This document outlines industry-standard coding practices for languages supported by SonarQube, focusing on standards adopted by leading technology companies and enterprise organizations. These standards ensure code quality, maintainability, security, and consistency across development teams.

**1. Java Coding Standards**

**Oracle/Sun Java Code Conventions & Google Java Style Guide**

**Naming Conventions:**

* Classes: PascalCase (e.g., CustomerService)
* Methods and variables: camelCase (e.g., calculateTotalAmount)
* Constants: UPPER\_SNAKE\_CASE (e.g., MAX\_RETRY\_COUNT)
* Packages: lowercase with dots (e.g., com.company.module)

**Code Structure:**

* Maximum line length: 120 characters
* Indentation: 4 spaces (no tabs)
* Braces: Egyptian style (opening brace on same line)
* One class per file, file name matches class name

**Best Practices:**

* Use meaningful variable and method names
* Avoid deep nesting (maximum 3-4 levels)
* Implement proper exception handling with specific exception types
* Use StringBuilder for string concatenation in loops
* Follow SOLID principles and design patterns
* Maintain cyclomatic complexity below 10 per method

**Security Standards:**

* Input validation and sanitization
* Proper resource management (try-with-resources)
* Avoid hardcoded credentials
* Use parameterized queries for database operations

**2. JavaScript/TypeScript Standards**

**Airbnb JavaScript Style Guide & Google TypeScript Style Guide**

**Naming Conventions:**

* Variables and functions: camelCase (e.g., getUserData)
* Constants: UPPER\_SNAKE\_CASE (e.g., API\_ENDPOINT)
* Classes: PascalCase (e.g., DataProcessor)
* Files: kebab-case (e.g., user-service.js)

**Code Structure:**

* Maximum line length: 100 characters
* Indentation: 2 spaces
* Semicolons: Always use
* Quotes: Single quotes for strings, double quotes for JSX attributes

**ES6+ Best Practices:**

* Use const and let instead of var
* Prefer arrow functions for short functions
* Use template literals for string interpolation
* Implement proper error handling with try-catch blocks
* Use async/await instead of Promises where possible

**TypeScript Specific:**

* Explicit type annotations for function parameters and return types
* Use interfaces for object shapes
* Enable strict mode in tsconfig.json
* Avoid any type, use unknown when necessary

**3. Python Standards (PEP 8 & Beyond)**

**PEP 8, PEP 257, and Google Python Style Guide**

**Naming Conventions:**

* Variables and functions: snake\_case (e.g., user\_data)
* Classes: PascalCase (e.g., DataProcessor)
* Constants: UPPER\_SNAKE\_CASE (e.g., MAX\_CONNECTIONS)
* Modules: lowercase with underscores (e.g., data\_utils)

**Code Structure:**

* Maximum line length: 88 characters (Black formatter standard)
* Indentation: 4 spaces
* Two blank lines before class definitions
* One blank line before method definitions

**Best Practices:**

* Use type hints for function parameters and return types
* Follow docstring conventions (PEP 257)
* Use list comprehensions and generator expressions appropriately
* Implement proper exception handling with specific exception types
* Use context managers for resource management
* Follow the Zen of Python principles

**Security and Performance:**

* Use secrets module for cryptographically secure random numbers
* Validate and sanitize all inputs
* Use pathlib for file system operations
* Avoid mutable default arguments

**4. C# Standards**

**Microsoft .NET Coding Conventions & C# Coding Standards**

**Naming Conventions:**

* Classes, methods, properties: PascalCase (e.g., CustomerService)
* Local variables, parameters: camelCase (e.g., customerData)
* Private fields: camelCase with underscore prefix (e.g., \_repository)
* Constants: PascalCase (e.g., MaxRetryCount)

**Code Structure:**

* Maximum line length: 120 characters
* Indentation: 4 spaces
* Braces: Allman style (opening brace on new line)
* One class per file

**Best Practices:**

* Use nullable reference types (C# 8+)
* Implement proper disposal pattern with using statements
* Use LINQ appropriately without overcomplicating
* Follow async/await patterns correctly
* Implement proper exception handling with specific exception types
* Use dependency injection and inversion of control

**Security Standards:**

* Input validation and sanitization
* Use SecureString for sensitive data
* Implement proper authentication and authorization
* Follow OWASP security guidelines

**5. C/C++ Standards**

**MISRA C, Google C++ Style Guide, and ISO C++ Core Guidelines**

**Naming Conventions:**

* Variables and functions: snake\_case or camelCase (consistent within project)
* Classes: PascalCase (e.g., DataProcessor)
* Constants: UPPER\_SNAKE\_CASE (e.g., MAX\_BUFFER\_SIZE)
* Macros: UPPER\_SNAKE\_CASE with descriptive names

