by contractors where the guidelines apply to Assembler Language, COBOL, C Language, C++ programming, and Java programming.

2.5.3.3.1  
(07-10-2020)  
**Goals**

- (1) The primary goal of structured programming is to produce working programs that are: modular, accurate, and self-documenting, so that they are easily read and maintained by someone other than the original author.
- (2) Structured programming includes the following activities:
- Developing specifications for the logic of each module
  - Writing structured code to implement the logic of the module
  - Using a structured testing methodology that gradually creates a working program as each module is introduced into the application system

2.5.3.3.2  
(07-10-2020)  
**Basic Principles**

- (1) Structured programming employs the use of limited syntax (constructs) for source code, single-entry/single-exit modules, and top-down development.
- (2) Base the logic of each module on various combinations of control structures. The three basic constructs are Sequence, Selection (If-Then-Else), and Repetition (Do-While)/(Test-First). Two optional constructs include Repetition (Do-Until)/(Test-Last) and Selection (Case).
- (3) Exhibit 2.5.3-4 depicts a flowchart and Structure diagram for each construct. The actual implementation of these structures will vary according to the requirements of the particular language being used.
- (4) Ensure that each module has only one entry point to and one exit point from the module.
- (5) Partition and organize each module, program, and application system into a hierarchical structure. Structure charts, module specifications, and structured code are part of the design of a system.

2.5.3.3.3  
(01-01-2004)  
**Design Specifications**

- (1) Various tools are commonly used to communicate and transition design specifications to source code. These tools are:
- a. Structure charts
  - b. Module specifications

IRM 2.5.12 - Design Techniques and Deliverables, provides comprehensive standards and guidelines regarding structure charts and module specifications during design.

2.5.3.3.4  
(07-10-2020)  
**Documenting, Testing,  
and Debugging Source  
Code**

- (1) This section addresses services that must be performed regardless of the language or platform selected.



2.5.3.3.4.1  
(07-10-2020)  
**Documenting Code**

- (1) Document each module and paragraph for future modifications or review/use/ modification of the code.
- (2) Ensure that all source code is well documented, clear, understandable, and easy to modify and maintain.
- (3) Make each module a small block of source code that does not exceed one page of printed output (exclusive of comments).
- (4) Indent source code statements.

2.5.3.3.4.2  
(07-10-2020)  
**Testing and Debugging Code**

- (1) Review, analyze and test the code for consistency, correctness, clarity, and completeness according to IRS coding standards.
- (2) Test the software according to IRM 2.127.1 Testing Standards and Procedures; and IRM 2.127.2 IT Test Policy and , Information Technology Testing Process and Procedures.

2.5.3.3.5  
(07-10-2020)  
**Selecting Programming Languages**

- (1) For new projects, select the programming language based on IRS standards, executive leadership mandates, engineering requirements; and Enterprise Architecture's recommendations.

2.5.3.3.6  
(07-10-2020)  
**Data Controls**

- (1) Data controls must be designed with the purpose of functions of the program, and variable data types in mind in order to reduce potential conflicts.
- (2) This subsection provides general guidelines for developing data controls and examples of data control types that are often used in system development. This is not an all-inclusive list of controls, but rather a general framework for control development.
- (3) Data controls must have one purpose for each variable.
- (4) Variable scope must be apparent and limited, i.e., the set of program functions which can access the variable.
- (5) Variables must only be "global" as needed. Only functions which require a variable must have access to it.
- (6) Data controls permit an operating entity to verify that the correct operations have been performed, in the correct manner, with the correct data.
- (7) Ensure that Data control considerations comprise an integral part of the design process.
- (8) Place controls as close as possible to the source of the data, e.g., verification of data immediately after it is entered; block balancing before data is released to update modules, etc..
- (9) Automate controls whenever possible.
- (10) Keep controls simple to read and balance, and easy to maintain.
- (11) Explain the purpose and use of controls. Describe how the totals were derived.
- (12) Record counts must be provided and broken down into logical records for each run.

2.5.3.3.6.1  
(07-10-2020)

**Basic Principles of Data Controls**

- (1) Controls refer to the manual and automated measures supported to:
  - Preserve the accuracy of data by detecting and/or preventing operator errors.
  - Ensure data is not lost or added, by monitoring balances between processes.
  - Ensure data integrity so programs do not unintentionally change the values of data.
  - Permit appropriate recovery/reconstruction of file data after a system failure or abnormal termination.
  - Safeguard sensitive data to prevent unauthorized access, embezzlement, and other breaches of security.

2.5.3.3.6.2  
(07-10-2020)

**Programming Considerations for Data Controls**

- (1) Integrate controls into the development effort. The types of controls, and the amount of detail are dependent upon the size and complexity of the application system.
- (2) Weigh each development effort based on the following operational considerations:
  - The amount of operator intervention
  - Multi-file/multi-cartridge processing
  - Checkpoint/restart capability
  - The file ID on all internal reports
  - Back-up of control file
  - Initialization of working storage and output buffers with spaces and zeros
  - Run to run balancing

2.5.3.3.6.3  
(07-10-2020)

**Internal Controls**

- (1) Internal controls are balancing procedures developed to verify the validity of the processing within a run. Internal controls are usually a response to user requirements for accuracy, completeness and security within an information system. Segment these controls into three classes:
  - a. Controls over input
  - b. Controls over processing
  - c. Controls over output

2.5.3.3.6.3.1  
(06-01-2002)

**Input Controls**

- (1) Input controls are the most important and the most numerous. Most errors are generated during input processing. Some common techniques are:
  - Check digit verification--Use check digits to review the accuracy of specific fields. For example, a check digit can help determine whether an account number is valid.
  - Consistency tests--If the application permits it, verify accuracy by comparing the values of various fields to determine whether the combinations make sense. For example, if the "Country" field indicates that the record concerns an organization in Canada, the "Postal Code" field should have a specific alphanumeric format.
  - Validity tests--In some cases, fields can take only a limited range of values, or must have a predetermined format. Matching the actual value to the allowable values will detect errors. For example, if a field is supposed to contain a valid U.S. postal abbreviation for a state, "AZ" would be valid but "A2" would not.

- Batch numbering--This technique ensures that transactions are not lost. Processor checks can be made to assure that all transactions are accounted for and processed in a logical order.
- Control totals--These totals help avoid errors during data entry. Various input fields (e.g., check amount or quantity received) are added both manually and automatically for comparison. In some cases, these totals are developed for fields that would normally not be added (e.g., account numbers or social security numbers). These are called hash totals. In either case, both the expected totals and the individual transactions are passed to the application system. The application system then recalculates the totals from the individual records received and compares them to the expected totals. If they don't match, an error has been detected.
- Transaction counts--Use this method to keep track of the number of transactions that should have been processed by the application system.

#### 2.5.3.3.6.3.2 (06-01-2002)

##### **Processing Controls**

- (1) There are two major types of processing controls:
  - a. Run to Run
  - b. File and Operator
- (2) Run to run controls consist of data generation controls and verification controls:
  - Use data generation controls to ensure that the correct version of the file is being used
  - Verification controls ensure that the totals or record counts for the prior run match the opening totals for the current run (e.g., header/trailer counts)
- (3) File and operator controls are actions that the operator can take to ensure that the application system is processing the right files and data. The controls can be as simple as checking a cartridge. Operator intervention should be kept to a minimum. Operator controls should be very specific and should be accompanied by sufficient operator instruction. For example, if the operator receives a message on the console: CARTRIDGE LABEL ERROR Enter "R" to retry, "N" to abort, "A" to accept. The operator should not be able to override this message.

#### 2.5.3.3.6.3.3 (06-01-2002)

##### **Output Controls**

- (1) There are three types of output controls:
  - a. Control Totals
  - b. Verification Controls
  - c. Distribution Controls
- (2) Use control totals to verify the correctness of the outputs. For example, if an accounts payable application system generates 236 checks with an expected value of \$395,000.12, the checks could be physically added to verify that the actual values of the checks were generated.
- (3) Use verification controls to coordinate internal and external processes. For example, to avoid unauthorized loss of blank checks, have the computer keep track of the expected serial numbers of the preprinted checks and print the expected number on the check. If the two numbers differ, something is wrong.

- (4) Use distribution controls to ensure that once an output is printed, it is delivered to the authorized recipients. This includes having users sign for reports on-site, as well as controls between sites.

2.5.3.3.6.4  
(07-10-2020)

**External Data Controls**

- (1) External controls consist of that information necessary for operations personnel to perform balancing between and within runs. These controls are manual in nature and should include precise instructions as to:
- Which output listing/file contains the control data
  - What type of control data is being generated, e.g., transaction counts, hash totals, etc.
  - How to balance the various elements of control data, e.g.,  $\text{ITEM 1} + \text{ITEM 2} = \text{ITEM 3}$
- (2) Accumulate and print controls at the end of processing must include, at a minimum:
- Counts of total inputs and outputs;
  - Balancing counts;
  - Information counts;
  - Run to run counts; and to
  - Generated, dropped and error records,

2.5.3.3.6.4.1  
(07-10-2020)

**Control Totals**

- (1) Keep a record of the data as it moves through an application system and is subjected to a series of manual and automated processes. This can be accomplished in two ways:
- a. Control Totals
  - b. Control File
- (2) Control totals can be embedded in the process itself. This is not the best approach since these totals are easily modified.
- (3) A separate, highly controlled (limited user access) "control file" is very effective in that it is not as accessible as the data files. This file should include the following:
- Block and/or record counts, hash totals, and total counts
  - Logical record counts, when they differ from tape-record counts
  - Controls on money amount fields (cumulative arithmetic totals)
  - Adequate controls to account for all records: including those dropped, by-passed or combined during processing

2.5.3.3.6.4.2  
(07-10-2020)

**Intra-Run Controls**

- (1) Intra-run controls generate and/or present control information to operations personnel during the execution of the run.
- (2) When designing a program, limit the amount of intervention required by operations personnel. As this is not always possible, consider the following ideas when developing intra-run controls:
- Enable the run to print all operationally controlled parameters used for the run

- Stack all control data to a separate tape/disk file and print at the end of the job. Don't clutter the console with control information during processing
- Print totals for each run, every time, even when the totals are in balance

2.5.3.3.6.5  
(07-10-2020)  
**Including Data Controls**

- (1) Include computer generated control lists with record counts by file, file number and name, money amounts and tape/disk numbers.
- (2) Make sure that programs generate identifying information on all internally used output (e.g., reports). The project/run/file ID will be printed on each page of printed output. Do not print this information on transcripts, taxpayer letters and notices, and externally distributed reports.
- (3) Include instructions for manually processing the control list.
- (4) Include computer generated cartridge numbers on all control lists:
  - Print cartridge file ID on controls page (from job number next to the corresponding count).
- (5) Computer generated hard copy control output for all runs.
- (6) List all control features in either the user handbook and/or the Computer Operators Handbook (COH), explaining:
  - The purpose and use of each control
  - How they were derived and their meaning
  - The cause and meaning of all programmed halts
- (7) Assign a unique identifier to each cartridge file.
- (8) When processing a multi-reel program that also has multi-file input, use halts at the end of each file if the accumulated counts are not equal to the record count in a trailer record.
- (9) Institute checkpoint/restart capabilities for any application with estimated or actual run times that exceed one hour normal processing time as well as for large programs that process extensive amounts of data.

2.5.3.3.7  
(01-01-2004)  
**File Design and Cartridge Interface Formats**

- (1) This subsection addresses file design and cartridge interface format considerations.

2.5.3.3.7.1  
(06-01-2002)  
**File Design Formats**

- (1) The following sections include the design of the sequential file and logical data record formats. They are concerned with the association or grouping of the data elements into groups and records.

2.5.3.3.7.1.1  
(06-01-2002)  
**Record Format Design**

- (1) Fixed length records--a file composed of records that are all the same length.
- (2) Variable-length records/multiple fixed formats--a file composed of a finite number of fixed length record sets, where the record lengths within any set are equal, but the record lengths between sets differ.

- (3) Variable-length records/variable subscripted format--a file composed of one or more sets of records whose format consists of a fixed portion followed by a variable number of repeating groups. These groups must either be fixed in length, or composed of a fixed portion plus a subgroup whose entries are fixed in length.
- (4) Variable-length records/variable string format--a file composed of records consisting of character strings of unspecified lengths.

2.5.3.3.7.1.2  
(07-10-2020)

#### **Defining Data Fields**

- (1) When defining data fields which will compose a file, do not assign multiple uses for the same field; e.g., if a field is labeled DATE, the values carried by that field should be date information in all cases.
- (2) Specify all the search key fields, and if possible, place them at the beginning of the record.
- (3) Reduce redundant data fields to the minimum.
- (4) Specify sensitivity levels for files. Classify all the sensitive data fields that require authorization for access.
- (5) Restrict data fields to one and only one data item. This is really a VERY important standard to enforce.
- (6) The name should comply with IRM 2.152.3 Information Technology, Data Engineering, Naming Data Elements(s)/Objects(s) to include characteristics such as:
  - a. The name must be easily defined
  - b. The name must reflect and be specific to what is in the field (e.g. IRS-Mailing-Dt.)
  - c. Data names must end in a class word, indicating the data type

2.5.3.3.7.1.3  
(06-01-2002)

#### **File Design**

- (1) Use "Fixed" and "variable multiple fixed" formats when possible.
- (2) Avoid variable length records/variable subscripted format (i.e., Nth dimensional groups, where N is greater than 2).
- (3) Do not use variable string formats.

2.5.3.3.7.2  
(07-10-2020)

#### **Tape Interface**

- (1) Tape interface standards reduce the difficulty of sharing data between different users and different application systems. They allow the users to consider only the logical structure of files, and simplify the transporting and maintenance of data.
- (2) All files created on an application system to be processed on another must:
  - Contain only ASCII character data
  - Be in either Fixed or Variable format
  - Carry signs (+ or -) as a separate, leading ASCII character for signed numeric data fields. The reason for carrying signs separately, is to maintain consistency between application systems because of the Implementor option
- (3) All files that are passed between application systems will be limited to 9995 characters per record.

- (4) Record lengths (for variable records) consist of four decimal (ASCII) characters in the Record Control Word (RCW). The RCW is automatically generated by the application system and precedes each logical record.

2.5.3.3.8  
(04-15-2004)  
**Date Fields**

- (1) This subsection pertains to date fields and addresses the following topics:
  - a. Year
  - b. Date
  - c. Gregorian Dates
  - d. Exceptions

2.5.3.3.8.1  
(04-15-2004)  
**Year**

- (1) The all year fields format output must be represented as “YYYY” .

2.5.3.3.8.2  
(04-15-2004)  
**Date**

- (1) Do not store non-date values in DATE fields (i.e., indicators, freeze codes).
- (2) Do not use any DATE field to store non-date information, as in the case of moving all 9 's to a field as an indicator of a particular status.
- (3) Do not store special characters in any DATE fields.
- (4) Make DATE field names meaningful and accurately descriptive of the date stored in the fields, (e.g., BIRTH-DATE ).
- (5) Add validity checks for DATE fields entered on screens or at their initial entry point into Service Application Systems. This includes External Trading Partners Processing.
- (6) Externalize literal usage of dates wherever possible. For example, interest rates that apply to certain date ranges would be established as a data file or database table rather than being hard-coded in the program. If at all possible, eliminate hard-coded dates.
- (7) Use system-wide standard DATE routines (either IRS-developed or COTS) in source code, wherever possible.

2.5.3.3.8.3  
(04-15-2004)  
**Gregorian Dates**

- (1) All Gregorian dates must be in (YYYYMMDD) format.

2.5.3.3.8.4  
(04-15-2004)  
**Exceptions**

- (1) Archive data no longer included in regularly scheduled processing need not be converted.
- (2) Transmittal numbers and data set names (including File Names) containing dates need not be converted.

2.5.3.4  
(07-10-2020)  
**COBOL Programming**

- (1) This subsection provides establishes controls to ensure COBOL programs are reliable, maintainable, and portable.



2.5.3.4.1  
(07-10-2020)  
**COBOL Overview**

- (1) The Common Business-Oriented Language (COBOL) is a high-level computer programming language developed during 1959 created for its portability and readability of programs as normal English instead of machine language. COBOL is known best for processing large quantities of business data through record and data structure methodology. For example, a record clusters heterogeneous data: ID, name, age, and address into a single unit. A committee of computer manufacturers, users, and U.S. government organizations created CODASYL (Committee on Data Systems and Languages) to establish, oversee the language standard to ensure COBOL's portability across dissimilar systems.
- (2) The controls prescribed are applicable to all IRS COBOL programs whether they are developed by the IRS or outside vendors for the IRS.

2.5.3.4.2  
(11-26-2001)  
**COBOL Basic Principles**

- (1) The development of structured COBOL programs in accordance with this section is dependent on structured design.
- (2) Structured COBOL code is the implementation of the logic depicted in module specifications. Module specifications directly correspond to the modules shown on the structure chart.
- (3) Structure charts, and therefore module specifications and structured code, are based on a top-down design of the application system. Each of the modules that constitute a structure chart should have a single entry point and a single exit point. The logic of each of the modules is based on various combinations of the three control structures: sequence, selection, and iteration.
- (4) These principles have been established with the understanding that COBOL programs are not always maintained by the original author. All structured programs will have the same visual format. Only the most common formats are discussed.

2.5.3.4.3  
(07-10-2020)  
**COBOL Structured Programming**

- (1) Structured programming is comprised of three logical structures:
  - a. Sequence
  - b. Selection
  - c. Iteration
- (2) **Sequence structure:** In a sequential structure, the commands are executed in sequence. The flow of the program is to complete one instruction and then drop down and execute the next instruction and then the next until something terminates the sequence such as the end of a paragraph.
- (3) **Selection structure:** In a selection structure the processing is dependent on a condition that is being tested. In COBOL, the selection structure is usually accomplished with an IF or an EVALUATE (the implementation of the case structure in COBOL) or with an implied IF such as the AT END clause in the READ statement.
- (4) **Iteration structure (LOOP STRUCTURE):** The iteration structure causes something to be executed over and over again until some condition terminates the repetition. Additional information is as follows:
  - a. This structure is essentially the looping structure that has been used in all of the programs.



- b. When defining iteration, there are two basic structures that a language may implement: Do-While and Do-Until.
- c. The difference between the two structures is when the condition is tested. In the Do-While structure the condition is tested before the loop is executed while in the Do-Until structure the condition is tested after the loop has been executed. This means that with the Do-While structure there is a possibility that the loop will never be executed.
- d. The PERFORM...UNTIL used in the sample programs is an example of the Do-While structure because the condition is tested before the loop is executed.

2.5.3.4.3.1  
(07-10-2020)  
**COBOL Programming  
Standards**

- (1) This section applies to all divisions of a COBOL program.
- (2) COBOL programs must be written in accordance with the American National Standard Institute (ANSI). Where a standard is not specified in this manual, the relevant ANSI standard will be considered the established standard.
- (3) Begin to insert comment line in these specified areas:
  - a. **IDENTIFICATION DIVISION:**
    - 1. Program Name (Prog-ID)
    - 2. Author Name
    - 3. Installation (Usually with multiple locations)
    - 4. Date Written
    - 5. Date Compiled
    - 6. Security Information
  - b. **ENVIRONMENT DIVISION:**
    - 1. Configuration Section:
      - a. Source-Computer - Describes the computer where the source will be compiled
      - b. Object-Computer - Specifies the system where the program is designated/stored
    - 2. INPUT/OUTPUT Section (Associated Input and Output files):
      - a. FILE-CONTROL
      - b. I/O CONTROL
  - c. **DATA DIVISION:**
    - 1. File Section - Defines the structure of data files
    - 2. Working-Storage Section - Describes data records not part of data files
    - 3. Linkage Section - Used if your program uses data from another program
  - d. **PROCEDURE DIVISION::**
    - 1. Sections within this division must always start with paragraph names e.g., MAIN or other descriptive paragraph names that describe their function.
- (4) Place division, section, and paragraph names on a line by themselves and start in column 8. This also applies to the module names corresponding to structure chart modules.
- (5) Insert a blank line between each Division name and the first statement of the Division.
- (6) Insert a blank line between each Section name and the first statement of the Section.

- (7) Do not split names or words between lines. If possible, avoid splitting literals between lines.
- (8) Only one statement per line is allowed.
- (9) With the exception of nested IF or EVALUATE constructs, end each statement with a period.
- (10) Indent statements that are continued on another line at least two spaces from the starting position of the initial line.
- (11) Use blank lines and page ejects effectively.
- (12) Use meaningful names. Ensure names conform to IRM 2.5.7 Data Naming Standards.

2.5.3.4.3.2  
(11-26-2001)  
**COBOL Identification  
Division**

- (1) Include the following paragraphs in the IDENTIFICATION DIVISION of all programs: AUTHOR, INSTALLATION, SECURITY, and REMARKS. When necessary, they will be annotated as COBOL comments.
- (2) The AUTHOR paragraph will include:
  - The name and office symbols of the section(s) responsible for the maintenance of the program.
  - At a minimum, the name of the last programmer/analyst to write or modify any of the code of the program.
  - It is a good practice to retain the names of the last few authors to allow quicker access to originators of code if problems arise.
- (3) The INSTALLATION paragraph will contain "INTERNAL REVENUE SERVICE".
- (4) The SECURITY paragraph will contain "FOR OFFICIAL USE ONLY".
- (5) REMARKS paragraph will describe the function of the program, the subprograms that are called, the files that are used by the program, and the effective date. At the developer's option, this paragraph may also list modified modules and reasons for modifications after the program has been in production. (This often leads to quicker resolution of problems.)

2.5.3.4.3.3  
(11-26-2001)  
**COBOL Environment  
Division**

- (1) Start each main clause (for example, the SELECT clause) in column 12.
- (2) Start each sub-clause in column 16.

2.5.3.4.3.4  
(07-10-2020)  
**COBOL Data Division**

- (1) Start FD and 01 entries in column 8. Clauses of FD entries will start in column 12, one clause per line.
- (2) Put level numbers in sequential order to allow for future growth (e.g., 01, 05, 10, 15 other than 01, 02, 03). This allows for adding of new fields under a section without having to renumber a file layout or copybook.
- (3) Indent level numbers 4 positions for each subordinate level.
- (4) Indent data names (including condition names) 2 columns to the right of the level number. For example, see the following figure.

**COBOL Example - Data Name Indentations**

Data Name Indentations		
<b>02</b>	<b>NO-MORE-MASTERS-FLAG</b>	<b>PIC X(5)</b>
	<b>88 NO-MORE-MASTERS</b>	<b>VALUE "TRUE".</b>
	<b>88 MORE-MASTERS</b>	<b>VALUE "FALSE".</b>

**Figure 2.5.3-1**

- (5) Start all PIC, VALUE, USAGE, OCCURS, and REDEFINES clauses in the same column, where possible.
- (6) PIC clauses must not contain sequences of more than two identical symbols (except for edited fields). For example, use PIC X(4) rather than PIC XXXX. An edited field such as PIC ZZ,ZZZ.99 will be allowed.
- (7) Group levels according to function type for example; counters and to be used as internal program documentation.
- (8) Ensure that local variables and constants associated with one module immediately follow each other in the DATA DIVISION. If a variable or constant is associated with more than one module, it should usually be defined with the highest level module that references it.
- (9) Do not give flags and indexes multiple uses.
- (10) Initialize constants, variables and output record areas by using the initialize statement or as follows:
  - Initialize constants in working storage, including FILLER fields, with a VALUE clause. Use VALUE SPACES or ZEROS, not "b" or "0".
  - Initialize the variables in working storage (i.e., those fields that are changed during execution of the program) by using specific statements in the PROCEDURE DIVISION.
  - Initialize output record areas to clear buffers that are not overlaid during program execution. One way to initialize an output record area is to move SPACES to the record as a group item, and then move ZEROS to the numeric fields.
- (11) Use meaningful data names derived from the problem being solved. Where applicable, data names should be consistent with those used in the structure charts. COBOL allows names of up to 30 characters. Ensure that names conform with IRM 2.152.3 .
- (12) Avoid data names that convey little meaning. For example, see the following figure

**COBOL Example - Data Naming Standard**

**INDEX, I, K, TR127, COUNTER, EOT.**

**Figure 2.5.3-2**

- (13) Use data names that convey meaning. For example, see the following figure.

**COBOL Example - Meaningful Data Names**

	<b>MESSAGE-INDEX</b>
	<b>TRANSACTION-COUNT</b>
	<b>NO-MORE-TRANSACTIONS-FLAG</b>

**Figure 2.5.3-3**

- (14) Apply the PIC 9 versus PIC X standard to date fields in the following manner:
- Use PIC 9 for date fields in situations where the particular date value in question will be used for numeric functions (e.g., calculations, computations, estimations, etc.) rather than for accepting input or direct display. Assign four positions to the year field (YYYY) and do not store non-date values or special characters in the date field.
  - Define the data as necessary (PIC X, PIC 9, PIC S9, or another format) in order to accommodate input that may be blank, coming from electronic files, External Trading Partners (e.g., SSA), taxpayer submitted files, DB2 special formats, unique database machine formats, or other formats. It is not necessary to use PIC 9 when defining fields that are accepting input.
  - Define the data as necessary (PIC X, PIC 9, PIC S9, or another format) in order to display data (e.g., reports, screens) or format/unformat display dates that contain special characters in edit fields (e.g., slashes, commas, dashes, etc., depending on the function requested). However, note that the Year Field must be four positions (YYYY). It is not necessary to use PIC 9 when displaying data.

2.5.3.4.3.5  
(07-10-2020)  
**COBOL Procedure  
Division**

- (1) A functional module should be limited to 50 lines of executable code as a general rule.
- (2) Each module (not each paragraph within a module) must start on a new page with comment lines indicating the module number of the Structure Chart that is represented by the code and the function of the module as described on the Module Specification. For example, see the following figure.

