Programming Languages, Part C

Status	Completed

Lecture 8:

01/24/24

Introduction to Ruby

- Ruby is a pure object oriented language (OOP), meaning all values are objects.
- Ruby is class-based, meaning every object has a class that determines its behavior.
- Ruby is a *very* dynamically typed programming language.
- Ruby is a modern "scripting language" popularly used for building server-side web applications.

```
# Use "ruby" to run a ruby program
$ ruby "program.rb"

# Type "irb" (Interactive Ruby) to run the Ruby REPL
$ irb

# Use "load" to run a ruby program in the REPL
$ load "program.rb"

# Type "quit" to exit the REPL
$ quit
```

Classes and objects

- There are rules that guide one's understanding of the Ruby programming language:
    1. All values are reference to objects.
    2. Objects communicate through method/message calls.
        a. Methods are functions that belong to objects.
    3. Each object has its own private state.
    4. Every object is an instance of a class.
        a. An object's class determines the object's behavior.
            i. How the object handles method calls.
            ii. The class contains method definitions.
- New classes are defined with methods, and objects can be initialized from those class definitions.
    - New objects can be created with the syntax `ClassName.new`.
    - Object methods can be called with the syntax `object.method_name(args)`.

```
# Anatomy of a Ruby class

class <ClassName>
    def <method_name> <method_args>...
        <expression>
    end
    ...
end


# Example "Hello" class that contains a method that prints "Hello, World"

class Hello
    def hello_world
                # "puts" is a method that prints a string
        puts "Hello, World!"
    end
end


# Create an object "x" whose class is the class "Hello"
x = Hello.new


# Evaluate "x" to an object, and call its "hello_world" method
x.hello_world
```

- `self` refers to the object *itself*.
    - Or more specifically, the current object that the method is called on.
    - One can pass, return, or store the whole object with the syntax `self` by itself.

```
# The method m2 will call its object's method m1, resulting in x + y + 34
# It sends the same object an "m1" method

class A
    def m1
        34
    end
        # method arguments are surrounded by parentheses and separated by commas
    def m2 (x, y)
                # Syntactic sugar for "self.m1" is just "m1"
        x + y + self.m1
    end
end
```

- Methods can use local variables, and their scope is the method body.

- There is no need to declare methods or variables, as they can be assigned anywhere.
    - Variables are mutable, and are also allowed a the “top-level” global scope.
    - The contents of a variable are always a reference to an object.
- In Ruby, new lines affect semantics, but indentation does not.

<u>Object state</u>

- In Ruby, all objects have state.
    - The object’s state persists, as in it can frow and change from the time the object is created.
    - An object’s state is only directly accessible from an object’s methods.
    - An object’s state consist of instance variables, commonly known as fields.
        - In Ruby, fields are assigned with `@`.
        - If an object tries to access an undefined field, it is evaluated to a `nil` object.
- Creating a new object always returns a reference to a new object.
- Because mutation exists, one should be careful with aliasing.
    - Variable assignment creates an alias, meaning multiple variables hold references to the same object.
        - Any changes to an object through one variable affect that same object, for all the other alias variables to that object.

```
# Create two distinct Counter objects
a = Counter.new
b = Counter.new

# Create two aliases to the same Counter object
a = Counter.new
b = a
```

- The built-in method `initialize` is called when objects are created.
    - `initialize` will be called before `ClassName.new` returns the object being created.
    - Arguments can be passed with `new` to the `initialize` method.
- This is not, like it is in many programming languages, a constructor.
    - In Ruby, one is not required to initialize fields in `initialize`, and it is only convention.
- Class variables are state that is shared by the entire class.
    - These variables are shared, and only accessible, by instances of the class.
    - These are syntactically written with `@@`.
- Class constants are variables that syntactically start with a capital letter.
    - These class constants are allowed to, but should not, be mutated.
    - They are accessible outside the class with `ClassName::Constant`.
- Class methods, also known as static methods, are written with the prefix `self.`.

    - Class methods are called with `ClassName.method_name(args)`.
    - These methods are part of the class, not a particular object instance of it.

