High Concepts

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

public static Boolean valueOf(boolean b) {

return b ? Boolean.TRUE : Boolean.FALSE;

}

1. One advantage of static factory methods is that, unlike constructors, they have names**.**
2. A second advantage of static factory methods is that, unlike constructors, they are not required to create a new object each time they’re invoked.This allows immutable classes (Item 15) to use preconstructed instances, or to cache instances as they’re constructed,
3. Boolean.valueOf(boolean) is example, which never creates a new object, example of flyweight pattern.
4. *instance-controlled class where a.equals(b)if only a==b so no two equal instance exists.*
5. A third advantage of static factory methods is that, unlike constructors,

they can return an object of any subtype of their return type.

1. 5. an API can return objects without making their classes public .This technique lends itself to *interface-based frameworks,*
2. Java Collections Framework has thirty-two convenience implementations of its collection interfaces, providing unmodifiable collections, synchronized collections, and the like. Nearly all of these implementations are exported via static factory methods in one noninstantiable class (java.util.Collections). The classes of the returned objects are all nonpublic.
3. The class java.util.EnumSet , introduced in release 1.5, has no public constructors, only static factories. It returns two implementation , regular and jumbo depending on the size.

A service provider framework

The class of the object returned by a static factory method need not even exist

at the time the class containing the method is written. Such flexible static factory

methods form the basis of *service provider frameworks*,

*1 Service interface -* which providers implement;

2 *provider registration API -* the system uses to register implementations, giving clients

access to them

*3.* a *service access API –* clients use to obtain an instance of the service.

*4. service provider interface (Optinal) –*

*In JDBC*

Connection plays the part of the service interface,

DriverManager.registerDriver is the provider registration API,

DriverManager.getConnection is the service access API, and

Driver is the service provider interface.

*7.* A fourth advantage of static factory methods is that they reduce the verbosity

of creating parameterized type instances.

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

The above can be replaced by :

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

public static <K, V> HashMap<K, V> newInstance() {

return new HashMap<K, V>();

}

1. The main disadvantage of providing only static factory methods is that classes without public or protected constructors cannot be subclassed.
2. A second disadvantage of static factory methods is that they are not readily distinguishable from other static methods. They do not stand out in API documentation in the way that constructors do, so it can be difficult to figure out

how to instantiate a class that provides static factory methods instead of constructors.

2 **Consider a builder when faced with many constructor parameters**

Java Beans constructions : Unfortunately, the JavaBeans pattern has serious disadvantages of its own. Because construction is split across multiple calls, a JavaBean may be in an

inconsistent state partway through its construction**.**

the telescoping constructor pattern works, but it is hard to write

client code when there are many parameters, and harder still to read it.



*Builder* pattern; Instead of making the desired object directly,

the client calls a constructor (or static factory) with all of the required parameters

and gets a *builder object*

Then the client calls setter-like methods on the builder

object to set each optional parameter of interest. Finally, the client calls a parameter less

build method to generate the object, which is immutable

NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8).

calories(100).sodium(35).carbohydrate(27).build();

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; }



the invalid parameters are passed, instead of waiting for build to be invoked.

A minor advantage of builders over constructors is that builders can have multiple

varargs parameters. Constructors, like methods, can have only one varargs

parameter. Because builders use separate methods to set each parameter, they can

have as many varargs parameters as you like, up to one per setter method.

A builder whose parameters have been set makes a fine *Abstract Factory*

In other words, a client can pass such a builder to a method to enable the method to create one or more objects for the client. To enable this usage, you need a type to represent the builder.

The traditional Abstract Factory implementation in Java has been the Class object, with the newInstance method playing the part of the build method

Problem with it :

The newInstance method always attempts to invoke the class’s parameterless constructor, which may not even exist. You don’t get a compile-time error if the class has no accessible parameterless constructor.

Instead, the client code must cope with InstantiationException or IllegalAccessException

at runtime, which is ugly and inconvenient

Also, the newInstance method propagates any exceptions thrown by the parameterless constructor, even though newInstance lacks the corresponding throws clauses. In other words, **Class.newInstance breaks compile-time exception checking.**

In summary, **the Builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters**, especially if most of those parameters are optional.

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

**Singleton pattern and serializable**

To maintain the singleton guarantee, you have to declare all instance fields transient and provide a readResolve method Otherwise, each time a serialized instance is deserialized, a new instance will be created, leading, in the case of our example, to spurious Elvis

sightings.

**// readResolve method to preserve singleton property**

private Object readResolve() {

// Return the one true Elvis and let the garbage collector

// take care of the Elvis impersonator.

return INSTANCE;

}

**Singleton and cloning ?**

No need to implement clonnable interface in singleton pattern.

2 approchs – public field vs privte field

**// Singleton with public final field**

public class Elvis {

**public static final Elvis INSTANCE = new Elvis();**

private Elvis() { ... }

a privileged client can invoke the private constructor reflectively (Item 53) with the aid of the AccessibleObject.setAccessible method. If you need to defend against this attack, modify the constructor to make it throw an exception if it’s 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; }

there is a third approach to implementing singletons. Simply make an enum type with one element:

a single-element enum type is the best way to implement a singleton. (need to explore more)

**4 Enforce noninstantiability with a private constructor**

1. For the classes which have static methods and static fields
2. **Attempting to enforce noninstantiability by making a class abstract does**

**not work.** The class can be subclassed and the subclass instantiated.

1. A default constructor is generated only if a class contains no explicit constructors, so **a class can be** **made noninstantiable by including a private constructor**:

**// Noninstantiable utility class**

public class UtilityClass {

**// Suppress default constructor for noninstantiability**

private UtilityClass() {

throw new AssertionError();

}

}

**\*\*\*5 Avoid creating unnecessary objects**

1. An object can always be reused if it is *immutable, re use can make our program faster (and stylish)and effiecient (consumes less memory)*

String s = new String("stringette") // do not do this

1. The statement creates a new String instance each time it is executed, and none of those object creations is necessary

String s = "stringette";

1. This version uses a single String instance, rather than creating a new one each time it is executed. Furthermore, it is guaranteed that the object will be reused by any other code running in the same virtual machine that happens to contain the same string literal
2. often avoid creating unnecessary objects by using *static factory methods* (Item 1) in preference to constructors on immutable classes
3. Whats wrng with this :
4. 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;}

1. Improved version :

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();

}

1. the original version takes 32,000 ms for 10 million invocations, while the improved version takes 130 ms
2. An adapter is an object that delegates to a backing object, providing an alternative interface to the backing object. Because an adapter has no state beyond that of its backing object, there’s no need to create more than one instance of a given adapter to a given object, meaning it should be singleton,
3. The keySet method of the Map interface returns a Set view of the Map object, consisting of all the keys in the map. it would seem that every call to keySet would have to create a new Set instance, but every call to keyset on a given Map object may return the same Set instance.
4. *Autoboxing*: a new way to create unnecessary objects in release 1.5.

*Co*nsider below

Long sum = 0L;

for (long i = 0; i < Integer.MAX\_VALUE; i++) {

sum += i;

}

System.out.println(sum);

**12**  variable sum is declared as a Long instead of a long, which means that the program constructs about 231 unnecessary Long instances. Changing the declaration of sum from Long to long reduces the runtime from 43 seconds to 6.8 seconds..

**13** prefer primitives to boxed primitives, and watch out for unintentional autoboxing.

**14** Avoiding object creation by maintaining your own *object pool* is a

bad idea unless the objects in the pool are extremely heavyweight. The classic

example of an object that *does* justify an object pool is a database connection. The

cost of establishing the connection is sufficiently high that it makes sense to reuse

these objects. Also, your database license may limit you to a fixed number of connections

**15**  however, maintaining your own object pools clutter your code, increases memory footprint, and harms performance. Modern JVM implementations have highly optimized garbage collectors that easily outperform such object pools on lightweight objects.

**Eliminate obsolete object references**

1. Whats wrong here :

public Object pop() {

if (size == 0)

throw new EmptyStackException();

return elements[--size]; }

1. If a stack grows and then shrinks, the objects that were popped off the stack will not be garbage collected, even if the program using the stack has no more references to them. This is because the stack maintains *obsolete references* to these objects.
2. An obsolete reference is simply a reference that will never be dereferenced again. In this case, any references outside of the “active portion” of the element array are obsolete. The active portion consists of the elements whose index is less than size.
3. Fix :

public Object pop() {

if (size == 0)

throw new EmptyStackException();

Object result = elements[--size];

**elements[size] = null; // Eliminate obsolete reference**

return result;

}

1. **Nulling out object references should be the exception rather than the norm.** The best way to eliminate an obsolete reference is to let the variable that contained the reference fall out of scope. This occurs naturally if you define each variable in the narrowest possible scope
2. *manages its own memory*. The *storage pool* consists of the elements of the elements array (the object reference cells, not the objects themselves). The elements in the active portion of the array (as defined earlier) are *allocated*, and those in the remainder of the array are *free*. The garbage collector has no way of knowing this; to the garbage collector,all of the object references in the elements array are equally valid
3. **Another common source of memory leaks is caches.**
4. If u r lucky enough to implement a cache for which an entry is relevant exactly so long as there are references to its key outside of the cache, represent the cache as a WeakHashMap; entries will be removed automatically after they become obsolete. Remember that WeakHashMap is useful only if the desired lifetime of cache entries is determined by external references to the key, not the value.

<http://stackoverflow.com/questions/10599710/weakhashmap-example>

1. More commonly, the useful lifetime of a cache entry is less well defined, with entries becoming less valuable over time. Under these circumstances, the cache should occasionally be cleansed of entries that have fallen into disuse. This can be done by a background thread (perhaps a Timer or ScheduledThreadPoolExecutor) or as a side effect of adding new entries to the cache.
2. The LinkedHashMap class facilitates the latter approach with its removeEldestEntry method. For more sophisticated caches, you may need to use java.lang.ref directly

1. **A third common source of memory leaks is listeners and other callbacks.** If you implement an API where clients register callbacks but don’t deregister them explicitly, they will accumulate unless you take some action. The best way to ensure that callbacks are garbage collected promptly is to store only *weak references* to them, for instance, by storing them only as keys in a WeakHashMap.
2. **Avoid finalizers (GTK)**
3. It can take arbitrarily long between the time that an object becomes unreachable and the time that its finalizer is executed.It means don’t do anything imp\time-critical in finalizers, like closing the file.
4. The promptness with which finalizers are executed is primarily a function of the garbage collection algorithm, which varies widely from JVM implementation to JVM implementation.
5. Don’t be seduced by the methods System.gc and System.runFinalization. They may increase the odds of finalizers getting executed, but they don’t guarantee it.
6. The only methods that claim to guarantee finalization are System.runFinalizersOnExit and Runtime.runFinalizersOnExit. These methods are fatally flawed and have been deprecated
7. if an uncaught exception is thrown during finalization, the exception is ignored, and finalization of that object terminates
8. Normally, an uncaught exception will terminate the thread and print a stack

trace, but not if it occurs in a finalizer—it won’t even print a warning.

1. there is a *severe* performance penalty for using finalizers. It slows down the system , the time to create and destroy a simple object is about5.6 ns. Adding a finalizer increases the time to 2,400 ns
2. It is better to provide a provide an *explicit termination method*, and require clients of the class to invoke this method on each instance when it is no longer needed then use finalizers
3. Typical examples of explicit termination methods are the close methods on InputStream, OutputStream, and java.sql.Connection, cancel method on java.util.Timer,
4. These methods are typically used in combination with the try-finally construct to ensure termination

Finalizers advantages

1. 1st is to act as a “safety net” in case the owner of an object forgets to call its

explicit termination method.

