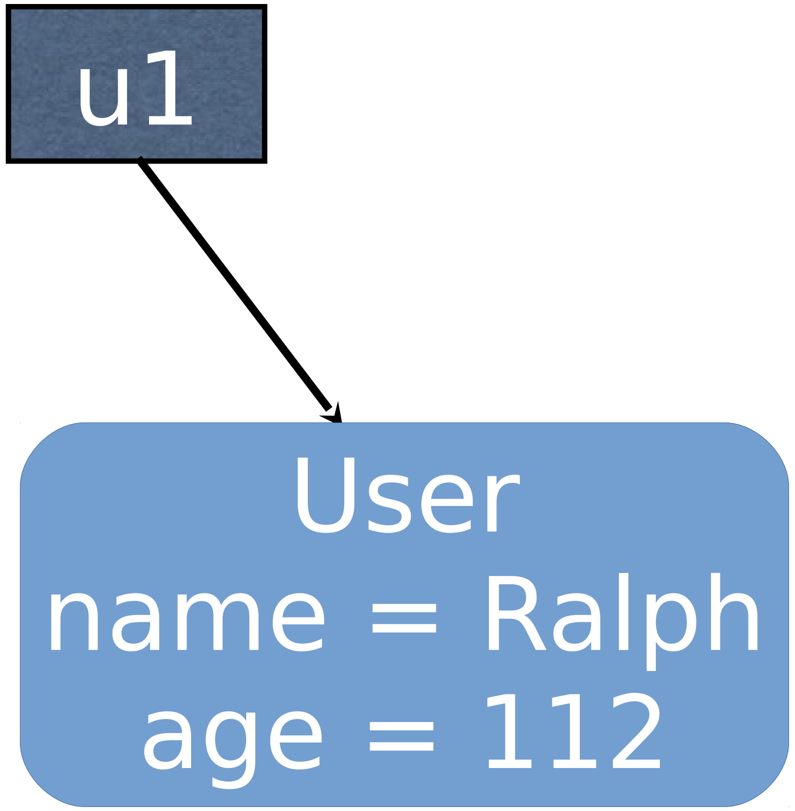
Clone and Cloneable

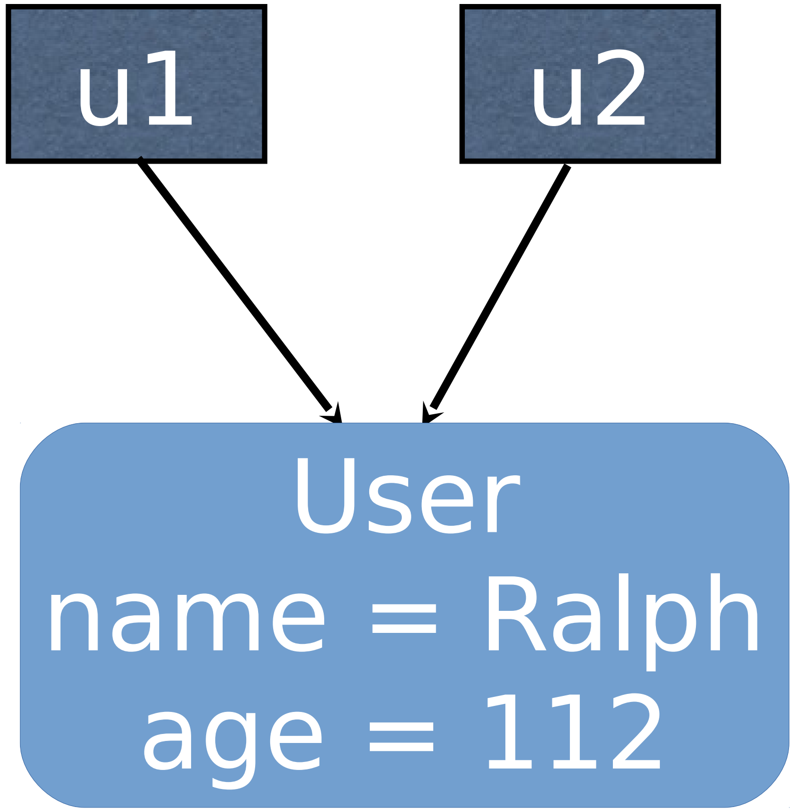
```
Class User {
   public String name;
   public int age;
   public User(String str,  int a) {
      Name = str; age = a;
   }

Class Test {
   public static void main(String[ ] args) {
      User u1 = new User("Ralph", 112);
      System.out.println(u1.name); // Ralph
      User u2 = u1;
      u2.name = "Bond";
      System.out.println(u1.name); // Bond
   }
}
```



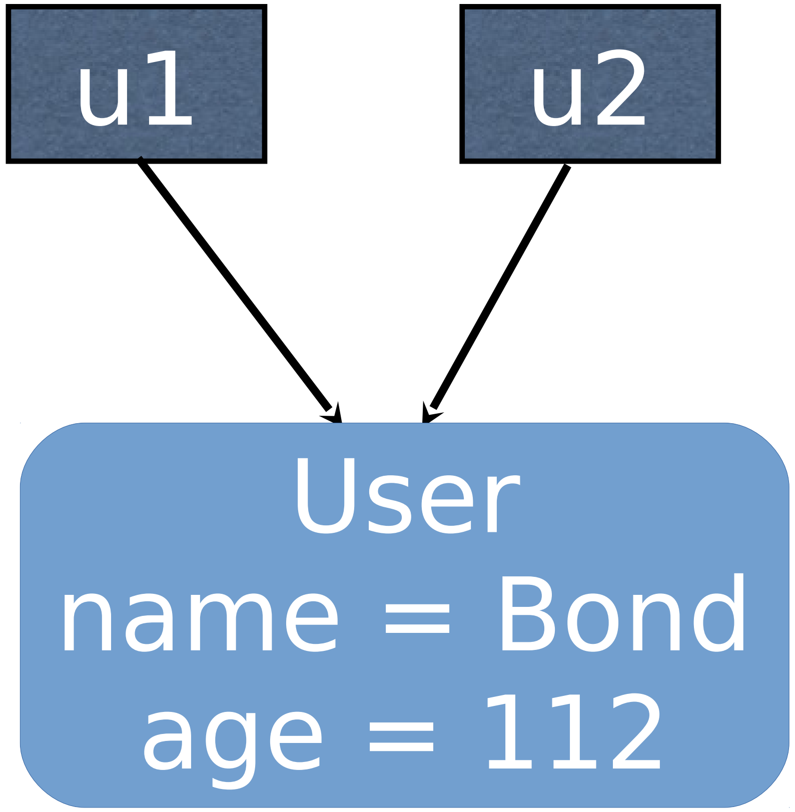
u1

User
name = Ralph
age = 112

```
Class User {
   public String name;
   public int age;
   public User(String str,  int a) {
      Name = str; age = a;
   }

Class Test {
   public static void main(String[ ] args) {
      User u1 = new User("Ralph", 112);
      System.out.println(u1.name); // Ralph
      User u2 = u1;
      u2.name = "Bond";
      System.out.println(u1.name); // Bond
   }
}
```



```
Class User {
   public String name;
   public int age;
   public User(String str,  int a) {
      Name = str; age = a;
   }

Class Test {
   public static void main(String[ ] args) {
      User u1 = new User("Ralph", 112);
      System.out.println(u1.name); // Ralph
      User u2 = u1;
      u2.name = "Bond";
      System.out.println(u1.name); // prints Bond
   }
}
```



But what if we want u2 to reference a *copy* of what u1 references?

Sometimes a copy or *clone* of an object is desired

- Even though Java only copies references, it is sometimes desirable that an assignment put a reference to a new object in the left hand side (LHS) variable
- *Cloning* is the Java mechanism for accomplishing this.
- The class for objects to be cloned *must* implement the interface *Cloneable*
- Invoking the *clone* method on an object produces a clone of the object.

```
class X {
   int n;
   X( ) {n = 3;}
   X(in n) {
      this.n = n;
   }
   getN( ) {
      return n;
   }
}

X xobj = new X(4);
```

- Even though X is a very simple class, it cannot invoke *clone* because it does not implement *cloneable*.
- We can duplicate X and its state, as shown below:

X xobj = new X(4);
Y yobj = new X(xobj.getN( ));

- But . . .
  - What if X was a hashmap or something else complicated?
  - What about private fields?
  - Encapsulation implies objects should copy themselves.

```
class X implements Cloneable {
   int n;
   X( ) {n = 3;}
   X(in n) {
      this.n = n;
   }
   int getN( ) {
      return n;
   }
   public Object clone( ) throws CloneNotSupportedExeception {
      return super.clone( );
   }
}

X xobj = new X(4);
X xobjClone = (X) xobj.clone( );
```

```
class X implements Cloneable {
   int n;
   X( ) {n = 3;}
   X(in n) {
      this.n = n;
   }
   int getN( ) {
      return n;
   }
   public Object clone( ) throws CloneNotSupportedExeception {
      return super.clone( );
   }
}

X xobj = new X(4);
X xobjClone = (X) xobj.clone( );
```

The *Object clone* function (super.clone( )) makes a byte-by-byte copy of the object referenced by *xobj* and returns a reference to it.

```
class X implements Cloneable {
   int n;
   X( ) {n = 3;}
   X(in n) {
      this.n = n;
   }
   int getN( ) {
      return n;
   }
   public Object clone( ) throws CloneNotSupportedExeception {
      return super.clone( );
   }
}

X xobj = new X(4);
X xobjClone = (X) xobj.clone( );
```

The *clone* method required by the Cloneable interface returns a reference to an object of type Object, hence the cast.