<u>Visibility</u>

- Hiding expressions and values is essential for modularity and abstraction.
- In Ruby, object state is always private, as in instance variables can only be accessed through an object's methods.
    - To make object state publicly visible, one should define getters and setters.

```
# Get the variable foo from the object
def get_foo
    @foo
end

# Set the variable foo for the object
def set_foo x
    @foo = x
end
```

- Ruby support syntactic sugar for these getters and setters.

```
# Get the variable foo from the object
def foo
    @foo
end

# Set the variable foo for the object
def foo= x
    @foo = x
end

# Call foo= two different ways
e.foo= y
# " =" is a method call to "="
e.foo = y
```

- Requiring private instance variables forces indirect interfacing with an object, without knowing about the actual implementation of the object.
    - This is essential for abstraction and modularity, as it allows class implementation to change without introducing bugs.
- There are three visibilities for Ruby methods:
    1. `private` methods are only available to the object itself.
        a. If a method is private, one can only call it with the shorthand `method_name(args)` and never with `self.method_name(args)`.
            i. This shorthand does not indicate what object the method is being called on.

2. `protected` methods are only available to objects of the same class or subclasses.
3. `public` methods are available to the whole program.

   - Methods are `public` by default.

01/25/24

Everything is an object

- Ruby is fully committed to object oriented program, meaning every value is a reference to an object.
  - This leads to simpler, smaller semantics.
  - The only operation on objects is calling methods on them.
    - All top-level methods are just added to the built-in `Object` class.
      - This `Object` class contains all built-in methods in Ruby.
    - All classes defined are subclasses of `Object`.
      - Therefore, all `Object` methods are passed down and able to be used by any object.
- Every class has an "undefined" method that is called when a method that doesn't exist is called for an object.
- `nil` is an object that is similar to the `null` value in Racket.
- All objects have built-in methods like `methods` and `class`.
  - `methods` returns an object's methods.
  - `class` returns an object's class
  - This process of querying information about an object during runtime is called reflection.
    - This an be useful in the REPL to explore what methods are available without having to consult the Ruby documentation.

Class definitions are dynamic

- Ruby programs can add, change, or replace methods while a program is running.
  - This can break abstractions and make programs difficult to analyze.
  - This helps re-enforce the rules of OOP.
    - Every object has a class, and a class determines its instances' behavior.
    - Since a class in an object itself, changes to a class will reflect those changes on it's object instances.

```
# Define a class Rational
class Rational
    def initialize x
        @value = x
    end
    ...
end

# Create an object with the class Rational
```

```
a = Rational.new

# Modify the class Rational
class Rational
    def double
        self + self
    end
end

# The object has not changed, but its class has
# Therefore, the object can now call the method "double"
a.double
```

- This behavior can even be used to modify Ruby's built-in classes.
  - Any new method defined at top-level is added to the `Object` class.
- Dynamic features like these can create interesting semantic questions that must be answered.
  - More dynamic features can lead to more questions that need to be answered, decreasing a programming language's performance.

Duck typing

> *"If it walks like a duck and quacks like a duck, it's a duck"*

- When one needs to pass an object into a method, in reality one usually only needs to pass in an object with the required certain properties (methods) of that object.
  - Duck typing means passing an object with the needed methods, instead of the entire original object itself.
    - It "walks" and "quacks" like the original method, and therefore for all practical purposes, it *is* the original method.
      - One advantage is that this allows for more code reuse, since methods an object receives is "all that matters".
      - One disadvantage is that abstraction is not as useful, since one cannot change the implementation of methods that assume duck typing.

```
# Example of "duck typing"

# The naive perspective:
# Method takes in a point and mirrors it along the x-axis

# The duck typing perspective:
# In reality it works for
# anything with x= and x (whose result has a * method), methods

def mirror_update point
    point.x = point.x * -1
end
```

01/26/24

Arrays

- The most commonly used data structure in Ruby is the `Array` class.
  - An array holds any number of other objects, and is indexed by number, starting at 0.
  - Similar to Python, Ruby interprets negative numbers as counting from the end of an array.
  - Array size is not fixed, and elements can be dynamically created or removed during runtime.

  ```
  # Initialize an array object
  a = [1, 2, 7, "Hello, World!", -1]
  # Initialize an array of length 7, where every element is nil
  b = Array.new(7)
  # Initialize an array of length 10, where every element is "hi"
  c = Array.new(10) { "hi" }

  # Bind the "i"th index of "a" to "d"
  d = a[i]

  # Set the "i"th index of "a" to "e"
  a[i] = e
  ```

