Unicon—History

One predecessor of Unicon is Idol, Icon-derived Object Language.

Idol was developed at the University of Arizona by Clint Jeffery in 1988 for a graduate course on object-oriented programming.

"Unicon" initially stood for "UNIX Icon"—a version of Icon with a set of POSIX extensions by Shamim Mohamed developed in 1997. Mohamed learned Icon at the U of A but, because of Icon's lack of access to many OS facilities, used Perl for a variety of systems programming tasks. He wrote:

> "*While it is true that Perl substitutes for a conglomeration of* `sed`, `awk` *and shell scripts, it does so with some of the worst language features from them.*"

Unicon was his solution.

In 1999 Jeffery and Mohammed merged their work and other elements, such as an ODBC interface, into a single system, which was tentatively called Icon-2.

The name Unicon was later recycled, now standing for "Unified Extended Icon".

Class and method basics

Here is a simple Unicon class that models a coordinate-less rectangle:

```
class Rectangle(width, height)
    method area()
        return width * height
    end

    method perimeter()
        return width*2 + height*2
    end

    method str()
        return "Rectangle(" || width || "x" ||
                               height || ")"
    end
end
```

The class name is `Rectangle`.

It has two *attributes* (or *fields*), `width` and `height`.

It has three methods: `area`, `perimeter`, and `str`.

The `str` method produces a value such as

```
"Rectangle(3x4)"
```

Class and method basics, continued

For reference:

```
class Rectangle(width, height)
    method area()
        return width * height
    end
    ...
end
```

We can create instances of `Rectangle` like this:

```
r := Rectangle(3,4)

Rs := [Rectangle(3,4), Rectangle(5.0,7)]

r2 := Rectangle("3.4", 7)
```

For this class the constructor is essentially a record constructor—the supplied values are assigned directly to the fields `width` and `height`.

Methods are invoked with a familiar syntax:

```
a := r.area()

write("Perim: ", r.perimeter())

every write((!Rs).str())

write(Rectangle(2.9, 9.02).perimeter())
```

Class and method basics, continued

Just like any other Icon procedure call or record construction, no error checking is done. A null value is used for missing arguments and extra arguments are ignored.

All of the following execute without error:

```
r1 := Rectangle(3);

r2 := Rectangle("abc", "xyz");

r3 := Rectangle(7, 9, "abc");
```

Question: Which methods work for which of the above instances?

Unicon has no provision for access specifications like "public" and "private"—all attributes and methods are accessible in any context. This works:

```
procedure main()
    rr := Rectangle(3,4)
    rr.width := 20
    rr.height := 30
    write(rr.area())
end
```

Question: How can encapsulation be enforced?

Class and method basics, continued

The constructor is a procedure and can be treated like any other procedure:

```
R := Rectangle

r1 := R(5,7)

r2 := [R][1](3,4)

r3 := ("Rect"||"angle")(3,4);
```

Class and method basics, continued

Here is a program that produces a memory fault on SunOS 5.9:

```
class X()
    method f()
        write("in f()...")
    end
end

procedure main()
    x := X()
    x.f()
    x.g()
end
```

Execution:

```
% bogus
in f()...

Run-time error 302
File bogus.icn; Line 10
memory violation
Traceback:
   main()
   {record X__state_1(record X__state_1(2),
    record X__methods_1(1)) . g} from line 10
   in bogus.icn
```

The `initially` section

The simplistic behavior of assigning values in a constructor call to the attribute in the corresponding position is often inadequate.

An `initially` section can be added to trigger processing when the constructor is called.

```
class Rectangle(width, height, _area)
    method area()
        return _area
    end
    ...other methods...
    initially(w, h)
        write("initially: ",
             Image([width, height, _area],3))
        width := w
        height := h
        _area := w * h
end
```

If present, `initially` must follow all methods.

The `end` that ends the class definition also ends the `initially` section.

```
][ rr := Rectangle(3,4);
initially: L1:[&null,&null,&null]
   r := ...lots...

][ rr.area();
   r := 12  (integer)
```

If `initially(...)` is present, no attributes are automatically initialized.

`initially`, continued

The `initially` section can be used to enforce constraints on the constructor's arguments.

```
class Rectangle(width, height, _area)
    ...
    initially(w, h)
        if /w | /h then fail
        if not numeric(w) |
           not numeric(h) then fail
        width := w
        height := h
        _area := width * height
end
```

Execution:

```
][ rr := Rectangle(3);
Failure

][ rr := Rectangle(3, "x");
Failure

][ rr := Rectangle(3, "3.4");
   r := ...lots...
```

Note that by default an `initially` section succeeds.

Problem: There is no overloading of method names or the `initially` section. How could, for example, an omitted height default to the same value as the width?

```
r := Rectangle(3)
```

`initially`, continued

If there is a parameterless `initially` section then the arguments of the constructor call are used to initialize the attributes.