The interface cloneable *is not a normal interface*

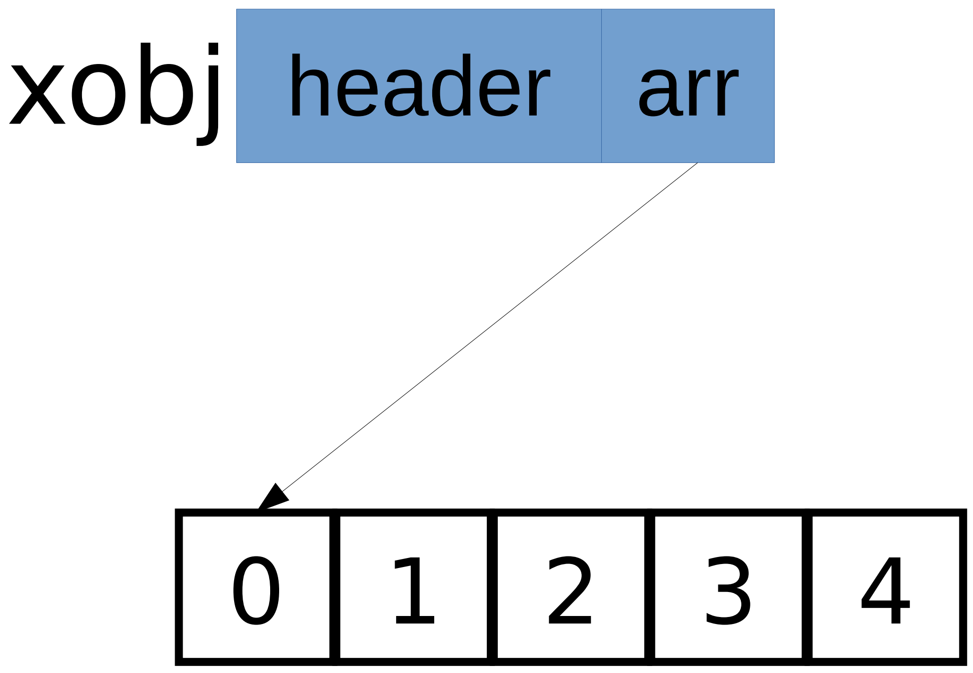
- *Cloneable* is empty -- a class implementing it doesn't have to actually implement anything (but can if it wants)
  - but if a *public Object clone( )* method is not implemented an error will result if an attempt is made to clone the object.
- *Implements Cloneable* is a signal to the *Object* class that it is ok for *Object's clone* to clone this object w/a byte for byte copy.
- This leverages the fact that *Object* is not a normal class. It, and some other systems classes, perform functionality not expressible in Java.

Calling *clone*

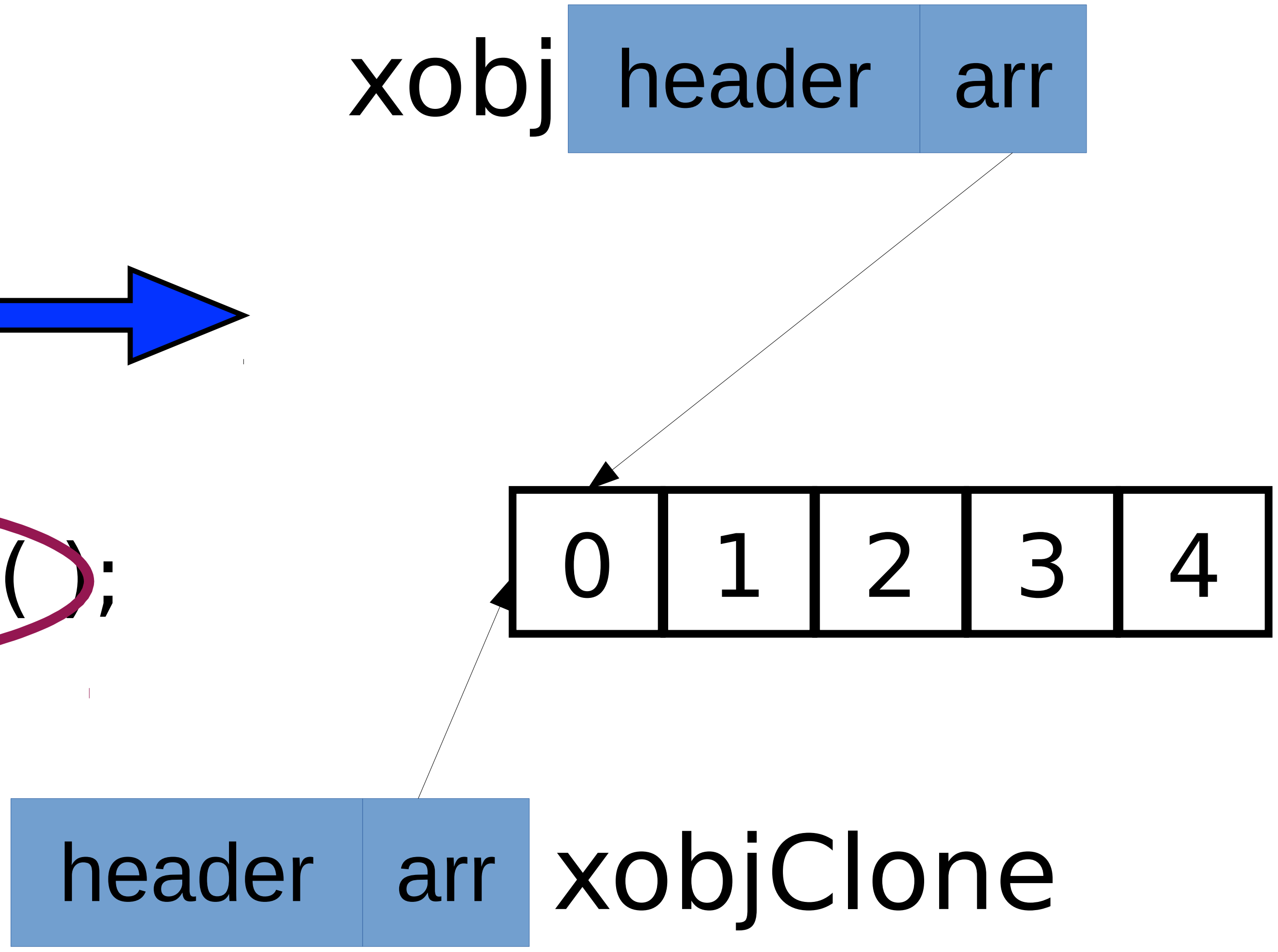
```
import java.util.*;
class X implements Cloneable {
   public int n;
   public X( ) {n=3;}
   public Object clone( ) throws CloneNotSupportedException {
      return super.clone( );
   }
}
. . .
try {
   xobj_clone = (X) xobj.clone( );
} catch (CloneNotSupportedExeception e) { ... }
. . .
```

Consider class X

```
class X implements Cloneable {
   public int[ ] arr = new int[5]
   public X( ) {
      Random ran = new Random( );
      int i = 0;
      while (i < 5) {
         arr[i++] = ran.nextInt(10);
      }
   }
   public Object clone( ) throws CloneNotSupportedExeception {
      return super.clone( );
   }
}
. . .
X xobj = new X( );
```



```
. . .
X xobj = new X( );
. . .
X xobjClone = xobj.clone( );
```



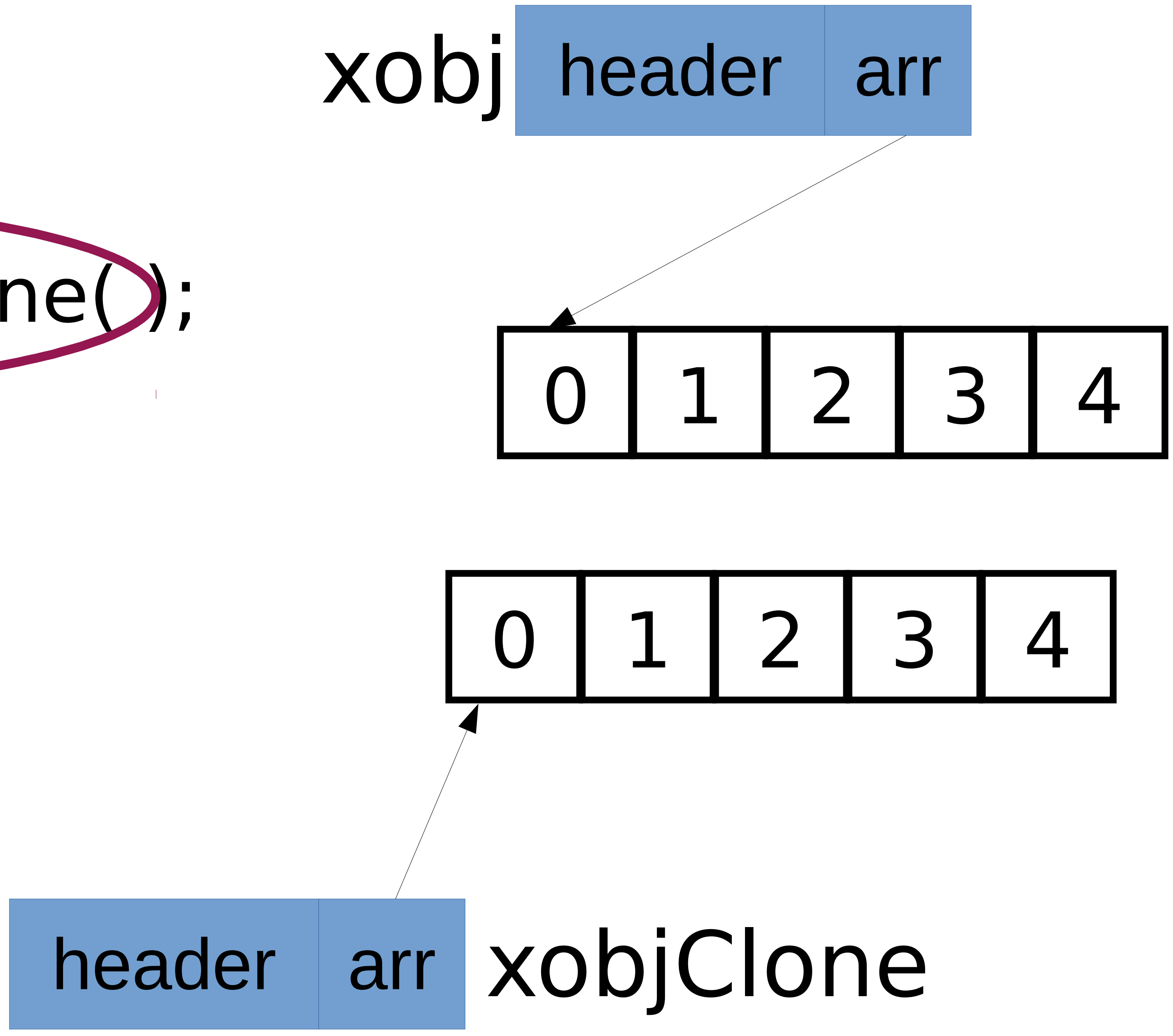
The default *clone* called in *super* (Object) will do this

What if we also wanted *arr* cloned?