- These arrays are much more flexible and dynamic (but less efficient) than in other programming languages.
- Arrays can be used as stacks, since they have the operations `push` and `pop`.
  - `push` adds an element to the end of an array.
  - `pop` removes and gets the last element of an array.
  - `unshift` adds an element to the beginning of an array.
  - `shift` removes and gets the first element of an array.
- Arrays, just like any other objects, can be aliased.
- Arrays can be sliced.

```
# Create a new array
a = [1, 2, 3, 4, 5, 6]

# Assign a slice of the array to b,
# starting at element 2 and giving 3 elements
# [3, 4, 5]
b = a[2, 3]
```

Blocks

- Blocks are a feature in Ruby that allow any easy way to pass anonymous functions to methods.
  - Blocks can take zero or more arguments.
  - Blocks use lexical scope, where the block body uses the environment where it was defined.

- Syntactically, one can use `{ ... }` or `do ... end` to enclose a block.

```
# Anatomy of a block
{ | args... | expression }
# Alternatively replace { } with do end
do | args.. | expression end

# Run a block 3 times
# Prints "Hello, World!" 3 times
3.times { puts "Hello, World!" }

# Call the block once for each element of an array
# Print the square of each element in the array
[1, 2, 3, 4].each { |x| puts (x * x) }

# Create an array of the first 5 multiples of 4, starting at 4
Array.new(5) { |i| 4 * (i + 1) }
```

- One can pass zero or one clocks with *any* message (method).
- Blocks can be used to apply higher-order functions to arrays.
    - Because blocks are so useful for applying higher-order functions to arrays, loops are very rarely used.

Using blocks

- When a callee uses a block argument, there is no name for that block argument.
- When defining the use of blocks inside a method, the keyword `yield(args)` is used to denote where a block is used, and what arguments the block uses.
    - If a caller tries run a method that contains `yield` without a block, an error will be raised.
    - `block_given?` evaluates to `true` if a block is passed by a caller, and `false` otherwise.

```
# Define the method silly which takes a block
def math_op
    yield(4, 5) + yield(100, 100)
end

# Call the method silly with a block
# Yield evaluates its arguments using the block it is passed
# Result is 103 = [(2 * 4) - 5] + [(100 * 2) - 100]
math_op { |a, b| (2 * a) - b }
```

Procs

- Procs are similar to blocks, excepts they are actual objects with the abilities of function closures.
    - Blocks are “second-class” expressions, as their only function is to be yielded too.
        - They cannot be returned, passed, stored, etc.

- Instances of the `Proc` class are “first-class” expressions.
    - First class indicates that an expressions can be the result of a computation, returned, stored, and so on.
- Using the method `lambda` on an `Object` object takes a block and returns its corresponding `Proc`.

```
# Trying to create an array of closures is not possible with blocks
# A block cannot pass in another block

# This would return an error
c = a.map { |i| { |y| i > y } }


# Trying to create an array of closures is possible with Procs

# The lambda method of Object is used to turn
# the block into an instance of Proc

# Because lambda is a method of the Object class, and all top-level functions
# are part of the Object class, there is no need to write Object.lambda
c = a.map { |i| (lambda { |y| i > y }) }
```

- First-class expressions make closures more powerful than blocks.
    - In simple cases, blocks can be more convenient to use.

Hashes and ranges

- Hashes and ranges are two standard classes in Ruby that are commonly used in Ruby programs.
    - Hashes are arrays, with mappings from keys to values.
        - The keys can be any type of object, and as a result there is no natural ordering of numeric indices that would be found in an array.

```
# Create an empty hash
hash = {}

# Assign the value "Nate" to the key "First Name"
# {"First Name"=>"Nate"}
hash["First Name"] = "Nate"

# Assign the value "Levine" to the key "Last Name"
# {"First Name"=>"Nate", "Last Name"=>"Levine"}
hash["Last Name"] = "Levine"

# Return an array of the hash's keys
# ["First Name", "Last Name"]
hash.keys

# Return an array of the hash's values
```

```
# ["Nate", "Levine"]
hash.values
```

- Ranges are arrays of contiguous numbers that are more efficiently represented than arrays.

```
# Create and range object from 1 to 100
range = (1..100)

# Add up all the numbers from 1 to 100 (5050) using the method inject
range.inject { |acc, elt| acc + elt }
```

- It is generally good style to:
  - Use ranges instead of arrays instead of numbers where one can.
  - Use hashes instead of arrays when non-numeric keys are better suited for representing data.