Example:

```
class Counter(count)
    method inc()
        count +:= 1
        return count
    end

    method value()
        return count
    end

    initially
        /count := 0
end
```

Usage:

```
][ A := Counter(10);
   r := ...lots...

][ B := Counter();
   r := ...lots...

][ A.value();
   r := 10  (integer)

][ B.value();
   r := 0  (integer)
```

The implicit variable `self`

Unicon's counterpart for Java's `this` is `self`.

One use is to distinguish between attributes and parameters:

```
class Rectangle(_area, width, height)
    initially(width,height)
        self.width := width
        self.height := height
        ...
end
```

Class specification—general form

Here is the general form of a class specification:

```
class classname(attribute1, attribute2, ..., attributeN)

        method method1(param1, param2, ..., paramN)
                ...code for method...
        end

        ...additional methods...

        initially(param1, param2, ..., paramN)
                ...code to execute upon construction...
end
```

Note that all attributes are specified in the list following the class name.

Here is a minimal class definition:

```
class X()
end
```

Method result sequences

Methods may fail, or produce a single result, or be generative, just like regular Icon procedures. Imagine a `side()` method that generates the width and height of a rectangle:

```
class Rectangle(width, height, _area)
    ...
    method side()
        suspend width | height
    end
    ...
end
```

Usage:

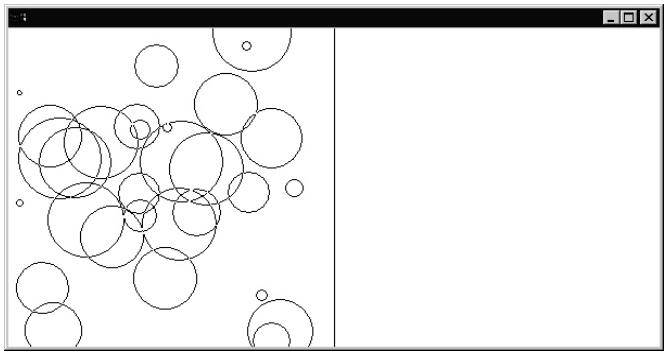
```
procedure main()
    rects := []
    every 1 to 20 do
        put(rects, Rectangle(?20, ?20))

    every r := !rects do
        if r.side() > 10 then
            write(r.str())
end
```

Output:

```
Rectangle(7x11)
Rectangle(2x15)
Rectangle(2x15)
Rectangle(11x13)
Rectangle(12x15)
Rectangle(15x5)
...
```

Circle drag/drop in Unicon

Recall this program from Graphics slide 31: (`drag1`)



```
record circle(x,y,r)
procedure main()
    WOpen("size=600,300","drawop=reverse")
    DrawLine(300,0,300,300)
    circles := make_circles()
    repeat case Event() of {
      &lpress:
        if c := point_in(circles, &x, &y) then {
          lastx := c.x; lasty := c.y
          r := c.r
          repeat case Event() of {
            &ldrag: {
              DrawCircle(lastx, lasty, r)
              DrawCircle(lastx := &x,lasty :=&y, r)
              }
            &lrelease: {
              DrawCircle(lastx, lasty, r)
              if &x <= 300 then {
                    DrawCircle(&x, &y, r)
                    c.x := &x; c.y := &y
                    }
              else
                    delete(circles, c)
                break
              }
            }
          }
        }
end
```

Circle drag/drop in Unicon, continued

Here is a version in Unicon. First, a `Circle` class:

```
class Circle(x, y, r)
    method has_pt(pt_x, pt_y)
        if sqrt((x-pt_x)^2+(y-pt_y)^2) < r then
            return self
    end

    method move_to(new_x, new_y)
        erase()
        x := new_x; y := new_y
        draw()
    end

    method erase()
        draw()
    end

    method draw()
        DrawCircle(x, y, r)
    end

    initially
        draw()
end
```

Note that the `initially` section counts on direct assignment of attributes from the constructor call.

The code above does not track the on-screen state (drawn or not) and thus places an additional responsibility on the caller.

Circle drag/drop in Unicon, continued

Main program:

```
procedure main()
    WOpen("size=600,300","drawop=reverse")
    DrawLine(300,0,300,300)

    circles := make_circles()

    repeat case Event() of {
      &lpress:
        if c := (!circles).has_pt(&x, &y) then {
          repeat case Event() of {

            &ldrag: c.move_to(&x, &y)

            &lrelease: {
              if &x <= 300 then
                    c.move_to(&x, &y)
              else {
                    c.erase()
                    delete(circles, c)
                    }
              break
              }
            }
          }
        }
end
```

Which version is better?

Inheritance

Here is a simple general form for specifying inheritance:

class *class-name* : *superclass-name* (*class-attributes*)
...
end

Here is a skeletal three class hierarchy to model geometric shapes:

```
class Shape(name)
end

class Rectangle: Shape (width, height)
end

class Circle: Shape (radius)
end
```

`Rectangle` is a subclass of `Shape` and has three attributes: `name`, `width`, and `height`.

`Circle` is a subclass of `Shape` and has two attributes: `name` and `radius`.

