Julia for Scientific Programming

Seminar 2

Marcus Thelander Andrén & Mattias Fält

Dept. of Automatic Control
Lund University

Today's Goal

1. Scope and modules (briefly)
2. Handling objects
    - Types
    - Methods
    - Constructors
    - Conversion
    - Promotion
3. Iteration and iterable collections

Discussion of previous assignment

Scope of Variables

- Where a variable name is visible in the code
- Constructs introducing scope block:
  - `function` bodies
  - `while` loops
  - `for` loops
  - `try` blocks
  - `catch` blocks
  - `let` blocks
  - `type` blocks
- Note: `begin` and `if` blocks don't introduce scope blocks
- When a variable is introduced into a scope, it is also inherited by all inner scopes unless one of those inner scopes explicitly overrides it.

Scope of Variables

- Function scope inherits variables from where the function was defined and NOT where it was called (*lexical scoping*)
- Example:

```
function foo()
  x
end

function bar()
  x = 1
  foo()
end

x = 2

julia> bar()
2
```

Modules

- Separate global variable workspaces
- Syntax:

```
module Name
...
end
```

- Use names from other modules with `import`, `importall` and `using`
- Specify which names are public with `export`

Modules

- Example:

```
module MyModule

export x, y

x() = "x"
y() = "y"
p() = "p"

end
```

- `x` and `y` exported, not `p`

Modules

- Different ways to load `MyModule`:
- `using MyModule` → `x`,`y`, `MyModule.x`, `MyModule.y` and `MyModule.p`
- `import MyModule` → `MyModule.x`, `MyModule.y` and `MyModule.p`
- `importall MyModules` → `x` and `y`

Types in Julia

- Julia uses dynamic typing
- However, still possible to explicitly declare types (to increase performance and robustness).
- All objects in Julia have a type, queried with `typeof()`
- Two sorts of types:
  1. Abstract types (can not be instantiated)
  2. Concrete types (can be instantiated)
- Any supertype must be an abstract type. All concrete types are final in Julia
- Abstract types are used to categorize the concrete types

Types in Julia

- Example of type hierarchy:

```
julia> x = 42;

julia> typeof(x)
Int64

julia> Int64 <: Real
true

julia> y = 42.0;

julia> typeof(y)
Float64

julia> Float64 <: Real
true
```

Defining Types

- Abstract type:

```
abstract <<name>>
abstract <<name>> <: <<supertype>>
```

- Concrete type:

```
type <<name>> end
type <<name>> <: <<supertype>> end
```

- Default supertype is `Any`

Defining Types

- Example:

```
abstract Feline

type Jaguar <: Feline
  age
  location
end

julia> fluffy = Jaguar(3, "South America")
```

- age and location are attributes. Must be specified in default constructor.

Type Declarations

- The `::` operator (read as "is an instance of")
- Two ways to use it:
  1. To assert a variable's type, useful when debugging:
     ```
     julia> (1+2)::FloatingPoint
     ERROR: type: typeassert: expected FloatingPoint,
     got Int64
     ```
  2. When declaring a variable, providing compiler with more info:
     ```
     x::Int8 = 10
     ```
- NOTE: Cannot declare variable type in global scope

Type Declarations

- Example cont'd:

Current Jaguar takes arguments of any type:

```
julia> strange = Jaguar("Three", "South America")
```

Solve with type declaration:

```
type Jaguar <: Feline
  age::Int64
  location::String
end
```

Trying to instantiate Jaguar like above will then throw an error

Type Parameters

- Consider:

```
julia> typeof([10, 20, 30])
Array{Int64,1}
```

- `Array` is a predefined type, and `Int64` and `1` are type parameters.
- Types with parameters are an indexed family of types, one type for each set of parameters

Defining Parametric Types

- Introduce type parameters for your type with T:

```
type FooBar{T}
  foo::T
  bar::T
end

julia> fb = FooBar(1.0, 2.0)
FooBar{Float64}(1.0,2.0)

julia> fb = FooBar(1, 2)
FooBar{Int64}(1,2)
```

- If arguments should be subtypes to e.g Number, you can write `type FooBar{T <: Number}`