What if we want this?

```
. . .
X xobj = new X( );
. . .
X xobjClone = xobj.clone( );
```

We need to write our own clone function that does something useful.



xobj

header	arr

0	1	2	3	4

0	1	2	3	4

header	arr

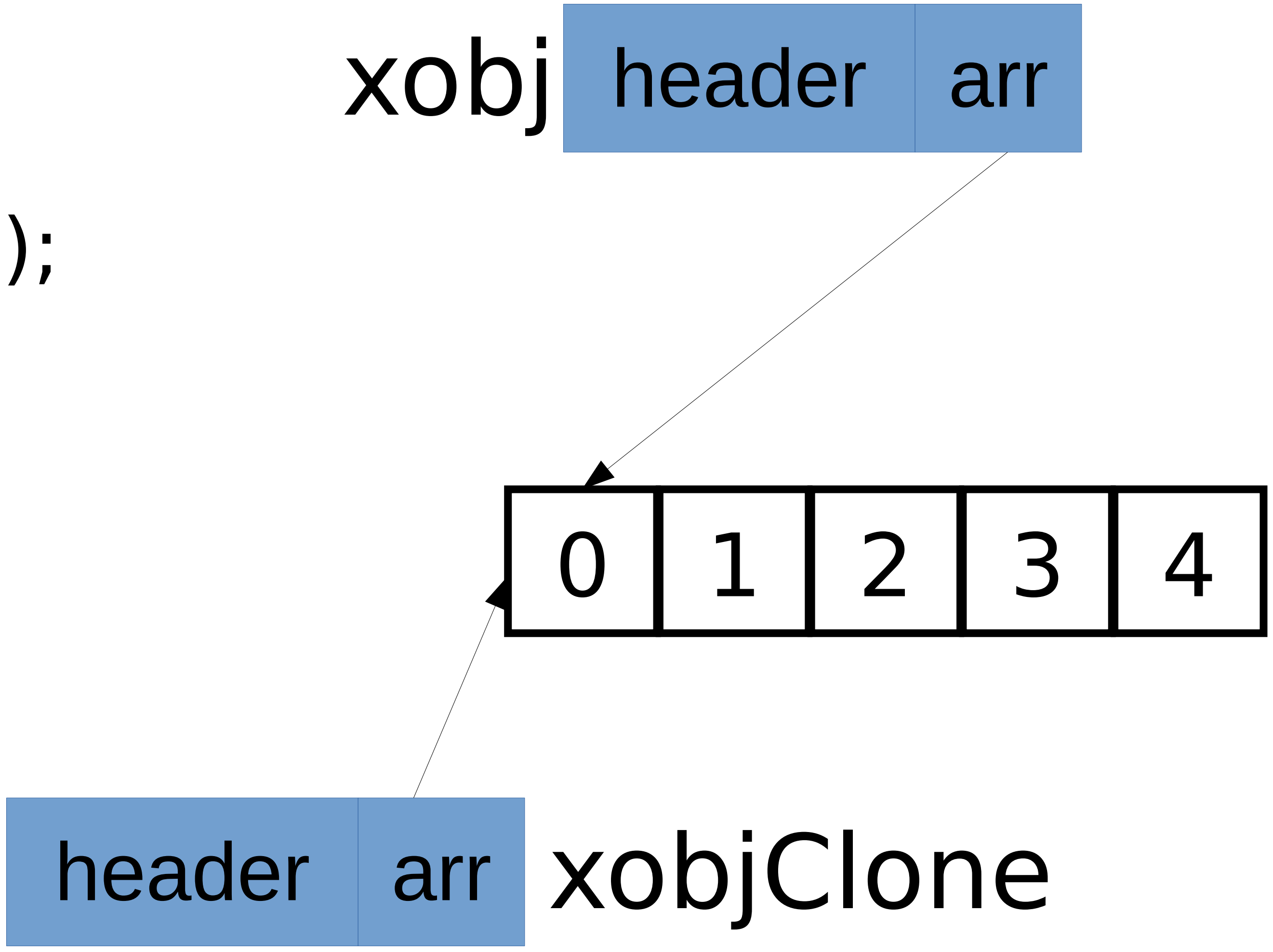
xobjClone

```
// X is as before
public Object clone( ) throws CloneNotSupportedException {
   X xob = null;
   xob = (X) super.clone( );
   // now clone the array
   xob.arr = (int[ ]) arr.clone( );
   return xob;
}
. . .
public static void main(String[ ] args) throws Exception {
   x xobj = new X( );
   x xobjClone = (X) xobj.clone( );
   System.out.println(xobj); // 0 4 5 2 5
   System.out.println(xobjClone); // 0 4 5 2 5
   xobj.arr[0] = 1000;
   System.out.println(xobj); // 1000 4 5 2 5
   System.out.println(xobjClone); // 0 4 5 2 5
```

declare a reference to the **new** cloned object.

clone the object using the Object clone

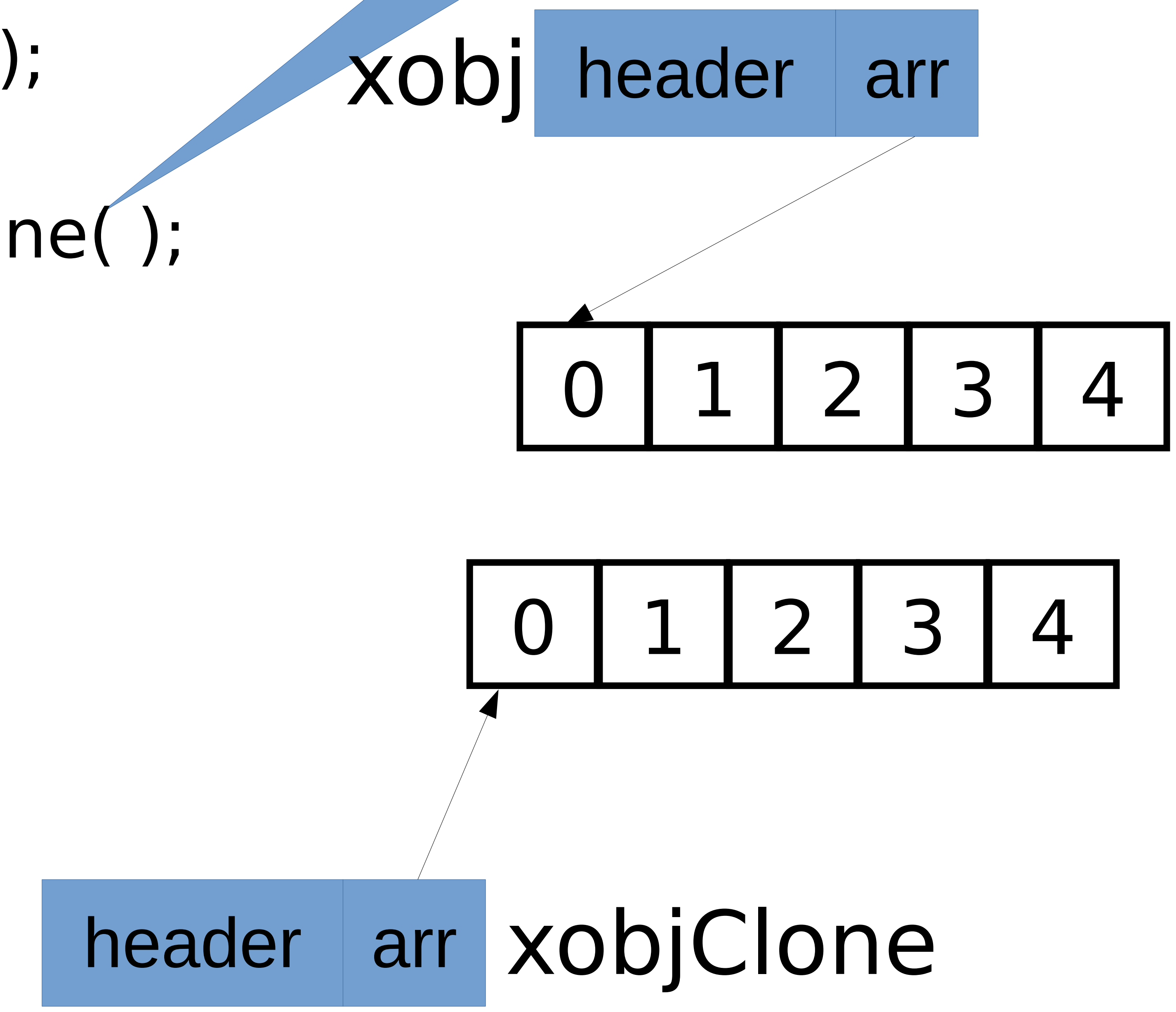
```
X is as before
public Object clone( ) throws CloneNotSupportedException {
   X xob = null;
   xob = (X) super.clone( );
   // now clone the array
   xob.arr = (int[ ]) arr.clone( );
   return xob;
}
. . .
```