1. **the finalizer should log a warning if it finds that the resource has not been**

**terminated**, as this indicates a bug in the client code,

1. The four classes cited as examples of the explicit termination method pattern (FileInputStream, FileOutputStream, Timer, and Connection) have finalizers that serve as safety nets in case their termination methods aren’t called. But these classes do not log the error .
2. It can be use to to terminate noncritical native resources : A second legitimate use of finalizers concerns objects with *native peers*. A native peer is a native object to which a normal object delegates via native methods. Because a native peer is not a normal object, the garbage collector doesn’t know about it and can’t reclaim it when its Java peer is reclaimed. A finalizer is an appropriate vehicle for performing this task, *assuming the native peer holds no critical resources*.
3. If the native peer holds resources that must be terminated promptly, the class should have an explicit termination method.
4. If a class (other than Object) has a finalizer and a subclass overrides it, the subclass finalizer must invoke the superclass finalizer manually
5. You should finalize the subclass in a try block and invoke the superclass finalizer in the corresponding finally block. This ensures that the superclass finalizer gets executed even if the subclass finalization throws an exception and vice versa

@Override protected void finalize() throws Throwable {

try {

... // Finalize subclass state

} finally {

**super.finalize();**

}

}

1. If a subclass implementor overrides a superclass finalizer but forgets to invoke it, the superclass finalizer will never be invoked.
2. Gurdian :

1. **Obey the general contract when overriding equals**

When to override equal : When 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 behaviour

1. Whats wrong with this :

@Override public boolean equals(Object o) {

if (o instanceof CaseInsensitiveString)

return s.equalsIgnoreCase(

((CaseInsensitiveString) o).s);

**if (o instanceof String) // One-way interoperability!**

**return s.equalsIgnoreCase((String) o);**

return false;

}

CaseInsensitiveString cis = new CaseInsensitiveString("Polish");

String s = "polish";

Cis.equals(s) will return but s.equals(cis) may return false, can throw exception .

1. **Transitivity , A = B , B=C then A=C**

**How to u compare 2 Colorpoint**

public class Point {

private final int x;

private final int y;

public Point(int x, int y) {

this.x = x;

this.y = y;

}

@Override public boolean equals(Object o) {

if (!(o instanceof Point))

return false;

Point p = (Point)o;

return p.x == x && p.y == y;

}

public class ColorPoint extends Point {

private final Color color;

public ColorPoint(int x, int y, Color color) {

super(x, y);

this.color = color;

}

Issue with this approach is : It breaks symmetry , How ?

Point p = new Point(1, 2);

ColorPoint cp = new ColorPoint(1, 2, Color.RED);

cp.equals(p) will return false.

Solution 1 :

@Override public boolean equals(Object o) {

if (!(o instanceof ColorPoint))

return false;

**return super.equals(o) && ((ColorPoint) o).color == color;**

}

Solution 2:

Issue with this approach is : It breaks transvity . How ?

Point p = new Point(1, 2);

ColorPoint cp = new ColorPoint(1, 2, Color.RED);

ColorPoint cp2 = new ColorPoint(1, 2, Color.Blue);

cp.equals(p) will return false.

p.equals(cp), true, p.equals(cp2); but cp.equals(cp2)- false

@Override public boolean equals(Object o) {

if (!(o instanceof Point))

return false;

**// If o is a normal Point, do a color-blind comparison**

**if (!(o instanceof ColorPoint))**

**return o.equals(this);**

// o is a ColorPoint; do a full comparison

return super.equals(o) && ((ColorPoint)o).color == color;

}

*There is no way to extend an instantiable class and add a value component while preserving the equals contract.*

**Solution**

**Favour composition over Inheritance**

Instead of having ColorPoint extend Point, give ColorPoint a private Point field

public class ColorPoint {

private final Point point;

private final Color color;

@Override public boolean equals(Object o) {

if (!(o instanceof ColorPoint))

return false;

ColorPoint cp = (ColorPoint) o;

return cp.point.equals(point) && cp.color.equals(color);

}

Note that above problem will not occur if base class is abstract. For example, you could have an abstract class Shape with no

value components, a subclass Circle that adds a radius field

No need to check for null explicitly

Don’t do below

@Override public boolean equals(Object o) {

if (o == null)

return false;

...

}

Do following

@Override public boolean equals(Object o) {

if (!(o instanceof MyType))

return false;

MyType mt = (MyType) o;

...

}

here’s a recipe for a high-quality equals method:

1. Use the == operator to check if the argument is a reference to this object. This is just a performance optimization, but one that is worth doing if the comparison is potentially expensive.
2. Use the instanceof operator to check if the argument has the correct type.If not, return false.
3. Cast the argument to the correct type. Because this cast was preceded by an

instanceof test, it is guaranteed to succeed.

1. Check on fields :
2. Use following to avoid NPE on field checking

(field == null ? o.field == null : field.equals(o.field))

1. For primitive fields whose type is not float or double, use the == operator for comparisons. for float fields, use the Float.compare method; and for double fields, use Double.compare.
2. When you are finished writing your equals method, ask yourself three questions: Is it symmetric? Is it transitive? Is it consistent?

**Always override hashCode when you override equals**

It simply presents what will happen if u don’t override hashcode. Did not find any thing new.

**// The worst possible legal hash function - never use!**

@Override public int hashCode() { return 42; }

**11 Override clone judiciously**

1. Clonable does not have clone method so it fails in achieving its objective.
2. if a class implements Cloneable, Object’s clone method returns a field-by-field copy of the object; otherwise it throws CloneNotSupportedException.
3. provision that “no constructors are called” is too strong. A well-behaved clone method can call constructors to create objects internal to the clone under construction. If the class is final, clone can even return an object created by a constructor
4. for x.clone().getClass() == x.getClass() to work in a inheritance model, super class should return an object obtained by calling super.clone
5. if you override the clone method in a nonfinal class, you should return an object obtained by invoking super.clone.
6. Cloning is easy in if object contains primitive types or immutable fields then :

Notice it is example of shallow cloning and method returns Phonenumber not an object as covariants are allowed post 1.5

@Override public PhoneNumber clone() {

try {

return (PhoneNumber) super.clone();

} catch(CloneNotSupportedException e) {

throw new AssertionError(); // Can't happen

}}

1. Clone method of stack class :

public class Stack {

private Object[] elements;

private int size = 0;

private static final int DEFAULT\_INITIAL\_CAPACITY = 16;

public Stack() {

this.elements = new Object[DEFAULT\_INITIAL\_CAPACITY];

}

If its clone method merely returns super.clone(), the resulting Stack instance will have the correct value in size but its elements field will refer to the same array as the original Stack instance. Modifying the original will destroy the invariants in the clone and

vice versa.

1. In effect, the clone method functions as another constructor; you must ensure that it does no harm to the original object and that it properly establishes invariants on the clone.
2. How to use clone in stack class :

@Override public Stack clone() {

try {

Stack result = (Stack) super.clone();

result.elements = elements.clone();

return result;

} catch (CloneNotSupportedException e) {

throw new AssertionError();

}

}

1. the above solution would not work if the elements field were final, because clone would be prohibited from assigning a new value to the field. This is a fundamental problem: the clone architecture is incompatible with normal use of **final** fields referring to mutable object
2. Eg to practice in : E:\vob\vobSnH\CoreJava\ clone package

Q. If I don’t implement clone method in my class and call myObj.clone then ?

A. CloneNotSupported

Q. If I implement clone method but do not override clone , will it work ?

Yes, it will be shallow cloning

Q. If my superclass don’t implement clone but my subclass does it throw exception?

Nops.

Q. If my superclass implement clone but my subclass does it throw exception?

Nops.

SalariedEmployee se = **new** SalariedEmployee("PJ", **new** Department("IT"),"500000");

SalariedEmployee se2 = (SalariedEmployee)se.clone();

1. Deep cloning in hash table eg:

**Incorrect way : (whats wrong)**

@Override public HashTable clone() {

try {

HashTable result = (HashTable) super.clone();

result.buckets = buckets.clone();

return result;

} catch (CloneNotSupportedException e) {

throw new AssertionError();

}

**Correct way**

Entry deepCopy() {

Entry result = new Entry(key, value, next);

for (Entry p = result; p.next != null; p = p.next)

p.next = new Entry(p.next.key, p.next.value, p.next.next);

return result;

}

Or

**// Recursively copy the linked list headed by this Entry**

Entry deepCopy() {

return new Entry(key, value,

next == null ? null : next.deepCopy());

}

@Override public HashTable clone() {

try {

HashTable result = (HashTable) super.clone();

result.buckets = new Entry[buckets.length];

for (int i = 0; i < buckets.length; i++)

if (buckets[i] != null)

result.buckets[i] = buckets[i].deepCopy();

return result;

} catch (CloneNotSupportedException e) {

throw new AssertionError();

}

1. If a class that is designed for inheritance overrides clone, the overriding method should mimic the behavior of Object.clone: it should be declared protected, it should be declared to throw CloneNotSupportedException, and the class should not implement Cloneable. This gives subclasses the freedom to implement Cloneable or not, just as if they extended Object directly.
2. The above is a suggestion not rule.
3. a clone method should not invoke any nonfinal methods on the clone under construction .

Summary

classes that implement Cloneable should override clone with a

public method whose return type is the class itself. This method should first call

super.clone and then fix any fields that need to be fixed. Typically, this means

copying any mutable objects that comprise the internal “deep structure” of the

object being cloned, and replacing the clone’s references to these objects with references

to the copies. While these internal copies can generally be made by calling

clone recursively, this is not always the best approach. If the class contains

only primitive fields or references to immutable objects, then it is probably the

case that no fields need to be fixed.

**Singleton and cloning**

**No need to implement clonnable interface in singleton pattern.**

**Immutable and cloning**

it doesn’t make sense for immutable classes to support object copying, because copies

would be virtually indistinguishable from the original.

1. A fine approach to object copying is to provide a *copy constructor* or *copy factory*

**Copy constructor :** A copy constructor is simply a constructor that takes a single argument

whose type is the class containing the constructor, for example,

public Yum(Yum yum);

A copy factory is the static factory analog of a copy constructor:

public static Yum newInstance(Yum yum);

a copy constructor or factory can take an argument whose type is an interface implemented by the class. For example, by convention all generalpurpose collection implementations provide a constructor whose argument is of type Collection or Map.

These are also know as *conversion constructors* and *conversion factories*

you have a HashSet s,

and you want to copy it as a TreeSet. The clone method can’t offer this functionality,

but it’s easy with a conversion constructor: new TreeSet(s).

1. Cloning through Serialization : If you have a object having multiple attributes then you can use serialization to achieve cloning.

**12 Consider implementing Comparabler**

1. Comparable obey the same restrictions imposed by the equals contract: reflexivity, symmetry, and transitivity. Therefore the same caveat applies: there is no way to extend an instantiable class with a new value component while preserving the compareTo contract, unless you are willing to forgo the benefits of object-oriented abstraction.
2. If you want to add a value component to a class that implements Comparable, don’t

extend it; write an unrelated class containing an instance of the first class

1. consider the BigDecimal class, whose compareTo method is inconsistent with equals. If you create a HashSet instance and add new BigDecimal("1.0") and new BigDecimal("1.00"), the set will contain two elements because the two BigDecimal instances added to the set are unequal when compared using the equals method. If, however, you perform the same procedure using a TreeSet instead of a HashSet, the set will contain only one element because the two BigDecimal instances are equal when compared using the compareTo method.
2. Will this work :

public int compareTo(PhoneNumber pn) {

// Compare area codes

int areaCodeDiff = areaCode - pn.areaCode;

if (areaCodeDiff != 0)

return areaCodeDiff;

// Area codes are equal, compare prefixes

int prefixDiff = prefix - pn.prefix;

if (prefixDiff != 0)

return prefixDiff;

// Area codes and prefixes are equal, compare line numbers

return lineNumber - pn.lineNumber;

}

Yes , bcaz that the contract for compareTo does not specify the magnitude of the return value, only the sign.

xc

**13 Minimize the accessibility of classes and members**

1. The single most important factor that distinguishes a well-designed module from a poorly designed one is the degree to which the module hides its internal data and other implementation details from other modules.
2. A well-designed module hides all of its implementation details, cleanly separating its API from its implementation. Modules then communicate only through their APIs and are oblivious to each others’ inner workings. This concept, known as *information hiding* or *encapsulation*,
3. Encapsulation may help in performance optimization, as module can be building isolation, independent of each other.
4. make each class or member as inaccessible as possible.
5. To facilitate testing, you may be tempted to make a class, interface, or member more accessible. This is fine up to a point. It is acceptable to make a private member of a public class package-private in order to test it, but it is not acceptable to raise the accessibility any higher than that. In other words, it is not acceptable to make a class, interface, or member a part of a package’s exported API to facilitate testing.
6. If a class can be made package-private, it should be. By making it package-private, you make it part of the implementation rather than the exported API, and you can modify it, replace it, or eliminate it in a subsequent release without fear of harming existing clients.
7. Make instance fileds final, make it pkg provate only if another class in the pkg wants to access it.
8. Instance fields should never be public, if a field is non final public or final having reference to mutable obj, then wegive up the ability to limit the values that can be stored in the field.
9. it is wrong for a class to have a public static final array field, or an accessor that returns such a field.

