Effective Java Textbook
CHAPTER 2: CREATING AND DESTROYING OBJECTS

Item 1: Static factory methods instead of constructors:

Advantages:

1) Static factory methods unlike constructors have names.
2) Static factory methods do not require a new object to be created each time the method is invoked.
3) Static factory methods can return an object of any subtype of their return type.
4) Static factory methods can reduce the verbosity of creating parameterized type instances.

Disadvantages:

1) If only static factory methods are provided classes without public or protected constructors cannot be subclassed
2) Static factory methods are not readily distinguishable from other static methods.

Item 2: Consider a builder when faced with many constructor parameters:

Telescoping constructor pattern:

- When using a telescoping constructor pattern you are providing a constructor with only the required parameters, another with a single optional parameter a third with two optional parameters and so on.
- The telescoping constructor pattern works, but it is hard to write client code when there are many parameters, and harder still to read it.

JavaBeans pattern:

- When using the JavaBeans pattern you call a parameterless constructor to create the object and then call setter methods to set each required parameter and each optional parameter of interest.
- Pattern has none of the disadvantages of the telescoping constructor pattern. It is easy to create instances and easy to read the resulting code.
- Disadvantages include that a JavaBean may be in an inconsistent state partway through its construction and also that the JavaBeans pattern precludes the possibility of making a class immutable.

Builder pattern:

- Instead of making the desired object directly, the client calls a constructor with all of the required parameters and gets a builder object. Then the client calls setter-like methods on the builder object to set each optional parameter of interest. Finally, the client calls a parameterless build method to generate the object, which is immutable.
- The Builder pattern simulates named optional parameters.
- The Builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters.

Item 3: Enforce the singleton property with a private constructor or an enum type:
- A singleton is simply a class that is instantiated exactly once. Making a class a singleton can make it difficult to test its clients as it's impossible to substitute a mock implementation for a singleton unless it implements an interface that serves as its type.
- A single-element enum type is the best way to implement a singleton as it provides an ironclad guarantee against multiple instantiation.

Item 4: Enforce noninstantiability with a private constructor:
- Attempting to enforce noninstantiability by making a class abstract does not work. The class can be subclassed and the subclass instantiated.
- A class can be made noninstantiable by including a private constructor. This guarantees that the class will never be instantiated as it is inaccessibly outside of the class. It also prevents the class from being subclassed.

Item 5: Avoid creating unnecessary objects:
- Prefer primitives to boxed primitives, and watch out for unintentional autoboxing.

Item 6: Eliminate obsolete object references:
- Nulling out obsolete object references is a simple way to fix the problem of memory leaks. Nulling out object references should be the exception rather than the norm.
- Whenever a class manages its own memory, the programmer should be alert for memory leaks.
- Another common source of memory leaks is caches. Once you put an object reference into a cache, it's easy to forget that it's there and leave it in the cache long after it becomes irrelevant.
- A third common source of memory leaks is listeners and other callbacks. If you implement an API where clients register callbacks but don't deregister them explicitly, they will accumulate unless you take some action. The best way to ensure that callbacks are garbage collected promptly is to store only weak references to them.

Item 7: Avoid finalizers:

- Finalizers are unpredictable, often dangerous and generally unnecessary. Their use can cause erratic behaviour, poor performance, and portability programs.
- You should never do anything time-critical in a finalizer as it can take arbitrarily long between the time than an object becomes unreachable and the time that its finalizer is executed.
- You should never depend on a finalizer to update critical persistent state as it is entirely possible that a program terminates without executing finalizers on some objects that are no longer reachable.
- There is a severe performance penalty for using finalizers.
- Instead of writing a finalizer an explicit termination method can be provided. These are typically used in combination with the try-finally construct to ensure termination.
- A finalizer can be used to act as a "safety net" in case the owner of an object forgets to call its explicit termination method. In this case the finalizer should log a warning if it finds that the resource has not been terminated.

