Outline

- How to Describe a Language
- Dynamic Typing
- Text Processing
- Data Manipulation
- Closures and Objects
- Web Programming
- Security
- Debugging

Soap-Box

How to Describe a Language

- About the language
- Related languages
- Example
- How to write + run code
- Types
- Variable declarations
- Type conversions
- Input and output
- Operators
- Arrays
- Hashes
- Control statements

Writing subroutines
Using objects
Structure of an application
Scopes and visibility
Embedded object model
Defining classes
Inheritance
Call-backs
Library functions
Reference documentation
Common mistakes
Suggestions for practice
Evaluating the language

Subtype Diagram

Subtyping: general to specific

<table>
<thead>
<tr>
<th>Subtype</th>
<th>General Type</th>
<th>Specific Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Array</td>
<td>Empty</td>
<td>Full</td>
</tr>
<tr>
<td>String</td>
<td>Single</td>
<td>Double</td>
</tr>
</tbody>
</table>

Conversions Table

<table>
<thead>
<tr>
<th>Value</th>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>False</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

BNF = Backus Naur Form

<table>
<thead>
<tr>
<th>Construct</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminal</td>
<td>concrete code</td>
<td>&quot;Courier Bold&quot;</td>
</tr>
<tr>
<td>Non-terminal</td>
<td>placeholder</td>
<td>&quot;Times Italic&quot;</td>
</tr>
<tr>
<td>Ellipsis</td>
<td>omitted code</td>
<td>&quot;...&quot;</td>
</tr>
<tr>
<td>Rule</td>
<td>non-terminal definition</td>
<td>&quot;Lhs ::= Rhs&quot;</td>
</tr>
<tr>
<td>Alternative</td>
<td>choose one</td>
<td>&quot;Alt1</td>
</tr>
<tr>
<td>Optional</td>
<td>zero or one times</td>
<td>&quot;[Square brackets]&quot;</td>
</tr>
<tr>
<td>Repeat</td>
<td>zero or more times</td>
<td>&quot;Kleene star*&quot;</td>
</tr>
<tr>
<td>Grouping</td>
<td>treat as unit</td>
<td>&quot;(parentheses)&quot;</td>
</tr>
</tbody>
</table>
### Operator Table

<table>
<thead>
<tr>
<th>Operator</th>
<th>Arity</th>
<th>Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>(, )</td>
<td>2</td>
<td>null</td>
</tr>
<tr>
<td>=</td>
<td>2</td>
<td>= left</td>
</tr>
<tr>
<td>=</td>
<td>2</td>
<td>R = right</td>
</tr>
<tr>
<td>+=</td>
<td>2</td>
<td>Multi-associative</td>
</tr>
<tr>
<td>=</td>
<td>2</td>
<td>L = left</td>
</tr>
<tr>
<td>/</td>
<td>2</td>
<td>= left</td>
</tr>
<tr>
<td>Mod</td>
<td>2</td>
<td>= left</td>
</tr>
<tr>
<td>&gt;, &lt;</td>
<td>2</td>
<td>= left</td>
</tr>
<tr>
<td>&gt;=, &lt;=</td>
<td>2</td>
<td>= left</td>
</tr>
<tr>
<td>Bit shift</td>
<td>2</td>
<td>= left</td>
</tr>
<tr>
<td>&lt;=, =&gt;</td>
<td>2</td>
<td>= left</td>
</tr>
<tr>
<td>!=</td>
<td>2</td>
<td>= left</td>
</tr>
<tr>
<td>Bit shift</td>
<td>2</td>
<td>= left</td>
</tr>
<tr>
<td>.</td>
<td>2</td>
<td>= left</td>
</tr>
<tr>
<td>++, --</td>
<td>2</td>
<td>= left</td>
</tr>
<tr>
<td>Set</td>
<td>2</td>
<td>Assignment statement</td>
</tr>
</tbody>
</table>

Precedence: from high to low

### Object Model

- Top-level "creatable objects"
- Simple object
- "has-a" / "contains" relationship
- "_is_a" relationship
- More objects (not shown)

### UML Class Diagram

- Fruit
  - weight
  - pluck()
  - prepare()
- Banana
  - weight
  - pluck()
  - prepare()
- Apple
  - weight
  - color
  - pluck()
  - prepare()
- Subclasses/ Derived classes

### UML Object Diagram

- `a1` : Apple
  - color = "red"
  - weight = 150
- `a2` : Apple
  - color = "green"
  - weight = 150
- Salad : Recipe
  - size = "large"
  - ingredients