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**

**15 Minimize mutability**

1. Java platform libraries contain many immutable classes, including String, the boxed primitive classes, and BigInteger and BigDecimal.
2. Immutable classes are easier to design, implement, and use than mutable classes. They are less prone to error and are more secure.
3. To make a class immutable, follow these five rules:
4. Don.t provide setter\mutator or method which changes object state
5. Ensure clas can not be extended, this can be done by making either class final or by making contrustor private and giving a static factory method to it.
6. Make fields final
7. Make instance field private
8. **Ensure exclusive access to any mutable components :** If your class has any fields that refer to mutable objects, ensure that clients of the class cannot obtain

references to these objects. Don’t return it from accessor, make its defensive copy in constructor, getter method and readObject method,

1. Example of how to make immutable class

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);

}

*The above approach is called functional* approach because methods return the result of applying a function to their operand without modifying it, which enables immutability.

Against *procedural* or *imperative* approach in which methods apply a procedure to their operand,causing its state to change.

1. Immutable objects remain in same state, state in which they were created.
2. Immutable objects are inherently thread-safe; they require no synchronization**.** They cannot be corrupted by multiple threads accessing them concurrently. This is far and away the easiest approach to achieving thread safety.
3. They can be shared freely\safly which can increase reusability.
4. A consequence of the fact that immutable objects can be shared freely is that you never have to make *defensive copies.*
5. you need not and should not provide a clone method or *copy constructor* (Item 11) on an immutable class. This was not well understood in the early days of the Java platform, so the String class does have a copy constructor, but it should rarely, if ever, be used.
6. that immutable objects make great map keys and set elements: you don’t have to worry about their values changing once they’re in the map or set,
7. **The only real disadvantage of immutable classes is that they require a**

**separate object for each distinct value, which can be costly.**

1. The alternative to making an immutable class final is to make all of its constructors private or package-private, and to add public *static factories* in place of the public constructors.
2. Eg:

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);

}

1. To its clients that reside outside its package, the immutable class is effectively final because it is impossible to extend a class that comes from another package and that lacks a public or protected constructor.
2. If you choose to have your immutable class implement Serializable and it contains one or more fields that refer to mutable objects, you must provide an explicit readObject orreadResolve method, or use the ObjectOutputStream.writeUnshared and ObjectInputStream.readUnshared methods, even if the default serialized form is acceptable. Otherwise an attacker could create a mutable instance of your notquite-

immutable class.

1. **If a class cannot be made immutable, limit its mutability as much as possible.**
2. **make every field final unless there is a compelling reason to make it nonfinal.**

**16 Favour composition over inheritance**

1. Unlike method invocation, inheritance violates encapsulation : a subclass depends on the implementation details of its superclass for its proper function. A change in super class may break sub class. (until subclass is designed and coded for extension)
2. Example

public class InstrumentedHashSet<E> extends HashSet<E> {

* 1. // The number of attempted

@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);

* 1. }
  2. What will it return : 3 or 6

InstrumentedHashSet<String> s =new InstrumentedHashSet<String>();

s.addAll(Arrays.asList("Snap", "Crackle", "Pop"));

1. A related cause of fragility in subclasses is that their superclass can acquire

new methods in subsequent releases. Predcate and collection wala example

1. Suppose a program depends for its security on the fact that all elements inserted into some collection satisfy some predicate. This can be guaranteed by subclassing the collection and overriding each method capable of adding an element to ensure that the predicate is satisfied before adding the element. This works fine until a new method capable of inserting an element is added to the superclass in a subsequent release. Once this happens, it becomes possible to add an “illegal” element merely by invoking the new method, which is not overridden in the subclass.
2. You might think that it is safe to extend a class if you merely add new methods and refrain from overriding existing methods. While this sort of extension is much safer, it is not without risk.
3. If the superclass acquires a new method in a subsequent release and you have the bad luck to have given the subclass a method with the same signature and a different return type, your subclass will no longer compile, and in case the super class has same return type then ?
4. Then we will be overriding it, without thinking about the contract, as it was not there when we have write the method in the sub class.
5. Instead of extending an existing class, give your new class a private field that references

an instance of the existing class. This design is called *composition* because the

existing class becomes a component of the new one.

1. InstrumentedSet class is known as a *wrapper* class because each InstrumentedSet instance contains (“wraps”) another Set instance. This is also known as the *Decorator* pattern because the Instrumented-Set class “decorates” a set by adding instrumentation.
2. Inheritance is appropriate only in circumstances where the subclass really is a

*subtype* of the superclass. In other words, a class *B* should extend a class *A* only if

an “is-a” relationship exists between the two classes.

1. There are a number of obvious violations of this principle in the Java platform

libraries. For example, a stack is not a vector, so Stack should not extend Vector.

Similarly, a property list is not a hash table, so Properties should not extend

Hashtable. In both cases, composition would have been preferable.

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

1. Constructors must not invoke overridable methods, directly or indirectly The superclass constructor runs before the subclass constructor, so the overriding method in the subclass will get invoked before the subclass constructor has run.
2. Example :

public class Super {

**// Broken - constructor invokes an overridable method**

public Super() {

overrideMe();

}

public void overrideMe() {

}

}

public final class Sub extends Super {

private final Date date; // Blank final, set by constructor

Sub() {

date = new Date();

}

**// Overriding method invoked by superclass constructor**

@Override public void overrideMe() {

System.out.println(date);

}

public static void main(String[] args) {

Sub sub = new Sub();

sub.overrideMe();

}

}

What will be the output : it will be null ,followed by date.

1. **Implementing clonable and Seriliazable (Tough to get, need more info)**

It is generally not a good idea for a class designed for inheritance to implement either of these interfaces.

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

**indirectly**. In the case of the readObject method, the overriding method will run

before the subclass’s state has been deserialized. In the case of the clone method,

the overriding method will run before the subclass’s clone method has a chance to fix the clone’s state. In either case, a program failure is likely to follow. In the case

of clone, the failure can damage the original object as well as the clone. This can

happen, for example, if the overriding method assumes it is modifying the clone’s

copy of the object’s deep structure, but the copy hasn’t been made yet.

designing a class for inheritance places substantial limitations on the class. Each time a change is made in such a class, there is a chance that client classes that extend the class will break .

**18 Prefer interfaces to abstract classes (GTK)**

**Advantage of interfaces over Abstract classes**

1. **Existing classes can be easily retrofitted to implement a new interface :** For example, many existing classes were retrofitted to implement the Comparable interface. Existing classes cannot, in general, be retrofitted to extend a new abstract class.
2. **Interfaces allow the construction of nonhierarchical type frameworks :** it is perfectly permissible for a single class to implement both Singer and Songwriter. In fact, we can define a third interface that extends both Singer and Songwriter and adds new methods that are appropriate to the combination:
3. **Interfaces enable safe, powerful functionality enhancements** via the *wrapper class* idiom,
4. You can combine the virtues of interfaces and abstract classes by providing an abstract *skeletal implementation* class to go with each nontrivial interface that you export
5. AbstractCollection, AbstractSet, AbstractList, and AbstractMap are eg, of skeletal framework,
6. **It is far easier to evolve an abstract class than an interface.**
7. If, in a subsequent release, you want to add a new method to an abstract class, you can always add a concrete method containing a reasonable default implementation. abstract class, you can always add a concrete method containing a reasonable default implementation.
8. **Once an interface is released and widely implemented, it is almost impossible to change.** You really must get it right the first time

**19 Use interfaces only to define types (GTK)**

Do not use them to define constant, use enum or utility classes to define const

20 **Prefer class hierarchies to tagged classes**

Occasionally you may run across a class whose instances come in two or more flavours and contain a *tag* field indicating the flavour of the instance. Example a singale class which represent both circle and rectangle. They hav lots of shortcoming, instead prefer class hierarchy,

**21 Use function objects to represent strategies**

Different sort orders can be obtained by passing in different comparator functions.

This is an example of the *Strategy* pattern, so comprator is example of strategy pattern.

To implement this pattern in Java, declare an interface to represent the

strategy, and a class that implements this interface for each concrete strategy.

When a concrete strategy is used only once, it is typically declared and instantiated

as an anonymous class. When a concrete strategy is designed for repeated

use, it is generally implemented as a private static member class and exported in a

public static final field whose type is the strategy interface

function objects – objects having only a single function , no parameter, so no state.

**22 Favor static member classes over nonstatic**

**Static member class**

1. A static member class is a static member of its enclosing class and obeys the same

accessibility rules as other static members.

1. If an instance of a nested class can exist in isolation from an instance of its enclosing

class, then the nested class *must* be a static member class:

1. A common use of private static member classes is to represent components of the object represented by their enclosing class.
2. For example : Many Map implementations have an internal Entry object for each key-value pair in the map. While each entry is associated with a map, the methods on an entry (getKey, getValue, and setValue) do not need access to the map. Therefore, it would be wasteful to use a nonstatic member class to represent entries: a private static member class is best

**nonstatic member class**

1. nonstatic member class is implicitly associated with an *enclosing instance* of its containing class
2. Within instance methods of a nonstatic member class, you can invoke methods on the enclosing instance or obtain a reference to the enclosing instance using the *qualified this* construct
3. The association between a nonstatic member class instance and its enclosing instance is established when the former is created;
4. If you declare a member class that does not require access to an enclosing instance, *always* put the static modifier in its declaration,
5. If you omit this modifier, each instance will have an extraneous reference to its enclosing instance. Storing this reference costs time and space, and can result in the enclosing instance being retained when it would otherwise be eligible for garbage collection
6. One common use of a nonstatic member class is to define an *Adapter.* that allows an instance of the outer class to be viewed as an instance of some unrelated class.
7. For example, implementations of the Map interface typically use nonstatic member classes to implement their *collection views*, which are returned by Map’s keySet, entrySet, and values methods.