CHAPTER 3: METHODS COMMON TO ALL OBJECTS

Item 8: Obey the general contract when overriding equals:

- Overriding the equals method seems simple, but there are many ways to get it wrong. The easiest way to avoid problems is not to override the equals method, in which case each instance of the class is equal only to itself.
- This is the right thing to do if
    - Each instance of the class is inherently unique.
    - You don't care whether the class provides a "logical equality" test.
    - A superclass has already overridden equals, and the superclass behaviour is appropriate for this class.
    - The class is private or package-private, and you are certain that its equals method will never be invoked.
- Reflexivity: The first requirement says merely that an object must be equal to itself.
- Symmetry: The second requirement says that any two objects must agree on whether they are equal.
- Transitivity: The third requirement of the equals contract says that if one object is equal to a second and the second object is equal to a third, then the first object must be equal to the third.
- Consistency – The fourth requirement of the equals contract says that if two objects are equal they must remain equal for all time unless one of them is modified.
- "Non-nullity"- The final requirement says that all objects must be unequal to null.
- Once you've violated the equals contract, you simply don't know how other objects will behave when confronted with your object.

- There is no way to extend an instantiable class and add a value component while preserving the equals contract.
- Do not write an equals method that depends on unreliable resources.
- To ensure a high-quality equals method:
    - Use the == operator to check if the argument is a reference to this object.
    - Use the instanceof operator to check if the argument has the correct type.
    - Cast the argument to the correct type.
    - For each field in the class, check if that field of the argument matches the corresponding field of this object.
    - When you are finished writing your equals method, ask yourself is it symmetric, transitive and consistent?
- Always override hashCode when you override equals.
- Don't substitute another type for Object in the equals declaration.

Item 9: Always override hashCode when you override equals:

- You must override hashCode in every class that overrides equals. Failure to do so will result in a violation of the general contract for Object.hashCode.
- The key provision that is violated when you fail to override hashCode is equal objects must have equal hash codes.
- Do not exclude significant parts of an object from the hash code computation to improve performance.

Item 10: Always override toString:

- Providing a good toString implementation makes your class much more pleasant to use.
- The toString method should return all of the interesting information contained in the object.
- Whether or not you decide to specify the format, you should clearly document your intentions.
- Provide programmatic access to all of the information contained in the value returned by toString.

Item 11: Override clone judiciously

- If you override the clone method in a nonfinal class, you should return an object obtained by invoking super.clone.
- A class that implements Cloneable is expected to provide a properly functioning public clone method.
- Never make the client do anything the library can do for the client.

- The clone method functions as another constructor; you must ensure that it does no harm to the original object and that it properly establishes invariants on the clone.
- The clone architecture is incompatible with normal use of final fields referring to mutable objects.
- If not implementing a well-behaved clone method you are better off providing an alternative means of object copying, or simply not providing the capability.
- A fine approach to object copying is to provide a copy constructor or copy factory. A copy factory is the static factory analog of a copy constructor.

Item 12: Consider implementing Comparable
- By implementing Comparable, a class indicates that its instances have a natural ordering.

CHAPTER 4: CLASSES AND INTERFACES

Item 13: Minimize the accessibility and members:
- Make each class or member as inaccessibly as possible.
- Private: The member is accessible only from the top-level class where it is declared.
- Package-private: The member is accessible from any class in the package where it is declared. This is technically known as default access.
- Protected: The member is accessible from subclasses of the class where it is declared and from any class in the package where it is declared.
- Public: The member is accessible from anywhere.
- Instance fields should never be public. If its public you give up the ability to limit the values that can be stored in the field. You also give up the ability to take any action when the field is modified so classes with public mutable fields are not thread-safe.
- A class should not have a public static final array field, or an accessor that returns such a field.

Item 14: In public classes, use accessor methods, not public fields:
- If a class is accessible outside its package, provide accessor methods. However if a class is package-private or is a private nested class, there is nothing inherently wrong with exposing its data fields.

Item 15: Minimize mutability:
- To make a class immutable
    - Don’t provide any methods that modify the object’s state (mutators)

    - Ensure that the class can't be extended.
    - Make all fields final.
    - Make all fields private.
    - Ensure exclusive access to any mutable components.
- Immutable objects are simple and are inherently thread-safe; they require no synchronization.
- Not only can immutable objects be shared, but you can also share their internals.
- Immutable objects make great building blocks for other objects.
- The only real disadvantage of immutable classes is that they require a separate object for each distinct value.
- Classes should be immutable unless there's a very good reason to make them mutable.
- If a class cannot be made immutable, limit its mutability as much as possible.
- Make every field final unless there is a compelling reason to make it nonfinal.