Subclassing

- Sub-classing is a concept essential to object oriented programming.
- A class definition always has a super-class/parent class.
  - This super-class is `Object` by default.
  - A class inherits all the method definitions of its super-class.
  - A class can override method definitions as desired.
- Unlike Java/C#/C++, classes in Ruby cannot inherit fields, since all instance variables in Ruby are not part of class definitions.
- The `super` method is used inside a sub-class's method to call a method of the same name from the super-class, that has been overridden by the sub-class.

```
# The superclass Point
class Point
    def initialize(a, b)
        @x = a
        @y = b
    end

    def distFromOrigin
        Math.sqrt(@x * @x + @y * @y)
    end
end


# The subclass ColorPoint
# The < denotes it is a subclass of the class Point
# ColorPoint inherits the method "distFromOrigin" from Point

# ColorPoint's initialize method overrides Point's initialize method
```

```
# "super" is used to call the superclass' method of
# the same name as a helper function
class ColorPoint < Point
    def initialize(x, y, c="clear")
        super(x, y)
        @color = calls
    end
end


# Get the superclass of ColorPoint (Point)
ColorPoint.superclass
```

- Sub-classes *are the same classes* as their super-classes.
    - Instances of sub-classes *are not instances* of their super-classes.

```
# ColorPoint is a Point
# true
Point.is_a? ColorPoint


# ColorPoint and Point are Object
# true
ColorPoint.is_a? Object
# true
Point.is_a? Object


# ColorPoint is not an instance of Point
# false
ColorPoint.instance_of? Point
```

- Using methods like `is_a?` and `instance_of?` is usually not OOP style, as it disallows techniques like duck typing where the class of an object can be arbitrary.

Why use subclassing?

- Sub-classing is a useful programming feature, though it tends to be overused in OOP programming languages.
- Sub-classing removes the need to add new methods to a super-class.
    - Adding new methods to super-class can break modularity.
- Sub-classing removes the need to have the same methods in multiple similar classes.
    - Creating subclasses that inherit from super-classes reduces code reuse.
- An alternative to sub-classing is having an instance of a super-class that initialized inside a “sub-class”.
    - Unfortunately, this is less convenient code reuse
    - In addition, the sub-class would not be the same class as the super-class.

- This means that the sub-class cannot be used where a program expects the super-class.

```
# Instead of making ColorPoint a sub-class of Point,
# Initialize a Point object inside the ColorPoint and forward
# messages to it through ColorPoint
class ColorPoint
    def initialize(x, y, c="clear")
        @pt = Point.new(x, y)
        @color = c
    end
    def x
        @pt.x
    end
    def
        @pt.y
    end
end
```

Overriding and dynamic dispatch

- With the examples so far, one can argue that objects are not so different from closures.
  - Unlike closures, objects have multiple methods.
  - Unlike closures, objects have instance variables rather than an environment.
  - Object inheritance avoids helper functions or code copying.
- The big difference is that overriding can make a method defined in the super-class call a different method in the sub-class.
  - Overriding a method in a sub-class is sometimes necessary to maintain correct logic for that method.
- When one has a method that makes another call on the same object (when that method uses `self`), that `self` is the entire object.
  - If `self` is an instance of the sub-class, then the sub-class's methods are used, *not* the super-class's methods.

Method-lookup rules

- To understand dynamic dispatch fully, it is important to precisely understand the semantics of method look-up in OOP.
- Dynamic dispatch (also knowns as late binding or virtual methods) refers to the semantics that when calling `self` on a method `m1` in another method `m2` in class `c`, `m2` can resolve to a method `m2` defined in a sub-class of `c`.
  - This is the most unique characteristic of OOP that distinguishes it from closures or functions.
- All lookup rules in Ruby are defined in terms of `self`.
  - `self` maps to some "current" object.
  - Instance variables `@x` are looked up in the object bound to `self`.
  - Class variables `@@x` are looked up in the object bound to `self.class`.
  - For methods calls:

1. All expressions called with a method are evaluated to objects.
2. Let `C` be the class of the expression that the method wall called on.
    a. `C` is the receiver of the method.
3. If the method is defined in `C`, use that method.
    a. Otherwise, check the super-class of `C` for that method, and that super-class's super-class, and so on until the method is found or `Object` is reached.
        i. If the `Object` class is reached, a `method_missing` error message is raised by it.
    b. Whatever class the method is found in first, that method from that class is called.
4. The body of the method picked is evaluated.
    a. `self` is bound to the expression that receives the method.
        i. This is the key point that implements dynamic dispatch.

