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
    $x == y$  // for primitive types
  - Compare the values of the objects
    $x.equals(y)$  // 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 4- or 8-byte number stored in the slot
  - Unique identifier for corresponding object

```ruby
msg = "shark"
msg.object_id #=> 47204497544380
```
Everything is an Object

- `a`: `34`
- `msg`: `47204497544380` -> "shark"
- `d`: width: 12, height: 15, color: "blue"
- `done`: `true`
- `list`: `<1,2,8,2>`
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
    
    ```plaintext
    x = 0
    x.object_id #=> 1 (0b00000001)
    y = 6
    y.object_id #=> 13 (0b00001101)
    ```

- Benefit: Performance
  - No change to model (everything is an object)

- Known as an “immediate” value
  - Other immediates: true, false, nil, symbols
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.equal? y`
- Ruby semantics are the opposite of Java!
  - `==` is object value equality
  - `.equal?` is reference value equality
- Example
  - `s1, s2 = "hi", "hi"
  - `s1 == s2 #=> true (obj values equal)`
  - `s1.equal? s2 #=> false (ref vals differ)`
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

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

```
a

b
```
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

```
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

```
<5, 1>
```

```
<3, 4>
```

```
a = b;
```
Aliasing Mutable Objects

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

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

- Question: What about numbers?

```python
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

```
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*
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

```java
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
Dynamically Typed Language

- **line**
  - Width: 12
  - Height: 15
  - Color: "blue"
  - Message: "hello"

- **msg**
  - Value: <1, 2, 8, 2>
  - Queue: Queue1L

- **d**
  - Width: 12
  - Height: 15
  - Color: "blue"
  - Type: 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**

  ```python
  def parse(s, i) ... e end
  out = parse t, x
  ```

  - You are on your own!
Type Can Change at Run-time

**Statically Typed**

```java
//a is undeclared
String a;
//a is null string
a = "hi;
//compile-time err
a = "hi"
```

**Dynamically Typed**

```javascript
# a is undefined
a = a
# a is nil
a = "hi"
# load-time error
a = "hi"
```

```javascript
a = 3
# a is now a number
a = 3
```

```javascript
a.push();
# compile-time err
a.push
# run-time error
```
Changing Dynamic Type

```
line
<1, 2, 8, 2>
Queue1L
```

```
msg
"hello"
String
```
Changing Dynamic Type

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

![Diagram showing the assignment of `msg` and `line`]

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

msg, line = line, msg

![Diagram showing dynamic type change]
Arrays: Static Typing

```java
String msg = "hello";
```
Arrays: Static Typing

```java
String msg = "hello";

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

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

```latex
msgs = ["hello", 3.14, ...];
```

```
msgs
```

```
"hello"
String
```

```
3.14
Float
```

```
17
Integer
```
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