Copy constructors:

* provide an attractive alternative to the rather pathological [clone](http://www.javapractices.com/topic/Topic71.cjp) method
* are easily implemented
* simply extract the argument's data, and forward to a regular constructor
* are unnecessary for [immutable objects](http://www.javapractices.com/topic/Topic29.cjp)

**Example** 

**public** **final** **class** Galaxy {

/\*\*

\* Regular constructor.

\*/

**public** Galaxy(**double** aMass, String aName) {

fMass = aMass;

fName = aName;

}

/\*\*

\* Copy constructor.

\*/

**public** Galaxy(Galaxy aGalaxy) {

**this**(aGalaxy.getMass(), aGalaxy.getName());

//no defensive copies are created here, since

//there are no mutable object fields (String is immutable)

}

/\*\*

\* Alternative style for a copy constructor, using a static newInstance

\* method.

\*/

**public** **static** Galaxy newInstance(Galaxy aGalaxy) {

**return** **new** Galaxy(aGalaxy.getMass(), aGalaxy.getName());

}

**public** **double** getMass() {

**return** fMass;

}

/\*\*

\* This is the only method which changes the state of a Galaxy

\* object. If this method were removed, then a copy constructor

\* would not be provided either, since immutable objects do not

\* need a copy constructor.

\*/

**public** **void** setMass(**double** aMass){

fMass = aMass;

}

**public** String getName() {

**return** fName;

}

// PRIVATE

**private** **double** fMass;

**private** **final** String fName;

/\*\* Test harness. \*/

**public** **static** **void** main (String... aArguments){

Galaxy m101 = **new** Galaxy(15.0, "M101");

Galaxy m101CopyOne = **new** Galaxy(m101);

m101CopyOne.setMass(25.0);

System.out.println("M101 mass: " + m101.getMass());

System.out.println("M101Copy mass: " + m101CopyOne.getMass());

Galaxy m101CopyTwo = Galaxy.newInstance(m101);

m101CopyTwo.setMass(35.0);

System.out.println("M101 mass: " + m101.getMass());

System.out.println("M101CopyTwo mass: " + m101CopyTwo.getMass());

}

}

Example run of this class:

>java -cp . Galaxy   
M101 mass: 15.0   
M101Copy mass: 25.0   
M101 mass: 15.0   
M101CopyTwo mass: 35.0

There are several ways to copy an array:

* use the various copyOf and copyOfRange methods of the [Arrays](http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html) class - probably the simplest method
* use System.[arraycopy](http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#arraycopy(java.lang.Object, int, java.lang.Object, int, int)) - useful when copying parts of an array
* call its clone method, and do a cast - the simplest style, but only a shallow clone is performed
* use a for loop - more than one line, and needs a loop index

**Example**

This example class demonstrates:

* relative performance of the various methods (in many cases the differences in speed will not be of practical benefit).
* how clone is a shallow copy, and leads to independent storage only for primitive, one dimensional arrays.

**import** java.util.\*;

**public** **final** **class** ArrayCopier {

**public** **static** **void** main (String... aArguments) {

String action = aArguments[0];

**int** numIterations = 0;

**if** (aArguments.length == 2) {

numIterations = Integer.parseInt(aArguments[1]);

}

**if** ("performance".equals(action)) {

demoPerformance(numIterations);

}

**else** **if** ("storage".equals(action)) {

demoIndependanceOfStorage();

}

}

/\*\*

\* Display the time it takes to copy an array in various ways.

\*/

**private** **static** **void** demoPerformance(**int** aNumIterations){

Stopwatch stopwatch = **new** Stopwatch();

**int**[] numbers = {1,2,3,4,5,6,7,8,9,10};

stopwatch.start();

copyUsingClone(numbers, aNumIterations);

stopwatch.stop();

log("Using clone: " + stopwatch);

stopwatch.start();

copyUsingArraycopy(numbers, aNumIterations);

stopwatch.stop();

log("Using System.arraycopy: " + stopwatch);

stopwatch.start();

copyUsingArraysCopyOf(numbers, aNumIterations);

stopwatch.stop();

log("Using Arrays.copyOf: " + stopwatch);

stopwatch.start();

copyUsingForLoop(numbers, aNumIterations);

stopwatch.stop();

log("Using for loop: " + stopwatch);

