

CS 228: Introduction to Data Structures

Lecture 7

Overriding the `clone()` Method

An alternative to writing our own (ad hoc) cloning method is to override Java's `Object.clone()` method. The default implementation of `clone()` creates a field-by-field copy — that is, a ***shallow copy*** — of its argument. Since shallow copying is not always appropriate, Java intentionally disables `clone()`, by declaring it as `protected`, not `public` — so you have to call it from the subclass using `super` — and by having it throw a `CloneNotSupportedException` when called.

To override `clone()`, you either have to explicitly declare that your class implements `Cloneable` or some superclass of your class must implement `Cloneable`. Thus, the declaration for `Point` would be

```
public class Point implements Cloneable{...}
```

Your public `clone()` method can then call the protected `clone` method to create a shallow copy, if that suffices. For the `Point` class, a shallow copy is enough.

```
@Override
public Object clone()
{
    Point copy = null;
    try
    {
        // super.clone() creates copies of
        // all fields
        copy = (Point) super.clone();
    }
    catch (CloneNotSupportedException e)
    {
        // Should never happen unless there's
        // a programming error
    }
    return copy;
}
```

Usage:

```
Point p = new Point(1, 2);
Point q = (Point) p.clone();
```

Shallow versus Deep Copying

A shallow copy suffices for `Point`, because both of its fields, `x` and `y`, are primitive. In general, though, an object may contain references to other objects. In this case, to get a completely independent copy, you have to recursively copy/clone the objects the object references — this is called a ***deep copy***.

Example: The `IntVector` Class

An `IntVector` has a *dimension* `dim` and an array `coords` of coordinates. Its class definition begins like this.

```
public class IntVector implements Cloneable
{
    private int dim;

    private int[] coords;
```

The constructor is

```
public IntVector(int dimension)
{
    if (dimension <= 0)
        throw
            new IllegalArgumentException();
    dim = dimension;
    coords = new int[dim];
}
```

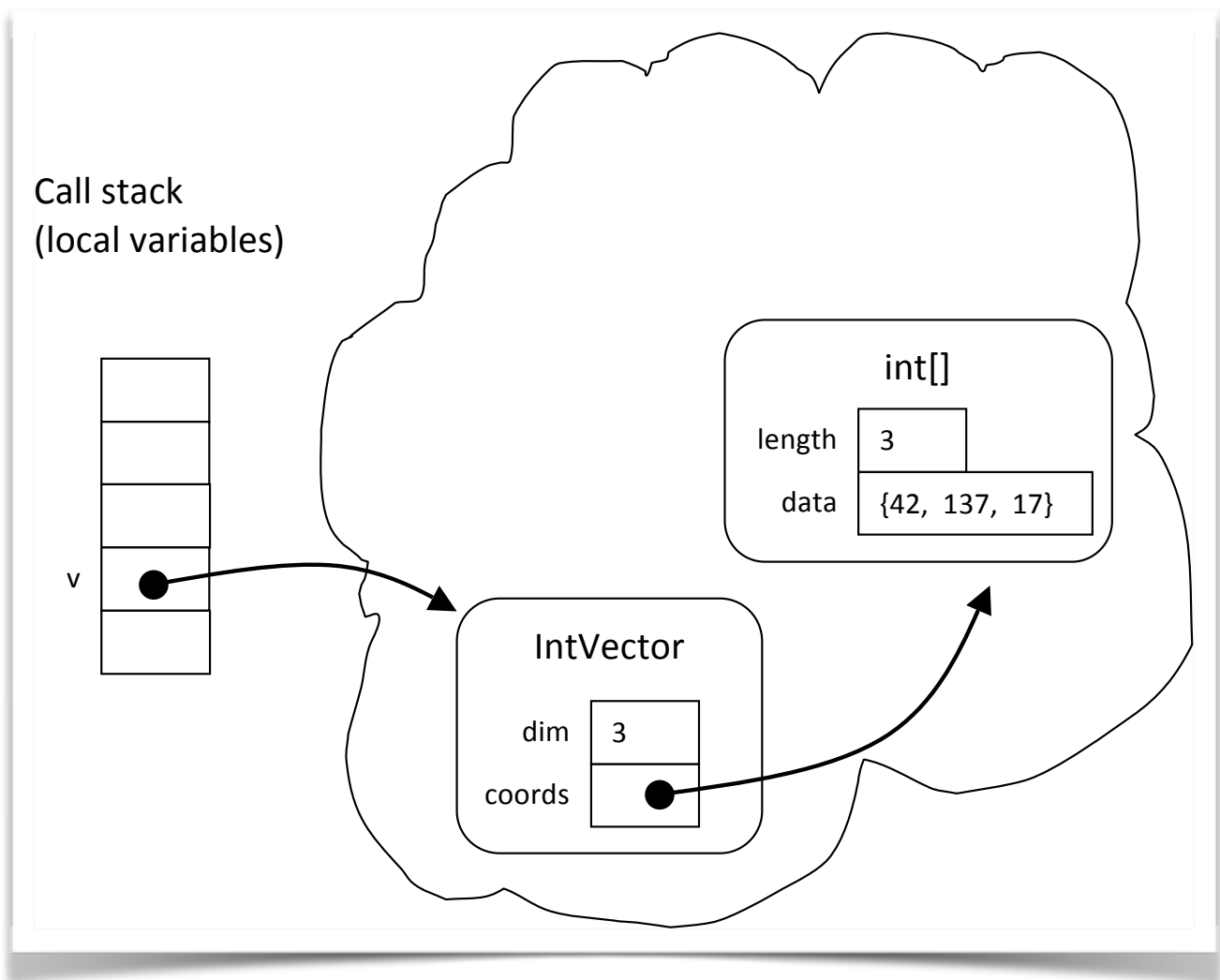
Here is the setter.

```
public void set(int index, int value)
{
    coords[index] = value;
}
```

