Summary of the book Effective Java 2nd Edition by Joshua Bloch

README.md

This is my summary of the Effective Java 2nd Edition by Joshua Bloch. I use it while learning and as quick reference. It is not intended to be an standalone substitution of the book so if you really want to learn the concepts here presented, buy and read the book and use this repository as a reference and guide.

If you are the publisher and think this repository should not be public, just write me an email at hugomatilla [at] gmail [dot] com and I will make it private.

Contributions: Issues, comments and pull requests are super welcome 😃

1. TABLE OF CONTENTS

2. CREATING AND DESTROYING OBJECTS

1. Use STATIC FACTORY METHODS instead of constructors

ADVANTAGES

- Unlike constructors, they have names
- Unlike constructors, they are not requires to create a new object each time they're invoked
- Unlike constructors, they can return an object of any subtype of their return type
- They reduce verbosity of creating parameterized type instances

DISADVANTAGES

- If providing only static factory methods, classes without public or protected constructors cannot be subclassed (encourage to use composition instead inheritance).
- They are not readily distinguishable from other static methods (Some common names (each with a different pourpose) are: valueOf, of, getInstance, newInstance, getType and newType)

```
public static Boolean valueOf(boolean b){
        return b ? Boolean.TRUE :  Boolean.FALSE;
}
```

2. Use BUILDERS when faced with many constructors

Is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters.

Builder pattern simulates named optional parameters as in ADA and Python.

```
public class NutritionFacts {
        private final int servingSize;
        private final int servings;
        private final int calories;
        private final int fat;
        private final int sodium;
        private final int carbohydrate;

        public static class Builder {
                //Required parameters
                private final int servingSize:
                private final int servings;

                //Optional parameters - initialized to default values
                private int calories            = 0;
                private int fat                         = 0;
                private int carbohydrate        = 0;
                private int sodium                      = 0;

                public Builder (int servingSize, int servings) {
                        this.servingSize = servingSize;
                        this.servings = servings;
                }

                public Builder calories (int val) {
                        calories = val;
                        return this;
                }

                public Builder fat (int val) {
                        fat = val;
                        return this;
                }

                public Builder carbohydrate (int val) {
                        carbohydrate = val;
                        return this;
                }

                public Builder sodium (int val) {
                        sodium = val;
                        return this;
                }

                public NutritionFacts build(){
                        return new NutritionFacts(this);
                }
        }

        private NutritionFacts(Builder builder){
                servingSize             = builder.servingSize;
                servings                = builder.servings;
                calories                = builder.calories;
                fat                     = builder.fat;
                sodium                  = builder.sodium;
                carbohydrate            = builder.carbohydrate;
        }
}
```

Calling the builder

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240,8).calories(100).sodium(35).carbohydrate(27).build();
```

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

There are different ways to create singletons:

Public final field

```
public class Elvis{
        public static final Elvis INSTANCE = new Elvis();
        private Elvis(){...}
        ...
        public void singASong(){...}
}
```

One problem is that a privileged client can invoke the private constructor reflectively. Against this attack the constructor needs to be modified to send an exception if it is asked to create a second instance.

Singleton with static factory

```
public class Elvis{
        private static final Elvis INSTANCE = new Elvis();
        private Elvis(){...}
        public static Elvis getInstance(){ return INSTANCE; }
        ...
        public void singASong(){...}
}
```

In this approach it can be change to a non singleton class without changing the class API.

Serialize a singleton

It is needed a *readResolve* method and declare all the fields *transient* in addition to the *implements Serializable* to maintain the singleton guarantee.

```
private Object readResolve(){
        //Return the one true Elvis and let the garbage collector take care of the Elvis impersonator
        return INSTANCE;
}
```

Enum Singleton, the preferred approach (JAVA 1.5)

```
public enum Elvis(){
        INSTANCE;
        ...
        public void singASong(){...}
}
```

Equivalent to the public field, more concise, provides serialization machinery for free, and guarantee against multiple instantiation, even for reflection attacks and sophisticated serialization. *It is the best way to implement a singleton*.

4. Enforce noninstantiability with a private constructor

For classes that group static methods and static fields.

Used for example to:

- Group related methods on primitive values or arrays.
- Group static methods, including factory methods, for objects that implement a particular interface.
- Group methods on a final class instead of extending the class.

Include a private constructor

```
public class UtilityClass{
        // Suppress default constructor for noninstantiability
        // (Add comment to clarify why the constructor is expressly provided)
        private UtilityClass(){
                throw new AssertionError();
        }
        ...
}
```

5. Avoid creating objects

REUSE IMMUTABLE OBJECTS

Don't do this

```
String s = new String("stringette");
```

Every call creates a new String instance. The argument "*stringette*" is itself one. This call in a loop would create many of them.

Do this

```
String s = "stringette";
```

This one uses a single String instance rather than creating a new one.

Use static factory methods in preference to constructors Item 1

Boolean.valueOf(String); Is preferable to the constructor *Boolean(String)*.

REUSE MUTABLE OBJECTS THAT WON'T BE MODIFIED

Don't do this

```
public class Person {
private final Date birthDate;
...
        public boolean isBabyBoomer(){
                // Unnecessary allocation of expensive object.
                Calendar gmtCal= Calendar.getInstance(TimeZone.getTimeZone("GMT"));
                gmtCal.set(1946,Calendar.JANUARY,1,0,0,0);
                Date boomStart = gmtCal.getTime();
                gmtCal.set(1965,Calendar.JANUARY,1,0,0,0);
                Date boomEnd = gmtCal.getTime();
                return birthDate.compareTo(boomStart) >= 0 &&birthDate.compareTo(boomEnd)<0;
        }
}
```

isBabyBoomer creates a new Calendar,TimeZone and two Date instances each time is invoked.

```
public class Person {
private final Date birthDate;
...
private static final Date BOOM_START;
private static final Date BOOM_END;

static {
                Calendar gmtCal= Calendar.getInstance(TimeZone.getTimeZone("GMT"));
                gmtCal.set(1946,Calendar.JANUARY,1,0,0,0);
                BOOM_START = gmtCal.getTime();
```

```
                gmtCal.set(1965,Calendar.JANUARY,1,0,0,0);
                BOOM_END = gmtCal.getTime();
        }
        public boolean isBabyBoomer(){
                return birthDate.compareTo(BOOM_START) >= 0 &&birthDate.compareTo(BOOM_END)<0;
        }
}
```

Prefer primitives to boxed primitives, and watch out for unintentional autoboxing

```
//Slow program. Where is the object creation?
public static void main (String[] args){
        Long sum = 0L;
        for (long i = 0 ; i<= Integer.MAX_VALUE; i++){
                sum+=i;
        }
        System.out.println(sum);
}
```

sum is declared as *Long* instead of *long* that means that the programs constructs Long instances.

Object polls are normally bad ideas

Unless objects in the pool are extremely heavyweight, like a database connections.

6. Eliminate obsolete object references

Can you spot the memory leak?

```
public class Stack{
        private Object[] elements;
        private int size = 0;
        private static final int DEFAULT_INITIAL_CAPACITY = 16;

        public Stack(){
                elements = new Object [DEFAULT_INITIAL_CAPACITY];
        }

        public void push(Object e){
                ensureCapacity();
                elements[size++] = e;
        }

        public Object pop(){
                if (size == 0)
                        throw new EmptyStackException();
                return elements[--size];
        }

        private void ensureCapacity(){
                if(elements.length == size)
                        elements = Array.copyOf(elements, 2 * size + 1);
        }
}
```

If the stack grows and shrinks the objects popped will not be garbage collected. The stack maintains obsolete references (a reference that will never be dereferenced).

Null out references

```
public pop(){
        if (size == 0)
```

```
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete references.
        return result;
    }
```

Nulling out objects references should be the exception not the norm. Do not overcompensate by nulling out every object.

Null out objects only in classes that manages its own memory.

Memory leaks in cache

Using *WeakHashMap* is useful when if desired lifetime of cache entries is determined by external references to the key, not the value.

Clean oldest entries in cache is a common practice. To accomplish this behaviors, it can be used: background threads, automatically delete older after a new insertion or the *LinkedHashMap* and its method *removeEldestEntry*.

Memory leaks in listeners and callbacks

If clients register callbacks, but never deregister them explicitly.

To solve it store only *weak references* to them, for example storing them as keys in a *WeakHashMap*.

Use a Heap Profiler from time to time to find unseen memory leaks

7. Avoid finalizers

Finalizers are unpredictable, often dangerous and generally.

Never do anything time-critical in a finalizer.

> There is no guarantee they'll be executed promptly.

Never depend on a finalizer to update critical persistent state.

> There is no guarantee they'll be executed at all.

Uncaught exceptions inside a finalizer won't even print a warning.

There is a severe performance penalty for using finalizers.

Possible Solution

Provide an *explicit termination method* like the *close* on *InputStream*, *OutputStream*, *java.sql.Connection*...

Explicit termination methods are typically used in combination with the *try-finally* construct to ensure termination.

