

# Ruby Inheritance

CSCI400

05 September 2017

# Color Key

- Clickable URL link
- Write down an answer to this for class participation
- Just a comment – don't confuse with yellow

# Class Participation

Get out a piece of paper, we'll be tracing some code today.

Code files (follow along with the slides)

# Basics

# Extending Class Behavior

- Can create subclasses (**inheritance**)
- May include/inherit methods from modules (**mix-ins**)
- Clients of class may also extend the class
  - Open classes
  - Adding singleton method to individual object

# Inheritance

- Every class has a single immediate superclass
  - `class Student < Person`
  - `Object` is the default superclass
- `BasicObject` is the parent of `Object`
  - Few methods, useful for wrapper classes
  - Can create completely separate hierarchy
    - e.g. `BasicObject` is not a superclass of `Kernel`

# Inheritance and Instance Variables

# Inheritance and Instance Variables

- Instance variables (IVs)
  - Are defined *within class methods*
  - Are created upon assignment (`@age = 0`)
  - Every Ruby object has them
- → Instance variables have nothing to do with inheritance
- However...
  - If all IVs defined in `initialize`, inheritance appears to work as expected



## Example: Variable 'Inherited'

```
class Person
  def initialize(name)
    @name = name
    puts "initializing"
  end
end

class Student < Person
  def to_s
    puts "Name: #{@name}"
  end
end

s = Student.new("Cyndi")
puts s
```

See: `ruby_inheritance-1a.rb`

## Example: Variable 'Inherited'

```
class Person
  def initialize(name)
    @name = name
    puts "initializing"
  end
end

class Student < Person
  def to_s
    puts "Name: #{@name}"
  end
end

s = Student.new("Cyndi")
puts s
```

See: `ruby_inheritance-1a.rb`

- Technically, `@name` *not inherited*
  - But `initialize` *is* called  
→ creates `@name`
  - *Appears* that variable is inherited
- An instance variable created in a parent method that the child does not call will *not* exist

## Example: Variable Not 'Inherited'

```
class Person
  def initialize(name)
    @name = name
    puts "initializing"
  end
  def setupEmail(email)
    @email = email
  end
  def sendEmail()
    puts "Emailing #{@email}"
  end
end
```

See: `ruby_inheritance-1b.rb`

## Example: Variable Not 'Inherited'

```
class Person
  def initialize(name)
    @name = name
    puts "initializing"
  end
  def setupEmail(email)
    @email = email
  end
  def sendEmail()
    puts "Emailing #{@email}"
  end
end
```

See: `ruby_inheritance-1b.rb`

```
class Student < Person
  def to_s
    puts "Name: #{@name}"
  end
end
```

## Example: Variable Not 'Inherited'

```
class Person
  def initialize(name)
    @name = name
    puts "initializing"
  end
  def setupEmail(email)
    @email = email
  end
  def sendEmail()
    puts "Emailing #{@email}"
  end
end
```

```
class Student < Person
  def to_s
    puts "Name: #{@name}"
  end
end
```

Trace: What is displayed?

```
p = Person.new("Devin")
p.setupEmail("dev@mines.edu")
s = Student.new("Gene")
p.sendEmail
s.sendEmail
```

See: `ruby_inheritance-1b.rb`

# Inheritance and Methods

# Inheritance and Overriding

- Child class can override parent methods
- Methods
  - ... are bound *dynamically* (when executed)
  - ... not statically (when parsed)
- Methods like `to_s` and `initialize` are automatically inherited (from `Object`)\*

\*If you don't know all of the methods of the parent class, you may accidentally override a method!

# Language Comparison

Run `ruby_inheritance-1.rb`

- Does Java automatically call parent constructor?
  - Read
- Compare to C++
  - Read
- Questions<sup>\*</sup>
  - ① In Java, when do you need to explicitly call the parent ctor?
  - ② In C++, why don't they use a keyword like `super` to call the parent ctor?

<sup>\*</sup>Not exam topics



# Language Comparison

Assume you're writing a C++ program with:

- ❶ Parent named `Bug`, child named `Mosquito`
  - ❷ A method in both parent/child named `bite`
- What do you need to make sure this is bound dynamically?
  - What happens if this is not bound dynamically?
    - Write a few lines of C++ (on paper) to illustrate

Helpful reminder

# Big Picture

Usually, when dealing with an OO language. . .