*COBOL Example - Structure Chart*

Structure Chart Example		
<b>/**</b>	<b>MODULE</b>	<b>2.4.3.7</b>
<b>*</b>		
<b>*</b>	<b>(Description of module)</b>	
<b>*</b>		
<b>*</b>	<b>GET-VALID-TRANSACTION.</b>	
	<b>READ TRANSACTION-FILE</b>	
	<b>AT END</b>	
	<b>MOVE "TRUE" TO NO-MORE-TRANSACTIONS-FLAG.</b>	
	<b>--rest of code--</b>	

**Figure 2.5.3-4**

- (3) Module names in a COBOL listing must correspond to Structure Chart module names.
- (4) Name a paragraph that is an implementation of a control structure (e.g., PERFORM-UNTIL, ELSE, CASE, nested IF-THEN-DO-UNTIL, etc.) in a way that explains its purpose. These paragraphs are not separate modules; they are paragraphs within the module.
- (5) Arrange modules in a program listing in either a horizontal or a vertical sequence corresponding to the Structure Chart level numbers, see Figure 2.5.3-8

*COBOL Example - Structure Chart Level Numbers*

Structure Chart	Level Numbers
0.0	0.0
1.0	1.0
2.0	1.1
3.0	1.2
1.1	2.0
1.2	2.1
2.1	2.1.1
2.2	2.1.2
2.3	2.1.2.1
2.1.1	2.1.2.2
2.1.2	2.1.3
2.1.3	2.2
2.2.1	2.2.1
2.2.2	2.2.2
2.1.2.1	2.3
2.1.2.2	3.0
etc.	etc.

**Figure 2.5.3-5**

- (6) Code the READ statement and WRITE statement options one per line, indented 2 columns. See the following figure.

*COBOL Example - Read/Write Statement Options*

READ file-name			WRITE record-name		
	AT END			AFTER ADVANCING identifier LINES	
		statements.			statement.
or,			or,		
READ file-name			WRITE record-name		
	INVALID-KEY			INVALID-KEY	
		statements.			statement.

**Figure 2.5.3-6**

- (7) Place phrases such as AT END, WHEN, and VARYING on the next line indented two columns.
- (8) Begin any statement not covered by other indentation rules in the same column as the statement above it.
- (9) Never use the ALTER verb (or any other method of dynamically altering the PROCEDURE DIVISION).
- (10) Do not use the GO TO verb, except in the implementation of the CASE construct or in the use of internal SORT exits.
- (11) Handle all file openings and closings in any given module with one OPEN or CLOSE statement. The following figure depicts these formats:

**COBOL Example - Opening and Closing Statements**

Opening and Closing Statements	
<b>OPEN INPUT</b>	<b>file-name-1</b>
	<b>file-name-2</b>
<b>OUTPUT</b>	<b>file-name-3</b>
	<b>file-name-4</b>
<b>CLOSE</b>	<b>file-name-1</b>
	<b>file-name-2</b>

**Figure 2.5.3-7**

- (12) Immediately follow the MOVE CORRESPONDING statement with a comment documenting all data items involved. This ensures thorough documentation. For example, see the following figure.

**COBOL Example - MOVE CORRESPONDING Statement**

<b>MOVE CORRESPONDING RECORD-A TO RECORD-B</b>
<b>*FIELD-1, FIELD-3, FIELD-5.</b>

**Figure 2.5.3-8**

- (13) Use the COMPUTE verb to develop Arithmetic operations with the following exceptions:
  - Use the DIVIDE statement to compute remainders.
  - ADD X to (counter) and SUBTRACT X from (counter) are allowed.
- (14) STOP RUN must only occur once as the last logical statement in the main procedure of a program. EXIT PROGRAM may only occur as the last logical statement of the main procedure of a subprogram. EXCEPTION: In some cases, it may be justifiable to use a STOP RUN in a low-level module of a very

large run. While using an On-Line program Customer Information Control System (CICS) in this case you need the GOBACK statement.

- (15) A PERFORM statement must be used one at a time and contain a coinciding EXIT statement for each, preventing errors or executing the wrong code. Explicitly identify paragraphs e.g., the following two figures. The first figure illustrates a statement that would not satisfy this standard. The second figure illustrates a statement that satisfies this standard.

***COBOL Example - PERFORM Statements***

**COBOL PERFORM Statements**

**PERFORM Paragraph-A.**

PERFORM 0100-COMPUTE THRU 0100-EXIT

**PERFORM Paragraph-B.**

PERFORM 0200-COMPUTE THRU 0200-EXIT

**Figure 2.5.3-9**

- (16) The following figure illustrates the CASE/CASE-END statements standard.

***COBOL Example - CASE/CASE-END Statements***

**PERFORM Case-Paragraph**

**THRU Case-End-Paragraph.**

**Figure 2.5.3-10**

- (17) When a module is invoked via a PERFORM statement, represent the parameter table shown on the Structure Chart with comment lines. In the following figure, Parm-3 is both input to and output from Module-X.

***COBOL Example - PERFORM Statement,***

	<b>PERFORM</b>	<b>MODULE-X.</b>
*	<b>** USING:</b>	<b>Parm-1,Parm-2,Parm-3</b>
*	<b>** GIVING:</b>	<b>Parm-3,Parm-4,Parm-5</b>

**Figure 2.5.3-11**

- (18) When a module is invoked via a CALL statement, do not list the USING phrase as a comment line as it is part of the syntax. Group all parameters shown on the Structure Chart Diagram that are passed between the main program and the called module so that all of the input parameters precede the output parameters. Represent the GIVING phrase as a comment line and identify the output parameters (since COBOL does not make the distinction between input and output parameters). In the following example Parm-1 through Parm-5 are listed in the USING phrase, but not as a comment line. Any output parameter



that is input to the module, such as Parm-3 illustrated in the following figure, is listed in the GIVING as a comment (so that it is not listed twice in the program code).

**COBOL Example - CALL Statement**

	<b>CALL</b>	<b>Module-X</b>
	<b>USING</b>	<b>Parm-1,Parm-2,Parm-3</b>
*	**	<b>GIVING</b>
		<b>Parm-3,</b>
		<b>Parm-4,Parm-5.</b>

**Figure 2.5.3-12**

(19) The PERFORM verb has 5 acceptable formats:

- PERFORM Paragraph-Name.--This format is used with USING and GIVING comment statements to implement a module call, or used without the comments to PERFORM paragraphs within a module (e.g., nested IFs, or the body of the PERFORM-UNTIL structure).
- PERFORM Paragraph-Name UNTIL Terminating--Condition.
- PERFORM Line-Spacing-Paragraph Line-Count TIMES--This option executes a procedure a set number of times.
- The PERFORM-UNTIL may also be used to vary a subscript or index as in a table-search routine. See the following figure.

**COBOL Example - PERFORM Verb and Acceptable Formats**

<b>PERFORM</b>	<b>Table-Search</b>
<b>VARYING</b>	<b>Table-Index</b>
<b>FROM 1 BY 1</b>	
<b>UNTIL</b>	<b>Match-Found</b>
<b>OR</b>	<b>Table-Index GREATER THAN Max-Entries.</b>

**Figure 2.5.3-13**

(20) Implement the DO-UNTIL structure in one of two ways. The first way is a PERFORM/PERFORM-UNTIL combination. See

**COBOL DO-UNTIL Example 1**

<b>PERFORM</b>	<b>Paragraph-Name.</b>
<b>PERFORM</b>	<b>Paragraph-Name</b>
<b>UNTIL</b>	<b>Terminating-Condition.</b>

**Figure 2.5.3-14**

(21) The second way to implement a DO-UNTIL structure is to use a switch to terminate the loop. See Figure 2.5.3-15

**COBOL DO-UNTIL Example 2**

```

      MOVE True to Loop-Predicate.
      PERFORM Paragraph-Name
            UNTIL Loop-Predicate = False.

      ...
      Paragraph-Name.
      . . . Statements . . .

            IF Terminating-Condition
*
                        THEN
                                Move False to
                                Loop-Predicate.
*
                        END-IF

```

**Figure 2.5.3-15**

- (22) The following figure illustrates the format of the IF-THEN-ELSE statement.

**COBOL Example - IF-THEN-ELSE Statement**

	<b>IF Condition</b>		
*		<b>THEN</b>	
			<b>True-Procedures</b>
		<b>ELSE</b>	
			<b>False-Procedures.</b>
*	<b>END-IF</b>		

**Figure 2.5.3-16**

- (23) The ELSE part of the IF statement is optional when there are no actions to be taken (i.e., "ELSE NEXT SENTENCE" is not required). The THEN and END-IF comments are required. The True-Procedure and False-Procedure statements are indented 2 spaces from their corresponding THEN or ELSE. The IF and the corresponding END-IF keywords start in the same column. The THEN and ELSE keywords are indented 2 spaces in from the IF. As a guideline, the positive condition (rather than the negative) should be tested in a conditional statement. In a compound conditional statement, negative and positive tests should not be mixed.
- (24) When there are compound conditions associated with an IF statement, ensure that the statement is as readable as possible. The best method of doing this depends on the particular condition. The following figures illustrate the two formats.

**COBOL Example 1, IF-THEN-ELSE Statement Based on Conditions**

Format 1 – Putting each condition on a separate line:			
	IF Condition-1		
		OR Condition-2	
*		THEN	
			True-Procedures
		ELSE	
			False-Procedures.
*	END-IF		

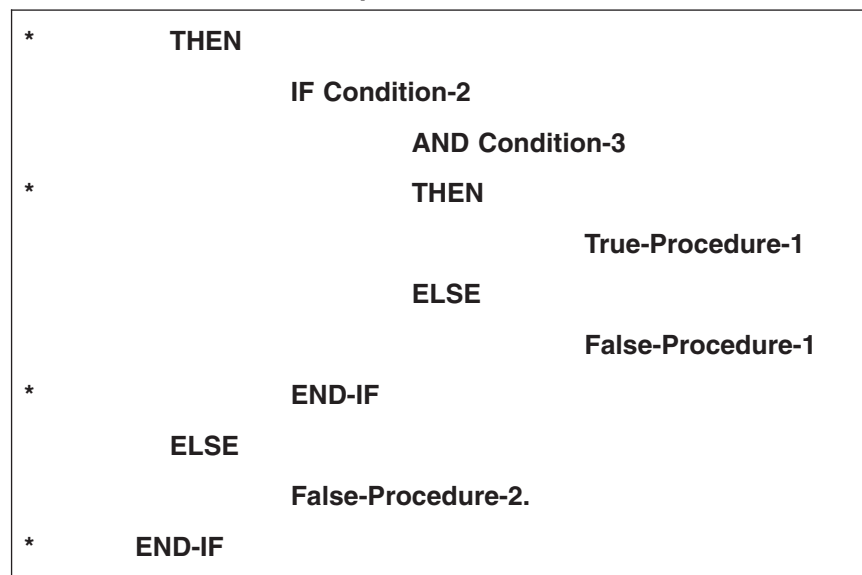
Figure 2.5.3-17

**COBOL Example 2, IF-THEN-ELSE Statement Based on Conditions**

Format 2 – Using parentheses to specify the order of evaluation for the individual conditions of more complex conditions:			
	IF ((Condition-1) OR (Condition-2))		
		AND Condition-3	
*		THEN	
			True-Procedures
		ELSE	
			False-Procedures.
*	END-IF		

Figure 2.5.3-18

- (25) Do not nest IF statements more than 3 levels deep. See the following figure.

**COBOL Example - Nested IF Statements****Figure 2.5.3-19**

- (26) If it appears that the nesting has to be more than 3 levels deep or even if the statement looks “cluttered” at 2 or 3 levels then PERFORM the inner test conditions. See the following figure.

**COBOL Example - PERFORM Inner Test on Nested IF Statements**

	Inner Test	on Nested IF Statements
<b>IF Condition-1</b>		
*	<b>THEN</b>	
	<b>PERFORM Inner-Test</b>	
	<b>ELSE</b>	
	<b>False-Procedure-2.</b>	
*	<b>END-IF</b>	
		.
		.
<b>Inner-Test.</b>		
<b>IF Condition-2</b>		
	<b>AND</b>	
*	<b>Condition-3</b>	
*	<b>THEN</b>	
		<b>True-Procedure-1</b>
	<b>ELSE</b>	
		<b>False-Procedure-1.</b>
*	<b>END-IF</b>	

**Figure 2.5.3-20**

- (27) Implement the “Nested IF” of the SELECT-CASE construct as prescribed in the following figure. Note that this format is different from a normal IF-THEN-ELSE statement.

***COBOL Example - Using SELECT-CASE for  
Nested IF-THEN ELSE Statements***

```
*      SELECT CASE.  
      IF Condition-1  
*      CASE-1:  
          Case-1-Statements  
      ELSE  
      IF Condition-2  
*      CASE-2:  
          Case-2-Statements  
      ELSE  
      IF Condition-3  
*      CASE-3:  
          Case-3-Statements  
      ELSE  
*      Error-CASE:  
          Error-Case-Statements.  
*      ENDCASE
```

**Figure 2.5.3-21**

- (28) Use the EVALUATE statement instead of long nested IF statements to test several conditions and specify different actions for each. The WHEN phrases determine selection. Case statements usually consist of the following commands; MOVE, ADD, PERFORM, etc..

**COBOL Example - EVALUATE Statement**

<b>EVALUATE STATEMENT</b>	
EVALUATE X	
WHEN 'A'	CASE 1 Statements (MOVE, ADD, PERFORM)
WHEN 'B'	CASE 2 Statements (MOVE, ADD, PERFORM)
WHEN OTHER'	Error-Case-Statements
END-EVALUATE:	
Using ALSO and WHEN:	WHEN (Age < 16 ALSO Gender = 'M')
Using AND	WHEN (Age < 16 AND Age > 13)

**Figure 2.5.3-22**

2.5.3.4.4  
(07-10-2020)  
**COBOL Compile  
Run-Time Warning  
Messages**

- (1) Currently IRS applications hosted on IRM Mainframes, and have migrated to Enterprise COBOL compiler version 6.2. When COBOL programs are written they have to be compiled into object-code from source-code in order to be read by the computer. Some compiler warnings are more severe than others. The following warnings messages identified must be cleared before source-code can be moved into production:
  - Warning messages like: **IGZ0279W, IGZ0316W and IGZ0318W** - will display for new COBOL compiler with the first two characters UL or UO (PROCGRP) with (e.g. **UL2NCL vs. DB2NCL, ULNBL vs NCNB** ), (ADD INITCHECK?)
  - Example Warning message - IGZ0279W The value data-item-value of data item data-name at the time of reference by statement number verb-number on line line-number in program program-name failed the NUMERIC class test or contained a value larger than the PICTURE clause as detected by the NUMCHECK compiler option. See exhibits (a-f) for more examples.

2.5.3.5  
(07-10-2020)  
**C Programming**

- (1) This section of the IRM provides guidelines for coding C programs and naming C program components.

2.5.3.5.1  
(07-10-2020)  
**C File Naming**

- (1) Create the file names from a base name and an optional period and suffix.
- (2) Store very large files by date (for archive or delete). Make the date a part of the name (e.g. log files).
- (3) Make the first character of the name a letter.
- (4) Assign a file name that is unique in as large a context as possible.
- (5) Use uppercase and lowercase letters to name source code files like "percentOfLargest" or "PercentOfLargest".
- (6) Include comments in the module so other programmers will understand the modules' purpose (ie., Title Section).
- (7) Use System Name followed by file name. For example, the application system is Telefile (or EMS, TEPS, EFDS, EFTPS, etc.) and the file name is Return-Data.
- (8) Include comments on the name
- (9) Maintain a consistent File Naming Convention (FNC) by referencing IRM 2.152.3 IT, Data Engineering, Naming Data Elements/Object(s) as a guide.

2.5.3.5.2  
(01-01-2004)  
**C Source Code Files**

- (1) Size Considerations:
  1. Limit the size of a source code file to 1000 lines as large source code files can be very cumbersome.
  2. Per each line in a source code file, limit the number of characters per line to 163 or fewer characters.
  3. Decompose long lines into smaller pieces, such that when the file is printed, all portions of the code will print out legibly.
  4. Indent subsequent sections of a longer line so that it is clear that these are continuations of the line above.
  5. In the length of a line, include any commentary that follows the code on the line.
  6. Where a function exceeds two pages, reexamine the design of the function.
  7. Especially consider if more than one function is involved or if sub-functions would be better in separate modules.
  8. If functions are short and related to each other, then place them in same source code file.
- (2) Composition:
  1. Prologue
  2. Includes
  3. Defines and Typedefs
  4. Global Definitions
  5. Function Placement

2.5.3.5.2.1  
(07-10-2020)  
**C Prologue**

- (1) Make the prologue first in the file as it indicates what is in that file.



- (2) Use a description of the purpose of the objects in the files (whether they be functions, external data declarations or definitions or something else) rather than a list of the object names. A description of the method(s) used is helpful for any complex function.
- (3) Avoid making descriptions so detailed that maintenance of the header takes more effort than is gained by increased understanding of the code itself.

#### 2.5.3.5.2.2 (07-10-2020) **C Includes**

- (1) Header files are files that are included in other files prior to compilation by the C preprocessor.
- (2) Place the header file after the prologue. If the include line is for a non-obvious reason, comment the reason. In most cases, application system includes files like "stdio.h" should be included before user include files.

#### 2.5.3.5.2.2.1 (07-10-2020) **C Header File Organization**

- (1) Relate all functions in a given header file to the same general function, i.e., declarations for separate sub-systems should be in separate header files. Example: all functions in the header file "stdio.h" either perform or assist in the performance of input and output.
- (2) Do not use any function implementations, except macros in Header files.
- (3) Within a header file, group functions that perform related tasks in the same section. Example: within the file "stdio.h", all functions in the scanf family must be placed together.
- (4) Define and include certain header files, such as "stdio.h" at the application system level and for any program using the standard I/O library.
- (5) Use Header files to contain data declarations and defines that are needed by more than one program.
- (6) Organize header files functionally, i.e., declarations for separate subsystems should be in separate header files.
- (7) If a set of declarations is likely to change when code is ported from one machine to another, place those declarations in a separate header file.
- (8) Use "<>" "<stdio.h>" for system include files and double quotes ("user.h") for user include files.

#### 2.5.3.5.2.2.2 (07-10-2020) **C Header File Inclusion in the File that defines the Function**

- (1) Include Header files that declare functions or external variables in the file that defines the function or variable. This allows the compiler to do type checking and the external declaration will always agree with the definition.
- (2) To prevent accidental double-inclusion, in each .h file, use code like the following.

#### ***C Example - Header File Inclusion for Functions***

Header File Inclusion
<pre>#ifndef EXAMPLE: #define EXAMPLE ... /* body of example.h file */</pre>

**Header File Inclusion**

```
#end-if /* EXAMPLE */
```

- (3) Use a general-purpose header file for commonly used symbolic constants.

2.5.3.5.2.2.3  
(07-10-2020)

**C Nested Header Files**

- (1) Do not nest Header files. The prologue for a header file must describe what other headers need to be included for the header to be functional.
- (2) Where a large number of header files are to be included in several different source files, put all common include statements in one include file.

2.5.3.5.2.2.4  
(07-10-2020)

**C Header File Names**

- (1) Avoid private header filenames that are the same as public header filenames. The statement `#include "math.h"` must include the standard library math header file if the intended one is not found in the current directory. If this is what you want to happen, comment this fact.
- (2) Don't use absolute pathnames for header files. Use the `<name>` construction for getting them from a standard place, or define them relative to the current directory. Use the "include-path" option of the C compiler (`-I` on many application systems) used in the Makefile to handle extensive private libraries of header files; it permits reorganizing the directory structure without having to alter source files.

2.5.3.5.2.3  
(07-10-2020)

**C Defines and Typedefs**

- (1) Place the defines and typedefs that apply to the file as a whole after the includes.
- (2) Place a Define before the header files so that they will apply to the header files.
- (3) Place "constant" macros first, then "function" macros, then typedefs and enums.

2.5.3.5.2.4  
(07-10-2020)

**C Global Definitions**

- (1) Place the global (external) data declarations after the Defines/Typedefs.
- (2) Use the order:
  - a. Externs
  - b. Non-static globals
  - c. Static globals
- (3) Place the defines immediately after the data declaration or embedded in structure declarations when a set of defines applies to a particular piece of global data (such as a flags word). Ensure defines are indented to allow one level deeper than the first keyword of the declaration to which they apply.

2.5.3.5.2.5  
(07-10-2020)

**C Function Placement**

- (1) Place the functions last.
- (2) Place like functions together.
- (3) Use a "breadth-first" approach (functions on a similar level of abstraction together) rather than depth-first (functions defined as soon as possible before or after their calls).

- (4) Use alphabetical order when defining large numbers of independent utility functions.

#### 2.5.3.5.3 (07-10-2020)

##### **C Other Files**

- (1) For operator-directed programs, establish a file called “Readme” to document both the file and issues for the program or a group of programs. For example, it is common to include a list of all conditional compilation flags and what they mean.
- (2) List files that are machine dependent, etc.

#### 2.5.3.5.4 (07-10-2020)

##### **C Global Variable Declarations**

- (1) The following subsections address global variable and structure declarations.

#### 2.5.3.5.4.1 (07-10-2020)

##### **C Global Variables**

- (1) Avoid the use of global variables unless you have cases where the use of global variables can actually make a program more readable by not cluttering function calls. Instead, pass variables by reference to functions that change their value.
- (2) Declare any global variables at the top of a file, before any function declarations.
- (3) Declare variables, which are global to only the functions in a single file, as “static”.
- (4) Use meaningful names (MaxLength=30 characters).
- (5) Separate the words of a compound variable by capitalizing the first letter of every word.
- (6) Start pointer names with “p”.
- (7) Separate unrelated declarations, even of the same type, on separate lines.
- (8) Tab the names, values, and comments so that they line up. See the following table.

##### ***C Example - Using Global Variables***

int	EventInit;	/* event_init performed */
char	TaxFormType;	/* type of the tax return form /
char	*pFirstEntry;	/* ptr to 1st entry */

#### 2.5.3.5.4.2 (07-10-2020)

##### **C Structure Declaration**

- (1) Declare each field in a structure on a separate line.
- (2) If you declare a local structure – use lower case for the names. If you declare a global structure – use mixed case for the names. The variables that comprise the structure (structure elements) must follow the same rule as local or global variables
- (3) Assign a structure to a variable in a separate statement.

## (4) Recommended Styles:

1. The following table illustrates a style where the opening brace ({} should be in column 1 on a next line after the structure tag, and the closing brace (}) should be in column 1.

**C Example 1 - Recommended Syntax Structure**

<b>struct ECT_REG_HEADER_S</b>
{
char *pFirstEntry; /* ptr to 1st registry entry */
int NumEntries; /* number of entries */
}

- 2) The following table illustrates a style where the opening brace ({} should be on the same line as the structure tag, and the closing brace (}) should be in column 1. Choose one of these styles for the opening brace ({} and consistently use it.

**C Example 2- Recommended Syntax Structure**

<b>struct ECT_REG_HEADER_S {</b>
char *pFirstEntry; /* ptr to 1st registry entry */
int NumEntries; /* number of entries */
}

2.5.3.5.4.3  
(07-10-2020)**C Typedef Declaration**

- (1) Typedef which stands for “type definition” is a reserved keyword in C and C++, and allows the programmer to create an alias that can be used anywhere in place of a complex type name.
  - Use in C language to assign an alternative name to existing datatypes
- (2) Structures may use Typedef keyword when they are declared. See example below

**C Example of Typedef**

<b>Reserved Keyword (Typedef)</b>
<b>typedef</b> <existing_name> <alias_name>

2.5.3.5.5  
(07-10-2020)**C Local Variable Declarations**

- (1) Do not use names with leading and trailing underscores for any user-created names as they are reserved for application system purposes. Most application systems use them for names that the user should not have to know.

## 2.5.3.5.5.1

(07-10-2020)

### C Local Variable Names

- (1) The following paragraphs cite guidelines.
- (2) Declare local variables at the start of, or just before, the block in which they are used. If the variable name is going to be reused in a different block in the same function then declare the variable at the start of the function.
- (3) Do not have a function contain two variables with the same name.
- (4) Avoid declaring variables within any block, (e.g., within a “for” block).
- (5) Use meaningful names (max\_length=30 characters).
- (6) Begin all variable names with a lowercase letter.
- (7) Place the pointer qualifier, \* with the variable name rather than with the type.
- (8) Separate unrelated declarations, even of the same type, on separate lines.
- (9) Include a comment describing the variable in the same line.
- (10) Tab the names, values, and comments so that they line up. An example follows.

#### C Example - Local Variable Names

	Local Variables Names	
int	event_init;	/* event_init performed */
Char	tax_form_type;	/* type of the tax return form */
Char	*first_entry_p;	/* ptr to 1st entry */

## 2.5.3.5.5.2

(07-10-2020)

### C Typedef Declaration

- (1) Give the struct and typedef the same name. Apply the same rules as for a structure declaration.

## 2.5.3.5.5.3

(07-10-2020)

### C Abbreviations for Common Variable

- (1) Use conventional abbreviations for common variables.

#### C Example - Common Variable Abbreviations

Variables	Abbreviations
average	avg
database	db
Length	len
message	msg
number	num
position	pos
String	str

2.5.3.5.6  
(07-10-2020)  
**C Constants**

- (1) This subsection addresses constants and enumeration data.

2.5.3.5.6.1  
(07-10-2020)  
**C Defining Constants**

- (1) Avoid coding numerical constants directly.
- (2) Declare numerical constants to facilitate changes when it is used throughout the whole program; and use actual numbers in small scope code.
- (3) Use symbolic constants to make the code easier to read.
- (4) Define the value in one place to make it easier to administer large programs since the constant value can be changed uniformly by changing only the define.

2.5.3.5.6.2  
(07-10-2020)  
**C Consistency of Constant Definitions**

- (1) Consistently define constants with their use, (e.g., use 540.0 for a double instead of 540 with an implicit float cast).

2.5.3.5.6.3  
(07-10-2020)  
**C Conventional Constants**

- (1) Use a conventional set of symbolic constants for constants common to C coding:

***C Example - Using Conventional Constants***

Conventional Set of	Symbolic Constants
TRUE	Defined by system supplier headers. Do not redefine.
FALSE	Defined by system supplier headers. Do not redefine.
#define FALSE 0	Defined by system supplier headers. Do not redefine.
#define TRUE !FALSE	Defined by system supplier headers. Do not redefine.
LF_CHAR	Line feed character
CR_CHAR	Carriage return character
EOS	End of string character
EOF	Defined by system supplier headers. Do not redefine.

2.5.3.5.6.4  
(07-10-2020)  
**C Enumeration Data**

- (1) Use the enumeration data type to declare variables that take on only a discrete set of values, since additional type checking is often available.
- (2) Declare each field in an enum on a separate line.
- (3) Ensure the enum type name has a tag, in upper case with “\_E” appended to their name.