Genrics

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

1. If you use raw types, you lose all the safety and expressiveness benefits of generics.
2. Issues if we don’t use generics

private final **Collection** stamps = ... ;

stamps.add(new Coin( ... ));

**// Now a raw iterator type - don't do this!**

for (**Iterator** i = stamps.iterator(); i.hasNext(); ) {

Stamp s = **(Stamp)** i.next(); **// Throws ClassCastException**

... // Do something with the stamp

}

**With Genrics**

**// for-each loop over a parameterized collection - typesafe**

for (Stamp s : stamps) { **// No cast**

... // Do something with the stamp

}

1. Just what is the difference between the raw type List and the parameterized type List<Object>?
2. Loosely speaking, the former has opted out of generic type checking, while the latter has explicitly told the compiler that it is capable of holding objects of any type.
3. While you can pass a List<String> to a parameter of type List, you can’t pass it to a parameter of type List<Object>.
4. What is the difference between the unbounded wildcard type Set<?> and the

raw type Set

1. **you can’t put any element other than null) into a Collection<?> Below will not compile**

**private** **static** **void** unsafeAdd(List <?> list, String o) {

list.add(o);

}

1. Safe vs Unsafe

|  |  |
| --- | --- |
| UnSafe | Safe |
| **// Use of raw type for unknown element type - don't do this!**  static int numElementsInCommon(Set s1, Set s2) {  int result = 0;  for (Object o1 : s1)  if (s2.contains(o1))  result++;  return result;  } | **// Unbounded wildcard type - typesafe and flexible**  static int numElementsInCommon(Set**<?>** s1, Set**<?>** s2) {  int result = 0;  for (Object o1 : s1)  if (s2.contains(o1))  result++;  return result;  } |

|  |  |
| --- | --- |
| UnBounded | Genric Type |
| static int numElementsInCommon(Set**<?>** s1, Set**<?>** s2) {  int result = 0;  for (Object o1 : s1)  if (s2.contains(o1))  result++;  return result;  } | **static** <E> **int** numElementsInCommon  (Set<E> s1, Set<E> s2) {  **int** result = 0;  **for** (E e : s1)  **if** (s2.contains(e))  result++;  **return** result;  } |

1. There are two minor exceptions to the rule that you should not use raw types in new code, both of which stem from the fact that generic type information is erased at runtime
2. **You must use raw types in class literals.** List.class, String[].class, and int.class are all legal, but List<String>.
3. The second exception to the rule concerns the instanceof operator. Because generic type information is erased at runtime, it is illegal to use the instanceof

operator on parameterized types

**// Legitimate use of raw type - instanceof operator**

if (o instanceof **Set**) { // Raw type

**Set<?>** m = **(Set<?>)** o; // Wildcard type

...

}

Q what is the difference between ? and object .

So what's the difference between:

<http://stackoverflow.com/questions/678822/what-is-the-difference-between-and-object-in-java-generics>

HashMap<String, ?> hash1;

and

HashMap<String, Object> hash2;

An instance of HashMap<String, String> matches Map<String, ?> but not Map<String, Object>

Say you want to write a method that accepts maps from Strings to anything: If you would write

public void foobar(Map<String, Object> ms) {

...

}

you can't supply a HashMap<String, String>. If you write

public void foobar(Map<String, ?> ms) {

...

}

**Eliminate unchecked warnings**

Nothing imp or useful

25 **Prefer lists to arrays**

1. Arrays differ from generic types in two important ways. First, arrays are *covariant*.
2. It means simply that if Sub is a subtype of Super, then the array type Sub[] is a subtype of Super[].
3. Generics, by contrast, are *invariant*: for any two distinct types Type1 and Type2, List<Type1> is neither a subtype nor a supertype of List<Type2>

**// Fails at runtime!**

Object[] objectArray = new Long[1];

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

but this one is not:

**// Won't compile!**

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

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

1. Either way you can’t put a String into a Long container, but with an array you find out that you’ve made a mistake at runtime; with a list, you find out at compile time
2. The second major difference between arrays and generics is that arrays are *reified.*
3. This means that arrays know and enforce their element types at runtime. If you try to store a String into an array of Long, you’ll get an ArrayStoreException
4. Generics, by contrast, are implemented by *erasure.*
5. This means that they enforce their type constraints only at compile time and discard (or *erase*) their element type information at runtime.
6. None of these array creation expressions are legal: new List<E>[], new List<String>[], new E[]. All will result in *generic array creation* errors at compile time.
7. Why is it illegal to create a generic array? Because it isn’t typesafe. If it were legal, casts generated by the compiler in an otherwise correct program could fail at runtime with a ClassCastException. This would violate the fundamental guarantee

provided by the generic type system.

1. In summary, arrays and generics have very different type rules. Arrays are covariant and reified; generics are invariant and erased. As a consequence, arrays provide runtime type safety but not compile-time type safety and vice versa forgenerics.

|  |  |
| --- | --- |
| **Array** | **List** |
| static <E> E reduce(List<E> list, Function<E> f, E initVal) {  E[] snapshot = list.toArray(); // Locks list  E result = initVal;  for (E e : snapshot)  result = f.apply(result, e);  return result;  } | **// List-based generic reduction**  static <E> E reduce(List<E> list, Function<E> f, E initVal) {  List<E> snapshot;  synchronized(list) {  snapshot = new ArrayList<E>(list);  }  E result = initVal;  for (E e : snapshot)  result = f.apply(result, e);  return result;  } |

**Item 26: Favor generic types**

**It is about stack program**

**// Initial attempt to generify Stack = won’t compile!**

public class Stack**<E>** {

private **E**[] elements;

public Stack() {

elements = new **E**[DEFAULT\_INITIAL\_CAPACITY];

}

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; // Eliminate obsolete reference

return result;

}

you can’t create an array of a non-reifiable type, such as E.

**So how to solve it**

1. elements = **(E[]) new Object[DEFAULT\_INITIAL\_CAPACITY]**;
2. change the type of the field elements from E[] to Object[]

**Item 27: Favor generic methods**

From

public static Set union(Set s1, Set s2) {

Set result = new HashSet(s1);

result.addAll(s2);

return result;

}

To

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

Set**<E>** result = new HashSet**<E>**(s1);

result.addAll(s2);

return result;

}

from

Map**<String, List<String>>** anagrams =

new HashMap**<String, List<String>>**();

to

Map<String, List<String>> anagrams = **newHashMap()**;

**29 Use bounded wildcards to increase API flexibilityv**

1. parameterized types are *invariant* , List<String> is not subtype of List<Object>.
2. **PECS stands for producer-extends, consumer-super.**

Its example is somewhat confusing and time-wasting

[**http://stackoverflow.com/questions/2723397/what-is-pecs-producer-extends-consumer-super**](http://stackoverflow.com/questions/2723397/what-is-pecs-producer-extends-consumer-super)

PECS" is from the collection's point of view. If you are only pulling items from a generic collection, it is a producer and you should use extends; if you are only stuffing items in, it is a consumer and you should use super. If you do both with the same collection, you shouldn't use either extends or super.

Suppose you have a method that takes as its parameter a collection of things, but you want it to be more flexible than just accepting a Collection<Thing>.

**Case 1: You want to go through the collection and do things with each item.**  
Then the list is a **producer**, so you should use a Collection<? extends Thing>.

The reasoning is that a Collection<? extends Thing> could hold any subtype of Thing, and thus each element will behave as a Thing when you perform your operation. (You actually cannot add anything to a Collection<? extends Thing>, because you cannot know at runtime which specific subtype of Thing the collection holds.)

**Case 2: You want to add things to the collection.**  
Then the list is a **consumer**, so you should use a Collection<? super Thing>.

The reasoning here is that unlike Collection<? extends Thing>, Collection<? super Thing> can always hold a Thing no matter what the actual parameterized type is. Here you don't care what is already in the list as long as it will allow a Thing to be added; this is what ? super Thingguarantees.

**class** A {}

**class** B **extends** A {}

**class** C **extends** B {}

**public** **void** testCoVariance(List<? **extends** B> myBlist) {

B b = **new** B();

C c = **new** C();

A a = **new** A();

myBlist.add(b); // does not compile

myBlist.add(c); // does not compile

A a = myBlist.get(0); // so can be used for producing

}

**public** **void** testContraVariance(List<? **super** B> myBlist) {

B b = **new** B();

C c = **new** C();

A a = **new** A();

myBlist.add(b);

myBlist.add(c);

myBlist.add(a);

A a = myBlist.get(0); // does not compile

}

**Item 29: Consider typesafe heterogeneous containers**

1. The most common use of generics is for collections, such as Set and Map, and single- element containers, such as ThreadLocal and AtomicReference.
2. class Class was generified in release 1.5. The type of a class literal is no longer simply Class, but Class<T>. For example, String.class is of type Class<String>, and Integer.class is of type Class<Integer>.

**In future try to again go through this chapter**

**30 Enums and Annotations**

1. **Use enums instead of int constants**
2. Java’s enum types are full-fledged classes
3. Enums are by their nature immutable
4. You cannot print String constant in sysout.

Why we don’t use new with enum bcaz

1. They are build as a replacement of constants
2. They represent constants
3. Enum types are effectively final, by virtue of having no accessible constructors. Because clients can neither create instances of an enum type nor extend it, there can be no instances but the declared enum constants.

In other words, enum types are instance-controlled (page 6). They are a generalization of singletons (Item 3), which are essentially single-element enums.

To associate data with enum constants, declare instance fields and write a constructor that takes the data and stores it in the fields.



*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;} };

abstract double apply(double x, double y);

**Excellent Nice Pay example**

**Problem Statement**

For example, consider an enum representing the days of the week in a payroll package. This enum has a method that calculates a worker’s pay for that day given the worker’s base salary

(per hour) and the number of hours worked on that day. On the five weekdays, any

time worked in excess of a normal shift generates overtime pay; on the two weekend

days, all work generates overtime pay.

Solution :

The idea is to move the overtime pay computation into a private nested enum, and to

pass an instance of this *strategy enum* to the constructor for the PayrollDay enum.

The PayrollDay enum then delegates the overtime pay calculation to the strategy

enum, eliminating the need for a switch statement or constant-specific method

**31 Use instance fields instead of ordinals**

Ordinals should not be used , nothing much

**32Use EnumSet instead of bit fields**

**33 Use EnumMap instead of ordinal indexing**

Enum ke ander ek enum, to address difficulties in managing them

1. Workers salary depending on day and extra time

enum PayrollDay {

MONDAY(PayType.WEEKDAY), TUESDAY(PayType.WEEKDAY),

WEDNESDAY(PayType.WEEKDAY), THURSDAY(PayType.WEEKDAY),

FRIDAY(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 hours <= HOURS\_PER\_SHIFT ? 0 :

(hours - HOURS\_PER\_SHIFT) \* payRate / 2;

}

},

WEEKEND {

double overtimePay(double hours, double payRate) {

return hours \* payRate / 2;

}

};

}

1. To reperents phase and their transition

public enum Phase {

SOLID, LIQUID, GAS;

public enum Transition {

MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),

BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),

SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);

final Phase src;

final Phase dst;

Transition(Phase src, Phase dst) {

this.src = src;

this.dst = dst;

}

}

**34 Emulate extensible enums with interfaces**

**while you cannot write an extensible enum type, you can emulate it by writing an interface to go with a basic enum type that implements the interface.** This allows clients to write their own enums that implementthe interface. These enums can then be used wherever the basic enum type can be

used, assuming APIs are written in terms of the interface.

**// Emulated extensible enum using an interface**

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("-") {

public double apply(double x, double y) { return x - y; }

},

TIMES("\*") {

public double apply(double x, double y) { return x \* y; }

},

DIVIDE("/") {

public double apply(double x, double y) { return x / y; }

};

**Prefer annotations to naming patterns**

the JUnit testing framework originally required its users to designate test

methods by beginning their names with the characters test

Naming pattern has many disadvantages compare to annotation

Example of annotation :

**// Program containing marker annotations**

public class Sample {

**@Test** public static void m1() { } // Test should pass

public static void m2() { }

**@Test** public static void m3() { // Test Should fail

throw new RuntimeException("Boom");

}

public static void m4() { }

**@Test** public void m5() { } **// INVALID USE: nonstatic method**

public static void m6() { }

**@Test** public static void m7() { // Test should fail

throw new RuntimeException("Crash");

}

public static void m8() { }

}

In sum, Sample contains four

tests: one will pass, two will fail, and one is invalid. The four methods that are not

annotated with the Test annotation will be ignored by the testing tool

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);

}

}

}

Now let’s add support for tests that succeed only if they throw a particular

exception. We’ll need a new annotation type for this:

**// Annotation type with a parameter**

import java.lang.annotation.\*;

/\*\*

\* Indicates that the annotated method is a test method that

\* must throw the designated exception to succeed.