```
Foo foo = new Foo(...);
try {
    // Do what must be done with foo
    ...
} finally {
    foo.terminate();  // Explicit termination method
}
```

They are good for this two cases

- As a safety net. Ask yourself if the extra protection is worth the extra cost.
- Use in native peers. Garbage collector doesn't know about this objects.

In this cases always remember to invoke super.finalize.

3. METHODS COMMON TO ALL OBJECTS

8. Obey the general contract when overriding *equals*

Don't override if:

- Each instance of the class is inherently unique. I.e.*Thread*
- You don't care whether the class provides a "logical equality" test. I.e. *java.util.Random*
- A superclass has already overridden *equals*, and the superclass behavior is appropriate for this class I.e. *Set*
- The class is private or package-private, and you are certain that its *equals* method will never be invoked

Override if:

A class has a notion of *logical equality* that differs from mere object identity, and a superclass has not already overridden *equals* to implement the desired behavior.

Equals implements an "equivalence relation"

- Reflexive: *x.equals(x)==true*
- Symmetric: *x.equals(y)==y.equals(x)*
- Transitive: *x.equals(y)==y.equals(z)==z.equals(x)*
- Consistent: *x.equals(y)==x.equals(y)==x.equals(y)==...*
- Non-nullity: *x.equals(null)->false*

The Recipe

1. Use the == operator to check if the argument is a reference to this object (for performance)
2. Use the *instanceof* operator to check if the argument has the correct type
3. Cast the argument to the correct type
4. For each "significant" field in the class, check if that field of the argument matches the corresponding field of this object
5. When you are finished writing your *equals* method, ask yourself three questions: Is it Symmetric? Is it Transitive? Is it Consistent? (the other 2 usually take care of themselves)

```
@Override
public boolean equals (Object o){
        if(o == this)
                return true;

        if (!(o instanceof PhoneNumber))
                return false;

        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNumber == lineNumber
                && pn.prefix == prefix
                && pn.areaCode == areaCode;
}
```

Never Forget

- Always override *hashCode* when you override *equals*
- Don't try to be too clever (simplicity is your friend)
- Don't substitute another type for *Object* in the *equals* declaration

9. Always override ***hashCode*** when you override ***equals***

Contract of hashCode

- Whenever *hashCode* is invoked in the same object it should return the same integer.
- If two objects are equals according to the *equals*, the should return the same integer calling *hashCode*.
- Is not required (but recommended) that two non *equals* objects return distinct *hashCode*.

The second condition is the one that is more often violated.

The Recipe

1. Store constant value i.e. 17 in and integer called *result*.
2. For each field *f* used in *equals* do:

- Compute *c*
  - boolean: *(f ? 1 : 0)*
  - byte, char, short or int: *(int) f*
  - long: *(int) (f ^ (.f >>> 32))*
  - float: *Float.floatToIntBits(f)*
  - double: *Double.doubleToLongBits(f)* and compute as a long
  - object reference: if *equals* of the reference use recutsivity, use recursivity for the *hashCode*
  - array: each element as a separate field.
- Combine: *result = 31 * result + c*

3. Return *result*
4. Ask yourself if equal instances have equal hash codes.

```
private volatile int hashCode; // Item 71 (Lazily initialized, cached hashCode)

@Override public int hashCode(){
        int result = hashCode;
        if (result == 0){
                result = 17;
                result = 31 * result + areaCode;
                result = 31 * result + prefix;
                result = 31 * result + lineNumber;
                hashCode = result;
        }
        return result;
}
```

10. Always override *toString*

Providing a good *toString* implementation makes your class much more pleasant to read.

When practical, the *toString* method return all of the interesting information contained in the object.

It is possible to specify the format of return value in the documentation.

Always provide programmatic access to all of the information contained in the value returned by *toString* so the users of the object don't need to parse the output of the *toString*

11. Override *clone* judiciously

Cloneable interface does not contain methods If a class implements Cloneable, Object's clone method returns a field-by-field copy of the object. Otherwise it throws CloneNotSupportedException.

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.

Simple clone method if object does **not** contain fields that refer to mutable objects.

```
@Override public PhoneNumber clone() {
        try {
                //PhoneNumber.clone must cast the result of super.clone() before returning it.
                return (PhoneNumber) super.clone();
        } catch(CloneNotSupportedException e) {
                throw new AssertionError(); // Can't happen
        }
}
```

If object **contains** fields that refer to mutable objects, we need another solution. Mutable fields will point to same objects in memory and the original and the cloned method will share these objects.

clone is another constructor and therefore it must ensure not harming the original object and establishing invariants.
Calling *clone* recursively in the mutable objects is the easiest way.

```
@Override public Stack clone() {
        try {
                Stack result = (Stack) super.clone();
                // From Java 1.5, don't need casting when cloning arrays
                result.elements = elements.clone();
                return result;
        } catch (CloneNotSupportedException e) {
                throw new AssertionError();
        }
}
```

Mutable objects and finals: The *clone* architecture is incompatible with normal use of final fields referring to mutable objects. More complex objects would need specific approaches where recursively calling *clone* won't work.

A *clone* method should not invoke any nonfinal methods on the clone under construction (Item 17).

Object's *clone* method is declared to throw *CloneNotSupportedException*, but overriding clone methods can omit this declaration.
Public *clone* methods should omit it. (Item 59).
If a class overrides clone, the overriding method should mimic the behavior of *Object.clone*:

- it should be declared protected,
- it should be declared to throw CloneNotSupportedException,
- it should not implement Cloneable.

Subclasses are free to implement Cloneable or not, just as if they extended Object directly

clone method must be properly synchronized just like any other method (Item 66).

Summary: classes that implement Cloneable should create a method that:

- override clone
- return type is the class
- call *super.clone*
- fix fields that need to be fixed

Better provide an alternative of object copying, or don't provide it at all.

Copy Constructor

```
public Yum(Yum yum);
```

Copy Factory

```
public static Yum newInstance(Yum yum);
```

These alternatives:

- don't rely on a risk-prone extra-linguistic object creation mechanism
- don't demand adherence to thinly documented conventions
- don't conflict with the proper use of final fields
- don't throw unnecessary checked exceptions
- don't require casts.

Furthermore they can use its Interface-based copy constructors and factories, *conversion constructors* and *conversion factories* and allow clients to choose the implementation type `public HashSet(Set set) -> TreeSet;`

12. Consider implementing ***Comparable***

Comparable is an interface. It is not declared in *Object*

Sorting an array of objects that implement *Comparable* is as simple as `Arrays.sort(a);`

The class will interoperate with many generic algorithms and collection implementations that depend on this interface. You gain lot of power with small effort.

Follow this provisions (Reflexive, Transitive, Symmetric):

1. `if a > b then b < a  if a == b then b == a  if a < b then b > a`
2. `if a > b and b > c then a > c`
3. `if a == b and b == c then a == c`
4. Strong suggestion: `a.equals(b) == a.compareTo(b)`

For integral primitives use `<` and `>` operators.

For floating-point fields use *Float.compare* or *Double.compare*

For arrays start with the most significant field and work your way down.

4. CLASSES AND INTERFACES

13. Minimize the accesibility of classes and members

Encapsulation:

- A well designed module hides all of its implementation details.
- Separates its API from its implementation.
- Decouples modules that comprise a system, allowing them to be isolated while:
  - developed (can be developed in parallel)
  - tested (individual modules may prove succesful even if the system does not)
  - optimized and modified (no harm to other modules)
  - understood (dont need other modules to be understood)
  - used

Make each class or member as inaccesible as possible

If a package-private top level class is used by only one class make it a private nested class of the class that uses it. (Item 22)

Is is acceptable to make a private member of a public class package-private in order to test it.

Instance fields should never be public (Item 14) Class will not be thread-safe.

Static fields can be public if contain primitive values or references to immutable objects. A final field containing a reference to a mutable object has all the disadvantages of a non final field.

Nonzero-length array is always mutable.

```
//Potential security hole!
public static final Thing[] VALUES = {...}
```

Solution:

```
private static final Thing[] PRIVATE_VALUES ={...}
public static final List<Thing> VALUES = Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

Or:

```
private static final Thing[] PRIVATE_VALUES ={...}
public static final Thing[] values(){
        return PRIVATE_VALUES.clone;
}
```

14. In public classes, use accessor methods, not public fields

Degenerate classes should not be public

```
class Point {
        public double x;
        public double y;
}
```

- The don't benefit from *encapsulation* (Item 13)
- Can't change representation without changing the API.
- Can't enforce invariants.
- Can't take auxiliary actions when a field is accessed.

Replace them with *accessor methods* (getters) and *mutators* (setters).

```
class Point {
        private double x;
        private double y;

        public Point(double x, double y){
                this.x = x;
                this.y = y;
        }

        public double getX() { return x; }
        public double getY() { return y; }

        public void setX(double x) { this.x = x; }
        public void setY(double y) { this.y = y; }
}
```

If a class is accessed **outside its package**, provide **accesor methods**.

If a class is **package-private or is a private nested class**, its **ok to expose** its data fields.

In **public classes** it is a questionable option to **expose immutable fields**.

15. Minimize Mutability

All the information of the instance is provided when it is created. They are easier to design, implement and use. And they are less prone to errors and more secure

- Don't provide any methods that modify the object's state (no mutators)
- Ensure that the class can't be extended
- Make all fields final
- Make all fields private
- Ensure exclusive access to any mutable component

```
public final class Complex {
        private final double re;
        private final double im;

        public Complex (double re, double im) {
                this.re = re;
                this.im = im;
        }

        // Accessors with no corresponding mutators
        public double realPart() { return re;}
        public double imaginaryPart() { return im;}

        public Complex add(Complex c){
                return new Complex(re + c.re, im + c.im);
        }

        public Complex subtract(Complex c){
                return new Complex(re - c.re, im - c.im);
        }

        ...

        @Override public boolean equals (Object o){...}
}
```

The arithmetic operation **create and return a new instance**. (Functional approach)

Immutable objects are simple. They only have one state for its lifetime.

Immutable objects are thread-safe. Synchronization is not required. They can be shared freely and can reuse existing instances.

```
public static final Complex ZERO = new Complex(0,0)
public static final Complex ONE = new Complex(1,0)
public static final Complex I = new Complex(0,1)
```

Using static factories can create constants of frequently requested instances and serve them in future requests.

Internals of the immutable objects can also be share.

They make great building blocks for other objects.

The disadvantages is that require a separate object for distinct values. In some cases it could reach to a performance problem.

How to deny subclassing in immutable objects

1. Making it final

2. Make all of its constructors private or package-private and add a public static factory

```
public class Complex {
        private final double re;
        private final double im;

        private Complex (double re, double im){
                this.re = re;
                this.im = im;
        }

        public static Complex valueOf(double re, double im){
                return new Complex(re,im);
        }

        ...
}
```

This technique allows flexibility of multiple implementations, it's possible to tune the performance and permit to create more factories with names that clarify its function.

Summary

Classes should be immutable unless there are good reasons to make them mutable.

If a class can not be immutable, limit its mutability as much as possible.

Make every field final unles there is a good reason not to do it.

Some of the rules can be lightened to improve performance (caching, lazy initialization...).

16. Favor composition over inheritance

Inheritance in this case is when a class extends another (*implementation inheritance*) Not interface inheritance.

