Design and Implementation Techniques

By Mohit Kumar

Consider Providing Static Factory methods instead of constructors

- First Advantage being that since it is a method, it can have a name.
- Second being that unlike constructors they are not required to construct new object every time they are called.
  - Which means the conceptual creation of an object need not be tide down to literal creation.

```
//Literal creation of boolean is absolutely nonsensical
Boolean b =new Boolean("true");
//there should really be only two boolean fields in the VM. And that should
//have been enforced by the library.However the static factory was added
//later.
Boolean b =Boolean.valueOf("true");
```

- Third being they can return any subtype as a return type.

Consider Providing Static Factory methods instead of constructors

- The disadvantage being a class without a public or protected constructor cannot be subclassed.
  - Actually it can be a blessing in disguise.

Enforce Singleton with a private constructor.

- *A singleton* is simply a class that is instantiated exactly once [Gamma95, p. 127]. Singletons typically represent some system component that is intrinsically unique, such as a video display or file system.
- There are two approaches to implementing singletons. Both are based on keeping the constructor private and providing a public static member to allow clients access to the sole instance of the class.
  - In one approach, the public static member is a final field:

```
// Singleton with final field
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { }
      ... // Remainder omitted
    }
}
```

  - The private constructor is called only once, to initialize the public static final field E1vis.INSTANCE. The lack of public or protected constructors *guarantees a* "monoelvistic" universe: Exactly one Elvis instance will exist once the Elvis class is initialized-no more, no less. Nothing that a client does can change this.

Enforce Singleton with a private constructor.

- In a second approach, a public static factory method is provided instead of the public static final field:
- All calls to the static method, Elvis. getInstance, return the same object reference, and no other Elvis instance will ever be created.

```
// Singleton with static factory
public class Elvis {
    private static final Elvis INSTANCE = new Elvis<);
    private Elvis() { }
    public static Elvis getInstance() { return INSTANCE; }
    .. // Remainder omitted
}
```

Enforce Singleton with a private constructor.

- The main advantage of the first approach is that the declarations of the members comprising the class make it clear that the class is a singleton:
- the public static field is final, so the field will always contain the same object reference. There may also be a slight performance advantage to the first approach, but a good JVM implementation should be able to eliminate it by inlining the call to the static factory method in the second approach.
- The main advantage of the second approach is that it gives you the flexibility to change your mind about whether the class should be a singleton without changing the API. The static factory method for a singleton returns the sole instance of the class but could easily be modified to return, say, a unique instance for each thread that invokes the method.
- On balance, then, it makes sense to use the first approach if you're absolutely sure that the class will forever remain a singleton. Use the second approach if you want to reserve judgment in the matter.

Enforce Singleton with a private constructor.

- To make a singleton class serializable, it is not sufficient merely to add implements Serializable to its declaration. To maintain the singleton guarantee, you must also provide a readResolve method. Otherwise, each deserialization of a serialized instance will result in the creation of a new instance, leading, in the case of our example, to spurious Elvis sightings.
- To prevent this, add the following readResolve method to the E1vis class:

```
// readResolve method to preserve singleton property
private Object readResolve() throws ObjectStreamException {
        /*Return the one true Elvis and let the garbage collector
         * take care of the Elvis impersonator.
         */
        return INSTANCE;
}
```

Enforce noninstantiability with private constructor

- Occasionally you write classes that have static methods.Classes like Collections,System etc.
- These classes represent algorithmic functions or system resources and instantiating them is nonsensical.
- Enforce noninstantiability by providing a private constructor.

Avoid creating duplicate objects.

- It is often appropriate to reuse a single object instead of creating a new Reuse can be both faster and more stylish. An object can always be reused if it is *immutable*.
- As an extreme example of what not to do, consider this statement:

```
String s = new String("silly"); // DON'T DO THIS!
```

- The statement creates a new String instance each time it is executed, and none of those object creations is necessary. The argument to the St ring constructor ("silly") is itself a String instance, functionally identical to all of the objects created by the constructor. If this usage occurs in a loop or in a frequently invoked method, millions of String instances can be created needlessly.
- The improved version is simply the following:

```
String s = "No longer silly";
```

Avoid creating duplicate objects.

- This item should not be misconstrued to imply that object creation is expensive and should be avoided. On the contrary, the creation and reclamation of small objects whose constructors do little explicit work is cheap, especially on modem JVM implementations. Creating additional objects to enhance the clarity, simplicity, or power of a program is generally a good thing.

Eliminate Obsolete Object References

- When you switch from a language with manual memory management, such as C or C++, to a garbage-collected language, your job as a programmer is made much easier by the fact that your objects are automatically reclaimed when you're through with them.
- It seems almost like magic when you first experience it. It can easily lead to the impression that you don't have to think about memory management, but this isn t quite true.

Eliminate Obsolete Object References

```
// Can you spot the "memory leak"? public class Stack {
private Object[] elements;
private int size = 0;
public Stack(int initialCapacity) { this.elements = new Object[initialCapacity]; }

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

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

Eliminate Obsolete Object References

```
    * Ensure space for at least one more element, roughly
    * doubling the capacity each time the array needs to grow.

    private void ensureCapacity() {
            if (elements.length == size) {
                    Object[] oldElements = elements;
                    elements = new Object[2 * elements.length + 1]; System.
            arraycopy(oldElements, 0, elements, 0, size);
            }
    }
}
```

Eliminate Obsolete Object References

- There's nothing obviously wrong with this program. You could test it exhaustively, and it would pass every test with flying colors, but there's a problem lurking.
- Loosely speaking, the program has a "memory leak," which can silently manifest itself as reduced performance due to increased garbage collector activity or increased memory footprint.
- In extreme cases, such memory leaks can cause disk paging and even program failure with an OutOfMemoryError, but such failures are relatively rare.
- So where is the memory leak? 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. 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.

Eliminate Obsolete Object References

- Memory leaks in garbage collected languages (more properly known as *unintentional object retentions)* are insidious.
- If an object reference is unintentionally retained, not only is that object excluded from garbage collection, but so too are any objects referenced by that object, and so on.
- Even if only a few object references are unintentionally retained, many, many objects may be prevented from being garbage collected, with potentially large effects on performance.

Eliminate Obsolete Object References

- The fix for this sort of problem is simple: Merely null out references once they become obsolete. In the case of our Stack class, the reference to an item becomes obsolete as soon as it's popped off the stack.
- The corrected version of the pop method looks like this:

```
public Object pop() {
    if (size==0)
       throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference return result;
}
```

Eliminate Obsolete Object References

- When programmers are first stung by a problem like this, they tend to overcompensate by nulling out every object reference as soon as the program is finished with it.
- This is neither necessary nor desirable as it clutters up the program unnecessarily and could **actually reduce performance**. Nulling out object references should be the exception rather than the norm.
- So when should you null out a reference'?
  - The golden rule being that when a programmer prolongs the life-cycle of the object by putting it in a global data structure which is managed by the programmer himself.
  - The data structure classes coming from the Java API will null out references for you if you ask it to remove the Object.

Eliminate Obsolete Object References

- Another common source of memory leaks is caches. Once you put an object reference into a cache, it's easy to forget that it's there and leave it in the cache long after it becomes irrelevant.
- This can be solved either by having a background thread cleaning up with the help of the timer or by using the appropriate Reference(consider using WeakHashMap) Object.
- You could also use a LinkedHashMap that has a “removeEldestEntry()” method.

Finalizers are not your friends

- Finalizers are unpredictable, often dangerous, and generally unnecessary. Their use can cause erratic behavior, poor performance, and portability problems.
- Finalizes have a few valid uses, which we'll cover later in this item, but as a rule of thumb, tinalizers should be avoided.
- Don’t use finalizer to do something time critical.
- Don’t use finalizer to update critical persistent state.
- There is only one(not so important) use of finalizers.
  - If an object represents a native resource that is conceptually tied down to the Object and the resource is not so critical then the resource can be released in the finalizer.

Finalizers are not your friends

- Now if a class A defines a finalizer(despite the caution), and if A's subclass B overrides the finalizer and forgets to call super.finalize() then we have a problem.
- To guard against such problems a finalize guardian can be placed.
- A single instance of the anonymous class, called *a finalizer guardian,* is created for each instance of the enclosing class. The enclosing instance stores the sole reference to its finalizer guardian in a private instance field so the finalizer guardian becomes eligible for finalization at the same time as the enclosing instance.
- When the guardian is finalized, it performs the finalization activity desired for the enclosing instance, just as if its finalizer were a method on the enclosing class

Finalizers are not your friends

```
public class Foo {
// Sole purpose of this object is to finalize outer Foo object
private final Object finalizerGuardian = new Object() {
        protected void finalize() throws Throwable {
                // Finalize outer Foo object

        }
        ... // Remainder omitted
    }
}//technique should be considered for every non final class that has a finalizer.
```

Finalizers are not your friends

- Remember writing a finalizer without any good reason(there are none or maybe a flimsy reason) is not advised. **It actually slows down garbage collection tremendously.**

Design and Implementation Techniques.

- Obey General contract(refer to javadoc) when overriding “equals()”.
- Always override “hashCode()” when overriding “equals()”.
- Always override “toString()”
- Consider implementing Comparable.

Use Clone Judiciously

- How does cloning work in java??