\*/

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

public @interface ExceptionTest {

**Class<? extends Exception> value();**

}

The @Retention(RetentionPolicy.RUNTIME)

meta-annotation indicates that Test annotations should be retained at runtime.

Without it, Test annotations would be invisible to the test tool.

The @Target(ElementType.METHOD) meta-annotation indicates that the Test annotation is

legal only on method declarations: it cannot be applied to class declarations, field

declarations, or other program elements.

public @interface ExceptionTest {

Class<? extends Exception> value();

}

public class Sample2 {

**@ExceptionTest(ArithmeticException.class)**

public static void m1() { // Test should pass

int i = 0;

i = i / i;

}

**@ExceptionTest(ArithmeticException.class)**

public static void m2() { // Should fail (wrong exception)

int[] a = new int[0];

int i = a[1];

}

**@ExceptionTest(ArithmeticException.class)**

public static void m3() { } // Should fail (no exception)

}

it is possible to envision a test that passes if it throws any one of several specified exceptions

public @interface ExceptionTest {

Class<? extends Exception>**[]** value();

}

Example of the code :

@ExceptionTest(**{ IndexOutOfBoundsException.class,**

**NullPointerException.class }**)

public static void doublyBad() {

List<String> list = new ArrayList<String>();

// The spec permits this method to throw either

// IndexOutOfBoundsException or NullPointerException

list.addAll(5, null);

}

if (m.isAnnotationPresent(ExceptionTest.class)) {

tests++;

try {

m.invoke(null);

System.out.printf("Test %s failed: no exception%n", m);

} catch (Throwable wrappedExc) {

Throwable exc = wrappedExc.getCause();

Class<? extends Exception>**[]** excTypes =

m.getAnnotation(ExceptionTest.class).value();

**int oldPassed = passed;**

**for (Class<? extends Exception> excType : excTypes) {**

**if (excType.isInstance(exc)) {**

**passed++;**

**break;**

**}**

**}**

**if (passed == oldPassed)**

**System.out.printf("Test %s failed: %s %n", m, exc);**

}

}

**Consistently use the Override annotation**

public boolean equals(Bigram b) {

return b.first == first && b.second == second;

}

Whats wrong here

@Override

public boolean equals(Bigram b) {

return b.first == first && b.second == second;

}

@Override public boolean equals(**Object o**) {

**if (!(o instanceof Bigram))**

**return false;**

**Bigram b = (Bigram) o;**

return b.first == first && b.second == second;

}

**Use marker interfaces to define types**

By implementing Serializable interface, a class indicates that its instances can be written

to an ObjectOutputStream

when should you use a marker annotation and when should you use a

marker interface? Clearly you must use an annotation if the marker applies to any

program element other than a class or interface, as only classes and interfaces can

be made to implement or extend an interface. If the marker applies only to classes

and interfaces, ask yourself the question, Might I want to write one or more methods

that accept only objects that have this marking? If so, you should use a marker

interface in preference to an annotation**.**

**Annotation tutorial**

**Built-in Java Annotations**

Java comes with three built-in annotations which are used to give the Java compiler instructions. These annotations are:

* @Deprecated
* @Override
* @SuppressWarnings

Suppose that a software group traditionally starts the body of every class with comments providing important information:

public class Generation3List extends Generation2List {

// Author: John Doe

// Date: 3/17/2002

// Current revision: 6

// Last modified: 4/12/2004

// By: Jane Doe

// Reviewers: Alice, Bill, Cindy

// class code goes here

}

To add this same metadata with an annotation, you must first define the *annotation type*. The syntax for doing this is:

@interface ClassPreamble {

String author();

String date();

int currentRevision() default 1;

String lastModified() default "N/A";

String lastModifiedBy() default "N/A";

// Note use of array

String[] reviewers();

}

@ClassPreamble (

author = "John Doe",

date = "3/17/2002",

currentRevision = 6,

lastModified = "4/12/2004",

lastModifiedBy = "Jane Doe",

// Note array notation

reviewers = {"Alice", "Bob", "Cindy"}

)

public class Generation3List extends Generation2List {

## Creating Your Own Annotations

|  |  |
| --- | --- |
| @interface MyAnnotation {  String value();  String name();  int age();  String[] newNames();  } | @MyAnnotation(  value="123",  name="Jakob",  age=37,  newNames={"Jenkov", "Peterson"}  )  public class MyClass {  } |
| @interface MyAnnotation {  String value() default "";  String name();  int age();  String[] newNames();  } | @MyAnnotation(  name="Jakob",  age=37,  newNames={"Jenkov", "Peterson"}  )  public class MyClass {  } |

### @Retention

### You can specify for your custom annotation if it should be available at runtime, for inspection via reflection.

@Retention(RetentionPolicy.RUNTIME)

@interface MyAnnotation {

String value() default "";

}

### @Target

You can specify which Java elements your custom annotation can be used to annotate. You do so by annotating your annotation definition with the @Target annotation. Here is a @Target Java annotation example:

import java.lang.annotation.ElementType;

import java.lang.annotation.Target;

@Target({ElementType.METHOD})

public @interface MyAnnotation {

String value();

}

@Target(ElementType.TYPE) means that the annotation can only be used ontop of types (classes and interfaces typically).

**Class Annotations**

You can access the annotations of a class, method or field at runtime. Here is an example that accesses the class annotations:

Class aClass = TheClass.class;

Annotation[] annotations = aClass.getAnnotations();

for(Annotation annotation : annotations){

if(annotation instanceof MyAnnotation){

MyAnnotation myAnnotation = (MyAnnotation) annotation;

System.out.println("name: " + myAnnotation.name());

System.out.println("value: " + myAnnotation.value());

}

}

**Method Annotations**

Here is an example of a method with annotations:

public class TheClass {

@MyAnnotation(name="someName", value = "Hello World")

public void doSomething(){}

}

You can access method annotations like this:

Method method = ... //obtain method object

Annotation[] annotations = method.getDeclaredAnnotations();

for(Annotation annotation : annotations){

if(annotation instanceof MyAnnotation){

MyAnnotation myAnnotation = (MyAnnotation) annotation;

System.out.println("name: " + myAnnotation.name());

System.out.println("value: " + myAnnotation.value());

}

}

**Parameter Annotations**

It is possible to add annotations to method parameter declarations too. Here is how that looks:

public class TheClass {

public static void doSomethingElse(

@MyAnnotation(name="aName", value="aValue") String parameter){

}

}

You can access parameter annotations from the Method object like this:

Method method = ... //obtain method object

Annotation[][] parameterAnnotations = method.getParameterAnnotations();

Class[] parameterTypes = method.getParameterTypes();

int i=0;

for(Annotation[] annotations : parameterAnnotations){

Class parameterType = parameterTypes[i++];

for(Annotation annotation : annotations){

if(annotation instanceof MyAnnotation){

MyAnnotation myAnnotation = (MyAnnotation) annotation;

System.out.println("param: " + parameterType.getName());

System.out.println("name : " + myAnnotation.name());

System.out.println("value: " + myAnnotation.value());

}

}

}

**39 Make defensive copies when needed**

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

public Period(Date start, Date end) {

if (start.compareTo(end) > 0)

throw new IllegalArgumentException(

start + " after " + end);

this.start = start;

this.end = end;

}

public Date start() {

return start;

}

Date start = new Date();

Date end = new Date();

Period p = new Period(start, end);

end.setYear(78);

1. How to fix above

**// Repaired constructor - makes defensive copies of parameters**

public Period(Date start, Date end) {

this.start = new Date(start.getTime());

this.end = new Date(end.getTime());

if (this.start.compareTo(this.end) > 0)

throw new IllegalArgumentException(start +" after "+ end);

}

1. Why not to use cloning

Because Date is nonfinal, the clone method is not guaranteed to return an object whose class is java.util.Date: it could return an instance of an untrusted subclass specifically designed for malicious mischief. Such a subclass could, for example, record a reference to each instance in a private static list at the time of its creation and allow the attacker to access this list. This would give the attacker free reign over all instances. To prevent this sort of attack, **do not use the clone** **method to make a defensive copy of a parameter whose type is subclassable** **by untrusted parties**

1. Second attack

Date start = new Date();

Date end = new Date();

Period p = new Period(start, end);

p.end().setYear(78);

public Date end() {

return new Date(end.getTime());

}

1. if you are considering using a client-provided object reference as an element in an internal Set instance or as a key in an internal Map instance, you should be aware that the invariants of the set or map would be destroyed if the object were modified after it is inserted
2. Remember that nonzero-length arrays are always mutable. Therefore, you should always make a defensive copy of an internal array before returning it to a client.
3. Defensive copying can have a performance penalty associated with it and isn’t always justified. If a class trusts its caller not to modify an internal component, perhaps because the class and its client are both part of the same package, then it

may be appropriate to dispense with defensive copying.

**40 Design method signatures carefully**

Did not find any thing very useful

**41 Use overloading judiciously**

1. selection among overloaded methods is static, while selection among overridden methods is dynamic.
2. What will be the out put here

public class CollectionClassifier {

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));

}

}

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";

}

1. How java Api ha avoided overriding : consider the class ObjectOutputStream. It has a variant of ita write method for every primitive type and for several reference types. Rather than overloading the write method, these variants have signatures like writeBoolean( boolean), writeInt(int), and writeLong(long)
2. What will be the output here :

The out pust was shking , not becuse of overloading, but because of resizing in the list ,

-2,0,2 for the linked list

public static void main(String[] args) {

Set<Integer> set = new TreeSet<Integer>();

List<Integer> list = new ArrayList<Integer>();

for (int i = -3; i < 3; i++) {

set.add(i);

list.add(i);

}

for (int i = 0; i < 3; i++) {

set.remove(i);

list.remove(i);

}

System.out.println(set + " " + list);

}

**Item 42: Use varargs judiciously**

1. Simple use of varargs

static int sum(int... args) {

int sum = 0;

for (int arg : args)

sum += arg;

return sum;}

1. The WRONG way to use varargs to pass one or more arguments!

static int min(int... args) {

if (args.length == 0)

throw new IllegalArgumentException("Too few arguments");

int min = args[0];

for (int i = 1; i < args.length; i++)

if (args[i] < min)

min = args[i];

return min;}

1. Correct ways of using it :

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;

}

1. Exercise care when using the varargs facility in performance-critical situations.Every invocation of a varargs method causes an array allocation and initialization.
2. Suppose you’ve determined that 95 percent of the calls to a method have three or fewer parameters. Then declare five overloadings of the method, one each with zero through three ordinary parameters, and a single varargs method for use when the number of arguments exceeds three:

public void foo() { }

public void foo(int a1) { }

public void foo(int a1, int a2) { }

public void foo(int a1, int a2, int a3) { }

public void foo(int a1, int a2, int a3, int... rest) { }

**43 Return empty arrays or collections, not nulls**

1. The incorrect way of returning array :

private final List<Cheese> cheesesInStock = ...;

public Cheese[] getCheeses() {

if (cheesesInStock.size() == 0)

return null;

}

1. So caller\client of this method has to handle null while calling this method :

Cheese[] cheeses = shop.getCheeses();

if (cheeses != null &&

Arrays.asList(cheeses).contains(Cheese.STILTON))

System.out.println("Jolly good, just the thing.");

instead of:

if (Arrays.asList(shop.getCheeses()).contains(Cheese.STILTON))

System.out.println("Jolly good, just the thing.");

1. returning null in place of an empty array also complicates the method that returns the array or collection.
2. It is sometimes argued that a null return value is preferable to an empty array because it avoids the expense of allocating the array.
3. It fals becaz unless ur profiling tool does not give an error u cann’t say so and

Empty arrays are immutable objects so method may return the same immutable object

1. 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];

public Cheese[] getCheeses() {

return cheesesInStock.toArray(EMPTY\_CHEESE\_ARRAY);

}

7) a collection-valued method can be made to return the same immutable empty collection every time it needs to return an empty collection. The Collections.emptySet, emptyList, and emptyMap methods provide exactly the same.