In Unicon there is no common superclass such as Java's `Object` class.

Superclass initialization

If a subclass has no `initially` section then the superclass's `initially` section is called.

The superclass's `initially` section is NOT CALLED if the subclass has an `initially` section.

Example:

```
class Shape(name)
    initially
        write("Shape's initially")
end

class Circle: Shape (radius)
end

class Rectangle: Shape (width, height)
    initially
        write("Rectangle's initially")
end

procedure main()
    c := Circle(5)
    r := Rectangle(3,4)
end
```

Output:

```
Shape's initially
Rectangle's initially
```

If a subclass requires an `initially` section then it should explicitly invoke the superclass `initially` section.

Superclass initialization, continued

Here is an example of invoking a superclass initially section:

```
class Shape(name)
    initially(nm)
        name := \nm | "<none>"
        write("Shape initially(), name = ", name)
end

class Rectangle: Shape (width, height)
    initially(w, h, nm)
        write("Rectangle initially()")
        width := w
        height := h
        self$Shape.initially(nm)
end

procedure main()
    r := Rectangle(3, 4)
    write(Image([r.name, r.width, r.height],3))

    r2 := Rectangle(5, 7, "B")
    write(Image([r2.name, r2.width, r2.height],3))
end
```

Output:

```
Rectangle initially()
Shape initially(), name = <none>
L1:["<none>",3,4]

Rectangle initially()
Shape initially(), name = B
L2:["B",5,7]
```

Note that there is no rule that specifies when superclass initialization must be done.

Method inheritance and overriding

Unicon's rule for method inheritance is a common one: Subclasses inherit superclass methods unless they supply their own version of a method.

```
class Shape()
    method area()
    end
end

class Rectangle: Shape (_width, _height)
    method area()
        return _width * _height
    end
end

class Circle: Shape (_radius)
end

procedure main()
    r := Rectangle(3, 4)
    c := Circle(5)

    write("r's area = ", r.area())
    write("c's area = ", c.area())
end
```

Output:

```
r's area = 12
```

Abstract classes

Unicon provides no means to declare a class or method as abstract.

One way to ensure that a subclass overrides a method is to add code that produces an error if an overriding method is forgotten:

```
class Shape()
    method area()
        stop("Shape.area() called!?")
    end
end
```

Question: Icon's association of type with values rather than variables implies that some errors are not detectable until the code is executed. Would it be possible to enforce an abstract declaration at compile time?

Inheritance and dynamic typing

Languages like Java use inheritance to allow code to be written in terms of a superclass and then be run with subclass instances.

```
public static Shape biggestArea(Shape shapes[ ]) {
    if (shapes.length == 0) return null;
    Shape it = shapes[0];
    for (int i = 1; i < shapes.length; i = i + 1) {
        if (shapes[i].getArea( ) > it.getArea( ))
            it = shapes[i];
        }
    return it;
    }
```

Because of Icon's value-based typing, inheritance is not needed to write such code.

In the following code there is no common superclass for `A` and `B`, but the routine `show_what()` can a handle a list of `A`s, `B`s, and any other objects that have a `what()` method.

```
class A()
    method what()
        return "I'm an A!"
    end
end

class B()
    method what()
        return "I'm a B..."
    end
end

procedure show_what(L)
    every o := !L do
        write(o.what())
end
```

Multiple inheritance

Unicon supports *multiple inheritance*—a class can have any number of superclasses. Here's an abstract example:

```
class A(_a)
   method f()
      write("A.f()")
   end
end

class B(_b1, _b2)
   method g()
      write("B.g()")
   end
end

class C(_c)
end
```

```
class D(_d1, _d2, _d3)
   method h()
      write("D.h()")
   end
end

class ABC: A : B : C (_abc1)
   method g()
      write("ABC.g()")
   end
end

class M : D : ABC (_m1, _m2)
end
```

A subclass inherits all attributes and methods of all its superclasses.

```
procedure main()
   abc := ABC()
   abc.f()  # calls A.f()
   abc.g()  # calls ABC.g()

   m := M()
   m.f()    # calls A.f()
   m.g()    # calls ABC.g()
   m.h()    # calls D.h()
end
```

Multiple inheritance, continued

A less abstract example—a `DrawableRectangle`:

```
class Drawable(_x, _y)
   method draw()
      stop("Drawable.draw() not overridden")
   end
   initially(x,y)
      _x := x; _y := y
end

class DrawableRectangle : Rectangle : Drawable ()
   method draw()
      DrawRectangle(_x, _y, _width, _height)
   end
   initially(w, h, x, y, nm)
      self$Rectangle.initially(w,h,nm)
      self$Drawable.initially(x,y)
end

procedure main()
   WOpen("size=300,300")
   rects := [ ]
   every i := 1 to 20 do
      put(rects, DrawableRectangle(?40, ?40, ?300, ?300))

   every r := !rects do
      if r.area() < 1000 then
         r.draw()

   WDone()
end
```