CSE M226: Programming Languages

Lecture 6:Duck Typing

Rokan Uddin Faruqui

Dept. of Computer Science and Engineering
University of Chittagong, Bangladesh
Email: rufaruqui@cu.ac.bd

Type Safety

- Static Type
- Dynamic Type

Classes Aren't Types

```
1 Customer c;
  c = database.findCustomer("dave"); /* Java */
3
```

Classes Aren't Types

- Java supports the concept of interfaces, which are a kind of emasculated abstract base class.
- A Java class can be declared as implementing multiple interfaces

Classes Aren't Types

```
public interface Customer {
    long getID();
    Calendar getDateOfLastContact();
    // ...
    }

public class Person
    implements Customer {
    public long getID() { ... }
    public Calendar getDateOfLastContact() { ... }
    // ...
    }
```

Classes Aren't Types

So, even in Java, the class is not always the *type* -

- sometimes the type is **a subset of the class**,
- sometimes objects **implement** multiple types.

Classes Aren't Types

In Ruby,

- the class is never (OK, almost never) the type.
- the type of an object is defined more by what that object can do. In Ruby, we call this duck typing.
- If an object walks like a duck and talks like a duck, then the interpreter is happy to treat it as if it were a duck.

Classes Aren't Types

```
class Customer
  def initialize(first_name, last_name)
    @first_name = first_name
    @last_name = last_name
  end

  def append_name_to_file(file)
      file << @first_name << " " << @last_name
  end
end
```

Classes Aren't Types

```
1   require 'test/unit'
    require_relative 'addcust'
3
    class TestAddCustomer < Test::Unit::TestCase
5   def test_add
      c = Customer.new("lma", "Customer")
7     f = File.open("tmpfile", "w") do |f|
      c.append_name_to_file(f)
9   end
    f = File.open("tmpfile") do |f|
11      assert_equal("lma Customer", f.gets)
     end
13
  ensure
15    File.delete("tmpfile") if File.exist?("tmpfile
   ")
  end
```

Test results

produce

```
Finished in 0.003934 seconds.
------------------------------
1 tests, 1 assertions, 0 failures, 0 errors, 0
  pendings, 0 omissions, 0 notifications
100% passed
------------------------------
```

Classes Aren't Types

```
require 'test/unit'
require_relative 'addcust'

class TestAddCustomer < Test::Unit::TestCase
  def test_add
    c = Customer.new("Ima", "Customer")
    f = ""
    c.append_name_to_file(f)
    assert_equal("Ima Customer", f.gets)
  end
end

```

Classes Aren't Types

```
require 'test/unit'
require_relative 'addcust'

class TestAddCustomer < Test::Unit::TestCase
def test_add
c = Customer.new("Ima", "Customer")
f = []
c.append_name_to_file(f)
assert_equal("Ima Customer", f.gets)
end
end
```

Duck Typing – Good for only TESTING ?

```
        def csv_from_row(op, row)
          res = ""
          until row.empty?
            entry = row.shift.to_s
            if /[,"]/ =~ entry
              entry = entry.gsub(/"/, '""')
              res << '"' << entry << '"'
            else
              res << entry
            end
            res << "," unless row.empty?
          end
          op << res << CRLF
        end
result = ""
query.each_row {|row| csv_from_row(result, row)}
http.write result
```

Duck Typing – Good for only TESTING ?

```
1       def csv_from_row(op, row)
       #as before
3       end
        result = []
5       query.each_row {|row| csv_from_row(result,
    row)}
        http.write result
7
```

Duck Typing – Good for only TESTING ?

```
def append_song(result, song)
      # test we're given the right parameters
    unless result.kind_of?(String)
      fail TypeError.new("String expected")
    end
    unless song.kind_of?(Song)
      fail TypeError.new("Song expected")
    end

    result << song.title << " (" << song.artist << ")"
  end

result = ""
append_song(result, song)

```

Duck Typing – Good for only TESTING ?

```
1 def append_song(result, song)
    result << song.title << " (" << song.artist << ")
      "
3 end

5 result = ""
  append_song(result, song)
7
```

Duck Typing – Good for only TESTING ?

```
def append_song(result, song)
  # test we're given the right parameters
  unless result.respond_to?(:<<)
    fail TypeError.new("'result' needs '<<' capability")
  end
  unless song.respond_to?(:artist) && song.respond_to?(:title)
    fail TypeError.new("'song' needs 'artist' and 'title'")
  end
  result << song.title << " (" << song.artist << ")"
end
result = ""
append_song(result, song)
```

conversion protocols – an object may elect to have itself converted to an object of another class.

- loose conversion– *to_s*, *to_i*
- strict conversion –*to_str*, *to_int*
- numeric coercion –$1 + 2$, $1 + 2.3$

Duck typing can generate controversy.

- Duck typing isn't a set of rules; it's just a style of programming

Duck typing can generate controversy.

- Duck typing isn't a set of rules; it's just a style of programming
- Design your programs to balance paranoia and flexibility.

Duck typing can generate controversy.

- Duck typing isn't a set of rules; it's just a style of programming
- Design your programs to balance paranoia and flexibility.
- If you feel the need to constrain the types of objects that the users of a method pass in, ask yourself why.

Duck typing can generate controversy.

- Duck typing isn't a set of rules; it's just a style of programming
- Design your programs to balance paranoia and flexibility.
- If you feel the need to constrain the types of objects that the users of a method pass in, ask yourself why.
- Try to determine what could go wrong if you were expecting a String and instead get an Array.

Duck typing can generate controversy.

- Duck typing isn't a set of rules; it's just a style of programming
- Design your programs to balance paranoia and flexibility.
- If you feel the need to constrain the types of objects that the users of a method pass in, ask yourself why.
- Try to determine what could go wrong if you were expecting a String and instead get an Array.
- Sometimes, the difference is crucially important. Often, though, it isn't.

Duck typing can generate controversy.

- Duck typing isn't a set of rules; it's just a style of programming
- Design your programs to balance paranoia and flexibility.
- If you feel the need to constrain the types of objects that the users of a method pass in, ask yourself why.
- Try to determine what could go wrong if you were expecting a String and instead get an Array.
- Sometimes, the difference is crucially important. Often, though, it isn't.
- Try erring on the more permissive side for a while, and see whether bad things happen.

Duck typing can generate controversy.

- Duck typing isn't a set of rules; it's just a style of programming
- Design your programs to balance paranoia and flexibility.
- If you feel the need to constrain the types of objects that the users of a method pass in, ask yourself why.
- Try to determine what could go wrong if you were expecting a String and instead get an Array.
- Sometimes, the difference is crucially important. Often, though, it isn't.
- Try erring on the more permissive side for a while, and see whether bad things happen.
- If not, perhaps duck typing isn't just for the birds. report

Duck typing can generate controversy.

- Duck typing isn't a set of rules; it's just a style of programming
- Design your programs to balance paranoia and flexibility.
- If you feel the need to constrain the types of objects that the users of a method pass in, ask yourself why.
- Try to determine what could go wrong if you were expecting a String and instead get an Array.
- Sometimes, the difference is crucially important. Often, though, it isn't.
- Try erring on the more permissive side for a while, and see whether bad things happen.
- If not, perhaps duck typing isn't just for the birds. report