

CS 228: Introduction to Data Structures

Lecture 6

Friday, September 2, 2016

Example (Continued). Recall Wednesday's two `Point` objects.

```
Point p = new Point(1, 2);  
Point q = new Point(3, 4);
```

If we now write:

```
p = q;
```

The object that `p` referenced before the statement (the `Point` with coordinates (1,2)) is now unreferenced — it becomes “garbage”. Unreferenced objects are eventually ***garbage collected***.

Object Types versus Primitive Types

The next table compares object types and primitive types.

	Object types	Primitive types
Contains a	reference	built-in
How defined?	class definition	9, 42.5, 'h', false
How created?	"new"	default (usually zero)
How used?	method	operators: +, -, *, etc
Testing equality	<code>equals()</code> (override)	<code>==</code>

The last row deserves further explanation . . .

Equality and the `equals()` Method

The operation "`x == y`" determines whether variables `x` and `y` have the same values. When `x` and `y` are of the same primitive type, this corresponds to the usual notion of equality testing. For instance, suppose `i` and `j` are `int` variables and we set `i = 2`. Then, if we set `j = 3`, `i == j` is false and if we set `j = 2`, `i == j` is true.

When `p` and `q` are of object types, however, “`p == q`” determines whether variables `p` and `q` **reference** the same object. It does not determine whether the objects are “the same”. For instance, suppose we execute the following statements.

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

Intuitively, `p` and `q` are “the same”; however, `p == q` is false, because `p` and `q` reference different objects.

In Java, the notion of equality testing is captured by the `equals()` method. The default implementation of `equals()` in `java.lang.Object` just compares object references. Thus, if we used this default implementation, `p.equals(q)` would be false; i.e., no different than if we used `p == q`. If we want `p.equals(q)` to return `true` if and only if the `x` and `y` coordinates of `p` and `q` match (the usual notion of “sameness” of `Point` objects), we need to **override** the default implementation of `equals()`¹.

¹The Java `Point` class (`java.awt.Point`) implements `equals()` exactly as we want it: Points are equal only if they have the same coordinates.

The code below — which should be inserted within the body of the `Point` class — illustrates the standard way to override `equals()`.

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

    Point other = (Point) obj;
    return x == other.x && y == other.y;
}
```

Note that `obj`, the method argument, must be an `Object`, not a `Point`, because the signature of `equals()` in `java.lang.Object` is

```
boolean equals(Object obj)
```

and, to override a method, we must match its signature.

Our implementation of `equals()` first verifies that `obj` is not `null` and that it is also a `Point` object (if the other object is not a `Point`, we should not attempt to examine its coordinates). After verifying that `obj` is indeed a `Point`, we need to downcast `obj` to the `Point` type, to let the compiler know that we can access the instance variables `x` and `y`.²

Note. Some textbook authors and developers use the `instanceof` operator in the `equals()` method to test if objects are of the same class:

```
if (!(obj instanceof Point))  
    return false;
```

We will not do that here; instead, we will always test for class equality using `getClass()`. The reason is that `instanceof` does not work correctly when inheritance is involved; in fact it will be incorrect for subtypes of `Point`. To fully understand the issues, you need to understand the formal definition of `equals()`. We provide this formal definition next, for the benefit of the mathematically inclined — we will not cover it in class.

² Note that we can access the instance variables `x` and `y`, because `equals()` is defined within the body of the `Point` class.

Formal Definition of `equals()`. The Java documentation³ specifies that `equals()` implements an *equivalence relation* on non-null object references. That is, it satisfies the following properties.

- **Reflexivity:** for any non-null reference value `x`, `x.equals(x)` should return `true`.
- **Symmetry:** for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- **Transitivity:** for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- **Consistency:** for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- **Nothing equals null except null:** For any non-null reference value `x`, `x.equals(null)` should return `false`.

³ [http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals\(java.lang.Object\)](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals(java.lang.Object))

Exercise: Which of these properties imply that we should use `getClass()` instead of `instanceof()` in `equals()`?

`equals()` and the `String` class

Suppose we do the following.

```
String s = "hurley";  
String t = "HURLEY".toLowerCase();
```

Now strings `s` and `t` contain the same characters, but they ***are not*** the same object.