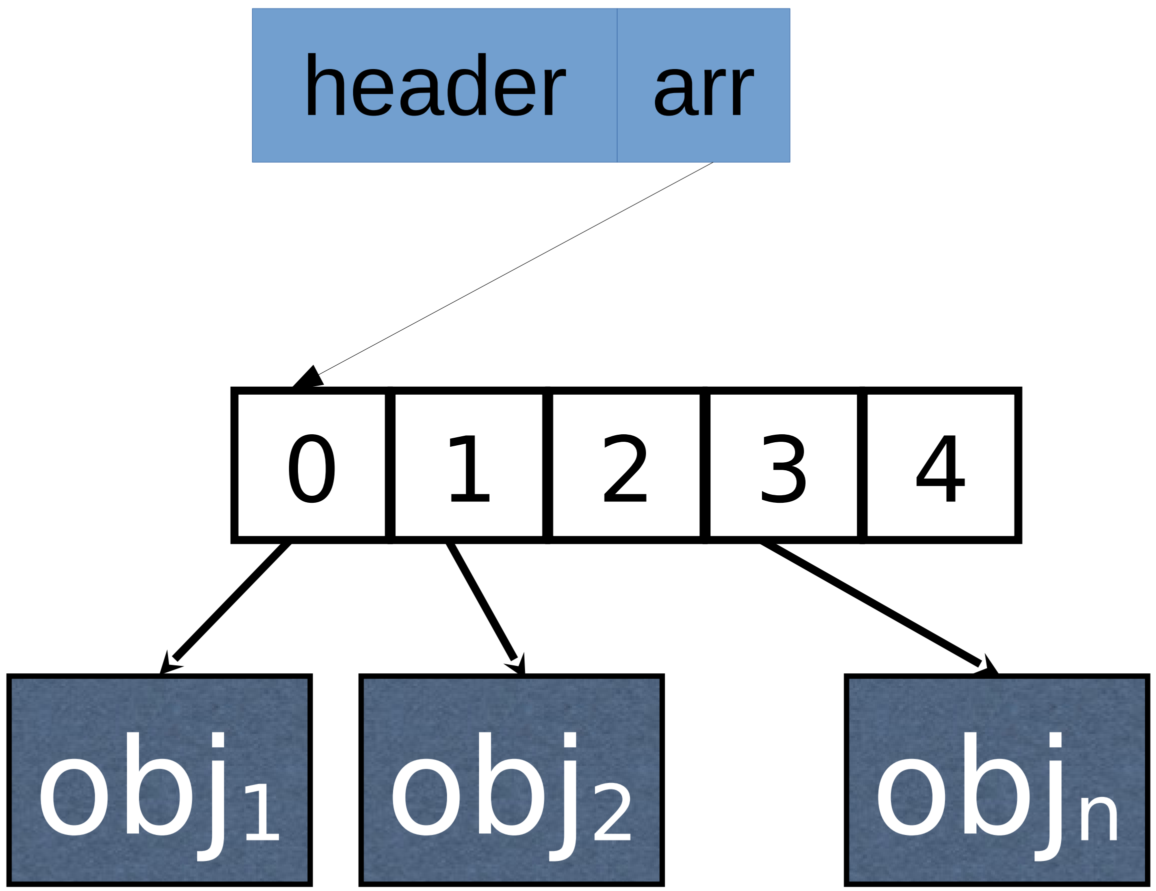
Clone
the array.

X is as before

```
public Object clone( ) throws CloneNotSupportedException {
   X xob = null;
   xob = (X) super.clone( );
   // now clone the array
   xob.arr = (int[ ]) arr.clone( );
   return xob;
}
```



What about arrays of references?



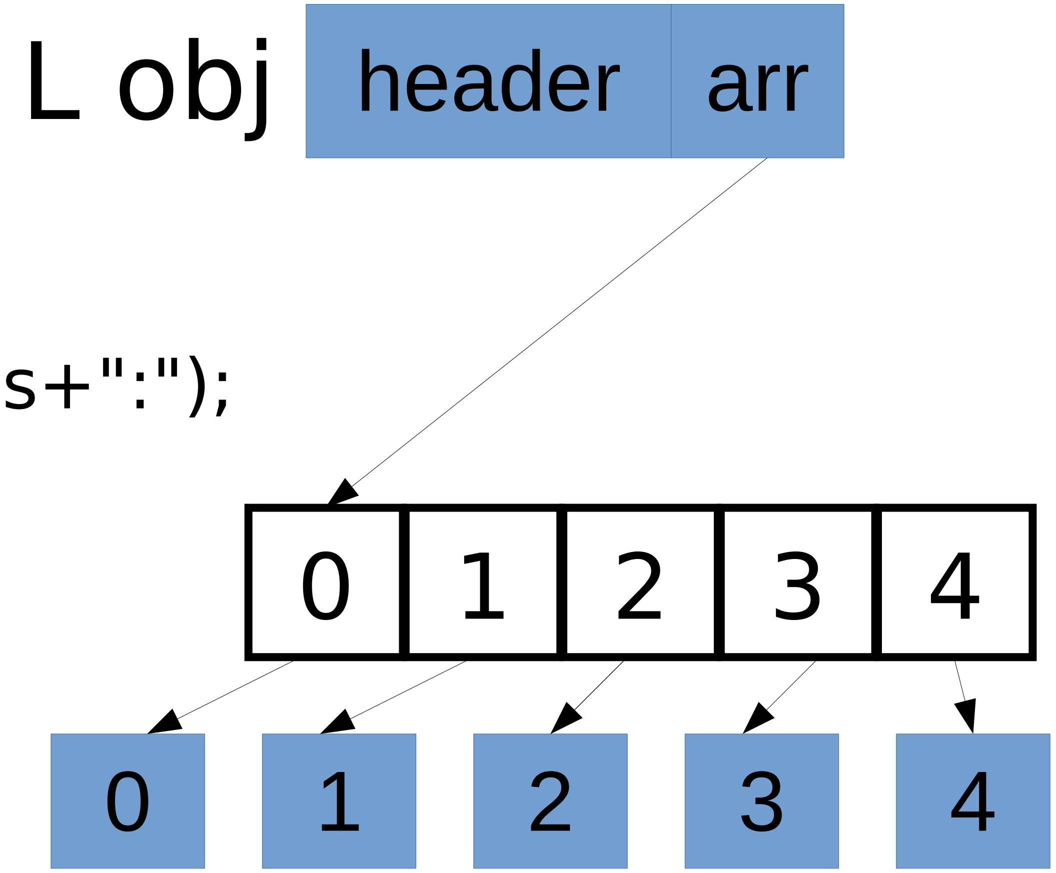
An example of cloning with arrays of objects

```
public class I implements Cloneable {

   int i;
   public I( ) {i = 0;}
   public I(int i) {this.i = i;}
   public void print( ) {System.out.println("i: "+i);}
   public Object clone( ) throws CloneNotSupportedException {
      return super.clone( );
   }
}
```

I is a class that holds an integer (could use Integer)

```
public class L implements Cloneable {

   I arryI[ ];

   public L( ) {
      arryI = new I[5];
      for (int i = 0; i < 5; i++) {
         arryI[i] = new I(i);
      }
   }

   void print(String s) {
      System.out.println("Printing L object "+s+":");
      for (int i = 0; i < arryI.length; i++)
         arryI[i].print( );
   }

   public void setElement(int i, int v) {
      arryI[i] = new I(v);
   }

   public Object clone( ) throws CloneNotSupportedException { // see next
slide }
```

L is a class that has a reference to an array of I objects.



L obj

header

arr

0 1 2 3 4

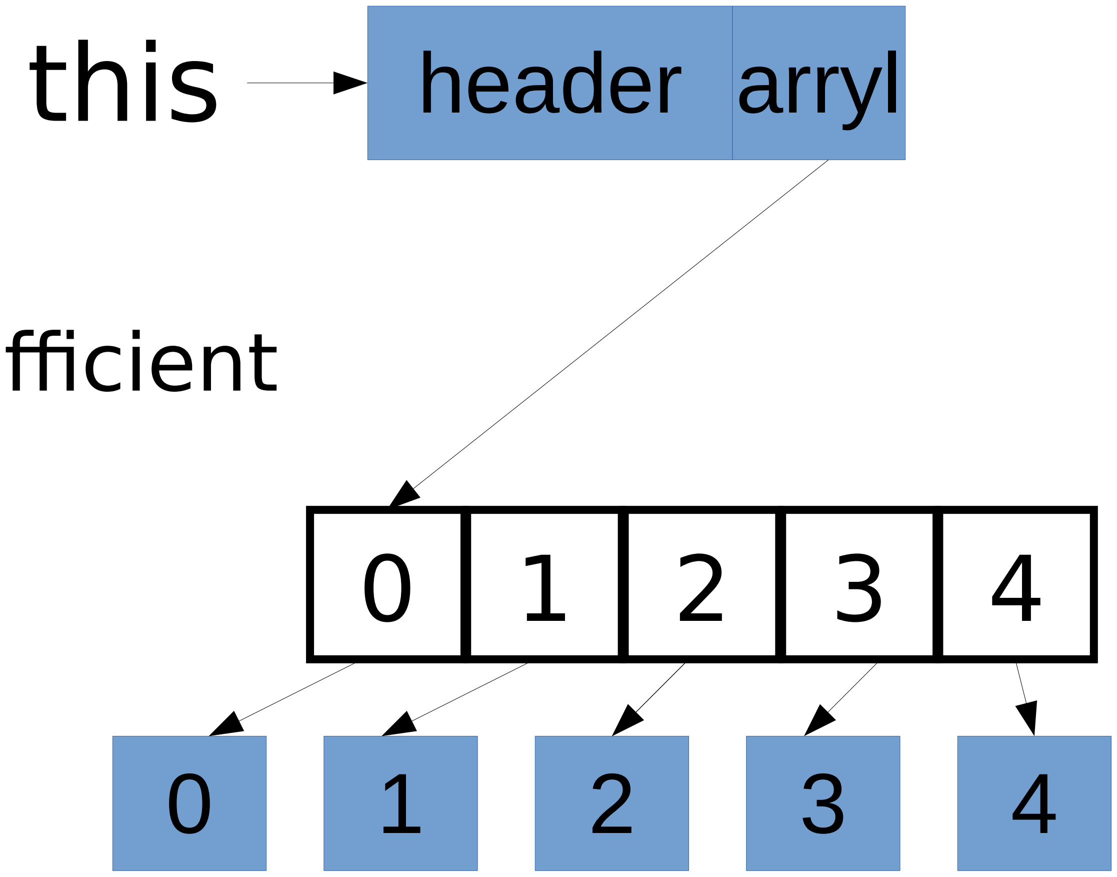
0 1 2 3 4

The clone method

```
public Object clone( ) throws CloneNotSupportedException {
  L lClone = (L) super.clone( );
  // lClone.arryI = arryI.clone( ); WRONG by itself and
  // inefficient in the best case (recopies contents of arryI which
  // are then written over
  I arryClone[ ] = new I[arryI.length];
  for (int i = 0; i < arryI.length; i++)
    arryClone[i] = (I) arryI[i].clone( );
  lClone.arryI = arryClone;
  return lClone;
}
```

