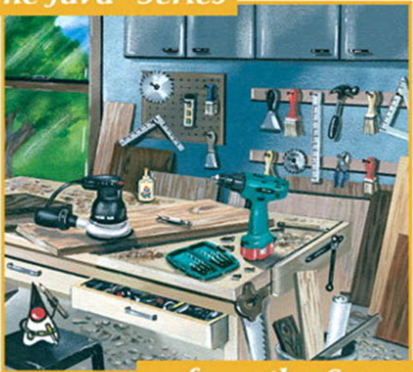
Joshua Bloch

Revised and Updated for Java SE 6

Effective Java™

Second Edition

The Java™ Series



...from the Source

Sun microsystems

Java Sun Microsystems

Item 11: Override `clone` judiciously

The `Cloneable` interface was intended as a *mixin interface* (Item 18) 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* (Item 53), 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 wide use so it pays to understand it. This item tells you how to implement a well-behaved `clone` method, discusses when it is appropriate to do so, and presents alternatives.

So what *does* `Cloneable` do, given that it contains no methods? It determines the behavior of `Object`'s protected `clone` implementation: if a class implements `Cloneable`, `Object`'s `clone` method returns a field-by-field copy of the object; otherwise it throws `CloneNotSupportedException`. This is a highly atypical use of interfaces and not one to be emulated. Normally, implementing an interface says something about what a class can do for its clients. In the case of `Cloneable`, it modifies the behavior of a protected method on a superclass.

If implementing the `Cloneable` interface is to have any effect on a class, the class and all of its superclasses must obey a fairly complex, unenforceable, and thinly documented 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` [JavaSE6]:

> 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.clone().equals(x)
> ```
>
> will be `true`, this is not an absolute requirement. 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. 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.

The `Cloneable` interface does not, as of release 1.6, spell out in detail the responsibilities that a class takes on when it implements this interface. **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.

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. This is the case, for example, for the `PhoneNumber` class in Item 9. In this case, all you need do in addition to declaring that you implement `Cloneable` is to provide public access to `Object`’s protected `clone` method:

```
@Override public PhoneNumber clone() {
    try {
        return (PhoneNumber) super.clone();
    } catch(CloneNotSupportedException e) {
        throw new AssertionError();  // Can't happen
    }
}
```

Note that the above `clone` method returns `PhoneNumber`, not `Object`. As of release 1.5, it is legal and desirable to do this, because *covariant return types* were introduced in release 1.5 as part of generics. In other words, it is now legal for an overriding method's return type to be a subclass of the overridden method's return type. This allows the overriding method to provide more information about the returned object and eliminates the need for casting in the client. Because `Object.clone` returns `Object`, `PhoneNumber.clone` must cast the result of `super.clone()` before returning it, but this is far preferable to requiring every caller of `PhoneNumber.clone` to cast the result. The general principle at play here is **never make the client do anything the library can do for the client.**

If an object contains fields that refer to mutable objects, using the simple `clone` implementation shown above can be disastrous. For example, consider the `Stack` class in Item 6:

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

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

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

    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)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

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`.

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 to call `clone` recursively on the `elements` array:

```
@Override public Stack clone() {
    try {
        Stack result = (Stack) super.clone();
        result.elements = elements.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

Note that we do not have to cast the result of `elements.clone()` to `Object[]`. As of release 1.5, calling `clone` on an array returns an array whose compile-time type is the same as that of the array being cloned.

Note also that the above solution would not work if the `elements` field were final, because `clone` would be prohibited from assigning a new value to the field. This is a fundamental problem: **the `clone` architecture is incompatible with normal use of final fields referring to mutable 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.

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. For performance, the class implements its own lightweight singly linked list instead of using `java.util.LinkedList` internally:

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;
    private static class Entry {
        final Object key;
        Object value;
        Entry  next;

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

    ... // Remainder omitted
}
```

Suppose you merely clone the bucket array recursively, as we did for `Stack`:

```
// Broken - results in shared internal state!
@Override public HashTable clone() {
    try {
        HashTable result = (HashTable) super.clone();
        result.buckets = buckets.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

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:

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

    private static class Entry {
        final 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());
        }
    }

    @Override public HashTable clone() {
        try {
            HashTable result = (HashTable) super.clone();
            result.buckets = new Entry[buckets.length];
            for (int i = 0; i < buckets.length; i++)
                if (buckets[i] != null)
                    result.buckets[i] = buckets[i].deepCopy();
            return result;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
    ... // Remainder omitted
}
```