```
System.out.println(s == t);           // false
```

As you probably saw in ComS 227, the proper way to test if two strings are the same (i.e., contain the same characters), is to use the `equals()` method.

```
System.out.println(s.equals(t));      // true
```

This behaves as you would expect, because the implementors of Java have done some work for you: the `String` class overrides `equals()` to check whether the characters are the same.

Another Example

The same pattern we used for the `Point` class to override `equals()` for other classes. Here is an `equals()` method for the `Bee` class:

```
@Override
public boolean equals(Object o)
{
    if (o == null ||
        o.getClass() != getClass())
    {
        return false;
    }

    // typecast o to Bee so that we can
    // compare data members
    Bee b = (Bee) o;

    // Compare the data members and return
    // accordingly
    return b.size == size
}
```



```
        && (b.color == color ||  
        b.color != null &&  
        b.color.equals(color))  
        && (b.swarm == swarm ||  
        b.swarm != null &&  
        b.swarm.equals(swarm));  
    }
```

Note how `equals()` calls itself. Of course, this (recursive) call is to the `equals()` method for the `String` class (which we assume is implemented).

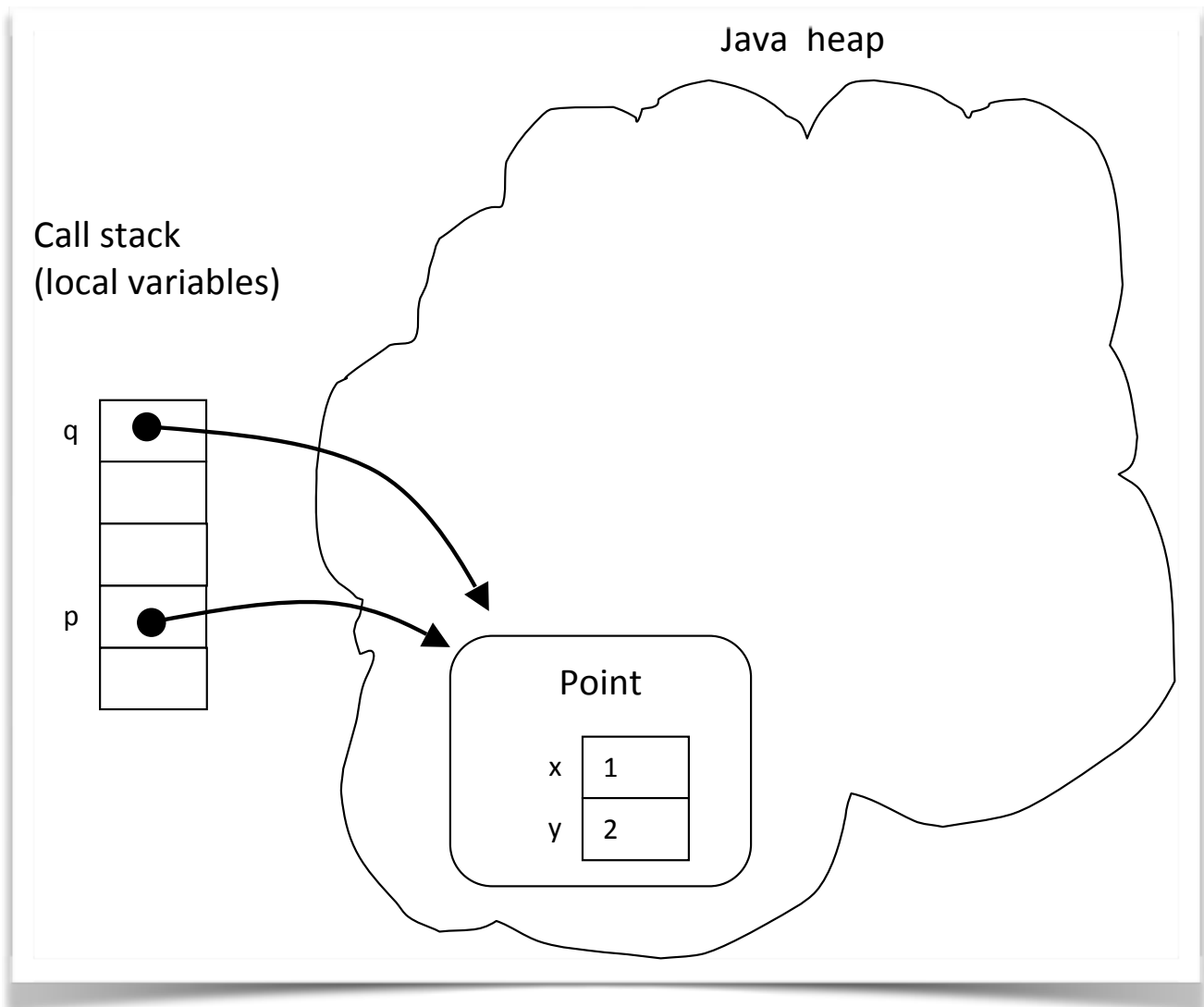
By inheritance, the same implementation can be used by any subclass of `Bee`, unless it is overridden by the subclass.

Copying and Cloning

If we execute these two statements

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

the result is this:



The second assignment does not make a copy of the `Point` object, it just assigns the reference. That is, `q` is now an ***alias*** for `p`. This is potentially dangerous, since any changes we make through `q` affect `p` as well, which might not be what we want.

In order to make an actual copy of an object, we have to implement a special method to do so. Two common options are to

- write a ***copy constructor*** or
- write a cloning method.

We will illustrate these approaches using the `Point` class.

Copy Constructors

A copy constructor initializes the object under construction (“this” object) using the values from an existing one:

```
public Point(Point existing)
{
    this.x = existing.x;
    this.y = existing.y;
}
```

Usage:

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

Cloning

A cloning method returns a **new** object from the values in this object. Here is an ad-hoc cloning method for the `Point` class.

```
public Point makeClone()  
{  
    Point copy = new Point();  
    copy.x = this.x;  
    copy.y = this.y;  
    return copy;  
}
```

Usage:

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

Another option is to override Java's `Object.clone()` method. For this, you either have to explicitly declare that your class implements the `Cloneable` interface or some superclass of your class must implement `Cloneable`. Thus, the declaration for `Point` would be

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

We will see how to override `clone()` next time.

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. The code is:

```
@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);
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 (the code is posted on BB).

```
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];
}
```

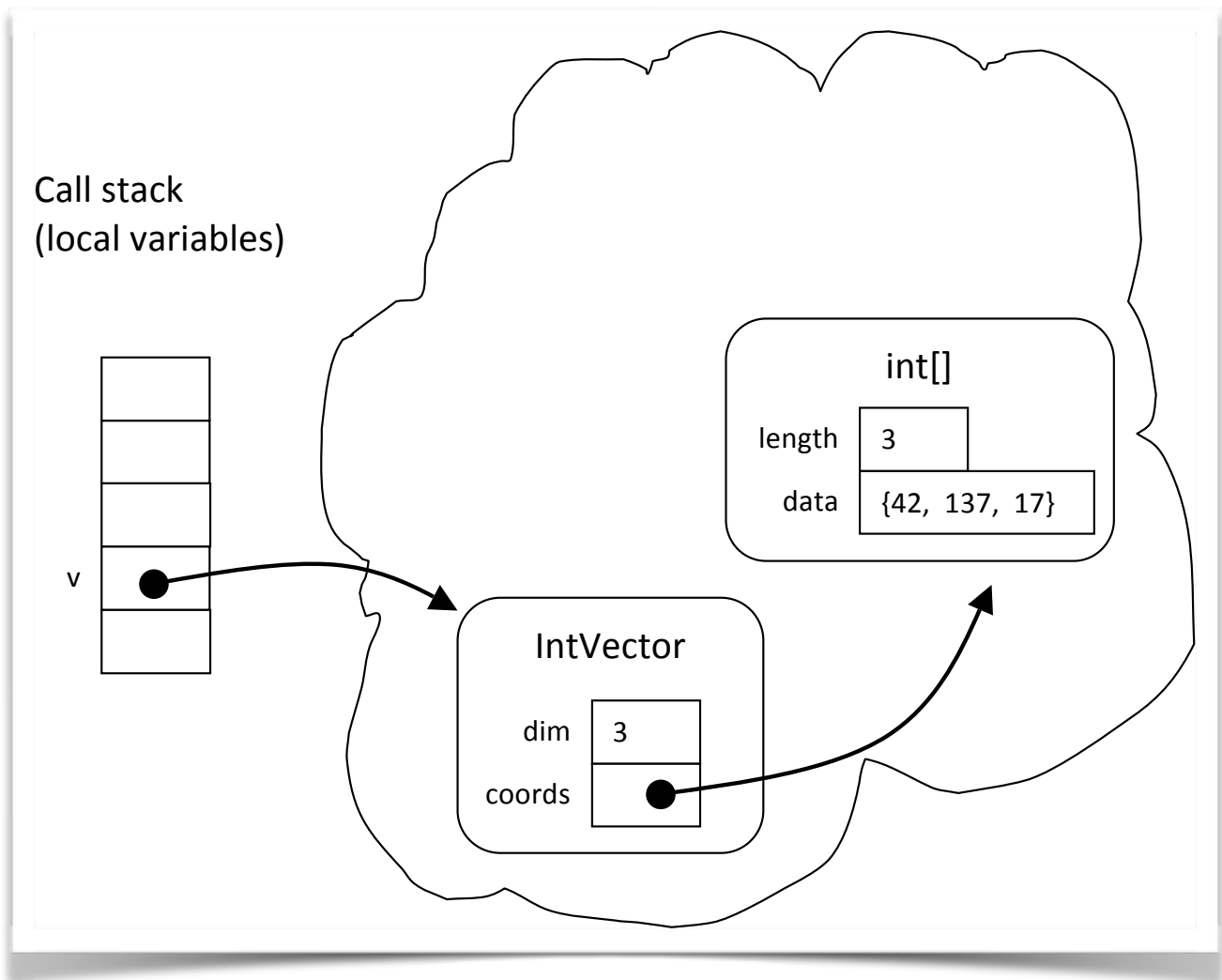
Here is the setter.

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

Now, suppose we execute the statements below:

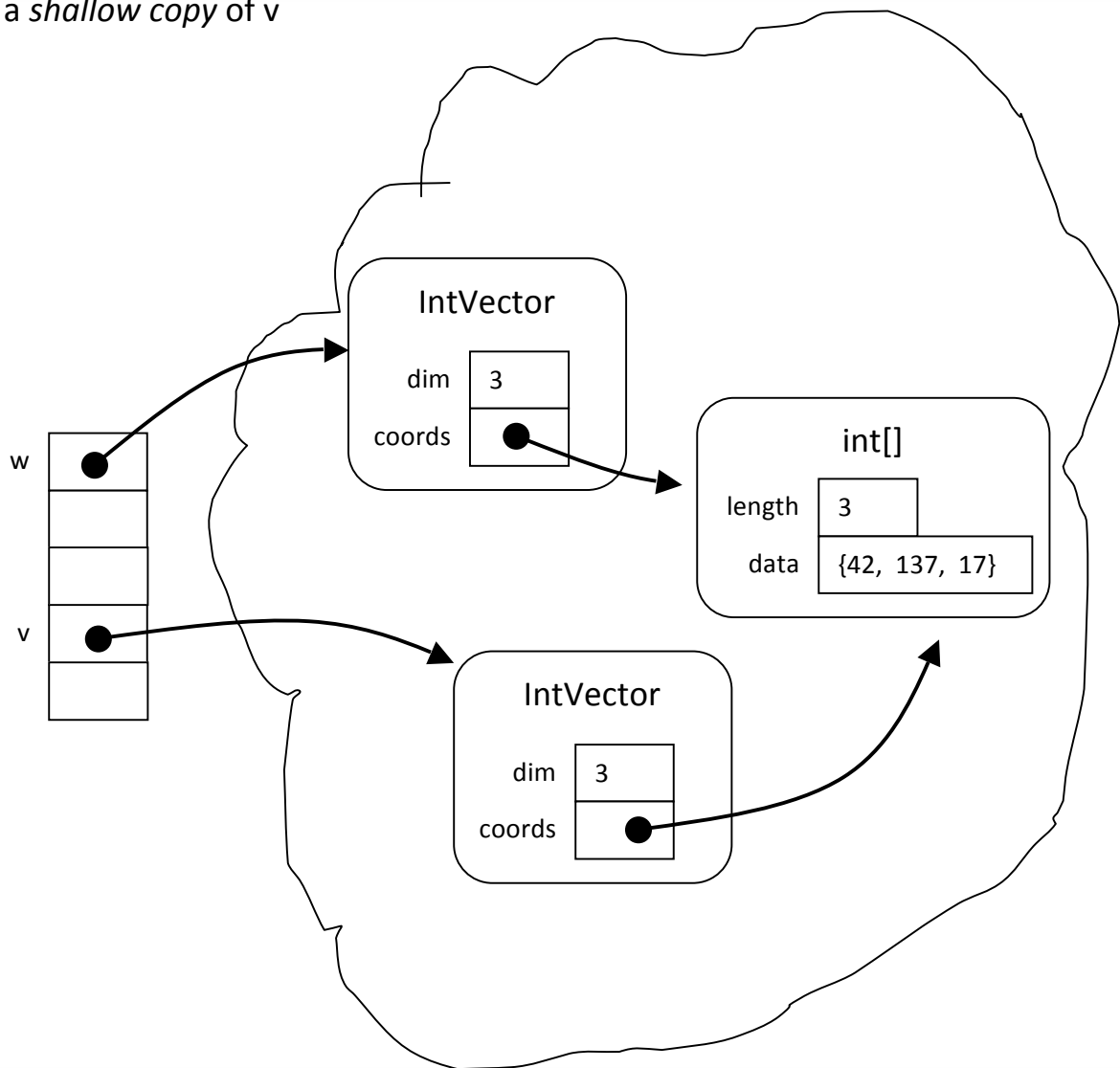
```
IntVector v = new IntVector(3);
v.set(0, 42);
v.set(1, 137);
v.set(2, 17);
```

The result is:



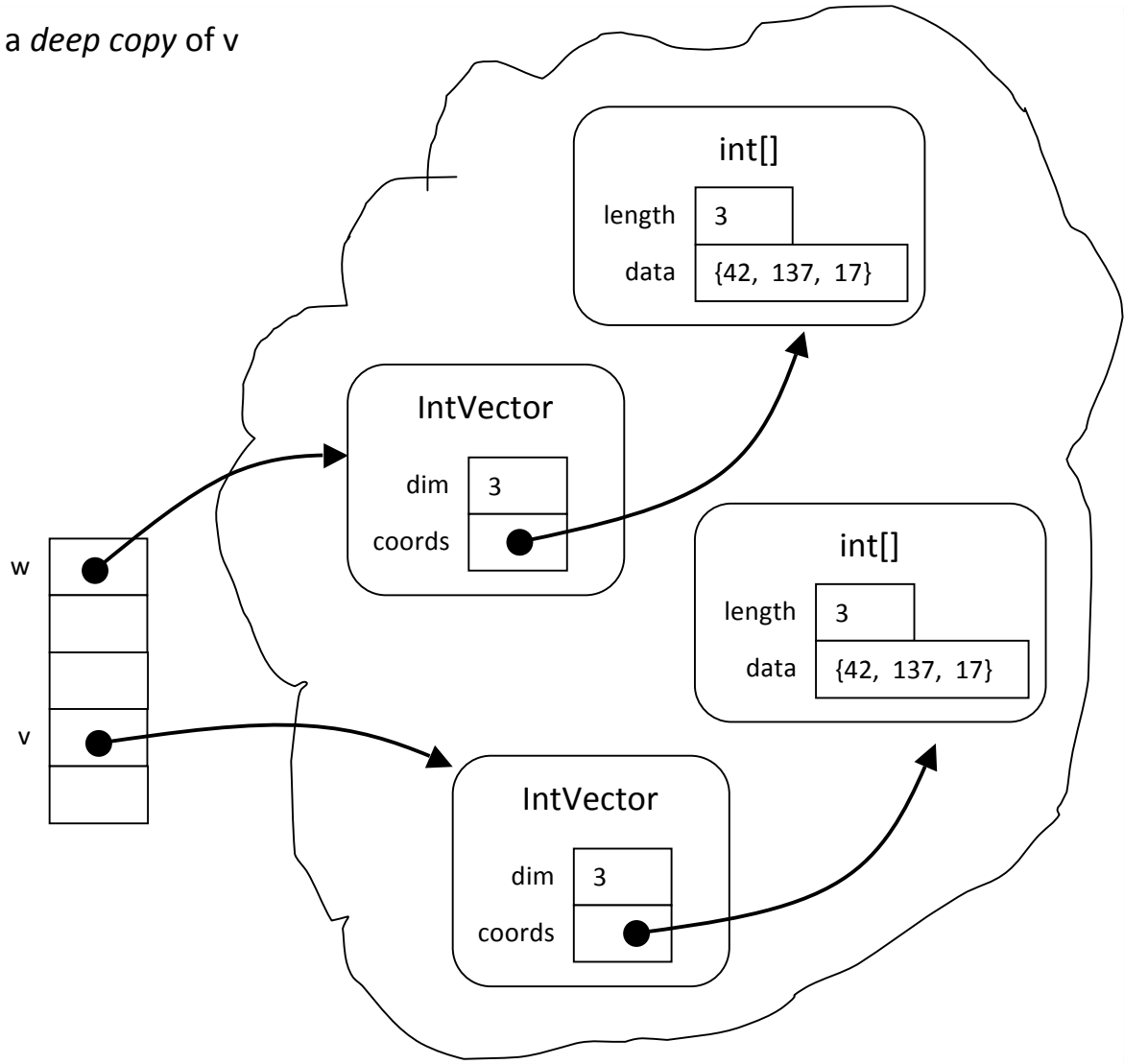
Suppose w is a shallow copy of v ; i.e., w is obtained by copying the fields of w . Since the `coords` field is a reference, we just copy the reference. Thus, the `coords` fields of v and w end up referring to the same array.

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()`⁴ 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.

⁴ 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.

static Fields and Methods

The keyword **static** in Java means “associated with the class as a whole, not with an instance”. Fields and methods can be static.

A **static field** is a single variable shared by a whole class of objects; its value does not vary from object to object. Thus, static fields are also called **class variables**. If we declare a field `static`, there is just one field for the whole class. One common use of static fields is to define constants, such as `Math.PI`, that are **static** and **final**. Here is another example.

Example. Suppose we want to keep track of the number of `Person` objects that we have constructed. It does not make sense for each object to have its own copy of this number: we would have to update every `Person`’s number whenever a new `Person` is created. It makes more sense to have a single variable, a static field, for the entire class that counts the number of people created thus far. The constructor increments this static field, called `numberOfPeople`, by one.

```
class Person {  
    public static int numberOfPeople;  
    public String name;  
    public Person(String name) {  
        this.name = name;  
        numberOfPeople++;  
    }  
}
```

If we want to look at the variable `numberOfPeople` from another class, we write it in the usual notation, but we prefix it with the class name rather than the name of a specific object. For example,

```
int kids = Person.numberOfPeople / 4;
```

The following works too, but has nothing to do with `joe` specifically.

```
int kids = joe.numberOfPeople / 4;
```

Don't do this; it is bad (confusing) style.

A **static method** does not implicitly pass an object as a parameter — in contrast, for example, the call `p.foo(q)` implicitly passes `p` as a parameter to `foo`. Thus, a static method can be used without creating an instance of the

class. One example is `Math.cos()`. Here's another one.

```
class Person {  
    ...  
    public static void printPopulation() {  
        System.out.println(numberOfPeople);  
    }  
}
```

Now, we can call `"Person.printPopulation()"` from another class. We can also call `"joe.printPopulation()"`, and it works, but it is bad style, and `joe` will NOT be passed along as `"this"`.

The `main()` method is always static, because when we run a program, we are not passing it an object.

Important: In a static method, there is no `"this"`! Any attempt to reference `"this"` will cause a compile-time error.