```
class MyObject implements Cloneable {
    private int n;
    public MyObject(int n) { this.n = n; }
    public Object clone() {
      Object o = null;
      try {
                o = super.clone();
      } catch(CloneNotSupportedException e) {
                System.err.println("MyObject can't clone");
      }
      return o;
    }
    public int getValue() { return n; }
    public void setValue(int n) { this.n = n; }
    public void increment() { n++; }
    public String toString() { return Integer.toString(n); }
}
```

Use Clone Judiciously

- To summarize you have to implement Cloneable interface override the clone method of the Object class and make it public which is by default protected and hence accessible to only subclasses.
- If you forget to implement the cloneable interface the super.clone() with throw a CloneNotSupportedException.
- The reason is that mixed into this design for cloneability was the thought that maybe you didn't want all types of objects to be cloneable.
- So **Object.clone( )** verifies that a class implements the **Cloneable** interface. If not, it throws a **CloneNotSupportedException** exception. So in general, you're forced to **implement Cloneable** as part of support for cloning.

Use Clone Judiciously

- **The effect of Object.clone( )**
  - What actually happens when **Object.clone( )** is called that makes it so essential to call **super.clone( )** when you override **clone( )** in your class?
  - The **clone( )** method in the root class is responsible for creating the correct amount of storage and making the **bitwise(shallow copy)** copy of the bits from the original object into the new object's storage.
  - That is, it doesn't just make storage and copy an **Object**; it actually figures out the size of the *real* object (not just the base-class object, but the derived object) that's being copied and duplicates that. Since all this is happening from the code in the **clone( )** method defined in the root class (that has no idea what's being inherited from it), you can guess that the process involves RTTI to determine the actual object that's being cloned.
  - This way, the **clone( )** method can create the proper amount of storage and do the correct bitwise copy for that type

Use Clone Judiciously

- **Why this strange design?**
  - If all this seems to be a strange scheme, that's because it is. You might wonder why it worked out this way. What is the meaning behind this design?
  - Originally, Java was designed as a language to control hardware boxes, and definitely not with the Internet in mind. In a general-purpose language like this, it makes sense that the programmer be able to clone any object. Thus, **clone( )** was placed in the root class **Object**, *but* it was a **public** method so you could always clone any object. This seemed to be the most flexible approach, and after all, what could it hurt?
  - Well, when Java was seen as the ultimate Internet programming language, things changed. Suddenly, there are security issues, and of course, these issues are dealt with using objects, and you don't necessarily want anyone to be able to clone your security objects.

Use Clone Judiciously

- So what you're seeing is a lot of patches applied on the original simple and straightforward scheme: **clone( )** is now **protected** in **Object**. You must override it *and* **implement Cloneable** *and* deal with the exceptions.

Use Clone Judiciously

```
// Can't clone this because it doesn't override clone():
class Ordinary {}

// Overrides clone, but doesn't implement Cloneable:
class WrongClone extends Ordinary {
	public Object clone() throws CloneNotSupportedException {
		return super.clone(); // Throws exception
	}
}

// Does all the right things for cloning:
class IsCloneable extends Ordinary implements Cloneable {
	public Object clone() throws CloneNotSupportedException {
		return super.clone();
	}
}
```

Use Clone Judiciously

```
// Turn off cloning by throwing the exception:
class NoMore extends IsCloneable {
    public Object clone() throws CloneNotSupportedException {
        throw new CloneNotSupportedException();
    }
}

class TryMore extends NoMore {
    public Object clone() throws CloneNotSupportedException {
        // Calls NoMore.clone(), throws exception:
        return super.clone();
    }
}
```

Use Clone Judiciously

```
class BackOn extends NoMore {
    private BackOn duplicate(BackOn b) {
        // Somehow(through serialization,reflexive, or copy contructor) make a copy
    of b and return that copy.
        // This is a dummy copy, just to make the point:
        return new BackOn();
    }
    public Object clone() {
        // Doesn't call NoMore.clone():
        return duplicate(this);
    }
}
```

Use Clone Judiciously

- The problems with clone.
  - The interface cloneable use is high atypical.
  - The Cloneabla interface was intended as *a mixin interface* for objects to advertise that they permit cloning.
  - Unfortunately, it fails to serve this purpose. Its primary flaw is that it lacks a clone method, and Object's clone method is protected.
  - You cannot, without resorting to *reflection* , invoke the clone method on an object merely because it implements Cloneable.
  - Even a reflective invocation may fail, as there is no guarantee that the object has an accessible clone method.
  - Despite this flaw and others, the facility is in sufficiently wide use that it pays to understand it.

Use Clone Judiciously

- In order for implementing the C1oneable a interface to have any effect on a class, it and all of its superclasses must obey a fairly complex, unenforceable, and largely undocumented protocol. The resulting mechanism *is extralinguistic:*
- **It creates an object without calling a constructor.**
- The general contract for the clone method is weak. Here it is, copied from the specification for java.lang.Object:
  - Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object x, the expression
  - x.clone() != x will be true, and the expression
  - x.clone().getClass() == x.getClass() will be true, **but these are not absolute requirements.**
  - **While it is typically the case that x.c**lone().equals(x) will be true, **this is not an absolute requirement.**

Use Clone Judiciously

- Copying an object will typically entail creating a new instance of its class, but it may require copying of internal data structures as well. No constructors are called.
- There are a number of problems with this contract. The 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.
- The provision that x.clone().getClass() should generally be identical to x.getClass(), however, is too weak.
  - In practice, programmers assume that if they extend a class and invoke super. clone from the subclass, the returned object will be an instance of the subclass.
  - The *only* way a superclass can provide this functionality is to return an object obtained by calling super.clone. If a clone method returns an object created by a constructor, it will have the wrong class.

Use Clone Judiciously

- Therefore, **if you override the clone method in** a **nonfinal class, you should return an object obtained** by **invoking super.clone.**
- If all of a class's superclasses obey this rule, then invoking super. clone will eventually invoke Object's clone method, creating an instance of the right class.
- **This mechanism is vaguely similar to automatic constructor chaining, except that it isn't enforced.**
- **In practice, a class that implements Cloneable is expected to provide a properly functioning public clone method.**
  - It is not, in general, possible to do so unless all of the class's superclasses provide a well-behaved clone implementation, whether public or protected.

Use Clone Judiciously

- Suppose you want to implement Cloneable in a class whose superclasses provide well-behaved clone methods.
- The object you get from super.clone() may or may not be close to what you'll eventually return, depending on the nature of the class.
- This object will be, from the standpoint of each superclass, a fully functional clone of the original object. The fields declared in your class (if any) will have values identical to those of the object being cloned.
- If every field contains a primitive value or a reference to an immutable object, the returned object may be exactly what you need, in which case no further processing is necessary.

Use Clone Judiciously

- If, however, your object contains fields that refer to mutable objects, using this clone implementation can be disastrous.

```
public class Stack {
    private Object[] elements;
    private int size = 0;

    public Stack(int initialCapacity) {
        this.elements = new Object[initialCapacity];
    }

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

Use Clone Judiciously

```
public Object pop() {
        if (size == 0)throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference return result;
    }

    // Ensure space for at least one more element.
    private void ensureCapacity() {
      if (elements.length == size) {
                    Object oldElements[] = elements;
                    elements = new Object[2 * elements.length + 1];
                    System. arraycopy(ol dEl ements, 0, elements, 0, size);
      }
    }
}
```

Use Clone Judiciously

- Suppose you want to make this class cloneable. If its clone method merely returns super.clone(), the resulting Stack instance will have the correct value in its size field, 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. You will quickly find that your program produces nonsensical results or throws a NullPointerException.

Use Clone Judiciously

- This situation could never occur as a result of calling the sole constructor in the Stack class.
- **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.**
- In order for the clone method on Stack to work properly, it must copy the internals of the stack. The easiest way to do this is by calling clone recursively on the elements array:

```
public Object clone() throws CloneNotSupportedException
{
    Stack result = (Stack) super.clone();
    result.elements =(Object[])elements.clone();
    return result;
}
```

Use Clone Judiciously

- Note that this solution would not work if the elements field were final because the clone method 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 objects,** except in cases where the mutable objects may be safely shared between an object and its clone.
- In order to make a class cloneable, it may be necessary to remove final modifiers from some fields.

Use Clone Judiciously

- It is not always sufficient to call clone recursively. For example, suppose you are writing a clone method for a hash table whose internals consist of an array of buckets, each of which references the first entry in a linked list of key-value pairs or is null if the bucket is empty.

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;
    private static class Entry {
        Object key;
        Object value;
        Entry next;
        Entry (Object key, Object value, Entry next) {
            this.key   = key;
            this.value = value;
            this.next = next;
        }
    }
    ... // Remainder omitted
}
```

Use Clone Judiciously

```
//Suppose you merely clone the bucket array recursively, as we did for Stack:
// Broken - results in shared internal state!
public Object clone() throws CloneNotSupportedException {
    HashTable result = (HashTable) super.clone();
    result.buckets = (Entry[]) buckets.clone();
    return result;
}
```

- Though the clone has its own bucket array, this array references the same linked lists as the original, which can easily cause nondeterministic behavior in both the clone and the original.
- To fix this problem, you'll have to copy the linked list that comprises each bucket individually. Here is one common approach:

Use Clone Judiciously

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;

    private static class Entry {
        Object key;
        Object value;
        Entry next;
        Entry(Object key, Object value, Entry next) {
            this.key   = key;
            this.value = value;
            this.next= next;
        }

        // Recursively copy the linked list headed by this Entry
        Entry deepCopy() {
            return new Entry (key, value,next == null ?
    null:next.deepCopy());
        }
    }
```

Use Clone Judiciously

```
    public Object clone() throws CloneNotSupportedException {
            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;
    }
    ... // Remainder omitted
}
```

Use Clone Judiciously

- The private class HashTable.Entry has been augmented to support a "deep copy" method. The clone method on HashTable allocates a new buckets array of the proper size and iterates over the original buckets array, deep-copying each nonempty bucket.
- The deep-copy method on Entry invokes itself recursively to copy the entire linked list headed by the entry.
- While this technique is cute and works fine if the buckets aren't too long, it is not a good way to clone a linked list because it consumes one stack frame for each element in the list.
- If the list is long, this could easily cause a stack overflow. To prevent this from happening, you can replace the recursion in deepCopy with iteration:

Use Clone Judiciously

```
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;
}
```

Use Clone Judiciously

- **Like a constructor and the readObject(), a clone method should not invoke any nonfinal methods on the clone under construction.**
- If clone invokes an overridden method, this method will execute before the subclass in which it is defined has had a chance to fix its state in the clone, quite possibly leading to corruption in the clone and the original.
- Object's clone method is declared to throw C1oneNotSupportedException, but overriding clone methods may omit this declaration.
- The clone methods of final classes should omit the declaration because methods that don't throw checked exceptions are more pleasant to use than those that do
- It is not essential that the foregoing advice be followed, as the clone method of a subclass that doesn't want to be cloned can always throw an unchecked exception, such as UnsupportedOperationException.