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:

```
// Iteratively copy the linked list headed by this Entry
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;
}
```

A final approach to cloning complex objects is to call `super.clone`, set all of the fields in the resulting object to their virgin state, and then call higher-level methods to regenerate the state of the object. In the case of our `HashTable` example, the `buckets` field would be initialized to a new bucket array, and the `put(key, value)` method (not shown) would be invoked for each key-value mapping in the hash table being cloned. This approach typically yields a simple, reasonably elegant `clone` method that generally doesn't run quite as fast as one that directly manipulates the innards of the object and its clone.

Like a constructor, a `clone` method should not invoke any nonfinal methods on the clone under construction (Item 17). 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. Therefore the `put(key, value)` method discussed in the previous paragraph should be either final or private. (If it is private, it is presumably the "helper method" for a nonfinal public method.)

`Object`'s `clone` method is declared to throw `CloneNotSupportedException`, but overriding `clone` methods can omit this declaration. Public `clone` methods *should* omit it because methods that don't throw checked exceptions are easier to use (Item 59). If a class that is designed for inheritance (Item 17) overrides `clone`, the overriding method should mimic the behavior of `Object.clone`: it should be declared `protected`, it should be declared to throw `CloneNotSupportedException`, and the class should not implement `Cloneable`. This gives subclasses the freedom to implement `Cloneable` or not, just as if they extended `Object` directly.

One more detail bears noting. If you decide to make a thread-safe class implement `Cloneable`, remember that its `clone` method must be properly synchronized just like any other method (Item 66). `Object`'s `clone` method is not synchronized, so even if it is otherwise satisfactory, you may have to write a synchronized `clone` method that invokes `super.clone()`.

To recap, all classes that implement `Cloneable` should override `clone` with a public method whose return type is the class itself. This method should first call `super.clone` and then fix any fields that need to be fixed. Typically, this means copying any mutable objects that comprise the internal "deep structure" of the object being cloned, and replacing the clone's references to these objects with references to the copies. While these internal copies can generally be made by calling `clone` recursively, this is not always the best approach. If the class contains only primitive fields or references to immutable objects, then it is probably the case that no fields need to be fixed. There are exceptions to this rule. For example, a field representing a serial number or other unique ID or a field representing the object's creation time will need to be fixed, even if it is primitive or immutable.

Is all this complexity really necessary? Rarely. If you extend a class that implements `Cloneable`, you have little choice but to implement a well-behaved `clone` method. Otherwise, **you are better off providing an alternative means of object copying, or simply not providing the capability**. For example, it doesn't make sense for immutable classes to support object copying, because copies would be virtually indistinguishable from the original.

A fine approach to object copying is to provide a *copy constructor* or *copy factory*. A copy constructor is simply a constructor that takes a single argument whose type is the class containing the constructor, for example,

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

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

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

The copy constructor approach and its static factory variant have many advantages over `Cloneable/clone`: they don't rely on a risk-prone extralinguistic object creation mechanism; they don't demand unenforceable adherence to thinly documented conventions; they don't conflict with the proper use of final fields; they don't throw unnecessary checked exceptions; and they don't require casts. While it is impossible to put a copy constructor or factory in an interface, `Cloneable` fails to function as an interface because it lacks a public `clone` method. Therefore you aren't giving up interface functionality by using a copy constructor or factory in preference to a `clone` method.

Furthermore, a copy constructor or factory can take an argument whose type is an interface implemented by the class. For example, by convention all general-purpose collection implementations provide a constructor whose argument is of type `Collection` or `Map`. Interface-based copy constructors and factories, more properly known as *conversion constructors* and *conversion factories*, allow the client to choose the implementation type of the copy rather than forcing the client to accept the implementation type of the original. Suppose you have a `HashSet s`, and you want to copy it as a `TreeSet`. The `clone` method can't offer this functionality, but it's easy with a conversion constructor: `new TreeSet(s)`.

Given all of the problems associated with `Cloneable`, it's safe to say that other interfaces should not extend it, and that classes designed for inheritance (Item 17) 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. If you design a class for inheritance, be aware that if you choose not to provide a well-behaved protected `clone` method, it will be impossible for subclasses to implement `Cloneable`.