Ruby: Objects and Dynamic Types

Lecture 6
Primitive vs Reference Types

- Recall Java type dichotomy:
  - Primitive: int, float, double, boolean,…
  - Reference: String, Set, NaturalNumber,…

- A variable is a “slot” in memory
  - Primitive: the slot holds the value itself
  - Reference: the slot holds a pointer to the value (an object)
Object Value vs Reference Value

- Variable of reference type has *both*:
  - Reference value: value of the *slot* itself
  - Object value: value of *object* it points to (corresponding to its mathematical value)

- Variable of primitive type has *just one*
  - Value of the slot itself, corresponding to its mathematical value
Two Kinds of Equality

- Question: “Is x equal to y?”
  - A question about the mathematical value of the variables x and y

- In Java, depending on the type of x and y we either need to:
  - Compare the values of the slots
    
    ```java
    x == y // for primitive types
    ```
  - Compare the values of the objects
    
    ```java
    x.equals(y) // for non-primitive types
    ```
Ruby: “Everything is an Object”

- In Ruby, every variable maps to an object
  - Integers, floats, strings, sets, arrays, ...
- Benefit: A more consistent mental model
  - References are everywhere
  - Every variable has both a reference value and an object value
  - Comparison of mathematical values is always comparison of object value
- Ruby terminology: Reference value is called the object id
  - The 8-byte number stored in the slot
  - Unique identifier for corresponding object

```ruby
tau = 6.28
tau.object_id #=> 565652113197773434
```
Everything is an Object

- a
  - Width: 12
  - Height: 15
  - Color: "blue"

- d
  - Value: 34
Everything is an Object

- a
  - width: 12
  - height: 15
  - color: "blue"
  - 34

- msg
  - "shark"

- tau
  - 56565211319773434
  - 6.28

- d
  - width: 12
  - height: 15
  - color: "blue"

- done
  - true

- list
  - <1,2,8,2>

Computer Science and Engineering  ■  The Ohio State University
Operational Detail: Immediates

- For small integers, the mathematical value is *encoded in the reference value!*
  - LSB of reference value is 1
  - Remaining bits encode value, 2's complement
    
    ```
    x = 0
    x.object_id #=> 1 (0b00000001)
    y = 6
    y.object_id #=> 13 (0b00001101)
    ```

- Known as an “immediate” value
  - Others: true, false, nil, symbols, string literals

- Benefit: Performance
  - No change to model, *everything is an object*
Objects Have Methods

- Familiar "." operator to invoke (instance) methods
  
  ```ruby
  list = [6, 15, 3, -2]
  list.size #=> 4
  ```

- Since numbers are objects, they have methods too!
  
  ```ruby
  3.to_s #=> "3"
  3.odd? #=> true
  3.lcm 5 #=> 15
  3.+ 5 #=> 8
  3.class #=> Integer
  3.methods #=> [:to_s, :inspect, :+, ...]
  ```
Pitfall: Equality Operator

- Reference value is still useful sometimes
  - “Do these variables refer to the same object?”
- So we still need 2 methods:
  - \( x == y \)
  - \( x.equals? y \)

- Ruby semantics are the **opposite** of Java!
  - == is *object value* equality
  - .equals? is *reference value* equality

- Example

```ruby
a1, a2 = [1, 2], [1, 2] # "same" array
a1 == a2 #=> true (obj values equal)
a1.equals? a2 #=> false (ref vals differ)
```
To Ponder

Evaluate (each is true or false):

3 == 3

3.equal? 3

[3] == [3]

Assignment (Just Like Java)

- Assignment copies the *reference value*
- Result: Both variables point to the *same* object (i.e., an “alias”)
- Parameter passing works this way too

![Diagram]

```plaintext
a: <5, 1>
b: <3, 4>
```
Assignment (Just Like Java)

- Assignment copies the *reference value*
- Result: Both variables point to the *same* object (ie an “alias”)
- Parameter passing works this way too

```java
a = b;
```

```
<5, 1>  <3, 4>  <5, 1>  <3, 4>
```
Assignment (Just Like Java)

- Assignment copies the *reference value*
- Result: Both variables point to the *same* object (ie an “alias”)
- Parameter passing works this way too

```
a b
<5, 1> <3, 4>
```

```
a = b;
```

```
a b
<5, 1> <3, 4>
```
Aliasing Mutable Objects

- When aliases exist, a statement can change a variable’s object value without mentioning that variable.

  ```
x = [3, 4]
y = x  # x and y are aliases
y[0] = 13  # changes x as well!
  ```

- Question: What about numbers?

  ```
i = 34
j = i  # i and j are aliases
j = j + 1  # does this increment i too?
  ```
Immutability

- Recall in Java strings are immutable
  - No method changes the value of a string
  - A method like concat returns a new instance
- Benefit: Aliasing immutable objects is safe
- Immutability is used in Ruby too
  - Numbers, true, false, nil, symbols
    ```ruby
    list = [3, 4]
    list[0] = 13  # changes list's object value
                  # list points to same object
    n = 34
    n = n + 1    # changes n's reference value
                  # n points to different object
    ```
- Pitfall: Unlike Java, strings in Ruby are mutable
- But objects (including strings) can be “frozen”
Freezing