Inheritance violates encapsulation

Fragility causes

1. A subclass depends on the implementation details of its superclass. If the superclass change the subclass may break.
2. The superclass can aquire new methods in new releases that might not be added in the subclass.

Composition

Instead of extending, give your new class a private field that references an instance of the existing class.

Each instance method in the new class (*forwarding class*)invokes the corresponding method (*forwarding methods*) on the contained instance of the existing class and returns the results.

Wrapper (Decorator Pattern)

```
// Wrapper class - uses composition in place of inheritance
public class InstrumentedSet<E> extends ForwardingSet<E> {
        private int addCount = 0;
        //It extends a class(inheritance),but it is a forwarding class that is actually a compositon of the Se
        (specifically a forwarding class), not the Set itself.
        public InstrumentedSet (Set<E> s){
                super(s)
        }

        @Override
        public boolean add(E e){
                addCount++;
```

```
                return super.add(e);
        }

        @Override
        public boolean addAll (Collection< ? extends E> c){
                addCount += c.size();
                return super.addAll(c);
        }

        public int getAddCount() {
                return addCount;
        }
}
```

```
// Reusable forwarding class
public class ForwardingSet<E> implements Set<E> {
        private final Set<E> s; // Here is the composition. It uses the Set but not extends it.
        public ForwardingSet(Set<E> s) { this.s = s ; }

        // It implemets the Set, using the interface, and create the forwarding methods.
        public void clear() {s.clear();}
        public boolean contains(Object o) { return s.contains(o)}
        ...
        public boolean add(E e) { return s.add(e)}
        public boolean addAll (Collection< ? extends E> c){return s.addAll(c)}
        ...
}
```

17. Design and document for inheritance or else prohibit it.

The class must document it *self-use* of overridable methods.

Methods and constructors should document which *overridable* methods or constructors (nonfinal, and public or protected) invokes. The description begins with the phrase "This implementation."

To document a class so that it can be safely subclassed, you must describe implementations details.

To allow programmers to write efficient subclasses without undue pain, a class may have to provide hooks into its internal working in the form of judiciously chosen protected methods.

Test the class for subclassing. The only way to test a class designed for inheritance is to write subclasses.

Constructors must not invoke overridable methods. For *Serializable* and *Cloneable* implementations neither *clone* nor *readObject* may invoke overridable methods.

Prohibit subclassing in classes that are not designed and documented to be safely subclassed. 2 options:

- Declare the class final
- Make all constructors private or package-private and add public static factories in place of the constructors. (Item 15)

Consider use Item 16 if what you want is to increase the functionality of your class instead of subclassing.

18. Prefer interfaces to abstract classes

Java permits only single Inheritance, this restriction on abstract classes severely contrains their use as type functions.

Intefaces is generally the best way to define a type that permits multiple implementations.

Existing classes can be easily retrofitted to implement a new interface.

Interfaces are ideal for defining mixins (a type that a class can implement in addition to its primary type to declare that it provides some optional bahaviour)

Interfaces allow the construction of nonhierarchical type frameworks.

Interfaces enable safe, powerful functionality enhancements (Wrapper class. Item 16)

Combine the virtues of interfaces and abstract classes, by providing an abstract **skeletal implementation** class to go with each **nontrivial interface** that you export.

```
//Concrete implementation built atop skeletal implementation
static List<Integer> intArrayAsList(final int[] a) {
        if (a == null) throw new NullPointerException();


        // From the documentation
        //This class provides a skeletal implementation of the List interface to minimize the effort required
        //To implement an unmodifiable list, the programmer needs only to extend this class and provide implem
        //To implement a modifiable list, the programmer must additionally override the set(int, E)

        return new AbstractList<Integer>(){
                public Integer get (int i){
                        return a[i]; // Autoboxing (Item 5)
                }

                @Override
                public Integer set(int i, Integer val){
                        int oldVal = a[i];
                        a[i] = val;             // Auto-unboxing
                        return oldVal;  // Autoboxing
                }

                public int size(){
                        return a.length;
                }
        }
}
```

Skeletal implementations are designed for inheritance so follow Item 17 guidelines.

simple implementation is like a skeletal implementation in that it implements the simplest possible working implementation.

Cons: 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.

19. Use interfaces only to define types

When a class implements an interface, the interface serves as a *type* that can be used to refer to instances of the class.

Any other use, like the *constant interface* should be avoided.

```
// Constant interface antipattern
public interface PhysicalConstants {
        static final double AVOGRADOS_NUMBER = 6.02214199e23;
        static final double BOLTZAN_CONSTANT = 1.3806503e-23;
        static final double ELECTRON_MASS = 9.10938188e-31;
}
```

Better use an enum type (Item 31), or a noninstantiable *utility class* (Item 4)

```
//Constant utility class
package com.effectivejava.science
```

```
public class PhysicalConstants{
        private PhysicalConstants(){} // Prevents instantiation

        public static final double AVOGRADOS_NUMBER = 6.02214199e23;
        public static final double BOLTZAN_CONSTANT = 1.3806503e-23;
        public static final double ELECTRON_MASS = 9.10938188e-31;
}
```

To avoid the need of qualifying use *static import*.

```
//Use of static import to avoid qualifying constants
import static com.effectivejava.science.PhysicalConstants.*

public class Test {
        double atoms(double mols){
                return AVOGRADOS_NUMBER * mols;
        }
        ...
        // Many more uses of PhysicalConstants justify the static import
}
```

20. Prefer class hierarchies to tagged classes

Tagged classes are verbose, error-prone and inefficient.

They have lot of boilerplate, bad readability, they increase memory footprint, and more shortcommings.

```
// Tagged Class
class Figure{
        enum Shaple {RECTANGLE, CIRCLE};

        final Shape shape;

        // Rectangle fields
        double length;
        double width;

        //Circle field
        double radius;

        // Circle Constructor
        Figure (double radius) {
                shape = Shape.CIRCLE;
                this.radius=radius;
        }

        // Rectangle Constructor
        Figure (double length, double width) {
                shape = Shape.RECTANGLE;
                this.length=length;
                this.width=width;
        }

        double area(){
                switch(shape){
                        case RECTANGLE:
                                return length*width;
                        case CIRCLE
                                return Math.PI * (radius * radius);
                        defalult
                                throw new AssertionError();
                }
        }
}
```

A tagged class is just a palid imitation of a class hierarchy.

- The code is simple and clear.
- The specific implementations are in its own class
- All fields are final
- The compiler ensures that each class's constructor initializes its data fields.
- Extendability and flexibility (Square extends Rectangle)

```
abstract class Figure{
        abstract double area();
}
class Circle extends Figure{
        final double radius;

        Circle(double radius) { this.radius=radius;}

        double area(){return Math.PI * (radius * radius);}
}
class Rectangle extends Figure{
        final double length;
        final double width;

        Rectangle (double length, double width) {
                this.length=length;
                this.width=width;
        }

        double area(){return length*width;}
}

class Square extends Rectangle {
        Square(double side){
                super(side,side);
        }
}
```

21. Use function objects to represent strategies

Strategies are facilities that allow programs to store and transmit the ability to invoke a particular function. Similar to *function pointers*, *delegates* or *lambda expression*.

It is possible to define a object whose method perform operations on other objects.

Concrete strategy

```
class StringLengthComparator{
        public int compare(String s1, String s2){
                return s1.length() - s2.length();
        }
}
```

Concrete strategies are typically *stateless* threfore they should be singletons.

To be able to pass different strategies, clients should invoke methods from an *strategy interface* instead of from a concrete class.

Comparator interface. *Generic*(Item 26)

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

```
class StringLengthComparator implements Comparator<String>{
        private StringLengthComparator(){} // Private constructor
        public static final StringLengthComparator INSTANCE = new StringLengthComparator(); // Singleton insta
        public int compare(String s1, String s2){
                return s1.length() - s2.length();
        }
}
```

Using **anonymous classes**

```
Arrays.sort(stringArray, new Comparator<String>(){
        public int compare(String s1, String s2){
                return s1.length() - s2.length();
        }
})
```

An anonymous class will create a new instance each time the call is executed. Consider a private static final field and reusing it.

Concrete strategy class don't need to be public, because the strategy interface serve as a type. A host class can export the a public static field or factory, whose type is the interface and the concrete strategy class is a private nested class.

```
// Exporting a concrete strategy
class Host{
        private static class StringLengthComparator implements Comparator<String>, Serializable {
                public int compare(String s1, String s2){
                        return s1.length() - s2.length();
                }
        }

        //Returned comparator is serializable
        public static final Comparator<String> STRING_LEGTH_COMPARATOR = new StringLengthComparator();

        ...
}
```

22. Favor static member classes over nonstatic

4 types of nested classes.

1. static
2. nonstatic
3. anonymous
4. local

Static, a member class that does not require access to an enclosing instance must be *static*.

Storing references cost time, space and can cost not wanted behaviors of the garbage collector(Item 6)

Common use of static member class is a public helper in conjuctions with its outer class. A nested class enum *Operation* in *Calculator* class. `Calculator.Operation.PLUS` ;

Nonstatic member class instances are required to have an enclosing instance.

Anonymous classes are us to create *function objects* on the fly. (Item 21)

Local class from the official docs: Use it if you need to create more than one instance of a class, access its constructor, or introduce a new, named type (because, for example, you need to invoke additional methods later).

Anonymous class, from the official docs: Use it if you need to declare fields or additional methods.

5. GENERICS

23. Don't use raw types in new code

Generic classes and interfaces are the ones who have one or more *type parameter* as *generic*, i.e. `List<E>`

Each generic type defines a set of *parametrized types* `List<String>`

Raw types is the generic type definition without type parameters. `List`

```
private final Collection stamps = ...

stamps.add(new Coin(...)); //Erroneous insertion. Does not throw any error

Stamp s = (Stamp) stamps.get(i); // Throws ClassCastException when getting the Coin


private final Collection<Stamp> stamps = ...

stamps.add(new Coin()); // Compile time error. Coin can not be add to Collection<Stamp>

Stamp s = stamps.get(i); // No need casting
```

Use of raw types lose safety and expressiveness of generics.