Item 16: Favour composition over inheritance:

- Unlike method invocation, inheritance violates encapsulation. In other words, a subclass depends on the implementation details of its superclass for its proper function. The superclass's implementation may change from release to release, and if it does, the subclass may break.
- Instead of extending an existing class, give your new class a private field that references an instance of the existing class. This design is called composition because the existing class becomes a component of the new one.
- Each instance method in the new class invokes the corresponding method on the contained instance of the existing class and returns the results. This is known as forwarding, and the methods in the new class are known as forwarding methods
- To summarize, inheritance is powerful, but it is problematic because it violates encapsulation.
- It is appropriate only when a genuine subtype relationship exists between the subclass and the superclass. Even then, inheritance may lead to fragility if the subclass is in a different package and the superclass is not designed for inheritance.
- To avoid this use composition and forwarding, especially if an appropriate interface to implement a wrapper class exists.

Item 17: Design and document for inheritance or else prohibit it:

- A class must document its self-use of overridable methods.
- A class may have to provide hooks into its internal workings in the form of judiciously chosen protected methods or, in rare instances, protected fields.

- The only way to test a class designed for inheritance is to write subclasses. When designing a class for inheritance it is likely to achieve wide use. Therefore you must test your class by writing subclasses before you release it.
- Constructors must not invoke overridable methods, directly or indirectly.
- Neither clone nor readObject may invoke an overridable method, directly or indirectly.
- Designing a class for inheritance places substantial limitations on the class.
- Subclassing should be prohibited in classes that are not designed and documented to be safely subclassed.

Item 18: Prefer interfaces to abstract classes:

- Existing classes can be easily retrofitted to implement a new interface. All you have to do is add the required methods if they don't yet exist and add an implements clause to the class declaration.
- Interfaces are ideal for defining mixins. A mixin is a type that a class can implement in addition to its "primary type" to declare that it provides some option behaviour.
- Interfaces allow the construction of non-hierarchical type frameworks.
- Interfaces enable safe, powerful functionality enhancements via the wrapper class idiom. If you use abstract classes to define types, you leave the programmer who wants to add functionality with no alternative but to use inheritance.
- You can combine the virtues of interfaces and abstract classes by providing an abstract skeletal implementation class to go with each nontrivial interface that you export.
- Using abstract classes to define types that permit multiple implementations has one great advantage over using interfaces. It is far easier to evolve an abstract class than an interface.
- Once an interface is released and widely implemented, it is almost impossible to change.

Item 19: Use interfaces only to define types:

- The constant interface pattern (which contains no methods) is a poor use of interfaces.

Item 20: Prefer class hierarchies to tagged classes:

- A tagged class is verbose, error-prone, and inefficient. It is just a pallid imitation of a class hierarchy.
- If you are tempted to write a class with an explicit tag field, think about whether the tag could be eliminated and the class replaced by a hierarchy. When you encounter an existing class with a tag field, consider refactoring it into a hierarchy.

Item 21: Use function objects to represent strategies

- A primary use of function pointers is to implement the Strategy pattern. To implement this pattern in Java, declare an interface to represent the strategy, and a class that implements this interface for each concrete strategy.
- When a concrete strategy is used only once, it is typically declared and instantiated as an anonymous class.
- When a concrete strategy is designed for repeated use, it is generally implemented as a private static member class and exported in a public static final field whose type is the strategy interface.

Item 22: Favour static member classes over nonstatic:

- If you declare a member class that does not require access to an enclosing instance, always put the static modifier in its declaration.
- There are four different kinds of nested classes.
- If a nested class needs to be visible outside of a single method or is too long to fit comfortably inside a method, use a member class. If each instance of the member class needs a reference to its enclosing instance, make it nonstatic; otherwise make it static.
- Assuming the class belongs inside a method, if you need to create instances from only one location and there is a pre-existing type that characterizes the class, make it an anonymous class; otherwise make it a local class.