- (4) Ensure each field in an enum type is in upper case separated by underscores.

## **C Example - Enumeration Data**

Example of Enumeration Data
enum ACMW_E
{
ACMW_IB_TP_INTERFACE = 5001,
ACMW_IB_USER_VALIDATION = 5002
}

2.5.3.5.6.5  
(07-10-2020)

## **C Symbolic Constants - #define**

- (1) Name all quantities that must remain unchanged throughout a program using the “#define” capability. The defined name must be in upper case letters.

## **C Example - Symbolic Constants**

Symbolic	Constants
#define FEP_FAILURE	“FEP ACKNOWLEDGMENT_FAILURE”
#define AT_AUDIT_NAME	“7”

2.5.3.5.7  
(07-10-2020)

## **C Functions**

- (1) This subsection addresses:

- return values
- parameter lists
- function body
- function prototype
- function naming

2.5.3.5.7.1  
(07-10-2020)

## **C Return Values**

- Explicitly declare all return values.
- Do not default to *int*; if the function does not return a value then give it return type *void*.
- If the value returned requires a long explanation, give it in the prologue.

2.5.3.5.7.2  
(07-10-2020)

## **C Parameter Lists**

- If the function and its parameter list is longer than one line, indent lines after the first one from the left margin so that the second line of the parameter list starts directly below where the parameter list begins on the first line.
- Ensure a function that returns information via one or more of its parameters only returns status information in its name.
- Ensure each parameter passed to a function occurs on a separate line in the function prologue with a short comment describing its function.

**2.5.3.5.7.3**  
(07-10-2020)**C Function Body**

- (1) Tab all local variable declarations and code within the body over one stop.
- (2) Place the opening brace, "{", opening the body of the function on a line by itself and left justified or at the end of the line which introduces the block. Any control statements will cause further indentation from this basic indentation.
- (3) If the function uses any external variables (or functions) that are not declared globally in the file, provide the declarations in the function body using the *extern* keyword.
- (4) Avoid local declarations that override declarations at higher levels.
- (5) Redeclare local variables in nested blocks.
- (6) Ensure that a single return statement is always present with a parameter if the function is not of type void. Use one even if the function has no return value and, therefore the C language does not require a return. This can be useful for setting a break point during debugging.
- (7) Place a closing brace, "}", closing the body of the function, on a line by itself and left justified.

**2.5.3.5.7.4**  
(07-10-2020)**C Function Prototype**

- (1) Generate Function Prototype for all functions generated.

**2.5.3.5.7.5**  
(07-10-2020)**C Function Naming**

- (1) Select a naming convention (capitalization, underscores, etc.) and use it consistently.
- (2) For naming services:
  - Limit service name to 12 characters. Longer names, when accepted, by Tuxedo, will be truncated to 12 characters.
  - Begin all service names with an uppercase letter.
  - Separate the words of compound service names by capitalizing the first letter of every word.

**2.5.3.5.8**  
(07-10-2020)**C Comments**

- (1) Use comments in your programs to meet the following three (3) goals:
  - a. Clear and concise
  - b. Place where needed
  - c. Useful to read the code
- (2) Strive to balance the amount of comments with the amount of whitespace to maintain readability and clarity.
- (3) Make sure the comments describe what is happening, how it is being done, what parameters mean, which globals are used and which are modified, and any restrictions or bugs.
- (4) Avoid comments that are clear from the code. Such information rapidly gets out of date, is redundant and clutters the code.



2.5.3.5.8.1

(07-10-2020)

## C Template for File and Header

- (1) Place the following header at the beginning of the file:

### *C Example - Header in a File*

```

/*****
//
// Internal
// Revenue Service
// For Official Use
// Only
//
//
// Filename:      Filename
// Description:   Describe the purpose of the objects in the file,
//               followed, in the case of source files, by a list of
//               functions whose definitions appear in the file
// Related Files: An identification of any routines or files that this
//               file may require
// Restrictions/  Known special cases where the file may not work
// Problems:
//
// Date Modified: Date: YYYY/MM/DD
// Version id:   Revision:
// Author:       Author: <First Name> <Last Name>
// Locked by:    $Locker: $
//
// Revision
// History:      To clearly identify all the changes, when doing
//               code reviews, print out the information. Look at
//               ClearCase to identify the revision date. If enough
//               information is there, look at the header for
//               specifics. Not putting it in the code reduces your
//               options to hoping the ClearCase information is
//               sufficient, and doing a line-by-line review in
//               ClearCase until you find the change.

```

- (2) Place the following header at the beginning of the function:

### *C Example - Header at Beginning of Function*

```

/*****
* Function Name:

```

```

* Description:          A description of the major task(s) performed
                        by
*
                        routine. It should be a series of one or more
                        simple
*
                        verb/object statements
* Input parameters:
* Output parameters
*****/

```

2.5.3.5.8.2  
(07-10-2020)

#### Function Comments

- (1) Place comments that describe data structures, algorithms, etc., in block comment form.
- (2) Present code in paragraph form prior to a block of code.
- (3) Use comments for cohesive blocks of code when they explain the purpose of the block in accomplishing a cohesive task. This enables the reader to understand the function being implemented. Thus, the reader will be able to quickly find the appropriate section of code without getting bogged down in coding details for other sections of code. For example:

#### *C Example 1- Function Comments*

```

/*****
*      LOCAL VARIABLES and CONSTANTS
*****/

/*****
*      cleanup_pipeline PROCESSING
*****/

```

- (4) Indent block comments to the same level as the block being described.
- (5) Indent one-line comments alone on a line to the tab setting of the code that follows. For example:

#### *C Example 2 - Function Comments*

```

Function Comments
if (argc > 1)
{
    /* Get input file from command line. */
    if (freopen(argv[1], "r", stdin) == NULL)
    {
        perror (argv[1]);
    }
}

```

**Function Comments**

```

        }
    }

```

- (6) Very short comments may appear on the same line as the code they describe, but must be tabbed over to separate them from the statements.
- (7) If more than one short comment appears in a block of code, they must be tabbed to the same tab setting. For example:

***C Example 3 - Use of Multiple Short Function Comments***

```

if (a == EXCEPTION)
{
    b = TRUE;                /* special case */
}
else
{
    b = isprime(a);          /* works only for odd a
                             */
}

```

2.5.3.5.9  
(07-10-2020)  
**C Statements**

- (1) This subsection addresses the following topics related to statements:
  - a. Statements per Line
  - b. Single Statement Blocks
  - c. Multiple Statement Blocks
  - d. Levels of Control Structure Nesting
  - e. Goto Statement
  - f. Break Statement
  - g. Null Statement
  - h. Conditional Statement
  - i. Exit Statement
  - j. Default Truth Value
  - k. Increment and Decrement Operators
  - l. Added Statements for Debugging

2.5.3.5.9.1  
(07-10-2020)  
**C Statements per Line**

- (1) Generally, source code should depict one statement per line.

2.5.3.5.9.2  
(07-10-2020)  
**C Single Statement Blocks**

- (1) Block off even a single statement following a “while”, “if”, “else”, etc.

- (2) Using the curly braces “{}” is required only when there is a block of more than one statement. However, putting in the braces makes the scope of the control statement very clear and helps to protect the code in the event that a second line is added to the block if the single line contains a macro, which translates into more than one line of code.

***C Example - Single Statement Blocks***

**Single Statement Blocks**

```
if (SomeCondition == TRUE)
{
    ThisVariable = SomeVariable;
}
```

2.5.3.5.9.3  
(07-10-2020)

**C Multiple Statement Blocks**

- (1) Statements that affect a block of code (i.e., more than one statement) must either have the opening brace “{” at the end of the line containing the control statement, or the opening brace must be on the line immediately below and lined up with the first letter of the control statement.
- (2) Indent the body of the block one step from the control statement.
- (3) Place the ending brace, “}” on a line by itself and at the same indentation level as the control statement.
- (4) Choose one of the two styles and use it consistently:

***C Example 1 - Multiple Statement Blocks***

**Multiple Statement Blocks**

**1st Style:**

```
If (condition)
{
    statements(s)
}
else
    if (condition)
    {
        statements(s)
    }
for (loop control expressions)
{
    statements(s)
}
```

**Multiple Statement Blocks**

```
while (condition)
{
    statements(s)
}

switch (expression)
{
    case constant1;
                                                statement(s)

    case constant2;
                                                statement(s)

    default;
                                                statement(s)
}
```

***C Example 2 - Multiple Statement Blocks*****2nd Style Multiple Statement Blocks**

```
if (condition) {
    statement(s)
} else if (condition) {
    statement(s)
}
```

```
for (loop control expressions) {
    statement(s)
}
```

```
while *condition) {
    statement(s)
}
```

**2nd Style Multiple Statement Blocks**

```

switch (expression) {
    case constant1;
        statement(s)
    case constant2;
        statement(s)
    default;
        statement(s)
}

```

2.5.3.5.9.4  
(07-10-2020)

**C Levels of Control Structure Nesting**

- (1) Do not nest conditional statements, such as “while”, “if”, “else”, more than 4 levels. If more levels are required, consider using a function at one of the higher levels.

2.5.3.5.9.5  
(07-10-2020)

**C Goto Statement**

- (1) Do not use the Goto statement.

2.5.3.5.9.6  
(07-10-2020)

**C Break Statement**

- (1) If a particular case in a switch statement is meant to drop through to the next case (i.e., it has the same effect), the fact that the earlier case has no “break” statement must be explicitly noted with a comment. For example:

**C Example - Switch & Break Statement****Switch & Break Statement**

```

switch(switch_form)
{
    case P1040 : /* same action as for 1065; no break */
    case P1065:
        process_1040();
        break;
    default : /* if not 1040 or 1065, no action taken */
        break;
}

```

2.5.3.5.9.7  
(07-10-2020)

**C Null Statement**

- (1) Generally, Null statements should include a comment line.

- (2) Place the null body of a “for” or “while” loop alone on a line and commented so that it is clear that the null body is intentional and not missing code.

## ***C Example - Null Statement***

### **C Null Statement**

```
while (*dest++ = *src++)
    ; /* VOID */
```

## 2.5.3.5.9.8 (07-10-2020) **Conditional Statement**

- (1) Break out the function call onto a separate line followed by a new line containing the conditional statement. Often a program will branch based on the success or failure of a function call. Consider the following excerpts of source code, the first excerpt is easier to understand than the second excerpt.

## ***C Example 1 - Conditional Statement***

### **Conditional Statement**

```
pFileHandle = fopen("some_file ", READ_ONLY);
if (pFileHandle == NULL)
{
    printf("Could not open file; program terminating.");
    TerminateApplication();
}
else
{
    DoSomething();
}
```

## ***C Example 2- Conditional Statement***

### **Conditional Statement**

```
if ((pFileHandle = open(" some_file", READ_ONLY))== NULL)
{
    printf("Could not open file; program terminating.");
    TerminateApplication();
}
Else
{
    DoSomething();
}
```

- (2) The preceding “fopen” example demonstrates a style, which helps to avoid internal side effects. “Side effect” as used in this example refers to the fact that the reader may focus on the “if (xxx == NULL)” aspect of the statement and not fully realize that there is a call to “fopen()”. The other danger of this type of compound statement is the potential for completely changing the meaning if the parentheses around the “pFileHandle = open()” part are left off.

2.5.3.5.9.9  
(01-01-2004)

#### **C Exit Statement**

- (1) Except for use in error-handling functions, avoid explicit use of the “exit();” statement.

2.5.3.5.9.10  
(01-01-2004)

#### **C Default Truth Value**

- (1) Use explicit comparison even if the comparison value will never change.

2.5.3.5.9.11  
(07-10-2020)

#### **C - Added Statements for Debugging**

- (1) If you include statements to print out information during debugging, use preprocessor switch(es) in the makefile to allow compile-time control of the debug output, and use #ifdef statements to control inclusion of the debug statements. For example:

#### ***C Example - Printing Out Statement for Debugging***

```
#ifdef DEBUG_1
printf(some debug statement);
Fprint(<pointer to FML buffer>);
#endif
```

2.5.3.5.10  
(01-01-2004)

#### **Operators**

- (1) Do not use the ternary conditional operator, “?:” in the main program, primarily to make the code more readable. It can still be used in macros.
- (2) An increment or decrement operation should be explicitly placed in a separate statement so it would be clear what is occurring and when.
- (3) Ensure that all operators which take two parameters have a single space on either side of the operator. This makes it very handy to use an editor to search for a variable assignment; you need only search for “a =” and not “a =” as well as “a= ”. It also makes the code more readable.
- (4) In contrast to binary operators, ensure that all unary operators (e.g., a minus sign or the address operator, “&”) have no space between the operator and the object.
- (5) Where operator precedence must be known to determine the meaning of an expression, use parentheses to eliminate any ambiguity, which might arise from lack of knowledge of operator precedence. For example, to increment the variable pointed to by the pointer “pNumTimes”, use “(\*pNumTimes)++”. This use of parentheses makes it clear that the contents of location “pNumTimes” is being incremented and not the address itself.
- (6) The ternary conditional operator may be useful in parameter lists and as a return value.



2.5.3.5.11  
(01-01-2004)  
**ESQL/C**

- (1) This subsection addresses:
  - a. Database Error Checks
  - b. Operations
  - c. Performance
  - d. SQL Statements

2.5.3.5.11.1  
(07-10-2020)  
**ESQL/C Database Error Checks**

- (1) Ensure that all calls to the database have error checking.
- (2) If a cursor is used, place an error check immediately after the declare, open, all fetches, and close cursor.
- (3) If a data retrieval error occurs, end the data retrieval processing and the variable that holds the unretrieved data will be populated with the default value for missing data as per the application program guidelines. A NULL value is not acceptable.

2.5.3.5.11.2  
(07-10-2020)  
**ESQL/C Operations**

- (1) Due to the constraint that not more than one database can be open at a time, always check to see if a database is open before there is a call to open one.
- (2) Before any program exits, close the open database.
- (3) Declare cursors in the same function they are used.
- (4) Close and free cursors after they are used.
- (5) If a SQL error occurs, log the message to display the SQL code, the function name, the file name, and the error message.

2.5.3.5.11.3  
(07-10-2020)  
**ESQL/C Performance**

- (1) Minimize overhead processing and optimize memory allocations.
- (2) Do not pass more than one variable or structure to an ESQL/C function.
- (3) If more than one data value needs to be populated, then create an array. Pass the pointer of the array to the ESQL/C function.
- (4) If more than one variable needs to be populated, then create a structure or a linked list and pass the structure or linked list pointer to the ESQL/C function.
- (5) Two calls must be made to retrieve data for arrays, structures, or linked lists. In the first call, there must be an ESQL/C function call that returns the number of values to be collected. The calling function will then allocate memory in the array or structure to contain the desired data.
- (6) In the second call, the data must be populated into the array, structure, or linked list.

2.5.3.5.11.4  
(07-10-2020)  
**SQL Statements**

- (1) Test SQL statements individually before embedding them in C code. This approach will increase the likelihood that the embedded code will work when properly incorporating the tested SQL statements. This should be a fast risk-reducing activity and a great enhancement.
- (2) Capitalize all reserved words, (i.e., SELECT, UPDATE, INSERT, etc.).
- (3) Use ORDER BY only when absolutely needed – if the calling program does not require sorted data, do not use ORDER BY.

- (4) For negative nested subqueries (selecting rows from a table where some condition in another table is not true), experiment with both the NOT EXISTS and NOT IN constructs to determine which is faster in your situation.
- (5) Avoid usage of “OR”.
- (6) For discrete lists of values, use the “IN” operator, ( e.g. “WHERE city in (‘New York’, ‘Sydney’)”) instead of “WHERE city = ‘New York’ OR city = ‘Sydney’”.
- (7) The SELECT statement follows these rules:
  - Keywords are left justified.
  - Alias table names will be used to prefix every selected column.
  - In the WHERE clause, list join expressions before restrictions.
- (8) Where the select statement does not fit on a single line, align the columns from the various lines as in the following excerpt of code.

**SQL Example - SELECT Statement**

```
SELECT e.emp_id, e.emp_nm, e.city_nm, d.dept_nm
FROM emp e, dept d
WHERE e.dept_id = d.dept_id;
```

- (9) The UPDATE statement follows these rules:
  - Keywords are left justified.
  - First line is “UPDATE tablename”
  - Next lines specify updated columns and their values
  - Final lines are the WHERE clause
- (10) The following excerpt of source code exhibits the above rules:

**SQL Example - Update Statement**

```
UPDATE emp
SET job_desc = V/,
City_nm = 'Sterling'
WHERE emp_nm = 'JOE';
```

- (11) The DELETE statement follows these rules:
  - Keywords are left justified
  - First line is “DELETE FROM tablename”
- (12) The following excerpt of source code table exhibits the above rules:

**SQL Example - DELETE Statement**

```
DELETE FROM emp
WHERE city_nm = 'Sterling';
```

- (13) User parentheses rather than the implicit SQL precedence rules for logical operators in a WHERE clause. Consider the following excerpts of source code, the first excerpt is incorrect, the second excerpt is correct.

## **SQL Example - Incorrect Use of WHERE Clause**

**Incorrect Use of Where clause**

```
SELECT COUNT(*)
FROM emp
WHERE job_desc LIKE 'widget%'
AND name LIKE 'J%'
OR city = 'Sterling';
```

## **SQL Example - correct Use of WHERE Clause**

**Correct Use of Where clause**

```
SELECT COUNT(*)
FROM emp
WHERE (job_desc LIKE 'widget%' AND name LIKE 'J%')
OR city = 'Sterling';
```

- (14) For further information refer to the Enterprise Standards Profile (ESP) Attachment 1 Enterprise Data Standards and Guidelines.

### 2.5.3.5.12 (07-10-2020) **Whitespace**

- (1) Use vertical and horizontal whitespace judiciously to make the program more readable.
- (2) Ensure that indentation and spacing reflect the block structure of the code.

### 2.5.3.5.12.1 (07-10-2020) **Vertical Spacing of Conditional Operators on Separate Lines**

- (1) Split a long string of conditional operators onto separate lines. For example, consider the following excerpt.

## **C Example - Conditional Operators' Spacing**

Example of	Conditional Operators' Spacing
if (foo->next == NULL	
	&& total_count < needed
	&& needed <= MAXLLOT
	&& ServerActive(current -input))
{	
	...
}	

- (2) Similarly, split elaborate “for” loops onto different lines.

***C Example - For Loops***

**Example For Loops**

```
for (curr = *listp, trail = pList;
    curr != NULL;
    trail = &(curr->next), curr = curr->next) {
    DoSomething();
    DoAnotherSomething();
}
```

2.5.3.5.12.2  
(01-01-2004)  
**C Spacing for  
Parentheses**

- (1) Do not separate keywords that are followed by expressions in parentheses from the left parenthesis.
- (2) Put blanks after commas in argument lists to help separate the arguments visually.

2.5.3.5.13  
(07-10-2020)  
**C Portability**

- (1) Portability for high-level programming represents the usability of the same software in different computer environments. The only changes would be the inclusion of possibly different header files, and the use of different compiler flags. The header files will contain #defines and typedefs that may vary from machine to machine.
- (2) For an application to be considered portable it must have the capability of moving across environments, not just across platforms. This environment can consist of: new machine/hardware, different operating system, different compiler, software interfaces, or any combination of these.
- (3) Writing code in C does not guarantee portability. The desired target platform must have a working C compiler available. To avoid writing C code that is non-portable be aware of the following:
- Avoid making assumptions about integers and pointers.
  - Avoid making assumptions about sizes of any data types, other than character.
  - Avoid making assumptions about how data is aligned in memory, and within structure.
  - Avoid making assumptions about how data is packed within structure.
  - Avoid making assumptions about byte ordering.
  - Avoid making assumptions about the size of a pointer, or that the pointer is the same size as an int.
  - Avoid making assumptions that software will always be executed on the machine for which it is originally designed.
  - C randomly seems to work in one environment, but perform unreliably in another environment.
  - The size of different data types may vary from platform to platform.
  - C uses a non-standard compiler extension, and may not be available on all compiler implementations.

- C uses a non-standard library extension, which may not be available on all platforms.

2.5.3.5.13.1  
(07-10-2020)  
**C Machine-Dependent  
Code Placement**

- (1) The “#ifdef ”and “#ifndef” directives have the same effect as the “#if ”directive when it is used with the defined operator.
- (2) Place all machine-dependent code in a separate file from all machine-independent code.
- (3) Machine-dependent code must use “#ifdef” so that an informative error message will result if the code is compiled on a machine other than for which it is designed.

2.5.3.5.13.2  
(07-10-2020)  
**C Machine-Dependent  
Code Usage**

- (1) Only write machine-dependent code when necessary.
- (2) Even if, for example, a particular piece of hardware requires that a machine-dependent routine be written, try to write any routines that support the machine-dependent code machine-independent.

2.5.3.6  
(07-10-2020)  
**C++ Programming  
Overview**

- (1) C++ is a high performance Object Oriented programming language based on C. It was developed and released by Bjarne Stroustrup in 1985, and was first standardized in 1998. Standards were issued again during 2003, 2007 and 2011. C++ strives to be portable to avoid reliance on features that are platform-dependent and is maintained by the International Standards Organization (ISO) committee. C++ is widely used in embedded systems software engineering, and is also popular with industries: federal, health care, finance, and Defense.

2.5.3.6.1  
(07-10-2020)  
**C++ Scope**

- (1) This subsection provides establishes controls to ensure coding of C++ programs are reliable, maintainable and portable whether developed by IRS or outside vendors. These standards apply to all C++ programming for any project however, these standards and guidelines do not apply to source code that is generated by a tool (such as a Graphical User Interface (GUI) builder), or purchased as pre-existing software from a third party.

2.5.3.6.2  
(07-10-2020)  
**C++ Classes**

- (1) Classes are a very significant part of your C++ programs. Classes are where most of the processing in your programs occurs. Classes are also one of the C++ constructs where you have choices such as different types of constructors and destructors or different types of operators and assignments. The following standards will help you in writing good C++ class code.

2.5.3.6.2.1  
(07-10-2020)  
**C++ Class Declaration**

- (1) All classes must declare an assignment operator.
- (2) Headers files must not contain more than one class definition.
- (3) Header files must not declare variables other than class member data.
- (4) Class names must be unique irrespective of case in any namespace including the IRS global namespace.
- (5) Class member data must have a trailing “\_” appended to their variable name, to distinguish them from local variables within member functions. No other variables must be named with a trailing underscore.

(6) Class declarations must be made using the following order:

- a. Friend declarations
- b. Public members
- c. Protected members
- d. Private members

**C++ Example - Class Declaration**

**Example:of C++ Class Declaration**

```
class C // correct access order { public: // ... protected: // ... private: //
... };
```

(7) Within the public, protected and private sections of the class declaration the following order must be followed:

- a. Type declarations
- b. Data members
- c. Constructors
- d. Destructors
- e. Mutators
- f. Accessors

(8) The name of the class definition header file must match the name of the class implementation file.

2.5.3.6.2.2  
(07-10-2020)

**C++ Constructors and Destructors**

- (1) Constructors are used to allocate memory (if needed) and has special class functions that performs initialization of every object. The Compiler will call the Constructor whenever an object is created. The Destructor is a special member function of a class that is executed whenever an object of its class delete expression is applied, i.e., used to clean-up when a class object is destroyed.
- (2) All classes must declare a copy constructor if dynamic memory allocation is involved.
- (3) Classes that are meant to be instantiated only by their subclasses must have their constructor(s) and destructor declared protected.
- (4) Constructors other than the copy constructor that have only one parameter must be declared explicitly.
- (5) Classes with virtual functions and Classes with children must define a virtual destructor.

2.5.3.6.2.3  
(07-10-2020)

**C++ Class Data Initialization**

- (1) Data members must be initialized in the order in which they are declared.
- (2) Never declare non-const public data variables. Classes must declare member data private or protected. Always access non-const data through public access methods.

2.5.3.6.2.4  
(07-10-2020)  
**C++ Class Execution**

- (1) A member function must never return a reference or a pointer to a local variable. This will result in a pointer referencing a variable that has gone out of scope when the function returns.
- (2) A member function that does not modify member data must be declared const.
- (3) A public member function must never return a non-const reference or pointer to member data.
- (4) A public member function must never return a non-const reference or pointer to data outside an object, unless the object shares the data with other objects.
- (5) No member functions must be implemented within the class header declaration. The only exceptions are accessors and mutators (gets and sets).
- (6) Data must not be modified by const member functions if the behavior of an object is dependent on data outside the object.
- (7) Avoid overloading on a pointer type.
- (8) An assignment operator must return a reference to the assigning object (as the assignment operator returns the left hand side (LHS)). Non-const reference is consistent with the behavior of built-in types.
- (9) Assignment to self must be checked by adding the check at the beginning of the assignment method when overloading the assignment operator.

2.5.3.6.2.5  
(07-10-2020)  
**C++ Inheritance**

- (1) Inheritance must not be used for parts-of relations. Use template classes for parts-of relations.
- (2) Use inheritance to implement a generalization to specialization (kind of) relationships.
- (3) Inherited non-virtual member functions must not be overridden.

2.5.3.6.2.6  
(07-10-2020)  
**C++ Initialization**

- (1) Uninitialized data present in variables, objects, etc., have led to many problems. Often the problems manifest themselves as the very difficult “unable to reproduce” kind of problem at test time since the behavior of programs is dependent on the content of uninitialized variables, which may or may not change with every execution.

2.5.3.6.2.6.1  
(07-10-2020)  
**C++ Initialization of Variables**

- (1) Initialize all static variables explicitly.
- (2) Always initialize object instance variables of all types.
- (3) Every variable that is declared must be initialized before it is used.

2.5.3.6.2.6.2  
(07-10-2020)  
**C++ Initialization of Classes**

- (1) Use constructor initializer lists to ensure that every instance variable in a class has a defined value.
- (2) When allocating objects, ensure that all pointers to the objects are initialized via an initializer list or within the body of the constructor.
- (3) Classes that allocate dynamic storage must define a destructor that releases that storage.

