**Effective Java 2nd Edition**

**Creating and Destroying Objects**

1. **Consider static factory methods instead of constructors**

Advantages:

* Static factory with a well-chosen name easier to use and understand
* Reuse the object
* Can return sub type
* Reduce the verbosity of creating parameterized type instances

public static <K, V> HashMap<K, V> newInstance() {  
 return new HashMap<K, V>();  
}

Map<String, List<String>> m = HashMap.newInstance();

Disadvantages:

* Classes without public or protected constructors cannot be subclassed
* They are not readily distinguishable from other static methods

1. **Consider a builder when faced with many constructor parameters**

* Telescoping constructor pattern (many parameters in constructors)

Hard to read, not scale well

* JavaBean pattern (Create object with a lot of setter methods)

No consistency in validating parameters, class is mutable

* Builder pattern (Pass all the necessary parameters to create a Builder object, fill in optional parameters and call build to create the object)
* Abstract factory using Builder interface)

// A builder for objects of type T  
 public interface Builder<T> {

public T build();  
 }

Tree buildTree(Builder<? extends Node> nodeBuilder) { ... }

1. **Enforce the singleton property with a private constructor or an enum type**

* Making a class a singleton can make it difficult to test its clients
* Singleton with a public final field and private constructor

Reflection attack: Call the private constructor

* Singleton with a static factory method

Problem in Serializable

* Singleton with only one instance in enum

Preferred approach

1. **Enforce non instantiability with a private constructor**

public class UtilityClass { // Noninstantiable utility class

// Suppress default constructor for noninstantiability  
 private UtilityClass() { throw new AssertionError(); }  
}

1. **Avoid creating unnecessary objects**

* An object can always be reused if it is *immutable*
* Static factory method with immutable classes, eg: valueOf
* Prefer primitives to boxed primitives, and watch out for unintentional autoboxing.
* Object pool is a bad idea unless the objects are extremely heavyweight

1. **Eliminate obsolete object references**

* Null out obsolete references
* Whenever an element is freed, any object references contained in the element should be nulled out
* Cache: Use WeakReference
* Listener and callback: Deregister, use WeakReference
* Anticipate memory leaks before they occur and prevent them from happening

1. **Avoid finalizers**

* Finalizers are unpredictable, often dangerous, and generally unnecessary
* Provide an *explicit termination method*
* Finalizer should log a warning if it finds that the resource has not been terminated
* Finalizer should call super.finalize()

// Finalizer Guardian Idiom

public class Foo {  
 // Sole purpose of this object is to finalize outer Foo object

private final Object finalizerGuardian = new Object() {  
 @Override protected void finalize() throws Throwable {  
 ... // Finalize outer Foo object  
 }  
 };

}

**Methods Common to All Objects**

1. **Obey the general contract when overriding equals**
2. **Always override hashCode when you override equals**
3. **Always override toString**
4. **Override clone judiciously**

* If override clone method, should return an object obtained by invoking super.clone

Object.clone will create the right class and copy all the data fields.

* Deep copy
* Better to provide a *copy constructor* or *copy factory*

1. **Consider implementing Comparable**

**Classes and Interfaces**

1. **Minimize the accessibility of classes and members**

* Most important factor is to hide internal data and other implementation details from other modules, clearly separate API and implementation
* Make each class or member as inaccessible as possible

private -> package-private (default) -> protected -> public

* Instance fields should never be public.
* Classes with public mutable fields are not thread-safe
* Public static final field should reference to an inmutable object

Wrong to have a public static final array object or return it

1. **In public classes, use accessor methods, not public fields**

* If a class is accessible outside its package, provide accessor methods, to preserve the flexibility to change the class’s internal representation
* If a class is package-private or is a private nested class, there is nothing inherently wrong with exposing its data fields

1. **Minimize mutability**

* 5 rules to create an immutable class:
  + Don’t provide any methods that modify the object’s state
  + Ensure that the class can’t be extended
  + Make all fields final
    - Alternative: private constructor and static factory, can return package private subclasses. May need type checking, pay attention to serialization
  + Make all fields private
  + Ensure exclusive access to any mutable components
* Benefits: Simple, thread-safe, reuse existing objects,
* Disadvantage: require a separate object for each distinct value, performance not good for multiple step operations
  + Can have a companion class, package private or even public, like String -> StringBuilder
* Classes should be immutable unless there’s a very good reason to make them mutable.
* If a class can- not be made immutable, limit its mutability as much as possible
* Make every field final unless there is a compelling reason to make it non final

