- Consider using classes instead of base types for values with physical properties, such as weight or size.
- Ensure that the unsigned and signed right shift operators are not confused with each other.
- Avoid manipulating numbers using unsigned arithmetic operations in class Integer.
- Use java.nio.ByteBuffer to convert byte order between little endian to big endian.
- Use thresholds in comparisons in lieu of equality.
- Use the strictfp keyword to ensure consistent floating point results across different JVMs and platforms.
- If possible, use integers instead of floating point numbers.
- Use the BigDecimal class to provide better precision such as for monetary or financial calculations and to mitigate rounding issues, when performing high precision arithmetic or where more granular control is needed by.
- For class-based enums, ensure that enum values are not mutable by making members in an enum type private, by setting the members in the constructor and by not providing setter methods.
- Set all enum fields to be final.
- Use an enum type to select from a limited set of choices to make possible the use of tools to detect omissions of possible values such as in switch statements, see clause 6.27.
- Check the value of a larger type before converting it to a smaller type to see if the value in the larger type is within the range of the smaller type.
- Use comments to document cases where intentional loss of data due to narrowing is expected and acceptable.
- Be aware that conversion from certain integral types to floating types can result in a loss of the least significant bits.
- Include checks for null prior to making use of objects. Less preferably, handle exceptions raised by attempts to dereference null values.
- Consider using the Optional class (java.util.Optional) to handle objects via the method isPresent() without risking raising an exception.
- Use defensive programming techniques to check whether an operation will overflow or underflow the receiving data type. For example

o Check that an operation on an integer value will not cause wrapping, unless it can be shown that wrapping cannot occur. Any of the following operators have the potential to wrap:

a + b a – b a * b a++ ++a a-- --a

a += b a -= b a *= b a << b a<<=b -a

o Check that an operation on a floating point value will not cause an overflow or underflow, unless it can be shown that either cannot occur. Any of the following operators have the potential to overflow or underflow:

a + b a – b a * b a/b a%b a++ ++a a--

--a a += b a -= b a *= b a /= b a %= b a << b

a <<= b -a

These techniques can be omitted if it can be shown by static analysis (e.g. at compile time) that overflow or underflow is not possible.

- Include both positive and negative values in any testing of calculations involving right shifts to ensure correct operation.
- Use names that are clear and non-confusing.
- Use consistency in choosing names.
- Keep names short and concise in order to make the code easier to understand.
- Choose names that are rich in meaning.
- Use compilers and analysis tools to identify potential dead stores in the program.
- Mark all variables observable by another thread or hardware agent as volatile, also see 6.61 Concurrent data access [CGX].
- Resolve all compiler warnings for unused variables. Having an unused variable in code indicates that either warnings were turned off during compilation or were ignored by the developer.
- Ensure that when the identifier that a method uses is identical to an identifier in the class that the correct identifier is used through the use or non-use of “this”.
- Choose unique names for any publicly visible identifiers, public utility classes, interfaces and packages.
- Use parentheses when combining operations in an expression to unambiguously specify the programmer’s intent.
- Do not embed ++, --, etc. in other expressions.
- Simplify expressions to reduce or eliminate side effects, to avoid potential confusion and to improve maintainability.
- Do not have side effects in assert statements.
- Explain statements with interspersed comments to clarify programming functionality and help future maintainers understand the intent and nuances of the code.

- Avoid assignments embedded within expressions.
- Give null statements a source line of their own. This, combined with enforcement by static analysis, would make clearer the intention that a statement was meant to be a null statement.
- Use "//" comment syntax instead of "/*…*/" comment syntax to avoid the inadvertent commenting out of sections of code.
- Use an IDE that adds additional capabilities to detect dead or unreachable code.
- Adopt a coding style that requires every nonempty case statement to be terminated with a break statement. Alternatively, if a direct fall through from one nonempty case to another is required that would violate the coding style, then this should be clearly documented by a comment, preferably one recognized by the analysis tool used.
- Adopt a coding style that permits your language processor and analysis tools to verify that all cases are covered. Where this is not possible, use a default clause that diagnoses the error.
- Adopt a coding style that requires the default clause to be either the first or last clause in the switch statement to assist the maintenance of complex switch statements.
- Consider using the Java switch expression in place of simple switch statements, where it is clearly applicable, such as day = (switch …)
- Enclose the bodies of if, else, while, for, and similar constructs in braces to disambiguate the control flow.
- Do not modify a loop control variable within a loop.
- Declare all enhanced for statement loop variables final to cause the Java compiler to flag and reject any assignments made to the loop variable.
- Do not use floating point types as a loop control variable.
- Use enhanced for loops to eliminate the need for a loop control variable.
- Use careful programming, testing of boundary conditions, and static analysis tools to detect off-by-one errors in Java.
- Write clear and concise structured code to make code as understandable as possible.
- Restrict the use of continue and break in loops to encourage more structured programming.
- Use care when using expressions with side effects as parameters to methods.
- Write code to account for potential aliasing among parameters, including the current instance this.
- Avoid the use of expressions with side effects for multiple parameters to functions, since the order in which the parameters are evaluated and hence the side effects occur is unspecified.

Do not use the variable argument feature except in rare instances. Instead, use arrays to pass parameters.

- If recursion is used, then catch the java.lang.OutOfMemoryError exception to handle insufficient storage due to r execution.