Use Clone Judiciously

- What the alternatives to clone()
  - Serialization will provide a default deep copy
  - Reflexive Cloning
  - Copy Constructor

Use Clone Judiciously

- The copy constructor approach and its static factory variant have many advantages over Cloneable/clone:
  - They do not rely on a risk-prone extralinguistic object creation mechanism; they do not demand unenforceable adherence to ill-documented conventions;
  - they do not conflict with the proper use of final fields; they do not require the client to catch an unnecessary checked exception;
  - and they provide a statically typed object to the client.
  - Furthermore, a copy constructor (or static factory) can take an argument whose type is an appropriate interface implemented by the class.

Use Clone Judiciously

- For example, all general-purpose collection implementations, by convention, provide a copy constructor whose argument is of type Col lection or Map. Interface-based copy constructors allow the client to choose the implementation of the copy, rather than forcing the client to accept the implementation of the original.
  - For example, suppose you have a LinkedList 1, and you want to copy it as an Arraylist. The clone method does not offer this functionality, but it's easy with a copy constructor: new ArrayList(l).

Use Clone Judiciously

- Given all of the problems associated with Cloneable, it is safe to say that other interfaces should not extend it and that classes designed for inheritance should not implement it.
- Because of its many shortcomings, some expert programmers simply choose never to override the clone method and never to invoke it except, perhaps, to copy arrays cheaply.
- Be aware that if you do not at least provide a well-behaved *protected* clone method on a class designed for inheritance, it will be impossible for subclasses to implement Cloneable

Minimize Accessibility

- The rule of the thumb is make each class or member as inaccessible as possible.
  - For top-level (non-nested) classes and interfaces, there are only two possible access levels: *package-private* and *public*. If you declare a top-level class or interface with the public modifier, it will be public; otherwise, it will be package-private. If a top-level class or interface can be made package-private, it should be. By making it package-private, you make it part of the package's implementation rather than its exported API, and you can modify it, replace it, or eliminate it in a subsequent release without fear of harming existing clients.
  - **If you make it public, you are obligated to support it forever to maintain compatibility(Boolean constructor is a good example of this failure).**

Minimize Accessibility

- If a package-private top-level class or interface is used only from within a single class, you should consider making it a private nested class (or interface) of the class in which it is used (not so important).
- For members (fields, methods, nested classes, and nested interfaces) there are four possible access levels, listed here in order of increasing accessibility:
  - **private-The** member is accessible only inside the top-level class where it is declared.
  - **package-private-The** member is accessible from any class in the package where it is declared. Technically known as *default* access, this is the access level you get if no access modifier is specified.
  - **protected-The** member is accessible from subclasses of the class where it is declared (subject to a few restrictions [JLS. 6.6.21) and from any class in the package where it is declared.
  - **public-**The member is accessible from anywhere.

Minimize Accessibility

- After carefully designing your class's public API, your reflex should be to make all other members private. Only if another class in the same package really needs to access a member should you remove the private modifier, making the member package-private.
- If you find yourself doing this often, you should reexamine the design of your system to see if another decomposition might yield classes that are better decoupled from one another.
- That said, both private and package-private members are part of a class's implementation and do not normally impact its exported API.
- These fields can, however, "leak" into the exported API if the class implements Serializable(That’s one reason why serialization must be done carefully).

Minimize Accessibility

- For members of public classes, a huge increase in accessibility occurs when the access level goes from package-private to protected. A protected member is part of the class's exported API and must be supported forever.
- Furthermore, a protected member of an exported class represents a public commitment to an implementation detail . The need for protected members should be relatively rare.
- There is one rule that restricts your ability to reduce the accessibility of methods. If a method overrides a superclass method, it is not permitted to have a lower access level in the subclass than it does in the superclass.

Minimize Accessibility

- This is necessary to ensure that an instance of the subclass is usable anywhere that an instance of the superclass is usable. If you violate this rule, the compiler will generate an error message when you try to compile the subclass.
- A special case of this rule is that if a class implements an interface, all of the class methods that are also present in the interface must be declared public. This is so because all methods in an interface are implicitly public.
- Public classes should rarely, if ever, have public fields (as opposed to public methods). If a field is nonfinal or is a final reference to a mutable object, you give up the ability to limit(anybody can store anything) the values that may be stored in the field by making it public.
- You also give up the ability to take any action when the field is modified. **A simple consequence is that classes with public mutable fields are not thread-safe.**

Minimize Accessibility

- Even if a field is final and does not refer to a mutable object, by making the field public, you give up the flexibility to switch to a new internal data representation.
- There is one exception to the rule that public classes should not have public fields. Classes are permitted to expose constants via public static final fields.
  - It is critical that these fields contain either primitive values or references to immutable Object. A final field containing a reference to mutable Object has all tile disadvantagcs of a nonfinal field

Minimize Accessibility

- Note that a nonzero-length array is always mutable, so **it is nearly always wrong to have public static final array field.** If a class has such a field, clients will be able to modify the contents of the array. This is a frequent source of security holes:

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

- The public array should be replaced by a private array and a public immutable list:

```
private static final Type[] PRIVATE VALUES ={... };
public static final List VALUES =
Collections.unmodifiableList(Arrays.asList(PRIVATE VALUES));
```

Minimize Accessibility

- Alternatively, if you require compile-time type safety and are willing to tolerate a performance loss, you can replace the public array field with a public method that returns a copy of a private array:

```
private static final Type[] PRIVATE VALUES ={... };

public static final Type[] values() { return (Type[]) PRIVATE VALUES.clone(); }
```

- To summarize, you should always reduce accessibility as much as possible. After carefully designing a minimal public API, you should prevent any stray classes, interfaces, or members from becoming a part of the API.
- With the exception of public static final fields, public classes should have no public fields. Ensure that objects referenced by public static final fields are immutable.

Favor Immutability

- An immutable class is simply a class whose instances cannot be modified. All of the information contained in each instance is provided when it is created and is fixed for the lifetime of the object.
- The Java platform libraries contain many immutable classes, including String, the primitive wrapper classes, and BigInteger and BigDecimal . There are many good reasons for this: Immutable classes are easier to design, implement, and use than mutable classes. They are less prone to error and are more secure.

Favor Immutability

- To make a class immutable, follow these five rules:
  - **Don't provide any methods that modify the object** (known as *mutators)(modification that is externally visible, internal modification is allowed).*
  - **Ensure that no methods may be overridden.** This prevents careless or malicious subclasses from compromising the immutable behavior of the class. Preventing method overrides is generally done by making the class final, but there are alternatives that we'll discuss later.
  - **Make all fields final.** This clearly expresses your intentions in a manner that is enforced by the system. Also, it may be necessary to ensure correct behavior if a reference to a newly created instance is passed from one thread to another without synchronization, depending on the results of ongoing efforts to rework the *memory model*.

Favor Immutability

- **Make all fields private.** This prevents clients from modifying fields directly. While it is technically permissible for immutable classes to have public final fields containing primitive values or references to immutable objects, it is not recommended because it precludes changing the internal representation in a later release.
- **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. Never initialize such a field to a client-provided object reference nor return the object reference from an accessor. Make *defensive copies* in contructors, accessors, and readObject methods.

Favor Immutability

- **Immutable objects are simple.** An immutable object can be in exactly one state, the state in which it was created.
  - Mutable objects, on the other hand, can have arbitrarily complex state spaces. If the documentation does not provide a precise description of the state transitions performed by mutator methods, it can be difficult or impossible to use a mutable class reliably.
- **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.
- Therefore **immutable objects can be shared freely.**
- Immutable classes should take advantage of this by encouraging clients to reuse existing instances wherever possible. One easy way to do this is to provide public static final constants for frequently used values.

Favor Immutability

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

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

    private static List cache=.....
    Complex(float re, float im) {
        this.re = re;
        this.im = im;
    }
    //Static factory method to aid caching, infact you can remove the public
    //accessibility of the constructor
    public static Complex valueOf(float re, float im){
        //1.check if there in the cache, if present return
        //2. else create new put in cache and return
    }
```

Favor Immutability

```
// Accessors with no corresponding mutators
    public float realPart() { return re; }
    public float 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);
    }
    public Complex multiply(Complex c) {
        return new Complex(re*c.re - im*c.im,
        re*c.im + im*c.re);
    }
    public Complex divide(Complex c) {
        float tmp = c.re*c.re + c.im*c.im;
        return new Complex((re*c.re + im*c.im)/tmp,
        (im*c.re - re*c.im)/tmp);
    }
}
```

Favor Immutability

- A consequence of the fact that immutable objects can be shared freely is that you never have to make *defensive copies*.
- **In fact, you never have to make any copies at all because the copies would be forever equivalent to the originals. Therefore you need not and should not provide a clone method or *copy constructor* 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

Favor Immutability

- Recall that to guarantee immutability, a class must not permit any of its methods to be overridden. In addition to making a class final, there are two other ways to guarantee this.
  - One way is to make each method of the class, but not the class itself, final. The sole advantage of this approach is that it allows programmers to extend the class by adding new methods built atop the old ones.
  - A second 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.

Favor Immutability

- However be **extremely careful** while ensuring immutability.

