# R Programming

# Using Multiple Inheritance

## Multiple Inheritance

- The S4 object system is very powerful because of its *multiple inheritance* capability.
- This makes it possible to combine properties of classes in a very flexble way.
- This can really simplify the programming of some kinds of software.
- The following slides will show just a sample of the power of multiple inheritance.

## Vcoords Revisited

- The previous implementation of a vcoords class added a numeric `value` slot to the coords class.
- This parallels the S3 implementation of the class.
- S4 makes it possible to do something much slicker using *multiple inheritance*.
- We'll now inherit properties directly from two classes; `coords` and `numeric`.
- Coordinate properties will be obtained from the `coords` part and the numeric numeric properties from the `numeric` class.

## Recall the Definition of Coords

The `coords` class has two *slots* named `x` and `y` that contain numeric vectors.

The class is defined as follows:

```
> setClass("coords",
           representation(x = "numeric",
                          y = "numeric"))
```

It has appropriate constructor function `coords` and accessor functions for the `x` and `y` slots.

## Redefinition of Vcoords

The vcoords class can now be defined as inheriting from both the `coords` and `numeric` classes.

```
> setClass("vcoords",
           contains = c("numeric", "coords"))
```

Note that there is now no `values` slot. Instead `vcoords` objects “are” `numeric` with additional coordinate properties.

## The Vcoords Constructor

Here is a simplified version of the constructor for the `vcoords` class.

```
> vcoords =
      function(x, y, value)
      {
          if (length(x) != length(value) ||
              length(y) != length(value))
                  stop("invalid arguments")
          new("vcoords", value, x = x, y = y)
      }
```

The numerical property is taken from the numeric value `value` and named `x` and `y` slot values are specified explicitly.

## Treating Vcoords Objects as Numeric

Because vcoords objects “are” numeric, they can be treated as such. This can be made clearer by discarding their non-numeric aspects.

```
> vpts = vcoords(round(runif(4, 0, 10), 2),
                 round(runif(4, 0, 10), 2),
                 round(runif(4, 0, 100)))
```

Calling `as.numeric` or `as.vector` on the object will strip the coords information.

```
> as.numeric(vpts)
[1]  2 90 55 58

> as.vector(vpts)
[1]  2 90 55 58
```

## A Values Accessor

Discarding the coords information provides a way of accessing the purely numeric part of a `vcoords` object. This makes it possible to provide a values “accessor” that just coerces the object to be numeric
The values accessor can be written:

```
> values = function(obj) as.vector(obj)
```

It behaves as expected.

```
> values(vpts)
[1]  2 90 55 58
```

## The Previous Show Method

Given this `values` definition, the display/print methods defined before will continue to work because they work in terms of the accessor functions `xcoords`, `ycoords` and `values`, not using the particular implementation of the class.

Here I'll be assuming that the show method for `vcoords` objects pastes up a vector of strings of the form "($x$, $y$; $v$)" and then prints them without quotes.

```
> vpts
[1] (4.26, 9.20;  2) (4.60, 8.90; 90)
[3] (7.50, 8.48; 55) (7.57, 0.20; 58)
```

## Mathematical Functions

Because vcoords objects “are” numeric, nothing special is required to make mathematical transformations work.

```
> sqrt(vpts)
[1] (4.26, 9.20; 1.414214)
[2] (4.60, 8.90; 9.486833)
[3] (7.50, 8.48; 7.416198)
[4] (7.57, 0.20; 7.615773)
```

All that happens here is that a new object is made from the old by transforming the values part and retaining all the other aspects.

## Arithmetic Operations

Again, because vcoords objects “are” numeric, no work is required to implement things like “numeric + vcoords” or “vcoords + numeric”, they just happen.

```
> vpts + 100
[1] (4.26, 9.20; 102) (4.60, 8.90; 190)
[3] (7.50, 8.48; 155) (7.57, 0.20; 158)

> 100 + vpts
[1] (4.26, 9.20; 102) (4.60, 8.90; 190)
[3] (7.50, 8.48; 155) (7.57, 0.20; 158)
```

## Arithmetic Operations (Continued)

Operations will also work with numeric vectors, provided the operation does not produce a longer value that the `vcoords` object.

```
> vpts + seq(100, 400, by = 100)
[1] (4.26, 9.20; 102) (4.60, 8.90; 290)
[3] (7.50, 8.48; 355) (7.57, 0.20; 458)

> seq(100, 400, by = 100) + vpts
[1] (4.26, 9.20; 102) (4.60, 8.90; 290)
[3] (7.50, 8.48; 355) (7.57, 0.20; 458)
```

(Note that there will be warning messages produced if the length of the vcoords object is not a multiple of the length of the numeric one.)

## Arithmetic Operations (Continued)

If the numeric vector being added is longer than the `vcoords` object it is added to, it isn't clear how the result can be made into a vcoords object. In such cases the result reverts to being numeric.

```
> vpts + 1:8
[1]  3 92 58 62  7 96 62 66
```

This behavior can be overridden by defining explicit methods for the *Arith* group.

## Arithmetic on Two Vcoords Objects

When two `vcoords` objects of the same length are added the result is a vcoords object with value equal to the sum of the values and *whose other slots are taken from the first object*.

```
> vpts + vpts
[1] (4.26, 9.20;   4) (4.60, 8.90; 180)
[3] (7.50, 8.48; 110) (7.57, 0.20; 116)
```

## Arithmetic on Two Vcoords Objects

When two vcoords objects of different sizes are added, an attempt is made to produce a sensible result. (The `x` and `y` slots from the longer object are used for the `x` and `y` slots of the result.)

```
> wpts = vcoords(round(runif(2, 0, 10), 2),
                 round(runif(2, 0, 10), 2),
                 round(runif(2, 0, 100)))
> vpts + wpts
[1] (4.26, 9.20;  69) (4.60, 8.90; 162)
[3] (7.50, 8.48; 122) (7.57, 0.20; 130)

> wpts + vpts
[1] (4.26, 9.20;  69) (4.60, 8.90; 162)
[3] (7.50, 8.48; 122) (7.57, 0.20; 130)
```

## Overriding Default Behaviour

If you want other kinds of object behaviour, it is always possible to produce it by explicitly defining a method for the `Arith` group.

The default inheritance behaviour of the object system is set up so that it makes programming easy. To quote John Chambers:

> *It has a simple goal: To turn ideas into software, quickly and faithfully.*

If you design your software with a little care, making good use of inheritance, then programming with the S4 system should be simple and enable you to get your software running quickly.

## Use of Object-Oriented Methods

Object-oriented methods make it possible to manage the creation of complex software components.

A Major advantage of the methods is that they make it possible to reuse code through method inheritance.

If you do object-oriented programming in R, you should only consider using the S4 system. Its more formal nature and careful design make it possible to complex software that behaves predictably and is easy to extend.