CHAPTER 5: GENERICS

Item 23: Don't use raw types in new code

- A class or interface whose declaration has one or more type parameters is a generic class or interface.
- If you use raw types, you lose all the safety and expressiveness benefits of generics.
- You lose type safety if you use a raw type like List, but not if you use a parameterized type like List<Object>
- You can't put any element (other than null) into a Collection<?>.
- When using the instanceof operator with generic types once o is determined to be a Set, you must cast it to the wildcard type Set<?>, not the raw type Set.

Summary:

Parameterized type	List<String>

Actual type parameter	String
Generic type	List<E>
Formal type parameter	E
Unbounded wildcard type	List<?>
Bounded type parameter	<E extends Number>
Recursive type bound	<T extends Comparable<T>>
Bounded wildcard type	List<? extends Number>
Generic method	static <E> List<E> asList(E[] a)
Type token	String.class
Raw type	List

Item 24: Eliminate unchecked warnings

- Eliminate every unchecked warning that you can. If you can't eliminate a warning, and you can prove that the code that provoked the warning is typesafe, then suppress the warning with an @SuppressWarnings("unchecked") annotation.
- Always use the Suppress-Warnings annotation on the smallest scope possible.
- Every time you use an @SuppressWarnings("unchecked") annotation, add a comment saying why it's safe to do so.

Item 25: Prefer lists to arrays

- Arrays are covariant which means that if Sub is a subtype of Super, then the array type Sub[ ] is a subtype of Super[ ].
- Generics are invariant and erased.
- Arrays provide runtime type safety but not compile-time type safety and vice versa for generics.
- Generally speaking, arrays and generics don't mix well. If you find yourself mixing them and getting compile-time errors or warnings, your first impulse should be to replace the arrays with lists.

Item 26: Favour generic types

- Generic types are safer and easier to use than types that require casts in client code.
- When you design new types, make sure that they can be used without such casts. This will often mean making the types generic.

- Generify your existing types as time permits. This will make life easier for new users of these types without breaking existing clients.

Item 27: Favour generic methods

- The type parameter list, which declares the type parameters, goes between the method's modifiers and its return type.
- Generic methods are safer and easier to use than methods that require their clients to cash input parameters and return values.
- You should make sure your new methods can be used without casts, which will often mean making them generic.
- You should generify your existing methods to make life easier for users without breaking existing clients.

Item 28: Use bounded wildcards to increase API flexibility

- For maximum flexibility, use wildcard types on input parameters that represent producers or consumers.
- Do not use wildcard types as return types. Rather than providing additional flexibility for your users, it would force them to use wildcard types in client code.
- If the user of a class has to think about wildcard types, there is probably something wrong with the class's API.
- Comparables are always consumers, so you should always use Comparable<? super T> in preference to Comparable<T>. The same is true of comparators.
- If a type parameter appears only once in a method declaration, replace it with a wildcard.
- If you write a library that will be widely used, the proper use of wildcard types should be considered mandatory.

Item 29: Consider typesafe heterogeneous contains

- The normal use of generics, exemplified by the collections APIs, restricts you to a fixed number of type parameters per container.
- You can get around this restriction by placing the type parameter on the key rather than the container.
- You can use Class objects as keys for such typesafe heterogeneous containers.
- A Class object used in this fashion is called a type token. You can also use a custom key type. For example, you could have a DatabaseRow type representing a database row (the container, and a generic type Column<T> as its key.

CHAPTER 6: ENUMS AND ANNOTATIONS:

Item 30: Use enums instead of int constants:

- An enumerated type is a type whose legal values consist of a fixed set of constants.
- To associate data with enum constants, declare instance fields and write a constructor that takes the data and stores it in the fields.
- Switches on enums are good for augmenting external enum types with constant-specific behaviour.

- Enums are far more readable, safer and powerful than int constants. Many enums require no explicit constructors or members, but many others benefit from associating data with each constant and providing methods whose behaviour is affected by this data.
- Far fewer enums benefit from associating multiple behaviours with a single method. In this case, prefer constant-specific methods to enums that switch on their own values.
- Consider the strategy enum pattern if multiple enum constants share common behaviours.

Item 31: Use instance fields instead of ordinals

- All enums have an ordinal method, which returns the numerical position of each enum constant in its type.
- Never derive a value associated with an enum from its ordinal; store it in an instance field instead.