- Inheritance is a part of the language
- There's a way to ensure parent/child vars are initialized
- Child classes can call parent class methods
- Child classes can override parent methods
  - Runtime: *dynamic/late binding*
  - Compile time: *static/early binding*

# Override Parent Method

```
class Person
  def initialize(name)
    @name = name
  end
  def introduce
    puts "Hi, I'm #{@name}"
  end
end

class Student < Person
  def introduce
    puts "I'm a student and "\
        "my name is #{@name}"
  end
end
```

See: [ruby\\_inheritance-2a.rb](#)

# Override Parent Method

```
class Person
  def initialize(name)
    @name = name
  end
  def introduce
    puts "Hi, I'm #{@name}"
  end
end

class Student < Person
  def introduce
    puts "I'm a student and "\
        "my name is #{@name}"
  end
end
```

```
joe = Person.new("Joe")
joe.introduce
jamie = Student.new("Jamie")
jamie.introduce
```

See: [ruby\\_inheritance-2a.rb](#)

# Ruby Method Visibility

## 1 Public

- Methods are *public by default*
- `initialize` is implicitly private (called by `new`)

## 2 Private

- Only visible to *other methods* of the class/subclass
- Implicitly invoked on `self`

## 3 Protected

- Like private, but can be invoked on *any instance* of class
- Allows objects of same type to share state (used infrequently)

These only apply to methods!

Instance vars are private, constants are public

# Method Visibility Example 1

```
class X
  # public methods by default
  def fn
    # ...
  end
  protected :fn
  def helper
    # ...
  end
  private :helper
end
```

# Method Visibility Example 1

```
class X
  # public methods by default
  def fn
    # ...
  end
  protected :fn
  def helper
    # ...
  end
  private :helper
end
```

- Can override visibility (**reference**)
  - `private_class_method :new`
- `private` and `protected`
  - Guard against unintended use\*

\*But, with metaprogramming, it's possible to call these methods

## Method Visibility Example 2 (1/2)

```
class Person
  def initialize(name)
    @name = name
    puts "initializing"
  end

  def talk_to(friend)
    puts "Talking to #{@friend}"
  end
  private :talk_to
end
```

See: `ruby_inheritance-2b.rb`



## Method Visibility Example 2 (2/2)

```
class Person
  def initialize(name)
    @name = name
    puts "initializing"
  end

  def talk_to(friend)
    puts "Talking to #{@friend}"
  end

  private :talk_to
end
```

```
p = Person.new("Yeezy")
p.talk_to("Weezy")
```

See: `ruby_inheritance-2b.rb`

# Abstract Class Methods

- Implicitly defined in Ruby
- Parent class calls methods that child must define

## Example: Abstract Class Methods (1/3)

```
class AbstractGreeter
  def greet
    puts "#{greeting} #{who}" # call abstract methods
  end
end

class WorldGreeter < AbstractGreeter
  def greeting; "Hello"; end
  def who; "Jerry"; end
end
```

See: `ruby_inheritance-3.rb`

## Example: Abstract Class Methods (2/3)

```
class AbstractGreeter
  def greet
    puts "#{greeting} #{who}" # call abstract methods
  end
end

class WorldGreeter < AbstractGreeter
  def greeting; "Hello"; end
  def who; "Jerry"; end
end
```

What makes AbstractGreeter an abstract class?

How does this compare to Java? C++?

See: `ruby_inheritance-3.rb`

## Example: Abstract Class Methods (3/3)

```
WorldGreeter.new.greet  
AbstractGreeter.new.greet  
AbstractGreeter.new.say_hi
```

See: `ruby_inheritance-3.rb`

## Example: Chaining Methods (1/3)

```
class Person
  def initialize(name)
    @name = name
  end
  def introduce
    puts "Hi, I'm #{@name}"
  end
end
```

See: `ruby_inheritance-4.rb`

## Example: Chaining Methods (2/3)

```
class Person
  def initialize(name)
    @name = name
  end
  def introduce
    puts "Hi, I'm #{@name}"
  end
end
```

```
class Student < Person
  def initialize(name)
    super(name)
    @major = major
  end
  def introduce
    super
    puts "I'm studying #{@major}"
  end
end
```

See: `ruby_inheritance-4.rb`

## Example: Chaining Methods (3/3)

```
p = Person.new("Lauryn")  
p.introduce  
s = Student.new("Shawn", "Poetry")  
s.introduce
```

See: `ruby_inheritance-4.rb`



# Class Variables

# Class Variables – Review

- When did we use `static` class vars in Java/C++?
- Ruby class variables can be used for similar purposes

## Example: Class Variables (1/2)

```
class Person
  def initialize(name)
    @name = name
    @@thing2 = "water"
  end
  def show
    puts "Person: #{@thing1}"
  end
end
```

```
class Student < Person
  def make_thing1
    @@thing1 = "oil"
  end
  def show
    puts "Student: #{@thing1}" \
      " and #{@thing2}"
  end
end
```

See: `ruby_inheritance-5.rb`

## Example: Class Variables (2/2)

```
a = Person.new("Amy")
b = Student.new("Bob")
# create class variable `thing1`
b.make_thing1
b.show
# all students can access `thing1`
c = Student.new("Charlie")
c.show
# parent cannot access `thing1`
a.show # error
```

See: `ruby_inheritance-5.rb`

# Class Instance Variables

May want to explore on your own **Class vs. Class-instance variables**\*

\* **Not on exam**