Refinements

Due to Ruby's open classes you can redefine or add functionality to existing classes. This is called a "monkey patch". Unfortunately the scope of such changes is global. All users of the monkey-patched class see the same changes. This can cause unintended side-effects or breakage of programs.

Refinements are designed to reduce the impact of monkey patching on other users of the monkey-patched class. Refinements provide a way to extend a class locally.

Here is a basic refinement:

```
class C
  def foo
    puts "C#foo"
  end
end

module M
  refine C do
    def foo
      puts "C#foo in M"
    end
  end
end
```

First, a class C is defined. Next a refinement for C is created using Module#refine. Refinements only modify classes, not modules so the argument must be a class.

Module#refine creates an anonymous module that contains the changes or refinements to the class (C in the example). self in the refine block is this anonymous module similar to Module#module_eval.

Activate the refinement with using:

```
using M

c = C.new

c.foo # prints "C#foo in M"
```

Scope

You may only activate refinements at top-level, not inside any class, module or method scope. You may activate refinements in a string passed to Kernel#eval that is evaluated at top-level. Refinements are active until the end of the file or the end of the eval string, respectively.

Refinements are lexical in scope. When control is transferred outside the scope the refinement is deactivated. This means that if you require or load a file or call a method that is defined outside the current scope the refinement will be deactivated:

```
class C
end

module M
  refine C do
    def foo
      puts "C#foo in M"
    end
  end
end

def call_foo(x)
  x.foo
end

using M

x = C.new
x.foo       # prints "C#foo in M"
call_foo(x) #=> raises NoMethodError
```

If a method is defined in a scope where a refinement is active the refinement will be active when the method is called. This example spans multiple files:

c.rb:

```
class C
end
```

m.rb:

```
require "c"

module M
  refine C do
    def foo
      puts "C#foo in M"
    end
  end
end
```

m_user.rb:

```
require "m"

using M

class MUser
  def call_foo(x)
    x.foo
  end
end
```

main.rb:

```
require "m_user"

x = C.new
m_user = MUser.new
m_user.call_foo(x) # prints "C#foo in M"
x.foo              #=> raises NoMethodError
```

Since the refinement M is active in m_user.rb where MUser#call_foo is defined it is also active when main.rb calls call_foo.

Since using is a method, refinements are only active when it is called. Here are examples of where a refinement M is and is not active.

In a file:

```
# not activated here
using M
# activated here
class Foo
  # activated here
  def foo
    # activated here
  end
  # activated here
end
# activated here
```

In eval:

```
# not activated here
```

```
eval <<EOF
  # not activated here
  using M
  # activated here
EOF
# not activated here
```

When not evaluated:

```
# not activated here
if false
  using M
end
# not activated here
```

When defining multiple refinements in the same module, inside a refine block all refinements from the same module are active when a refined method is called:

```
module ToJSON
  refine Integer do
    def to_json
      to_s
    end
  end

  refine Array do
    def to_json
      "[" + map { |i| i.to_json }.join(",") + "]"
    end
  end

  refine Hash do
    def to_json
      "{" + map { |k, v| k.to_s.dump + ":" + v.to_json }.join(",") + "}"
    end
  end
end

using ToJSON

p [{1=>2}, {3=>4}].to_json # prints "[{\"1\":2},{\"3\":4}]"
```

You may also activate refinements in a class or module definition, in which case the refinements are activated from the point where using is called to the end of the class or module definition:

```
# not activated here
class Foo
  # not activated here
  using M
  # activated here
  def foo
    # activated here
  end
  # activated here
end
# not activated here
```

Note that the refinements in M are not activated automatically even if the class Foo is reopened later.

Method Lookup

When looking up a method for an instance of class C Ruby checks:

- If refinements are active for C, in the reverse order they were activated:

  ```
  * The prepended modules from the refinement for C
  * The refinement for C
  * The included modules from the refinement for C
  ```

- The prepended modules of C
- C
- The included modules of C

If no method was found at any point this repeats with the superclass of C.

Note that methods in a subclass have priority over refinements in a superclass. For example, if the method / is defined in a refinement for Integer 1 / 2 invokes the original Fixnum#/ because Fixnum is a subclass of Integer and is searched before the refinements for the superclass Integer.

If a method foo is defined on Integer in a refinement, 1.foo invokes that method since foo does not exist on Fixnum.

super

When super is invoked method lookup checks:

The included modules of the current class. Note that the current class may be a refinement. If the current class is a refinement, the method lookup proceeds as in the Method Lookup section above. If the current class has a direct superclass, the method proceeds as in the Method Lookup section above using the superclass. Note that super in a method of a refinement invokes the method in the refined class even if there is another refinement

which has been activated in the same context.

Indirect Method Calls

When using indirect method access such as Kernel#send, Kernel#method or Kernel#respond_to? refinements are not honored for the caller context during method lookup.

This behavior may be changed in the future.

Refinements and module inclusion

Refinements are inherited by module inclusion. That is, using activates all refinements in the ancestors of the specified module. Refinements in a descendant have priority over refinements in an ancestor.

Further Reading

See bugs.ruby-lang.org/projects/ruby-trunk/wiki/RefinementsSpec for the current specification for implementing refinements. The specification also contains more details.