### UML Sequence Diagram

- VBA code
  - `frmLemonStar.Show()`
- Interpreter
  - wait for user input
- User
  - edit text box
  - click button
  - cmdPaint_Click()
**How to Write + Run Code**

<table>
<thead>
<tr>
<th>Concept</th>
<th>Bash</th>
<th>Perl</th>
<th>Ruby</th>
<th>PHP</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Batch</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Shebang</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>One-liner</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Embedded (application)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Embedded (HTML)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Type Conversions**

<table>
<thead>
<tr>
<th>Context Provider</th>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Boolean</td>
<td>String</td>
</tr>
<tr>
<td>String</td>
<td>Boolean</td>
<td>Number</td>
</tr>
<tr>
<td>Array</td>
<td>Boolean</td>
<td>String</td>
</tr>
</tbody>
</table>

**Perl**

**Context**

- When you use a value in a different context than the value’s type, Perl converts it.
  - E.g., using array as scalar → int length
- Context provided by:
  - Operator: e.g., string . string
  - Function: e.g., print list
  - Assignment: e.g., $a = list
  - Statement: e.g., if (boolean) {...}
- Functions provide context for parameters, and can react to context for return value

**Context Example**

<table>
<thead>
<tr>
<th>Source code</th>
<th>Original Value</th>
<th>Context provider</th>
<th>Context</th>
<th>Converted value</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 + (10, 11, 12)</td>
<td>List literal (10, 11, 12)</td>
<td>Operator +</td>
<td>Scalar (number)</td>
<td>3 + 12 = 15</td>
</tr>
<tr>
<td>8 * a = (10, 11, 12); 3 * $a</td>
<td>Array variable $a</td>
<td>Function length</td>
<td>Scalar (string)</td>
<td>length &quot;3&quot; = 6</td>
</tr>
<tr>
<td>8 * a = (10, 11, 12); length $a</td>
<td>Array variable $a</td>
<td>Assignment</td>
<td>List</td>
<td>(&quot;hi&quot;)</td>
</tr>
<tr>
<td>$a = &quot;hi&quot;;</td>
<td>Scalar literal &quot;hi&quot;</td>
<td>Assignment</td>
<td>List</td>
<td>(&quot;hi&quot;)</td>
</tr>
<tr>
<td>sort 12, 10, 11</td>
<td>List literal (10, 11, 12)</td>
<td>Function sort</td>
<td>List</td>
<td>(10, 11, 12)</td>
</tr>
</tbody>
</table>

**List Conversions**

<table>
<thead>
<tr>
<th>Value</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>identity</td>
</tr>
<tr>
<td>List</td>
<td>1-element list</td>
</tr>
<tr>
<td>List literal</td>
<td>last element</td>
</tr>
<tr>
<td>flattened sublists</td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td>length</td>
</tr>
<tr>
<td>identity</td>
<td></td>
</tr>
<tr>
<td>Hash</td>
<td>empty</td>
</tr>
<tr>
<td>&quot;0&quot;</td>
<td>empty list</td>
</tr>
<tr>
<td>non-empty</td>
<td><em>size</em>bucket*</td>
</tr>
<tr>
<td>m/</td>
<td>true if match</td>
</tr>
<tr>
<td>all groups</td>
<td></td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>end of file</td>
</tr>
<tr>
<td>\not EOF</td>
<td>empty array</td>
</tr>
<tr>
<td>line</td>
<td>all lines</td>
</tr>
</tbody>
</table>
Type System Variations

- **Duck Typing**
  - Object-oriented dynamic typing
  - Requires more dynamic method lookup than Java-like languages
  - Examples: Ruby, Python, PHP, Perl, VBA

- **Gradual Typing**
  - User chooses between static and dynamic typing on a per-variable basis (e.g. VBA)

- **Pluggable Type Systems**
  - User chooses to enable additional static checks on a per-file basis (e.g. Perl pragmas)

Outline

- How to Describe a Language
- Dynamic Typing
- Text Processing
- Data Manipulation
- Framework Support
- Web Programming
- Security
- Debugging

Text Processing Example

Python
```
for line in sys.stdin.readlines():
    qty, unit, ing = re.search(r'([0-9.]+) ([\w]+) ([\w]+)', line).groups()
    ... 
```

Perl
```
while (<>) {
    my ($qty, $unit, $ing) = /([0-9.]+) ([\w]+) ([\w]+)/;
    ...
}
```

Ruby
```
while gets
    $qty, $unit, $ing = (/$([0-9.]+) ([\w]+) ([\w]+)/)
    ...
end
```