Now, suppose we execute the statements below:

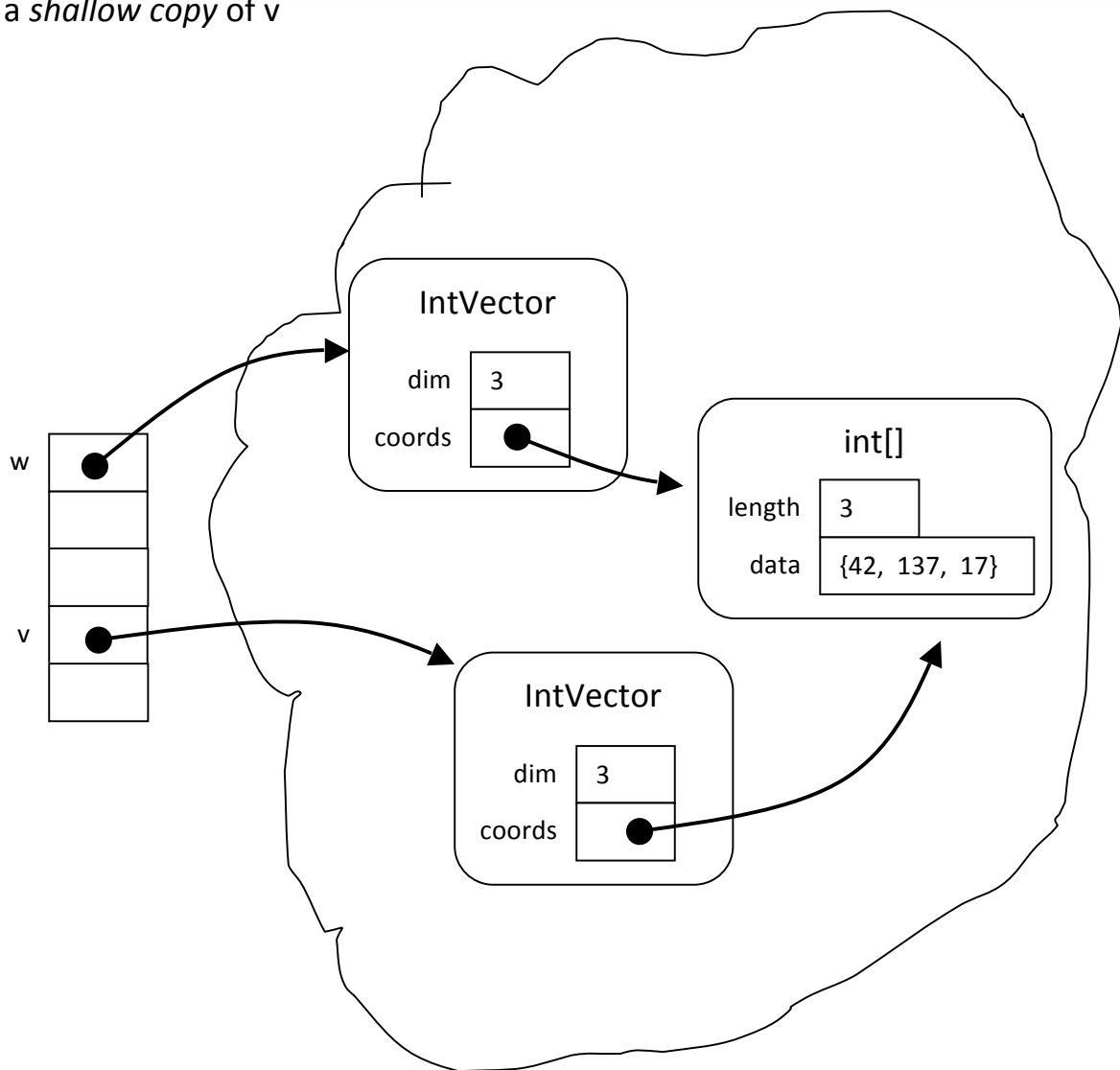
```
IntVector v = new IntVector(3);
v.set(0, 42);
v.set(1, 137);
v.set(2, 17);
```