**Code Structure:**

* Maximum line length: 80-120 characters
* Indentation: 2-4 spaces (consistent within project)
* Braces: K&R or Allman style (consistent within project)

**Best Practices:**

* Use RAII (Resource Acquisition Is Initialization)
* Prefer smart pointers over raw pointers
* Use const correctness throughout
* Avoid C-style casts, use C++ style casts
* Implement proper exception handling
* Use standard library containers instead of C arrays
* Follow rule of zero/three/five

**Safety and Security:**

* Avoid buffer overflows with bounds checking
* Use static analysis tools
* Initialize all variables
* Check return values of system calls
* Use secure coding practices for memory management

**6. PHP Standards**

**PSR Standards (PSR-1, PSR-2, PSR-4, PSR-12)**

**Naming Conventions:**

* Classes: PascalCase (e.g., UserController)
* Methods and variables: camelCase (e.g., getUserData)
* Constants: UPPER\_SNAKE\_CASE (e.g., MAX\_FILE\_SIZE)
* Namespaces: PascalCase with backslashes

**Code Structure:**

* Maximum line length: 120 characters
* Indentation: 4 spaces
* PHP tags: Always use <?php, never short tags
* File encoding: UTF-8 without BOM

**Best Practices:**

* Use type declarations for function parameters and return types
* Implement proper error handling with exceptions
* Use prepared statements for database queries
* Follow MVC or similar architectural patterns
* Use dependency injection and autoloading (Composer)
* Implement proper validation and sanitization

**7. Go Standards**

**Effective Go and Go Code Review Comments**

**Naming Conventions:**

* Packages: lowercase, single word when possible
* Variables and functions: camelCase (e.g., userName)
* Exported identifiers: PascalCase (e.g., UserService)
* Constants: camelCase or PascalCase based on visibility

**Code Structure:**

* Use gofmt for consistent formatting
* Maximum line length: 120 characters (soft limit)
* Braces: K&R style
* Import grouping: standard library, third party, local

**Best Practices:**

* Use meaningful variable names, even for short-lived variables
* Handle errors explicitly, don't ignore them
* Use goroutines and channels appropriately
* Implement proper context handling for cancellation
* Use interfaces to define behavior
* Write clear, concise comments for exported functions

**8. Ruby Standards**

**Ruby Style Guide and RuboCop Rules**

**Naming Conventions:**

* Variables and methods: snake\_case (e.g., user\_name)
* Classes and modules: PascalCase (e.g., UserService)
* Constants: UPPER\_SNAKE\_CASE (e.g., MAX\_RETRY\_COUNT)
* Files: snake\_case matching class name

**Code Structure:**

* Maximum line length: 120 characters
* Indentation: 2 spaces
* String literals: Single quotes when no interpolation needed
* Use meaningful method and variable names

**Best Practices:**

* Use blocks and iterators appropriately
* Prefer && and || over and and or
* Use symbols for hash keys when possible
* Implement proper exception handling
* Follow Ruby idioms and conventions
* Use RuboCop for style enforcement

**9. Kotlin Standards**

**Kotlin Coding Conventions and Android Kotlin Style Guide**

**Naming Conventions:**

* Classes and objects: PascalCase (e.g., UserRepository)
* Functions and variables: camelCase (e.g., getUserData)
* Constants: UPPER\_SNAKE\_CASE (e.g., MAX\_RETRY\_COUNT)
* Package names: lowercase with dots

**Code Structure:**

* Maximum line length: 120 characters
* Indentation: 4 spaces
* Use trailing commas in multi-line declarations
* Prefer expression bodies for simple functions

**Best Practices:**

* Use nullable types appropriately with safe calls
* Prefer val over var when possible
* Use data classes for simple data holders
* Implement proper coroutine usage for asynchronous operations
* Use extension functions judiciously
* Follow functional programming principles where appropriate

**10. Swift Standards**

**Swift Style Guide and Apple's API Design Guidelines**

**Naming Conventions:**

* Types: PascalCase (e.g., UserService)
* Variables and functions: camelCase (e.g., userName)
* Constants: camelCase (e.g., maxRetryCount)
* Protocols: adjectives ending in -able, -ible, or -ing when possible

**Code Structure:**

* Maximum line length: 120 characters
* Indentation: 4 spaces for code, 2 spaces for continuation
* Use trailing closures when appropriate
* Prefer explicit types when clarity is improved