Methods

- A method is a definition of one possible behaviour of a function
- A function may have several methods
- Dispatch looks for the method which is most specific
- Define a new method simply by defining the same function with new arguments

Methods

- Example:

```
julia> f(x::Float64, y::Float64) = 2x + y;

julia> f(2.0, 3.0)
7.0

julia> f(2.0, 3)
ERROR: `f` has no method matching
f(::Float64, ::Int64)

julia> f(x::Number, y::Number) = 2x - y;

julia> f(2.0, 3)
1.0
```

Parametric Methods

- Method definitions may have type parameters
- Example:

```
julia> same_type{T}(x::T, y::T) = true;

julia> same_type(x,y) = false;
```

Constructors

Empty type automatically creates constructor

```
type f{T}
  x::T
  y::T
end
```

```
> methods(f)
f{T}(x::T,y::T)
```

Which is equivalent to

```
type f{T}
  x::T
  y::T
  f(x,y) = new(x,y)
end
f{T}(x::T, y::T) = f{T}(x,y)
```

```
> methods(f{Int})
f(x,y)

> f{Int}(1,2.0)
```

Observe the different meanings of "T"!

Constructors

Empty type automatically creates constructor

```
type f{T}
  x::T
  y::T
end
```

```
> methods(f)
f{T}(x::T,y::T)
```

Which is equivalent to

```
type f{T}
  x::T
  y::T
  f(x,y) = new(x,y)
end
f{T}(x::T, y::T) = f{T}(x,y)
```

```
> methods(f{Int})
f(x,y)

> f{Int}(1,2.0)

f{Int}(1,2)
```

Observe the different meanings of "T"!

Defining new Constructors

```
type f{T}
    foo::T
    bar::T
end

julia> f(1, 2.0)
ERROR: 'f{T}' has no method matching
f{T}(::Int64,::Float64)

julia> f{Float64}(1, 2.0)
f{Float64}(1.0,2.0)
```

We can declare new constructor instead:

```
julia> f(x::Int64, y::Float64) = f(convert(Float64,x),y)
julia> f(1,1.0)
f{Float64}(1.0,2.0)
```

Promotion

- Instead of declaring all possible conversions, we can use `promote`

```
> f(x::Real, y::Real) = f{Real}(promote(x,y)..
> f(1,2.0)
f{Real}(1.0,2.0)
```

- What happens if we define this instead?

```
> f(x::Real, y::Real) = f(promote(x,y)...)
> f(1,2.0)
```

Promotion

- Instead of declaring all possible conversions, we can use `promote`

```
> f(x::Real, y::Real) = f{Real}(promote(x,y)..
> f(1,2.0)
f{Real}(1.0,2.0)
```

- What happens if we define this instead?

```
> f(x::Real, y::Real) = f(promote(x,y)...)
> f(1,2.0)
```

ERROR: stack overflow
The new function is more specific than default constructor

```
f{T}(x::T, y::T)
```

Iterators

- Lists, Arrays, Sets etc. are iterable collections

```
> a = Set(1:10)∪Set(20:30)
> 15∈a
false
```

Iterators

- Lists, Arrays, Sets etc. are iterable collections

```
> a = Set(1:10)∪Set(20:30)
> 15∈a
false

x = 0;
> for val in a
    x = x + val
  end
330
```

Defining an iterator

```
for i = I
  # body
end
```

⇔

```
state = start(I)
while !done(I, state)
  (i, state) = next(I, state)
  # body
end
```

Defining an iterator

```
for i = I
  # body
end
```

⇔

```
state = start(I)
while !done(I, state)
  (i, state) = next(I, state)
  # body
end
```

```
type MyArray
  data::AbstractArray
end
Base.start(a::MyArray) = 1
Base.done(a::MyArray,state) = length(a.data) == state-1
Base.next(a::MyArray,state) = a.data[state], state+1

x = MyArray([1,2,3])
for i in x
  println(i)
end
```

Assignment for next seminar

Implement a linked list type:

- Should consist of nodes, defined as a specific type (e.g `type Node`)
- All nodes in the same list should contain the same type of value
- Should be able to iterate over the list (e.g `for node in list...`)
- Should be able to call functions to add node at beginning and end of list