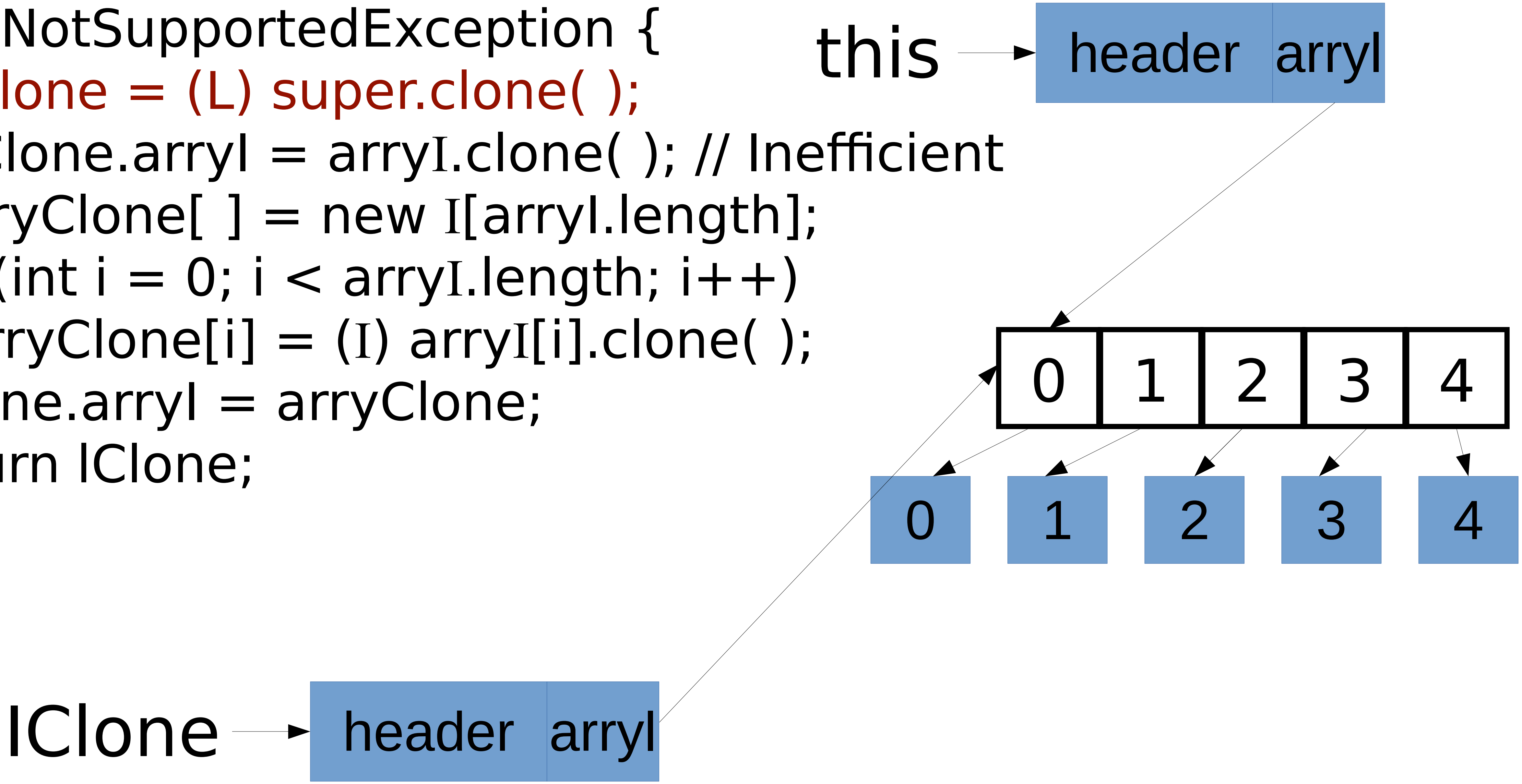
Let's see what this does pictorially

```
public Object clone( ) throws
CloneNotSupportedException {
   L lClone = (L) super.clone( );
   // lClone.arryI = arryI.clone( ); // Inefficient
   I arryClone[ ] = new I[arryI.length];
   for (int i = 0; i < arryI.length; i++)
      arryClone[i] = (I) arryI[i].clone( );
   lClone.arryI = arryClone;
   return lClone;
}
```



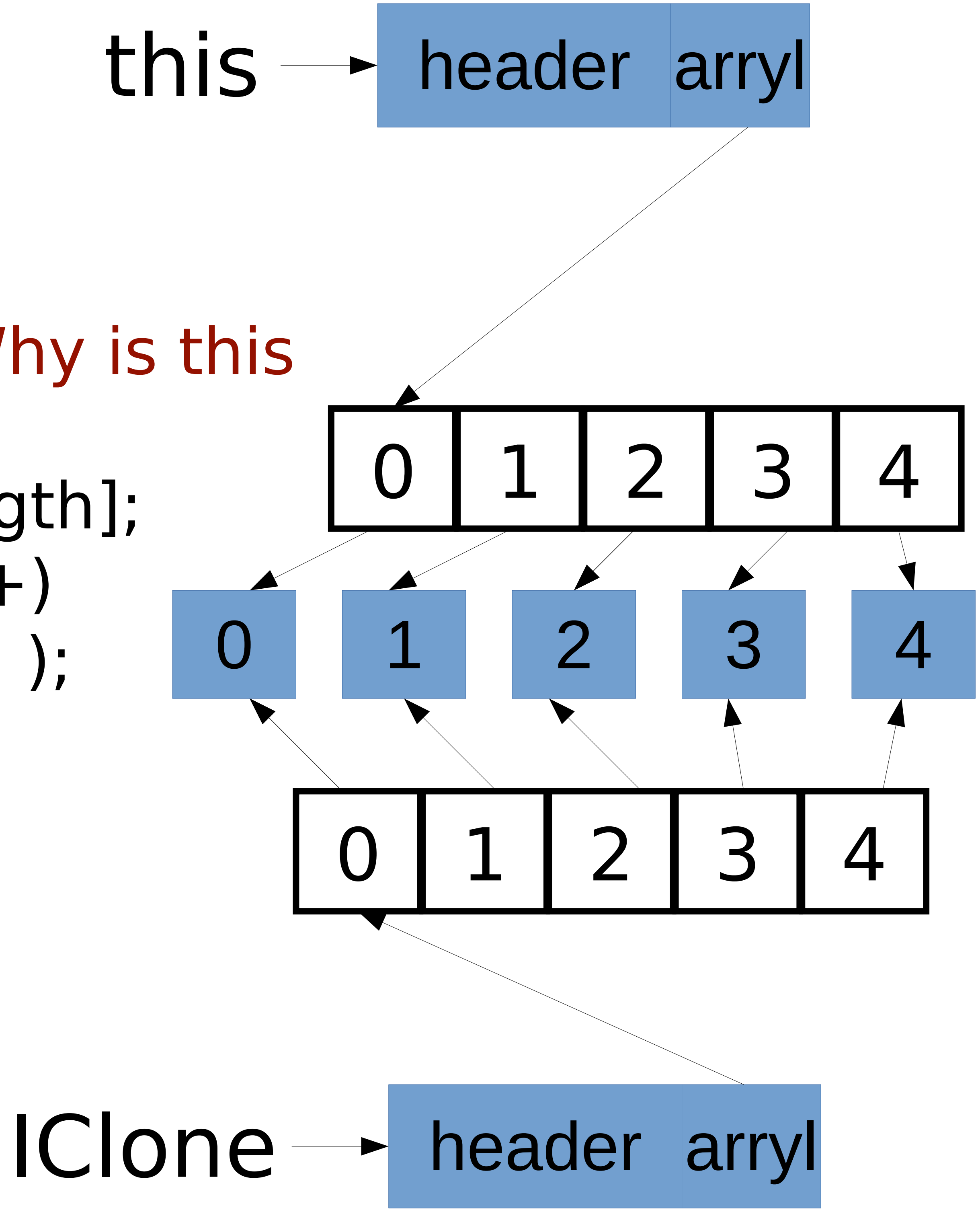
Let's see what this does pictorially

```
public Object clone( ) throws
CloneNotSupportedException {
   L lClone = (L) super.clone( );
   // lClone.arryI = arryI.clone( ); // Inefficient
   I arryClone[ ] = new I[arryI.length];
   for (int i = 0; i < arryI.length; i++)
      arryClone[i] = (I) arryI[i].clone( );
   lClone.arryI = arryClone;
   return lClone;
}
```



Let's see what this does pictorially

```
public Object clone( ) throws
CloneNotSupportedException {
   L lClone = (L) super.clone( );
   lClone.arryI = arryI.clone( );  // Why is this
suboptimal?
   // I arryClone[ ] = new I[arryI.length];
   for (int i = 0; i < arryI.length; i++)
      arryClone[i] = (I) arryI[i].clone( );
   lClone.arryI = arryClone;
   return lClone;
}
```