1. **Favor composition over inheritance**

* Inheriting from ordinary concrete classes across package boundaries is dangerous
* Inheritance violates encapsulation: Subclass depends on the implementation details of its superclass for its proper function

Eg: addAll() may or may not call add(); same function signature may override or compile error if the superclass changes

* Composition: Wrap the class and forward the function calls (Decorator pattern)

Disadvantages: Callbacks

1. **Design and document for inheritance or else prohibit it**

* The class must document precisely the effects of overriding any method
* A class may have to provide hooks into its internal workings in the form of judiciously chosen protected methods
* The *only* way to test a class designed for inheritance is to write subclasses: Test before release
* Constructors must not invoke overridable methods
* Not a good idea for a class designed for inheritance to implement Serializable and Clonable

neither clone nor readObject may invoke an overridable method, directly or indirectly

1. **Prefer interfaces to abstract classes**

* Existing classes can be easily retrofitted to implement a new interface
* Interfaces are ideal for defining mixins (some optional behavior, eg: comparable)
* Interfaces allow the construction of nonhierarchical type frameworks (eg: Singer also as a SongWriter)
* Interfaces enable safe, powerful functionality enhancements via the wrapper class
* AbstractInterface: Providing an abstract skeletal implementation class to go with each nontrivial interface exported
* Abstract advantage: Far easier to evolve an abstract class than an interface

Add a function with default implementation

Almost impossible to change an interface once it is released and widely implemented

1. **Use interfaces only to define types**

* Don’t use interface to define constants: User may implement the interface
* Use class with a private constructor

1. **Prefer class hierarchies to tagged classes**

* Tagged class: One class implements more than one flavors or behaviors
* Use hierachied classes

1. **Use function objects to represent strategies**

// Sample Strategy interface  
 public interface Comparator<T> {  
 public int compare(T t1, T t2);  
 }

1. **Favor static member classes over nonstatic**

* 4 nested classes:

*static member classes*, *nonstatic member classes*, *anonymous classes*, and *local classes*

* *always* use static member class if it does not require access to the enclosing instanc

**Generics**

1. **Don’t use raw types in new code**

* If you use raw types, you lose all the safety and expressiveness benefits of generics
* Use List<String> instead of List.
* If need to use generic type List, use List<?> (Almost like an immutable list)
  + you can’t put any element (other than null) into a Collection<?>

1. **Eliminate unchecked warnings**

* Eliminate every unchecked warning that you can
* Always use the @SuppressWarnings("unchecked") annotation on the smallest scope possible.

1. **Prefer lists to arrays**

* arrays are covariant: if Sub is a subtype of Super, then the array type Sub[] is a subtype of Super[]

// Fails at runtime!  
 Object[] objectArray = new Long[1];

objectArray[0] = "I don't fit in"; // Throws ArrayStoreException

// Won't compile!

List<Object> ol = new ArrayList<Long>(); // Incompatible types

* arrays are reified: arrays know and enforce their element types at runtime
* Generics are implemented by erasure: they enforce their type constraints only at compile time and discard (or erase) their element type information at runtime

// None are legal. generic array creation errors at compile time.

new List<E>[], new List<String>[], new E[]

* arrays provide runtime type safety but not compile-time type safety and vice versa for generics

1. **Favor generic types**
2. **Favor generic methods**

* The type parameter list, which declares the type parameter, goes between the method’s modifiers and its return type.

// Generic method

public static <E> Set<E> union(Set<E> s1, Set<E> s2) {

Set<E> result = new HashSet<E>(s1); result.addAll(s2); return result; }

1. **Use bounded wildcards to increase API flexibility**

* Iterable<String> is not a sub type of Iterable<Object>

// Wildcard type for parameter that serves as an E producer

public void pushAll(Iterable<? extends E> src) {

for (E e : src)  
 push(e);  
}

// Wildcard type for parameter that serves as an E consumer

public void popAll(Collection<? super E> dst) {

while (!isEmpty())  
 dst.add(pop());  
}

* Use wildcard types on input parameters that represent producers or consumers

PECS way: Stands for producer-extends, consumer-super

Comparable and Comparator are always consumers

Always use Comparable<? super T> in preference to Comparable<T>, Comparator<? super T> in preference to Comparator<T>.