- A method body is evaluated in an environment where `self` refers to the object that the method is called on.

```
# obj.b evaluates to "Hello, World!"
# 1. When obj calls method "b", method "a" is called
# 2. The current object "obj" has class B
#  - Because class B contains no method "a",
#    "obj" searches its super-class A
# 3. method "c" is called in class A, and that method calls method "b"
# 4. Because the current object (the object that received method "a")
#    has class B, the method "c" in class B is called,
#    not the method "c" in class A
# 5. Method "c" in class B evaluates to "Hello, World!"
obj = B.new
obj.b

# Class A is a super-class of class B
class A
    def a
        self.c
    end
    def c
        42
    end
end

# Class B is a sub-class of class A
class B < A
    def b
        # Method "a" will evaluate using the object with class B as "self"
        self.a
    end
    def c
```

```
        "Hello, World!"
    end
end
```

- Although these semantics for methods are more complex than function closures, it does not mean they are inferior or superior to them.

Lecture 9:

01/31/24

OOP versus functional decomposition

- When comparing OOP and functional programming, one should consider how these two programming methods decompose problems into more manageable pieces.
  - Functional programming breaks problems down into functions that each perform their own operation.
  - OOP breaks programs down into classes that give behaviors to data.

	eval	toString	hasZero	...
Int				
Add				
Negate				
...				

- Functional programming essentially breaks down to applying operations to variants (types of data)
  - This application of operations to variants can be summarized in a table.
  - Each column is evaluated through a case expression, therefore the code is organized primarily by the columns (operators) of the “grid”.

```
(* Example of function-based organization in ML *)


(* The functions are organized by operations, and each function
   uses a case expression to evaluate different variants *)
fun toString e =
    case e of
        Int i => ...
    | Add (e1, e2) => ...
    | Negate e1 => ..
```

- OOP defines classes with one abstract method for each operation, and subclasses are defined for each variant.
  - Each row in the “grid” represents a class with one method implementation for each “grid” column.
  - This is essentially the opposite of functional programming, as in this case the code is organized primarily by the rows (classes) of the “grid”.

```
# Example of class-based organization in Ruby


# The classes are organized by variants, and each class has methods
# that operate on expressions
class Int < Exp
    def Int
        ...
    end

    def Add
        ...
    end

    def Negate
        ...
    end
end
```

- In a sense, functional programming and object oriented programming are doing the same thing in the exact opposite way.
  - The question “boils down to” whether one wants to organize their program by rows or by columns.

Adding operations or variants

- Planning for software extensibility can influence which programming style is used.
  - If a new variant will need to be added to a program, OOP style may be used to add another row to the “grid”.
  - If a new operation will need to be added to a program, functional style may be used to add another column to the “grid”.
- Making extensible software is often difficult yet valuable, as code can be reused more.
  - Some modern languages, such as Scala, attempt to support extensibility in both directions.

Binary methods with functional decomposition

- Often operations are defined over multiple variants.
- For example, one may want to redefine an `Add` operation to work over the variants `Int`, `String`, and `Rational`.
  - Addition is a binary method/operation, because it takes two variants to evaluate.
  - This operation needs to be redefined using another “grid”.
  - In functional implementation, this is achieved through a helper function.

```
(* Example of function-based organization in ML with a binary operation *)


(* Helper function which decides the evaluation of a binary method "Add" *)
fun add_values (v1, v2) =
    case (v1, v2) of
        (Int i, Int j) => ...
    | (Int i, String s) => ...
  | (Int i, Rational (i, j) => ...
        ...
    | (String s1, String s2) => ...
    | (String s, Rational (i, j)) => ...
    (* Else case covers the case where v1 or v2 are
         not Int, String, or Rational *)
    | _ => ...


fun toString e =
    case e of
        Int i => ...
    (* A helper function is used to decide the evaluation of Add *)
    | Add (e1, e2) => add_values (eval e1, eval e2)
    | Negate e1 => ..
    | String s => ...
    | Rational i/j => ...
```