1. 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 gTk**

**45 Minimize the scope of local variables**

1. The most powerful technique for minimizing the scope of a local variable

is to declare it where it is first used.

1. Declaring a local variable prematurely can cause its scope not only to extend

too early, but also to end too late. The scope of a local variable extends from the

point where it is declared to the end of the enclosing block. If a variable is

declared outside of the block in which it is used, it remains visible after the program

exits that block. If a variable is used accidentally before or after its region of intended use, the consequences can be disastrous.

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

Can not find any thing new

**47 Know and use the libraries**

every programmer should be familiar with the contents of java.lang, java.util, and, to a lesser extent, java.io.

**48 Avoid float and double if exact answers are required**

The float and double types are particularly illsuited for monetary calculations because it is impossible to represent 0.1 (or any other negative power of ten) as a float or double exactly.

**use BigDecimal, int, or long for monetary calculations**.

**49 Prefer primitive types to boxed primitives**

There are three major differences between primitives and boxed primitives

1. First, primitives have only their values, whereas boxed primitives have identities distinct from their values. In other words, two boxed primitive instances can have the same value and different identities.
2. Second, primitive types have only fully functional values, whereas each boxed primitive type has one nonfunctional value, which is null, in addition to all of the functional values
3. Last, primitives are generally more time- and space-efficient than boxed primitives.
4. Whats the issue here

Comparator<Integer> naturalOrder = new Comparator<Integer>() {

public int compare(Integer first, Integer second) {

return first < second ? -1 : (first == second ? 0 : 1);

} };

1. What will be out put of : naturalOrder.compare(new Integer(42), new Integer(42))

Its 1 ? how come : ?

1. Consider this :

public class Unbelievable {

static Integer i;

public static void main(String[] args) {

if (i == 42)

System.out.println("Unbelievable");

}

}

**50 Avoid strings where other types are more appropriate**

1. Strings are poor substitutes for enum types.
2. Strings are poor substitutes for aggregate types.

String compoundKey = className + "#" + i.next();

1. Thread local class

**51 Beware the performance of string concatenation**

1. Using the string concatenation operator repeatedly to concatenate *n* strings requires time quadratic in *n*
2. **Inappropriate use of string concatenation - Performs horribly!**

public String statement() {

String result = "";

for (int i = 0; i < numItems(); i++)

result += lineForItem(i); // String concatenation

return result;

}

1. **To achieve acceptable performance, use a StringBuilder in place of a String** to store the statement under construction.

public String statement() {

StringBuilder b = new StringBuilder(numItems() \* LINE\_WIDTH);

for (int i = 0; i < numItems(); i++)

b.append(lineForItem(i));

return b.toString();

}

1. The difference in performance is dramatic. If numItems returns 100 and lineForItem returns a constant 80-character string, the second method is eightyfive times faster than the first on my machine

**52 Refer to objects by their interfaces**

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

Gud use

List<Subscriber> subscribers = new Vector<Subscriber>();

**// Bad - uses class as type!**

Vector<Subscriber> subscribers = new Vector<Subscriber>();

**If you get into the habit of using interfaces as types, your program will be much more flexible.** If you decide that you want to switch implementations, allyou have to do is change the class name in the constructor (or use a different staticfactory). For example, the first declaration could be changed to readList<Subscriber> subscribers = new **ArrayList**<Subscriber>();

**53 Prefer interfaces to reflection (GTK)**

Item discuss prons and cons of reflection

Cons of reflection

1. You lose all the benefits of compile-time type checking,
2. The code required to perform reflective access is clumsy and verbose. It is

tedious to write and difficult to read.

1. Performance suffers. Reflective method invocation is much slower than

normal method invocation

1. If the appropriate constructor has no parameters, then you don’t even need to use java.lang.reflect; the Class.newInstance method provides the required functionality.

**54: Use native methods judiciously**

**Can Skip**

**55: Optimize judiciously**

1. Making a public type mutable may require a lot of needless defensive copying (Item 39).
2. Similarly, using inheritance in a public class where composition would have been appropriate ties the class forever to its superclass, which can place artificial

limits on the performance of the subclass (Item 16).

1. As a final example, using an implementation type rather than an interface in an API ties you to a specific implementation, even though faster implementations may be written in the future (Item 52).
2. do not strive to write fast programs—strive to write good ones; speed will follow.
3. Profiling tools can help you decide where to focus your optimization efforts. Such tools give you runtime information, such as roughly how much time each method is consuming and how many times it is invoked.
4. it’s difficult to guess where your program is spending its time. The part of the program that you think is slow may not be at fault, in which case you’d be wasting your time trying to optimize it. Common wisdom says that programs spend 80 percent of their time in 20 percent of their code.
5. In addition check method\algorithm complexity

**57 Adhere to generally accepted naming conventions**

1. Classes, including enum types, are generally named with a singular noun or noun phrase, for example, Timer, BufferedWriter, or Chess- Piece
2. Interfaces are named like classes, for example, Collection or Comparator, or with an adjective ending in able or ible, for example, Runnable, Iterable, or Accessible
3. Methods that perform some action are generally named with a verb or verb phrase (including object), for example, append or drawImage. Methods that return a boolean value usually have names that begin with the word is or, less commonly, has, followed by a noun, noun phrase, or any word or phrase that functions as an adjective.

**Exception (GTK)**

**57 Use exceptions only for exceptional conditions**

1. Do not use exception for decide on program flow.
2. exceptions are, as their name implies, to be used only for exceptional conditions; they should never be used for ordinary control flow**.**
3. Eg:

// Horrible abuse of exceptions. Don't ever do this!

try {

int i = 0;

while(true)

range[i++].climb();

} catch(ArrayIndexOutOfBoundsException e) {

}

1. Why ? 3 things wrong in doing so :
   * 1. Design for exceptionalcircumstances,, there is no incentive in JVM implementors to make them as fast
     2. Placing code inside a try-catch block inhibits certain optimizations that modern JVM implementations might otherwise perform
     3. the exception-based idiom is far slower than the standard one on modern JVM implementations.
2. What wil happen if iterators don’t hve hasNext method
   1. try {
   2. Iterator<Foo> i = collection.iterator();
   3. while(true) {
   4. Foo foo = i.next();
   5. ...
   6. }
   7. } catch (NoSuchElementException e) {

}

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

1. use checked exceptions for conditions from which the caller can reasonably be expected to recover
2. There are two kinds of unchecked throwables: runtime exceptions and errors. They are identical in their behavior: both are throwables that needn’t, and generally shouldn’t, be caught. If a program throws an unchecked exception or an error, it is generally the case that recovery is impossible and continued execution would do more harm than good.
3. there is a strong convention that errors are reserved for use by the JVM to indicate resource deficiencies, invariant failures, or other conditions that make it impossible to continue execution. Given the almost universal acceptance of this convention, it’s best not to implement any new Error subclasses. Therefore, **all of the unchecked throwables you implement should subclass RuntimeException**

**59 Avoid unnecessary use of checked exceptions**

One technique for turning a checked exception into an unchecked exception is

to break the method that throws the exception into two methods, the first of which

returns a boolean that indicates whether the exception would be thrown. This API

refactoring transforms the calling sequence from this:

**// Invocation with checked exception**

try {

obj.action(args);

} catch(TheCheckedException e) {

// Handle exceptional condition

...

}

to this:

**// 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 (GTK)**

1. IllegalArgumentException.: throw when the caller passes in an argument whose

value is inappropriate. Eg to pass negative number in counter

1. IllegalStateException.: the exception to throw if the invocation is illegal because of the state of the receiving object. Eg: if the caller attempted to use some object before it had been properly initialized.
2. ConcurrentModificationException : This exception should be thrown if an object that wasdesigned for use by a single thread or with external synchronization detects that it is being (or has been) concurrently modified

Concurrency

**Synchronize access to shared mutable data**

1. synchronized keyword ensures that only a single thread can execute a method or block at one time.
2. reading or writing a variable is *atomic* unless the variable is of type long or doubleSynchronization is required for reliable communication between threads as well as for mutual exclusion. This is due to a part of the language specification known as the *memory model*, which specifies when and how changes made by one thread become visible to others
3. A recommended way to stop one thread from another is to have the first thread poll a boolean field that is initially false but can be set to true by the second thread to indicate that the first thread is to stop itself. Because reading and writing a boolean field is atomic.
4. Broken! - How long would you expect this program to run?

public class StopThread {

private static boolean stopRequested;

The problem is that in the absence of synchronization, there is no guarantee as to when, if ever, the background thread will see the change in the value of stop- Requested that was made by the main thread

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;

}}

1. **How to fix it :**

**// Cooperative thread termination with a volatile field**

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;

}

}

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()**;

}}

1. Follwing is not thread safe, even with atomic :

**// Broken - requires synchronization!**

private static **volatile** int nextSerialNumber = 0;

public static int generateSerialNumber() {

return nextSerialNumber++;

}

1. The problem is that the increment operator (++) is not atomic. It performs *two* operations on the nextSerialNumber field: first it reads the value, then it writes back a new value, equal to the old value plus one. If a second thread reads the field between the time a thread reads the old value and writes back a new one, the second thread will see the same value as the first and return the same serial number. This is a *safety failure*: the program computes the wrong results
2. How to fix above use synchronization and remove volatile keyword or use the class AtomicLong
3. There are many ways to safely publish an object reference: you can store it in a static field as part of class initialization; you can store it in a volatile field, a final field, or a field that is accessed with normal locking; or you can put it into a concurrent collection

**Avoid excessive synchronization**

1. Depending on the situation, excessive synchronization can cause reduced performance, deadlock, or even nondeterministic behaviour
2. To avoid liveness and safety failures, never cede control to the client within a synchronized method or block
3. Broken - invokes alien method from synchronized block

private final List<SetObserver<E>> observers =

new ArrayList<SetObserver<E>>();

public void addObserver(SetObserver<E> observer) {

synchronized(observers) {

observers.add(observer);

}

}

private void notifyElementAdded(E element) {

**synchronized(observers) {**

for (SetObserver<E> observer : observers)

**observer.added(this, element);**

**}** }

1. **What will happen after this**

set.addObserver(new SetObserver<Integer>() {

public void added(ObservableSet<Integer> s, Integer e) {

System.out.println(e);

**if (e == 23) s.removeObserver(this);**

}

});

1. **It**  throws a ConcurrentModificationException. The problem is that notifyElementAdded is in the process of iterating over the observers list when it invokes the observer’s added method. The added method calls the observable set’s removeObserver method, which in turn calls observers.remove.
2. The iteration in the notifyElementAdded method is in a synchronized block to prevent concurrent modification, but it doesn’t prevent the iterating thread itself from calling back into the observable set and modifying its observers list.
3. **Observer that uses a background thread needlessly**

public void added(final ObservableSet<Integer> s, Integer e) {

System.out.println(e);

if (e == 23) {

ExecutorService executor =

Executors.newSingleThreadExecutor();

final SetObserver<Integer> observer = this;

try {

executor.submit(new Runnable() {

public void run() {

s.removeObserver(observer);

}

}).get();

} catch (ExecutionException ex) {

throw new AssertionError(ex.getCause());}

1. This time we don’t get an exception; we get a deadlock. The background thread calls s.removeObserver, which attempts to lock observers, but it can’t acquire the lock, because the main thread already has the lock. All the while, the main thread is waiting for the background thread to finish removing the observer, whic explains the deadlock.
2. **How to fx above :**

**// 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);

}

1. **Another solution is to use** CopyOnWriteArrayList. It is ArrayList in which all write operations are implemented by making a fresh copy of the entire underlying array. Because the internal array is never modified,

**// Thread-safe observable set with CopyOnWriteArrayList**

private final List<SetObserver<E>> observers =

new CopyOnWriteArrayList<SetObserver<E>>();

public void addObserver(SetObserver<E> observer) {

observers.add(observer);

}

public boolean removeObserver(SetObserver<E> observer) {

return observers.remove(observer);

}

private void notifyElementAdded(E element) {

for (SetObserver<E> observer : observers)

observer.added(this, element);

}

1. **Synchronization and performance** : In a multicore world, the real cost of excessive synchronization is not the CPU time spent obtaining locks; it is the lost opportunities for parallelism and the delays imposed by the need to ensure that every core has a consistent view of memory
2. Another hidden cost of oversynchronization is that it can limit the VM’s ability to optimize codeexecution.
3. If a method modifies a static field, you *must* synchronize access to this field, even if the method is typically used only by a single thread. It is not possible for clients to perform external synchronization on such a method because there can be no guarantee that unrelated clients will do likewise.