2.5.3.6.3  
(04-28-2006)  
**Variables Scope**

- (4) After deleting the object in a destructor, set the pointer to NULL.
- (1) Header files must use forward declarations instead of including header file(s).
- (2) Global variables including using the C style extern declaration must not be used unless the need is clearly documented and described for maintainers. Use namespaces to declare broad scope variables.
- (3) File scope and static variables must be defined using the unnamed namespace.
- (4) Names occurring in close proximity to the current scope, including an enclosed scope, must not be redefined.
- (5) Do not write code that is dependent on the lifetime of a temporary variable.

2.5.3.6.4  
(04-28-2006)  
**Data Types**

- (1) A const must never be converted to a non-const, except when handling an exception.
- (2) Use struct when a user-defined type contains only data. Use classes when a user-defined type contains data and code that processes the data.
- (3) Predefined types must not be redefined.
- (4) Do not define boolean types. The predefined boolean values "true" and "false" must be used.
- (5) Do not declare the structure type and its associated variable in the same statement. Instead, declare the structure type first, and then declare a variable of that type.
- (6) Use the const type qualifier to identify data that should not change during execution. The compiler will flag attempts to change const variables, which aids debugging.
- (7) The size of an array must be declared with an enum constant or const. This makes modifying the size of an array easier, and is more meaningful to the reader.
- (8) Do not use the volatile qualifier. The purpose of volatile is to suppress optimization that can "optimize out" low level access to hardware.
- (9) The use of numeric values (magic numbers) in code must be avoided. Use descriptive constants instead, with the exception that the numbers 0 and 1 are permitted if their use is self-explanatory.

2.5.3.6.5  
(04-28-2006)  
**Conditional Constructs**

- (1) Logical expressions of the type if(test) or if(!test) must not be used when test is a pointer, or the test is the return of a function which returns a non-boolean value.
- (2) The "?:" operators must not be used.
- (3) A break statement must terminate the code following each case label.
- (4) The flow control primitives if, else, while, for and do must be followed by a block, even if it is an empty block.



- (5) The switch statement must always utilize the default statement.
- (6) For boolean expressions ('if', 'for', 'while', and 'do') involving non-boolean values, always use an explicit test of equality or non-equality.
- (7) Avoid conditional expressions that always have the same result.

#### 2.5.3.6.6 (07-10-2020) File Prologs

- (1) A file prolog must be used at the beginning of all source code files (header, implementation and main program files), appearing as the first item of the file. The following section provide the required prologs for C++ header and implementation files (including main programs). The following file naming convention must be used:

##### *C++ Example - File Prolog*

File Type	File Name/Suffix
C++ Header file	.h
C++ Implementation file (non-main)	.cpp
C++ Implementation file (main)	main.cpp

#### 2.5.3.6.6.1 (04-28-2006) File Size and Structure

- (1) This subsection cites standards for File Size and Structure.

##### 2.5.3.6.6.1.1 (04-28-2006) File Size

- (1) Unless precluded by the system or project requirements, C++ classes must define class member functions for a single class in a single file (i.e., one .cpp file per class, defining all methods for that class).

##### 2.5.3.6.6.1.2 (04-28-2006) File Structure

- (1) Developers must declare classes, data structures, layouts, functions, variables, and constants in a header file named classname.h.
- (2) The implementation of class methods, functions, and definition of non-static storage variables for each header file must be located in a corresponding classname.cpp file.

#### 2.5.3.6.6.2 (07-10-2020) C++ Name Conventions

- (1) While such names may be permitted by C++ standard and supported by compilers, names that vary solely by case tend to create readability problems for the maintenance programmer.

##### 2.5.3.6.6.2.1 (07-10-2020) C++ General Naming Conventions

- (1) Do not use names that vary solely by case.
- (2) Do not use typedef to emphasize distinct numeric categories. Use a class definition instead.
- (3) C++ names for all constructs such as classes, functions, variables, etc., must use the intercap convention. Intercap runs all words of a descriptive name together without underscores. An exception is made for names for constants
- (4) The following must use the Capital form:

- a. Classes and Structures (nominal Classes)
- b. Enum tagnames

2.5.3.6.6.2.1.1  
(07-10-2020)

**C++ Identifiers**

- (1) Identifiers must adhere to the conventions for ANSI standard C++.
- (2) Variable and function identifiers must use the normal form of the intercap notation. That is, the names of variables and functions must begin with a lowercase letter.
- (3) Do not use single variable names unless it is an index in a loop.

2.5.3.6.6.2.1.2  
(07-10-2020)

**C++ Functions and Parameter**

- (1) The names of formal arguments to functions must be specified and must be the same both in the function declaration and in the function definition.

2.5.3.6.6.2.1.3  
(07-10-2020)

**C++ Constants**

- (1) Define constants using fully capitalized words separated by underscore.
- (2) Define constants using const or enum. Do not use #define.

2.5.3.6.7  
(07-10-2020)

**C++ Formatting**

- (1) The IRS C++ standard indentation level is defined as 4 spaces; the programmer must follow proper indentations to ensure readability and consistency of the C++ source code.
- (2) Block statements must be indented one indentation level to show scope and structural coherence.

2.5.3.6.7.1  
(07-10-2020)

**C++ Indentation**

- (1) Indent the continued part of a statement at least one indentation level beyond the start of the statement.
- (2) Indentation must be performed using spaces instead of tab characters.
- (3) Indentation must be consistent throughout all source files.

2.5.3.6.7.2  
(07-10-2020)

**C++ Spacing**

- (1) The '\*' and '&' must be directly connected with the type names in declarations and definitions. I.e., no space must be placed between the '\*' or '&' and the type names in declarations and definitions.
- (2) There must be no spaces between a function name and the left parentheses.

2.5.3.6.7.3  
(07-10-2020)

**Grouping**

- (1) Begin each statement on a separate line.
- (2) Limit the length of source statements so that complete source statements are visible on the screen without having to scroll right and left repeatedly.
- (3) Braces ("{}") that enclose a block must be placed in the same column, on separate lines directly before and after the block.

2.5.3.6.7.4  
(07-10-2020)

**Includes**

- (1) Each header file must have "#include "any other headers on which it directly depends.
- (2) The "#includes" must precede any declarations or definitions in a file.

- (3) The “#include” must not be used to insert executable code into a file or to insert part of a class definition from an external file unless it is required by the use of software developed externally to the project, such as commercial off-the-shelf (COTS) products.
- (4) Place“ #include” statements in the following order:
  - a. (If a .cpp file) the corresponding header file
  - b. System header files
  - c. COTS header files
  - d. Project specific header files

#### 2.5.3.6.8 (07-10-2020) **Functions**

- (1) This subsection cites standards for Functions and Declarations.

#### 2.5.3.6.8.1 (07-10-2020) **Declarations**

- (1) Function prototypes and external variables must be declared in a header file (to avoid multiple declarations), not in implementation files.

#### 2.5.3.6.8.2 (07-10-2020) **Function Parameters**

- (1) The return type of a function must always be stated explicitly. Functions that do not return a value must be declared as void.
- (2) Pass parameters in input, modify, and output order.
- (3) Variable argument lists must not be used (ellipses notation).
- (4) Where the return value from functions are stored for future use, the function invocation must not be embedded in conditional or other function invocations. Functions may be embedded if return value is only used within that source.
- (5) Check the fault codes that may be received from library functions even if these functions seem fool proof.

#### 2.5.3.6.8.3 (07-10-2020) **Function Invocation, Execution, and Return**

- (1) Do not depend on the order of evaluation of arguments to a function.
- (2) Always return a value from main().
- (3) For functions with non-void return type, all paths must have a return statement that contains an expression of the return type.

#### 2.5.3.6.9 (04-28-2006) **Error Handling**

- (1) Error notification, and memory cleanup is very important in error-handling situations. The programs must have a consistent way of handling memory deallocations.

#### 2.5.3.6.9.1 (07-10-2020) **General Error Handling**

- (1) Calling functions must check for errors reported from functions.

#### 2.5.3.6.9.2 (04-28-2006) **Throwing Exceptions**

- (1) Do not throw exceptions from within destructors.
- (2) Throw only objects of class type.

2.5.3.6.9.3 (04-28-2006) <b>Handling Exceptions</b>	(1) Avoid memory leaks in exception handling code by using references in parameter lists for exception handlers.
2.5.3.6.10 (04-28-2006) <b>Expressions</b>	(1) Avoid using shift operations instead of arithmetic operations and validate arguments to be used in shift operators.
2.5.3.6.10.1 (07-10-2020) <b>Expression Arithmetic</b>	(1) Pointer arithmetic must not be used. (2) Do not depend on the behavior of underflow or overflow. (3) Apply unary minus to operands of signed type only.
2.5.3.6.10.2 (07-10-2020) <b>Type Conversions</b>	(1) Code must not depend on implicit type conversions. (2) The following assignments must not be used: <ol style="list-style-type: none"> <li>Assigning a pointer value to a non-pointer object</li> <li>Assigning a non-pointer value to a pointer object</li> <li>Assigning a function-pointer value to a data-pointer object</li> <li>Assigning a data-pointer value to a function-pointer object</li> </ol> (3) Do not convert floating values to integral types except through use of standard library routines.
2.5.3.6.10.3 (07-10-2020) <b>Pointers in Expressions</b>	(1) A pointer must not be compared to NULL or assigned NULL without casting the NULL to the proper type.
2.5.3.6.11 (07-10-2020) <b>Comments</b>	(1) Comments in computer programming is a programmer's readable annotation in the source code. The purpose is to allow a better understanding of the program's intent for other programmers accessing the code. Create the comments as the following; <ol style="list-style-type: none"> <li>Use // for in-line comments</li> <li>Use /* and */ for block comments</li> </ol>
2.5.3.6.12 (07-10-2020) <b>Memory Management</b>	(1) If a function is returning an allocated object that the caller must free, then the memory for the allocated object must be documented in the filename.h file as well as in any function comment header block that is used. (2) When memory allocated to a pointer has been deleted, a new value must be assigned to the pointer, or the pointer must be deleted.
2.5.3.6.12.1 (04-28-2006) <b>Heap and Stack Memories</b>	(1) Do not allocate dynamic objects contained within other objects as instance data. (2) Class authors must provide default values for dynamic objects within other objects and/or constructors that can be used on initializer lists.

2.5.3.6.12.2  
(07-10-2020)  
**Memory Leaks**

- (1) Save the address of allocated memory to prevent orphaned allocation.
- (2) Explicitly deallocate a dynamically allocated segment prior to assigning a new value.
- (3) Avoid memory leaks in exception handling code.
- (4) Do not use code that can throw an exception in a destructor.
- (5) Avoid memory leaks involving external components.
- (6) Avoid memory leaks involving reuse of “dead” objects.
- (7) Avoid memory leaks by explicitly clearing memory after use.
- (8) Avoid memory leaks by unfreezing frozen stream objects.
- (9) Avoid memory leaks by ensuring that you write the “Operator Delete” whenever the “Operator New” has been written for a class.

2.5.3.6.12.3  
(07-10-2020)  
**Buffers Overflows**

- (1) The size of an array must be declared with an enum constant or const. This makes modifying the size of an array easier, and is more meaningful to the reader.
- (2) When reading array contents, check to ensure that the index does not exceed the array size.
- (3) When adding to an array, check to ensure that the size of the array can hold the existing content in addition to the new contents to be added.
- (4) Always test a pointer’s value for NULL before using it.

2.5.3.7  
(07-10-2020)  
**Assembler Language Code (ALC) Programming**

- (1) This section of the IRM established controls to ensure coding of Assembler Language Code (ALC) are reliable, maintainable, and portable whether developed by IRS or outside vendors.
- (2) The controls established are applicable to all ALC programs whether they are developed by IRS government or contract employees. These guidelines apply to all assembler programming within the IRS. System specific commands, or technical information concerning addressing, registers, instructions and control language with Assembler should be referenced using IRS, IBM or UNISYS language references or standards.

2.5.3.7.1  
(07-10-2020)  
**Assembler Language Code (ALC) Overview**

- (1) Assembly languages have the same structure and set of commands as machine languages, but enable a programmer to use names instead of numbers. Since computers only understands (reads) bits which are comprised of machine language code of (1 and 0),. programmers write their instructions in symbolic language (Assembly). These symbolic source statements (source code) must be translated into machine language object statements (object code) before the computer can execute these instructions. An operating system program is used for this process. This translation process is known as assembling (assembly) the program, and serves to translate the source module (symbolic code) into a machine language object module.
- (2) ALC programming used within the IRS is developed on two mainframe systems :

- a. IBM operating system - IBM (JCL) Job Control Language
- b. UNISYS - (ECL) Executive Control Language

- (3) Since the processes for each system are unique with different functions, programmers should consult the specific industry standard language and programming references for specific instructions, commands, and directives.

#### 2.5.3.7.2 (07-10-2020)

##### **Assembler Language Code (ALC) Basic Principles**

- (1) Assembler language is a symbolic language use to code instructions instead of coding in machine language.
- (2) Source statements are interpreted by the Assembler and output as a machine language version of the program along with messages and listings.
- (3) The Binder outputs the Assembler program into an executable module.
- (4) IRS programmers use two mainframe systems and compilers to develop applications using ALC; IBM High Level Assembler (HLASM) and UNISYS Meta-Assembler (MASM).
- (5) Both systems have individual operating systems and control languages: IBM uses JCL (Job Control Language) and UNISYS uses ECL (Executive Control Language).
- (6) Each system has unique commands, instructions, directives, functions, and macros which are included in ALC programs.
- (7) Programmers write their instructions in symbolic language (Assembly) or high-level languages such as; (COBOL, C, JAVA). These symbolic source statements (source code) must be translated into machine language object statements (object code) before the computer can execute these instructions.

#### 2.5.3.7.3 (07-10-2020)

##### **Assembler Language Code (ALC) Program Comments and Documentation**

- (1) All programs must be commented for quick reference by other programmers. Beginning comments must include:
  - a. Project and Run numbers
  - b. Brief description of the run
  - c. Programmer's name
  - d. Current production assembly numbers and dates
- (2) Use Title statements to print the run number on each page.
- (3) Use Page ejects and spacing to separate routines.
- (4) As programs are updated or changed comments should be updated.
- (5) Paragraph type comments should not be on instruction statements since instructions could be pulled and comments left behind would be meaningless. Place comments at the beginning of routines they describe.
- (6) Constants that require periodic changes must be avoided. However, if changes are authorized because of management approval, comments must describe why and how they are changed.

- (7) Documentation for ALC are Computer Program Books (CPB's) for each program which contain file specifications, run descriptions, and a Computer Operator Handbook (COH) with instructions on all possible messages generated during operation.
- (8) Flowcharts and Pseudocode must be used to map the logical control of sequences and steps using a "Top Down" approach to make the programs main logic path recognizable.
- (9) Relative addressing is a great way to fix programs that are running out of base registers.
- (10) Do not code RECFM, BLKSIZE, LRECL, TRTCH or DENSITY into program DCB include these parameters in JCL to allow for flexibility. Exceptions to this are SYSOUT.
- (11) To conserve disk space use "GETMAIN" instead of "DS", "DC", this conserves space by covering fewer base registers.
- (12) When writing low volume local (MCC) print data sets such as program controls use SYSOUT. However; when using SYSOUT datasets for any purpose steps must be taken to prevent production problems at "checkpoint/restart" time. All SYSOUT datasets must be CLOSED or not yet opened when checkpoints are taken.
- (13) Using Abend Codes – in programs where halt issues conditions exist there should be a corresponding user abend condition.

2.5.3.7.4  
(07-10-2020)  
**Assembler Language  
Coding Conventions  
(ALC)**

- (1) Programs should be setup so that the "Patch" routine is executed first.
- (2) Write to Output (WTO's) are to be used minimally and primarily for required dynamic control of the executing program by operators.
- (3) Unless important traffic should be routed away from the console by issuing WTO's with a ROUTCD=13. Providing post-execution information for review is recommended using SYSOUT datasets instead.
- (4) Hold "GETS" and "PUTS to a minimum and set them up in closed subroutines.
- (5) Use closed subroutines and program modules as much as possible to enforce straightforward logic.
- (6) Use IRS Standard macros and load modules to take advantage of debugging routines.
- (7) Keep related routines together.
- (8) Code programs so that the flow of execution avoids crossing an MVS page boundary (4k of storage, covered by one base register) if your reading the entire (IMF/BMF).
- (9) Relative addressing is a great way to fix programs that are running out of base registers.
- (10) Do not code: RECFM, BLKSIZE, LRECL, TRTCH or DENSITY into program DCB, include these parameters in JCL to allow for flexibility. Exceptions to this are SYSOUT.



- (11) To conserve disk space use “GETMAIN” instead of “DS”, “DC”, this conserves space by covering fewer base registers.
- (12) When writing low volume local (MCC) print data sets like program controls use SYSOUT. However; when using SYSOUT datasets for any purpose, steps must be taken to prevent production problems at “checkpoint/restart” time. All SYSOUT datasets must be CLOSED, or not opened when checkpoints are taken.
- (13) **Using Abend Codes** – in programs where halt issues conditions exist there must be a corresponding user abend condition:
  - a. **Input File (SYSIN) Exceptions:**
    - 1) 20 - Missing Statement/Record
    - 2) 21 - Bad Date Statement/Record
    - 3) 22 - Bad Segment Statement/Record
    - 4) 23 - Bad CP23 Statement/Record
    - 5) 24 - 29 (Available)
  - b. **Input File Exceptions:**
    - 1) 30 - Wrong Input File
    - 2) 21 - Bad Date Statement/Record
    - 3) 22 - Bad Segment Statement/Record
    - 4) 23 - Bad CP23 Statement/Record
    - 5) 24- 29 (Available)
  - c. **Data Exception:**
    - 1) 40 - Out of Sequence Record
    - 2) 41 - Invalid S.C./D.O./Region Code
    - 3) 42 - Duplicate Record
    - 4) 43 - Tax Module with no Entity
    - 5) 44 - Invalid TIN
    - 6) 45 - Bad Byte Count
    - 7) 46 Bad Year Digits
    - 8) 47 - Bad Name Control
    - 9) 48 - TIN out of Segment Range
    - 10) 49 - 59 (Available)
  - d. **End Of File/End Of Job Exceptions:**
    - 1) 60 - Controls out of Balance
    - 2) 61 - 69 (Available)
- (14) At a good EOJ (End of Job) set return code to 0 (Load register 15 with zeros).
- (15) Do not use the last two bytes of the Record Descriptor Word (RDW) as a user data field. The RDW is referred to as “Byte Count” in IRS documentation, the first two bytes hold the record length the last two are reserved by IBM.
- (16) Any reserved bytes used by IBM or Unisys for their operating systems should not be used by applications.

#### 2.5.3.7.4.1 (07-10-2020)

#### **Assembler Language Code (ALC) Defining Constants and Storage**

- (1) Fields are defined in Assembler using Define Constants (DC) and Define Storage (DS) statements. These statements provide the field's address and length. The DC also provides an initial value for the field.
- (2) Data definition statements may be used for:
  - a. Define a constant value used in a program.
  - b. Define and describe storage area receiving the input record read.



- c. Define and describe storage area to build or modify output data.
  - d. Set aside and label work areas to store data for later use.
- (3) Constants used are the following:
- a. As Counters to count the number of records read, written or containing errors.
  - b. Accumulators to track the amount of payments made to an account, or track the total balance for a tax record.
  - c. Headings for printed reports.
  - d. Predetermined messages printed as result of an operation.
- (4) The Assembler generates the formatted data in its assigned address when it processes a DC statement. The assembler is instructed to reserve an area in storage that:
- a. Has a specified length (either explicitly stated or implied).
  - b. Contains data in a specified format and contains an initial value.

\*Example Statement

**Assembler Language Code (ALC) Example - Using DC Statement**

Assembler Language Code Example			
OBJECT CODE	STATEMENT		
C1C2C3	ALPHAS	DC C'ABC'	Implied Length of 3 bytes
1C	ONE	DC P'1	Implied Length of 1 byte
00000C	COUNTER	DC PL3'0'	Explicit Length of 3 bytes

**Defined Example Format**

ADDONE DC 4 PL 5' 1' Name/Label Name OPCODE Duplication  
Type Length Constant Factor (Operand Subfield) Type and Constant  
(required)

- (5) The Name field gives a symbolic (mnemonic) addressability to the memory location created for the defined constant. They should be helpful, descriptive, and unique. They are referred to as "Symbols", or "Labels". The following are rules for name fields:
- a. Use on instructions as well as data fields
  - b. It must begin in column 1
  - c. It can be from 1-63 characters long but is usually 8 or fewer
  - d. The first character must be an alphabetic or national symbol. The first character must be (A-Z), (ALC is not case sensitive) or ( \$ # @ ). Generally, nationals are avoided as first characters because they frequently imply special meanings.
  - e. The remaining characters can be alphabetic, numeric, national or under-score symbol. This includes (A-Z), (0-9), and ( \$ # @ )

- f. No blanks within the name. A blank in the first column indicates no name is present. Underscores are commonly used in place of blanks.
  - g. A unique character combination can only be defined once. It can be used ("referenced") any number of times as an operand.
  - h. External labels, (such as the label on the SLINK macro and CSECT statements), may not use the underscore and may not exceed eight characters.
- (6) Address constants (ADCONS) used in DC statements are used to place the address of one storage area into the storage location of another. Refer to the specific language reference for your system for examples.
  - (7) The constant or literal is the initial value of the data field. Constants can be changed in ALC.
  - (8) Since Y2K, the standard is to use 4-digit years in all date fields.
  - (9) Zeros – indicate the absence of significant digits; Blanks are not Noting – they indicate the absence of significant data.
  - (1) The reference that covers IRS macros used in Assembler is Chapter 8 of the IBM Systems Standards Manual. The subject areas covered, and brief descriptions are the following:

- a. **Housekeeping** - see table below:

***ALC Example - Housekeeping Macros and Description***

Macros	Description
<b>CNVDATE</b>	<b>Performs various date conversions</b>
DATE	Provides current date in different formats
EOVCKPT	Takes checkpoints on file SYSCKEOV
EQREG	Equates registers to symbolic names
IRCKPT	Takes checkpoints on file SYSCKPT
SLINK	Provides standard linkage and save area
STATUS90	Provides status history within a tax module

- b. **Data Management** - see table below:

***ALC Example - Data Management Macros and Description***

Macros	Description
BLKPT	Converts SSN/EIN to IDRS block pointer
Data Compression Macros	SHRINK and EXPAND
Data Definition Macros	IETyy, ITXyy, ENTyy, TAXyy
ETRANS98	Entity transaction search
RGTAB	Translates district office

#### 2.5.3.7.5

(07-10-2020)

#### **Assembler Language Code (ALC) Standard Macros**

Macros	Description
RGTABSM	Translates district office (sort/merge modules)
TRANSyy	Tax module transaction search

c. **Input/Output (I/O)** - see table below:

***ALC Example - Input/Output (I/O) Macros and Description***

Macros	Description
<b>FEOV</b>	<b>Forces End of Volume</b>
PUT	IBM macro to write records
TRUNC	truncate block before FEOV

d. **LARS** - see table below:

***ALC Example - LARS Macros and Description***

Macros	Description
CTL00	Issues balancing instruction message
CTL01	Issues CNTRL001 message-single value
CTL1A	Issues CNTRL001 message-array of values
CNVDATE	Performs various date conversions (Julian and calendar date formatting)

- e. **CNVDATE:** This macro performs various date conversions (Julian and calendar date formatting)
- f. **REGISTER Usage:** Registers 0, 1 and 15 are used during the execution of this macro and must be saved by the user. Registers 2 - 14 may be used in the operands, enclosed in parentheses, to point to a date field.
- g. **ABEND Messages:** The SVC called by this macro checks for invalid data if found the job abends with a S0C4 (register 10 should point to the invalid data) after issuing one of following messages:

***ALC Example - IBM Abend Message Descriptions***

IBM Abend	Message Descriptions
IGC0024A	DATE PASSED IS NOT UNSIGNED CHARACTER (REG 10). YYDDD or MMDDYY data is not numeric characters.
IGC0024A	DDD FIELD IS NOT IN 001-366 RANGE (REG 10). The DDD portion of the Julian date you passed was not valid.
IGC0024A	DD FIELD IS NOT IN 01-31 RANGE (REG 10). The DD portion of the Calendar date you passed was not valid.
IGC0024A	DD FIELD IS NOT IN 01-31 RANGE (REG 10). The DD portion of the Calendar date you passed was not valid.

- h. **DATE:** Creates the availability of the time, current date in Julian format, and standard MMDDYY format to the programmer.
- i. **EOVCKPT (Similar to IRCKPT):** Used to take checkpoints at logical points and saves main storage and job queue information related to the job step. When the checkpoint operation is complete, execution of the program continues. It also generates a DCB for the checkpoint dataset for the programmer. The DDNAME for the checkpoint file is SYSCKEOV.
- j. **EQREG:** This macro generates assembler instructions to equate the sixteen general registers to symbolic names. This allows programmers to use the symbolic name "R#" instead of the actual register number when coding. The symbolic names also appear in the cross-reference table of the assembly listing and can be easily referenced.
- k. **SLINK:** This macro provides the standard IBM linkage conventions needed in all stand-alone programs.
  - Saves all registers in the calling programs save area.
  - Provides an 18 word save area to be used by modules called by the problem program.
  - Register 13 points to the save area a required when calling another module.
  - Must be coded as the **FIRST** executable instruction in the module.
  - No **START** statement is necessary since **SLINK** generates its own **START** statement.
  - A label is required.
  - Operands are not in this macro.
  - Because this macro generates its own **START** statement any additional assembler statements must be coded after **SLINK** for example: CCW, CNOP, CSECT, CXD, DC, DROP, DS, END, EQU, LTORG, ORG, START and USING.
- l. **STATUS90(Status Search macro):** Locates and makes available to the programmer all status histories within a Tax module. This macro also locates the latest of current status for either IMF or BMF.
- m. **BLKPT:** This macro will convert a nine-byte zoned decimal SSN or EIN to a block pointer for IDRS. The block pointer will be returned to register 14 at the NSI below the macro.
- n. **DATA DEFINITION:** The macros in this category generate DSECT's which define the fixed portion of the IMF or BMF tax modules or entity modules. The labels generated within the DSECT can be referred in the program as stated below:
  - IMF Entity Module - IETyy
  - IMF Tax Module - ITXyy
  - BMF Entity Module - ENTyy
  - BMF Tax Module - TAXyy
- o. **ETRANS98:** This macro locates and makes available to the programmer all transaction within an entity module either IMF or BMF
- p. **RGTAB and RGTABSM:** These macros translate any valid District Office or Service Center code into the appropriate Region and Service Center codes. Additionally, the RGTAB generates a 256-byte translate table which will translate a character as posted to the IMF to an IBM EBCDIC character. Alpha and numeric remain the same. Special characters are translated. Refer to exhibit 1 for translations.
- q. **TRANSyy – (Tax Module Transaction Search):** This macro locates and makes available to the programmer all transactions within a tax module, either IMF or BMF.