Type safety is kept in a parametrized type like `List<Object>` but not in raw types (`List`).

There are subtyping rules for generics. For example `List<String>` is a subtype of `List` but not of `List<Object>` (Item 25)

Unbounded Wildcard Types `Set<?>` Used when a generic type is needed but we don't know or care the actual type.

Never add elements (other than null) into a `Collection<?>`

2 exceptions (because generic type information is erased at runtime):

- Use raw types in class literals `List.class` , `String[].class` are legal, `List<String>.class` , `List<?>.class` are not.
- Use of instanceof

```
if (o instanceof Set){
        Set<?> = (Set<?>) o;
}
```

Term	Example	Item
Parametrized type	`List<String>`	23
Actual type parameter	`String`	23
Generic type	`List<E>`	23, 26
Formal type parameter	`E`	23
Unbounded wildcard type	`List<?>`	23
Raw type	`List`	23
Bounded type parameter	`<E extends Number>`	26
Recursive type bound	`<T extends Comparable<T>>`	27
Bounded wildcard type	`List<? extends Number>`	28

Term	Example	Item
Generic method	`static <E> List<E> asList(E[] a)`	27
Type token	`String.class`	29

24. Eliminate unchecked warnings

Eliminate every unchecked warning that you can, if you can´t use *Suppress-Warnings* annotation on the smallest scope possible.

```
Set<Lark> exaltation = new HashSet(); Warning, unchecked convertion found.
Set<Lark> exaltation = new HashSet<Lark>(); Good
```

25. Prefer lists to arrays

Arrays are *covariant*: if `Sub` is a subtype of `Super`, `Sub[]` is a subtype of `Super[]`
Generics are *invariant*: for any two types `Type1` and `Type2`, `List<Type1>` in neither sub or super type of `List<Type1>`

```
// 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
ol.add("I don't fit in")
```

Arrays are *reified*: Arrays know and enforce their element types at runtime. Generics are *erasure*: Enforce their type constrains only at compile time and discard (or *erase*) their element type information at runtime.

Therefore it is illegal to create an array of a generic type, a parameterized type or a type parameter.

`new List<E>[]`, `new List<String>`, `new E[]` will result in *generic array creation* errors.

26. Favor generic types

Making Item 6 to use generics.

```
public class Stack{
        private E[] elements;
        private int size = 0;
        private static final int DEFAULT_INITIAL_CAPACITY = 16;

        public Stack(){
                elements = new E [DEFAULT_INITIAL_CAPACITY];//Error: Can't create an array of a non-reifiable
        }

        public void push(E e){
                ensureCapacity();
                elements[size++] = e;
        }

        public E pop(){
                if (size == 0)
                        throw new EmptyStackException();
                E result = elements[--size];
                elements[size] = null;
                return result;
        }
        ...
}
```

There will be one error:

```
//Error: Generic array creation. Can't create an array of a non-reifiable type.
elements = new E [DEFAULT_INITIAL_CAPACITY];
```

First option (more commonly used.)

```
//Warning: Compiler can not prove the type safe, but we can.
// This elements array will contain only E instances from push(E).
// This is sifficient to ensure type safety, but the runtime
//type of the array won't be E[]; it will always be Object[]!
@SupressWarnings("unchecked")
public Stack(){
        elements = (E[]) new Object [DEFAULT_INITIAL_CAPACITY];
}
```

Second Option

```
...
private Object[] elements;
...
result = elements[--size] // Error: found Object, required E
```

A cast will generate a warning. Beacuse E is a non-reifiable type, there is no way the compiler can check the cast at runtime.

```
result = (E) elements[--size]
```

The appropriate suppression of the unchecked warning

```
public E pop(){
        if (size == 0)
                throw new EmptyStackException();

        // push requires elements to be of type E, so cast is correct.
        @SupressWarnings("unchecked") E result = elements[--size];

        elements[size] = null;
        return result;
}
```

27. Favor generic Methods

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;
}
```

Type inference: Compiler knows because of `Set<String>` that `E` is a *String*

In generic constructors the type parameters have to be on both sides of the declaration. (Java 1.7 might have fix it)

```
Map<String,List<String>> anagrams = new HashMap<String, List<String>>();
```

To avoid ic create a *generic static factory method*

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

//Use
Map<String,List<String>> anagrams = newHashMap();
```

Generic Singleton Pattern Create an object that is immutable but applicable to many different types.

```
public interface UnaryFunction<T>{
        T apply(T arg);
}

// Generic singleton factory pattern
private static UnaryFunction<Object> IDENTITY_FUNCTION = new UnaryFunction<Object>(){
        public Object apply(Object arg) {return arg;}
};

//IDENTITY_FUNCTION is stateless and its type parameter is unbounded so it's safe to share one instance across
@SuppressWarnings("unchecked")
public static<T> UnaryFunction<T> identityFunction(){
        return(UnaryFunction<T>) IDENTITY_FUNCTION;
}
```

Recursive Type Bound : When the type paremeter is bounded by some expression involving that type parameter itself.

```
public static<T extends Comparable<T>> T max (List<T> list){
        Iterator <T> i = list.iterator():
        T result = i.next();
        while (i.hasNext()) {
                T t = i.next();
                if(t.compareTo(result) > 0)
                        result = t;
        }
        return result;
}
```

28. Use bounded wildcards to increase API flexibility

Parameterized types are invariant.(Item 25) Ie `List<String>` is not a subtype of `List<Object>`

```
public void pushAll(Iterable<E> src){
        for(E e : src)
                puhs(e)
}

// Integer is a subtype of Number
Stack<Number> numberStack = new Stack<Number>();
Iterable<Integer> integers = ...
numberStack.pushAll(integers); //Error message here: List<Integer> is not a subtype of List<Number>
```

Bounded wildcard type

Producer

```
public void pushAll(Iterable<? Extends E> src){
        for (E e : src)
                push(e);
}
```

Consumer

```
public void popAll(Collection<? super E> dst){
        while(!isEmpty())
                dst.add(pop());
}
```

PECS: producer-extends, consumer-super

If the parameter is a producer and a conusmer don't use *wildcards*

Never use *wildcards* in return values.

Type inference in generics

```
Set<Integer> integers =...
Set<Double> doubles =...
Set<Number> numbers = union(integers,doubles);//Error

//Needs a 'explicit type parameter'
Set<Number> numbers = Union.<Number>union(integers,doubles);
```

Comparable and Comparators are always consumers. Use `Comparable<? super T>` and `Comparator<? super T>`

If a type parameter appears only once in a method declaration, replace it with a wildcard.

29. Consider *typesafe heterogeneous containers*

A container for accessing a heterogeneous list of types in a typesafe way.

Thanks to the type of the class literal. `Class<T>`

API

```
public class Favorites{
        public void putFavorites(Class<T> type, T instance);
        public <T> getFavorite(Class<T> type);
}
```

Client

```
Favorites f = new Favorites();
f.putFavorites(String.class, "JAVA");
f.putFavorites(Integer.class, 0xcafecace);
f.putFavorites(Class.class, Favorite.class);

String s = f.getFavorites(String.class);
int i =f.getFavorites(Integer.class);
Class<?> c = f.getFavorites(Class.class);
```

Implementation

```
public class Favorites{
        private Map<Class<?>, Object> favorites = new HashMap<Class<?>, Object>();
```

```
        public <T> void putFavorites(Class<T> type, T instance){
                if(type == null)
                        throw new NullPointerException("Type is null");
                favorites.put(type, type.cast(instance));//runtime safety with a dynamic cast
        }

        public <T> getFavorite(Class<T> type){
                return type.cast(favorites.get(type));
        }
}
```

6. ENUMS AND ANNOTATIONS

30. Use enums instead of *int* constants

Enums are classes that export one instance for each enumeration constant via a public static final field. Clients can not create instances or extend them. They are a generalization of singletons(Item 3) They are compile-time type safe.

Enums can have data associated

```
public enum Planet{
        MERCURY(3.334e+23,2.234e6)
        VENUS(4.234e+23,6.636e6)
        EARTH(5.865e+23,6.256e6)
        ...

        private final double mass;
        private final double radius;
        private final double surfaceGravity;

        private static final double G = 6.67300E-11;

        Planet(double mass, double radius){
                this.mass = mass;
                this.radius = radius;
                surfaceGravity = G * mass/(radius * radius);
        }

        public double mass() {return mass;}
        public double radius() {return radius;}
        public double surfaceGravity() {return surfaceGravity;}

        public double surfaceWeight(double mass){
                return mass * surfaceGravity;
        }
}
```

Enums are immutable so their fields should be final(Item 15) Make fields private (Item 14)

Enums should be a member class inside a top-level class if it is not generally used.

Enum type with constant-specific method implementations

```
public enum Operation{
        PLUS { double apply(double x, double y){return x + y;}},
        MINUS { double apply(double x, double y){return x - y;}},
        TIMES { double apply(double x, double y){return x * y;}},
        DIVIDE { double apply(double x, double y){return x / y;}};

        // The abstract method force us not to forget to implement the method.
        abstract double apply(double x, double y);
```

```
}
```

Strategy enum pattern Use it, if multiple enum constants share common behaviors.

```
enum PayrollDay{
        MONDAY(PayType.WEEKDAY),
        TUESDAY(PayType.WEEKDAY),
        ...
        SATURDAY(PayType.WEEKEND),
        SUNDAY(PayType.WEEKEND);

        private final PayType payType;

        PayrollDay(PayType payType) {this.payType = payType;}

        double pay(double hoursWorked, double payRate){
                return payType.pay(hoursWorked, payRate);
        }
        //The strategy  enum type
        private enum PayType{
                WEEKDAY{
                        double overtimePay(double hours, double payRate) { return ...}
                };
                WEEKEND{
                        double overtimePay(double hours, double payRate) { return ...}
                };
                private static final int HOURS_PER_SHIFT = 8;

                abstract double overtimePay(double hours, double payRate);

                double pay(double hoursWorked, double payRate){
                        double basePay = hoursWorked * payRate;
                        return basePay + overtimePay(hoursWorked, payRate);
                }
        }
}
```

31. Use instance fields instead of ordinals

Never derive a value of an enum to its ordinal

```
public enum Ensemble{
        SOLO, DUET, TRIO...;
        public int numberOfMusicians() {return ordinal() + 1}
}
```

Better approach

```
public enum Ensemble{
        SOLO(1), DUET(2), TRIO(3)...TRIPLE_QUARTET(12);
        private final int numberOfMusicians;
        Ensemble(int size) {this.numberOfMusicians = size;}
        public int numberOfMusicians() {return numberOfMusicians;}
}
```

32. Use EnumSet instead of bit fields

If the elements of an enumarated are used primarily in sets, use EnumSet.

```
public class Text{
        public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

        //Any Set could be passed. Best EnumSet
        public void applyStyles(Set<Style> styles){ ... }
}

//Use
text.applyStyles(EnumSet.of(Style.BOLD, Style. ITALIC));
```

It is a good practice to accept the interface `Set` instead of the implementation `EnumSet`.