The result is:



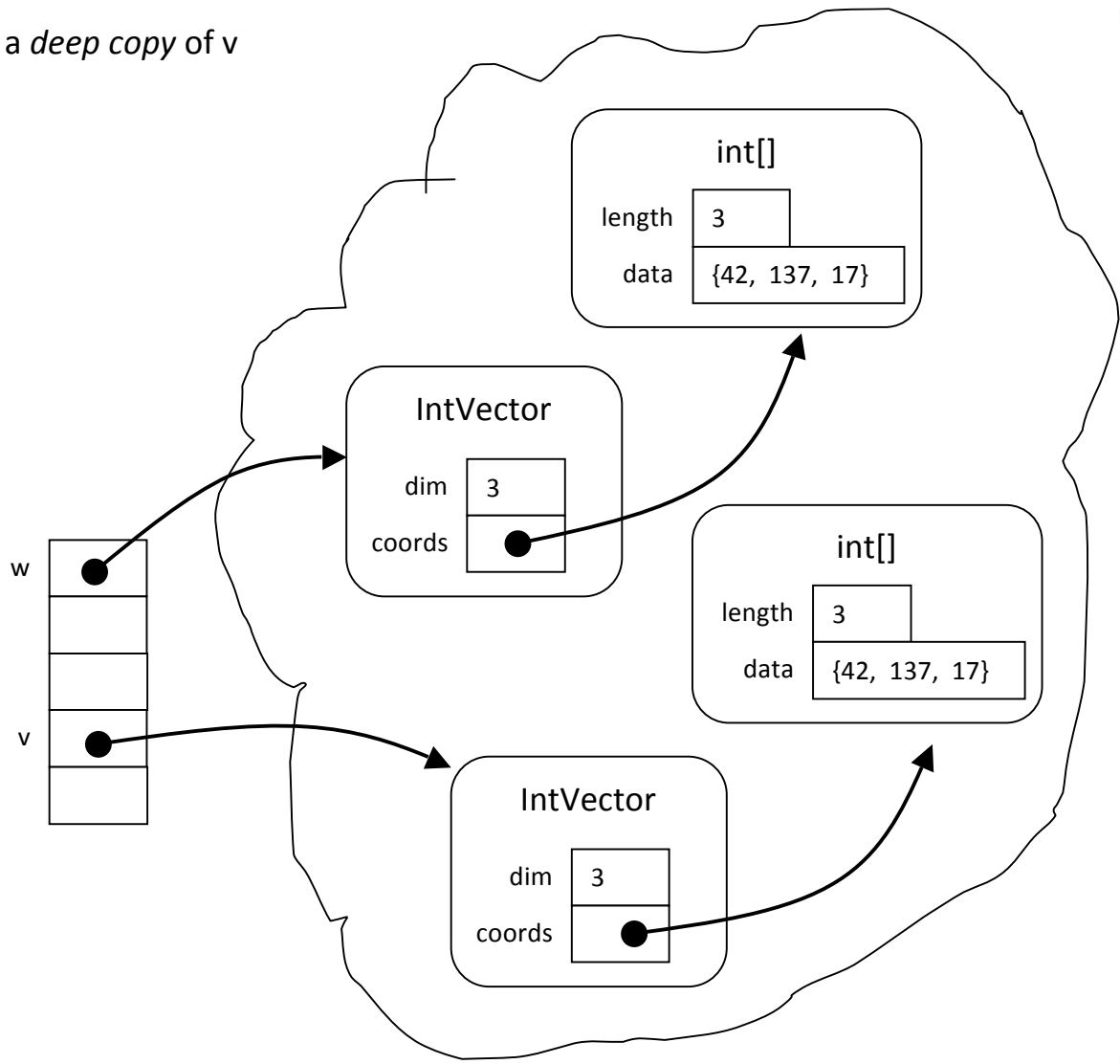
Suppose `w` is a shallow copy of `v`; i.e., `w` is obtained by copying the fields of `w`. Since the `coords` field is a reference, we just copy the reference. Thus, the `coords` fields of `v` and `w` end up referring to the same array.