	Int	String	Rational
Int			
String			
Rational			

Double dispatch

- In OOP, there are no case functions to take care of binary operations.
    - This means that one variant needs to call a binary method on another variant.
- When this happens, it is impossible to know the class of the variant `v` being passed into the binary method.
    - To solve this problem, a method must be called on `v` to tell it what variant of the receiver of the method `self` is, by passing `self` into that method.
    - This programming idiom is called double dispatch.

```
# The class Int has a method "add_values" which takes a variant v
# and adds itself (an Int) to v

# It does this by passing itself into a method v (which could be an Int,
# String, Rational, etc.), whose class has a method "addInt" which
# decides how it adds and Int to itself

class Int
    # First dispatch
    def add_values v
        v.addInt self
    end

    # Dispatches for when Int is the variant passed into an "add_values" method
    def addString v
        ...
    end
    def addNatural v
        ...
    end
    ...
end

class <Type of v>
    # Second dispatch
    def addInt v
        ...
    end

    # Dispatches for other variants
    def addString v
        ...
    end
    def addNatural v
        ...
    end
    ...
end
```

- Every class has the appropriate “cases” (not actually cases like in SML, but methods) to handle how they evaluate binary methods with both their class and another class.

Multi-methods

- An alternative to double dispatch is multiple dispatch/multi-methods.
    - For example, each class would have multiple methods of the same name, each one corresponding to an operation on a different class.

        - Each method indicates what instance of what class it takes as an argument.
    - However, interactions of multiple dispatch with subclassing can confuse an end user over which methods should be called for what situations.
    - This is not an appropriate idiom for Ruby, as Ruby:
        1. Places no restrictions of what arguments can be passed into a method.
        2. Does not allow multiple methods with the same name.

Multiple inheritance

- It is important to distinguish the relation between different classes with more specific terminology than just “sub-class” or “super-class”.
- Class hierarchies form a tree:
    - Each node is a class.
    - A sub/super-class is an immediate sub/super-class when they are separated by one level of inheritance.
        - Parents are immediate super-classes.
    - A sub/super-class is a transitive sub/super-class when they are separated by more than one level of inheritance.
- Multiple inheritance allows for a sub-class to have more than one super-class.
    - Languages like C++ allow this, where other languages like Ruby and Java do not.
    - With multiple inheritance, the class hierarchy is no longer a tree.

```
graph TD
  X --> V
    X --> W
    V --> Y
    W --> Y
```

- In the tree above, a diamond is formed when multiple paths show that X is a transitive superclass of Y.
    - This can lead to ambiguity in a program.
        - For example, if both V and W define a method `m`, which method `m` does Y inherit?

Mixins

- Ruby has a feature called mixins, which are an alternative to multiple inheritance.
    - Mixins are collections of methods.
    - Mixins are not classes, and therefore cannot be instantiated.
- Ruby allows for a sub-class to have only one superclass, but multiple mixins.
    - Semantically, mixins make their methods part of the class.
    - Mixins are more powerful than helper methods, as mixin methods can access methods on `self`.

    - Mixins allow classes to gain many methods for a small amount of work, without the complexity of multiple inheritance.
- Ruby mixins are defined using the `module` keyword.
- Mixins can be included in classes using the `include` keyword.

```
# Example of how the mixin Doubler can be included in Pt
# to give Pt a "double" method


# Mixin "Doubler"
module Doubler
    def double
        # We assume that any class that includes "Doubler" will have a "+" method
        # When "double" is called in Pt, Pt will call the "+" method.
        self + self
    end
end


class Pt
    attr_accessor :x, :y
    # Now if Pt.double is called, a new point with double the original
    # point's x and y coordinates will be returned
    include Doubler
    def + other
        ans = Pt.new
        ans.x = self.x + other.x
        ans.y = self.y + other.y
    end
end

# Return a point with x = 6, y = 8
Pt.double(Pt.new(3, 4))
```

- There are two popular mixins in Ruby, those being:
    1. `Comparable`, which defines `<`, `>`, `==`, `!=`, `>=`, and `<=` in terms of `<=>`.
        a. `<=>` is sometimes called the “spaceship operator”.
    2. `Enumerable`, which defines many iterators such as `map`, `find`, and so on.

<u>Interfaces</u>

- Another alternative to multiple inheritance and mixins is with Java/C# style interfaces.
    - Interfaces rely on static typing, and are therefore not semantically possible in Ruby.
- Static typing works slightly differently for OOP than it does for functional programming.
    - Sound typing for OOP prevents “method missing” errors.