33. Use EnumMap instead of ordinal indexing

Use EnumMap to associate data with an enum

```
Map<Herb.Type, Set<Herb>> herbsByType = new EnumMap<Herb.Type, Set<Herb>>(Herb.Type.class);

for (Herb.Type t : Herb.Type.values())
        herbsByType.put(t, new HashSet<Herb>())

for (Herb h : garden)
        herbsByType.get(h.type).add(h);

System.out.println(herbsByType);
```

In case you need a multidimensional relationship use `EnumMap<..., EnumMap<...>>`

34. Emulate extensible enums with interfaces

Enums types can not extend another enum types.

Opcodes as a use case of enums extensibility.

```
public interface Operation{
        double apply(double x, double y);
}
public enum BasicOperation implements Operation{
        PLUS("+"){
                public double apply(double x, double y) {return x + y}
        },
        MINUS("-"){...},TIMES("*"){...},DIVIDE("/"){...};

        private final String symbol;
        BasicOperation(String symbol){
                this.symbol = symbol;
        }
        @Override
        public String toString(){ return symbol; }
}
```

BasicOperation is not extensible, but the interface type *Operation* is, and it is the one used to represent operations in APIs.

Emulated extension type

```
public enum ExtendedOperation implements Operation{
        EXP("^"){
                public double apply(double x, double y) {return Math.pow(x,y)}
        }
        REMAINDER("%"){
                public double apply(double x, double y) {return x % y}
```

```
        }

        private final String symbol;
        ExtendedOperation(String symbol){
                this.symbol = symbol;
        }
        @Override
        public String toString(){ return symbol; }
}
```

35. Prefer annotations to naming patterns

Sample of the *@Test* annotation

Marker

```
//Marker annotation type declaration
import java.lang.annotation.*;

//Indicates that the annotated method is a test method.
//Use only on parameterless static methods.
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

`@Retention` and `@Target` are *meta-annotations*

`@Retention(RetentionPolicy.RUNTIME)` : indicates that the Test annotation should be retained at runtime.(It makes them visible to the test tool)

`@Target(ElementType.METHOD)` indicates that is legal only on method declarations. Not in class, fields or other programs declarations

Retention RetentionPolicies

Enum	Description
CLASS	Retain only at compile time, not runtime
RUNTIME	Retain at compile and also runtime
SOURCE	Discard by the compiler

Target ElementTypes

Enum	Valid on...
ANNOTATION_TYPE	Annotation type declaration
CONSTRUCTOR	constructors
FIELD	the field (includes also enum constants)
LOCAL_VARIABLE	local variables
METHOD	methods
PACKAGE	packages
PARAMETER	parameter declaration
TYPE	class, interface, annotation and enums declaration

Enum	Valid on...
TYPE_PARAMETER	type parameter declarations
TYPE_USE	the use of a specific type

Use

```
public class Sample{
        @Test public static void m1(){}         // Test pass
        public static void m2(){}                       // Not test applied
        @Test public static void m3(){          // Test fail
                throw new RuntimeException("Boom")
        }
        @Test public void m4(){}                        // Invalid use. Non static method.
}
```

Process annotations

```
import java.lang.reflect.*

public class RunTests{
        public static void main(String[] args) throws Exception {
                int tests = 0;
                int passed  = 0;
                Class testClass = Class.forName(args[0]);
                for (Method m : testClass.getDeclaredMethods()){
                        if (m.isAnnotationPresent(Test.class)){
                                tests++;
                                try{
                                        m.invoke(null);
                                        passed++;
                                } catch(InvocationTargetException wrappedExc){
                                        Throwable exc = wrappedExc.getCause();
                                        System.out.println(m + " failed: " + exc);
                                } catch (Exception exc){
                                        System.out.println("INVALID @Test: " + m);
                                }
                        }
                }
                System.out.printf("Passed: %d, Failed: %d%n", passed, tests - passed);
        }
}
```

36. Consistently use the *Override* annotation

Use the *Override* annotation on every method declaration that you believe to override a super class declaration.

```
public class Bigram {
        private final class first;
        private final class second;
        public Bigram(char first, char second){
                this.first = first;
                this.second = second;
        }
        public boolean equals(Bigram b){ //ERROR.
                return b.first == first && b.second == second;
        }
        ...
}
```

We are overloading equals instead of overriding it.
The correct sign to override the super method is:

```
public boolean equals(Object o){}
```

With the use of Override the compiler would alert us about our mistake.

37. Use marker interfaces to define types

Marker interface in Java is interfaces with no field or methods or in simple word empty interface in java is called marker interface.

A *marker interface* is an interface that contains no method declarations, but "marks" a class that implements the interface as having some property.

When your class implements `java.io.Serializable` interface it becomes Serializable in Java and gives compiler an indication that use Java Serialization mechanism to serialize this object.

- Marker interfaces define a type that is implemented by instances of the marked class; marker annotations do not. (Catch errors in compile time).
- They can be targeted more precisely than marker annotations.
- It's possible to add more information to an annotation type after it is already in use.

7. METHODS

38. Check parameters for validity

Check parameters before execution as soon as possible.

Add in public methods *@throw*, and use *assertions* in non public methods

Do it also in constructors.

39. Make defensive copies when needed.

You must program defensively, with the assumption that clients of your class will do their best to destroy its invariants.

```
//Broken "immutable" time period
public final class Period{
        private final Date start;
        private final Date end;
        /**
        * @param start the beginning of the period
        * @param end the end of the period; must not precede start;
        * @throws IllegalArgumentException if start is after end
        * @throws NullPointerException if start or end is null
        */
        public Period(Date start, Date end) {
                if(start.compare(end) > 0)
                        throw new IllegalArgumentException(start + " after " + end );
                this.start = start;
                this.end = end;
        }

        public Date start(){
                return start;
        }

        public Date end(){
                return end;
        }
```

```
        ...
}
```

Attack. Because the client keep a copy (pointer) of the parameter, it can always change it after the constructor.

```
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
end.setYear(78)// Modifies internal of p!
```

Make a *defensive copy* of each mutable parameter to the constructor.

```
public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
        if(start.compare(end) > 0)
                throw new IllegalArgumentException(start + " after " + end );
        }
```

Defensive copies are made before checking the validity of the parameter (Item 38), and the validity check is performed on the copies rather than on the originals. It protects the class against changes to the parameters from another thread during the time between the parameters are checked and the time they are copied.(*Window of vulnerability*,time-of-check/time-of-use *TOCTOU* attack)

Do not use *clone* method to make a defensive copy of a parameter whose type is subclass-able by untrusted parties.

Second Attack. Because the accessors returns the object used in the Period class, the client can change its value without passing the constrains.

```
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end.setYear(78)// Modifies internal of p!
```

Return *defensive copies* of mutable internal fields.

```
public Date start(){
        return new Date(start.getTime());
}

public Date end(){
        return new Date(end.getTime());
}
```

Preferable is to use **immutable objects**(Item 15)

40. Design method signatures carefully

- Choose method names carefully. (Item 56)
- Don't go overboard in providing convenience methods. Don't add too many.
- Avoid long parameter list. Make a subset of methods, helper classes (Item 22), or a builder (Item 2) instead.
- For parameter types, favor interfaces over classes (Item 52) No reason to write a method that takes a *HashMap* on input, use *Map* instead.
- Prefer two-element enum types to *boolean* parameters. `public enum TemperatureScale {CELSIUS, FARENHEIT}`

41. Use overloading judiciously

The choice of which overloading to invoke is made at compile time. Selection among overloaded methods is static, while selection among overridden methods is dynamic.

```
// Broken! - What does this program print?
public class CollectionClassifier {
        public static String classify(Set<?> s) {
                return "Set";
        }
        public static String classify(List<?> lst) {
                return "List";
        }
        public static String classify(Collection<?> c) {
                return "Unknown Collection";
        }
        public static void main(String[] args) {
                Collection<?>[] collections = {
                        new HashSet<String>(),
                        new ArrayList<BigInteger>(),
                        new HashMap<String, String>().values()
                };
        for (Collection<?> c : collections)
                System.out.println(classify(c)); // Returns "Unknown Collection" 3 times
        }
}
```

Overriding works different. The “most specific” overriding method always gets executed.

```
class Wine {
        String name() { return "wine"; }
}
class SparklingWine extends Wine {
        @Override String name() { return "sparkling wine"; }
}
class Champagne extends SparklingWine {
        @Override String name() { return "champagne"; }
}
public class Overriding {
        public static void main(String[] args) {
                Wine[] wines = {
                        new Wine(), new SparklingWine(), new Champagne()
                };
                for (Wine wine : wines)
                        System.out.println(wine.name()); // prints: wine, sparkling wine, and champagne
        }
}
```

Overloading does not give the functionallity we want in the first sample. A possible solution is:

```
public static String classify(Collection<?> c) {
        return  c instanceof Set ? "Set" :
                        c instanceof List ? "List" : "Unknown Collection";
}
```

Do not have overloaded methods in APIs to avoid confusing the clients of the API.

A conservative policy to is never to export two overloadings with the same number of parameters. Use different names. `writeBoolean(boolean)` , `writeInt(int)` , and `writeLong(long)`

For constructors you can use static factories (Item 1)

If parameters are radically different this rules can be violate but always ensure that all overloadings behave identically when passed the same parameters. To ensure this, have the more specific overloading forward to the more general.