```
//Broken immutable Employee. There is a gaping security hole.
public final class Employee{
        private  Date birthdate;
        private  Date joindate;
        private Address address;
        private String name;
        public Employee(String n,Date birthdate, Date joindate,Address address) {
                this.name=n;
                this.birthdate =(Date) birthdate.clone();
                this.joindate= (Date) joindate.clone();
                if (this.birthdate.compareTo(this.joindate) > 0)throw new
                IllegalArgumentException();
                this.address=address;
        }
        public Date birthDate () {
                return (Date) birthdate.clone();
        }
```

Favor Immutability

```
        public Date joinDate ()             {
                return (Date) joindate.clone();
        }
        public Address address(){
                return address;
        }
        public String toString() { return birthdate +" - "                    + joindate+"
                "+address;
        }

}
```

Favor Immutability

```
public class Address implements Cloneable{
    private String street=null;
    private String city=null;
    private String state=null;
    private String zip=null;

    public String getCity() {
        return city;
    }
    public String getState() {
        return state;
    }
    public String getStreet() {
        return street;
    }
    public String getZip() {
        return zip;
    }
```

Favor Immutability

```
public Address(String street, String city, String state, String zip) {
        super();
        // TODO Auto-generated constructor stub
        this.street = street;
        this.city = city;
        this.state = state;
        this.zip = zip;
}
public static Address valueOf(String street, String city, String state, String zip){
        return new Address(street,city,state,zip);

}
public Object clone() {
        try {return super.clone();} catch (CloneNotSupportedException e) {
                // will never happen because we have implmented Cloneable
                return null;
        }
}//other methods from object not shown for lack of space
}
```

Favor Immutability

```
//modified constructors to rebut the attack
public Employee(String name,Date birthdate, Date joindate,Address address) {
        this.name=name;
        if ((birthdate.getClass()!= Date.class)||(joindate.getClass()!= Date.class)){
                //cant trust clone for subclass.
                System.out.println("Not Trusted");
                this.birthdate = new Date(birthdate.getTime());
                this.joindate           = new Date(joindate.getTime());
        }else{
                this.birthdate =(Date) birthdate.clone();
                this.joindate           = (Date) joindate.clone();
                if (this.birthdate.compareTo(this.joindate) > 0)throw new
                IllegalArgumentException();
        }
```

Favor Immutability

```
        // if address is not under your control like BigInteger,BigDecimal.
        //They are immutable themselves but permit subclassing and overriding
        if (address.getClass()!= Address.class){
                //cant trust clone for subclass.
                System.out.println("Not Trusted");
                this.address =Address.valueOf
(address.getStreet(),address.getCity(),address.getState(),address.getZip());
        }else{//you know that clone for address works fine.
                this.address =(Address) address.clone();
        }
        //if address is under your control then you can fix it and comment the above
        //lines and uncomment the below line.
        //this.address=address;
}//constructor ending.
```

Favor Immutability

- While ensuring immutability you need to careful about a few things.
  - Mutable Objects that are extendable must be effectively and carefully copied(Date).
  - Immutable Objects that are extendable must be effectively and carefully copied(they are even more dangerous because we may get careless with them).
    - BigDecimal,BigInteger are examples of such java classes.
    - It was not widely understood that immutable classes had to be effectively final when BigInteger and BigDecimal were written, so all of their methods may be overridden.
    - Unfortunately, this could not be corrected after the fact while preserving upward compatibility. If you write a class whose security depends on the immutability of a BigInteger or BigDecimal argument from an untrusted client, you must check to see that the argument is a"real" BigInteger or BigDecimal, rather than an instance of an untrusted subclass.

Favor Immutability

- If it is the latter, you must defensively copy it under the assumption that it might be mutable(Like the Address was copied).

- And ensuring that values passed by clients are exclusive and clients cannot gain access to mutable Objects through constructors,accessor methods, clone or readObject method(Later).
- However if the class(Address after fixing it) is effectively immutable like the String class then it is a pure joy to deal with.And no matter how incompetent or malicious a programmer he cant go wrong with it.

- Finally Immutable Object can also share there internals(String is a fantastic example again.You could also make the employees share the addresses if the address is fixed).
- They make absolutely fantastic building blocks for other complex objects mutable or immutable.

Favor Immutability

- The one minor disadvantage **of immutable classes is that they require a separate object for each distinct value.** Creating these objects can be costly, especially if they are large.
- The performance problem is magnified if you perform a multistep operation that generates a new object at every step, eventually discarding all objects except the final result. There are two approaches to coping with this problem.
  - The first is to guess which multistep operations will be commonly required and provide them as primitives. If a multistep operation is provided as a primitive, the immutable class does not have to create a separate object at each step.
    - Internally, the immutable class can be arbitrarily clever. For example, BigInteger has a package-private mutable "companion class" that it uses to speed up multistep operations such as modular exponentiation. It is much harder to use the mutable companion class for all of the reasons outlined earlier, but luckily you don't have to. The implementors of BigInteger did all the hard work for you.

Favor Immutability

- If not, then your best bet is to provide *a public* mutable companion class. The main example of this approach in the Java platform libraries is the String class, whose mutable companion is StringBuffer. Arguably, BitSet plays the role of mutable companion to BigInteger under certain circumstances.

Favor Immutability

- If not, then your best bet is to provide *a public* mutable companion class. The main example of this approach in the Java platform libraries is the String class, whose mutable companion is StringBuffer. Arguably, BitSet plays the role of mutable companion to BigInteger under certain circumstances.

- To summarize, resist the urge to write a set method for every get method. **Classes should be immutable unless there's a very good reason to make them mutable.** (There are several classes in the Java platform libraries, such as java.uti1.Date and java.awt.Point, that should have been immutable but aren't.)
- There are some classes for which immutability is impractical, including "process classes" such as Thread and TimerTask. **If a class cannot be made immutable, you should still limit its mutability as much as possible.** Reducing the number of states in which an object can exist makes it easier to reason about the object and reduces the likelihood of errors.

Favor Composition over Inheritance

- Inheritance is a powerful way to achieve code reuse, but it is not always the best tool for the job. Used inappropriately, it leads to fragile software.
- It is safe to use inheritance within a package, where the subclass and the superclass implementation are under the control of the same programmers.
- It is also safe to use inheritance when extending classes specifically designed and documented for extension .
- Inheriting from ordinary concrete classes across package boundaries, however, is dangerous. As a reminder, the word "inheritance" to mean *implementation inheritance* (when one class extends another). The problems discussed in this item do not apply to *interface inheritance* (when a class implements an interface or where one interface extends another).

Favor Composition over Inheritance

- **Unlike method invocation, inheritance breaks encapsulation** [Snyder86]. In other words, a subclass depends on the implementation details of its superclass for its proper function.
- The superclass's implementation may change from release to release, and if it does, the subclass may break, even though its code has not been touched.
- As a consequence, a subclass must evolve in tandem with its superclass, unless the superclass's authors have designed and documented it specifically for the purpose of being extended.

Favor Composition over Inheritance

- To make this concrete, let's suppose we have a program that uses a HashSet. To tune the performance of our program, we need to query the HashSet as to how many elements have been added since it was created (not to be confused with its current size, which goes down when an element is removed).
- To provide this functionality, we write a HashSet variant that keeps count of the number of attempted element insertions and exports an accessor for this count. The HashSet class contains two methods capable of adding elements, add and addAll, so we override both of these methods:

Favor Composition over Inheritance

```
// Broken - Inappropriate use of inheritance!
public class InstrumentedHashSet extends HashSet {
    // The number of attempted element insertions
    private int addCount = 0;
    public InstrumentedHashSet() { }
    public InstrumentedHashSet(Collection c) {
        super(c);
    }
    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }
```

Favor Composition over Inheritance

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

    //This class looks reasonable, but it doesn't work. Suppose we create an instance
    //and add three elements using the addAl l method:

    InstrumentedHashSet s = new InstrumentedHashSet(); s.addAll(Arrays.asList(new
    String[] {"Snap","Crackle","Pop"}));

    // We would expect the getAddCount method to return three at this point, but it
    //returns six. What went wrong? Internally, HashSet's addAll method is
    //implemented on top of its add method, although HashSet, quite reasonably, does
    //not document this implementation detail.
```

Favor Composition over Inheritance

- We could "fix" the subclass by eliminating its override of the addAll method. While the resulting class would work, it would depend for its proper function on the fact that HashSet's addAll method is implemented on top of its add method.
- This "self-use" is an implementation detail, not guaranteed to hold in all implementations of the Java platform and subject to change from release to release. Therefore, the resulting InstrumentedHashSet class would be fragile.
- It would be slightly better to override the addAll method to iterate over the specified collection, calling the add method once for each element.

Favor Composition over Inheritance

- This would guarantee the correct result whether or not HashSet's addAll method were implemented atop its add method because HashSet's addAll implementation would no longer be invoked.
- **This technique, however, does not solve all our problems. It amounts to reimplementing superclass methods that may or may not result in self-use, which is difficult, time-consuming, and error prone.**
- **Additionally, it isn't always possible, as some methods cannot be implemented without access to private fields inaccessible to the subclass.**

Favor Composition over Inheritance

- A related cause of fragility in subclasses is that their superclass can acquire new methods in subsequent releases. 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 adding an element is added to the superclass in a subsequent release. Once this happens, it becomes possible to add an "illegal" element to an instance of the subclass merely by invoking the new method, which is not overridden in the subclass.
- This is not a purely theoretical problem. Several security holes of this nature had to be fixed when Hashtable and Vector were retrofitted to participate in the Collections Framework.

Favor Composition over Inheritance

- Both of the above problems stem from overriding methods. 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. 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.

Favor Composition over Inheritance

- Luckily, there is a way to avoid all of the problems described earlier. Instead of extending an existing class, give your new class a private field that references an instance of the existing class.
- This design is called *composition* because the existing class becomes a component of the new one. Each instance method in the new class invokes the corresponding method on the contained instance of the existing class and returns the results.
- This is known as *forwarding,* and the methods in the new class are known as *forwarding methods.* The resulting class will be rock solid, with no dependencies on the implementation details of the existing class.
- Even adding new methods to the existing class will have no impact on the new class. To make this concrete, here's a replacement for InstrumentedHashSet that uses the composition/forwarding approach:

Favor Composition over Inheritance

```
public class InstrumentedSet implements Set {
private final Set s;
private int addCount = 0;
    public InstrumentedSet(Set s) { this.s = s;}
    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() { return addCount; }
    // Forwarding methods
    public void clear() { s.clear(); }
    public boolean contains(Object o) { return s.contains(o); }
    // all methods not shown
    ..................
}
```

Favor Composition over Inheritance

- Besides being robust, this design is extremely flexible. The InstrumentedSet class implements the Set interface and has a single constructor whose argument is also of type Set.
- In essence, the class transforms one Set into another, adding the instrumentation functionality.
- Unlike the inheritance-based approach, which works only for a single concrete class and requires a separate constructor for each supported constructor in the superclass, the wrapper class can be used to instrument any Set implementation and will work in conjunction with any preexisting constructor.

```
Set sl = new InstrumentedSet(new TreeSet(list));
Set s2 = new InstrumentedSet(new HashSet(capacity, loadFactor));
```

Favor Composition over Inheritance

- 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. If you are tempted to have a class *B* extend a class A, ask yourself the question: "Is every *B* really an A?" If you cannot truthfully answer yes to this question, *B* should not extend A
  - 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 appropriate.
    - And by the time they(JAVA Designers) figured this out it was too late.