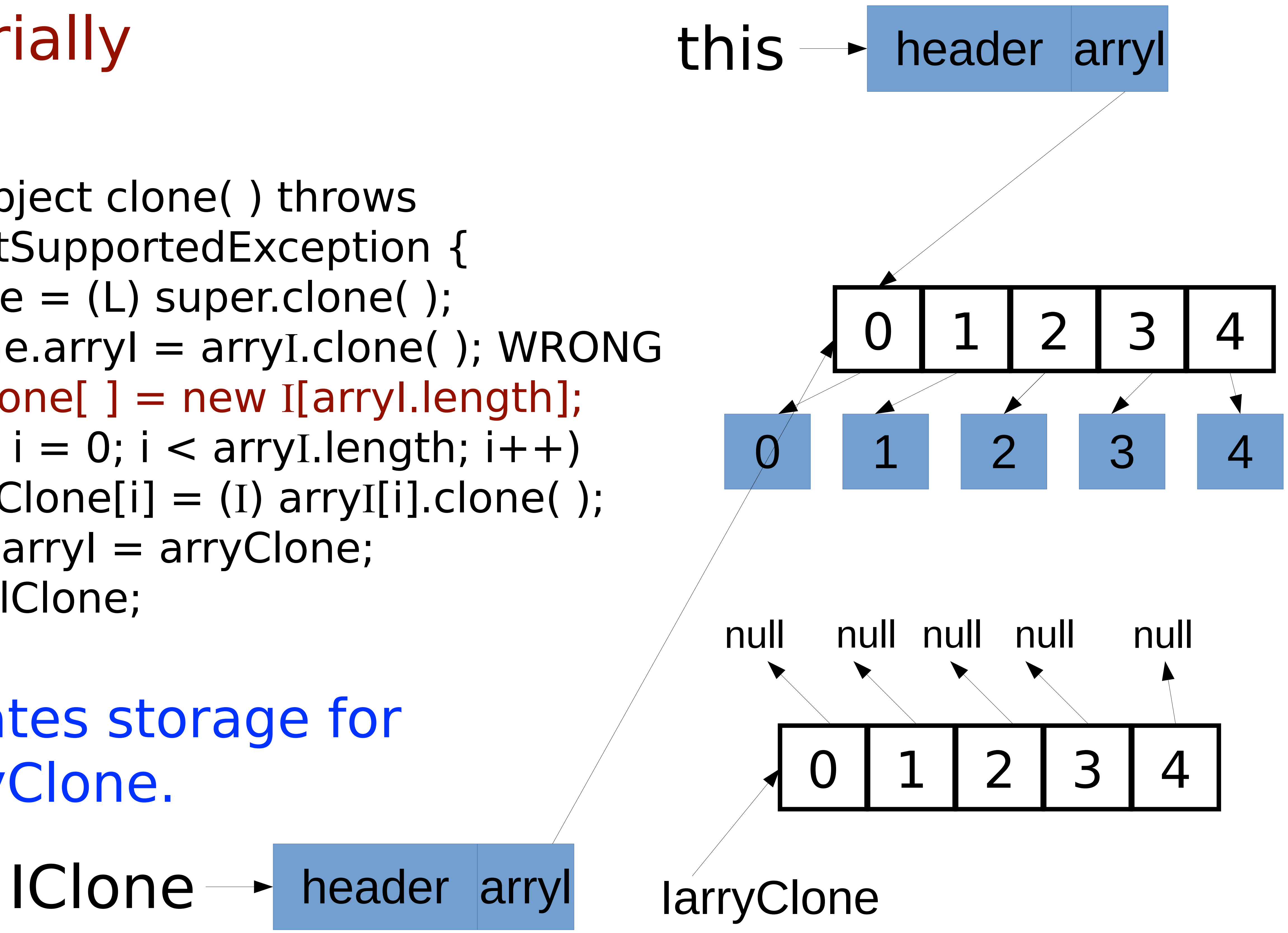


Creates storage for arryI, copies data from *this.arryI*, but doesn't clone the actual objects. There is a wasted copy.

Let's see what this does pictorially

```
public Object clone( ) throws
CloneNotSupportedException {
   L lClone = (L) super.clone( );
   // lClone.arryl = arryI.clone( ); WRONG
   IarryClone[ ] = new I[arryI.length];
   for (int i = 0; i < arryI.length; i++)
      larryClone[i] = (I) arryI[i].clone( );
   lClone.arryl = arryClone;
   return lClone;
}
```

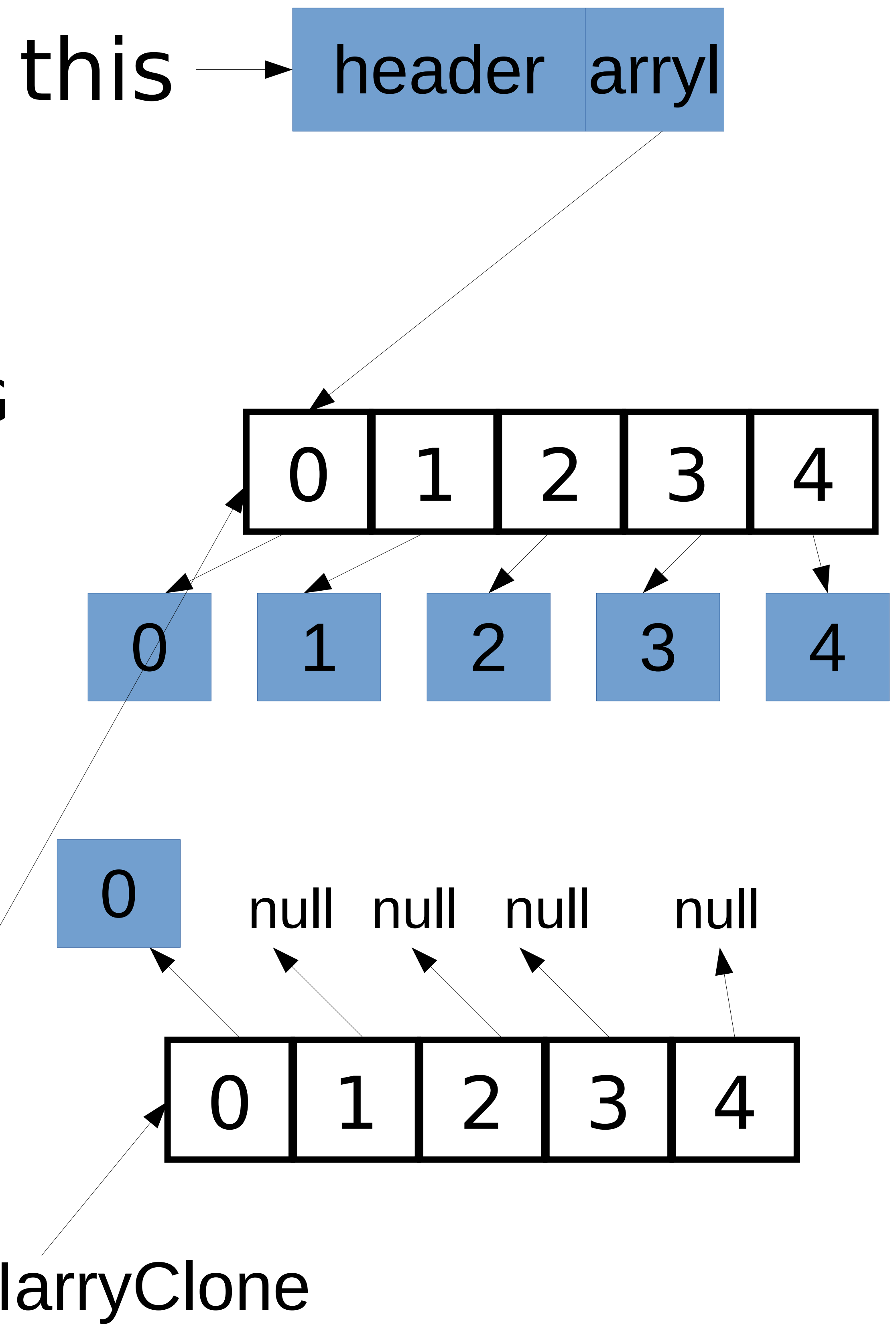
Creates storage for larryClone.

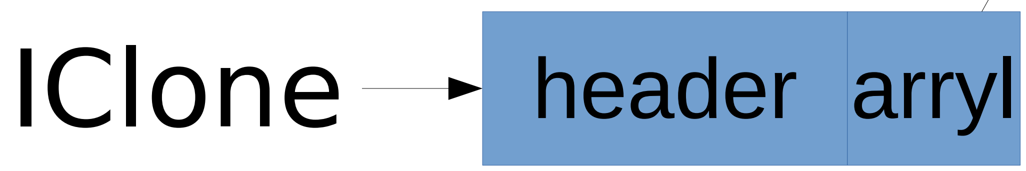


Let's see what this does pictorially

```
public Object clone( ) throws
CloneNotSupportedException {
   L lClone = (L) super.clone( );
   // lClone.arryl = arryI.clone( ); WRONG
   IarryClone[ ] = new I[arryl.length];
   for (int i = 0; i < arryI.length; i++)
      larryClone[i] = (I) arryI[i].clone( );
   lClone.arryl = arryClone;
   return lClone;
}
```