- r. **BODTAB and BODTABSM:** Translate and validate the following:
  - Universal Location Codes (ULC)
  - Business Operating Divisions (BOD)
  - Expanded Area Offices (MF/TIF AO), or File Location Codes (FLC) into the appropriate BOD Area Offices (BODAO), MF/TIF AO, Service Center codes, and the substitute (dump) FLC (when applicable).
- s. IMF no longer uses 7074 – **Funny Pack**

2.5.3.8  
(07-10-2020)  
**Java Programming  
Language**

- (1) This section of the IRM provides controls to ensure Java programs are reliable, maintainable, and portable. The controls established are applicable to all Java programming projects whether they are developed by IRS government or contract employees.

2.5.3.8.1  
(07-10-2020)  
**Java Programming  
Overview**

- (1) Java programming is a robust general-purpose computer programming language that has the ability of running several programs, or parts of a program in parallel. This allows a program to achieve high performance, and throughput.
- (2) Java has its own structure, syntax rules, and programming framework which is based on the concept of object-oriented programming, and is designed to have as few implementation dependencies as possible This allows application developers to “write once, run anywhere” (WORA) which means compiled Java code can run on all platforms that support Java without requiring recompiling.
- (3) Structurally, Java is comprised of the following:
  - a. **Package:** This is a namespace mechanism consisting of classes.
  - b. **Classes:** An user defined template from which objects are created consisting of methods, variables, constants, and constructors.
  - c. **Object:** Consist of the State (attributes), this behavior is (represented by method as an object) pertaining to the following:
    - An object and response of an object with other objects.
    - Identity (unique name to an object), and enables one object to interact with other objects.
  - d. **Java compiler:** Java platform source code is written to .java files for compiling, the compiler checks the developers code against the language’s syntax rules them writes out the bytecode in .class files.

2.5.3.8.2  
(07-10-2020)  
**Program Objectives**

- (1) Java applications are composed of one or more source files. Each source file must be assigned to a package.
- (2) A Java assembly contains one or more related packages.
- (3) Java source files must follow the following structure and naming conventions as listed below.

2.5.3.8.2.1  
(07-10-2020)  
**Source File Structure**

- (1) A source file is composed of the following sections. Each section should be separated by a blank lines and optional comment identifying each section.
  - a. Beginning comments
  - b. Package and import statements
  - c. Main public class declaration
  - d. Private class and interfaces associated to the main class

- (2) A source file must contain only one public class, and the file must be named the same as the public class name.
- (3) Avoid source files exceeding 2000 lines of code.

2.5.3.8.2.1.1  
(07-10-2020)

#### Beginning Comments

- (1) Begin all source files with a C-style comment that lists the class name, version information and date.
- (2) The comment should also include a revision history when the file is modified with a brief description of the changes made. Each revision should contain the date saved, first initial and last name of programmer with SEID, and the change information.

#### Java Programming Example - Comments

Java Programming Comment Example
<pre> /*  * ClassName  *  * Revision History  * 08/24/2013  A. Programmer (SEID1)  *           Initial Release  */ </pre>

2.5.3.8.2.1.2  
(07-10-2020)

#### Package and Import Statements

- (1) The main package and all sub-packages should be in a single Java Archive for distribution.
- (2) When building web applications, the compiled source code will be bundled with web resources and other necessary JAR files into either a Web Archive (WAR) or Enterprise Archive (EAR).
- (3) Assemblies should only contain compiled Java source (.class file) that will be used by calling applications. They should not include source (.java) or compiled unit test files.
- (4) Assemblies must have a manifest file (META-INF/MAINIFEST.MF) present, for more information on Java Manifest files see <https://docs.oracle.com/javase/tutorial/deployment/jar/manifestindex.html>.
- (5) The first non-comment line must be the package statement.
- (6) The package name is always in lowercase ASCII letters and must start with "gov.irs.project", or "gov.irs.program.project ."
- (7) List import statements in alphabetic order.
- (8) Import statements follow the package statement with a blank line.
- (9) Import statements must fully qualify the class name imported. Do not use wildcards for importing an entire package.
- (10) To import static members of a class, use the static import statement.

#### Java Programming Example - Import Statements

##### Example of Import Statement

```
import java.awt.Canvas;
import static java.lang.Math.PI;

// NOT THIS
import java.awt.*;
```

- (11) More information: <https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html>.

2.5.3.8.2.2  
(07-10-2020)

#### Naming Conventions

- (1) The following conventions should be used when determining names.

2.5.3.8.2.2.1  
(07-10-2020)

#### Capitalization Conventions

- (1) PascalCasing capitalizes the first character of each word. Acronyms are always in uppercase. For example, "MyClass" and "URLEncoder".
- (2) CamelCasing capitalizes the first character of each word except the first. Acronyms are always in lowercase. For example, "propertyName" or "ioStream".
- (3) Below is a table of commonly used program identifiers and the appropriate capitalization scheme.

#### Java Programming Examples - Capitalization

Identifier	Casing	Example
Package	Lower	package gov.irs.myproject.core;
Type	Pascal	public class StreamReader { ... }
Interface	Pascal	public interface Enumerable { ... }
Method	Camel	public String toString() { ... }
Field	Camel	private long timeElapsed = 0L;
Constant	Upper	public static final int MAXIMUM_DELAY = 1000;
Enum Value	Upper	public enum Color { RED, WHITE, ROYAL_BLUE }

2.5.3.8.2.2.2  
(07-10-2020)

#### Type Member Names

- (1) Give methods a name that are verb or verb phrases.

**Java Programming Example - Method Naming Convention**

Method Naming Convention Example
<pre>public class String {     public int compareTo() { }     public String[] split() { }     public String trim() { } }</pre>

- (2) When accessing attributes of a class, use the get/set method pattern.

2.5.3.8.2.2.3  
(07-10-2020)

**General Names**

- (1) Compound terms are treated as single words for capitalization purposes, refer to Exhibit 2.5.3-18 for commonly used terms.
- (2) Choose readable identifier names. For example, “Horizontal-Alignment” is more readable than “Alignment-Horizontal”.

2.5.3.8.2.2.4  
(07-10-2020)

**Assembly Names**

- (1) The assembly, or Java archive, must be named with the top-level package name and version number. The initial prefix, “gov.irs”, must not be included in the assembly name, see Figure 2.5.3-23 .

**Java Programming Example - Java Archive Naming Convention**

Java Archive Naming	Convention Example
Top Package Name	Assembly (JAR file) Name
gov.irs.myprogram.myproject (v1.2)	myprogram-myproject-1.2- RELEASE.jar

**Figure 2.5.3-23**

2.5.3.8.2.2.5  
(07-10-2020)

**Package Names**

- (1) Package names must always be in lowercase.
- (2) Prefix all package names with “gov.irs.”
- (3) Follow the prefix with the program and project name.
- (4) Subsequent portions of the package name must be grouped by related technologies.
- (5) Use plural namespaces where appropriate, e.g., **package gov.irs.myproject-models**.

2.5.3.8.2.2.6  
(07-10-2020)

**Resource Names**

- (1) Resources should be obtained from one of the standard resource bundle classes, e.g., “java.util.PropertyResourceBundle”.
- (2) Bundle files should be named with a noun or noun phrase indicating the resource bundle content, along with a suffix to indicate the language. Optionally include a country code and platform if differentiation is required, e.g., a bundle that contains error messages in US English would be called “ErrorMessage\_en\_US.properties”.



- (3) Bundle keys should be noun or noun phrases in PascalCase, such as “OkKey” or “InvalidLoginError”.
- (4) For more information, see <https://docs.oracle.com/javase/tutorial/i18n/resbundle/concept.html>.

## 2.5.3.8.3 (07-10-2020) Layout Conventions

- (1) Use the Eclipse IDE defaults when possible.
  - a. Use a tab size of four (4) characters. Insert spaces when using the tab key.
  - b. Use an indent spacing of four (4) characters.
  - c. Do not qualify member access with “this” keyword unless required.

### *Java Programming Example - Syntax Format*

Syntax Format
<pre>private int mode = 1; public int getMode() {     return mode; }  public void setMode(int mode) {     this.mode = mode; }</pre>

- d. Prefer the primitive type (int, boolean) over the object type (Integer, Boolean) when declaring locals, and parameters. Use the object type for member access expressions.

### *Java Programming Example - Primitive Object Usage*

Primitive Object Usage
<pre>public void resetMode() {     mode = Integer.MAX_VALUE; }</pre>

- e. Use one indent for block contents within a code block.
- (2) Write only one declaration per line.
- (3) Write only one statement per line.

### *Java Programming Example - Declaration and Statement Formatting*

Declaration and Statement Formatting
<pre>String name = "Billy"; int weight = 0; int height = 0; int bodyMass = weight / height;  // Do not do this. char c = 'c'; char d = c = 'x';</pre>

**IRS Standard Java Format Example**

Standard Java Format
<pre> { public class MyIntStack  private final LinkedList fStack;  public MyIntStack();  fStack = new LinkedList(); } public int pop() {      return ((Integer) fStack.removeFirst()).intValue(); }  public void push(int elem) { {    fStack.addFirst(new Integer(elem)); } public boolean isEmpty() { {    return fStack.isEmpty(); } } } </pre>

2.5.3.8.3.1  
(07-10-2020)

**Java Programming  
Example - Wrapping  
Lines**

- (1) When an expression will not fit on a single line, break it according to these general principles, see table below:
  - Break after a comma
  - Break before an operator
  - Prefer higher-level breaks to lower-level breaks
  - For arithmetic statements, try to keep expressions in parentheses together
  - Align the new line with the beginning of the expression at the same level on the previous line
  - Indent four spaces if these rules lead to confusing code or to code that is up against the right margin
- (2) Line wrapping of “if” statements should generally use one indent to make it easier to see the body of the statement.
- (3) Ternary expressions can be formatting in the following way, see table below:

**Java Programming Example - Ternary Expressions**

Example of Ternary Expressions
<pre> var alpha = (aLongBooleanExpression)? beta : gamma; var alpha = (aLongBooleanExpression) ? beta : gamma; </pre>

2.5.3.8.4  
(07-10-2020)  
**Java Programming  
Commenting  
Conventions**

- (1) Ensure comments provide a code overview and additional information that is not available in the code itself.
- (2) Ensure comments contain only information that is relevant to reading and understanding the program, e.g., do not include information about how the corresponding package is built or in what directory it resides as a comment.
- (3) Providing non-trivial or non-obvious design decisions in comments is appropriate, but avoid giving information that is clear from the code.
- (4) Avoid any comments that are likely to become dated as the code evolves.
- (5) Do not enclose comments in large boxes drawn with asterisks or other characters.
- (6) Never include special characters such as those for form-feed and backspace in comments.

2.5.3.8.4.1  
(07-10-2020)  
**Java Programming  
Single Line Comments**

- (1) Place comments on a separate line, not at the end of a line of code.
- (2) Precede a block comment with a blank line.
- (3) Begin comment text with an upper-case letter.
- (4) End comment text with a period.
- (5) Insert one space between the comment delimiter ("`/*`") and the comment text.
- (6) Do not create formatted blocks of asterisks around comments.
- (7) Align the comment to the current statement level.

2.5.3.8.4.2  
(07-10-2020)  
**Java Programming  
Block Comments**

- (1) Use block comments to provide descriptions of files, data structures, and algorithms.
- (2) Block comments inside a function or methods should be indented to the same level as the code they describe.
- (3) Precede a block comment with a blank line.
- (4) Comment line between the start and ending block comment tags must not have asterisks or some other identifying character. For an example, see table below:

**Java Programming Example - Block Comments**

Example - Block Comments
<pre> /* On Windows machines numbers are in little-endian format.  */ int winBytesToNumber(byte[] digits) {     /*         This must be done since Windows stores bytes with         the least significant digit first.     */     int[] bigEndianDigits = IntStream.rangeClosed(1, digits.length)         .map(i -&gt; digits[digits.length-i])         .toArray();     return bytesToNumber(bigEndianDigits); } </pre>

- (5) Place the documentation comment directly above the class or class member requiring documentation.
- (6) If information about a class, interface, variable, or method that is not appropriate for documentation is required, use a block comment immediately after the declaration. For example, place details about the implementation of a class in such an implementation block comment following the class statement, not in the class doc comment.
- (7) Examples for use in documentation should have no more than 70 characters.
- (8) Include any security-related information such as required permissions, security related exceptions, caller sensitivity, and any security related preconditions or postconditions.
- (9) For additional information about writing documentation comments, see <https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>.

#### 2.5.3.8.5 (07-10-2020) **Class Design**

- (1) A class is defined as a template/blueprint that describes the behavior/state of the software object. A software object's state is stored in fields and behavior is shown via methods.
- (2) An important class design goal is to have self-contained classes with functional code that is more maintainable, testable, and reusable.

#### 2.5.3.8.5.1 (07-10-2020) **Packages**

- (1) All classes must be assigned to a package.
- (2) A package helps organize classes into logical chunks of functionality.
- (3) Classes and other top-level objects that do not have an access modifier are only visible to any other object in that package.
- (4) Mark the package as sealed in the manifest when creating the JAR file for the package.
- (5) Set the security property "package.access" to prevent untrusted classes from other class loaders to use reflection, and access the package through private APIs.

## 2.5.3.8.5.2 (07-10-2020) Interfaces

- (1) An interface contains definitions for a group of related functionalities that a class can implement.
- (2) By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important because the language doesn't support multiple inheritance of classes, see table below.

### Java Interface Example

Example - Java Interface
<pre>interface Drivable {     public int MAXIMUM_SPEED = 200;     void toggleSignal(boolean state);     void setDirection(int degrees);     int getSpeed();     static String getDescription(int wheelCount) {     }     default int getRevolutionsPerMinute(int wheelSize, int speed) {     } }</pre>

- (3) Any class that implements the interface as shown above must implement the two methods with the same signature, like a contract. That class can also be safely recast as an instance of that interface. The implemented methods must be "public" in scope.
- (4) Interfaces may extend from one or more interfaces.
- (5) In addition to empty method signatures, interfaces may also include:
  - Constants – values that cannot change once defined in the interface
  - Default Methods – default implementations of methods that can be overridden by implementing classes
  - Static methods – public methods that implementing classes cannot override
- (6) Additional information on Class interfaces can be found at <https://docs.oracle.com/javase/tutorial/java/landl/createinterface.html>

## 2.5.3.8.5.3 (07-10-2020) Classes

- (1) A type that is defined as a class is a reference type.
- (2) Variables assigned to a reference type initially have no value, or "null". To assign a value to a reference type, either use the "new" operator, use a pointer to an existing reference, or use a method that creates new reference type instances.

### Java Programming - Reference Type

Example - Reference Type
<pre>// [access modifier] - [class] - [identifier] public class Customer {     // Fields, properties, methods and events go here }</pre>

- (3) Classes may only extend from one class, but may implement one or more interfaces. If a class does not explicitly extend from another class it extends from "java.lang.Object".
- (4) Provide the ability to create safe copies of the class. Do not implement the interface "java.lang.Cloneable".
- (5) Do not rely on the method "Object.equals" as the sole determination that two objects are equivalent.
- (6) Always override the methods "equals()" and "hashCode()" so that two instances of the same call are functionally equivalent.
- (7) Follow the general contract when overriding the method "compareTo()".
- (8) Compare class instances and not class names.
- (9) For more information see, <https://docs.oracle.com/javase/tutorial/java/javaOO/classes.html>.

2.5.3.8.5.3.1  
(07-10-2020)

#### Abstract Classes

- (1) The purpose of an abstract class is to provide a common function set that multiple derived classes can share.
- (2) Abstract classes may also define abstract methods. Abstract methods must be implemented in any class that extends from the abstract class.
- (3) Use an abstract class over an interface if:
  - Code must be shared over several closely related classes if unrelated classes will implement your interface.
  - Classes that extend the abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
  - Declare non-static or non-final fields, allowing a common set of methods that can access, and modify the state of the object to which they belong.
- (4) Use an interface over an abstract class if the interfaces "Comparable" and "Cloneable" are implemented by many unrelated classes.
  - Behavior of a data type must be specified, but not concerned about who implements its behavior
  - Use multiple type inheritance
- (5) For more information see, <https://docs.oracle.com/javase/tutorial/java/andl/abstract.html>.

2.5.3.8.5.3.2  
(07-10-2020)

#### Sealed Classes

- (1) Sealed, or "final" classes, cannot be extended like other classes.
- (2) Immutable classes, like "java.lang.String", must always be sealed.
- (3) While a static class may have non-static methods, only one instance of a static class exists. All references to a static class point to that one instance.
- (4) Unlike other inner classes, static classes do not have access to members of the enclosing class, see table below for example:

### Java Programming Example - Sealed Classes

Example of Sealed Classes
<pre> class OuterClass {     public OuterClass() {         StaticNestedClass.setProperty("color");     }     static class StaticNestedClass {         private String prop;         void setProperty(String prop) { this.prop = prop; }         void String getProperty() { return prop; }     } } public static void main(String[] args) {   OuterClass.StaticNestedClass.setProperty("color"); }</pre>

#### 2.5.3.8.5.3.3 (07-10-2020) Static Classes

- (1) In object-oriented programming a static class is any class variable that is declared with a static modifier where a single copy exists regardless of how many instances of the class exist. Static classes also have the following characteristics:
  - a. Static classes like sealed classes, cannot be extended.
  - b. A static class may only exist inside another class.
  - c. While a static class may have non static methods, only one instance of a static class can exist.
  - d. Unlike other inner classes, static classes do not have access to members of the enclosing class.

#### 2.5.3.8.5.3.4 (07-10-2020) Inner Classes

- (1) Use a non-static nested class (inner class) if you require access to an enclosing instance's non-public fields and methods. Use a static nested class if you don't require this access.
- (2) Inner classes, unlike static classes, have access to the private members of the enclosing class.
- (3) Use inner classes when:
  - It makes sense to logically groups classes that are only used in one place
  - It increases encapsulation
  - It can lead to more readable and maintainable code
- (4) While permissible, avoid shadowing member variables in inner classes, see table below:

**Java Programming Example - Inner Classes****Example of Inner Classes**

```

public class ShadowTest {
    public int x = 0;
    class FirstLevel {
        public int x = 1;
        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
        }
    }
    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
x = 23
this.x = 1
ShadowTest.this.x = 0

```

2.5.3.8.5.3.5  
(07-10-2020)

**Immutable Classes**

- (1) Consider making classes immutable to prevent member data from changing after creation.
- (2) Immutable classes should be sealed (marked "final").
- (3) Hide default constructors and provide only constructors that populate the class properties.
- (4) Mark property accessors as final.
- (5) If a class property is a mutable reference value, create copies of those values.

2.5.3.8.5.3.6  
(07-10-2020)

**Objects**

- (1) A class definition is like a blueprint that specifies what the type can do. An object is basically a block of memory that has been allocated and configured according to the blueprint.
- (2) Except for static classes, a program may create multiple instances of a class that exist independently of each other.
- (3) Since classes are reference types, assigning a variable to an existing reference type instance simply assigns a pointer to that class instance. Changes made through either reference affects both references., see Exhibit 2.5.3-2 Exhibit 2.5.3-3
- (4) Classes that inherit from abstract classes must implement any members marked as abstract. An abstract class that inherits from an abstract class does not have to implement those abstract members from the base class.



- (5) An object can be recast to any of the object's base classes. The recast object will no longer have access to the derived classes members. If recasting an object to a derived class, the class must be explicitly specified in the assignment. If the recast is invalid, a runtime exception of "java.lang.ClassCastException" is thrown.
  - a. Point pt = new Point
  - b. Object o = pt;
  - c. Point pt2 = (Point)o
- (6) If an object is recast to a base class, the actual class instance is used to determine which overridden method to use.
- (7) To ensure a cast is valid, wrap any explicit casts using the "instanceof" check.
- (8) For more information, see <https://docs.oracle.com/javase/tutorial/java/landl/subclasses.html>.

## 2.5.3.8.5.3.7 (07-10-2020)

### Class Access Modifiers

- (1) Top level classes may only be accessible by other classes within the same package (no modifier), or available outside of the package ("public" modifier).
- (2) Inner classes have the same set of permissions as other members of a class: "public", "protected", "private", or package-access permission.
- (3) For more information, see <https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>.

## 2.5.3.8.5.3.8 (07-10-2020)

### Fields

- (1) A field is a variable that is declared directly in a class. Use fields when data is shared between methods and must be available beyond the lifecycle of a single method.
- (2) A class can have fields that are unique for each instance of the class or shared between all instances ("static").
- (3) Unique fields should only have "private" or "protected" scope. To access or modify a field outside of the class hierarchy, use "get" and "set" methods.
- (4) While a field with "protected" scope is visible to all classes within the same package, do not modify that field directly except from derived classes. Create protected get/set methods so derived classes can modify the field if necessary.
- (5) To make a field read only, add the modifier "final" to the field. There is no restriction to the initial assignment: it may be a constant or use a "new" statement to create a new instance, see table below..