```
public boolean contentEquals(StringBuffer sb) {
        return contentEquals((CharSequence) sb);
}
```

42. Use varargs judiciously

varargs methods are a convenient way to define methods that require a variable number of arguments, but they should not be overused.

```
// The right way to use varargs to pass one or more arguments
static int min(int firstArg, int... remainingArgs) {
        int min = firstArg;
        for (int arg : remainingArgs)
                if (arg < min)
                        min = arg;
                return min;
}
```

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

Return an immutable empty array instead of null.

```
// The right way to return an array from a collection
private final List<Cheese> cheesesInStock = ...;

private static final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];

/**
* @return an array containing all of the cheeses in the shop.
*/
public Cheese[] getCheeses() {
        return cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

In Collections emptySet, emptyList and emptyMap methods do the same job.

```
// The right way to return a copy of a collection
public List<Cheese> getCheeseList() {
     if (cheesesInStock.isEmpty())
             return Collections.emptyList(); // Always returns same list
     else
             return new ArrayList<Cheese>(cheesesInStock);
}
```

44. Write *doc comments* for all exposed API elemnts

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, what the method does rather than how it does its job.
- The *preconditions* described implicity by the *@throws* tag.
- The *postconditions*, things that will be true after the invocation has completed successfully.

- The *side effects*, change in the state of the system.
- The *thread safety* of the class and methods
- The *summary description of the element*, the first "sentence" of each doc comment.Should not contains a space after a period.

```
/**
* Returns the element at the specified position in this list. *
* <p>This method is <i>not</i> guaranteed to run in constant
* time. In some implementations it may run in time proportional * to the element position.
*
* @param index index of element to return; must be
* non-negative and less than the size of this list
* @return the element at the specified position in this list
* @throws IndexOutOfBoundsException if the index is out of range
*               ({@code index < 0 || index >= this.size()})
*/
E get(int index);
```

Have special care in:

- Generics: document all type parameters
- Enums: document all the constants, the type and the public methods.
- Annotatons: document all members an the type.

Don't forget to documment:

- The *thread-safety level* (Item 70)
- The *serialized form* (Item 75), if the class is *serializable*

8. GENERAL PROGRAMMING

45. Minimize the scope of local variables.

Declare local variable where it is first used.
Most local variable declaration should contain an initializer.
Prefer for loops to while loops.
Keep methods small and focused.

46. Prefer for-each loops to traditional for loops.

```
 // No longer the preferred idiom to iterate over a collection!
        for (Iterator i = c.iterator(); i.hasNext(); ) {
                doSomething((Element) i.next()); // (No generics before 1.5)
}
// No longer the preferred idiom to iterate over an array!
   for (int i = 0; i < a.length; i++) {
       doSomething(a[i]);
}
```

Use for each loop:

```
// The preferred idiom for iterating over collections and arrays
   for (Element e : elements) {
       doSomething(e);
}
```

Error when iterating in nested loops

```
// Can you spot the bug?
  enum Suit { CLUB, DIAMOND, HEART, SPADE }
  enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
              NINE, TEN, JACK, QUEEN, KING }
  ...
  Collection<Suit> suits = Arrays.asList(Suit.values());
  Collection<Rank> ranks = Arrays.asList(Rank.values());
  List<Card> deck = new ArrayList<Card>();
  for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
      for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
          deck.add(new Card(i.next(), j.next())); // i.next() should be run only in the outer loop
```

A solution

```
// Fixed, but ugly - you can do better!
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); ) {
   Suit suit = i.next();
   for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
               deck.add(new Card(suit, j.next()));
}
```

For each loop fix this directly

```
// Preferred idiom for nested iteration on collections and arrays
   for (Suit suit : suits)
       for (Rank rank : ranks)
           deck.add(new Card(suit, rank));
```

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 remove 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 it- erators or index variables can be advanced in lockstep (as demonstrated unin-tentionally in the buggy card examples above).

47. Know and use libraries

By using a standard library:

- Advantage of the knowledge of the experts who wrote it and the experience of those who used it before you.
- Don't have to waste your time writing ad hoc solutions to problems that are only marginally related to your work.
- Their performance tends to improve over time
- Your code will be easily readable, maintainable, and reusable.

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:

- java.lang
- java.util
- java.io
- java.util.concurrent

48. Avoid float and double if exact answer are required

For monetary calculations use *int*(until 9 digits) or *long* (until 18 digits) taken you care of the decimal part and you don't care too much about the rounding. Use *BigDecimal* for numbers bigger that 18 digits and if you need full control of the rounding methods used.

49. Prefer primitive types to boxed primitives

Primitives: *int*, *double*, *boolean*
Boxed Primitives: *Integer*, *Double*, *Boolean*
Differences:

- Two boxed primitives could have the same value but different identity.
- Boxed primitives have one nonfunctional value: *null*.
- Primitives are more space and time efficient.

Don't use == between boxed primitives.

```
first = new Integer(1);
second = new Integer(1);
first == second; //Uses unboxing  Don't have to be true.
```

Use Auto-unboxing to create new primitives

```
...
int f = first;  //Auto-unboxing
int s = second  //Auto-unboxing
f == s;// This is true
```

If a Boxed primitive is not initialize it will return null

```
Integer  i;
i == 42 // NullPointerException
```

Performance can be perturbed when boxing primitives values due to the creation of unnecessary objects.

When you **must** use boxed primitives:

- As elements, keys and values in Collections
- As type parameters in parametrized types (Chapter 5)
- When making reflective invocations (Item 53)

In other cases prefer primitives.

50. Avoid Strings where other types are more appropriate

- Strings are more cumbersome than other types.
- Strings are less flexible than other types.
- String are slower than other types.
- Strings are more error-prone than other types.
- 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*.

So, use String to represent text!

51. Beware the performance of string concatenation

Using the string concatenation operator repeatedly to concatenate *n* strings requires time quadratic in *n*.

```
// Inappropriate use of string concatenation - Performs horribly!
public String statement()
        String result = "";
        for (int i = 0; i < numItems(); i++)
                result += lineForItem(i);
        return result;
```

To achieve acceptable performance, use StringBuilder in place of String.

```
public String statement(){
        StringBuilder b = new StringBuilder(numItems() * LINE_WIDTH);
        for (int i = 0; i < numItems(); i++)
                b.append(lineForItem(i));
        return b.toString();
}
```

52. Refer to objects by their interface

If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types.

```
// Good - uses interface as type
List<Subscriber> subscribers = new Vector<Subscriber>();
```

rather than this:

```
// Bad - uses class as type!
Vector<Subscriber> subscribers = new Vector<Subscriber>();
```

It makes the program much more flexible. We could change the implementation of the `subscribers` changing just one line.

```
List<Subscriber> subscribers = new ArrayList<Subscriber>();
```

Caveat: if the original implementation has a special functionality not required the interface contract and the code depended on that functionality, the new implementation must provide this functionality.

If there is not an appropriate interface we can refer to the object by a class. Like:

- Value classes: String, BigDecimal...
- Framework classes
- Classes that extend the interface functionality with extra methods.

53. Prefer interfaces to reflection

java.lang.reflection offers access to information about loaded classes.

Given a *Class* object, you can obtain *Constructor*, *Method* and *Field* instances.

Allows one class to use another, even if the latter class did not exist when the former was compiled.

- Lose of all benefits of compile-time type checking
- Code to perform reflective access is clumsy and verbose

- Performance suffers.

As a rule, objects should not be accessed reflectively in normal applications at runtime

Obtain many of the benefits of reflection incurring few of its costs by **creating instances reflectively and access them normally via their interface or superclass**.

```
// Reflective instantiation with interface access
public static void main (String[] args){
        // Translate the class name into a class object
        Class<?> cl =  null;
        try{
                cl = Class.forName(args[0]);// Class is specified by the first command line argument
        }catch(ClassNotFoundException e){
                System.err.println("Class not found");
                System.exit(1);
        }

        //Instantiate the class
        Set<String> s = null;
        try{
                s = (Set<String>) cl.newInstance(); //  The class can be either a HashSet or a TreeSet
        } catch(IllegalAccessException e){
                System.err.println("Class not accessible");
                System.exit(1);
        }catch(InstantionationException e){
                System.err.println("Class not instantiable");
                System.exit(1);
        }

        //Excercise the Set
        // Print the remaining arguments. The order depends in the class. If it is a HashSet
        // the order will be random, if it is a TreeSet it will be alphabetically
        s.addAll(Arrays.asList(args).subList(1,args.length));
        System.out.println(s);
}
```

A legitimate use of reflection is to manage a class's dependencies on other classes, methods or fields that may be absent at runtime.

Reflection is powerful and useful in some sophisticated systems programming tasks. It has many disadvantages. Use reflection, if possible, only to instantiate objects and access the objects using an interface or a superclass that is known at compile time.

54. Use native methods judiciously

Historically, native methods have had three main uses.

- They provided access to platform-specific facilities.
- They provided access to libraries of legacy code.
- To write performance-critical parts

New Java versions make use of NDK rarely advisable for improve performance.

55. Optimize judiciously

Strive to write good programs rather than fast ones, speed will follow.
If a good program is not fast enough, its architecture will allow it to be optimized.

- More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity.
- We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
- We follow two rules in the matter of optimization:

- Rule 1. Don't do it.
- Rule 2 (for experts only). Don't do it yet — that is, not until you have a perfectly clear and unoptimized solution.

If you finally do it **measure performance before and after each attempted optimization**, and focus firstly in the choice of algorithms rather than in low level optimizations.

