Operator overloading

NOTE: This is a replacement set. Please discard the two-sheet handout that has Ruby slides 159-167.

Operators as methods

It is possible to express most operators as method calls. Here are some examples:

```
>> 3.+(4)              => 7
>> "abc".*(2)          => "abcabc"
>> "testing".[ ](2)    => 115
>> "testing".[ ](2,3)  => "sti"
>> 10.==20             => false
```

In general, *expr1 op expr2* can be written as *expr1.op expr2*

Unary operators require a little more syntax:

```
>> 5.-@()          => -5
```

Problem: What are some binary operations that can't be expressed as a method call in Ruby?

Operator overloading

In most languages at least a few operators are "overloaded"—an operator stands for more than one operation.

Examples:

C: + is used to express addition of integers, floating point numbers, and pointer/integer pairs.

Java: + is used to express addition and string concatenation.

Icon: *x produces the number of...
- characters in a string
- values in a list
- key/value pairs in a table
- results a "co-expression" has produced
- and more...

What are examples of overloading in Ruby? In ML?

Operator overloading, continued

As a simple vehicle to study overloading in Ruby, imagine a dimensions-only rectangle:

```
class Rectangle
   def initialize(w,h); @width = w; @height =h; end
   def area; @width * @height; end
   attr_reader :width, :height

   def inspect
      "%g x %g Rectangle" % [@width, @height]
   end
end
```

Usage:

```
>> r = Rectangle.new(3,4)     => 3 x 4 Rectangle
>> r.area                     => 12
>> r.width                    => 3
```

Note that an inspect method is supplied to produce a more concise representation in irb.

Operator overloading, continued

Let's imagine that we can compute the "sum" of two rectangles:

```
>> a = Rectangle.new(3,4)     => 3 x 4 Rectangle

>> b = Rectangle.new(5,6)     => 5 x 6 Rectangle

>> a + b                      => 8 x 10 Rectangle

>> c = a + b + b              => 13 x 16 Rectangle

>> (a + b + c).area           => 546
```

As shown above, what does Rectangle + Rectangle mean?

Operator overloading, continued

Our vision:

```
>> a = Rectangle.new(3,4)     => 3 x 4 Rectangle
>> b = Rectangle.new(5,6)     => 5 x 6 Rectangle
>> a + b                      => 8 x 10 Rectangle
```

Here's how to make it so:

```
class Rectangle
   def + rhs
      Rectangle.new(self.width + rhs.width, self.height + rhs.height)
   end
end
```

Remember that a + b is equivalent to a.+(b). We are invoking the method "+" on a and passing it b as a parameter. The parameter name, rhs, stands for "right-hand side".

Instead of the above, would the following work? Would it be better in some cases?

```
Rectangle.new(@width + rhs.width, @height + rhs.height)
```


Operator overloading, continued

For reference:

```
def + rhs
   Rectangle.new(self.width + rhs.width, self.height + rhs.height)
end
```

Here is a faulty implementation, and usage of it:

```
def + rhs
   @width += rhs.width; @height += rhs.height
end

>> a = Rectangle.new(3,4)     => 3 x 4 Rectangle

>> b = Rectangle.new(5,6)     => 5 x 6 Rectangle

>> a+b                        => 10

>> a                          => 8 x 10 Rectangle
```

What's the problem?

Operator overloading, continued

Just like with regular methods, we have complete freedom to define what's meant by an expression using an overloaded operator. For example, a traditionally applicative operator can be turned into an imperative operation.

Here is a method for Rectangle that defines unary minus to be an imperative "rotation":

```
def -@     # Note: '-@' is used instead of just '-' to distinguish the unary form
   @width, @height = @height, @width         # "parallel assignment" to swap
   self
end

>> a = Rectangle.new(2,5)     => 2 x 5 Rectangle

>> a                          => 2 x 5 Rectangle

>> -a                         => 5 x 2 Rectangle

>> a + -a                     => 4 x 10 Rectangle

>> a                          => 2 x 5 Rectangle
```

Any surprises?

Operator overloading, continued

Consider "scaling" a rectangle by some factor. Example:

```
>> a = Rectangle.new(3,4)      => 3 x 4 Rectangle

>> b = a * 5                   => 15 x 20 Rectangle

>> c = b * 0.77                => 11.55 x 15.4 Rectangle
```

Implementation:

```
def * rhs
   Rectangle.new(self.width * rhs, self.height * rhs)
end
```

A problem:

```
>> a                           => 3 x 4 Rectangle

>> 3 * a
TypeError: Rectangle can't be coerced into Fixnum
```

What's wrong?