#### Java Programming Example - Creating a Read-Only Field

##### Java Programming - Example of Making a Field Read-Only

```
class Animal {
    private String name;
    protected int number;
    final Animal defaultAnimal = new Dog();
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

- (6) Prevent class initialization cycles through static field assignments. If a static field need to be assigned to a reference value, ensure the reference value doesn't have association with the class with the static value directly, nor indirectly.
- (7) When using collection maps, use only immutable types for the key parameter.

#### 2.5.3.8.5.3.9 (07-10-2020) Types

- (1) Java is a strongly-typed language. Every field, every expression that evaluates to a value, and every method has explicitly typed input parameters and return value.
- (2) To find detailed type information, use the method "getClass()" on instances or the "class" property on classes.

#### **Java Programming Example - getClass Method**

Java Programming example of getClass
<pre>class instanceClass = collie.getClass(); class dogClass = Dog.class;</pre>

- (3) Once a variable is assigned a type, the type cannot be changed except by creating a new variable.

#### **Java Programming Example - Type Assigned to Variable**

Java Programming example - Type
<pre>int a = 5; a = false; // compile-time error boolean flag = (a &gt; 10);</pre>

- (4) Prefer using the "int" data type over "short" and "byte" for arithmetic operations since the processor will automatically promote them to an "int" prior to performing the operation.
- (5) When manipulating characters (the "char" data type), use the "int" data type when manipulating bits of characters to avoid sign issues.
- (6) When assigning a large number to a variable, consider using underscores for readability.

#### **Java Programming Example - Use of Underscores**

Java Programming - Use of Underscores
<pre>int creditCardNumber = 5424_1234_5678_1234; int socialSecurityNumber = 999_99_9999; long hexNumber = 0xCAFE_BABE;</pre>

- (7) For more information on Java primitives *More information on Java primitives see, <https://cs.fit.edu/~ryan/java/language/java-data.html> and <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>.*

2.5.3.8.5.3.9.1  
(07-10-2020)  
**Autoboxing and  
Unboxing Types**

- (1) Autoboxing refers to the promotion of a primitive type to a reference type, for example from “int” to “Integer”. Unboxing refers to the opposite process.
- (2) Generic types require reference types as the generic type parameter. You can add the appropriate primitive values directly and the value is automatically promoted

**Java Programming Example - Using Autoboxing Type**

Autoboxing
<pre>List &lt;Integer&gt; intLiist = new ArrayList&lt;&gt;(); intList.add(42); int addListValues(List &lt;Integer&gt; intList){ int sum = 0;     for (Integer item : intList) { sum += item;     } return sum; } String s = “The sum is: “ + 42;</pre>

- (3) Unboxing can occur either through method invocation or assignment, see Figure 2.5.3-24

**Java Programming Example - Unboxing Type**

Java Programming Unboxing Type
<pre>int calculateResult(int value) { }  Integer intObject = 42;  int result = calculateResult(intObject); List&lt;Integer&gt;: intList) = new ArrayList&lt;&gt;( ) for (int i : intList) { }</pre>

**Figure 2.5.3-24**

- (4) In general, use the primitive data type unless the number needs to be converted to another number type such as “byteValue()” or working with generic classes.
- (5) Do not use equality operators (“==” and “!=”) to compare object data type values.
- (6) For more information: <https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html>.

2.5.3.8.5.3.9.2  
(07-10-2020)  
**Enumeration Types**

- (1) An enumeration is a special data type that limits the value of a variable to a predefined set of constants.
- (2) Use an enumeration over a set of constants.
- (3) Enumerations may contain additional methods and constructor parameters, see Exhibit 2.5.3-6

2.5.3.8.5.3.9.3

(07-10-2020)

**Nullable Types**

- (1) The use of the “Optional” class eliminates potential null pointer exception checking and runtime errors, making more readable code.
- (2) Apply the “Optional” as return types as shown in Figure 2.5.3-27 Do not use “Optional” as method parameters or in constructors. Consider using “Optional” as field variables for reference types that are settable by outside callers .
- (3) For more information : <https://www.baeldung.com/java-optional>.

**Java Programming Example - Optional Class**

Optional Class
<pre>import java.util.Optional; class Account {     private Optional&lt;String&gt; name = Optional.empty();     public Account(String name) {         this.name = Optional.ofNullable(name);     }     public Optional getName() {         return Optional.ofNullable(name);     } } public class OptionalTest {     static public void main(String[] args) {         OptionalTest test = new OptionalTest();         System.out.printf("Valid account name: [%s]", test.getAccountName(new Account("john")));         System.out.printf("Null account name: [%s]", test.getAccountName(new Account(null)));         System.out.printf("Null account: [%s]", test.getAccountName(null));     }     public String getAccountName(Account acct) {         Optional accountOptional = Optional.ofNullable(acct);         return accountOptional.flatMap(Account::getName).orElse("");     } }</pre>

**Figure 2.5.3-25**

### Java Programming Example - Anonymous Class

Java Programming Anonymous Class
<pre> interface Greeting {     public String getMessage(String name); }  public class AnonymousClassExample {     static public void main(String[] args) {         Greeting englishGreeting = new Greeting() {             public String getMessage(String name) {                 Optional nameOptional = Optional.ofNullable(name);                 return "Hello " + nameOptional.orElse("Anonymous");             }         };          System.out.println(englishGreeting.getMessage("Bob"));         System.out.println(englishGreeting.getMessage(null));     } </pre>

Figure 2.5.3-26

### Java Programming Example - Anonymous Class Output

Anonymous Class Output
<pre> // Output Hello Anonymous 4) Lambda expressions utilize anonymous classes that have only one method signature, like the example above. A typical usage is applying a lambda expression over a list. import java.util.Arrays; import java.util.List; import java.util.Optional; import java.util.function.Consumer; import java.util.function.Function; i} </pre>

Figure 2.5.3-27

#### 2.5.3.8.5.3.9.4 (07-10-2020) Nested Classes

- (1) A class may contain either inner classes or static inner classes within a class definition.
- (2) While permissible, do not create fully defined local classes within a method.
- (3) It is acceptable to create anonymous classes from interfaces from within a method. import java.util.Optional;
- (4) Lambda expressions utilize anonymous classes that have only one method signature, see Figure 2.5.3-28 and Figure 2.5.3-27 . A typical usage is applying a lambda expression over a list. import java.util.Arrays; import java.util.List; import java.util.Optional; import java.util.function.Consumer; import java.util.function.Function

**Java Programming Example - Lambda Expressions Using  
Anonymous Class and Output**

Example of Anonymous Class
<pre> import java.util.Arrays; import java.util.List; import java.util.Optional; import java.util.function.Consumer; import java.util.function.Function;  interface Greeting {     public String getMessage(String name); }  public class AnonymousClassExample {     static public void main(String[] args) {         Greeting englishGreeting = new Greeting() {             public String getMessage(String name) {                 Optional nameOptional = Optional.ofNullable(name);                 return "Hello " + nameOptional.orElse("Anonymous");             }         };         List nameList = Arrays.asList("Bob", "Sally", null, "Fred");         nameList.stream().map(n -&gt; englishGreeting.getMessa- ge(n))             .forEach(msg -&gt; System.out.println(msg));     } } // Output Hello Bob Hello Sally Hello Anonymous Hello Fred </pre>

**Figure 2.5.3-28**

- (5) Use local classes when creating more than one instance of a class, access its constructor, or introduce a new named type to invoke additional methods later.
- (6) Use anonymous classes when implementing an interface is enough, and the instance does not have to exist outside of the method.
- (7) Use lambda expressions when encapsulating a single unit of behavior that needs to be passed to other code.
- (8) Create nested classes when the inner class should be shared with other classes, and access to local variables or method parameters is not required.
- (9) Do not expose the parent class private fields though a public interface in the inner class.
- (10) For more information: <https://docs.oracle.com/javase/tutorial/java/javaOO/whentouse.html> .

2.5.3.8.5.3.9.5  
(07-10-2020)  
**Numeric Types**

- (1) Detect and prevent integer overflow conditions.
- (2) Do not perform bitwise and arithmetic operations on the same variable.
- (3) Ensure division and remainder operations do not result in divide by zero errors.
- (4) Use integer types that can fully represent the possible range of unsigned data.
- (5) Do not use floating point numbers for precise computation.
- (6) Do not attempt comparisons with “NaN”.
- (7) Check floating point inputs for exceptional values.
- (8) Do not use floating point numbers as loop counters.
- (9) Do not construct “BigDecimal” objects from floating-point literals.
- (10) Do not compare or inspect the string value of floating point values.
- (11) Ensure conversions of numeric types to narrower types do not result in lost or misinterpreted data.
- (12) Avoid loss of precision when converting primitive integers to floating point.
- (13) Use shift operators correctly.

2.5.3.8.5.3.9.6  
(07-10-2020)  
**Generics**

- (1) Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods, allowing for class reuse with different inputs.
- (2) Code that uses generics instead of passing instances of “java.lang.Object” benefit from:
  - Stronger type checks at compile time
  - Elimination of casts
  - Enable generic algorithm implementation
- (3) The most common usage of generics can be found in “java.util.Collection” and subclasses.
- (4) The generic parameter must be a reference type and not a primitive

**Java Programming Example - Generic Class**

Example of Generic Class
<pre> class GenericClass &lt;T&gt; {     private T item;     public GenericClass(T item) { this.item = item; }     public T getItem() { return item; } }  public class GenericExample {     static public void main(String[] args) {         GenericClass&lt;int&gt;Example = new GenericClass&lt;&gt;(15);         System.out.println(intExample.getItem() + 22);         GenericClass&lt;String&gt;textExample = new GenericClass&lt;&gt;("Hello");         System.out.println(textExample.getItem() + " World");     } }  // Output 37 Hello World </pre>

**Figure 2.5.3-29**

- (5) The most commonly used type parameter names are:
- E - Element (used extensively by the Java Collections Framework)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S, U, V etc. - 2nd, 3rd, 4th types
- (6) Use the diamond notation as shown above when instantiating a new instance of the generic type (only specify the generic type on the left-side of the expression).
- (7) Generic types may include more than one type, such as "java.util.Map". Limit the number of generic types in the class definition to no more than three .
- (8) Generic classes may have methods that utilize a different generic parameter than defined at the class level. The parameter must be enclosed in angle brackets.



**Java Programming Example - Generic Class using Generic Parameter**

Example - Generic Class
<pre> class GenericClass &lt;T&gt; { private T item;  public GenericClass(T item) { this.item = item; } public T getItem() { return item; } public &lt;N extends Number &gt; void add(N number) { } }                     </pre>

**Figure 2.5.3-30**

- (9) A generic method can be restricted further by specifying that a method parameter must optionally extend a class and implement one or more interfaces.
- (10) A wildcard can be used in generics to specify an upper bound or lower bound. A wildcard with no qualifier will permit any reference value, including classes not derived from “java.lang.Object” such as “java.lang.Number” and null values.
- (11) Consider applying an upper bound to generic method parameters.
- (12) Consider applying a lower bound to generic method return values.
- (13) When using the collection library, methods that accept an “Object” instance should be cast to the same object type as the parameter type used to create the collection.
- (14) For more information: <https://docs.oracle.com/javase/tutorial/java/generics/index.html>

**2.5.3.8.6  
(07-10-2020)  
Statements**

- (1) Program actions are expressed in statements.
- (2) Common types of statements include variable declaration, assigning values through expressions, conditional statements, iteration statements, exception handling, jump, and multithreading statements.
- (3) Use only one statement per line, see figure Figure 2.5.3-31

**Java Programming Example 1- Statement Use**

Statement Use
<pre> // Correct Usage  counter++; itemsRemaining--;                     </pre>

**Figure 2.5.3-31**

- (4) Do not perform additional assignments in an assignment statement, see Figure 2.5.3-32.

**Java Programming Example 2 - Statement Use**

Correct Use of Statement
<pre>// Correct Usage  final int authnum = get(); number = ((31 * (number + 1)) * authnum) + (authnum &gt; threshold ? 0 : -2);</pre>
<pre>// Not this  number = ((31 * ++number) * (number=get())) + (number &gt; threshold ? 0 : -2);</pre>

**Figure 2.5.3-32**

2.5.3.8.6.1  
(07-10-2020)

**Variable Declaration**

- (1) Variable declaration statements consist of the variable type, name, and optionally an initial value.
- (2) Unless the variable will be defined immediately after declaration, always provide an initial value to the declared variable.
- (3) If a method variable, declare the variable immediately prior to usage and not at the top of the method with all the other method variables. see Figure 2.5.3-33

**Java Programming Example - Declaring Variables**

Declaring Variables
<pre>Optional &lt;String&gt; name = Optional.empty(); double area = 0.0d;</pre>

**Figure 2.5.3-33**

2.5.3.8.6.2  
(07-10-2020)

**Expressions**

- (1) Expressions statements may include assignment, object creation with assignment, and method invocation and end with a semicolon

**Java Programming Example - Using Expressions**

Use of Expressions
<pre>area = Math.PI * Math.pow(radius, 2); System.out.println("Hello!"); name = Optional.ofNullable("Bob");</pre>

**Figure 2.5.3-34**

2.5.3.8.6.3  
(07-10-2020)

**Conditional Statements**

- (1) Conditional statements allow certain blocks of code to run based on a certain condition. Two types of structures exist: if/else and switch statements.

- (2) For “If/Else” statements, always surround the embedded statements in curly braces.

**Java Programming Example - Using Conditional Statements**

Conditional Statements
<pre> if (condition) { // statements } else if (condition) { // statements } else { // statements } </pre>

**Figure 2.5.3-35**

- (3) For “switch” statements, all potential values of the variable must be considered in the “case” blocks.
- (4) Consider always using an enumeration in a “switch” statement. If necessary, add a static method in the enumeration to convert the non-enumerated value to an enumerated value.
- (5) A “default” block should be used to catch any value not explicitly considered. A separate “default” block should
- (6) Explicitly provide a comment if one “case” block falls through to another “case” block, Exhibit 2.5.3-9

2.5.3.8.6.4  
(07-10-2020)  
**Iteration Statement**

- (1) Iteration statements are used to loop over a block of code repeatedly until the termination condition is met. Statements in an iteration statement are executed in order unless a jump statement is encountered.
- (2) Valid jump statements include:.
- “break”: exits the iteration to the next statement outside the iteration statement
  - “continue”: goes back to the evaluation statement
  - “return”: exits the function
  - “throws”: throws an exception to be caught in the function or thrown outside the function.
- (3) An iteration statement may be labeled. If an iteration statement is labeled, the “break” and “continue” statements should reference the label unless it is not intended to exit / return to the top-level iteration statement.

**Java Programming Example - Iteration Statement**

Iteration
<pre> loops: for (int i = 0; i &lt; 5; i++) { for (int j = 0; j &lt; 3; j++) { System.out.printf("[i=%d,j=%d]", i, j); if (i == 1) { continue loops; } else if (i == 2 &amp;&amp; j == 1) { break; } else if (i == 3 &amp;&amp; j == 0) { break loops; } } } } </pre>
<pre> // Output [i=0,j=0][i=0,j=1][i=0,j=2] [i=1,j=0] [i=2,j=0][i=2,j=1] [i=3,j=0] </pre>

- (4) The “For” statement is typically used to iterate over a fixed index range. In this form, it is composed of an initialization statement, condition statement, and update statement.
- (5) Variables declared in the initialization statement cannot be accessed outside of the “for” statement.
- (6) Readability purposes - consider doing non-trivial work inside the “for” block instead of doing it all in the update statement, see Figure 2.5.3-36

### Java Programming Example 1- For Statement

For Statement
<pre>// More readable. for (int i = 3; i &gt; 0; i--) {     displayCounter(i); } private void displayCounter(int counter) {     System.out.print(counter + "..."); } // Consider using a different iteration statement. private int timeToLaunch = 3; public void liftOff() {     for (startTime();isReady();countDown(),displayTime()); } private void startTime() { timeToLaunch = 5; } private boolean isReady() { return timeToLaunch &gt; 0; } private void countDown() { timeToLaunch--; } private void displayTime() {     System.out.print(timeToLaunch + "..."); } }</pre>

**Figure 2.5.3-36**

- (7) For statements may contain more than one variable but limit the number to no more than three (3) variables., see Figure 2.5.3-37

### Java Programming Example 2- For Statement with Multiple Variables

For Statement with Multiple Variables
<pre>for (int i = 0, j = 0, k = 0; i * j * k &lt; 100; i += 5, j += 2, k++) {     System.out.printf("i=%d, j=%d, k=%d", i, j, k); }</pre>
<p><b>// Output</b></p> <pre>i=0, j=0, k=0 i=5, j=2, k=1 i=10, j=4, k=2</pre>

**Figure 2.5.3-37**

- (8) A “for” statement may also be used to iterate over a collection. In this form, it is composed of a reference variable and a collection.
- (9) When iterating over a collection, do not modify that collection, see Figure 2.5.3-38 below.

**Java Programming Example 3- For Statement with Reference Variable & Collection.**

<b>For Statement with Reference Variable and Collection.</b>
<pre>List&lt;String&gt; nameList = Arrays.asList("Bob", "Sally", null, "Fred"); for(String s : nameList) { }</pre>

**Figure 2.5.3-38**

- (10) Consider replacing iterating over a collection with a “for” statement and use a lambda expression instead.
- (11) If a statement block should run until a condition is met and does not need to run once, use a “while” statement.
- (12) Insure that the “while” statement is condition will be met. Avoid using an empty “while” condition.
- (13) If a statement block should run until a condition is met and must run once, use a “do-while” statement.
- (14) Insure that the “do-while” statement is condition will be met. Avoid using an empty “do-while” condition., see Figure 2.5.3-39

**Java Programming Example - Do While Statement**

<b>Do While Statement</b>
<pre>int rating = 90; do {     rating -= 5; } while (rating &gt; 50);</pre>

**Figure 2.5.3-39**

**2.5.3.8.6.5**  
(07-10-2020)  
**Empty Statement**

- (1) The empty statement is used to indicate that a no operation should take place.
- (2) Empty statements should be used sparingly since it can be easily missed during a code walkthrough, see Figure 2.5.3-40 .

**Java Programming Example - Empty Statement**

<b>Use of Empty Statement</b>
<pre>for (;repeatUntilLimitReached()); if (arg == "right");</pre>

**Figure 2.5.3-40**

**2.5.3.8.6.6**  
(07-10-2020)  
**Assertion Statement**

- (1) Use assertion statements in test projects only. Do not use assertions in place of exception handling or other validation methods.
- (2) Do not modify the state of any local or class variable in an assertion statement.

2.5.3.8.7  
(07-10-2020)  
**Expressions**

- (1) An expression is a sequence of one or more operands and zero or more operators that can be evaluated to a single value, object, method, or namespace. Expressions can consist of a literal value, a method invocation, an operator and its operands, or a simple name. Simple names can be the name of a variable, type member, method parameter, namespace or type.
- (2) Numeric expressions may cause overflows if the value is larger than the maximum value of the value's type. In Java, exceeding the maximum value will roll the value over to the minimum value, see Figure 2.5.3-41 .

**Java Programming Example 1 - Using Expressions**

Expressions and Output
<pre>public class ExpressionExample {     static public void main(String[] args) {         int largeNumber = Integer.MAX_VALUE - 10;         largeNumber += 20;         System.out.println(largeNumber);     } }</pre>
<pre>// Output  -2147483639</pre>

**Figure 2.5.3-41**

- (3) Expressions are evaluated by the rules of associativity and operator precedence. For clarity, use parentheses extensively to indicate the precedence of evaluating the expression, see Figure 2.5.3-42

**Java Programming Example - Expression Rules**

Expression Rules
<pre>x + y / 100 // ambiguous (x + y) / 100 // unambiguous, recommended</pre>

**Figure 2.5.3-42**

- (4) Do not ignore values returned by methods.

2.5.3.8.7.1  
(07-10-2020)  
**Lambda Expressions**

- (1) When trying to pass functionality as an argument to another method, lambda expressions treat functionality as a method argument, or code as data.
- (2) Lambda functions have the following functionality blocks:
  - Collection: one source collection to process, like a list
  - Predicate: zero to many filters to apply to the collection
  - Function: optional method that transforms the collection item to something else
  - Consumer: single function or code block that processes the transformed item
- (3) Consider using aggregate operations that accept lambda expressions as parameters over creating boilerplate code that iterates over the collection.

- (4) Detailed walkthrough: <https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>.

#### 2.5.3.8.8 (07-10-2020)

##### Operators

- (1) An operator is a program element that is applied to one or more operands in an expression or statement.
- (2) A unary operator contains a single operator and operand, e.g. `y++`;
- (3) A binary operator, typically an assignment operation, contains the variable being assigned and the expression that is assigned to that variable. The expression can contain one or more operands.
- (4) To avoid exceptions and increase performance by skipping unnecessary comparisons, use “&&” or “||” instead of “&” and “|” respectively.
- (5) A ternary operator contains a condition expression, followed by expression if the condition is met and an expression if the condition is not met.
- (6) If the condition expression contains a binary operator, surround the expression in parentheses, see Figure 2.5.3-43

#### **Java Programming Example - Using Operators**

Binary Operator
<code>int formNeeded = (filer == Filer.INDIVIDUAL) ? 1040 : 8832;</code>

**Figure 2.5.3-43**

- (7) For additional information: <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html>

#### 2.5.3.8.9 (07-10-2020)

##### Member Design

- (1) This section contains guidelines for designing members found in classes and interfaces.

#### 2.5.3.8.9.1 (07-10-2020)

##### Member Overloading

- (1) Member overloading is the process of using the same member name for different members that vary only by the input parameters.
- (2) Overloading by parameter count makes it possible to provide simpler versions of constructors and methods by chaining them together
- (3) Overloading by parameter type makes it possible to use the same member name for members performing identical operations on a selected set of different types, see Figure 2.5.3-44



**Java Programming Example - Member Overloading**

Example of Overloading by Parameter Type
<pre> public class PrintStream { public PrintStream(File file) { } public PrintStream(File file, String csfn) { } public format(Locale l, String format, Object... args) { } public format(String format, Object... args) { } } </pre>

**Figure 2.5.3-44**

- (4) Use descriptive parameter names to indicate the default used by shorter overloads.
- (5) Avoid varying parameter names in overloads. If a parameter in one overload represents the same input parameter in another overload, they should have the same name.
- (6) Avoid being inconsistent in parameter order in overloads.
- (7) Subclasses should only override the longest overload since the shorter overloads should simply call the longer one with default values.
- (8) Do not have overloads with same type and position as other overloads but have completely different meanings.
- (9) Allow “null” to be passed for optional arguments.
- (10) Constructors can either be an instance constructor or a type constructor.

2.5.3.8.9.2  
(07-10-2020)  
**Constructor Design**

- (1) Constructors are the most natural way to create instances of a type. Most developers will search and try to find a constructor before looking for alternative methods, such as factory methods.
- (2) Constructors can either be an instance constructor or a type constructor. Exhibit 2.5.3-9 and Exhibit 2.5.3-10
- (3) Always provide a default (no parameter) constructor.
- (4) Consider providing simple constructors containing primitive parameters for properties that are commonly populated.
- (5) Consider using a static factory method instead of a constructor if it doesn't seem natural to use a constructor. For example: class “java.util.Calendar”, method “getInstance()”..
- (6) Do use constructor parameters as shortcuts for setting main properties.
- (7) Do use the same name for constructor properties and a property if the constructor simply sets the property.

**Java Programming Example 1 - Constructor Design**

Example - Constructor Design
<pre>public DesignExample(LocalDateTime instanceDate) { this.instanceDate = instanceDate; }  { return instanceDate; }</pre>

**Figure 2.5.3-45**

- (8) Do minimal work in the constructor. This eliminates the need to throw exceptions when trying to create instances.
- (9) If appropriate, do throw exceptions from instance constructors.
- (10) Do not throw exceptions from type constructors.
- (11) Instead of type constructors, consider initializing static fields when defined.

**Java Programming Example 2 - Constructor Design**

Constructor Design - Initializing static fields when defined.
<pre>// Do this static LocalDateTime firstDate = LocalDateTime.now(); // Instead of this static LocalDateTime firstDate; static { firstDate = LocalDateTime.now(); }</pre>

**Figure 2.5.3-46**

- (12) For sensitive classes, utilize a static method to create instances over a constructor. This allows for security checks to take place for an object is created. Return null if an exception occurred during creation.
- (13) Prevent the construction of sensitive classes. Pre-create instances and share them with classes that need that instance.
- (14) Do not call methods that can be overridden in a constructor.
- (15) For more information: <https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html>.

2.5.3.8.9.3  
(07-10-2020)  
**Finalizer Design**

- (1) The “Object.finalize()” method is intended to be called just prior to an object being claimed by the garbage collector. However, use of the “finalize” method leads to performance issues, hangs, and deadlocks. It is completely unknown when the garbage collector will pick up an inaccessible object.
- (2) If using a pooled resource, always explicitly release the resource when finished using the resource.
- (3) For example, a database connection is needed to work over several non-concurrent operations. Instead of creating a new connection on each call, a connection pool can be established on instance creation / “open” method. Sub-

sequent calls to that instance use the connection pool established. When the instance is no longer needed, call a “close” method that disposes all the connections in the pool.

- (4) To prevent malicious classes from implementing or overriding the method “finalize” and obtaining sensitive information in an instance, perform a security manager check before calling the super constructor.

#### 2.5.3.8.9.4 (07-10-2020) Field Design

- (1) The principle of encapsulation, a cornerstone of object-orientated programming, states that data stored within an object should be accessible to only that object.
- (2) Do not provide instance fields that are public or protected. Use property methods for accessing fields.
- (3) Do use public constant fields that cannot be changed.
- (4) For more information <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/field>

#### 2.5.3.8.9.5 (07-10-2020) Property Design

- (1) Java properties are accessible using the “getProperty()” and “setProperty()” methods. For boolean properties, use “isProperty()” instead of “getProperty()”.
- (2) If explicit element control is needed for a collection property, replace “setProperty()” with “addItem()” and optionally “removeItem()” methods instead.
- (3) If the property should not be changed after creation, require the property in the instance constructor and only provide a “getProperty()” method.
- (4) Do not make the “setProperty()” method more accessible than the “getProperty()” method. For example, do not make the setter “public” and the getter “protected”.
- (5) Provide only property methods that are needed and assign them with the most restrictive modifier permissible. Provide only property methods that are needed and assign them with the most restrictive modifier permissible.
- (6) Provide sensible default values for all properties, ensuring that the defaults do not result in a security hole, inefficient code, or lead to null pointer exceptions.
- (7) If two or more properties need to be set together, consider providing a method that sets them at the same time instead of two individual setter methods.
- (8) If a setter method throws an exception, preserve the previous value of that property.
- (9) Avoid throwing exceptions from getter methods.
- (10) If necessary, collect a list of callers that can listen for property changes. When the property changes, notify the listeners that the event occurred.
- (11) Follow input validation and data sanitization rules listed below.

#### 2.5.3.8.9.5.1 (07-10-2020) Abstract Properties

- (1) Create abstract properties where a family of classes have a common property but the base class cannot actually implement the property, see Exhibit 2.5-3-11

2.5.3.8.9.5.2  
(07-10-2020)  
**Constants**

- (1) Constants are fields set at compile time and can never be changed. Always mark a constant value as “final”
- (2) Use constants over literals for special values.
- (3) Consider using an enumeration if a set of constants can be logically grouped together.
- (4) If an enumeration does not make logical sense but a set of constants could be considered global, or potentially be associated with many classes, consider creating a “final” class that holds all the related constants together.
- (5) Constant values in interfaces are by default public, static and final.
- (6) Only use immutable types, such as numbers, strings, or custom types, as constant types, see Figure 2.5.3-46

**Java Programming Example - Constants**

**Using Immutable Types: Numbers, Strings, or Custom Types, as Constant Types**