Item 32: Use EnumSet instead of bit fields

- Just because an enumerated type will be used in sets, there is no reason to represent it with bit fields.
- The EnumSet class combines the conciseness and performance of bit fields with all the many advantages of enum types.
- The one real disadvantage of EnumSet is that it is not possible to create an immutable EnumSet.

Item 33: Use EnumMap instead of ordinal indexing

- It is rarely appropriate to use ordinals to index arrays; instead use EnumMap.

Item 34: Emulate extensible enums with interfaces

- While you cannot write an extensible enum type, you can emulate it by writing an interface to go with a basic enum type that implements the interface.
- This allows the clients to write their own enums that implement the interface.
- The enums can then be used wherever the basic enum type can be used, assuming the APIs are written in terms of the interface.

Item 35: Prefer annotations to naming patterns

- Naming patterns can mean that typographical errors may result in silent failures. There is also no way to ensure that they are used only on appropriate program elements. They also provide no good way to associate parameter values with program elements.
- There is simply no reason to use naming patterns now that we have annotations.
- All programmers should use the predefined annotation types provided by the Java platform.

Item 36: Consistently use the Override annotation

- Use the Override annotation on every method declaration that you believe to override a superclass declaration.
- The compiler can protect you from a great many errors if you use the Override annotation on every method declaration that you believe to override a supertype declaration, with one exception.
- In concrete classes, you need not annotate methods that you believe to override abstract method declarations. (Though it is no harmful to do so)

Item 37: Use marker interfaces to define types

- A marker interface is an interface that contains no method declarations, but merely designates a class that implements the interface as having some property.
- Marker interfaces define a type that is implemented by instances of the marked class; marker annotations do not.
- If you find yourself writing a marker annotation type whose target is ElementType.TYPE, take the time to figure out whether it really should be an annotation type, or whether a marker interface would be more appropriate.

CHAPTER 7: METHODS

Item 38: Check parameters for validity

- Each time you write a method or constructor you should think about what restrictions exist on its parameters.
- You should document these restrictions and enforce them with explicit checks at the beginning of the method body.
- The modest work that it entails will be paid back with interest the first time a validity check fails.

Item 39: Make defensive copies when needed

- Java is a safe language. This means in absence of native methods it is immune to buffer overruns, array overruns, wild pointers, and other memory corruption errors.
- You must program defensively, with the assumption that clients of your class will do their best to destroy its invariants.
- It is essential to make a defensive copy of each mutable parameter to the constructor.
- Defensive copies are made before checking the validity of the parameters, and the validity check is performed on the copies rather than on the originals.
- Do not use the clone method to make a defensive copy of a parameter whose type is subclassable by untrusted parties.
- Modify accessor to return defensive copies of mutable internal fields.
- If the cost of the copy would be prohibitive and the class trusts its clients not to modify the components inappropriately, then the defensive copy may be replaces by documentation outlining the client's responsibility not to modify the affected components.

Item 40: Design method signatures carefully

- Choose method names carefully. Names should always obey the standard naming conventions.
- Don’t go overboard in providing convenience methods. When in doubt, leave it out.
- Avoid long parameter lists. Aim for four parameters or fewer.
- Long sequences of identically types parameters are especially harmful. Not only won’t users be able to remember the order of the parameters, but when they transpose parameters accidentally, their programs will still compile and run.
- For parameter types, favour interfaces over classes.
- Prefer two-element enum types to Boolean parameters.

Item 41: Use overloading judiciously

- The choice of which overloading to invoke is made at compile time.
- Selection among overloaded methods is static, while among overridden methods is dynamic.
- Avoid confusing users of over-loading. A safe, conservative policy is never to export two overloadings with the same number of parameters.
- In some cases, especially where constructors are involved, it may be impossible to follow this advice. In that case, you should at least avoid situations where the same set of parameters can be passed to different overloadings by the addition of casts.

Item 42: Use varargs judiciously

- Formally known as variable arity methods, these accept zero or more arguments of a specified type.
- Don’t retrofit every method that has a final array parameter; use varargs only when a call really operates on a variable-length sequence of values.
- Varargs should not be overused as they can produce confusing results if used inappropriately.