56. Adhere to generally accepted naming conventions

Typographical naming conventions

Indentifier Type	Examples
Package	com.google.inject, org.joda.time.format
Class or Interface	Timer, FutureTask, LinkedHashMap, HttpServlet
Method or Field	remove, ensureCapacity, getCrc
Constant Field	MIN_VALUE, NEGATIVE_INFINITY
Local Variable	i, xref, houseNumber
Type Parameter	T, E, K, V, X, T1, T2

Grammatical naming conventions

Type	Convention	Example
Classes and enum types	Singular noun or noun phrase	Timer, BufferedWriter, ChessPiece
Interfaces	Like classes	Collection, Comparator
Interfaces	With an adjective ending in *able* or *ible*	Runnable, Iterable, Accessible
Annotation types	Nouns, verbs, prepositions, adjectives ...	BindingAnnotation, Inject, ImplementedBy, Singleton
Static factories (common names)	---	valueOf, of, getInstance, newInstance, getType, newType
Methods that...	---	---
perform actions	verb or verb phrase	append, drawImage
return a boolean	names beginning with *is* or, *has*	isDigit, isProbablePrime, isEmpty, isEnabled, hasSiblings
return a non-boolean or attribute	noun, a noun phrase, or begin with *get*	size, hashCode, or getTime
convert the type of an object	*toType*	toString, toArray
return a view (Item 5) of a different type	*asType*	asList
return a primitive with the same value	*typeValue*	intValue

9. EXCEPTIONS

57. Use exceptions only for exceptional conditions

Exceptions are for exceptional conditions.
Never use or (expose in the API) exceptions for ordinary control flow.

58. Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

Throwables:

- checked exceptions: for conditions from which the caller can reasonably be expected to recover
- unchecked exceptions: shouldn't, be caught. recovery is impossible and continued execution would do more harm than good.
  - runtime exceptions: to indicate programming errors. The great majority indicate precondition violations.
  - errors : are reserved for use by the JVM. (as a convention)

Unchecked throwables that you implement should **always** subclass *RuntimeException*.

59. Avoid unnecessary use of checked exceptions

Use checked exceptions only if these 2 conditions happen:

- The exceptional condition cannot be prevented by proper use of the API
- The programmer using the API can take some useful action once confronted with the exception.

Refactor the checked exception into a unchecked exception to make the API more pleasant.

Invocation with checked exception

```
try {
        obj.action(args);
} catch(TheCheckedException e) {
        // Handle exceptional condition
        ...
}
```

Invocation with state-testing method and unchecked exception

```
if (obj.actionPermitted(args)) {
        obj.action(args);
} else {
        // Handle exceptional condition
        ...
}
```

60. Favor 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

Java 8 Exceptions

AclNotFoundException	InvalidMidiDataException	RefreshFailedException
ActivationException	InvalidPreferencesFormatException	RemarshalException
AlreadyBoundException	InvalidTargetObjectTypeException	RuntimeException
ApplicationException	IOException	SAXException
AWTException	JAXBException	ScriptException
BackingStoreException	JMException	ServerNotActiveException
BadAttributeValueExpException	KeySelectorException	SOAPException
BadBinaryOpValueExpException	LambdaConversionException	SQLException
BadLocationException	LastOwnerException	TimeoutException
BadStringOperationException	LineUnavailableException	TooManyListenersException
BrokenBarrierException	MarshalException	TransformerException
CertificateException	MidiUnavailableException	TransformException
CloneNotSupportedException	MimeTypeParseException	UnmodifiableClassException
DataFormatException	MimeTypeParseException	UnsupportedAudioFileException
DatatypeConfigurationException	NamingException	UnsupportedCallbackException
DestroyFailedException	NoninvertibleTransformException	UnsupportedFlavorException
ExecutionException	NotBoundException	UnsupportedLookAndFeelException
ExpandVetoException	NotOwnerException	URIReferenceException
FontFormatException	ParseException	URISyntaxException
GeneralSecurityException	ParserConfigurationException	UserException
GSSException	PrinterException	XAException
IllegalClassFormatException	PrintException	XMLParseException
InterruptedException	PrivilegedActionException	XMLSignatureException
IntrospectionException	PropertyVetoException	XMLStreamException
InvalidApplicationException	ReflectiveOperationException	XPathException

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.

```
// Exception Translation
try {
        // Use lower-level abstraction to do our bidding
        ...
} catch(LowerLevelException e) {
        throw new HigherLevelException(...);
}
```

Do not overused. The best way to deal with exceptions from lower layers is to avoid them, by ensuring that lower-level methods succeed.

Exception chaining When the lower-level exception is utile for the debugger, pass the lower-level to the higher-level exception, with an accessor method (Throwable.getCause) to retrieve the lower-level exception.

```
// Exception with chaining-aware constructor
class HigherLevelException extends Exception {
        HigherLevelException(Throwable cause) {
        super(cause);
        }
}
```

62. Document all exceptions thrown by each method

Unchecked exceptions generally represent programming errors (Item 58), and familiarizing programmers with all of the errors they can make helps them avoid making these errors.

Always declare checked exceptions individually, and document precisely the conditions under which each one is thrown using the Javadoc @throws tag.

Do not use the throws keyword to include unchecked exceptions in the method declaration.

63. Include failure-capture information in detail messages

It is critically important that the exception's `toString` method return as much information as possible concerning the cause of the failure. 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 is to require this information in their constructors instead of a string detail message. Also provide accessors to this parameters could help useful to recover from the failure

```
// Alternative IndexOutOfBoundsException.
public IndexOutOfBoundsException(int lowerBound, int upperBound, int index) {...}
```

64. Strive for failure atomicity

A failed method invocation should leave the object in the state that it was in prior to the invocation. Options to achieve this:

- Design immutable objects
- Order the computation so that any part that may fail takes place before any part that modifies the object.
- Write recovery code (Undo operation)
- Perform the operation on a temporary copy of the object, and replace it once is completed.

65. Don't ignore exceptions

Don't let catch blocks empty.

```
// Empty catch block ignores exception - Highly suspect!
try {
...
} catch (SomeException e) {
}
```

10. CONCURRENCY

66. Synchronize access to shared mutable data

Synchronization prevent a thread from observing an object in an inconsistent state. Synchronization ensures that each thread entering a synchronized method or block sees the effects of all previous modifications that were guarded by the same lock.

In Java reading or writing a variable is atomic unless type long or double, but for all atomic operations it does not guarantee that a value written by one thread will be visible to another.

Synchronization is required for reliable communication between threads as well as for mutual exclusion.

```
// Broken! - How long would you expect this program to run?
public class StopThread {
        private static boolean stopRequested;

        public static void main(String[] args) throws InterruptedException {
                Thread backgroundThread = new Thread(new Runnable() {
                        public void run() {
                                int i = 0;
                                while (!stopRequested)
                                        i++;
                        }
                });
                backgroundThread.start();

                TimeUnit.SECONDS.sleep(1);
                stopRequested = true;
                }
}
```

Because of *hoisting* the while loop is translated to this:

```
if (!done)
        while (true)
                i++;
```

and therefore the loop never stops.

```
// Properly synchronized cooperative thread termination
public class StopThread {
        private static boolean stopRequested;

        private static synchronized void requestStop() {
                stopRequested = true;
        }
        private static synchronized boolean stopRequested() {
                return stopRequested;
        }

        public static void main(String[] args) throws InterruptedException {
                Thread backgroundThread = new Thread(new Runnable() {
                        public void run() {
                                int i = 0;
                                while (!stopRequested())
                                        i++;
                        }
                });
                backgroundThread.start();

                TimeUnit.SECONDS.sleep(1);
                requestStop();
                }
}
```

Synchronization has no effect unless both read and write operations are synchronized.

Other solution is using *volatile* modifier, it performs no mutual exclusion, but it guarantees that any thread that reads the field will see the most recently written value.

```
public class StopThread {
        private static volatile boolean stopRequested;

        public static void main(String[] args) throws InterruptedException {
                Thread backgroundThread = new Thread(new Runnable() {
                        public void run() {
                                int i = 0;
                                while (!stopRequested)
                                        i++;
                        }
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
        }
}
```

Be careful with *volatile* when using non atomic functions like ++

```
private static volatile int nextSerialNumber = 0;

public static int generateSerialNumber() {
        return nextSerialNumber++;
}
```

We can synchronize the access to nextSerialNumber and remove *volatile* or use AtomicLong. AtomicLong can help us with the synchronization of long values

```
private static final AtomicLong nextSerialNum = new AtomicLong();

public static long generateSerialNumber() {
        return nextSerialNum.getAndIncrement();
}
```

effectively immutable: data object modified by one thread to modify shared it with other threads, synchronizing only the act of sharing the object reference. Other threads can then read the object without further synchronization, so long as it isn't modified again.

safe publication: Transferring such an object reference from one thread to others.

In general: When multiple threads share mutable data, each thread that reads or writes the data must perform synchronization

Best thing to do: **Not share mutable data.**

67. Avoid excessive synchronization

Inside a synchronized region, do not invoke a method (*alien*) that is designed to be overridden, or one provided by a client in the form of a function object (Item 21). Calling it from a synchronized region can cause exceptions, deadlocks, or data corruption. Move alien method invocations out of synchronized blocks. Taking a “snapshot” of the object that can then be safely traversed without a lock.

```
// Alien method moved outside of synchronized block - open calls
private void notifyElementAdded(E element) {
        List<SetObserver<E>> snapshot = null;
        synchronized(observers) {
```

```
                snapshot = new ArrayList<SetObserver<E>>(observers);
            }
            for (SetObserver<E> observer : snapshot)
                observer.added(this, element);
    }
```

Or use a *concurrent collection* (Item 69) known as CopyOnWriteArrayList. It is a variant of ArrayList in which all write operations are implemented by making a fresh copy of the entire underlying array. The internal array is never modified and iteration requires no locking.

open call: An alien method invoked outside of a synchronized region

As Rule:

- **do as little work as possible inside synchronized regions**
- **limit the amount of work that you do from within synchronized regions**

68. Prefer executors and tasks to threads

Creating a work queue:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

Submit a runnable for execution:

```
executor.execute(runnable);
```

Terminate gracefully the executor

```
executor.shutdown();
```

ExecutorService possibilities:

- wait for a particular task to complete: `background thread SetObserver`
- wait for any or all of a collection of tasks to complete: `invokeAny` or `invokeAll`
- wait for the executor service's graceful termination to complete: `awaitTermination`
- retrieve the results of tasks one by one as they complete: `ExecutorCompletionService` *...

For more than one thread use a *thread pool*. For lightly loaded application, use: `Executors.new-CachedThreadPool` For heavily loaded application, use: `Executors.newFixedThreadPool`

executor service: mechanism for executing tasks

task: unit of work. Two types.