**Best Practices:**

* Use optionals appropriately with safe unwrapping
* Prefer value types (structs) over reference types (classes)
* Use guard statements for early returns
* Implement proper error handling with do-catch blocks
* Use protocol-oriented programming
* Follow memory management best practices with ARC

**11. Scala Standards**

**Scala Style Guide and Effective Scala**

**Naming Conventions:**

* Classes and traits: PascalCase (e.g., UserService)
* Methods and variables: camelCase (e.g., getUserData)
* Constants: PascalCase (e.g., MaxRetryCount)
* Package names: lowercase with dots

**Code Structure:**

* Maximum line length: 120 characters
* Indentation: 2 spaces
* Use meaningful type annotations for public APIs
* Prefer immutable data structures

**Best Practices:**

* Use functional programming principles appropriately
* Prefer val over var when possible
* Use pattern matching effectively
* Implement proper error handling with Try, Either, or Option
* Use for-comprehensions for sequential operations
* Follow functional programming and object-oriented design principles

**12. Infrastructure as Code (IaC) Standards**

**Terraform, CloudFormation, and Docker Best Practices**

**Terraform Standards:**

* Use lowercase with underscores for resource names
* Organize code into modules
* Use variables and outputs appropriately
* Implement proper state management
* Use consistent formatting with terraform fmt

**Docker Standards:**

* Use official base images when possible
* Minimize number of layers
* Use specific version tags, avoid latest
* Implement multi-stage builds for optimization
* Follow security best practices (non-root user, minimal attack surface)

**General IaC Principles:**

* Version control all infrastructure code
* Use parameterization for reusability
* Implement proper secret management
* Use linting tools for validation
* Document infrastructure components and dependencies

**13. Cross-Language Security Standards**

**OWASP Guidelines and Security Best Practices**

**Input Validation:**

* Validate all inputs at application boundaries
* Use whitelist validation when possible
* Implement proper encoding for outputs
* Sanitize data based on context (HTML, SQL, etc.)

**Authentication and Authorization:**

* Implement multi-factor authentication
* Use strong password policies
* Implement proper session management
* Follow principle of least privilege

**Data Protection:**

* Encrypt sensitive data at rest and in transit
* Use secure communication protocols (HTTPS, TLS)
* Implement proper key management
* Follow data retention policies

**14. Code Quality Metrics and Thresholds**

**Industry Standard Thresholds**

**Complexity Metrics:**

* Cyclomatic Complexity: ≤ 10 per method
* Cognitive Complexity: ≤ 15 per method
* Nesting Depth: ≤ 4 levels
* Method Length: ≤ 50 lines

**Coverage Metrics:**

* Unit Test Coverage: ≥ 80%
* Branch Coverage: ≥ 70%
* Critical Path Coverage: 100%

**Maintainability Metrics:**

* Technical Debt Ratio: ≤ 5%
* Code Duplication: ≤ 3%
* File Size: ≤ 500 lines (language dependent)

**Security Standards:**

* Zero critical security vulnerabilities
* All high-severity vulnerabilities addressed within 48 hours
* Regular security audits and penetration testing

**15. Implementation Guidelines**

**Recommended Tools and Processes**

**Static Analysis Tools:**

* SonarQube for comprehensive code analysis
* Language-specific linters (ESLint, Pylint, RuboCop, etc.)
* Security scanners (Snyk, Checkmarx)
* Dependency vulnerability scanners

**Code Formatting:**

* Use automated formatters (Prettier, Black, gofmt)
* Integrate formatters into CI/CD pipelines
* Configure IDE/editor integration
* Enforce formatting in pre-commit hooks

**Code Review Process:**

* Mandatory peer reviews for all code changes
* Use standardized review checklists
* Automated checks before human review
* Documentation of review decisions

**Continuous Integration:**

* Automated testing at multiple levels
* Quality gates with defined thresholds
* Fail-fast approach for quality violations
* Regular dependency updates and security patches

**16. Compliance and Regulatory Standards**

**Industry-Specific Requirements**

**Financial Services (PCI DSS, SOX):**

* Enhanced security controls
* Audit trails for all code changes
* Segregation of duties in development process
* Regular compliance assessments

**Healthcare (HIPAA):**

* Data privacy and protection measures
* Access controls and logging
* Encryption requirements
* Business associate agreements for third-party tools

**Government (FedRAMP, FISMA):**

* Security controls framework
* Continuous monitoring requirements
* Incident response procedures
* Supply chain risk management