        - The type checker will check if objects possess the methods called on them.
    - Each class has a type, and each method has argument and result types.
        - Any subclass is also a subtype.
        - A subtype can be used anywhere a supertype is allowed.
- Interfaces are types, not classes.
    - Interfaces do not contain method definitions, only method signatures.
    - Interfaces are not instantiable.
- A class can implement any number of interfaces.
    - For a class to type check, each method in the interface must have the correct type.
    - If a class type-checks, it is a subtype of the interface.
        - If class A implements interface I, A is a subtype (not a subclass) of I.
- Interfaces are primarily used for making a type system more flexible, so a callee can call certain methods regardless of their class.
- Because interfaces allow what is essentially the default in many dynamically typed programming languages, they have very little use in those languages.

Abstract methods

- One should consider how required overriding is implemented in a statically typed programming language.
    - Required overriding refers to when a super-class requires its sub-classes to override one or more of its methods.
- Often a class is written in such a way that it expects its sub-classes to override some methods.
    - The purpose of the super-class in this case is to abstract common functionality.
    - Because the super-class is "abstract", it does not make sense to make an instance of the super-class.
- Java/C#/C++ let super-classes give a type signature for methods that sub-classes need to provide.
    - These are called abstract methods in Java, and pure virtual methods in C#/C++.
    - This also disallows instances of the super-class from being created.
- Abstract methods do not make these programming languages more powerful.
    - In fact, their point is to limit the functionality of classes, not expand it.

▼ Lecture 10:

02/02/24

Subtyping from the beginning

- This lecture will only contain pseudocode, as none of the three programming languages studied in this course have subtyping as a feature.
- Most core subtyping ideas can be covered by considering records with mutable fields.
    - Therefore the pseudocode will be most similar to SML, as that programming languages has records with *immutable* fields.

- Pseudocode syntax is as follows:
  - Each record field `f` contains an expression `e` that evaluates to `v` and has a type `t`.

```
# Record creation (field = expression)
# evaluate e's, make a record
{f1=e1, f2=e2, ..., fn=en}

# Record field accessing
# Evaluate e to a record v with a field f
e.f

# Record field updating
# Evaluate e1 to a record v1, e2 to a record v2, change v1's field to v2
# e1 and e2 must have the same type t
e1.f = e2


# Each field f has a type t
# e.f has type t
{f1:t1, f2:t2, ..., fn:tn}


# Example value
val point : {x:real, y:real} = {x=3.0, y=4.0}
# Access x and y from point
point.x
point.y
```

- If an expression has type `{f1:t1, f2:t2, ..., fn:tn}`, then subtyping would allow it to have some fields removed.

```
# This call to "dist" would only work with subtyping,
# as without it color_pt contains too many fields

fun dist (p:{x:real, y:real}) =
    ...

val color_pt : {x:real, y:real, color:string} = ...
val _ = dist(color_pt)
```

The subtype relation

- Subtyping rules are defined exactly, in a way that doesn't change any typing rules that have already been encountered, while also providing more flexibility to a type system.
  - Programming languages already have a lot of typing rules that would be too complicated to change.
- This can be done by adding two rules to a language:
  1. The subtyping rule, `t1 <: t2`, stating that `t1` is a subtype of `t2`.

2. One new typing rule that uses subtyping.
    a. If `e` has type `t1` and `t1 <: t2`, then `e` also has type `t2`.

- These rules are not a matter of opinion, since any variation from these subtyping rules may result in an unsound programming language.
    - If a programming language is sound before adding subtyping rules, it should be kept that way.
    - For example, one may be able to access record fields that do not exist.
    - These rules achieve the principle of substitutability, meaning that if `t1 <: t2`, then any value `t1` is usable in any way `t2` is.
- These rules follow from the substitutability principle:
    1. Width subtyping: A wider record can be a subtype of a slimmer record, as long as the slimmer record has the same types as the wider record.
    2. Permutation subtyping: A supertype can have the same fields as a subtype with the same types, just in a different order.
    3. Transitivity: If `t1 <: t2` and `t2 <: t3`, then `t1 <: t3`.
    4. Reflexivity: Every type is a subtype of itself, so `t <: t`.