- Runnable
- Callable, similar to Runnable but returns a value

69. Prefer concurrency utilities to *wait* and *notify*

Given the difficulty of using wait and notify correctly, you should use the higher-level concurrency utilities instead.

- Executor Framework (Item 68)
- Concurrent Collections
- Synchronizers

Concurrent Collections: High-performance concurrent implementations of standard collection interfaces (List, Queue, and Map)
Use concurrent collections in preference to externally synchronized collections
Some interfaces have been extended with blocking operations, which wait (or block) until they can be successfully performed. This allows blocking queues to be used for work queues (*producer-consumer queues*). One or more producer threads enqueue work items and from which one or more consumer threads dequeue and process items as they become available. ExecutorService implementations, including ThreadPoolExecutor, use a BlockingQueue (Item 68).

Synchronizers: objects that enable threads to wait for one another, allowing them to coordinate their activities (CountDownLatch, Semaphore, CyclicBarrier, Exchanger)

wait: Always use the wait loop idiom to invoke the wait method; never invoke it outside of a loop. The loop serves to test the condition before and after waiting.

```
// The standard idiom for using the wait method
synchronized (obj) {
        while (<condition does not hold>){
                obj.wait(); // (Releases lock, and reacquires on wakeup)
        }
        ... // Perform action appropriate to condition
}
```

notify: Wakes a single waiting thread, assuming such a thread exists.

notifyAll: Wakes all waiting threads.

Use always use *notifyAll* (and not forget to use the wait loop explained before) You may wake some other threads, but these threads will check the condition for which they're waiting and, finding it false, will continue waiting.

There is seldom, if ever, a reason to use wait and notify in new code. Use higher-level language

70. Document thread safety

Looking for the synchronized modifier in a method declaration is an implementation detail. To enable safe concurrent use, a class must clearly document what level of thread safety it supports.

- immutable: No external synchronization is necessary (i.e. String, Long, BigInteger)
- unconditionally thread-safe: mutable but with internal synchronization. No need for external synchronization (i.e. Random, ConcurrentHashMap)
- conditionally thread-safe: some methods require external synchronization.(i.e. Collections.synchronized wrappers)
- not thread-safe: external synchronization needed (i.e. ArrayList, HashMap.)
- thread-hostile: not safe for concurrent use (i.e. System.runFinalizersOnExit)

Thread safety annotations are Immutable, ThreadSafe, and NotThreadSafe. To document a conditionally thread-safe class indicate which invocation sequences require external synchronization, and which lock must be acquired to execute these sequences.

Use private lock object idiom to prevent users to hold the lock for a long period of time in unconditionally thread-safe classes.

```
// Private lock object idiom - thwarts denial-of-service attack
private final Object lock = new Object();

public void foo() {
        synchronized(lock) {
                ...
        }
}
```

71. Use lazy initialization judiciously

Use it if a field is accessed only on a fraction of the instances of a class and it is costly to initialize the field.
It decreases the cost of initializing a class or creating an instance, but increase the cost of accessing it.
For multiple threads, lazy initialization is tricky.

```
// Normal initialization of an instance field
private final FieldType field = computeFieldValue();
```

To break an initialization circularity: **synchronized accessor**

```
// Lazy initialization of instance field - synchronized accessor
private FieldType field;
        synchronized FieldType getField() {
                if (field == null)
                        field = computeFieldValue();
                return field;
}
```

For performance on a static field: **lazy initialization holder class idiom**, adds practically nothing to the cost of access.

```
// Lazy initialization holder class idiom for static fields
private static class FieldHolder {
        static final FieldType field = computeFieldValue();
}
static FieldType getField() { return FieldHolder.field; }
```

For performance on an instance field: **double-check idiom**.

```
// Double-check idiom for lazy initialization of instance fields
private volatile FieldType field;
FieldType getField() {
        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;
}
```

Instance field that can tolerate repeated initialization: **single-check idiom.**

```
// Single-check idiom - can cause repeated initialization!
private volatile FieldType field;
        private FieldType getField() {
                FieldType result = field;
                if (result == null)
                        field = result = computeFieldValue();
                return result;
}
```

72. Don't depend on thread scheduler

Thread scheduler determines which runnable, get to run, and for how long.Operating systems will try to make this determination fairly, but the policy can vary. So any program that relies on the thread scheduler for correctness or performance is likely to be non portable.
To ensure that the average number of runnable threads is not significantly greater than the number of processors. Threads should not run if they aren't doing useful work, tasks should be:

- reasonably small but not too small or dispatching overhead
- independent of one another
- not implement busy-wait

73. Avoid thread groups

Thread groups are obsolete.

11. SERIALIZATION

74. Implement ***Serializable*** judiciously

Adding `implements Serializable` is the easiest way to serialize a class, but it decreases the flexibility to change a class's implementation once it has been released. The byte-stream encoding (or serialized form) becomes part of its exported API.

It has three major drawbacks:

- Class's private and package-private instance fields become part of its exported API (Item 13)
- Change the class's internal representation, will cause make old versions of serialized objects incompatible.
- Increases the likelihood of bugs and security holes.
- Increases the testing burden associated with releasing a new version of a class.

Implementing the Serializable interface has many real costs.

should implement Serializable:

- value classes such as *Date* and *BigInteger*
- as should most collection classes

rarely implement Serializable:

- Classes representing active entities, such as thread pools
- Classes designed for inheritance
- Interfaces should rarely extend it
- Inner classes (Item 22)

A subclass of a not serializable class can not be serializable, unless it has a parameterless constructor.

75. Consider using a custom serialized form

Do not accept the default serialized form without first considering whether it is appropriate.
The default serialized form is likely to be appropriate if an object's physical representation is identical to its logical content. Like a Point or Person Name.
Even if you decide that the default serialized form is appropriate, you often must provide a `readObject` method to ensure invariants and security

Using the default serialized form when an object's physical representation differs substantially from its logical data content has four disadvantages:

- It permanently ties the exported API to the current internal representation.

- It can consume excessive space.
- It can consume excessive time.
- It can cause stack overflows.

Every instance field that is not labeled *transient* will be serialized when the defaultWriteObject method is invoked (Whether or not you use the default serialized)
Every instance field that can be made transient should be made so.(i.e. computed from "primary data fields) Mark *nontransient* every field whose value is part of the logical state of the object.

Impose synchronization on object serialization that you would impose on any other method that reads the entire state of the object.

```
// writeObject for synchronized class with default serialized form
private synchronized void writeObject(ObjectOutputStream s)
        throws IOException {
                s.defaultWriteObject();
}
```

Declare an explicit serial version UID in every serializable class you write.

```
private static final long serialVersionUID = randomLongValue ;
```

76. Write *readObject* methods defensively

readObject method is a public constructor that takes a byte stream as its sole parameter. It demands same care as any other public constructor:

- check its arguments for validity (Item 38)
- make defensive copies of parameters where appropriate (Item 39)

```
// readObject method with defensive copying and validity checking
private void readObject(ObjectInputStream s) throws IOException, ClassNotFoundException {
        s.defaultReadObject();

        // Defensively copy our mutable components
        start = new Date(start.getTime());
        end = new Date(end.getTime());

        // Check that our invariants are satisfied
        if (start.compareTo(end) > 0)
                throw new InvalidObjectException(start +" after "+ end);
}
```

Every serializable immutable class containing private mutable components must defensively copy these components in its readObject method.

Do not use the *writeUnshared* and *readUnshared* methods(Java 1.4). Are faster but not safer than defensive copying.

Summary guidelines:

- For classes with object reference fields that must remain private, defensively copy each object in such a field. Mutable components of immutable classes fall into this category.
- Check any invariants and throw an InvalidObjectException if a check fails. The checks should follow any defensive copying.
- If an entire object graph must be validated after it is deserialized, use the *ObjectInputValidation* interface [JavaSE6, Serialization].
- Do not invoke any overridable methods in the class, directly or indirectly.

77. For instance control, prefer *enum* types to *readResolve*

Singleton classes would no longer be singletons if they "implements Serializable". The *readResolve* feature allows you to substitute another instance for the one created by *readObject*. So the original instance is returned.

To prevent attacks when using *readResolve* for instance control, all instance fields with object reference types must be declared transient.

Another way to prevent attacks in instance-controlled classes and when instances are known at compile time,is using *enum*. JVM guarantees that only will be one instance.

Accessibility: *readResolve* method on:

- final class: private.
- nonfinal class:
  - private: will not apply to any subclasses.
  - package-private: it will apply only to subclasses in the same package.
  - protected or public: it will apply to all subclasses that do not override it.

78. Consider serialization proxies instead of serialized instances

serialization proxy: A private static nested class of the serializable class that represents the logical state of an instance of the enclosing class.
It has a single constructor, whose parameter type is the enclosing class, and copies the data from its arguments.
No need of consistency checking or defensive copying. Both the enclosing class and its serialization proxy must be declared to implement Serializable.

```
        // Serialization proxy for Period class
        private static class SerializationProxy implements Serializable {
                private final Date start;
                private final Date end;

                SerializationProxy(Period p) {
                        this.start = p.start;
                        this.end = p.end;
                }

                private static final long serialVersionUID = 234098243823485285L; // Any number will do (Item
        }


}
```

writeReplace translates an instance of the enclosing class to its serialization proxy prior to serialization. The serialization system will never generate a serialized instance of the enclosing class.

```
// writeReplace method for the serialization proxy pattern
private Object writeReplace() {
        return new SerializationProxy(this);
}
```

If an attacker fabricates a serialized object in an attempt to violate the class's invariants, we throw an Exception.

```
// readObject method for the serialization proxy pattern
private void readObject(ObjectInputStream stream) throws InvalidObjectException {
        throw new InvalidObjectException("Proxy required");
}
```

Add a *readResolve* method on the *SerializationProxy* class to return a logically equivalent instance of the enclosing class.

```
// readResolve method for Period.SerializationProxy
private Object readResolve() {
        return new Period(start, end); // Uses public constructor
}
```

Limitations, not compatible with:

- classes that are extendable by their clients (Item 17)
- some classes whose object graphs contain circularities