Favor Composition over Inheritance

- If you use inheritance where composition is appropriate, you needlessly expose implementation details. The resulting API ties you to the original implementation, forever limiting the performance of your class.
- More seriously, by exposing the internals you let the client access them directly. At the very least, this can lead to confusing semantics.
- There is one last set of questions you should ask yourself before deciding to use inheritance rather than composition.
  - Does the class that you're contemplating extending have any flaws in its API? If so, are you comfortable propagating those flaws into the API of your class? Inheritance propagates any flaws in the superclass's API, while composition allows you to design a new API that hides these flaws.

Favor Composition over Inheritance

- To summarize, inheritance is powerful, but it is problematic because it violates encapsulation. It is appropriate only when a genuine subtype relationship exists between the subclass and the superclass. Even then, inheritance may lead to fragility if the subclass is in a different package from the superclass and the superclass is not designed for extension. To avoid this fragility, use composition

Design and document for Inheritance or prohibit it

- We alerted you to the dangers of subclassing a"foreign" class that was not designed and documented for inheritance. So what does it mean for a class to be designed and documented for inheritance?
- First, **the class must document precisely the effects of overriding any method.** In other words, the class must document its *self-use of* overridable methods: For each public or protected method or constructor, its documentation must indicate which overridable methods it invokes, in what sequence, and how the results *of* each invocation affect subsequent processing. (By *overridable,* we mean nonfinal and either public or protected.)
- More generally, a class must document any circumstances under which it might invoke an overridable method. For example, invocations might come from background threads or static initializers.

Design and document for Inheritance or prohibit it

- Here's an example, copied from the specification for java.util.AbstractCollection
  - public boolean remove(Object o)
  - Removes a single instance *of* the specified element from this collection, *if* it is present (optional operation). More formally, removes an element e such that (o==null ? e==null : o.equals(e)), if the collection contains one or more such elements. Returns true if the collection contained the specified element.
  - This implementation iterates over the collection looking for the specified element. If it finds the element, it removes the element from the collection using the iterator's remove method. Note that this implementation throws an UnsupportedOperationException if the iterator returned by this collection's iterator method does not implement the remove method.

Design and document for Inheritance or prohibit it

- This document leaves no doubt that overriding the “iterator()” method will affect the behavior of remove method and the details thereof.
- Contrast this to previous situation where the programmer subclassing HashSet simply could not tell whether overriding the “add()” method would affect the “addall()” behavior.
- **But doesn't this violate the dictum that good API documentation should describe *what a* given method does and not how it does it? Yes it does! This is an unfortunate consequence of the fact that inheritance violates encapsulation.**
- **To document a class so that it can be safely subclassed, you must describe implementation details that should otherwise be left unspecified.**
- Another example being the removeRange() from AbstractList(refer javadoc) .

Design and document for Inheritance or prohibit it

- So how do you decide what protected methods or fields to expose when designing a class for inheritance? Unfortunately, there is no magic bullet.
- The best you can do is to think hard, take your best guess, and then test it by writing some subclasses. You should provide as few protected methods and fields as possible because each one represents a commitment to an implementation detail.
- On the other hand, you must not provide too few, as a missing protected method can render a class practically unusable for inheritance.
- When you design for inheritance a class that is likely to achieve wide use, realize that you are committing *forever* to the self-use patterns that you document and to the implementation decisions implicit in its protected methods and fields.
- These commitments can make it difficult or impossible to improve the performance or functionality of the class in a subsequent release

Design and document for Inheritance or prohibit it

- However having decided to go for inheritance you must take special care.
  - There are a few more restrictions that a class must obey to allow inheritance. **Constructors must not invoke overridable methods,** directly or indirectly. If this rule is violated, it is likely that program failure will result.

Design and document for Inheritance or prohibit it

```
public class Super {// Broken - constructor invokes overridable method
    public Super() {
        m();
    }
    public void m() { }
}
final class Sub extends Super {
    private final Date date; // Blank final, set by constructor
    Sub() {
        date = new Date();
    }
    public void m() {
        System.out.println(date);
    }
    public static void main(String[] args) {
        Sub s = new Sub();
        s.m();
    }
}
```

Use Serialization Judiciously

- Allowing a class's instances to be serialized can be as simple as adding the words "implements Serializable" to its declaration.
- Because this is so easy to do, there is a common misconception that serialization requires little effort on the part of the programmer. The truth is far more complex.
- While the immediate cost to make a class serializable can be negligible, the long-term costs are often substantial.

- **A major cost of implementing Serializable is that it decreases the flexibility to change a class's implementation once it has been released.**
  - When a class implements Serializable, its byte-stream encoding (or *serialized form)* becomes part of its exported API.
  - Once you distribute a class widely, you are generally required to support the serialized form forever, just as you are required to support all other parts of the exported API.

Use Serialization Judiciously

- If you accept the default serialized form and later change the class's internal representation, an incompatible change in the serialized form may result. Clients attempting to serialize an instance using an old version of the class and deserialize it using the new version will experience program failures.
- It is possible to change the internal representation while maintaining the original serialized form (using Object0utputStream. putFields and ObjectlnputStream.readFields)
- But it can be difficult and leaves visible warts in the source code.
- **Therefore you should carefully design a high-quality serialized form that you are willing to live with for the long haul. Doing so will add to the cost of development, but it is worth the effort.**

Use Serialization Judiciously

- A simple example of the constraints on evolution that accompany serializability concerns *stream unique identifiers,* more commonly known as ***serial version UIDs***.
- Every serializable class has a unique identification number associated with it. If you do not specify the identification number explicitly by declaring a private static final long field named serialVersionUID, the system automatically generates it by applying a complex deterministic procedure to the class.
- 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 he broken.

Use Serialization Judiciously

- **A second cost of implementing Serial izable is that it increases the likelihood of bugs and security holes.**
  - Normally, objects are created using constructors: serialization is an *extra-linguistic(constructor not called) mechanism* for creating objects.
  - Whether you accept the default behavior or override it, deserialization is a "hidden constructor" with all of the same issues as other constructors.
  - Because there is no explicit constructor, it is easy to forget that you must ensure that deserialization guarantees all of the invariants established by real constructors and that it does not allow an attacker to gain access to the internals of the object under construction.
  - Relying on the default deserialization mechanism can easily leave objects open to invariant corruption and illegal access

Use Serialization Judiciously

- **A third cost of implementing Serializable is that it increases the testing burden associated with releasing a new version of a class.**
  - 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.
  - The amount of testing required `is` thus proportional to the product of the number of serializable classes and the number of releases, which can be large.

Use Serialization Judiciously

- **Implementing the Serializable interface is not a decision to be under *taken lightly*.**
  - *It offers real benefits: It is essential if a class is to participate in* some framework that relies on serialization for object transmission or persistence.
  - Furthermore, it greatly eases the use of a class as a component in another class that must implement Serializable.
  - There are, however, many real costs associated with implementing Serializable. Each time you implement a class, weigh the costs against the benefits.
  - 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.

Use Serialization Judiciously

- **Classes designed for inheritance should rarely implement Serializable, and interfaces should rarely extend it.**
  - Violating this rule places a significant burden on anyone who extends the class or implements the interface. There are times when it is appropriate to violate the rule.
  - For example, if a class or interface exists primarily to participate in some framework that requires all participants to implement Serializable, then it makes perfect sense for the class or interface to implement or extend Serializable.
  - There is one caveat regarding the decision *nor* 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.**

Use Serialization Judiciously

- Often this requires no effort because many classes designed for inheritance have no state, but this is not always the case.
  - It is best to create objects with all of their invariants already established. If client-provided information is required to establish these invariants, this precludes the use of a parameterless constructor. Naively adding a parameterless constructor and an initialization method to a class whose remaining constructors establish its invariants would complicate the class's state-space, increasing the likelihood of error.

Use Serialization Judiciously

- Here is a way to add a parameterless constructor to a nonserializable extendable class that avoids these deficiencies. Suppose the class has one constructor:

```
public AbstractFoo(int x,int y) { ... }
```

- The following transformation adds a protected parameterless constructor and an initialization method. The initialization method has the same parameters as the normal constructor and establishes the same invariants:

Use Serialization Judiciously

```
class AbstractFoo {
    private int x, y; // The state
    private Boolean initialized = false;

    public AbstractFoo(int x, int y) { initialize(x, y); }
    /**
        This constructor and the following method allow subclass's
        readObject method to initialize our internal state.
    **/

    protected AbstractFoo() { }

    protected final void initialize(int x, int y) {
        if (initialized)throw new IllegalStateException("Already initialized");
        this.x = x; this.y = y;
        // Do anything else the original constructor did
        initialized = true;
    }
```

Use Serialization Judiciously

```
/**
        These methods provide access to internal state so it can
        be manually serialized by subclass's writeObject method.
    **/
    protected final int getX() { return x; }
    protected final int getY() { return y; }

    // Must be called by all public instance methods
    private void checkInit() throws IllegalStateException
    {
        if (!initialized)
            throw new IllegalStateException("Uninitialized");
    }
    ... // Remainder omitted
}
```

Use Serialization Judiciously

- All instance methods in AbstractFoo must invoke checkInit before going about their business. This ensures that method invocations fail quickly and cleanly if a poorly written subclass fails to initialize an instance. With this mechanism in place, it is reasonably straightforward to implement a serializable subclass:

Use Serialization Judiciously

```
//Serializable subclass of nonserializable stateful class
public class Foo extends AbstractFoo implements Serializable {
    private void readObject(ObjectInputStream s)throws IOException,
    ClassNotFoundException
    {       s.defaultReadtObject();
            // Manually deserialize and initialize superclass state
            int x = s.readInt();
            int y = s.readInt();
            initialize(x, y);
    }
    private void writeObject(ObjectOutputStream s)throws IOException
    {       s.defaultWriteObject();
            // Manually serialize superclass state
            s.writeInt(getX());
            s.writeInt(getY());
    }
    // Constructor does not use the fancy mechanism public Foo(int x, int y) { super(x,
    y);}
}
```

Use Serialization Judiciously

- *Inner classes* should rarely, if ever, implement Serializable. They use compiler-generated *synthetic fields* to store references to *enclosing instances* and to store values of local variables from enclosing scopes.
- How these fields correspond to the class definition is unspecified, as are the names of anonymous and local classes.
- Therefore, the default serialized form of an inner class is ill-defined.
- *A static member class* can, however, implement Serializable.