Comparison

<table>
<thead>
<tr>
<th>Text Processing Features</th>
<th>Perl</th>
<th>Python</th>
<th>Ruby</th>
<th>PHP</th>
<th>Java/Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit variable $x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RegExp literal /.../</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RegExp match operator =~</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Implicit variables $1, $2, ...</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>String interpolation: sigils-vari.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>String interpolation: expression</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>String formatting operator %</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Chomsky Hierarchy

<table>
<thead>
<tr>
<th>Type</th>
<th>Languages</th>
<th>Automata</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Recursively enumerable</td>
<td>Turing machine</td>
<td>c&lt;-&gt;d</td>
</tr>
<tr>
<td>1</td>
<td>Context sensitive</td>
<td>Linear bounded Turing machine</td>
<td>c&lt;</td>
</tr>
<tr>
<td>2</td>
<td>Context free</td>
<td>Pushdown automaton</td>
<td>d&lt;</td>
</tr>
<tr>
<td>3</td>
<td>Regular</td>
<td>Finite state machine</td>
<td>A-&gt;c, C-&gt;d</td>
</tr>
</tbody>
</table>

- Formal regexp only has *essentials*
- Perl adds shortcuts and non-regular features

Inside a Regular Expression

<table>
<thead>
<tr>
<th>Kind</th>
<th>Construct</th>
<th>Syntax</th>
<th>Example Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Essentials</td>
<td>Character</td>
<td>itself</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>Concatenation</td>
<td></td>
<td>c</td>
<td>a, b, c</td>
</tr>
<tr>
<td></td>
<td>Alternative (or)</td>
<td></td>
<td>c</td>
<td>a, b, c</td>
</tr>
<tr>
<td></td>
<td>Repetition (u0)</td>
<td></td>
<td>a</td>
<td>a, a, a, a</td>
</tr>
<tr>
<td></td>
<td>Grouping</td>
<td>(c)</td>
<td>(a</td>
<td>b)</td>
</tr>
<tr>
<td>Quantifiers</td>
<td>Optional</td>
<td></td>
<td>(a</td>
<td>b)?</td>
</tr>
<tr>
<td></td>
<td>Repetition (u1)</td>
<td></td>
<td>a</td>
<td>a, a, a, a</td>
</tr>
<tr>
<td></td>
<td>Char class</td>
<td>Custom class</td>
<td>[a-c]</td>
<td>a, b, c</td>
</tr>
<tr>
<td></td>
<td>Wildcard character</td>
<td>.</td>
<td>.</td>
<td>a, ., .</td>
</tr>
<tr>
<td></td>
<td>Shortcut class</td>
<td>\</td>
<td>d</td>
<td>0.1, 2, 9</td>
</tr>
</tbody>
</table>

Assertion | Zero-width anchor | ^ | \w | (word boundary)
Outline

- How to Describe a Language
- Dynamic Typing
- Text Processing
- Data Manipulation
- Closures and Objects
- Web Programming
- Security
- Debugging

Comparison

<table>
<thead>
<tr>
<th>String/Text Type</th>
<th>Creation</th>
<th>Indexing</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Length</th>
<th>Slicing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala</td>
<td>a+(4,9)</td>
<td>a(4,9)</td>
<td>a+(4,9)</td>
<td>a(4,9)</td>
<td>a(4,9)</td>
<td>a(4,9)</td>
</tr>
<tr>
<td>Python</td>
<td>s=a[0]</td>
<td>s[0]</td>
<td>s=a[0]</td>
<td>s[0]</td>
<td>s[0]</td>
<td>s[0]</td>
</tr>
<tr>
<td>JavaScript</td>
<td>s=a[0]</td>
<td>s[0]</td>
<td>s=a[0]</td>
<td>s[0]</td>
<td>s[0]</td>
<td>s[0]</td>
</tr>
<tr>
<td>PHP</td>
<td>$a[0]</td>
<td>$a[0]</td>
<td>$a[0]</td>
<td>$a[0]</td>
<td>$a[0]</td>
<td>$a[0]</td>
</tr>
<tr>
<td>Ruby</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
</tr>
<tr>
<td>Bash</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
</tr>
<tr>
<td>VBA</td>
<td>a[0]</td>
<td>a[0]</td>
<td>a[0]</td>
<td>a[0]</td>
<td>a[0]</td>
<td>a[0]</td>
</tr>
</tbody>
</table>