**Prefer executors and tasks to threads**

**Choosing the executor service**

1. If you’re writing a small program, or a lightly loaded server, using Executors.new-CachedThreadPool is generally a good choice,
2. is not a good choice for a heavily loaded production server! In a cached thread pool, submitted tasks are not queued but immediately handed off to a thread for execution. If no threads are available, a new one is created. If a server is so heavily loaded that all of its CPUs are fully utilized, and more tasks arrive, more threads will be created, which will only make matters worse.
3. in a heavily loaded production server,you are much better off usi Executors.newFixedThreadPool, which gives you a pool with a fixed number of threads, or using the ThreadPoolExecutor class directly,
4. In executor framework The key abstraction is the unit of work, which is called a *task*. There are two kinds of tasks: Runnable and its close cousin, Callable replacement for java.util.Timer, is ScheduledThreadPoolExecutor.
5. A timer uses only a single thread for task execution, which can hurt timing accuracy in the presence of long running tasks. If a timer’s sole thread throws an uncaught exception, the timer ceases to operate. A scheduled thread pool executor supports multiple threads and recovers gracefully from tasks that throw unchecked exceptions

**Prefer concurrency utilities to wait and notify**

1. The higher-level utilities in java.util.concurrent fall into three categories: the Executor Framework, concurrent collections; and synchronizers
2. To provide high concurrency, these implementations manage their own synchronization internally
3. This means that clients can’t atomically compose method invocations on concurrent collections
4. String intern : intern method is supposed to return the String from the String pool if the String is found in String pool, otherwise a new string object will be added in String pool and the reference of this String is returned.
5. java automatically interns String literals. This means that in many cases, the == operator appears to work for Strings in the same way that it does for ints or other primitive values.Since interning is automatic for String literals, the intern() method is to be used on Strings constructed with new String()

Concurrent Collection:

1. Besides offering excellent concurrency, ConcurrentHashMap is very fast. On my machine the optimized intern method above is over six times faster than String.intern ()

public static String intern(String s) {

String previousValue = map.putIfAbsent(s, s);

return previousValue == null ? s : previousValue;

}

1. BlockingQueue extends Queue and adds several methods, including take, which removes and returns the head element from the queue, waiting if the queue is empty.
2. This allows blocking queues to be used for *work queues* (also known as *producer-consumer queues*), to which one or more *producer threads* enqueue work items and from which one or more *consumer threads* dequeue and process items as they become available.
3. most ExecutorService implementations, including ThreadPoolExecutor, use a BlockingQueue

*Synchronizers*

1. Countdown latches are single-use barriers that allow one or more threads to wait for one or more other threads to do something.
2. CountDownLatch takes an int that is the number of times the countDown method must be invoked on the latch before all waiting threads are allowed to proceed.
3. Eg of framwwork which records the time for workers thred to finish their work.

**// Simple framework for timing concurrent execution**

public static long time(Executor executor, int concurrency,

final Runnable action) throws InterruptedException {

final CountDownLatch ready = new CountDownLatch(concurrency);

final CountDownLatch start = new CountDownLatch(1);

final CountDownLatch done = new CountDownLatch(concurrency);

for (int i = 0; i < concurrency; i++) {

executor.execute(new Runnable() {

public void run() {

ready.countDown(); // Tell timer we're ready

try {

start.await(); // Wait till peers are ready

action.run();

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

} finally {

done.countDown(); // Tell timer we're done

}

}

});

}

ready.await(); // Wait for all workers to be ready

long startNanos = System.nanoTime();

start.countDown(); // And they're off!

done.await(); // Wait for all workers to finish

return System.nanoTime() - startNanos;}

1. If a worker thread catches an InterruptedException, it reasserts the interrupt using the idiom Thread.currentThread().interrupt() and returns from its run method. This allows the executor to deal with the interrupt as it sees fit, which is as it should be.

Wait and notify

1. Always use the wait loop idiom to invoke the wait method; never invoke it

outside of a loop.

1. A related issue is whether you should use notify or notifyAll to wake waiting threads
2. NotifyAll will always yield correct results because it guarantees that you’ll wake the threads that need to be awakened. You may wake some other threads, too, but this won’t affect the correctness of your program. These threads will check the condition for which they’re waiting and, finding it false, will continue waiting.
3. As an optimization, you may choose to invoke notify instead of notifyAll if all threads that could be in the wait-set are waiting for the same condition and only one thread at a time can benefit from the condition becoming true.
4. using notifyAll in place of notify protects against accidental or malicious waits by an

unrelated thread.

**70: Document thread safety**

1. Can we say if a method is thread-safe by looking for the synchronized modifier in its documentation :
2. Its wrong on several counts. In normal operation, Javadoc does not include the synchronized modifier in its output, and with good reason. The presence of the synchronized modifier in a method declaration is an implementation detail, not a part of its exported API.

**Classificaton of classes**

* **Immutable : thread safe ,** Instances of this class appear constant. No external synchronization is necessary. Examples include String, Long, and BigInteger
* **unconditionally thread-safe**, Instance of class are mutable, but the class has sufficient internal synchronization that its instances can be used concurrently without the need for any external synchronization. Examples include Random and ConcurrentHashMap.
* **conditionally thread-safe :** some methods require external synchronization for safe concurrent use. Examples include the collections returned by the Collections.synchronized wrappers, whose iterators require external synchronization
* **not thread-safe** : Instances of this class are mutable. To use them concurrently, clients must surround each method invocation (or invocation sequence) with external synchronization of the clients’ choosing. Examples include the general-purpose collection implementations, such as ArrayList and HashMap.
* **thread-hostile :** not thread safe even used by external synchronization. Thread hostility usually results from modifying static data without synchronization. No one writes a thread-hostile class on purpose; such classes result from the failure to consider concurrency

1. Typically it is the lock on the instance itself, but there are exceptions. If an object represents a *view* on some other object, the client generally must synchronize on the

backing object, so as to prevent its direct modification

For example, the documentation for Collections.synchronizedMap says this:

It is imperative that the user manually synchronize on the returned map when

iterating over any of its collection views:

Map<K, V> m = Collections.synchronizedMap(new HashMap<K, V>());

...

Set<K> s = m.keySet(); // Needn't be in synchronized block

...

synchronized(m) { // Synchronizing on m, not s!

for (K key : s)

key.f();

}

Failure to follow this advice may result in non-deterministic behaviour

**Private lock vs public lock :**

1. When a class commits to using a publicly accessible lock, it enables clients to execute a sequence of method invocations atomically, but this flexibility comes at a price . It is incompatible with high-performance internal concurrency control, of

the sort used by concurrent collections such as ConcurrentHashMap and ConcurrentLinkedQueue.

1. Also, a client can mount a denial-of-service attack by holding the publicly accessible lock for a prolonged period. This can be done accidentally or intentionally.
2. private lock object idiom can be used only on *unconditionally* thread-safe classes. Conditionally thread-safe classes can’t use this idiom because they must document which lock their clients are to acquire when performing certain method invocation sequences.
3. To prevent this denial-of-service attack, you can use a *private lock object*

instead of using synchronized methods

**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**

1. Under most circumstances, normal initialization is preferable to lazy initialization.

**// Normal initialization of an instance field**

private final FieldType field = computeFieldValue();

1. While lazy initialization is primarily an optimization, it can also be used to break harmful circularities in class and instance initialization.
2. It decreases the cost of initializing a class or creating an instance, at the expense of increasing the cost of accessing the lazily initialized field
3. In the presence of multiple threads, lazy initialization is tricky. If two or more threads share a lazily initialized field, it is critical that some form of synchronization be employed, or severe bugs can result

**How to do it :**

1. Using Synchronzation :

private FieldType field;

**synchronized** FieldType getField() {

if (field == null)

field = computeFieldValue();

return field;

}

1. If you need to use lazy initialization for performance on a static field, use

the *lazy initialization holder class idiom*. This idiom (also known as the *initializeon-demand holder class idiom*) exploits the guarantee that a class will not be initializeduntil it is used

**// Lazy initialization holder class idiom for static fields**

private static class FieldHolder {

static final FieldType field = computeFieldValue();

}

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

When the getField method is invoked for the first time, it reads Field- Holder.field for the first time, causing the FieldHolder class to get initialized

The beauty of this idiom is that the getField method is not synchronized and performs only a field access, so lazy initialization adds practically nothing to the cost of access.

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

* This idiom avoids the cost of locking when accessing the field after it has been initialized
* The idea behind the idiom is to check the value of the field twice (hence the name *double-check*): once without locking, and then, if the field appears to be uninitialized, a second time with locking.
* Only if the second check indicates that the field is uninitialized does the call initialize the field. Because there is no locking if the field is already initialized, it s *critical* that the field be declared volatile

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

private **volatile** FieldType field;

@NotThreadSafe

public class DoubleCheckedLocking {

private static **volatile** Resource resource;

public static Resource getInstance() {

if (resource == null) {

synchronized (DoubleCheckedLocking.class) {

if (resource == null)

resource = new Resource();

}

return resource;}

}

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;

}

* Prior to release 1.5, the double-check idiom did not work reliably because the semantics of the volatile modifier were not strong enough to support it.
* **What the error we may get if we don’t use the volatile in dcl**: partially intialzed obj (think abt it )

**Single check locking :** you may need to lazily initialize an instance field that can tolerate repeated initialization

**// 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;

}

If you don’t care whether *every* thread recalculates the value of a field, and the type of the field is a primitive other than long or double, then you may choose to remove the volatile modifier from the field declaration in the single-check idiom. This variant is known as the *racy single-check idiom.*

It speeds up field access on some architectures, at the expense of additional initializationsIt is, however, used by String instances to cache their hash codes.

**72: Don’t depend on the thread scheduler**

1. The best way to write a robust, responsive, portable program is to ensure that the average number of *runnable* threads is not significantly greater than the numberof processors.
2. The main technique for keeping the number of runnable threads down is to have each thread do some useful work and then wait for more. **Threads should not run if they aren’t doing useful work.**
3. In terms of the Executor Framework (Item 68), this means sizing your thread pools appropriately [Goetz06 8.2], and keeping tasks reasonably small and independent of one another
4. Threads should not *busy-wait*, repeatedly checking a shared object waiting for something to happen
5. Besides making the program vulnerable to the vagaries of the scheduler, busy-waiting greatly increases the load on the processor, reducing the amount of useful work that others can accomplish.
6. Bad implementation of countdown latch

**// Awful CountDownLatch implementation - busy-waits incessantly!**

public class SlowCountDownLatch {

private int count;

public SlowCountDownLatch(int count) {

if (count < 0)

throw new IllegalArgumentException(count + " < 0");

this.count = count;

}

public void await() {

**while (true) {**

**synchronized(this) {**

**if (count == 0) return;**

**}**

**}**

public synchronized void countDown() {

if (count != 0)

count--;

}

}

1. Whats wrong? How to fix it ?
2. resist the temptation to “fix” the program by putting in calls to Thread.yield.

this might work in one jvm not in another.

1. Use Thread.sleep(1).

**Serialization**

**Serialization from Cathy**

Q what can happen if there is no serialization ? Each developer has to write and specify its own format\protocol.