- To summarize, the ease of implementing Serializable a is specious. Unless a class is to be thrown away after a short period of use, implementing Serializable is a serious commitment that should be made with care.

Consider using a custom serialized form

- **Do not accept the default serialized form without first considering whether it is appropriate.**
  - Accepting the default serialized form should be a conscious decision on your part that this encoding is reasonable from the standpoint of flexibility, performance, and correctness. Generally speaking, you should accept the default serialized form only if it is largely identical to the encoding that you would choose if you were designing a custom serialized form.
  - The default serialized form of an object is a reasonably efficient encoding of the **physical** representation of the object graph rooted at the object.
  - In other words, it describes the data contained in the object and in every object that is reachable from this object.
  - It also describes the topology by which all of these objects are interlinked. The ideal serialized form of an object contains only the *logical* data represented by the object. It is independent of the physical representation.

Consider using a custom serialized form

- **The default serialized form is likely to be appropriate if an object's physical representation is identical to its logical content.**
  - For example, the default serialized form would be reasonable for the following class, which represents a person's name:

```
// Good candidate for default serialized form
public class Name implements Serializable {
    /**
    * Last name. Must be non-null.
    * @serial
    */
    private String lastName;
    /**
    * First name. Must be non-null.
    * @serial
    */
    private String firstName;
}
```

Consider using a custom serialized form

- **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 could ensure that lastName and firstName were non-null. (discussed later)

Consider using a custom serialized form

```
// Awful candidate for default serialized form
public class StringList implements Serializable {
    private int size = 0;
    private Entry head = null;
    private static class Entry implements Serializable {
        String data;
        Entry next;
        Entry previous;
    }
    ... // Remainder omitted
}
```

Consider using a custom serialized form

- Logically speaking, this class represents a sequence of strings. Physically, it represents the sequence as a doubly linked list. If you accept the default serialized form, the serialized form will painstakingly mirror every entry in the linked list and all the links between the entries, in both directions.
- **Using the default serialized form when an object's physical representation differs substantially from its logical data content has four disadvantages:**
  - **permanently ties the exported API to the internal representation.**
  - **can consume excessive space**
  - **can consume excessive time.**
  - **can cause stack overflows.**

Consider using a custom serialized form

- default serialization procedure performs a recursive traversal of the object graph, which can cause stack overflows even for moderately sized object graphs. Serializing a StringList instance with 1200 elements causes the stack to overflow on my machine.

- A reasonable serialized form for StringList is simply the number of strings in the list, followed by the strings themselves. This constitutes the logical data represented by a StringList, stripped of the details of its physical representation

Consider using a custom serialized form

```
// StringList with a reasonable custom serialized form
public class StringList implements Serializable {
    private transient int size   = 0;
    private transient Entry head = null;
    // No longer Serializable!
    private static class Entry {
        String data;
        Entry next;
        Entry previous;
    }
    // Appends the specified string to the list
    public void add(String s) { ... }
```

Consider using a custom serialized form

```
/**
*       Serialize this <tt>StringList</tt> instance. *
*       @serialData The size of the list (the number of strings
*       it contains) is emitted (<tt>int</tt>), followed by all of
*       its elements (each a <tt>String</tt>), in the proper
*       sequence.
*/
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);
}
```

Consider using a custom serialized form

```
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.readl)bject());
    }

    ... // Remainder omitted
}
```

Consider using a custom serialized form

- Note that the writeObject method invokes defaultWrite0bject and the readObject method invokes defaultReadObject, even though all of Stringlist's fields are transient.
- **If all instance fields are transient, it is technically permissible to dispense with invoking defaultWriteAbject and defaultReadObject, but it is not recommended.**
- 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.
- If an instance is serialized in a later version and deserialized in an earlier version, the added fields will be ignored. Had the earlier version's readObject method failed to invoke defaultReadObject, the deserialization would fail with a StreamCorruptedException.

Consider using a custom serialized form

- Whether or not you use the default serialized form, every instance field that is not labeled transient will be serialized when the defaultWriteObject method is invoked.
- Therefore every instance field that can be made transient should be made so. This includes redundant fields. whose values can be computed from "primary data fields." such as a cached hash value. It also includes fields whose values.

- Default serialization was bad for StringList, but it could have been worse for other classes like hashTable.

Consider using a custom serialized form

- c**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 . There is also a small performance benefit. If no serial version UID is provided, an expensive computation is required to generate one at run time.
  - Declaring a serial version UID is simple. Just add this line to your class:

private static final long serialVersionUID = *randomlongValue ;*

Consider using a custom serialized form

- **Write readObject methods defensively as you would have written a constructor**
  - This class goes to great lengths to preserve its invariants and its immutability by defensively copying Date objects in its constructor and accessors.

```
// Immutable class that uses defensive copying
public final class Period {
    private final Date start;
    private final Date end;
    /**
    * @param start the beginning of the period.
    * @param end the end of the period; must not precede start.
    * @throws IllegalArgument if start is after end.
    * @throws Nu1lPointerException if start or end is null.
    */
```

Consider using a custom serialized form

```
public Period(Date start, Date end) {
      this start = new Date(start.getTime());
      this.end    = new Date(end.getTimeo);
      if (this.start.compareTo(this.end) > 0)throw new
    I1legalArgumentException(start +" > "+ end);
    }
    public Date start () {
      return (Date) start.clone();
    }
    public Date end ()        {
      return (Date) end clone();
    }
    public String to5tring() { return start +        -              + end; }
    ... // Remainder omitted
}
```

Consider using a custom serialized form

- Suppose you decide that you want this class to be serializable. Because the physical representation of a Period object exactly mirrors its logical data content, it is not unreasonable to use the default serialized form.
- Therefore, it might seem that all you have to do to make the class serializable is to add the words "implements serializable " to the class declaration. If you did so, however, the class would no longer guarantee its critical invariants.
  - **The problem is that the readObject method is effectively another public constructor only difference being that it takes byte stream as parameter, and it demands all of the same care as any other constructor. Just as a constructor must check its arguments for validity and make defensive copies.**

Consider using a custom serialized form

```
public class BogusPeriod {
//     Byte stream could not have come from real Period instance
      private static final byte[] serializedForm = new byte[] {
            (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 000, 0x78 };

      public static void main(String[] args) {
            Period p = (Period) deserialize(serializedForm);
            System.out.println(p);
      }
//     Returns the object with the specified serialized form
      public static Object deserialize(byte[] sf) {
            try {
                  InputStream is = new ByteArrayInputStream(sf);
                  ObjectInputStream ois = new ObjectInputStream(is);
                  System.out.println("");
                  return ois.readObject();
            } catch (Exception e) {}
      }
}
```

Consider using a custom serialized form

- If you run this program, it prints “
  - Fri Jan 01 12:00:00 PST 1999 - Sun Jan 01 12:00:00 PST 1984:" Making Period serializable enabled us to create an object that violates its class invariants.
  - To fix this problem, provide a readObject method for Period that calls defaultReadObject and then checks the validity of the deserialized object. If the validity check fails, the readObject method throws an InvalidObjectException, preventing the deserialization from completing:

Consider using a custom serialized form

```
// Immutable class that uses defensive copying
private void readObject(ObjectInputStream s)
throws IOException, ClassNotFoundException
{
    s.defaultReadObject();
    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
    throw new InvalidObjectException(start +" after "+ end);
}
```

- While this fix prevents an attacker from creating an invalid Period instance, there is a more subtle problem still lurking.

Consider using a custom serialized form

- It is possible to create a mutable Period instance by fabricating a byte stream that begins with a byte stream representing a valid Period instance and then appends extra references to the private Date fields internal to the Period instance.
- The attacker reads the Period instance from the ObjectInputStream and then reads the "rogue object references" that were appended to the stream. These references give the attacker access to the objects referenced by the private Date fields within the Period object.
- By mutating these Date instances, the attacker can mutate the Period instance. The following class demonstrates this attack:

Consider using a custom serialized form

```
public class MutablePeriod {
    // A period instance
    public final Period period;
    // period's start field, to which we shouldn't have access
    public final Date start;
    // period's end field, to which we shouldn't have access
    public final Date end;
    public MutablePeriod() {
        try {
            ByteArrayOutputStream bos =new ByteArrayOutputStream();
            ObjectOutputStream out =new ObjectOutputStream(bos);
            //Serialize a valid Period instance
            out.writeObject(new Period(new Date(), new Date()));
            /*
             * Append rogue "previous object refs" for internal
             * Date fields in Period. For details, see "Java
             * Object Serialization Specification," Section 6.4.
            */
```

Consider using a custom serialized form

```
                byte[] ref = { 0x71, 0, 0x7e, 0, 5 };
                // Ref #5
                bos.write(ref);
                // The start field
                ref [4] = 4;
                // Ref # 4
                bos.write(ref);
                // The end field

                // Deserialize Period and "stolen" Date references
                ObjectInputStream in = new ObjectInputStream( new
ByteArrayInputStream(bos.toByteArray()));
                period = (Period) in.readObject();
                start = (Date)in.readObject();
                end= (Date)in.readObject();
        } catch (Exception e) {
                throw new RuntimeException(e.toString());
        }
}
```

Consider using a custom serialized form

```
//To see the attack in action, run the following program:
    public static void main(String[] args) {
        MutablePeriod mp = new MutablePeriod();
        Period p = mp.period;
        Date pEnd=mp.end;
        //Let's turn back the clock
        pEnd.setYear(78);
        System.out.println(p);

        //Bring back the 60's!
        pEnd.setYear(69);
        System.out.println(p);
    }
}
```

Consider using a custom serialized form

- The details are technical.
- While the Period instance is created with its invariants intact, it is possible to modify its internal components at will. Once in possession of a mutable Period instance, an attacker might cause great harm by passing the instance on to a class that depends on Period's immutability for its security.
- This is not so farfetched: There are classes that depend on String's immutability for their security.
- The source of the problem is that Period's readObject method is not doing enough defensive copying.
- **When an object is deserialized, it is critical to defensively copy any field containing an object reference that a client must not possess.**
- Therefore every serializable immutable class containing private mutable components must defensively copy these components in its readObject method. The following readObject method suffices to ensure Period's invariants and to maintain its immutability:

Consider using a custom serialized form

```
private void readObject(ObjectInputStream s) throws IOException,
    ClassNotFoundException
{
    s.defaultReadObject();
    // Defensively copy our mutable components
    start = new Date(start.getTime());
    end    = new Date (end.getTime());
    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start +" after "+ end);
}
```

Consider using a custom serialized form

- Note also that defensive copying is not possible for final fields. To use the readObject method, we must make the start and end fields nonfinal.
- This is unfortunate, but it is clearly the lesser of two evils. With the new readObject method in place and the final modifier removed from the start and end fields, the MutablePeriod class is rendered ineffective.