Arrays

<table>
<thead>
<tr>
<th>String/Text Type</th>
<th>Creation</th>
<th>Indexing</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Length</th>
<th>Slicing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala</td>
<td>a+(4,9)</td>
<td>a(4,9)</td>
<td>a+(4,9)</td>
<td>a(4,9)</td>
<td>a(4,9)</td>
<td>a(4,9)</td>
</tr>
<tr>
<td>Python</td>
<td>s=a[0]</td>
<td>s[0]</td>
<td>s=a[0]</td>
<td>s[0]</td>
<td>s[0]</td>
<td>s[0]</td>
</tr>
<tr>
<td>JavaScript</td>
<td>s=a[0]</td>
<td>s[0]</td>
<td>s=a[0]</td>
<td>s[0]</td>
<td>s[0]</td>
<td>s[0]</td>
</tr>
<tr>
<td>PHP</td>
<td>$a[0]</td>
<td>$a[0]</td>
<td>$a[0]</td>
<td>$a[0]</td>
<td>$a[0]</td>
<td>$a[0]</td>
</tr>
<tr>
<td>Ruby</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
</tr>
<tr>
<td>Bash</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
</tr>
<tr>
<td>VBA</td>
<td>a[0]</td>
<td>a[0]</td>
<td>a[0]</td>
<td>a[0]</td>
<td>a[0]</td>
<td>a[0]</td>
</tr>
</tbody>
</table>

Associative Arrays (“Hash”/“Dict”)

<table>
<thead>
<tr>
<th>String/Text Type</th>
<th>Creation</th>
<th>Indexing</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Length</th>
<th>Slicing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala</td>
<td>a+(4,9)</td>
<td>a(4,9)</td>
<td>a+(4,9)</td>
<td>a(4,9)</td>
<td>a(4,9)</td>
<td>a(4,9)</td>
</tr>
<tr>
<td>Python</td>
<td>s=a[0]</td>
<td>s[0]</td>
<td>s=a[0]</td>
<td>s[0]</td>
<td>s[0]</td>
<td>s[0]</td>
</tr>
<tr>
<td>JavaScript</td>
<td>s=a[0]</td>
<td>s[0]</td>
<td>s=a[0]</td>
<td>s[0]</td>
<td>s[0]</td>
<td>s[0]</td>
</tr>
<tr>
<td>PHP</td>
<td>$a[0]</td>
<td>$a[0]</td>
<td>$a[0]</td>
<td>$a[0]</td>
<td>$a[0]</td>
<td>$a[0]</td>
</tr>
<tr>
<td>Ruby</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
</tr>
<tr>
<td>Bash</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
<td>@a[0]</td>
</tr>
<tr>
<td>VBA</td>
<td>a[0]</td>
<td>a[0]</td>
<td>a[0]</td>
<td>a[0]</td>
<td>a[0]</td>
<td>a[0]</td>
</tr>
</tbody>
</table>

Perl

Array and Hash Gotchas

- Subscripts use scalar sigil
  - `@a = (4,9)` # create array @a
  - print @a[0] # index into array @a, result 4
  - print @a # error: undefined variable @a
- Contents must be scalars
  - Can't directly nest arrays/hashe
  - Work-around: use references, they are scalars
  - `@a = (4,9)` # create array @a
  - `$z = [2,7]` # create reference $z to array
  - `@m = ([@a], $z)` # create array @m with two refs
  - print @m[1][0] # same as $z->[0], result 2

Data Structures

```
$recipe = {
    qty => 2,
    unit => "tbsp",
    ing => "flour",
};
$recipe = {
    qty => 4,
    unit => "oz",
    ing => "sugar",
};
$recipe = {
    qty => 3,
    unit => "tbsp",
    ing => "milk",
};
```

© Martin Hirzel
**List Comprehensions**

- Python calls them "generator expressions"
- Concise syntax for generating lists:
  ```python
  listCompreh = [exp for forClause compClause]
  forClause ::= for id in exp
  compClause ::= for Clause ifClause
  ifClause ::= if exp
  ```
- Example:
  ```ruby
  l = [1, 2, 3, 4]
  t = 'a',
  c2 = [(x, y) for x in l if x < 3 for y in t]
  ```

**Generators**

```python
#SURE/bin/env python
def myGenerator(x):
    print '1st call:'
    yield x
    print '2nd call:'
    yield x
    print '3rd call:'
    yield x
    print '4th call:'
    yield x

myCoroutine = myGenerator(1)
print '1st call to myCoroutine.next()':
print myCoroutine.next() #10
print '2nd call to myCoroutine.next()':
print myCoroutine.next() #11
print '3rd call to myCoroutine.next()':
print myCoroutine.next() #12
print '4th call to myCoroutine.next()':
print myCoroutine.next() #13
```