w is a *shallow copy* of v



This can be dangerous, since any modification to the coords array through w also affects v. What we probably want is, in fact, a completely independent copy — a **deep copy** — of this array, like this:

w is a *deep copy* of v



A Copy Constructor for `IntVector`

Here is the code for a copy constructor that builds a deep copy of an `IntVector` object.

```
public IntVector(IntVector existing)
{
    dim = existing.dim;
    coords = new int[dim];

    for (int i = 0; i < dim; ++i)
    {
        coords[i] = existing.coords[i];
    }
}
```

Note that we could use `System.arraycopy()`¹ instead of the for loop.

clone() for IntVector

By default, `Object.clone()` creates shallow copies. That is OK for `Point`, but not for `IntVector`. Here is how to make a deep copy.

¹ See <http://docs.oracle.com/javase/8/docs/api/java/lang/System.html>.


```
@Override
public IntVector clone()
{
    try
    {
        IntVector copy
            = (IntVector) super.clone();

        // Object.clone() copies fields, now
        // make it into deep copy

        copy.coords = new int[dim];
        for (int i = 0; i < dim; ++i)
        {
            copy.coords[i] = coords[i];
        }
        return copy;
    }
    catch (CloneNotSupportedException e)
    {
        // should never happen...
        return null;
    }
}
```

Shallow versus Deep Comparison

The shallow versus deep issue also arises when implementing `equals()`. For example, `ArrayList`'s `equals()` method does a shallow comparison of two `ArrayList`s: they are “equal” if they have the same length and contain identical values in the same order. To implement `IntVector`'s `equals()` properly, we must do a deep comparison:

```
@Override
public boolean equals(Object obj)
{
    if (obj == null ||
        obj.getClass() !=
            this.getClass()) return false;
    IntVector other = (IntVector) obj;

    if (dim == other.dim)
    {
        // Check whether all coordinates are
        // the same
        for (int i = 0; i < dim; ++i)
        {
            if (coords[i] != other.coords[i])
            {
                return false;
            }
        }
        return true;
    }
    else
    {
        return false;
    }
}
```

Note. For comparing the `int` arrays, you could also use the utility

```
Arrays.equals(coords, other.coords).
```

However, since the class `int[]` does not override `equals()`, the following will **not** work:

```
coords.equals(other.coords)
```

Comments on Overriding Methods

Notice that the return type in `IntVector.clone()` is `IntVector`, while the return type in `Point.clone()` is `Object`. Either way is correct. The potential advantage of the former is that we can avoid the cast we needed with `Point`.

Here are some additional things you can and cannot do when you override a method.

- You **cannot** change the method's name or parameter types.

- You **can** change the return type, as long as the new type is **compatible** with the original.
- You **can** change a method from protected to public, but you **cannot** make the access more restrictive.
- You **can** omit a throws declaration, but you **cannot** add a throws declaration.