Consider using a custom serialized form

- There is a simple litmus test for deciding whether the default readObject method is acceptable. Would you feel comfortable adding a public constructor that took as parameters the values for each nontransient field in your object and stored the values in the fields with no validation whatsoever?
- If you can't answer yes to this question, then you must provide an explicit readObject method, and it must perform all of the validity checking and defensive copying that would be required of a constructor.
- There is one other similarity between readObject methods and constructors, concerning nonfinal serializable classes. A readObject method must not invoke an overridable method(discussed later), directly or indirectly.
- If this rule is violated and the method is overridden, the overriding method will run before the subclass's state has been deserialized. A program failure is likely to result.

Consider using a custom serialized form

- Here, in summary form, are the guidelines for writing a bulletproof readObject method:
  - For classes with object reference fields that must remain private, defensively copy each object that is to be stored in such a field. Mutable components of immutable classes fall into this category.
  - For classes with invariants, check invariants and throw an InvalidObjectException if a check fails. The checks should follow any defensive copying.
  - If an entire object graph must be validated after it is deserialized, the ObjectInputValidation interface should be used. The use of this interface is beyond the scope. A sample use may be found in *The Java Class Libraries, Second Edition, Volume 1* [Chan98, p. 12561.
  - Do not invoke any overridable methods in the class, directly or indirectly.

Consider using a custom serialized form

- **Provide a readResolve method when necessary**
  - Code describes the *Singleton* pattern and gives the following example of a singleton class. This class restricts access to its constructor to ensure that only a single instance is ever created:

```
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { }          ...
    ... // Remainder omitted
}
```

Consider using a custom serialized form

- This class would no longer be a singleton if the words "implements Serializable" were added to its declaration. It doesn't matter whether the class uses the default serialized form or a custom serialized form, nor does it matter whether the class provides an explicit readObject method.
- 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.
- Prior to the 1.2 release, it was impossible to write a serializable singleton class.
- In the 1.2 release, the readResolve feature was added to the serialization facility [Serialization, 3.61. If the class of an object being deserialized defines a readResolve method with the proper declaration, this method is invoked on the newly created object after it is deserialized.

Consider using a custom serialized form

- The object reference returned by this method is then returned in lieu of the newly created object.
- In most uses of this feature, no reference to the newly created object is retained: the object is effectively stillborn, immediately becoming eligible for garbage collection.
- If the Elvis class is made to implement Serializable, the following readResolve method suffices to guarantee the singleton property:

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

Consider using a custom serialized form

- This method ignores the deserialized object, simply returning the distinguished El vi s instance created when the class was initialized.
- Therefore the serialized form of an Elvis instance need not contain any real data; all instance fields should be marked transient. This applies not only to Elvis, but to all singletons.
- **A readResolve method is necessary not only for singletons, but for all other *instance-controlled* classes,** in other words, for all classes that strictly control instance creation to maintain some invariant.
- Another example of an instancecontrolled class is *a typesafe enum*, whose readResolve method must return the canonical instance representing the specified enumeration constant.
- As a rule of thumb, if you are writing a serializable class that contains no public or protected constructors, consider whether it requires a readResolve method.

Consider using a custom serialized form

- **A second use for the readResol ve method is as a conservative alternative to the defensive readObject method recommended before.**
  - In this approach, all validity checks and defensive copying are eliminated from the readObject method in favor of the validity checks and defensive copying provided by a normal constructor. If the default serialized form is used, the readObject method may be eliminated entirely.
  - As explained ,this allows a malicious client to create an instance with compromised invariants. However, the potentially compromised deserialized instance is never placed into active service; it is simply mined for inputs to a public constructor or static factory and discarded.

Consider using a custom serialized form

- The beauty of this approach is that it virtually eliminates the extralinguistic component of serialization, making it impossible to violate any class invariants that were present before the class was made serializable. To make this technique concrete, the following readResol ve method can be used in lieu of the defensive readObject method in the Period example :

```
// The defensive readResolve idiom. No check required because
// the constructor is getting called.
private Object readResolve() throws ObjectStreamException {
    return new Period(start, end);
}
```

Consider using a custom serialized form

- This readResolve method stops both of the attacks described above dead in their tracks. The defensive readResolve idiom has several advantages over a defensive readObject.
- It is *a mechanical* technique for making a class serializable without putting its invariants at risk. It requires little code and little thought, and it is guaranteed to work.
- **Finally, it eliminates the artificial restrictions that serialization places on the use of final fields.**

Consider using a custom serialized form

- **While the defensive readResolve idiom is not widely used, it merits serious consideration.**
- Its major disadvantage is that it is not suitable for classes that permit inheritance outside of their own package.
- This is not an issue for immutable classes, as they are generally final.
- A minor disadvantage of the idiom is that it slightly reduces deserialization performance because it entails creating an extra object. On my machine, it slows the deserialization of Period instances by about one percent when compared to a defensive readObject method.

Consider using a custom serialized form

- **The accessibility of the readResolve method is significant.**
- If you place a readResolve method on a final class, such as a singleton, it should be private.
- If you place a readResolve method on a nonfinal class, you must carefully consider its accessibility.
- If it is private, it will not apply to any subclasses.
- If it is packageprivate, it will apply only to subclasses in the same package.
- If it is protected or public, it will apply to all subclasses that do not override it.

Consider using a custom serialized form

- If a readResolve method is protected or public and a subclass does not override it, deserializing a serialized subclass instance will produce a superclass instance, which is probably not what you want.
- To summarize, the above are the reasons the readResolve method may not be substituted for a defensive readObject method in classes that permit inheritance. If the superclass's readResolve method were final, it would prevent subclass instances from being properly deserialized. If it were overridable, a malicious subclass could override it with a method returning a compromised instance.
- You must use a readResolve method to protect the "instancecontrol invariants" of singletons and other instance-controlled classes.
- In essence, the readResolve method turns the readObject method from a de facto public constructor into a de facto public static factory . The readResolve method is also useful as a simple alternative to a defensive readObject method for classes that prohibit inheritance outside their package.

Replace enum with classes

- The C enum construct was omitted from the Java programming language. Nominally, this construct defines an *enumerated type:* a type whose legal values consist of a fixed set of constants.
- Unfortunately, the enum construct doesn't do a very good job of defining enumerated types. It just defines a set of named integer constants, providing nothing in the way of type safety and little in the way of convenience. Not only is the following legal C:

```
typedef enum {FUJI, PIPPIN, GRANNY_SMITH} apple_t;
typedef enum {NAVEL, TEMPLE, BLOOD} orange_t;
orange_t myFavorite = PIPPIN;          /* Mixing apples and oranges */
//but so is this atrocity:
orange_t x = (FUJI - PIPPIN)/TEMPLE;  /* Applesauce! */
```

Replace enum with classes

- Types defined with the enum construct are brittle. Adding constants to such a type without recompiling its clients causes unpredictable behavior, unless care is taken to preserve all of the preexisting constant values.
- Multiple parties cannot add constants to such a type independently, as their new enumeration constants are likely to conflict.
- The enum construct provides no easy way to translate enumeration constants into printable strings or to enumerate over the constants in a type.
- Unfortunately, the most commonly used pattern for enumerated types in the Java programming language, shown here, shares the shortcomings of the C enum construct:

Replace enum with classes

```
// The int enum pattern - problematic!!
public class PlayingCard {
    public static final int SUIT-CLUBS    = 0;
    public static final int SUIT-DIAMONDS = 1;
    public static final int SUIT-HEARTS   = 2;
    public static final int SUIT-SPADES   = 3;

}
```

Replace enum with classes

- You may encounter a variant of this pattern in which String constants are used in place of int constants. This variant should never be used. While it does provide printable strings for its constants, it can lead to performance problems because it relies on string comparisons.
- Furthermore, it can lead naive users to hard-code string constants into client code instead of using the appropriate field names. If such a hard-coded string constant contains a typographical error, the error will escape detection at compile time and result in bugs at run time.
- **Luckily, the Java programming language presents an alternative that avoids all the shortcomings of the common i nt and String patterns and provides many added benefits.**
- It is called the *typesafe enum* pattern. Unfortunately, it is not yet widely known. The basic idea is simple: Define a class representing a single element of the enumerated type, and don't provide any public constructors.
- Instead, provide public static final fields, one for each constant in the enumerated type. Here's how the pattern looks in its simplest form:

Replace enum with classes

```
// The typesafe enum pattern
public class Suit {
    private final String name;
    private Suit(String name) {
        this.name = name;
    }
    public String toString()
    {
        return name;
    }
    public static final Suit CLUBS    = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS   = new Suit("hearts");
    public static final Suit SPADES   = new Suit("spades");
}
```

Replace enum with classes

- Because there is no way for clients to create objects of the class or to extend it, there will never be any objects of the type besides those exported via the public static final fields.
- Even though the class is not declared final, there is no way to extend it: Subclass constructors must invoke a superclass constructor, and no such constructor is accessible.
- As its name implies, the typesafe enum pattern provides compile-time type safety. If you declare a method with a parameter of type Suit, you are guaranteed that any non-null object reference passed in represents one of the four valid suits.
- Any attempt to pass an incorrectly typed object will be caught at compile time, as will any attempt to assign an expression of one enumerated type to a variable of another.
- Multiple typesafe enum classes with identically named enumeration constants coexist peacefully because each class has its own name space.