}

**private** **static** **void** copyUsingClone(**int**[] aArray , **int** aNumIterations) {

**for**(**int** idx = 0 ; idx < aNumIterations; ++idx) {

**int**[] copy = (**int**[])aArray.clone();

}

}

**private** **static** **void** copyUsingArraycopy(**int**[] aArray , **int** aNumIterations) {

**for**(**int** idx = 0 ; idx < aNumIterations; ++idx) {

**int** [] copy = **new** **int**[aArray.length];

System.arraycopy( aArray, 0, copy, 0, aArray.length );

}

}

**private** **static** **void** copyUsingArraysCopyOf( **int**[] aArray , **int** aNumIterations) {

**for**(**int** idx = 0 ; idx < aNumIterations; ++idx) {

**int**[] copy = Arrays.copyOf(aArray, aArray.length);

}

}

**private** **static** **void** copyUsingForLoop( **int**[] aArray , **int** aNumIterations) {

**for**(**int** iterIdx = 0 ; iterIdx < aNumIterations; ++iterIdx) {

**int** [] copy = **new** **int**[aArray.length];

**for** (**int** idx = 0; idx < aArray.length; ++idx) {

copy[idx] = aArray[idx];

}

}

}

**private** **static** **void** log(String aMessage){

System.out.println(aMessage);

}

/\*\*

\* (The for-loop and System.arraycopy styles clearly have independent

\* storage, and are not exercised in this method.)

\*/

**private** **static** **void** demoIndependanceOfStorage() {

//a clone of a one-dimensional array has independent storage

**int**[] numbers = {1,1,1,1};

**int**[] numbersClone = (**int**[])numbers.clone();

//set 0th element to 0, and compare

numbersClone[0] = 0;

log("Altered clone has NOT affected original:");

log("numbersClone[0]: " + numbersClone[0]);

log("numbers[0]: " + numbers[0]);

//the clone of a multi-dimensional array does \*not\* have

//independant storage

**int**[][] matrix = { {1,1}, {1,1} };

**int**[][] matrixClone = (**int**[][])matrix.clone();

//set 0-0th element to 0, and compare

matrixClone[0][0] = 0;

log("Altered clone has affected original:");

log("matrixClone element 0-0:" + matrixClone[0][0]);

log("matrix element 0-0: " + matrix[0][0]);

//the clone of an array of objects as well is only shallow

Date[] dates = {**new** Date()};

log("Original date: " + dates[0]);

Date[] datesClone = (Date[])dates.clone();

datesClone[0].setTime(0);

log("Altered clone has affected original:");

log("datesClone[0]:" + datesClone[0]);

log("dates[0]: " + dates[0]);

}

}

System.arraycopy seems to have slightly better performance. Differences between the various styles are small, however, and would often be regarded as a micro-optimization. (As usual, such judgements depend on the context).

>java -cp . -Xint ArrayCopier performance 250000

Using clone: 108.168 ms

Using System.arraycopy: 125.334 ms

Using Arrays.copyOf: 190.490 ms

Using for loop: 392.026 ms

The above example use the [-Xint](http://docs.oracle.com/javase/7/docs/technotes/tools/windows/java.html#nonstandard) option to turn off the Just In Time compiler. Here, bytecodes are interpreted at runtime, but never compiled by the HotSpot compiler into native code. This provides a uniform environment for executing tests of relative execution time, since there is no "warm-up" period.

Example run demonstrating independence of storage, or lack thereof:

>java -cp . ArrayCopier storage   
Altered clone has NOT affected original:   
numbersClone[0]: 0   
numbers[0]: 1   
Altered clone has affected original:   
matrixClone element 0-0:0   
matrix element 0-0: 0   
Original date: Mon Sep 30 15:47:58 EDT 2002   
Altered clone has affected original:   
datesClone[0]:Wed Dec 31 19:00:00 EST 1969   
dates[0]: Wed Dec 31 19:00:00 EST 1969