This is yet another set of replacements.

The following slides supersede slides 180-184 in the set of 159-184 that was distributed on October 19.

Inheritance: Java vs. Ruby, continued

At hand:

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

battery = alarm_battery("8:00", 10, 5)

battery << WeatherClock.new("x")

activate_alarms(battery)
```

Do we really need activate_alarms?

Inheritance: Java vs. Ruby, continued

activate_alarms simply invokes the on method of each element in an array. Here's another way to do the same thing:

```
battery.each {|c| c.on }
```

Things work nicely because although the misguided WeatherClock folks don't consider their invention to be an AlarmClock, they did happen to choose "on" as the name of the method to activate it.

What could we do if instead of naming it "on" they had named it "activate"?

Inheritance: Java vs. Ruby, continued

Option 1: Add an on method to WeatherClock:

```
class WeatherClock
   def on
      activate
   end
end
```

However, if they've frozen it with WeatherClock.freeze, we'll see this:

```
TypeError: can't modify frozen class
```

Now what?

Inheritance: Java vs. Ruby, continued

Option 2: Borrow a little bit from the ADAPTER design pattern:

```
class AdaptedWeatherClock < WeatherClock
   def on
      activate
   end
end

battery = alarm_battery("8:00", 10, 5)

battery << AdaptedWeatherClock.new("x")

battery.each {|c| c.on }
```

Fact: Much of the complexity in some OO design patterns rises from accommodation of compile-time type checking.

interface in Java

Question:

What capability is provided by Java's interface construct?

One answer:

An interface specifies a set of operations that must be provided by every class that implements the interface. In turn, it can be determined at compile time whether a class claiming to implement the interface provides all the required methods.

For example, if a Java class wants to implement Iterable, it must provide a method called iterator. iterator must return an instance of a class that implements Iterator that in turn must provide hasNext, next, and remove methods.

Challenge: Keeping track of the time you spend, create a Java class X such that this code,

```
X x = new X();
for (Object o: x)
   System.out.println(o);
```

produces this output: x

Sidebar: const methods and in C++

In C++ a method can be declared as const, which indicates that the method won't change the value of any instance variables. For example, getting the length of a list shouldn't cause a change in the list, right? In C++ we'd say this:

```
class IntList {
   public:
      void addValue(int value) { ... }
      int get() { ...removes the first element and returns it...}
      int getLength() const { ... }
      ...
   };
```

const can be applied to a parameters to indicate that the parameter should not be modified:

```
void f(const IntList& ilist)        // w/o const, this is like f(IntList ilist) in Java
{
   int len = ilist.getLength();     // OK

   ilist.addValue(7);               // compilation error — ilist is const
}
```

Question: Is there a situation where getLength() might want to change a value?

Sidebar, continued

Here is C++ code inspired by actual events at one of Tucson's largest software companies:

```
void printInts(const IntList& ilist)
{
   for (int i = 0; i < ilist.getLength(); i++) {
        cout << ilist.get() << endl;
      }
}
```

Judicious use of const can catch bugs at compile time. Is the complexity it adds worth the benefit?

It is common for self-taught adopters of C++ to not see the value of const at first. When they do see the light, they then have a "const Day" to convert all the code over to using const. (As a rule you need to use const everywhere or nowhere; a mix usually doesn't work.)

A Big Question

Java's compile-time type checking can ensure that we won't have a run-time error because we forgot a method. Ruby doesn't provide that assurance. Example:

```
% cat iter3.rb
class X; end

for o in X.new          # Recall that 'for' expects the "in" value to respond to 'each'
   puts o
end

% ruby iter3.rb
iter3.rb:3: undefined method `each' for #<X:0x2838618> (NoMethodError)
```

The fix:

```
class X
   def each
      yield "x"
   end
end
```


A Big Question, continued

Big Question:

> Does the benefit of detecting missing methods at compile time outweigh the extra code and complexity required to allow that compile-time detection?

When would a forgotten each method most likely turn up in a Ruby application? In development? In testing? In production?

With a non-trivial iterator in mind, which is harder: remembering to write each or getting the iteration correct? Would there be tests focused on the operation of the iterator?

What does Test Driven Development add to the question?

You might think of statically-typed languages as offering a deal: "If you'll follow these rules I'll be able to detect certain types of errors when the program is compiled."

How does the execution model of a language change this question? For example, is compile-time type-checking as important in Java or Ruby as it is in C or C++?

Consider this: Before generics were added to Java, lots of successful systems were developed using Hashtable and Vector with extensive use of downcasts, like x = (X)v.get(i). Lots of Java shops are still not using generics.

http://www.madbean.com/anim/jarwars

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? What would be required to produce a comparable effect?

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

Modules and superclasses

The Ruby library makes extensive use of mixins.

The class method ancestors can be used to see the superclasses and modules that contribute methods to a class:

```
>> Array.ancestors      => [Array, Enumerable, Object, Kernel]

>> Fixnum.ancestors     => [Fixnum, Integer, Precision, Numeric, Comparable,
                            Object, Kernel]
```

The method included_modules shows the modules that a class includes. instance_methods can be used to see what methods are in a module:

```
>> Array.included_modules              => [Enumerable, Kernel]

>> Enumerable.instance_methods
=> ["collect", "detect", "max", "sort", "partition", "any?", "reject", "zip", "find", "min",
"member?", "entries", "inject", "all?", "select", "each_with_index", "grep", "to_a",
"map", "include?", "find_all", "sort_by"]

>> Comparable.instance_methods         => ["==", ">=", "<", "<=", "between?", ">"]
```

Modules and superclasses, continued

All classes include Kernel. If no superclass is specified, a class subclasses Object.

Example:

```
>> class X; end              => nil

>> X.ancestors               => [X, Object, Kernel]

>> X.included_modules        => [Kernel]

>> X.superclass              => Object
```


The Enumerable module

Some Ruby modules provide a collection of methods implemented in terms of a single operation. One such module is Enumerable:

```
>> Enumerable.instance_methods.sort
=> ["all?", "any?", "collect", "detect", "each_with_index", "entries", "find", "find_all",
"grep", "include?", "inject",  "map", "max", "member?", "min", "partition", "reject",
"select", "sort", "sort_by", "to_a", "zip"]
```

All of the methods of Enumerable are written in terms of a single method, each, which is an iterator.

If class implements each and includes Enumerable then all those 22 methods become available to instances of the class.

Enumerable, continued

Here's a class whose instances simply hold three values:

```
class Trio
   include Enumerable
   def initialize(a,b,c); @values = [a,b,c]; end
   def each
      @values.each {|v| yield v }
   end
end
```

Because Trio provides each and includes Enumerable, we can do a lot with it:

```
>> t = Trio.new(10,"twenty",30)      => #<Trio:0x28e9958 @values=[10, "twenty", 30]>

>> t.member?(30)                     => true

>> t.map { |e| e * 2 }               => [20, "twentytwenty", 60]

>> t.partition { |e| e.is_a? Numeric }       => [[10, 30], ["twenty"]]
```


The Comparable module

Another common mixin is Comparable. These methods,

```
>> Comparable.instance_methods
=> ["==", ">=", "<", "<=", "between?", ">"]
```

are implemented in terms of <=>.

Let's compare rectangles on the basis of areas:

```
class Rectangle
   include Comparable
   def <=> rhs
      diff = self.area - rhs.area
      case
      when diff < 0 then -1
      when diff > 0 then 1
      else 0
      end
   end
end
```

Comparable, continued

Usage:

```
>> r1 = Rectangle.new(3,4)    => 3 x 4 Rectangle
>> r2 = Rectangle.new(5,2)    => 5 x 2 Rectangle
>> r3 = Rectangle.new(2,2)    => 2 x 2 Rectangle
>> r1 < r2                    => false
>> [r1,r2,r3].sort            => [2 x 2 Rectangle, 5 x 2 Rectangle, 3 x 4 Rectangle]
>> [r1,r2,r3].min             => 2 x 2 Rectangle
>> r2.between?(r1,r3)         => false
>> r2.between?(r3,r1)         => true
```


Graphics with Tk

Tk Basics

Hello in Tk

Component callbacks

Updating a component

Sidebar: Named parameters

Drawing on a canvas

Mouse events

Grand finale: Pulsing circles

Tk basics

Tk is a library of components for building GUIs. It provides a typical collection of elements including buttons, labels, lists, scrollbars, text fields, and more.

Tk was created by John Outsterout at UCB for use with Tcl ("tickle"), a scripting language.

Many languages, including Ruby, have a library for working with Tk. Aside from Tcl, Tk is perhaps most often used with Perl.

There are other Ruby GUI toolkits but Tk is available on the widest range of platforms. (?)

The Ruby-specific documentation on Tk is pretty skimpy. (Take a look at the Tk section in http://ruby-doc.org/stdlib!) Tk tutorials generally refer readers to the Perl Tk documentation.

Disclaimer: The instructor only has spent a few hours experimenting with Tk. In a few minutes you'll know at least as much as he does about Tk.

Hello in Tk

Here is a nearly minimal Tk program that puts up a window with a label that says "Hello!"

```
# tkhello.rb
require 'tk'                # Later examples don't show this line but it is still needed.
root = TkRoot.new

TkLabel.new(root) {
   text "Hello!"
   font "Courier 36 bold italic"
   pack
   }

Tk.mainloop
```

Notes:

TkRoot.new creates a new top-level frame.

TkLabel.new creates a label and puts it in the root. It is configured via method calls in the associated block.

pack invokes a geometry manager. Without this call, only an empty frame appears.

Component callbacks

If a button is configured with a command, the associated block is executed when the button is pressed.

Here's a program that creates a button for each command line argument and prints as buttons are clicked. The button text size is proportional to the length of the label.

```
class Button                    # tkclick1.rb
   def initialize(label)
      TkButton.new{
         text label; font "Jokewood #{label.size*10}";
         command { puts "Got a click on '#{label}'" }
         pack
         }
   end
end
ARGV.each { |label| Button.new(label) }
Tk.mainloop
```

This example takes advantage of the fact that we really don't need to specify the root.

Updating a component

The following code creates a label that is updated with a count when a button is pressed:

```
def fmt_clicks n; "Clicks: %d" % n; end          # tkclick2.rb

label = TkLabel.new {
   text fmt_clicks(0); font "HelterSkelter 36"; pack }

TkButton.new {
   text "Push Here";
   font "Jokewood 36";
   clicks = 0
   command { clicks += 1
             label.configure(:text => (fmt_clicks clicks))
           }
   pack
   }
Tk.mainloop
```

A call to label.configure is used to update the text. The syntax ':text => ...' creates and passes a hash as the argument. (A sidebar on this follows.)

Sidebar: Named parameters

Some languages support *named parameters*—instead of parameters being specified by position, they are specified by name and are position-independent. Here is a generic example of equivalent calls using named parameters:

```
f(x = 3, y = 4)
f(y = 4, x = 3)
```

Ruby does not provide named parameters but does provide an interesting facility with a hash:

```
def f(a, b, h)
   p [a, b, h]
end

f(10, 20, :x => 10, :y => 2, "x"*5 => [ ])
```

The result:

```
[10, 20, {:x=>10, :y=>2, "xxxxx"=>[ ]}]
```

The hash specification must be the last part of the argument list. It is common to see Symbols used for the keys.

Drawing on a Canvas

The TkCanvas class provides a drawing surface.