**ObjectOutputStream and ObjectInputStream**

1. The magic of basic serialization happens with just two methods: one to serialize objects and write them to a stream, and a second to read the stream and deserialize objects.

ObjectOutputStream.writeObject() // serialize and write

ObjectInputStream.readObject() // read and deserialize

1. **Exmaple**

FileOutputStream fs = new FileOutputStream("testSer.ser");

ObjectOutputStream os = new ObjectOutputStream(fs);

**os.writeObject(c);**

FileInputStream fis = new FileInputStream("testSer.ser");

ObjectInputStream ois = new ObjectInputStream(fis);

**C = (Cat) ois.readObject(); // 4**

ois.close();

1. If we have a coller object inside a Dog object, Dog is seriliazed but coller is not then would os.writeObject(dog) will work ?

Ans: No it will throw exception: java.io.NotSerializableException: Collar

Collar c = new Collar(3);

Dog d = new Dog(c, 8);

Os.writeObject(c)

Q. Then how t make it work

1) Coller should also implements seriliazable

2) Make coller transient and override readObject and write Object value

private **transient** Collar theCollar;

1. Just making the coller **transient** would **not** work , as it will initialize object coller with its default value that is null. So following may throw NPE

System.out.println("after: collar size is " + d.getCollar().getCollarSize());

1. Why seriliazing the Coller is not possible always

Ans : It might be private, might be a 3rd party jar, may have other non transient \ non serilizable objects.

**Using writeObject and readObject:**

1. What can we do to make sure that when the Dog is deserialized, it gets a new Collar that matches the one the Dog had when the Dog was saved?
2. Meet write and read object method :
3. private void writeObject(ObjectOutputStream os) { throws IOException { os.defaultWriteObject();

os.writeInt(theCollar.getCollarSize());

}

private void readObject(ObjectInputStream is) throws IOException, ClassNotFoundException {

is.defaultReadObject();

theCollar = new Collar(is.readInt());

}

1. why wouldn't *all* Java classes be serializable? Why isn't class Object serializable?

There are some things in Java that simply cannot be serialized because they are runtime specific. Things like streams, threads, runtime, etc. and even some GUI classes

**Inheritance And Serialization**

1. *If a superclass is Serializable then all subclasses will be serilizable*
2. *What happens if sub class is serializable but super is not ?*
3. *Consider below :*

class Animal { // not serializable !

int weight = 42;

}

class Dog extends Animal implements Serializable {

String name;

Dog(int w, String n) {

weight = w; // inherited

name = n; // not inherited

}

}

Public static vois main (String k []){

Dog d = new Dog(35, "Fido");

Sysout(d.weight)

Os.writeobject(d);

d = (Dog) ois.readObject();

Sysout(d.weight)

}

Whats out put :

35

43

If I change the program as below :

**class** Dog **extends** Animal **implements** Serializable {

String name;

Dog(**int** w, String n) {

**super**(w);

name = n; // not inherited

}

}

**class** Animal { // not serializable !

**int** weight = 42;

**public** Animal (**int** w){

t**his**.weight = w ;

}

}

Check in E:\vob\vobSnH\CoreJava\src\ Serilization

1. If you are a serializable class, but your superclass is NOT serializable, then any instance variables you INHERIT from that superclass will be reset to the values they were given during the original construction of the object. This is because the nonserializable class constructor WILL run! (**Something is missing here**)

What is serialization?

1. Encoding an object as a byte stream is known as *serializing* the object; the reverse process is known as *deserializing* it.
2. Once an object has been serialized, its encoding can be transmitted from one running virtual machine to another or stored on disk for later deserialization.
3. Serialization provides the standard wire-level object representation for remote communication, and the standard persistent data format for the JavaBeans component architecture

**Implement Serializable judiciously**

1. A major cost of implementing Serializable is that it decreases the flexibility to change a class’s implementation once it has been released.
2. If you accept the default serialized form and later change the class’s internal representation, an incompatible change in the serialized form might result.
3. Clients attempting to serialize an instance using an old version of the class and deserialize it using the new version will experience program failures.
4. A simple example of the constraints on evolution that accompany serializability concerns *stream unique identifiers*, more commonly known as *serial version UIDs*.
5. Every serializable class has a unique identification number associated with it. If you do not specify this number explicitly by declaring a static final long field named serialVersionUID, the system automatically generates it at runtime by applying a complex procedure to the class.
6. The automatically generated value is affected by the class’s name, the names of the interfaces it implements, and all of its public and protected members. If you change any of these things in any way, for example, by adding a trivial convenience method, the automatically generated serial version UID changes. If you fail to declare an explicit serial version UID, compatibility will be broken, resulting in an InvalidClassException at runtime.
7. **A second cost of implementing Serializable is that it increases the likelihood of bugs and security holes.**
8. deserialization is a “hidden constructor” with all of the same issues as other constructors
9. Because there is no explicit constructor associated with deserialization, it is easy to forget that you must ensure that it guarantees all of the invariants established by the constructors and that it does not allow an attacker to gain access to the internals of the object under construction.
10. A third cost of implementing Serializable is that it increases the testing burden associated with releasing a new version of a class
11. When a serializable class is revised, it is important to check that it is possible to serialize an instance in the new release and deserialize it in old releases, and vice versa.
12. *Along with binary compatibility*, you must test for *semantic compatibility*. In other words, you must ensure both that the serialization- deserialization process succeeds and that it results in a faithful replica of the original object.
13. Implementing the Serializable interface is not a decision to be undertaken lightly**.**
14. As a rule of thumb, value classes such as Date and BigInteger should implement Serializable, as should most collection classes. Classes representing active entities, such as thread pools, should rarely implement Serializable
15. Classes designed for inheritance (Item 17) should rarely implement Serializable, and interfaces should rarelyextend it.
16. Classes designed for inheritance that *do* implement Serializable include Throwable, Component, and HttpServlet. Throwable implements Serializable so exceptions from remote method invocation (RMI) can be passed from server to client. Component implements Serializable so GUIs can be sent, saved, and restored. HttpServlet implements Serializable so session state can be cached.
17. There is one caveat regarding the decision *not* to implement Serializable. If a class that is designed for inheritance is not serializable, it may be impossible to write a serializable subclass. Specifically, it will be impossible if the superclass does not provide an accessible parameterless constructor. Therefore, **you should consider providing a parameterless constructor on nonserializable classes designed for inheritance**.

**Consider using a custom serialized form**

1. Do not accept the default serialized form without first considering whether it is appropriate.
2. The default serialized form is likely to be appropriate if an object’s physical representation is identical to its logical content.

Eg is Simple person class which contains 3 field first name , last name and middle name

1. Even if you decide that the default serialized form is appropriate, you often must provide a readObject method to ensure invariants and security. In the case of Name, the readObject method must ensure that lastName and first-Name are non-null.
2. Complex case may be a list class which holds list of string, A reasonable serialized form for StringList is simply the number of strings in the list, followed by the strings themselves.

private void writeObject(ObjectOutputStream s)

throws IOException {

s.defaultWriteObject();

s.writeInt(size);

// Write out all elements in the proper order.

for (Entry e = head; e != null; e = e.next)

s.writeObject(e.data);

}

private void readObject(ObjectInputStream s)

throws IOException, ClassNotFoundException {

s.defaultReadObject();

int numElements = s.readInt();

// Read in all elements and insert them in list

for (int i = 0; i < numElements; i++)

add((String) s.readObject());

}

1. Using the default serialized form when an object’s physical representation differs substantially from its logical data content has four disadvantages:
2. It permanently ties the exported API to the current internal representation.
3. It can consume excessive space
4. It can consume excessive time.
5. It can cause stack overflows.
6. Even if all instance fields are transient, invoking defaultWriteObject affects the serialized form, resulting in greatly enhanced flexibility. The resulting serialized form makes it possible to add nontransient instance fields in a later release while preserving backward and forward compatibility.
7. **Before deciding to make a field nontransient, convince yourself that its value is part of the logical state of the object**
8. If you are using the default serialized form and you have labeled one or more fields transient, remember that these fields will be initialized to their *default values* when an instance is deserialized: null for object reference fields, zero for numeric primitive fields, and false for boolean fields
9. **you must impose any synchronization on object serialization that you would impose on any other method that reads the entire state of the object.** So, for example, if you have a thread-safe object (Item 70) that achieves its thread safety by synchronizing every method, and you elect to use the default serialized form, use the following writeObject method:

**// writeObject for synchronized class with default serialized form**

private **synchronized** void writeObject(ObjectOutputStream s)

throws IOException {

s.defaultWriteObject();

}

1. Regardless of what serialized form you choose, declare an explicit serial version UID in every serializable class you write. This eliminates the serial version UID as a potential source of incompatibility
2. There is also a smallperformance benefit. If no serial version UID is provided, an expensive computation is required to generate one at runtime.
3. If you ever want to make a new version of a class that is *incompatible* with existing versions, merely change the value in the serial version UID declaration.

**Write readObject methods defensively**

**Can be escaped**

The problem is that the readObject method is effectively another public constructor, and it demands all of the same care as any other constructor. readObject is a constructor that takes a byte stream as its sole parameter.

Here is a simple litmus test for deciding whether the default readObject method is acceptable for a class: would you feel comfortable adding a public constructor that took as parameters the values for each nontransient field in the object and stored the values in the fields with no validation whatsoever

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

**Instance control = singleton**

1. **Consider the class**

public class Elvis {

**public static final Elvis INSTANCE = new Elvis();**

**private** Elvis() { ... }

public void leaveTheBuilding() { ... }

}

2) this class would no longer be a singleton if the words “implements Serializable” were added to its declaration.

3) Any readObject method, whether explicit or default, returns a newly created instance, which will not be the same instance that was created at class initialization time.

If the Elvis class is made to implement Serializable, the following readResolve method suffices to guarantee the singleton property:

**// readResolve for instance control - you can do better!**

private Object readResolve() {

// Return the one true Elvis and let the garbage collector

// take care of the Elvis impersonator.

return INSTANCE;

}

This method ignores the deserialized object, returning the distinguished Elvis instance that was created when the class was initialized .

You could fix the problem by declaring the favorites field transient, but you’re better off fixing it by making Elvis a single-element enum type (Item 3).

Historically, the readResolve method was used for all serializable instance-controlled classes. As of release 1.5, this is no longer the best way to maintain instance control in a serializable class.

If instead you write your serializable instance-controlled class as an enum,

you get an ironclad guarantee that there can be no instances besides the declared

constants. The JVM makes this guarantee, and you can depend on it. It requires no

special care on your part. Here’s how our Elvis example looks as an enum:

**// Enum singleton - the preferred approach**

public enum Elvis {

INSTANCE;

private String[] favoriteSongs =

{ "Hound Dog", "Heartbreak Hotel" };

public void printFavorites() {

System.out.println(Arrays.toString(favoriteSongs));

}

}

**Consider serialization proxies instead of serialized Instances**

The decision to implement Serializable increases the likelihood of bugs and security problems,

because it causes instances to be created using an extralinguistic mechanism in

Place of ordinary constructors.

ReasResolve vs readObject

WriteResolve vs writeObject

<http://thecodersbreakfast.net/index.php?post/2011/05/12/Serialization-and-magic-methods>

**Java interview question**

**Ishi systems**

Composition Vs Aggregation

Linked list question to find middle elements

Abstraction vs Encapsulation

Loose Coupling Vs High Cohesion, how to achieve

Concurrent HashMap advantages

How to make class immutable

How to implement unmodified collection

Major vs minor GC

Does GC runs in pream space

Spring

How to call a method post object creation

How to declare a singleton class

Hibrenate

What is proxy type of object return from get method

What is lazy initialization exception

How hibernates implements concurrency

Design Pattern

Builder: Optional parameter

Fly weight

Iris

UI problem of access control (authorization) based on

**Interview by Polaris**

String,intern

Why password are stored as charter array not as string

Web Service :

how you will create wsdl from xsd

1. Cache invalidation

How memory is organized by Hazelcast

**2**