clone the objects pointed to by the elements of larryClone

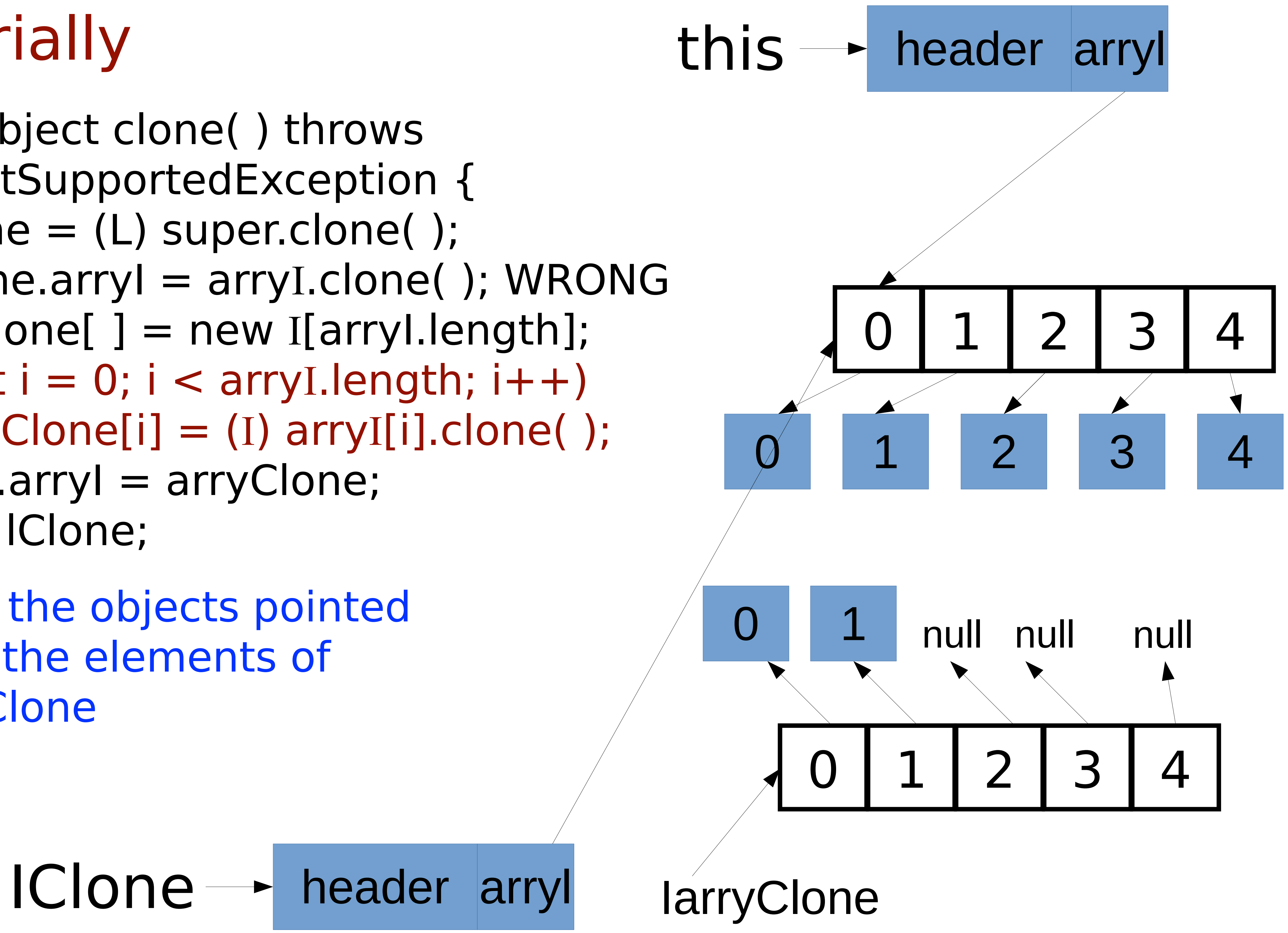




Let's see what this does pictorially

```
public Object clone( ) throws
CloneNotSupportedException {
   L lClone = (L) super.clone( );
   // lClone.arryl = arryI.clone( ); WRONG
   IarryClone[ ] = new I[arryI.length];
   for (int i = 0; i < arryI.length; i++)
      larryClone[i] = (I) arryI[i].clone( );
   lClone.arryl = arryClone;
   return lClone;
}
```

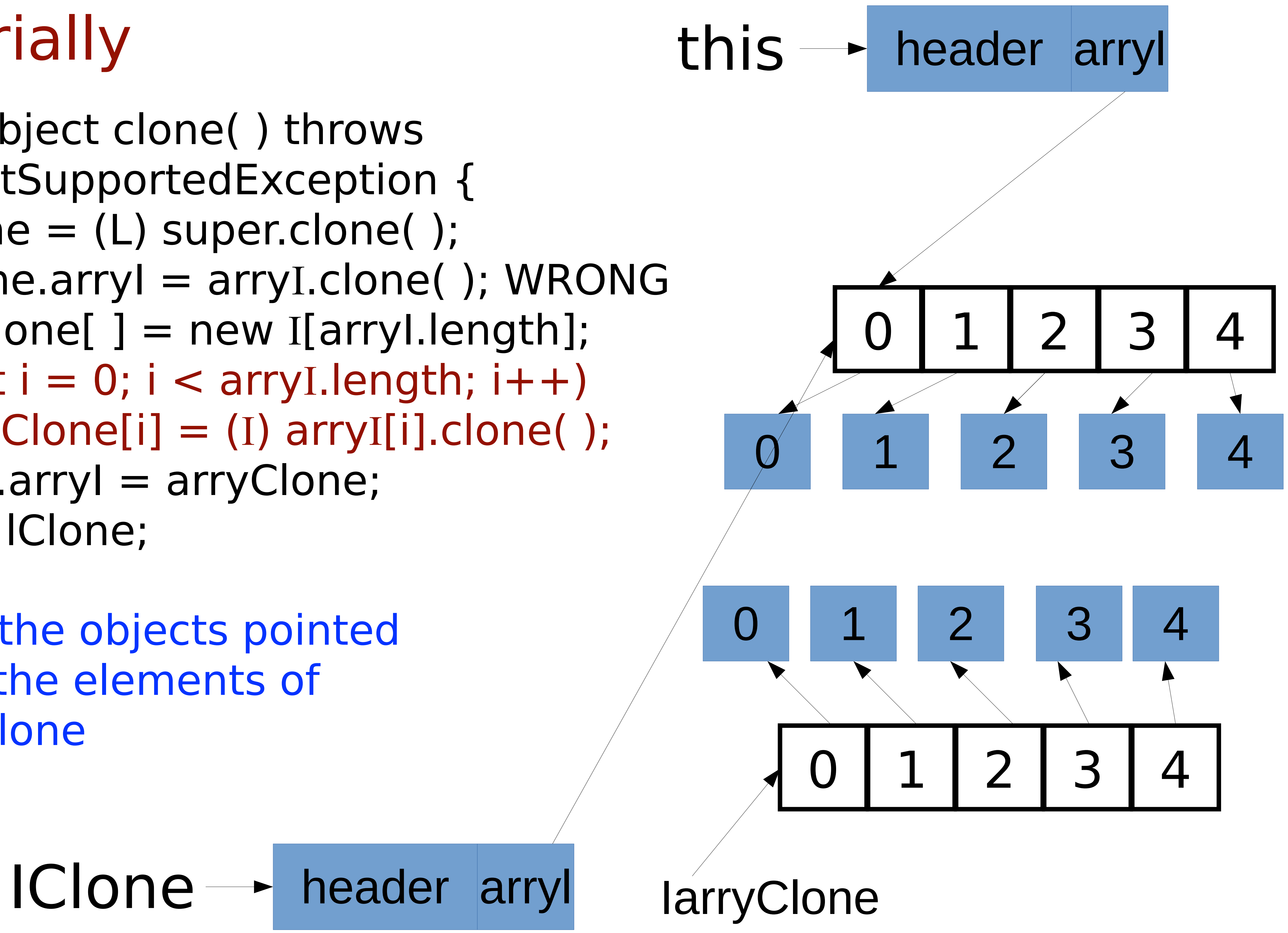
clone the objects pointed to by the elements of larryClone



Let's see what this does pictorially

```
public Object clone( ) throws
CloneNotSupportedException {
   L lClone = (L) super.clone( );
   // lClone.arryl = arryI.clone( ); WRONG
   IarryClone[ ] = new I[arryl.length];
   for (int i = 0; i < arryI.length; i++)
      larryClone[i] = (I) arryI[i].clone( );
   lClone.arryl = arryClone;
   return lClone;
}
```

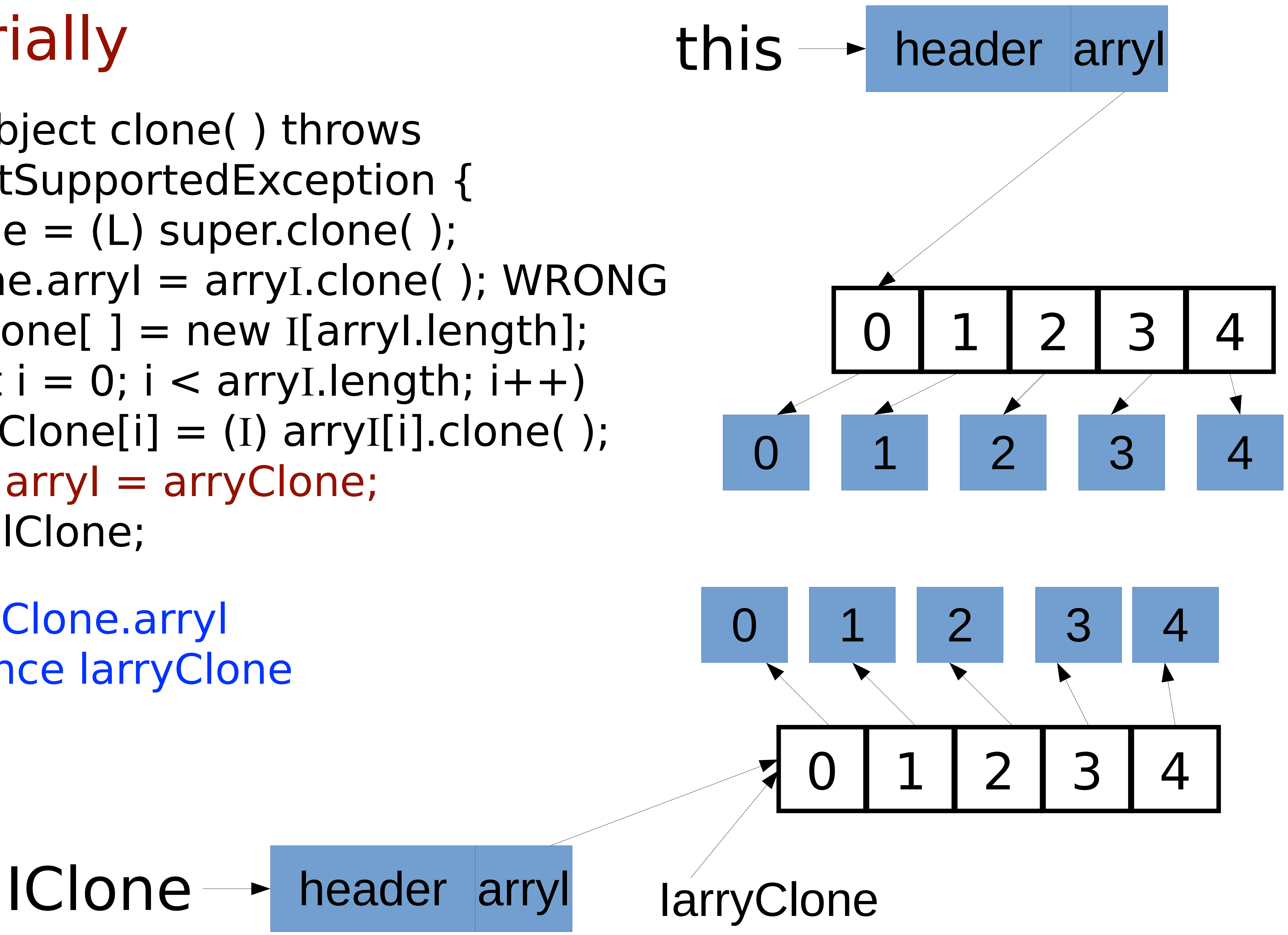
clone the objects pointed to by the elements of larryClone



