CSE 413 Autumn 2008

Ruby Classes, Modules & Mixins

Organizing Large(r) Programs

- Issues
  - Need to divide code into manageable pieces
  - Want to take advantage of reusable chunks of code (libraries, classes, etc.)
- Strategy: Split code into separate files
  - Typically, one or more classes per file
  - But what if the parts don't really form a class?

Namespaces & Modules

- Idea: Want to break larger programs into pieces where names can be reused independently
  - Avoids clashes combining libraries written by different organizations or at different times
- Ruby solution: modules
  - Separate source files that define name spaces, but not necessarily classes

Example (from Programming Ruby)

```
module Trig
  PI = 3.14
  def Trig.sin(x)
    # …
  end
  def Trig.cos(x)
    # …
  end
end
```

```
module Moral
  VERY_BAD = 0
  BAD          = 1
  def Moral.sin(badness)
    # …
  end
end
```

Using Modules

```
# …
require ‘trig’
require ‘moral’
y = Trig.sin(Trig::PI/4)
penance = Moral.sin(
        Moral::VERY_BAD)
# …
```

- Key point: Each module defines a namespace
  - No clashes with same names in other modules
- Module methods are a lot like class methods

Mixins

- Modules can be used to add behavior to classes – *mixins*
  - Define instance methods and data in module
  - “include” the module in a class – incorporates the module definitions into the class
    - Now the class has its original behavior plus whatever was added in the mixin
  - Provides most of the capabilities of multiple inheritance and/or Java interfaces

Example

```
module Debug
  def trace
    # …
  end
end
class Something
  include debug
  # …
end
```

```
class SomethingElse
  include debug
  # …
  end
```

- Both classes have the trace method defined, and it can interact with other methods and data in the class

Exploiting Mixins – Comparable

- The real power of this is when mixins build on or interact with code in the classes that use them
- Example: library mixin: Comparable
  - Class must define operator <=>
    - (a <=> b returns -1, 0, +1 if a<b, a==b, a>b)
  - Comparable uses <=> to define <, <=, ==, >=, >, and between? for that class

Another example – Enumerable

- Container/collection class provides an each method to call a block for each item in the collection
- Enumerable module builds many mapping-like operations on top of this
  - map, include?, find_all, …
  - If items in the collection implement <=> you also get sort, min, max, …