* Do not use wildcard types as return types
* If the user of a class has to think about wildcard types, there is probably something wrong with the class’s API
* if a type parameter appears only once in a method declaration, replace it with a wildcard

// Two possible declarations for the swap method  
public static <E> void swap(List<E> list, int i, int j);  
public static void swap(List<?> list, int i, int j);

1. **Consider typesafe heterogeneous containers**

**Enums and Annotations**

1. **Use enums instead of int constants**

* Enums are far more readable, safer, and more powerful
* To associate data with enum constants, declare instance fields and write a constructor that takes the data and stores it in the fields

1. **Use instance fields instead of ordinals**

* ordinal(): 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

1. **Use EnumSet instead of bit fields**

* EnumSet: A set of enums

Usage: EnumSet.of(Style.BOLD, Style.ITALIC);

1. **Use EnumMap instead of ordinal indexing**
2. **Emulate extensible enums with interfaces**

public interface Operation { double apply(double x, double y); }

public enum BasicOperation implements Operation { PLUS; MINUS; MULTIPLY; DEVICE; }

public enum ExtendedOperation implements Operation { EXP; REMAINDER; }

1. **Prefer annotations to naming patterns**
2. **Consistently use the Override annotation**

* use the Override annotation on every method declaration that you believe to override a superclass declaration

1. **Use marker interfaces to define types**

* Marker interface is an interface that contains no method declarations, but merely designates (or “marks”) a class that implements the interface as having some property
* Use interface instead of marker annotation: Interfaces define a type that is implemented by instances of the marked class; marker annotations do not

Methods

1. **Check parameters for validity**
2. **Make defensive copies when needed**

* Essential to make a defensive copy of each mutable parameter to the constructor and to use the copies as components in place of the originals
* For thread-safety, defensive copies are made before validating the parameters, and the validation is performed on the copies rather than on the originals
* Do not use the clone() to make a defensive copy of a parameter whose type is sub-classable by untrusted parties.

Use constructor to create the copy

* Return defensive copies of mutable internal fields

clone can be used if the internal instance is created using its constructor so that the type is certain

* Always make a defensive copy of an internal array: array can be modified

1. **Design method signatures carefully**

* Choose method names carefully
* Don’t go overboard in providing convenience methods
* Avoid long parameter lists: Some techniques:
  + break the method up into multiple methods
  + create helper classes to hold groups of parameters
  + adapt the Builder pattern from object construction to method invocation
* For parameter types, favor interfaces over classes
* Prefer two-element enum types to boolean parameters

1. **Use overloading judiciously**

* The choice of which overloading to invoke is made at compile time

String classify(Set<?> s) { return "Set"; }

String classify(List<?> lst) { return "List"; }

String classify(Collection<?> c) { return "Unknown Collection"; }

Collection<?>[] collections = { new HashSet<String>(), new ArrayList<BigInteger>(), new HashMap<String, String>().values()};

// The compile-time type is Collection. Only String classify(Collection<?> c) will be invoked

for (Collection<?> c : collections)

System.out.println(classify(c));

* selection among overloaded methods is static, while selection among overridden methods is dynamic
* A safe, conservative policy is never to export two overloadings with the same number of parameters

1. **Use varargs judiciously**

* 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

1. **Return empty arrays or collections, not nulls**
2. **Write doc comments for all exposed API elements**

* Must precede every exported class, interface, constructor, method, and field declaration with a doc comment
* Method doc should describe succinctly the contract between the method and its client
  + Document what the method does rather than how it does its job, with the exception of methods designed for inheritance.
  + Check list: Preconditions, postconditions, checked and unchecked exceptions, params, returns, side effects, thread-safety
* 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 mem- bers

General Programming

1. **Minimize the scope of local variables**

* Declare it where it is first used
* Nearly every local variable declaration should contain an initializer
* Prefer for loops to while loops

for (Iterator<Element> i = c.iterator(); i.hasNext(); )

* Keep methods small and focused

1. **Prefer for-each loops to traditional for loops**

* for (Element e : elements)
* No performance penalty
* Iterate over any object that implements the Iterable interface

1. **Know and use the libraries**
2. **Avoid float and double if exact answers are required**