Depth subtyping

- Depth subtyping suggests that if `ta <: tb`, then a record with a field `tb` can be subtyped with a new record where that field now has type `ta`.

```
# What a depth subtyping rule may look like
if ta <: tb
then {f1:t1, ..., f:ta, ..., fn:tn} <: {f1:t1, ..., f:tb, ..., fn:tn}
```

- Unfortunately, depth subtyping is unsound, as it allows programs to access missing record fields.
    - For example, a function could mutate a record to take a subtype of that record with fewer fields.
        - Now when trying to access one of the fields that has been removed, the program breaks.

```
# sphere is defined with a z field
val sphere:{center:{x:real, y:real, z:real}, r:real} =
    {center={x=3.0, y=4.0, z=5.0}, r=1.0}

# get2dCenter removes the z field from the sphere's
# center's z field through depth subtyping
fun get2dCenter (c:{center:{x:real, y:real}, r:real}) =
    c.center = {x=x, y=y}

val sphere2dCenter = get2dCenter(sphere)

# sphere's center's z field cannot be accessed, as it has no z field anymore
val sphere_z = sphere.center.z
```

- This a good example of how when one creates a rule for a programming language, it cannot just be evaluated it on the programs that one wants to work.
    - One also has to make sure that the programs one *does not want* are still not allowed.
- For depth subtyping to work, record fields would need to be immutable.

02/03/24

Function subtyping

- Subtyping already allows for:
    - Function callers to pass in a subtype of a parameter as an argument.
    - Functions to return the subtype of their resulting value(s).
- One should consider how a programming language defines when a function itself is a subtype of another function.
- Following from the two base subtyping rules, one can derive the rule that if `ta <: tb`, then `t->ta <: t->tb`.
    - This means that the functions are covariant in their return type.
- This *does not* mean that if `ta <: tb`, then `ta->t <: tb->t`.
    - This rule breaks soundness.
- However, the opposite rule that if `tb <: ta`, then `ta->t <: tb->t` is sound.
    - This means that the function arguments are contravariant.
- Therefore if `t3 <: t1` and `t2 <: t4`, then `t1->t2 <: t3->t4`.
    - Function subtyping is contravariant in its arguments, and covariant in its results.

Subtyping for OOP

- Our understanding of subtyping for records are functions can be used to understand subtyping for class-based OOP, and how a static type checker works for OOP.
    - Class names are types, and therefore sub-classes are sub-types.
- An instance of a sub-class should be usable anywhere in place of an instance of its super-class.
- In a sense, objects are essentially records that hold *mutable fields*, and *immutable methods* that have access to `self`.
- It is important to understand the distinction between classes and types.
    - A class defines and object's behavior.
        - Subclassing inherits behaviors and changes that behavior.
    - A type describes an object's methods' argument and return types.
- `self` is special in the fact that it is an exception to the contravariant argument rule for methods/functions.
    - `self` can be used as a covariant argument for class methods.

Bounded polymorphism

- Generics are polymorphic type variables, like `'a` in SML.
- While generics and subtyping are applicable for separate use cases, programming languages can have both features.

    - These two features can be combined to create bounded polymorphism, where any type `'b` can be a subtype of `'a`.
- Subtyping alone cannot be used for certain functions.

```
# This would not be allowed, since List<ColorPoint>
# is not a subtype of List<Point>
# This is due to the fact that depth subtyping is not allowed,
# as in the elements of one list being subtypes of the elements of
# the other list is not considered

List<Point> inCircle (List<Point> pts) =
    ...

# Passing in a list of objects with the class Point is allowed
inCircle(pts)

# Passing in a list of objects with the class ColorPoint is not allowed
inCircle(color_pts)
```

- In addition, generics alone cannot be used for certain functions.

```
# This would not be allowed, since List<T> can pass in
# anything, not just objects with x and y fields

List<T> inCircle (<T> pts) =
    ...
```

- Therefore, it is useful to create a generic that is a subtype of a specific supertype.

```
# This is allowed as long as T is a subtype of Point, or is type Point

# T <: Point
List<T> inCircle (List<T> pts) =
    ...
```

Wrap-up

- This course has taught the skills to:
    - Distinguish functional programming and OOP.
    - Learn new programming languages quickly.
    - Master specific programming language concepts.
    - Evaluate programming languages and their constructs.
- Software is all about taking a few ideas (language constructs), and combining them with human ingenuity to create complex systems that run the world.