Replace enum with classes

- Constants may be added to a typesafe enum class without recompiling its clients because the public static object reference fields containing the enumeration constants provide a layer of insulation between the client and the enum class.
- The constants themselves are never compiled into clients as they are in the more common int enum pattern and its String variant.
- Because typesafe enums are full-fledged classes, you can override the toString method as shown earlier, allowing values to be translated into printable strings.
- You can, if you desire, go one step further and internationalize typesafe enums by standard means.
- **Note that string names are used only by the toString method; they are not used for equality comparisons, as the equals implementation, which is inherited from Object, performs a reference identity comparison.**

Replace enum with classes

- More generally, you can augment a typesafe enum class with any method that seems appropriate. Our Suit class, for example, might benefit from the addition of a method that returns the color of the suit or one that returns an image representing the suit.
- A class can start life as a simple typesafe enum and evolve over time into a full-featured abstraction.
- Because arbitrary methods can be added to typesafe enum classes, they can be made to implement any interface. For example, suppose that you want Suit to implement Comparable so clients can sort bridge hands by suit.
- Here's a slight variant on the original pattern that accomplishes this feat. A static variable, nextOrdinal, is used to assign an ordinal number to each instance as it is created. These ordinals are used by the compareTo method to order instances:

Replace enum with classes

```
// Ordinal-based typesafe enum
public class Suit implements Comparable {
    private final String name;

    // Ordinal of next suit to be created
    private static int nextOrdinal = 0;

    // Assign an ordinal to this suit
    private final int ordinal = nextOrdinal++;

    private Suit(String name) {
        this.name = name;
    }
    public String toString() {
        return name;
    }
```

Replace enum with classes

```
    public int compareTo(Object o) {
            return ordinal - ((Suit)o).ordinal;
    }
    public static final Suit CLUBS    = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS   = new Suit("hearts");
    public static final Suit SPADES   = new Suit("spades");

    private static final Suit[] PRIVATE VALUES ={ CLUBS, DIAMONDS, HEARTS,
    SPADES };
    public static final List VALUES
    =Collections.unmodifiableList(Arrays.asList(PRIVATE VALUES));

}
```

Replace enum with classes

- Because typesafe enum constants are objects, you can put them into collections. For example, suppose you want the Suit class to export an immutable list of the suits in standard order.
- Merely add these two field declarations to the class:

```
private static final Suit[] PRIVATE VALUES ={ CLUBS, DIAMONDS,
HEARTS, SPADES };
public static final List VALUES
=Collections.unmodifiableList(Arrays.asList(PRIVATE VALUES));
```

Replace enum with classes

- Unlike the simplest form of the typesafe enum pattern, classes of the ordinalbased form above can be made serializable with a little care. It is not sufficient merely to add i mpl ements Serializable to the class declaration. You must also provide a readResolve method

```
private Object readResolve() throws ObjectStreamException {
        return PRIVATE VALUES[ordinal]; // Canonicalize
}
```

Replace enum with classes

- This method, which is invoked automatically by the serialization system, prevents duplicate constants from coexisting as a result of deserialization.
- This maintains the guarantee that only a single object represents each enum constant, avoiding the need to override Object. equals. Without this guarantee, Object equals would report a false negative when presented with two equal but distinct enumeration constants.
- Note that the readResolve method refers to the PRIVATE VALUES array, so you must declare this array even if you choose not to export VALUES. Note also that the name field is not used by the readResolve method, so it can and should be made transient.

Replace enum with classes

- The resulting class is somewhat brittle; constructors for any new values must appear after those of all existing values, to ensure that previously serialized instances do not change their value when they're deserialized.
- This is so because the serialized form of an enumeration constant consists solely of its ordinal. If the enumeration constant pertaining to an ordinal changes, a serialized constant with that ordinal will take on the new value when it is deserialized.

Replace enum with classes

- There may be one or more pieces of behavior associated with each constant that are used only from within the package containing the typesafe enum class.
- Such behaviors are best implemented as package-private methods on the class. Each enum constant then carries with it a hidden collection of behaviors that allows the package containing the enumerated type to react appropriately when presented with the constant.
- If a typesafe enum class has methods whose behavior varies significantly from one class constant to another, you should use a separate private class or anonymous inner class for each constant.
- This allows each constant to have its own implementation of each such method and automatically invokes the correct implementation.

Replace enum with classes

```
// Typesafe enum with behaviors attached to constants
public abstract class Operation {
    private final String name;
    Operation(String name)      {
            this.name  name;
    }
    public String toString() {
            return this name;
    }
    // Perform arithmetic operation represented by this constant
    abstract double eval(double x, double y);
    public static final Operation PLUS = new Operation("+") {
                        double eval(double x, double y) {
                                return x + y;
                        }
    };
```

Replace enum with classes

```
    public static final Operation MINUS   new Operation("-") {
            double eval(double x, double y) {
                    return x - y;
            }
    };
    public static final Operation TIMES = new Operation("*") {
            double eval(double x, double y) {
                    return x * y;
            }
    };
    public static final Operation DIVIDED-BY=new Operation("/") {
            double eval(double x, double y) {
                    return x / y;
            }
    };
}
```

Replace enum with classes

- Typesafe enums are, generally speaking,comparable in performance to int enumeration constants. Two distinct instances of a typesafe enum class can never represent the same value, so reference identity comparisons, which are fast, are used to check for logical equality.
- Clients of a typesafe enum class can use the == operator instead of the equals method; the results are guaranteed to be identical, and the == operator may be even faster.

Replace enum with classes

- The basic typesafe enum pattern, as exemplified by both Suit implementations shown earlier, is *fixed:* It is impossible for users to add new elements to the enumerated type, as its class has no user-accessible constructors.
- To make a typesafe enum extensible, merely provide a protected constructor. Others can then extend the class and add new constants to their subclasses.
- You needn't worry about enumeration constant conflicts as you would if you were using the int enum pattern. The extensible variant of the typesafe enum pattern takes advantage of the package namespace to create a "magically administered" namespace for the extensible enumeration.
- Multiple organizations can extend the enumeration without knowledge of one another, and their extensions will never conflict.

Replace enum with classes

- It is a good idea for extensible typesafe enum classes to override the equals and hashCode methods with final methods that invoke the Object methods.
- This ensures that no subclass accidentally overrides these methods, maintaining the guarantee that all equal objects of the enumerated type are also identical (a. equals (b) if and only if a==b):

```
// Override-prevention methods
public final boolean equals(Object that) {
    return super.equals(that);
}
public final int hashCode() {
    return super.hashCode<);
}
```

Replace enum with classes

- Note that the extensible variant is not compatible with the comparable variant; if you tried to combine them, the ordering among the elements of the subclasses would be a function of the order in which the subclasses were initialized, which could vary from program to program and run to run.
- The extensible variant of the typesafe enum pattern is compatible with the serializable variant, but combining these variants demands some care.
- Each subclass must assign its own ordinals and provide its own readResolve method. In essence, each class is responsible for serializing and deserializing its own instances

Replace enum with classes

```
public abstract class Operation implements Serializable {
    private final transient String name;
    protected Operation(String name) { this.name = name;
        System.out.println("NAME :: "+name+" ORDINALVALUE :: "+ordinal);
    }
//   Perform arithmetic operation represented by this constant
    protected abstract double eval(double x, double y);
    public static final Operation PLUS = new Operation("+") {
        protected double eval (double x, double y) { return x+y; }
    };
    public static final Operation MINUS = new Operation("-") {
        protected double eval (double x, double y) { return x-y; }
    };
    public static final Operation TIMES = new Operation("*") {
        protected double eval(double x, double y) { return x*y; }
    };
    public static final Operation DIVIDE = new Operation("/") {
        protected double eval(double x, double y) { return x/y; }
    };
```

Replace enum with classes

```
    public String toString() {    return name+ "  ORDINALVALUE :: "+ordinal; }
// Prevent subclasses from overriding Object.equals
    public final boolean equals(Object that) {
        return super.equals(that);
    }
    public final int hashCode() {
        return super.hashCode();
    }
    // The 4 declarations below are necessary for serialization
    private static int nextOrdinal = 0;
    private final int ordinal = nextOrdinal++;
    private static final Operation[] VALUES ={ PLUS, MINUS, TIMES, DIVIDE };
    Object readResolve() throws ObjectStreamException {
        return VALUES[ordinal]; // Canonicalize
    }
}
```

Replace enum with classes

```
//Subclass of extensible, serializable typesafe enum
abstract class ExtendedOperation extends Operation {
    ExtendedOperation(String name) { super(name); }

    public static Operation LOG = new ExtendedOperation("log") {
            protected double eval(double x, double y) {
                    return Math.log(y) / Math.log(x);
            }
    };

    public static Operation EXP = new ExtendedOperation("exp") {
            protected double eval(double x, double y) {
                    return Math.pow(x, y);
            }
    };
```

Replace enum with classes

```
// The 4 declarations below are necessary for serialization
    private static int nextOrdinal = 0;
    private final int ordinal = nextOrdinal++;
    private static final Operation[] VALUES = { LOG, EXP };
    Object readResolve() throws ObjectStreamException {
        return VALUES[ordinal]; // Canonicalize
    }
}
```

Replace enum with classes

- Note that the readResolve methods in the classes just shown are package-private rather than private.
- This is necessary because the instances of Operation and ExtendedOperation are, in fact, instances of anonymous subclasses, so private readResolve methods would have no effect
- There are two minor disadvantages to using enums.
  - One that they cant be used in a switch statement.
  - Two the class loading time.

Replace enum with classes

- Starting with JDK 5.0, you can define your own enumerated type whenever such a situation arises. An enumerated type has a finite number of named values.
- For example :enum Size { SMALL, MEDIUM, LARGE, EXTRA-LARGE}
- Now you can declare variables of this type: Size s = Size.MEDIUM;
- A variable of type Size can hold only one of the values listed in the type declaration or the special value null that indicates that the variable is not set to any value at all.
- **It does not require any customization while serialization. It just works**

Replace enum with classes

- You can, if you like, add constructors, methods, and fields to an enumerated type• course, the constructors are only invoked when the enumerated constants are constructed.
- All enumerated types are subclasses of the class Enum. They inherit a number of methods from that class. The most useful one is toString, which returns the name of the enun constant.
- The converse of toString is the static valueOf method.

  For example, the statement

  Size s = (Size)Enum.value0f(Size.class, "SMALL");

  sets s to Size.SMALL.
- Each enumerated type has a static values method that returns an array of all values i enumeration:

  Size[] values = Size.values();