Item 43: Return empty arrays or collections, not nulls

- There is no reason ever to return null from an array or collection-valued method instead of returning an empty array or collection.

Item 44: Write doc comments for all exposed API elements

- To document your API properly, you must precede every exported class, interface, constructor, method, and field declaration with a doc comment.
- The doc comment for a method should describe succinctly the contract between the method and its client.
- It is no longer necessary to use the HTML <code> or <tt> tags in doc comments: the Javadoc {@code} tag is preferable because it eliminates the need to escape HTML metacharacters.

- No two members or constructors in a class or interface should have the same summary description.
- When documenting a generic type or method, be sure to document all type parameters.
- When documenting an enum type, be sure to document the constants.
- When documenting an annotation type, be sure to document any members.

CHAPTER 8: GENERAL PROGRAMMING

Item 45: Minimize the scope of local variables

- The most powerful technique for minimizing the scope of a local variable is to declare it where it is first used.
- Nearly every local variable declaration should contain an initialize.
- Keep methods small and focused. If you combine two activities in the same method, local variables relevant to one activity may be in the scope of the code performing the other activity. To prevent this, simply separate the method into two.

Item 46: Prefer for-each loops to traditional for loops

- A for-each loop provides compelling advantages over the traditional for loop in clarity and bug prevention.
- Unfortunately there are three common situations where you can't use a for-each loop.
    - Filtering: If you need to traverse a collection and remove selected elements, then you need to use an explicit iterator so that you can call its remover method.
    - Transforming: If you need to traverse a list or array and replace some or all of the values of its elements, then you need the list iterator or array index in order to set the value of an element.
    - Parallel iteration: If you need to traverse multiple collections in parallel, then you need explicit control over the iterator or index variable, so that all iterators or index variables can be advanced in lockstep.
- In any of the above situations use a regular for loop.

Item 47: Know and use the libraries

- By using a standard library, you take advantage of knowledge of the experts who wrote it and the experience of those who used it before you.
- Numerous features are added to the libraries in every major release, and it pays to keep abreast of these additions.
- Every programmer should be familiar with the contents of java.lang, java.util, and to a lesser extent, java.io.

Item 48: Avoid float and double if exact answers are required

- The float and double types are particularly ill-suited for monetary calculations because it is impossible to represent 0.1 (or any other negative power of ten) as a float or double exactly.
- Use BigDecimal if you want the system to keep track of the decimal point and you don't mind the inconvenience and cost of not using a primitive type.
- If the quantities don't exceed nine decimal digits, you can use int.
- If the quantities don't exceed eighteen digits, you can use long.
- If the quantities might exceed eighteen digits, you must use BigDecimal.

Item 49: Prefer primitive types to boxed primitives

- Applying the == operator to boxed primitives is almost always wrong.
- When you mix primitives and boxed primitives in a single operation, the boxed primitive is auto-unboxed.
- You can't put primitives in collections, so you're forced to use boxed primitives.
- Autoboxing reduces the verbosity, but not the danger, of using boxed primitives.
- When your program does unboxing, it can throw a NullPointerException.

Item 50: Avoid strings where other types are more appropriate

- Strings are poor substitutes for other value types.
- Strings are poor substitutes for enum types.
- Strings are poor substitutes for aggregate types.
- Strings are poor substitutes for capabilities.
- Used inappropriately, strings are more cumbersome, less flexible, slower and more error prone than other types.

Item 51: Beware the performance of string concatenation

- The string concatenation operator (+) is a convenient way to combine a few strings into one.
- Using the string concatenation operator repeatedly to concatenate n strings requires time quadratic in n. Unless performance is irrelevant don't use the string concatenation operator.
- To achieve acceptable performance, use a StringBuilder in place of a string.

Item 52: Refer to objects by their interfaces

- If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types.
- If you get into the habit of using interfaces as types, your program will be much more flexible.
- It is entirely appropriate to refer to an object by a class rather than an interface if no appropriate interface exists.