* 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

1.03 - .42 = 0.6100000000000001

* Use BigDecimal, int, or long for monetary calculations

1. **Prefer primitive types to boxed primitives**

* Applying the == operator to boxed primitives is almost always wrong.
* NullPointerException if no value and auto-unboxed

1. **Avoid strings where other types are more appropriate**
2. **Beware the performance of string concatenation**

* Achieve acceptable performance, use a StringBuilder in place of a String

1. **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

1. **Prefer interfaces to reflection**

* Objects should not be accessed reflectively in normal applications at runtime

1. **Use native methods judiciously**

* It is rarely advisable to use native methods for improved performance

1. **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

1. **Adhere to generally accepted naming conventions**

**Exceptions**

1. **Use exceptions only for exceptional conditions**

* 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

1. **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
* Don’t throw error, throw subclass of RuntimeException

1. **Avoid unnecessary use of checked exceptions**
2. **Favor the use of standard exceptions**
3. **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
* Try to avoid exceptions from lower layer by validating execution conditions

1. **Document all exceptions thrown by each method**

* Always declare checked exceptions individually, and document precisely the conditions under which each one is thrown
* 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

1. **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.”

1. **Strive for failure atomicity**

* Generally speaking, a failed method invocation should leave the object in the state that it was in prior to the invocation
  + design immutable objects
  + check parameters for validity before performing the operation
  + order the computation so that any part that may fail takes place before any part that modifies the object
  + write *recovery code* that intercepts a failure and rollback
  + 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

1. **Don’t ignore exceptions**

**Concurrency**

1. **Synchronize access to shared mutable data**

* Required for reliable communication between threads as well as for mutual exclusion

No effect unless both read and write operations are synchronized

* Volatile guarantees read and write on atomic type

// Not thread safe:

// ++ consists of two operations, read the value, plus one and write back

volatile int nextSerialNumber = 0;

nextSerialNumber++;

* confine mutable data to a single thread

1. **Avoid excessive synchronization**

* Never cede control to the client within a synchronized method or block

synchronized(lock) {

for (Listener listener : ListenerList)

listener.onAction(); // May have ConcurrentModificationException or deadlock, if the listener modifies the list

}

Do a snapshot and then broadcast or use CopyOnWriteArrayList

* As a rule, do as little work as possible inside synchronized regions

1. **Prefer executors and tasks to threads**
2. **Prefer concurrency utilities to wait and notify**

* Given the difficulty of using wait and notify correctly, should use the higher-level concurrency utilities instead
* For interval timing, always use System.nanoTime in preference to System.currentTimeMillis.
* Always use the wait loop idiom to invoke the wait method; never invoke it outside of a loop.

1. **Document thread safety**

* Level of thread safety:
  + immutable: Instances are constant
  + unconditionally thread-safe: Has sufficient internal synchronizations and the its instance can be used concurrently without the need for external synchronizations
  + conditionally thread-safe: Like above, except some methods require external synchronization
  + notthread-safe: Instances are mutable, need to access with synchronization
  + thread-hostile: Not safe for concurrent usage even if synchronized

1. **Use lazy initialization judiciously**

* Under most circumstances, normal initialization is preferable to lazy initialization
* If need to use for performance on a static field, use the lazy initialization holder class idiom

// Lazy initialization holder class idiom for static fields

private static class FieldHolder {

static final FieldType field = computeFieldValue();

}

static FieldType getField() { return FieldHolder.field; }

* If you need to use lazy initialization for performance on an instance field, use the double-check idiom.

// Double-check idiom for lazy initialization of instance fields

private volatile FieldType field;

FieldType getField() {

// The use of local variable of result is to prevent reading field many times

FieldType result = field;

if (result == null) { // First check (no locking)

synchronized(this) {

result = field;

if (result == null) // Second check (with locking)

field = result = computeFieldValue();

} }

return result;

}

1. **Don’t depend on the thread scheduler**

* Any program that relies on the thread scheduler for correctness or performance is likely to be nonportable
* Threads should not run if they aren’t doing useful work

1. **Avoid thread groups**

Serialization

1. Implement Serializable judiciously
2. Consider using a custom serialized form
3. Write readObject methods defensively
4. For instance control, prefer enum types to readResolve
5. Consider serialization proxies instead of serialized instances