Operator overloading, continued

Imagine a case where it is useful to reference width and height uniformly, via subscripts:

```
>> a = Rectangle.new(3,4)      => 3 x 4 Rectangle

>> a[0]                        => 3

>> a[1]                        => 4

>> a[2]                        ArgumentError: out of bounds
```

Remember that a[0] is a.[ ](0).

Implementation:

```
def [ ] n
   case n
   when 0 then width
   when 1 then height
   else raise ArgumentError.new("out of bounds")   # Exception!
   end
end
```


Ruby is mutable

The ability to define meaning for operations like Rectangle + Rectangle leads us to say that Ruby is *extensible*.

But Ruby is not only extensible, it is also *mutable*—we can change the meaning of expressions.

For example, if we wanted to be sure that a program never used integer addition or negation, we could do this:

```
class Fixnum
   def + x
      raise "boom!"
   end
   def -@
      raise "boom!"
   end
end
```

In contrast, C++ is extensible, but not mutable. In C++, for example, you can define the meaning of Rectangle * int but you can't change the meaning of integer addition, as we do above.

Inheritance

Inheritance in Ruby

Java vs. Ruby

Modules and mixins

More to come on inheritance...

Inheritance in Ruby

A simple example of inheritance can be seen with clocks and alarm clocks. An alarm clock is a clock with a little bit more. Here are trivial models of them in Ruby:

```
class Clock
   def initialize time
      @time = time
   end
   attr_reader :time
end
```

```
class AlarmClock < Clock
   attr_accessor :alarm_time
   def initialize time
      super(time)
   end
   def on;  @on = true end
   def off;  @on = false end
end
```

The less-than symbol specifies that AlarmClock is a subclass of Clock.

Just like Java, a call to super is used to pass arguments to the superclass constructor.

Ruby supports only single inheritance but "mixins" provide a solution for most situations where multiple inheritance is useful. (More on mixins later.)

Inheritance, continued

Usage is not much of a surprise:

```
>> c = Clock.new("12:00")          => #<Clock:0x2c44198 @time="12:00">

>> c.time                          => "12:00"

>> ac = AlarmClock.new("12:00")    => #<AlarmClock:... @time="12:00">

>> ac.time                         => "12:00"

>> ac.alarm_time = "8:00"          => "8:00"

>> ac.on                           => true

>> ac
=> #<AlarmClock:0x2c30c38 @on=true, @time="12:00", @alarm_time="8:00">
```

Note that AlarmClock's @on and @alarm_time attributes do not appear until they are set.

To keep things simple, times are represented with strings.

Inheritance, continued

The method alarm_battery creates a "battery" of num_clocks AlarmClocks. The first is set for whenn. The others are set for intervals of interval minutes.

```
def alarm_battery(whenn, num_clocks, interval)
   battery = [ ]
   num_clocks.times {
      c = AlarmClock.new("now")              # Imagine this works
      c.alarm_time = whenn
      whenn = add_time(whenn, interval)      # Imagine this method
      battery << c
      }
   battery
end
```

Usage:

```
>> battery = alarm_battery("8:00", 10, 5)    => Array with ten AlarmClocks

>> battery.size                              => 10
>> p battery[2]
#<AlarmClock:0x2c19d94 @alarm_time="8:10", @time="22:06">
```

Inheritance: Java vs. Ruby

Here is a Ruby method to activate all the alarms in a battery:

```
def activate_alarms(clocks)
   for ac in clocks
      ac.on
   end
end
```

Here is an analog in Java:

```
static void activate_alarms ( ArrayList<AlarmClock> clocks ) {
   for (AlarmClock ac: clocks)
      ac.on();
   }
```

Is there a practical difference between the two? Or are these just two ways to express the same computation?