```
width = 400             # tkcanvas1.rb
height = 300

canvas = TkCanvas.new { width width+10; height height+10 }
canvas.pack

args = [canvas, 5, 5, width, height]
TkcOval.new *args
TkcLine.new *args
TkcRectangle.new *args

Tk.mainloop
```

Objects are added to the canvas by creating instances of classes like TkcOval (canvas oval).

Mouse events

The bind method of Canvas specifies a "Proc" to be called whenever a particular event is recognized.

```
def do_press(x, y)          # tkmouse.rb
   printf("Button down at %d, %d\n", x, y)
end

def do_motion(x, y); printf("Motion at %d, %d\n", x, y); end

def do_release(x, y); printf("Button up at %d, %d\n", x, y); end

canvas = TkCanvas.new; canvas.pack

canvas.bind("1", lambda {|e| do_press(e.x, e.y)})
canvas.bind("B1-Motion", lambda {|x, y| do_motion(x, y)}, "%x %y")
canvas.bind("ButtonRelease-1", lambda {|x, y| do_release(x, y)}, "%x %y")

Tk.mainloop
```

Kernel#lamba creates a Proc from the associated block. Think of a Proc like an anonymous function in ML.

Grand finale: Pulsing Circles

```
class Circle
   SZ = 200
   def initialize(canvas, x, y)
      @canvas = canvas; @inc = 1; @ux = x - SZ/2; @uy = y - SZ/2
      @lx = @ux + SZ; @ly = @uy + SZ
      @oval = TkcOval.new(@canvas, @ux, @uy, @lx, @ly)
      @canvas.after(1) { tick }
   end
   def tick
      @inc *= -1 if @ux >= @lx or (@ux - @lx).abs > SZ
      @ux += @inc; @uy += @inc; @lx -= @inc; @ly -= @inc
      @oval.coords(@ux, @uy, @lx, @ly)
      @canvas.after(1) { tick }
   end
end

$canvas = TkCanvas.new { width 400; height 300; pack }
$canvas.bind("1", lambda {|e| do_press(e.x, e.y)})

def do_press(x, y);  Circle.new($canvas, x, y); end
Tk.mainloop()
```


Miscellaneous

JRuby

My first practical Ruby program

JRuby

"JRuby is an 100% pure-Java implementation of the Ruby programming language."
—Home page at jruby.codehaus.org

Here's a bash script, jruby, that runs it:

```
java -jar c:/dnload/jruby-0.9.0/lib/jruby.jar $*
```

Usage:

```
% time jruby mtimes.rb mtimes.1 the
user    0m0.061s
...

% time ruby mtimes.rb mtimes.1 the
user    0m0.015s
...
```

So what's the big deal?

JRuby, continued

```
require 'java'   # This is swing2.rb from the JRuby samples.
include_class "java.awt.event.ActionListener"
include_class ["JButton", "JFrame", "JLabel", "JOptionPane"].
          map {|e| "javax.swing." + e}

frame = JFrame.new("Hello Swing")
button = JButton.new("Klick Me!")

class ClickAction < ActionListener
      def actionPerformed(evt)
         JOptionPane.showMessageDialog(nil,
            "<html>Hello from <b><u>JRuby</u></b>.<br>" +
            "Button '#{evt.getActionCommand()}' clicked.")
      end
end
button.addActionListener(ClickAction.new)

frame.getContentPane().add(button) # Add the button to the frame

frame.setDefaultCloseOperation(JFrame::EXIT_ON_CLOSE) # Show frame
frame.pack(); frame.setVisible(true)
```


My first practical Ruby program

September 3, 2006:

```
n = 1
d = Date.new(2006, 8, 22)
incs = [2,5]
pos = 0
while d < Date.new(2006, 12, 6)
   if d != Date.new(2006, 11, 23)
      printf("%s %s, #%2d\n",
         if d.cwday() == 2: "T"; else "H";end, d.strftime("%m/%d/%y"), n)
      n += 1
   end
   d += incs[pos % 2]
   pos += 1
end
```

Output:

```
T 08/22/06, # 1
H 08/24/06, # 2
T 08/29/06, # 3
...
```