```
static final int READ_ONLY = 1;
final class PhysicalConstants {
    static final public int SPEED_OF_LIGHT = 299_792_458;
    // m/s static final public double GRAVITATIONAL_CONSTANT =
    6.67408e-11;
    // N m^2/kg^2
    static final public double PLANCK_CONSTANT = 6.626_070_15e-34;
    // J*s
}
```

**Figure 2.5.3-47**

2.5.3.8.9.6  
(07-10-2020)  
**Parameter Design**

- (1) Parameters are primitives and reference variables that are part of a method or constructor member.
- (2) Use the least derived parameter type that provides the functionality required by the member. For example, if a parameter consists of a list and the method does not need to access elements by index, have the method accept a “java.util.Collection” instance instead.
- (3) Do not use reserved parameters. Every parameter passed into a member should be used in some way within that member.
- (4) Do not have public methods that use multi-dimensional arrays as parameters. Redesign the API to pass a collection of an object that contains a collection.
- (5) Be consistent in naming parameters when overriding members or implementing instance members.
- (6) If a method requires two or more boolean parameters, consider using an enumeration value instead.
- (7) Do not use boolean parameters unless it is absolutely certain that the parameter will never need two states.

2.5.3.8.9.6.1  
(07-10-2020)  
**Variable Length  
Parameter**

- (8) Consider using boolean values for constructor parameters that are truly two-state values and that initialize boolean properties.
- (9) Never use assertions to validate method parameters.

- (1) Members may have a single parameter that can take a variable number of arguments. This parameter must be the last parameter in the member definition.

**Java Programming Example - Variable Length Parameter**

Using a Variable Length Parameter
<pre>class Polygon extends Shape { int[] sides = new int[0]; public Polygon(int... sides) { this.sides = sides; } } Polygon poly = new Polygon(4, 5, 12, 13);</pre>

- (2) If callers will typically pass in large quantities of a particular type or always pass in a collection of items, consider having the method simply accept a collection instead of a variable length parameter.
- (3) Do not use the single parameter form above if the array will be modified by the member.
- (4) Consider using a variable length parameter in a simple overload, even if a more complex overload could not use it.
- (5) Try to order parameters to make it possible to use a variable length parameter.
- (6) Consider providing special overloads and code paths for calls with a small number of arguments instead of variable length parameters when performance is critical.
- (7) Do check a variable length parameter for “null” values. Polygon poly = new Polygon((int[])null);

2.5.3.8.9.6.2  
(07-10-2020)  
**Event Design**

- (1) Events are the most commonly used form of callbacks.
- (2) Two common groups of events include events raised before state changes and events raised after state changes. Most events in the AWT are in the latter category, such as in the class “MouseListener”: “mouseDragged”, and “mouseMoved”.
- (3) To create a new event:
  - Create an event object
  - Create a listener interface that has one or more events that could be triggered with the event object.
  - A class that holds a set of listeners that are notified when an event is triggered

- (4) If a listener interface contains several events, create a listener adapter class that provides no-operation implementations for each event and subclasses then override only the events that the class is interested. For example, the interface “MouseListener” is implemented by the class “MouseListenerAdapter”
- (5) Use the term “raise” for event notification methods instead of “fire” or “trigger”.
- (6) Consider extending the event class from “java.util.EventObject” instead of using the base class.
- (7) Make the event notification method protected for subclasses to override the event method. Use private only when the class cannot be subclassed.
- (8) Only pass in the event object into the event notification method. If additional properties are needed, enhance the event object to contain those properties. The parameter name should be “e”.
- (9) Only pass “null” as the event object source only for static classes when the instance object is unavailable or not important.
- (10) Do not pass “null” into event properties.
- (11) If the event raised is a pre-event, consider adding a mechanism that can cancel the event from ultimately taking place.
- (12) Event methods should always return “void”.
- (13) The first parameter in an event should always be of type “Object” and be named “source”.

2.5.3.8.9.7  
(07-10-2020)

#### Methods

- (1) Do not use deprecated or obsolete methods.
- (2) Do not increase the accessibility when overriding methods.

2.5.3.8.9.8  
(07-10-2020)

#### Language Guidelines

- (1) Coding conventions for creating: arrays, exceptions, improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

2.5.3.8.9.8.1  
(07-10-2020)

#### Arrays

- (1) Use concise syntax when initializing arrays., see Figure 2.5.3-48

#### Java Programming Example - Syntax Readability

Readable Code
// Preferred notation. String[] nameArray = { "Bill", "Joan", "Ted" }; String[] nameArray = new String[] { "Bill", "Joan", "Ted" };
// Have to initialize each element one at a time. String[] sizedArray = new String[5]; sizedArray[0] = "Bill";

Figure 2.5.3-48

2.5.3.8.9.9  
(07-10-2020)  
**Exceptions**

- (1) Use exception handling when execution failure occurs, disrupting the normal flow of statements in a member. Do not return error codes.
- (2) Use exceptions to separate error-handling code from regular code.
- (3) Errors thrown are propagated up the call stack. If a method chooses not to resolve the exception thrown by a statement, the method can simply throw the exception to its caller.
- (4) Exceptions can be grouped by execution failure type, such as input/output or illegal argument errors. Avoid using general purpose exception handlers such as "java.lang.Exception" as the sole exception handler.
- (5) Checked exceptions must be caught and handled as described below.
- (6) Exceptions that are external to the application that cannot be anticipated or recovered from extend from "java.lang.Error". Applications should not try to catch these exceptions except to perform logging or display a notification.
- (7) Exceptions that are internal to the application that cannot be anticipated or recovered from extend from "java.lang.RuntimeException". Consider designing the application to eliminate the sources of these exceptions rather than simply catching them.
- (8) Do not use exceptions for the normal flow of control. By their name, exceptions should be raised only in exceptional circumstances.
- (9) When logging exceptions, always sanitize content that came from an untrusted source.
- (10) Consider another mechanism instead of exceptions if the application will throw more than 100 exceptions per second.
- (11) Document all exceptions publicly thrown in the documentation comments.
- (12) Do not have public members that can turn off a thrown exception by setting a parameter.
- (13) Do not return exceptions in the return value.
- (14) Consider using exception builder methods to create a new exception instance before being sent.
- (15) Avoid throwing exceptions from "finally" blocks.
- (16) For more information: <https://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html>.
- (17) For more information on errors *More information on errors:* <https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/Error.html>.

2.5.3.8.9.9.1  
(07-10-2020)  
**Catching and Handling Exceptions**

- (1) A "try" statement starts a block of code where exceptions may be thrown and a corresponding set of exception handlers ends the block to capture the types of exceptions that the block may throw.
- (2) Associate exception handlers with one or more "catch" blocks that follow directly after the "try" block. The exception type indicates exception the "catch" block can handle.

- (3) Exception handlers can handle more than one type of exception that are not related to each other by using the pipe “|” operator.
- (4) Organize exception handlers from most specific to most general.
- (5) Always use the first letter of the exception name parts as the exception handler variable name.
- (6) When using a resource that is only needed in a “try” block, specify the resource with the “try” statement to insure it is closed on exit.
- (7) Optionally, the method can choose to simply pass the exception to the caller with or without handling the exception first.
- (8) Always do something with the exception, catch or throw it up to the caller, but never both, see Figure 2.5.3-49 for more guidance.

**Java Programming Example - Catching and Handling Exceptions**

Catching/Throwing an Exception
<pre> import java.io.FileInputStream; import java.io.FileNotFoundException; import java.io.IOException; import java.io.InvalidClassException; import java.io.ObjectInputStream; class Person { private String id; public Person(String id) { this.id = id; } public String getID() { return id; } } public class ExceptionExample { static public void main(String[] args) throws Exception { Person p = null; try (ObjectInputStream objectStream = new ObjectInputStream(new FileInputStream(args[0]))) { p = (Person)objectStream.readObject(); } catch (FileNotFoundException fnfe) { // do something } catch (ClassNotFoundException InvalidClassException se) { // do something } catch (IOException ioe) { // do something } } } </pre>

**Figure 2.5.3-49**

- (9) Never return the method “printStackTrace()” back to the client application.
- (10) Never catch instances of “java.lang.Throwable”.



	(11) Clean up any resources not controlled in the resource section of the “try” block.
	(12) Never explicitly catch “NullPointerException”. Rewrite the code to mitigate the null condition.
2.5.3.8.9.9.2 (07-10-2020) <b>Throwing Exceptions</b>	<p>(1) Never throw an exception from the “finally” block.</p> <p>(2) When creating custom exception classes, use the initial exception as a parameter to the custom exception instance so the stack trace is not lost.</p> <p>(3) Always provide context of what occurred in the exception thrown with the stack trace information for development purposes.</p> <p>(4) .Only include known, acceptable information in exception details rather than trying to filter out the sensitive properties.</p> <p>(5) Do not include file path information, account names, or home directory information that enable hackers to guess the underlying file or data structure.</p> <p>(6) Consider sanitizing the exception type name, such as “FileNotFoundException”.</p>
2.5.3.8.9.9.3 (07-10-2020) <b>Unchecked Exception Best Practices</b>	<p>(1) Don’t convert checked exceptions to unchecked (runtime) exceptions.</p> <p>(2) If the method emits unchecked exceptions, such as a “NullPointerException”, document when that will occur but do not specifically indicate that in the method definition.</p> <p>(3) Do not create exceptions that are extended classes of “java.lang.RuntimeException”.</p> <p>(4) General rule: if the caller can recover from an exception, make the method throw a checked exception. If the caller cannot recover from an exception, make the method throw an unchecked exception.</p>
2.5.3.8.9.10 (07-10-2020) <b>Concurrency</b>	(1) Applications can utilize multiple threads of the executing processor using concurrent programming techniques.
2.5.3.8.9.10.1 (07-10-2020) <b>Threads</b>	<p>(1) Threads are lightweight processes that enable different blocks of code to run concurrently</p> <p>(2) To directly control thread creation and management, create instances of class “Thread” to initiate an asynchronous task..</p> <p>(3) To create a thread, create a class that implements the interface “java.lang.Runnable” and use it to create a new “Thread” instance. Do not subclass from the class “java.lang.Thread”, see Figure 2.5.3-50 for more guidance.</p>

**Java Programming Example - Thread Creation****Class that Implements the Interface “java.lang.Runnable” for New Thread Instance**

```

class HelloRunnable implements Runnable {
    @Override public void run() {
        System.out.println("Hello from a thread.");
    }
}

public class ThreadExample {
    static public void main(String[] args) {
        Thread t = new Thread(new HelloRunnable());
        t.start();
    }
}

```

**Figure 2.5.3-50**

- (4) A thread can be paused, or put to sleep, for certain duration. The thread will pause for at least the specified amount of time but not guaranteed to be precise.
- (5) When a thread is interrupted, the thread should terminate and clean up any used resources.
- (6) A thread can be paused by waiting on another thread to complete by joining them together.
- (7) Use synchronized methods to prevent shared data from being modified by multiple threads or receiving inconsistent views.
- (8) Consider using synchronized blocks only when it is safe to interleave access of the affected fields.
- (9) Insure thread operations are independent of each other to avoid a deadlock condition.
- (10) Insure that threads lock resources only when necessary and promptly release the lock to avoid starvation by other threads.
- (11) If a thread must react to the action taken by another thread, do not have the original thread react to the action taken by the other thread.
- (12) Always invoke the method “wait” inside a loop that tests for the condition being waited for. Do not assume that the interrupt was for the particular condition or that the condition is true.
- (13) Use the method “Object.notifyAll” method to notify every thread that the resource has been updated., see Exhibit 2.5.3-13
- (14) Consider defining immutable classes so that the properties of an object cannot change once created. This eliminates the concurrency issues when working with threads.
- (15) Avoid starting a new thread with an untrusted object of type “Runnable”.

- (16) For more information about threads see, <https://docs.oracle.com/javase/tutorial/essential/concurrency/threads.html>.

2.5.3.8.9.10.2  
(07-10-2020)

#### High-Level Concurrency

- (1) High-level concurrency features are located in the package “java.util.concurrent”.
- (2) Unlike implicit lock objects in synchronized blocks, the class “Lock” allows a thread to try to get the lock and back out if it cannot.
- (3) Instead of managing threads directly, specific implementations of the interface “Executor” determine when and how the thread is created.
- (4) To process independent tasks according to the scheme “one thread per task”, use the interface “ExecutorService” to run standard “Runnable” tasks, which do not return a value, and “Callable” tasks which do return a value. This service can manage the status of tasks, submit large quantity of tasks, and gracefully shut down tasks within the executor.
- (5) To schedule a task to run in the future, use the interface “ScheduledExecutorService”.
- (6) Always shutdown an executor service when it is no longer needed.
- (7) Consider creating thread pools with a finite number of threads managed by the executor. Be sure to size the thread pool properly to optimize the pool overhead with the number of concurrent requests expected to be received., see Figure 2.5.3-51

**Java Programming Example - Creating Thread Pools and Output****Thread pools with a Finite Number of Threads**

```

import java.util.Arrays;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException; import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ExecutorExample {
    static public void main(String[] args) {
        Runnable helloTask = () -> {
            System.out.println("Hello from a thread.");
        };
        Callable callbackTask = () -> {
            return "Here is your sign.";
        };
        ExecutorService service = Executors.newFixedThreadPool(10);
        List > callbackList = Arrays.asList(callbackTask);
        try {
            service.execute(helloTask);
            String result = service.invokeAny(callbackList);
            System.out.println(result);
        }
        catch (InterruptedException e) {
        }
        catch (ExecutionException e) {
        }
        service.shutdown
    }
}

```

**// Output**

```

Hello from a thread.
Here is your sign.

```

**Figure 2.5.3-51**

- (8) Utilize “Future” instances from the executor when it is necessary to monitor or cancel a long running task.
- (9) Use “CompletableFuture” instances when the following conditions are needed:
  - The task may need to be manually completed
  - Further action may be needed on a “Future” result without blocking
  - Need to chain or combine multiple features together
  - Require exception handling
- (10) Consider using the fork/join framework instead of an executor service for breaking down a task into recursively smaller tasks and then have the results collated.
- (11) For more information on executors, see *More information on executors: <https://www.baeldung.com/java-executor-service-tutorial>*.

2.5.3.8.9.11  
(07-10-2020)  
**Native Code  
Interoperability**

- (12) For more information on futures and “Fork/Join” see, <https://www.baeldung.com/java-future>.
  - (13) For more information on “Completable Futures” see, <https://www.baeldung.com/java-completablefuture>.
- 
- (1) The Java Native Interface (JNI) is a gateway to allow Java code to work with non-Java resources.
  - (2) Use JNI in the following situations:
    - Integrate with legacy code to avoid a rewrite
    - Implement functionality missing in Java libraries
    - Integrate with code best written in C or C++
    - Address special circumstances that must be resolved by leaving the Java Virtual Machine
  - (3) When accessing fields and methods in Java objects, cache any results when finding resource identifiers.
  - (4) Get or update only the parts of an array that the native method needs.
  - (5) Get or update as much of the array at one time.
  - (6) Provide all the information the native method needs to execute in method parameters.
  - (7) Minimize the transitions between native code and Java callbacks. This includes structuring the data, so it exists on the correct side of the boundary.
  - (8) Delete local references when no longer used and not wait until the native method call ends.
  - (9) If many local references are needed (more than 16), notify the JVM to optimize the handling of local references.
  - (10) Only use the “JNIEnv” with the single thread to which it is assigned.
  - (11) Always check for exceptions after making JNI calls that can raise exceptions.
  - (12) Always check the return value from a JNI method and include code paths to handle errors.
  - (13) For each “GetXXX” call, always call “ReleaseXXX” when the resource is no longer required.
  - (14) Never make an additional JNI call or block code running between the “GetXXXCritical” and “ReleaseXXXCritical” statements.
  - (15) Always keep track of global references and delete them when the reference is no longer needed.
  - (16) Define wrappers around native methods by making the native method call private and use a public Java method that calls the native method.
  - (17) For more information see, <https://www.ibm.com/developerworks/library/j-jni/index.html>. Exhibit 2.5.3-13

- 2.5.3.8.9.12  
(07-10-2020)  
**Design for Extensibility**
- (1) Carefully consider how the framework can be extended over time.
  - (2) Always choose the least costly extensibility mechanism that meets the requirements. Adding more extensibility later is easier than attempting to take it away. The sections below are organized from least costly to most costly.
- 2.5.3.8.9.12.1  
(07-10-2020)  
**Unsealed Classes**
- (1) Consider making all new classes unsealed without protected members and make public methods “final” to prevent overrides.
- 2.5.3.8.9.12.2  
(07-10-2020)  
**Protected Members**
- (1) Consider using protected members for advanced customizing without exposing features to classes outside the package or class hierarchy.
  - (2) Use the same defensive coding practices on protected members as public members. This includes documentation, security, and compatibility analysis.
- 2.5.3.8.9.12.3  
(07-10-2020)  
**Events and Callbacks**
- (1) Throw standard checked exceptions on methods when it makes sense to do so.
  - (2) Throw custom checked exceptions when the callers can reasonably continue to operate if the exception is thrown.
  - (3) Avoid throwing runtime exceptions and instead sanitize the input or throw a checked exception.
  - (4) Consider accepting standard lambda expressions as parameters to methods to execute custom code when useful.
  - (5) Consider by accepting standard lambda expressions or custom functional interfaces, arbitrary code is being executed, leading to potential security, correctness, and compatibility issues, see Figure 2.5.3-52.

**Java Programming Example - Events and Callbacks**

Events and Callbacks
<pre> class NamedEmployee {     private LocalDate loginDate = LocalDate.MIN;     private String name = "";     public NamedEmployee() {     }     public NamedEmployee(String name) {         this.name = name;     }     public String getName() { return name;     } } public LocalDate getLoginDate() { return loginDate; } public void setLoginDate(LocalDate loginDate) {     this.loginDate = loginDate; } }  public class PassByFunction {     static public void main(String[] args) {         Map&lt;String, LocalDate&gt;employeeData = new HashMap&lt;&gt;();         employeeData.put("Fred", LocalDate.of(2018, 5, 4));         employeeData.put("Sally", LocalDate.of(2018, 10, 16));          Function&lt;Map.Entry&lt;String, LocalDate&gt;, NamedEmployee&gt; convert-         Function = (entry) _&gt; {             NamedEmployee emp = new NamedEmployee(entry.getKey());             emp.setLoginDate(entry.getValue());             return emp;         };          PassByFunction app = new PassByFunction();         app.writeToDatabase(employeeData.entrySet(), convertFunction);     }     public &lt;T&gt; void writeToDatabase(Collection&lt;T&gt; source,         Function&lt;T,NamedEmployee&gt; func) {         source.stream().map(func).forEach(this::writeRecord);     }     private void writeRecord(NamedEmployee emp) {         System.out.printf("Name=%s,Date=%s\n", emp.getName(),         emp.getLoginDate());     } } </pre>
<pre> // Output Name=Sally,Date=2018-10-16 Name=Fred,Date=2018-05-04 </pre>

**Figure 2.5.3-52**

2.5.3.8.9.12.4  
(07-10-2020)  
**Virtual Members**

- (1) Overriding members, like methods, seems natural to object-orientated design by changing the behavior of the base class.
- (2) An overridden method is costly to design, test, and maintain because of the potential impact on related methods in the base class.

- (3) The contract and documentation on methods that can be overridden must be extremely clear to subclass implementations.
- (4) Non-final methods are much slower than final methods because they cannot be optimized by the compiler.
- (5) Use protected accessibility over public accessibility for methods that can be overridden.
- (6) When the public method results vary between subclasses, have the public method call a protected method that can be overridden.

**Java Programming Example - Overriding Members (Methods)**

**Using Protected Accessibility over Public Accessibility for Methods Overridden**

```

abstract public class Shape {
    final public double getArea() {
        return calculateArea();
    }
    abstract protected double calculateArea();
}
class Circle extends Shape {
    private int radius = 0;
    public Circle() {
    }
    public Circle(int radius) {
        this.radius = radius;
    }
    final public int getRadius() { return radius; }
    final public void setRadius(int radius) {
        this.radius = radius;
    }
    @Override
    protected double calculateArea() { return Math.pow(radius, 2) *
        Math.PI; }
}

```

**Figure 2.5.3-53**

2.5.3.8.9.12.5  
(07-10-2020)  
**Abstractions**

- (1) Abstract classes provide a contract with partial implementation details. It is extremely difficult to design an abstraction that provides just the right amount of functionality and no more for subclasses to use.
- (2) Too many abstractions make the overall framework difficult to understand and use. In addition, poor naming choices can lead to confusion over which classes are abstract and which are not.
- (3) Abstractions are an essential part of many architectural patterns and extremely important for framework testing.
- (4) Do not provide abstractions unless they are tested by several concrete implementations and the APIs that consume the abstractions.



- (5) Unless the abstract class provides significant reusability to subclasses, consider using an interface instead.
  - (6) Provide reference tests for concrete implementations of abstract classes. These tests will enable developers to test that their implementations correctly implement the contract.
- 2.5.3.8.9.12.6  
(07-10-2020)  
**Base Classes for Implementing Abstractions**
- (1) Base classes are abstract classes that extend from another base class.
  - (2) Base classes add complexity to the framework and increase the depth of the inheritance hierarchy.
  - (3) Only create base classes that provide significant functionality from its base class to developers using the framework and not for other framework components. For internal framework components, delegate the functionality to an internal implementation.
  - (4) Make base classes abstract even if they do not contain abstract methods to clearly indicate the call must not be used directly.
  - (5) Place base classes in a separate namespace from the mainline scenario types.
  - (6) Be aware that secure subclasses may become unsecure by adding new functionality to base classes, or default interface methods.
- 2.5.3.8.9.12.7  
(07-10-2020)  
**Sealing**
- (1) Consider making a class final to prevent malicious subclassing or for the following reasons:
    - The class contains only static methods and cannot be instantiated.
    - The class stored security-sensitive secrets in inherited protected members.
    - The class contains many overridable members, and the cost of sealing them individually is an expensive operation.
  - (2) Do not declare protected or overridable members in sealed classes since no additional extension classes can exist.
  - (3) Consider sealing overridden members.
- 2.5.3.8.9.13  
(07-10-2020)  
**Secure Coding Guidelines**
- (1) The following guidance is for the standard edition (Java SE).
  - (2) For more information on secure coding see,: <https://www.oracle.com/technetwork/java/seccodeguide-139067.html> and <https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java>.
- 2.5.3.8.9.13.1  
(07-10-2020)  
**Fundamentals**
- (1) Design and write code that does not require clever logic to see it is safe.
  - (2) Insure that only vetted classes are used in the framework design. Using a subclass from an unknown source may contain malicious code that adds finalizers or overrides random methods.
  - (3) Inspect any method or class that utilizes the class “SecurityManager”.
  - (4) Refactor any duplicated code or data so that changes are uniformly accessed and modified throughout the application.

- (5) Restrict privileges of the Java Virtual Machine by assigning policy files that restrict permissions. Avoid running applications with all permissions.
- (6) Sanitize and validate all data crossing a trust boundary. For example, data contained in a web request must be validated before business logic is applied. Permanent storage, like a database, must validate that the business layer data is correct before updating the storage facility.
- (7) Minimize the number of security checks to “SecurityManager” by getting a set of permissions only at key points and using that information when needed.
- (8) Classes, packages, and modules should only contain a coherent set of behaviors and nothing more.
- (9) Test all code by performing: peer reviews, unit testing, and regression testing. This will ensure all application defects, design, and security flaws are mitigated

2.5.3.8.9.13.2  
(07-10-2020)

#### Denial of Service

- (1) Be cautious of the following:
  - Large vector images such as SVG and font files
  - Creating object graphs from a text or binary streams
  - Highly compressed ZIP files
  - XML files that dynamically grow on entity expansion
  - Inserting large numbers of keys in a map with the same hashcode
  - Regular expressions that may have catastrophic backtracking
  - XPath expressions that consume arbitrary amounts of processor time
  - Deserializing malicious data
  - Integer overflow errors
  - Detailed log entries that produce excessive output
  - Parsing corner case data that results in infinite loops
- (2) Always release resources when they are no longer needed. Consider using “execute around method” and “try with resource” idioms for handling resources. Use the standard resource and acquisition and release pattern for resources that cannot use either idiom, see Figure 2.5.3-54

#### *Java Programming Example - Releasing Resources*

Using “Try with Resource”
<pre> public R locked(Action action)  {     lock.lock();     try {         return action.run();     } finally {         lock.unlock();     } } </pre>

**Figure 2.5.3-54**

- (3) If outputting data to a stream, always flush the buffer before closing the stream.
- (4) Insure that accessing a resource cannot indefinitely block or uses untrusted code that prevents the cleanup code from executing.

	(5) Consider using the methods in the class “java.util.Math” for arithmetic calculations if there is a possibility for integer overflow errors.
2.5.3.8.9.13.3 (07-10-2020) <b>Confidential Information</b>	(1) Do not log sensitive information, such as social security numbers or passwords. This includes working with low-level libraries that utilize generic text. Consider using only one-way hashes for password checks and flushing object content directly rather than waiting for the garbage collector to dispose of the object.
2.5.3.8.9.13.4 (07-10-2020) <b>Input Validation and Data Sanitization</b>	<p>(1) Validate return values from called methods before using them.</p> <p>(2) Do not expose collections that can be modified outside of the class without first providing either a copy or making the collection unmodifiable.</p> <p>(3) Validate all arguments passed to public, protected, or explicitly implemented members. If an argument is not valid, throw an appropriate runtime exception <a href="https://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html">https://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html</a> such as “IndexOutOfBoundsException” or “IllegalArgumentException”.</p> <p>(4) Be aware that mutable object content may change while using the object. This is especially true in multi-threaded situations, but even in normal processing the object properties may change during the lifecycle of a method.</p> <p>(5) Always validate input from untrusted sources. This includes method arguments and external streams.</p> <p>(6) Always make a defensive copy of reference values before performing validating input.</p> <p>(7) Consider making copies of internal reference properties prior to sending to external methods.</p> <p>(8) Normalize string values and do not form strings with partial characters</p> <p>(9) Canonicalize path names and use a safe subset of ASCII characters before validating them.</p> <p>(10) Safely extract files from a compressed source.</p> <p>(11) Exclude unsanitized user input from format strings.</p> <p>(12) Sanitize untrusted data passed to the method “Runtime.exec()” and included in regular expressions</p> <p>(13) If dealing with locale-dependent data, specify an appropriate locale during comparison.</p> <p>(14) Use compatible character encoding on both sides of file or network input/output.</p> <p>(15) When working with web forms, never trust the content of hidden fields.</p> <p>(16) If a method accepts a collection, validate both the collection and elements within the collection are not “null”.</p>

2.5.3.8.9.13.5  
(07-10-2020)

#### **Injection and Inclusion**

- (1) Parse data that requires a certain input and perform only limited correction such as converting quotation marks to an acceptable pattern. In all other cases, reject the data if it does not meet the input requirements before parsing.
- (2) Use well-tested libraries instead of ad-hoc code. Use the standard library for creating XML or JSON files instead of raw text. Create classes that only handle formatting of unusual formats.
- (3) Avoid dynamic SQL statements. When accessing a database using JDBC, always use "java.sql.PreparedStatement" and "java.sql.CallableStatement".
- (4) Consider using a well-tested library to output HTML and XML to clients from untrusted data sources such as input from an HTML form.
- (5) Avoid entering or expecting untrusted data on the command line.
- (6) Restrict XML inclusion by preventing local or intranet files from being added to an XML file.
- (7) Take care when processing BMP files by restricting privileges to read included file references.
- (8) Disable HTML functionality in Swing components.
- (9) Take care interpreting untrusted code. Some examples include:
  - Interaction between browser JavaScript and native code
  - XSLT interpreter runs with extensions to call Java code
  - Long Term Persistence of JavaBeans components supports execution of Java statements
  - Playing sounds
  - Remote Method Invocation may allow loading of remote code specified by the remote connection.
  - LDAP allows loading of remote code in a server response.
  - SQL implementations allow execution of code with effects outside of the database
- (10) Prevent injection of exceptional floating-point values. Use the "Double.isNaN" and "Double.isInfinite" methods to check if a number is valid.

2.5.3.8.9.13.6  
(07-10-2020)

#### **Accessibility and Extensibility**

- (1) Isolate unrelated code by keeping code from different origins separated.
- (2) Limit the exposure of ClassLoader instances.
- (3) Purge sensitive information from exceptions. For example, if a method calls java.io.FileInputStream constructor to read an underlying configuration file and that file is not present, a java.io.FileNotFoundException containing the file path is thrown. for more information see, <https://www.oracle.com/technetwork/java/seccodeguide-139067.html>

2.5.3.8.9.13.7  
(07-10-2020)

#### **Serialization and Deserialization**

- (1) Avoid deserializing untrusted data.
- (2) Avoid serializing security sensitive classes.
- (3) Do not include sensitive data during serialization.
- (4) Follow the same guidance for object constructors when deserializing data.

- (5) Duplicate the security manager checks enforced during serialization and deserialization.
- (6) Consider applying security manager limitations if serializing and deserializing classes.

2.5.3.8.9.13.8  
(07-10-2020)  
**Access Control**

- (1) This section covers utilizing the security manager feature, pertaining to the list below.
  - Understand how permissions are checked.
  - Properly transfer context when using callback methods used in security-sensitive classes.
  - Understand how to safely invoke and restrict privileges in the method “doPrivileged”.
  - Do not cache the result of privileged operations.
  - Consider carefully the security ramifications of using reflection on untrusted objects.
  - Methods that perform a security check must be declared “private” or “final”.

2.5.3.8.9.13.9  
(07-10-2020)  
**Defensive Use of the  
Java Native Interface  
(JNI)**

- (1) Only use JNI when necessary.
- (2) Be aware of the C/C++ threat model.
- (3) Expect that JNI code can violate visibility and isolation rules.
- (4) Secure the JNI implementation from the Java side.
- (5) Properly test JNI code for concurrent access.
- (6) Secure library loading.
- (7) Perform input validation at the language boundary.
- (8) Expect and handle exceptions when calling JNI into Java.
- (9) Follow secure development practices for the native target platform.
- (10) Ensure any bundled JVMs and JREs meet Java’s secure baselines.

**This Page Intentionally Left Blank**

**Exhibit 2.5.3-1 (07-10-2020)**
**Java Programming- Example of Wrapping Lines**

Example of Wrapping Lines
<pre> / comma break void computeResult(String parameter1, String parameter2, String parameter3, String parameter4) throws InvalidOperationException, NumberFormatException, IllegalArgumentException { } // operator break int specialCharacteristic = longOperand1 + longOperand2 longOperand3 + longOperand4; // leveled break String accountName = accountPrefix + " " + getFullName(accountFirstName, accountLastName) + " " + accountSuffix; int numericResult = longName1 + longName2 * (longName3 – longName4 + longName5) / longName6; // indent four space rule void aVeryLongMethodNameThatIsHardToCompress(String parameter1, String parameter2) throws IOException { } </pre>

**Exhibit 2.5.3-2 (07-10-2020)****Java Programming Example of Objects****Comparison of public class ObjectTest and class Point**