Inheritance: Java vs. Ruby, continued

Another group has developed a WeatherClock. It is essentially an alarm clock but lets the user specify adjustments based on the weather. You might want be awakened earlier than normal if it is snowy, to provide time to shovel out the car. Here is WeatherClock, in Java:

```
class WeatherClock extends Clock {
   public void on() ...
   public void addCondition(WeatherCondition c, AlarmAdjustment a) ...
   ...
   }
```

We'd like to add a WeatherClock to our alarm battery,

```
ArrayList<AlarmClock> battery = alarm_battery(...)   // Java
WeatherClock wc = new WeatherClock(...);
battery.add(wc);      # line 31
```

but there is a problem:

```
clock.java:31: cannot find symbol 'method add(WeatherClock)'
location: class java.util.ArrayList<AlarmClock>
      battery.add(wc);
```

What's wrong? How can we fix it?

Inheritance: Java vs. Ruby, continued

At hand:

```
ArrayList<AlarmClock> battery = alarm_battery(...)
WeatherClock wc = new WeatherClock(...);
battery.add(wc);      # line 31

clock.java:31: cannot find symbol 'method add(WeatherClock)'
```

Recall WeatherClock:

```
class WeatherClock extends Clock { ... }
```

Because WeatherClock is a Clock not an AlarmClock, we can't put it in an ArrayList of AlarmClock.

How about relaxing the types? Can we use ArrayList battery instead of ArrayList<AlarmClock>battery? That is, just have battery hold Objects instead of AlarmClocks?

Inheritance: Java vs. Ruby, continued

Excerpts from an Object-based version:

```
ArrayList battery = alarm_battery(...);          # Holds instances of Object
WeatherClock wc = new WeatherClock(...);
battery.add(wc);                                  # OK: WeatherClock is-a Object
activate_alarms(battery);
```

and...

```
static void activate_alarms(ArrayList clocks) {
   for (Object o: clocks) {
      AlarmClock ac = (AlarmClock)o;              # line 24
      ac.on();
      }
   }
```

Result:

```
Exception in thread "main" java.lang.ClassCastException: WeatherClock
      at clock2.activate_alarms(clock2.java:24)
```

Now what's wrong? Does this call for a design pattern?

Inheritance: Java vs. Ruby, continued

In Ruby, things are simpler:

```
class WeatherClock < Clock   # Just like the Java version, WeatherClock is-a Clock
   ...                       # not an AlarmClock.
   def on; ... end
end
```

and...

```
def activate_alarms(clocks)
   for ac in clocks
      ac.on
   end
end
```

and...

```
battery = alarm_battery("8:00", 10, 5)
battery << WeatherClock.new(...)
activate_alarms(battery)
```

It works! (How?!)

Modules

A Ruby *module* can be used to group related methods for organizational purposes.

Imagine a collection of methods to comfort a homesick ML programmer at Camp Ruby:

```
module ML
   def ML.hd a
      a[0]
   end
   def ML.drop a, n
      a[n..-1]
   end
   ...more...
end

>> a = [10, "twenty", 30, 40.0]      => [10, "twenty", 30, 40.0]

>> ML.hd(a)                          => 10

>> ML.drop(a, 2)                     => [30, 40.0]

>> ML.tl(ML.tl(ML.tl(a)))            => [40.0]
```


Modules as "mixins"

In addition to providing a way to group related methods, a module can be "included" in a class. When a module is used in this way it is called a "mixin" because it mixes additional functionality into a class.

Here is a revised version of the ML module:

```
module ML
   def hd; self[0]; end

   def tl; self[1..-1]; end

   def drop n; self[n..-1]; end

   def take n; self[0,n]; end
end
```

Note that these methods have one less parameter, operating on self instead of the parameter a. For comparison, here's the first version of tl:

```
def ML.tl a
   a[1..-1]
end
```


Mixins, continued

We can mix our ML methods into the Array class like this:

```
require 'ML'          # loads ML.rb if not already loaded
class Array
   include ML
end
```

After loading the above code, we can use those ML methods on arrays:

```
>> ints = (1..10).to_a      => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>> ints.hd                  => 1

>> ints.tl                  => [2, 3, 4, 5, 6, 7, 8, 9, 10]

>> ints.drop 3              => [4, 5, 6, 7, 8, 9, 10]
```

How could we add these same capabilities to the String class?

Mixins, continued

An include is all we need to add the same capabilities to String:

```
require 'ML'
class String
   include ML
end

>> s = "testing"       => "testing"

>> s.tl                => "esting"

>> s.hd                => 116          # How could we get "t" instead?

>> s.drop 5            => "ng"
```

Does Java have any sort of mixin capability?

In addition to the include mechanism, what other aspect of Ruby facilitates mixins?

(This slide intentionally left blank.)