Let's see what this does pictorially

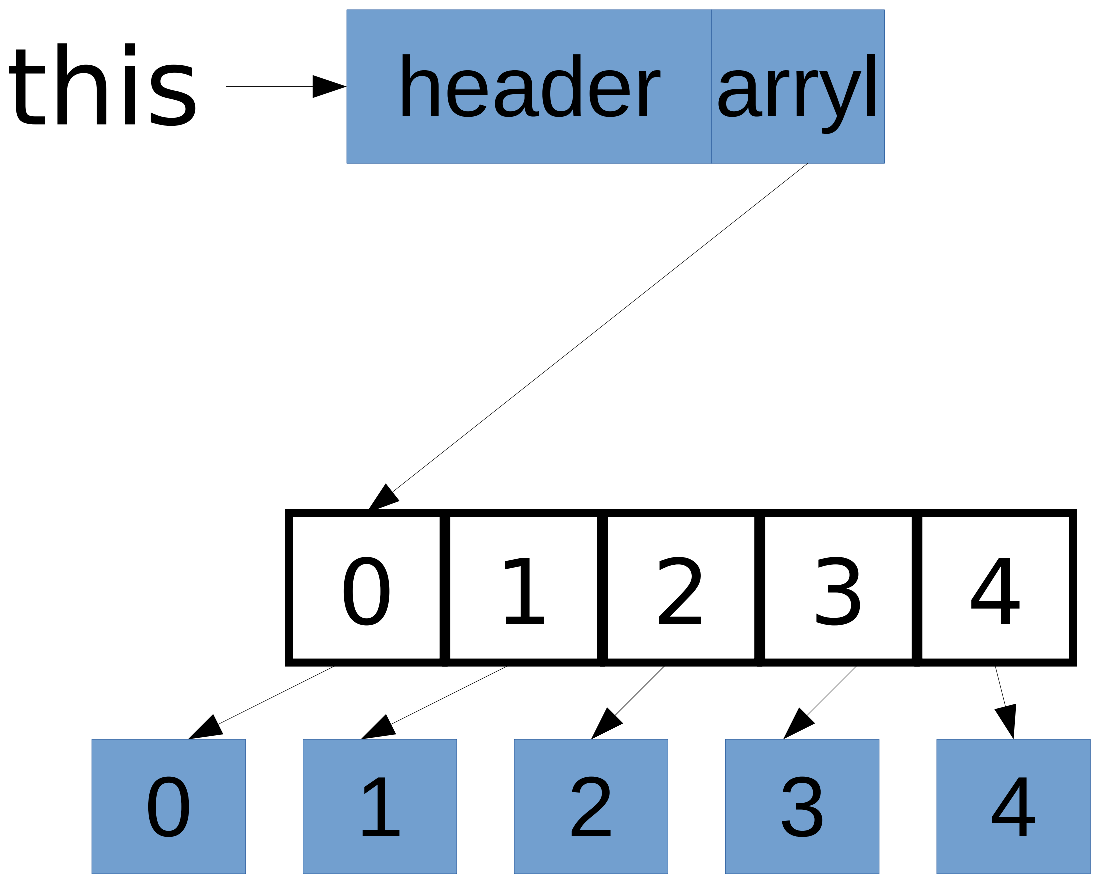
```
public Object clone( ) throws
CloneNotSupportedException {
   L lClone = (L) super.clone( );
   // lClone.arryI = arryI.clone( ); WRONG
   IarryClone[ ] = new I[arryI.length];
   for (int i = 0; i < arryI.length; i++)
      IarryClone[i] = (I) arryI[i].clone( );
   lClone.arryI = arryClone;
   return lClone;
}
```

Make lClone.arryI reference IarryClone

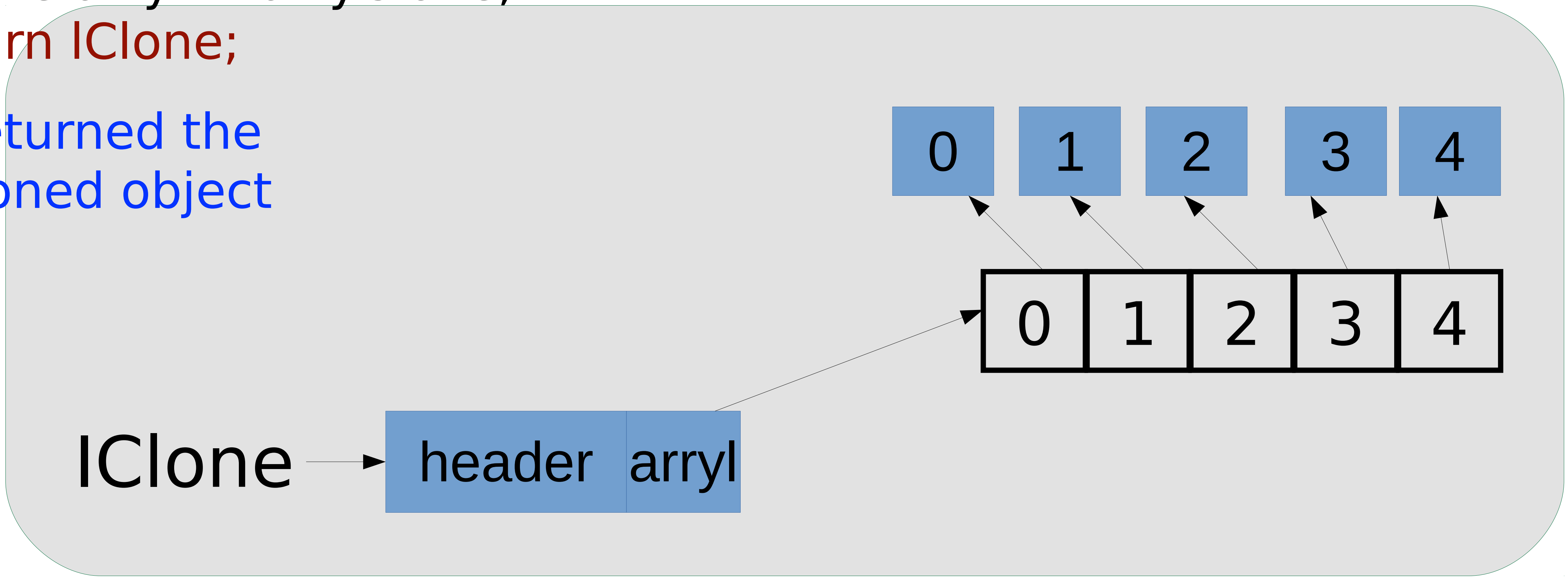


Let's see what this does graphically

```
public Object clone( ) throws
CloneNotSupportedException {
   L lClone = (L) super.clone( );
   // lClone.arryI = arryI.clone( ); WRONG
   IarryClone[ ] = new I[arryI.length];
   for (int i = 0; i < arryI.length; i++)
      IarryClone[i] = (I) arryI[i].clone( );
   lClone.arryI = arryClone;
   return lClone;
}
```



Returned the cloned object



In summary . . .

```
public Object clone( ) throws CloneNotSupportedException
{
   L lClone = (L) super.clone( );
   // lClone.arryI = arryI.clone( ); WRONG!
   I arryClone[ ] = new I[arryI.length];
   for (int i = 0; i < arryI.length; i++)
      arryClone[i] = (I) arryI[i].clone( );
   lClone.arryI = arryClone;
   return lClone;
}
```

This clones the fields of the object, but not what those fields point to.

In summary . . .

```
public Object clone( ) throws
CloneNotSupportedException {
   L lClone = (L) super.clone( );
   // lClone.arryI = arryI.clone( ); WRONG!
   I arryClone[ ] = new I[arryI.length];
   for (int i = 0; i < arryI.length; i++)
      arryClone[i] = (I) arryI[i].clone( );
   lClone.arryI = arryClone;
   return lClone;
}
```

Create a new array of I objects for the cloned L object

In summary . . .

```
public Object clone( ) throws
CloneNotSupportedException {
   L lClone = (L) super.clone( );
   // lClone.arryI = arryI.clone( ); WRONG!
   I arryClone[ ] = new I[arryI.length];
   for (int i = 0; i < arryI.length; i++)
      arryClone[i] = (I) arryI[i].clone( );
   lClone.arryI = arryClone;
   return lClone;
}
```

Clone each object in the original L object *arry* and assign a reference to it to the new L object's array.

Driver code for the example

```
public class Test {

   public static void main(String[] args) throws Exception {
      L lobj = new L( );
      lobj.print("lobj" );
      L lobjCloned = (L) lobj.clone( );
      lobjCloned.print("lobj Cloned" );
      lobjCloned.setElement(2,500);
      lobj.print("lobj" );
      lobjCloned.print("lobjCloned" );
   }
}
```

```
Printing L object lobj:
i: 0
i: 1
i: 2
i: 3
i: 4
Printing L object lobj Cloned:
i: 0
i: 1
i: 2
i: 3
i: 4
```

```
public class Test {
  public static void main(String[] args) throws Exception {
    L lobj = new L( );
    lobj.print("lobj" );
    L lobjCloned = (L) lobj.clone( );
    lobjCloned.print("lobj Cloned" );
    lobjCloned.setElement(2,500);
    lobj.print("lobj" );
    lobjCloned.print("lobjCloned" );
  }
}
```

Change L.java's clone to ...

```
public Object clone( ) throws CloneNotSupportedException {
   return super.clone( );
}
```

```
Printing L object lobj:
i: 0
i: 1
i: 2
i: 3
i: 4
Printing L object lobj Cloned:
i: 0
i: 1
i: 2
i: 3
i: 4
```

lobjCloned.arryI[2]=500

```
Printing L object lobj:
i: 0
i: 1
i: 500
i: 3
i: 4
Printing L object lobjCloned:
i: 0
i: 1
i: 500
i: 3
i: 4
```

```
public class I implements Cloneable {

   int i;
   public I( ) {i = 0;}
   public I(int i) {this.i = i;}
   public void print( ) {System.out.println("i: "+i);}
//   public Object clone( ) throws CloneNotSupportedException {
//      return super.clone( );
//   }
}
```

```
javac Test.java
./L.java:16: clone() has protected access in java.lang.Object
        arryClone[i] = (I) arryI[i].clone( );
                         ^
1 error
```

This comes from not implementing a public clone method in the class being cloned.