- Use try-with-resources which extends the behaviour of the try/catch construct to allow access to resources without having to close them afterwards as the resource closures are done automatically.
- Use unchecked exceptions just in case an unanticipated exception occurs.
- Use try-with-resources for automatic resource management.
- Consider segregating intended reinterpretation operations into distinct subprograms, as the presence of reinterpretation greatly complicates program understanding and static analysis,
- Ensure that deep-copied objects are initialized properly.
- Be careful of excessive memory use when using deep copying.
- Use a heap-analyzer tool to assist in detecting memory leaks.
- Enable verbose garbage collection to see a detailed trace of the garbage collector actions.
- Use Java profiler tools that monitor and diagnose memory leaks.
- Set references to null once they are no longer needed so that the garbage collector can collect the designated object.
- Use reference objects from the java.lang.ref package instead of directly referencing objects to allow them to be easily garbage collected.
- Use generic wildcards carefully and only when needed.
- Follow the acronym PECS for “Producer Extends, Consumer Super” – use extends when getting values out of a data structure, use super when putting values into a data structure, and use and explicit type when doing both. See 6.42 Violations of the Liskov substitution principle or the contract model.
- Use different names for methods to get different signatures.
- Use composition as an alternative to inheritance
- Use interfaces when multiple inheritance is required.
- Keep the inheritance graph as shallow as possible to simplify the review of inheritance relationships and method overridings.
- Use assertions to implement precondition and postcondition checks.
- Prevent redispatching where it is not necessary, and document the behaviour.
- Do not make assumptions about the values of parameters.
- Use preconditions to validate parameters

- Do not assume that the calling or receiving function will be range checking a parameter. Therefore, establish a strategy for each interface to check parameters in either the calling or receiving routines.

- Use a foreign function interface such as JNI to provide a clear separation between Java and the other language.

- Use foreign function interfaces carefully as they can be error prone and lack transparency making debugging harder.

- Be aware that native code can lack many of the protections afforded by Java such as bounds checks on structures not being performed on native methods and explicitly perform the necessary checks.

- Minimize the use of those issues known to be error-prone when interfacing between languages, such as:

1. passing character strings
2. dimension, bounds and layout issues of arrays
3. interfacing with other parameter mechanisms such as call by reference, value or name
4. handling faults, exceptions and errors, and
5. bit representation.

- Do not dynamically modify classes, unless there is a documented rationale and the rationale is carefully reviewed.

- As appropriate, verify through the use of signatures that dynamically linked or shared code being used is the same as that which was tested.

- If possible, retest when dynamically linked or shared code has changed before using the application.

- Use a tool, if possible, to automatically create interface wrappers.

- Be wary of making assumptions about argument lists, data structures and error handling mechanisms, as other languages are likely to have differences in these areas.

- Always have an appropriate response for checked exceptions since even things that should never happen do happen occasionally.

- Analyze the Java warnings “uses unsafe or unchecked operations” to determine whether action is needed or whether it is appropriate to leave the code as is.

- Only use the class sun.misc.Unsafe in specialized instances where the capabilities it provides are essential. It should not be used for everyday use to evade Java protections.

- Document all uses of unsafe code with in-place comments and provide evidence that all such uses function correctly and safely.

- Name unsafe extensions with names that retain the unsafe nomenclature.
- Specify coding standards that restricts or bans the use of features or combinations of features that have been observed to lead to vulnerabilities in the operational environment for which the software is intended.
- Do not rely on unspecified behaviour because the behaviour can change at each instance. Any code that makes assumptions about the behaviour of something that is unspecified should be replaced.
- Reduce the number of temporary objects to minimize the impact and need for garbage collection.
- Increase the Java heap size to reduce the frequency and amount of time spent doing garbage collection.
- Enable verbose garbage collection and profiling to locate and fix memory leaks to reduce need for garbage collection.
- Use the Java annotation and a Javadoc tag to indicate deprecation of classes, methods, or member fields
- Rewrite code that uses deprecated language features to remove such use, whenever possible.
- Check the maximum number of allowed processes per user limit and raise the limit if appropriate. For example, on Linux systems, check the limit using the “ulimit –u” command.
- Increase the amount of native memory available by lowering the size of the Java heap by using the –Xmx option.
- Lower the number of threads if possible.
- Check the amount of free disk space. For example, on Linux systems, check the amount of free disk space by using the “df” command.
- Consider making the head of task groups … (research – AI – Stephen)
- Use a synchronized condition to indicate that a thread should exit.
- Alternatively, use Thread.interrupt() method to interrupt a thread to indicate that the thread should exit.
- Form happens-before relationships through the use of the java.util.concurrent package.
- Use the volatile keyword to force a data element to always go to main memory for its reads and writes
- Mark as private all data components that are accessed by multiple threads.
- Apply the synchronized keyword to methods that access the same data components of an object to prevent multiple invocations of methods on the same object from interleaving.
- Access all private data components only through getter and setter methods.

- Use the java.lang.Thread.isAlive() method to check as needed to see if a thread is still active.
- Use the Java ExecutorService framework for thread group management.
- Use the Thread.setDefaultUncaughtExceptionHandler() method in thread groups to handle unexpected exceptions.
- Use the intrinsic monitor features coupled with conventional techniques to avoid lock protocol errors.
- Normalize strings before validating them.
- Canonicalize path names and other strings that have more than one possible representation.
- Use Java classes for importing, exporting, and manipulating strings.