Item 53: Prefer interfaces to reflection

- Constructor, method and field instances let you manipulate their underlying counterparts reflectively: you can construct instances, invoke methods, and access fields of the underlying class by invoking methods on the Constructor, Method, and Field instances.
- Reflection allows one class to use another, even if the latter class did not exist when the former was compiled.
- This comes at a price however:
    - You lose all benefits of compile-time type checking, including exception checking.
    - The code required to perform reflective access is clumsy and verbose.
    - Performance suffers. Reflective method invocation is much slower than normal method invocation.
- As a rule, objects should not be accessed reflectively in normal applications at runtime.
- You can obtain many of the benefits of reflection while incurring few of its costs by using it only in a very limited form.
- You can create instances reflectively and access them normally via their interface or superclass.

Item 54: Use native methods judiciously

- Native methods are special methods written in native programming languages such as C or C++.
- It is rarely advisable to use native methods for improved performance.
- Native methods have serious disadvantages. Because native languages are not safe, applications using native methods are no longer immune to memory corruption errors.

Item 55: Optimize judiciously

- Strive to write good programs rather than fast ones.
- Strive to avoid design decisions that limit performance.
- Consider the performance consequences of your API design decisions.
- It is a very bad idea to warp an API to achieve good performance.
- Measure performance before and after each attempted optimization.

Item 56: Adhere to generally accepted naming conventions

Identifier Type	Examples
Package	com.good.inject, org.joda.time.format

Class or Interface	Timer, FutureTask, LinkedHashMap, HttpServlet
Method or Field	Remove, ensureCapacity, getCrc
Constant Field	MIN_VALUE, NEGATIVE_INFITITY
Local Variable	i, xref, houseNumber
Type Parameter	T, E, K, V, X, T1, T2

CHAPTER 9: EXCEPTIONS

Item 57: Use exceptions only for exceptional conditions

- When used to best advantage, exceptions can improve a program's readability, reliability, and maintainability.
- Exceptions are to be used only for exceptional conditions; they should never be used for ordinary control flow.
- A well-designed API must not force its clients to use exceptions for ordinary control flow.

Item 58: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

- Use checked exceptions for conditions from which the caller can reasonably be expected to recover.
- Use runtime exceptions to indicate programming errors.
- All the unchecked throwables you implement should subclass RuntimeException.

Item 59: Avoid unnecessary use of checked exceptions

- The overuse of checked exceptions can make an API far less pleasant to use.
- If a method throws one or more checked exceptions, the code that invokes the method must handle the exceptions in one or more catch blocks, or it must declare that it throws the exceptions and let them propagate outward.
- An additional burden caused by a checked exception is substantially higher if it is the sole checked exception thrown by a method.
- One way to turn a checked exception into an unchecked exception is to break the method that throws the exception into two methods. The first returns a Boolean that indicated whether the exception would be thrown.

Item 60: Favour the use of standard exceptions

Exception	**Occasion for use**
IllegalArgumentException	Non-null parameter value is

	inappropriate
IllegalStateException	Object state is inappropriate for method invocation
NullPointerException	Parameter value is null where prohibited
IndexOutOfBoundsException	Index parameter value is out of range
ConcurrentModificationException	Concurrent modification of an object has been detected where it is prohibited
UnsupportedOperationException	Object does not support method

Item 61: Throw exceptions appropriate to the abstraction

- Higher layers should catch lower-level exceptions and, in their place, throw exceptions that can be explained in terms of the higher-level abstraction. This is known as exception translation.
- While exception translation is superior to mindless propagation of exceptions from lower layers, it should not be overused. The best way to deal with exceptions from lower layers is to avoid them, by ensuring that lower-level methods succeed.
- If it is impossible to prevent exceptions from lower layers, the next best thing is to have the higher-level silently work around these exceptions.

Item 62: Document all exceptions thrown by each method

- Always declare checked exceptions individually, and document precisely the conditions under which each one is thrown using the Javadoc @throws tag.
- Use the Javadoc @throws tag to document each unchecked exception that a method can throw, but do not use the throws keyword to include unchecked exceptions in the method declaration.
- If an exception is thrown by many methods in a class for the same reason, it is acceptable to document the exception in the class' documentation comment rather than documenting it individually for each method.
- If you fail to document the exceptions that your methods can throw, it will be difficult or impossible for others to make effective use of your classes and interfaces.

Item 63: Include failure-capture information in detail messages

- To capture the failure, the detail message of an exception should contain the values of all parameters and fields that "contributed to the exception."