- Makes a (single) object immutable
  - The object value can not change

```ruby
list = [1, 2, 8, 2].freeze
list.length #=> 4
list[0] = 3  # error: can't modify a
             # frozen object
list = [7, -1]  # ok: ref value changed
```

Frozen object (shallow)
Assignment Operators

- Parallel assignment
  \[ x, y, z = y, 10, \text{radius} \]

- Arithmetic contraction
  \[ += -= *= /= %= **= \]
  Pitfall: no ++ or -- operators (use += 1)

- Logical contraction
  \[ ||= &&= \]
  Idiom: ||= for initializing potentially nil variables
  Pitfall (minor):
  - \[ x ||= y \] not quite equivalent to \[ x = x || y \]
  - Better to think of it as \[ x || x = y \]
  - Usually amounts to the same thing
Declared vs Dynamic Types

- In Java, types are associated with both
  - Variables (declared / static type), and
  - Objects (dynamic / run-time type)

```java
Queue line = new Queue1L();
```

- Recall: Programming to the interface

- Compiler uses declared type for checks

```
line.inc(); // error no such method
line = new Set1L(); // err. wrong type
```

```java
boolean isEmpty (Set s) {...}
if isEmpty(line) ... // error arg type
```
Statically Typed Language

- Line:
  - Queue: `<1, 2, 8, 2>`
  - Queue1L

- Msg:
  - String: "hello"

- D:
  - Shape:
    - Rectangle:
      - width: 12
      - height: 15
      - color: "blue"
Dynamically Typed Language

line

<1, 2, 8, 2>
Queue1L

msg

"hello"
String

d

width: 12
height: 15
color: "blue"
Rectangle
Dynamically Typed Language

- Equivalent definitions:
  - No static types
  - Dynamic types only
  - Variables do not have type, objects do
Function Signatures

- Statically typed

  ```java
  String parse(char[] s, int i) { ... return e;}
  out = parse(t, x);
  ```
  
  - Declare parameter and return types
  - See `s`, `i`, and `parse`
  - The **compiler** checks conformance of
    - (Declared) types of arguments (`t`, `x`)
    - (Declared) type of return expression (`e`)
    - (Declared) type of expression *using* `parse` (`out`)

- Dynamically typed

  ```ruby
  def parse(s, i) ... e end
  out = parse t, x
  ```
  
  - You are on your own!
**Type Can Change at Run-time**

<table>
<thead>
<tr>
<th>Statically Typed</th>
<th>Dynamically Typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>//a is undeclared</td>
<td># a is undefined</td>
</tr>
<tr>
<td>String a;</td>
<td>a = a</td>
</tr>
<tr>
<td>//a is null string</td>
<td># a is nil</td>
</tr>
<tr>
<td>a = &quot;hi;&quot;</td>
<td>a = &quot;hi&quot;</td>
</tr>
<tr>
<td>//compile-time err</td>
<td># load-time error</td>
</tr>
<tr>
<td>a = &quot;hi&quot;;</td>
<td>a = &quot;hi&quot;</td>
</tr>
<tr>
<td>a = 3;</td>
<td>a = 3</td>
</tr>
<tr>
<td>//compile-time err</td>
<td># a is now a number</td>
</tr>
<tr>
<td>a.push();</td>
<td>a.push</td>
</tr>
<tr>
<td>//compile-time err</td>
<td># run-time error</td>
</tr>
</tbody>
</table>
Changing Dynamic Type

- **line**: <1, 2, 8, 2>
  - Queue1L
- **msg**: "hello"
  - String
Changing Dynamic Type

```
msg, line = line, msg
```
Changing Dynamic Type

```python
msg, line = line, msg
```

Diagram:

- `line` connected to `<1, 2, 8, 2>` Queue1L
- `msg` connected to "hello" String

Diagram:

- `line` connected to `<1, 2, 8, 2>` Queue1L
- `msg` connected to "hello" String
Arrays: Static Typing

String msg = "hello";
Arrays: Static Typing

```java
String msg = "hello";
String[] msgs = ["hello", "world", ...];
```
Arrays: Dynamic Typing

msg = "hello";

msgs = ["hello", "world", ...];
Consequence: Heterogeneity

```java
msgs = ["hello", 3.14, ...];
```
Tradeoffs

**Statically Typed**
- Earlier error detection
- Clearer APIs
- More compiler optimizations
- Richer IDE support

**Dynamically Typed**
- Less code to write
- Less code to change
- Quicker prototyping
- No casting needed
Strongly Typed

- Just because variables don’t have types, doesn’t mean you can do anything you want

```ruby
>> "hi".upcase
=> "HI"
>> "hi".odd?
NoMethodError: undefined method `odd?' for String
```

```ruby
>> puts "The value of x is " + x
TypeError: can't convert Integer to String
```
Summary

- Object-oriented
  - References are everywhere
  - Assignment copies reference value (alias)
  - Primitives (immediates) are objects too
  - == vs .equal? are flipped

- Dynamically type
  - Objects have types, variables do not

- Strongly Typed
  - Incompatible types produce (run time) error