```
public class ObjectTest {
public static void main(String[] args) {
Point p1 = new Point(4, 6);
Point p2 = new Point(7, 9);
Point p3 = p2; System.out.println("Before change:");
System.out.printf("Point 1 = (%d,%d)", p1.getX(), p1.getY());
System.out.printf("Point 2 = (%d,%d)", p2.getX(), p2.getY());
System.out.printf("Point 3 = (%d,%d)", p3.getX(), p3.getY());
p3.setX(10);
p3.setY(20);
System.out.println("After change:");
System.out.printf("Point 1 = (%d,%d)", p1.getX(), p1.getY());
System.out.printf("Point 2 = (%d,%d)", p2.getX(), p2.getY());
System.out.printf("Point 3 = (%d,%d)", p3.getX(), p3.getY());
}
}
class Point {
private int x;
private int y;
public Point(int x, int y) {
this.x = x;
this.y = y;
}
public int getX() { return x; }
public void setX(int x) { this.x = x; }
public int getY() { return y; }
public void setY(int y) { this.y = y; }
}
```



**Exhibit 2.5.3-3 (07-10-2020)**
**Java Programming - Object Test Results**

Before change:	After change:
Point 1 = (4,6) Point 2 = (7,9) Point 3 = (7,9)	Point 1 = (4,6) Point 2 = (10, 20) Point 3 = (10, 20)

**Exhibit 2.5.3-4 (07-10-2020)****Java Programming Example of “instanceof”****Wrapping Explicit Cast using “Instanceof” Check**

```
public class RecastTest {
public static void main(String[] args) {
Dog collie = new Dog();
collie.setName("Collie");
System.out.println(collie.getName());
System.out.println("Leg count: " + collie.getLegs());
Animal animal = (Animal)collie;
System.out.println(animal.getName());
// this generates a compiler error
// System.out.println("Leg count: " + animal.getLegs());
if (animal instanceof FourLeggedAnimal) {
System.out.println("Leg count: " + ((FourLeggedAnimal)animal).getLegs());
}
}
}
```

**Exhibit 2.5.3-5 (07-10-2020)****Java Programming Example -Subclass****Java Programming Subclass Example**

```
class Animal {private String name;
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
class FourLeggedAnimal extends Animal {
    public int getLegs() { return 4; }
}

class Dog extends FourLeggedAnimal {
    @Override
    public String getName() { return "Shaggy " + super.getName();
}
}
```

**Exhibit 2.5.3-6 (07-10-2020)****Java Programming Example - Enumeration Type****Enumeration Type Example**

```

enum Month {
    JANUARY(31), FEBRUARY(28), MARCH(31), APRIL(30), MAY(31), JUNE(30), JULY(31),
    AUGUST(31), SEPTEMBER(30), OCTOBER(31), NOVEMBER(30), DECEMBER(31);
    private int days = 0;
    private Month(int days) {
        this.days = days;
    }
    public int getDays(int year) {
        if (days == 28) {
            if (year % 4 == 0 && year % 100 > 0) {
                return 29;
            }
            else if (year % 100 == 0 && year % 400 > 0) {
                return 28;
            }
            else if (year % 400 == 0) {
                return 29;
            }
            else {
                return 28;
            }
        }
        else {
            return days;
        }
    }
}

public class EnumTest {
    static public void main(String[] args) {
        System.out.printf("Days in January 2020: %d", Month.JANUARY.getDays(2020));
        System.out.printf("Days in April 2020: %d", Month.APRIL.getDays(2020));
        System.out.printf("Days in February 1900: %d", Month.FEBRUARY.getDays(1900));
        System.out.printf("Days in February 2000: %d", Month.FEBRUARY.getDays(2000));
        System.out.printf("Days in February 2019: %d", Month.FEBRUARY.getDays(2019));
        System.out.printf("Days in February 2020: %d", Month.FEBRUARY.getDays(2020));
    }
} // Output
Days in January 2020: 31
Days in April 2020: 30 Days in February 1900: 28
Days in February 2000: 29 Days in February 2019: 28
Days in February 2020: 29

```

**Exhibit 2.5.3-7 (07-10-2020)**
**Java Programming Example - Nested Classes**

Java Programming Example of Nested Classes
<pre>import java.util.Optional; interface Greeting {     public String getMessage(String name); }</pre>
<pre>interface Greeting {     public String getMessage(String name); }</pre>
<pre>public class AnonymousClassExample {     static public void main(String[] args) {         Greeting englishGreeting = new Greeting() {             public String getMessage(String name) {                 Optional nameOptional = Optional.ofNullable(name);                 return "Hello " + nameOptional.orElse("Anonymous");             }         };         System.out.println(englishGreeting.getMessage("Bob"));         System.out.println(englishGreeting.getMessage(null));     } }</pre>
<pre>// Output Hello Bob Hello Anonymous</pre>

**Exhibit 2.5.3-8 (07-10-2020)****Java Programming Class Switch Example**

Java Programming Class Switch
<pre>enum Quarter { Q1, Q2, Q3, Q4; static public Quarter getQuarter(int index) { if (index == 1) { return Q1; } else if (index == 2) { return Q2; } else if (index == 3) { return Q3; } else { return Q4; } } }</pre>
<pre>public class SwitchExample { static public void main(String[] args) { Quarter q1 = Quarter.valueOf("Q1"); Quarter q4 = Quarter.getQuarter(4); displayRequirements(q1); displayRequirements(q4); }</pre>
<pre>static private void displayRequirements(Quarter currentQuarter) { boolean taxesDue = false; boolean runAudit = true; switch (currentQuarter) { case Q2: runAudit = false; break; case Q1: runAudit = false; /* falls through */ case Q3: taxesDue = true; break; case Q4: break; } System.out.printf("%s: taxesDue=%s, runAudit=%s", currentQuarter.toString(), taxesDue, runAudit); } }</pre>

**Exhibit 2.5.3-8 (Cont. 1) (07-10-2020)**
**Java Programming Class Switch Example**

Java Programming Class Switch
<pre>// Output Q1:  taxesDue=true, runAudit=false Q4: taxesDue=false, runAudit=true</pre>

**Exhibit 2.5.3-9 (07-10-2020)****Java Programming Design Example**

Design Example
<pre> import java.time.LocalDateTime ; public class DesignExample { private static LocalDateTime firstDate; private LocalDateTime instanceDate; // This is the type constructor static { firstDate = LocalDateTime.now(); } static public LocalDateTime getFirstDate() { return firstDate; } static public void main(String[] args) { DesignExample ex1 = new DesignExample(); System.out.printf("Example 1: First=[%s],Instance[%s]", DesignExample.getFirstDate(), ex1.getInstanceDate()); // Put in an execution pause so a date difference can be detected. try { Thread.sleep(5000); } catch (InterruptedException e) { // TODO Auto-generated catch block e.printStackTrace(); } DesignExample ex2 = new DesignExample(); System.out.printf("Example 2: First=[%s],Instance[%s]", DesignExample.getFirstDate(), ex2.getInstanceDate()); } // This is the instance constructor. public DesignExample() { instanceDate = LocalDateTime.now(); } public LocalDateTime getInstanceDate() { return instanceDate; } } </pre>
<p><b>// Output</b></p> <p>Example 1: First=[2019-05-30T15:07:08.520],Instance[2019-05-30T15:07:08.566]</p> <p>Example 2: First=[2019-05-30T15:07:08.520],Instance[2019-05-30T15:07:13.581]</p>



**Exhibit 2.5.3-10 (07-10-2020)****Java Programming Constructor Example**

Java Programming Constructor
<pre> import java.time.LocalDateTime; public class DesignExample { private static LocalDateTime firstDate; private LocalDateTime instanceDate; // This is the type constructor static { firstDate = LocalDateTime.now(); } static public LocalDateTime getFirstDate() { return firstDate; } static public void main(String[] args) { DesignExample ex1 = new DesignExample(); System.out.printf("Example 1: First=[%s],Instance[%s]", DesignExample.getFirstDate(), ex1.getInstanceDate()); // Put in an execution pause so a date difference can be detected . try { Thread.sleep(5000); } catch (InterruptedException e) { // TODO Auto-generated catch block e.printStackTrace(); } DesignExample ex2 = new DesignExample(); System.out.printf("Example 2: First=[%s],Instance[%s]", DesignExample.getFirstDate(), ex2.getInstanceDate()); } // This is the instance constructor. public DesignExample() { instanceDate = LocalDateTime.now(); } public LocalDateTime getInstanceDate() { return instanceDate; } } </pre>
<pre> // Output Example 1: First=[2019-05-30T15:07:08.520],Instance[2019-05-30T15:07:08.566]  Example 2: First=[2019-05-30T15:07:08.520],Instance[2019-05-30T15:07:13.581] </pre>

**Exhibit 2.5.3-11 (07-10-2020)****Java Programming Abstract Properties**

Abstract Properties
<pre>abstract public class Shape { abstract public double getArea(); } class Circle extends Shape { private int radius = 0; public Circle() { } public Circle(int radius) { this.radius = radius; } public void setRadius(int radius) { this.radius = radius; } public int getRadius() { return radius; } @Override public double getArea() { return Math.pow(radius, 2) * Math.PI; } }  class Rectangle extends Shape { private int width = 0; private int height = 0; public Rectangle() { } public Rectangle(int width, int height) { this.width = width; this.height = height; } @Override public double getArea() { return width * height; } }  class Square extends Rectangle { public Square() { } public Square(int size) { super(size, size); } }</pre>

**Exhibit 2.5.3-12 (07-10-2020)**
**Java Programming Example of Event Design**

Event Design
<pre> import java.util.ArrayList; import java.util.EventListener; import java.util.EventObject; import java.util.List; class PropertyEvent extends EventObject {     private static final long serialVersionUID = 6111548293142751985;     private String name = "";     public PropertyEvent(Object source, String name) {         super(source);         this.name = name;     }     public String getName() { return name;     } } interface PropertyListener extends EventListener {     void propertyChanged(PropertyEvent e); } public class ListenerExample {     private List&lt;PropertyListener&gt; listeners = new ArrayList&lt;&gt;();     private String prop = "";     public void addListener(PropertyListener listener) {         listeners.add(listener);     }     public void removeListener(PropertyListener listener) {         listeners.remove(listener);     }     public String getProperty() { return prop;     }     public void setProperty(String value) {         prop = value;         raisePropertyChangeEvent(new PropertyEvent(this, "Property"));     }     protected void raisePropertyChangeEvent(PropertyEvent e) {         for (PropertyListener listener : listeners) {             listener.propertyChanged(e);         }     } } </pre>

**Exhibit 2.5.3-13 (07-10-2020)****Java Programming Thread example**

Thread Example
<pre> class NirvanaRunnable implements Runnable { private ThreadExample example = null; public NirvanaRunnable(ThreadExample example) { this.example = example; } @Override public void run() { example.guardedJoy(); } } class EatingRunnable implements Runnable { private ThreadExample example = null; public EatingRunnable(ThreadExample example) { this.example = example; } @Override public void run() { example.notifyJoy(); } } public class ThreadExample { private boolean joy = false; static public void main(String[] args) { ThreadExample example = new ThreadExample(); Thread nirvanaThread = new Thread(new NirvanaRunnable(example)); nirvanaThread.start(); System.out.println("Started nirvana thread..."); Thread eatingThread = new Thread(new EatingRunnable(example)); eatingThread.start(); System.out.println("Started eating thread..."); } public synchronized void guardedJoy() { // This guard only loops once for each special event, which may not // be the event we're waiting for. while(!joy) { try { wait(); } catch (InterruptedException e) {} } System.out.println("Joy and efficiency have been achieved!"); } public synchronized void notifyJoy() { joy = true; System.out.println("Reached joy!"); notifyAll(); } } </pre>
<pre> // Output Started nirvana thread... Started eating thread... Reached joy! Joy and efficiency have been achieved! </pre>

**Exhibit 2.5.3-14 (07-10-2020)**
**Java Programming Examples of Single Words used for Capitalization Purposes**

<b>Java Programming Single Words used for Capitalization Purposes</b>		
<b>Pascal</b>	<b>Camel</b>	<b>Not</b>
BitFlag	bitFlag	Bitflag
Callback	callback	CallBack
Canceled	canceled	Cancelled
DoNot	doNot	Don't
Email	email	EMail
Endpoint	endpoint	Endpoint
FileName	fileName	Filename
Gridline	gridline	GridLine
Hashtable	hashtable	HashTable
Id	id	ID
Indexes	indexes	Indices
LogOff	logOff	LogOut
LogOn	logOn	LogIn
Metadata	metaData	MetaData,
Multipanel	multipanel	MultiPanel
Multiview	multiview	MultiView
Namespace	namespace	NameSpace
Ok	ok	OK
Pi	pi	PI
Placeholder	placeholder	PlaceHolder
SignIn	signIn	SignOn
SignOut	signOut	SignOff
UserName	userName	Username
WhiteSpace	whiteSpace	Whitespace
Writable	writable	Writable

**Exhibit 2.5.3-15 (07-10-2020)****C Programming Source Code Template**

<b>C Programming Source Code Template</b>
/******
//
// Internal Revenue Service
// For Official Use Only
//
// Filename: Filename
// Description: Describe the purpose of the objects in the file,
// followed, in the case of source files, by a list
// of functions whose definitions appear in the file
// Related Files: An identification of any routines or files that
// this file may require
// Restrictions/ Known special cases where the file may not work
// Problems:
//
// Date Modified: YYYY/MM/DD
// Version id: Revision:
// Author: <First Name> <Last Name>
// Locked by: \$Locker\$
//
// Revision History: Will be provided by ClearCase
*****/

**Example of C Programming Header Files number 1**

<b>C Programming Header Files number 1</b>
/*h*****
* H E A D E R F I L E S *
*****/
/* System header files */
#include <stdio.h>

**Exhibit 2.5.3-15 (Cont. 1) (07-10-2020)**  
**C Programming Source Code Template**

C Programming Header Files number 1	
/*h*****	
#include <stdlib.h>	
#include <string.h>	
#include <limits.h>	
#include <unistd.h>	
#include <signal.h>	
#include <errno.h>	
/* User include files */	
#include "archive.h"	
#include "acdbApi.h"	
#include "scdbApi.h"	

**C Programming Example of Defines number 1**

C Programming - Defines number 1		
*	DEFINES	*
/*d*****		
*****/		
/* Debug flags */		
#ifdef FOR_MAC		
	#define SIGALRM 14	
#end-if		
/* Constants */		
#define SUCCESS 0		
#ifndef TRUE		
	#define TRUE 1	
	#define FALSE 0	
#end-if		
/* Macros */		
#define MIN(a,b) ((a)<(b)) ? (a) : (b)		

**Exhibit 2.5.3-15 (Cont. 2) (07-10-2020)**  
**C Programming Source Code Template**

***C Programming example of TypeDef Number 1***

C Programming - TypeDef Number 1		
/*t*****		
*	T Y P E D E F	*
*****/		
typedef struct ECT_REG_HEADER_S		
{		
char *pFirstEntry; /* ptr to 1st registry entry */		
int NumEntries; /* number of entries */		
} ECT_REG_HEADER_T;		

***C Programming example of Enums number 1***

* E N U M S *		
/*e*****		
*****/		
enum TAX_FORMS_E		
{		
F_1040 = 0,		
F_1065,		
F_941_ELF,		
F_941_OLF };		

***C Programming Example of Definitions number 1***

* DEFINITIONS*		
/*g*****		
*****/		
/* External data */		
extern char *pStr; /* comments */		
extern int GlobalExt; /* comments */		
/* Non-static data */		



## Exhibit 2.5.3-15 (Cont. 3) (07-10-2020)

### C Programming Source Code Template

#### \* DEFINITIONS\*

```
int DataGl; /* comments */
char *pStr; /* comments */
/* Static data */
static int DataGl; /* comments */
static char *pStr; /* comments */
```

#### C Programming Function Examples number 1

#### \* FUNCTION PROTOTYPES (alphabetized) \*

```
/*fp*****
******/

int FunctionName1(int par1, char *par2_p);
Void FunctionName2(int par1, char *par2_p);
*
/*f*****
* Function Name: FunctionName1
* Description: A description of the major task(s) performed by
* routine. It should be a series of one or more
* simple verb/object statements
* Input parameters : par1 - description
*      par2_P - description
* Output parameters: *par2_p - description
*      Function return - SUCCESS or FAIL
*****/

int FunctionName1(int par1, char *par2_p)
{
    /* LOCAL VARIABLES and CONSTANTS*/
    /* FUNCTION BODY */
    return return_code;
}
```

**Exhibit 2.5.3-15 (Cont. 4) (07-10-2020)**  
**C Programming Source Code Template*****Example of C Function with Multiple Routines*****\* Function Name: FunctionName2**

```
/******  
* Description: A description of the major task(s) performed by  
* routine. It should be a series of one or more  
* simple verb/object statements  
* Input parameters : par1 - description  
* par2_P - description  
* Output parameters: *par2_p - description  
*****/  
void FunctionName2(int par1, char *par2_p)  
{  
/* LOCAL VARIABLES and CONSTANTS*/  
/* FUNCTION BODY */  
}
```

## Exhibit 2.5.3-16 (07-10-2020) C Language Header File Template

```

C Language Header File

// Description: Describe the purpose of the objects in the file,
/*****
//
//2
// Internal Revenue Service
// For Official Use Only
//
// Filename: Filename
//
// followed, in the case of source files, by a list
// of functions whose definitions appear in the file
// Related Files: An identification of any routines or files that
// this file may require
// Restrictions/ Known special cases where the file may not work
// Problems:
//
// Date Modified: Date: YYYY/MM/DD
// Version id: Revision:
// Author: Author: <First Name> <Last Name>
// Locked by: $Locker:$
//
// Revision History: Will be provided by ClearCase
*****/

```

### *C Language Defines Template number 2*

```

* D E F I N E S Template
*

#ifndef TEMPLATE
#define TEMPLATE
/*****
*****/
/* Debug flags */

```

**Exhibit 2.5.3-16 (Cont. 1) (07-10-2020)**  
**C Language Header File Template**

**\* D E F I N E S Template**

```

#ifndef FOR_MAC
#define SIGALRM 14
#endif-IF
/* Constants /
#define SUCCESS
0
#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif-IF
/* Macros */
#define MIN(a,b) ((a)<(b)) ? (a) : (b)

```

***C Programing Example of TypeDefs number 2***

**\* T Y P E D E F S Template**

```

/*****
*****/

typedef struct ECT_REG_HEADER_S
{
char *pFirstEntry; /* ptr to 1st registry entry */
int NumEntries; /* number of entries */
} ECT_REG_HEADER_T;

```

***C Programming Example of Enums number 2***

**\* E N U M S**

```

/*****
*****/

enum TAX_FORMS_E
{
    F_1040 = 0,

```

**Exhibit 2.5.3-16 (Cont. 2) (07-10-2020)**  
**C Language Header File Template**

```

*                                     E N U M S                                     *

    F_1065,
    F_941_ELF,
    F_941_OLF
};

```

***C Programming Example of Functions***

```

*      F U N C T I O N P R O T O T Y P E S (alphabetized) *
*****/

    #END-IF /* TEMPLATE */

    void FunctionName2(int par1, char *par2_p);
int FunctionName1(int par1, char *par2_p);
/*****

```

**Exhibit 2.5.3-17 (07-10-2020)**  
**Acronyms and Terms**

***Acronyms and Terms***

<b>Acronym</b>	<b>Terms</b>
ACIO	Assistant Chief Information Officer
ALC	Assembler Language Code
ASCII	American Standard Code for Information Interchange
CICS	Customer Information Control System
COBOL	Common Business-Oriented Language
ECL	Executive Control Language
EOF	End of File
GAO	Government Accountability Office
IBM	International Business Machines Corporation
ISBN	International Standard Book Number
HLASM	High-Level Assembler
HTML	Hypertext Markup Language
IC	Internal Controls
IT	Information Technology
ISO	International Organization for Standardization
JCL	Job Control Language
JNI	Java Native Interface
JRE	Java Runtime Environment
JSON	Java Script Object Notation
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
MASM	Meta-Assembler
OMB	Office of Management and Budget
QA	Quality Assurance
OS	Operating System
OWASP	Open Web Application Security Project
PNG	Portable Network Graphics
UNIYSIS	UNIVAC Systems Corporation
XML	Extensible Markup Language

**Exhibit 2.5.3-18 (07-10-2020)**  
**Terms and Definitions**

Terms	Definitions
Case Structure	A control structure used when there are numerous paths to be followed depending on the contents of a given field or variable.
Conditional Statement	An instruction that use the word “ <b>IF</b> ” to test for the existence of a condition.
Framework	A set of functions within a system, and how they interrelate
Lightweight Directory Access Protocol	Open vendor industry standard application protocol for accessing and maintaining distributed directory information services over an Internet Protocol (IP) network. Allows sharing of information about users.
High-Level Assembler	IBM’s assembler programming language and the assembler itself for the IBM z/OS, z/VM, OS/390, MVS, VM and VSE operating systems. Released June 1992
Hypertext Markup Language	An application of the Standard Generalized Markup Language, which is the international standard for Markup, and is the primary markup language used to write content on the web.
Internal Controls	Management controls that provide reasonable assurance that obligations and cost are in compliance with applicable laws, funds, property; and other assets are safeguarded against waste, loss, unauthorized use or misappropriation.
International Organization for Standardization	The International Organization for Standardization is an independent, non-governmental organization, the members of which are the standards organizations of the 164[1] member countries. It is the world’s largest developer of voluntary international standards and facilitates world trade by providing common standards between nations. Over twenty thousand standards have been set covering everything from manufactured products and technology to food safety, agriculture and healthcare
Java Bean	An object-oriented programming interface that allows you to build re-usable applications or program building blocks called components. Java Bean can be deployed in a network on any major operating system platform
Java Native Interface	Programming framework that enables Java code running in a Java Virtual Machine to call and be called by a native application (programs specific to a hardware and operating system platform and libraries written in other languages e.g., C, C++ and Assembler.

**Exhibit 2.5.3-18 (Cont. 1) (07-10-2020)**  
**Terms and Definitions**

Terms	Definitions
Java Runtime Environment (JRE)	Also known as Java Runtime is part of the Java Development kit that contains: Java Virtual Machine(JVM), Java Platform core classes, and supporting Java platform libraries.
Java Script Object Notation	A text-based human readable interchanged format used for representing simple data structures and objects in Web browser based code.
Java Virtual Machine(JVM)	JVM has two functions: to allow Java programs to run on any device or OS, (known as the “Write Once, run Anywhere” principle), and to Manage and optimize program memory..
OWASP	The Open Web Application Security Project is a nonprofit organization focused on improving the security of software. OWASP provides impartial information about AppSec to individuals, corporations, universities, government agencies, and other organizations worldwide.
Portable Network Graphics	A raster graphics file format that supports loss-less data compression, PNG was created as an improved replacement for Graphics Interchanged Format (GIF)
Program	A set of instructions that operate on input data and convert it to output.
Refactor	Altering an application's source code without changing its external behavior. The purpose of code refactoring is to improve some of the nonfunctional properties of the code, e.g. readability, complexity, maintainability, and extensibility
Extensible Stylesheet Language Transformation (XSL)	A language for transforming XML document into other XML documents or other formats such as HTML for web pages, plain text or XSL Formatting Objects which may be converted to other formats, such as PDF, PostScript and PNG supported in modern web browsers.
z/OS	A 64-bit operating system for IBM mainframes, produced by IBM, and derived by successor OS/390.



**Exhibit 2.5.3-19 (07-10-2020)****Language Code (ALC) Standards and References**

- IBM Systems Standard Manual Version 5
- BCPA 40 ALC Student Guide
- IRS Messages and Codes Edition 2
- High Level Assembler for z/OS & z/VM & z/VSE Language Reference Version 1R6
- High Level Assembler for z/OS & z/VM & z/VSE Programmer's Guide Version 1R6
- UNISYS ClearPath OS2200 Meta-Assembler (MASM) Programming Reference Manual Level 6R3J
- UNISYS ClearPath OS2200 Executive Control Language (ECL) and FURPUR Reference Manual
- Assembler H Version 2 Application Programming Guide , SC26-4036
- Assembler H Version 2 Application Programming Language Reference, SC26-4037
- MVS JCL Reference, GC28-1352
- MVS/XA Linkage Editor & Loader User's Guide, GC26-4143
- MVS/XA Message Library: System Messages Vol 1, GC28-1376
- MVS/XA Message Library: System Messages Vol 2, GC28-1377
- MVS/XA Message Library: System Codes, GC28-1157
- MVS/XA Data Administration, GC26-4149
- MVS/XA Data Administration: Utilities, GC26-4150
- MVS/XA Data Administration: Macro Instruction Reference, GC26-4141
- MVS/XA Utilities Messages , GC26-4021
- MVS/XA TSO Terminal User's Guide, GC28-1274
- TSO/E TSO Command Language Reference, GC28-0646
- ISPF/PDF Program Reference, SC34-2139