- One way to ensure that exceptions contain adequate failure-capture information in their detail messages is to require this information in their constructors instead of a string detail message.

Item 64: Strive for failure atomicity

- A failed method invocation should leave the object in the state that it was in prior to the invocation. A method with this property is said to be failure atomic.
- If an object is immutable, failure atomicity is free. If an operation fails, it may prevent a new object from getting created, but it will never leave an existing object in an inconsistent state.
- Another approach is to order the computation so that any part that may fail takes place before any part that modifies the object.
- Another approach is to perform the operation on a temporary copy of the object and to replace the contents of the object with the temporary copy once the operation is complete.
- Where this rule is violated, the API documentation should clearly indicate what state the object will be left in.
-

Item 65: Don't ignore exceptions

- An empty catch block defeats the purpose of exceptions.
- At the very least, the catch block should contain a comment explaining why it is appropriate to ignore the exception.

CHAPTER 10: CONCURRENCY

Item 66: Synchronize access to shared mutable data

- The synchronized keyword ensures that only a single thread can execute a method or block at one time.
- Synchronization is required for reliable communication between threads as well as for mutual exclusion.
- This is due to a part of the language specification known as the memory model, which specifies when and how changes made by one thread become visible to others.
- The libraries provide the Thread.stop method, but this method was deprecated long ago because it is inherently unsafe – it can result in data corruption.
- Synchronization has no effect unless both read and write operations are synchronized.
- Mutable data should be not shared. Either share immutable data, or don't share at all. Confine mutable data to a single thread.
- When multiple threads share mutable data, each thread that reads or writes the data must perform synchronization. Without it there is no guarantee that one thread's changes will be visible to another.
- If you need only inter-thread communication, and not mutual exclusion, the volatile modifier is an acceptable form of synchronization, but it can be tricky to use correctly.

Item 67: Avoid excessive synchronization

- To avoid liveness and safety failures, never cede control to the client within a synchronized method or block.
- In other words, inside a synchronized region, do not invoke a method that is designed to be overridden, or one provided by a client in the form of a function object.
- To make this concrete, implement an observable set wrapper. It allows clients to subscribe to notifications when elements are added to the set. This is the Observer pattern.
- Observers subscribe to notifications by invoking the addObserver method and unsubscribe by invoking the removeObserver method.
- As a rule, you should do as little work as possible inside synchronized regions.
- To avoid deadlock and data corruptions, never call an alien method from within a synchronized region.
- Synchronize your class internally only if there is a good reason to do so, and document your decision clearly.

Item 68: Prefer executors and tasks to threads

- A work queue is a class that allowed clients to enqueue work items for asynchronous processing by a background thread.
- An executor framework is a flexible interface-based task execution facility. It creates a work queue that is better in every way with a single line of code.
- With an executor you can
    - Wait for a particular task to complete.
    - Wait for any or all of a collection of tasks to complete.
    - Retrieve results of tasks one by one as they complete.
- If you want more than one thread to process request from the queue simply call a different static factory that creates a different kind of executor service called a thread pool.

Item 69: Prefer concurrency utilities to wait and notify

- Given the difficultly of using wait and notify correctly, you should use higher-level concurrency utilities instead.
- The concurrent collections provide high-performance concurrent implementations of standard collection interfaces such as List, Queue, and Map.
- It is impossible to exclude concurrent activity from a concurrent collection; locking it will have no effect but slow the program.
- ConcurrentHashMap is optimized for retrieval operations, such as get.
- Use ConcurrentHashMap in preference to Collections.synchronizedMap or Hashtable.
- For interval timing, always use System.nanoTime in preference to System.currentTime-Millis.

- Always use the wait loop idiom to invoke the wait method never invoke it outside of a loop.

Item 70: Document thread safety

Item 71: Use lazy initialization judiciously

Item 72: Don’t depend on the thread scheduler

Item 73: Avoid thread groups

CHAPTER 11: SERIALIZATION

Item 74: Implement Serializable judiciously

Item 75: Consider using a custom serialized form

Item 76: Write readObject methods defensively

Item 77: For instance control, prefer enum types to readResolve

Item 78: Consider serialization proxies instead of serialized instances