Python can also treat a generator result as an iterator:
```ruby
for y in myGenerator(1):
    print y
```

---

**Writing Subroutines**

<table>
<thead>
<tr>
<th>Positional parameters</th>
<th>Named parameters</th>
<th>Default values</th>
<th>Call-by-value</th>
<th>Call-by-reference</th>
<th>Nested subroutines</th>
<th>Subroutine references</th>
<th>Anonymous subroutines</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

---

**Outline**

- How to Describe a Language
- Dynamic Typing
- Text Processing
- Data Manipulation
- Closures and Objects
- Web Programming
- Security
- Debugging

---

**Concepts**

**Closures**

- Closure = subroutine reference + environment
- Motivation: keep state, e.g., for AJAX call-back
- Subroutine reference may be named or anonymous
- Environment = variable bindings from old scope chain

---

**Example in JavaScript**

```javascript
// print 1
var f = function() { return 1; };
console.log(f());
```

**Example in Python**

```python
# closure = function() { return 1; }
```

---

© Martin Hirzel

© Martin Hirzel

© Martin Hirzel
Emulating Objects using Closures

```python
#!/usr/bin/env python
def stack():
    arr = [] # empty list
    def push(x):
        arr.append(x)
    def pop():
        return arr.pop()
    return (push, pop)
(s_push, s_pop) = stack()
s_push("apple")
s_push("pear")
print(s_pop) # pear
print(s_pop) # apple
```

Java does the opposite: emulating closures using objects (e.g., java.util.Arrays.sort(Object[]a, Comparator c))

Using (Real) Objects

```vba
Dim a1 As Apple
Dim a2 As Apple
Set a1 = New Apple
Set a2 = New Apple
a1.color = "green" ' set property, differently
a2.color = "red" ' for a1 and a2
a1.prepare("slice") ' call method, passing
a2.prepare("squeeze") ' string parameter
```

Properties vs. Fields

- Both: dot notation look&feel
  - Writable: a1.color = "red"
  - Readable: Debug.print a1.color
- Properties only: associated getter/setter behavior
  - E.g., update graphical representation
- Realization:
  - Perl: _get__/_set_
  - Python: __get__/__set__ (new-style classes)
  - Ruby: private fields, mandatory getter/setter
  - JavaScript: _defineGetter_, _defineSetter_
  - VBA: Property Get/Let/Set

Progressive Disclosure

- Many users only use objects, but do not know how to define classes
  - Sufficient for small script using library/framework
- Of those users who define classes, many don’t know how to define properties
  - Sufficient for large script or small library/framework
- Language design encourages this:
  - Learn subset of language to do most important tasks
  - Learn more to do more
- **Progressive disclosure** user experience
  - Disclose (reveal) progressively (step by step)

Prototypes Example

```javascript
function Fruit(weight) {
    this.weight = weight;
}
Fruit.prototype.pluck = function() {
    return 'fruit(' + this.weight + 'g)';
}

function Apple(weight, color) {
    this.weight = weight;
    this.color = color;
}
Apple.prototype = new Fruit();
delete Apple.prototype.weight;
Apple.prototype.constructor = Apple;
Apple.prototype.pluck = function() {
    return this.color + ' apple';
}

a = new Apple(150, "green");
document.write(a.pluck());
```

Outline

- How to Describe a Language
- Dynamic Typing
- Text Processing
- Data Manipulation
- Closures and Objects
- Web Programming
- Security
- Debugging
**HTML (HyperText Markup Language)**

- **Comment** <!-- … -->
- **Element** `<table>…</table>`
- **Attribute** `border=5`
- **Contents** (between matching tags)
- **Empty element** `<hr>`
- **End tag** `</table>`

**HTTP (HyperText Transfer Protocol)**

- Run PHP, which
  creates HTML
  with
  result

**Embedding Code in Web Pages**

- Server replaces PHP code by its own output, browser does not see it sent:
  - XML style (preferred!): `<?php ... ?>`
  - SGML style: `<? ... ?>`
  - SGML style (implicit echo): `<?=` ...
- User can trigger new page load
- Browser runs JavaScript while parsing HTML, user does not see it rendered:
  - Script style `<script>...</script>`
  - User can trigger JavaScript event handler
  - JavaScript can trigger new page load

**AJAX (Asynchronous JavaScript And XML)**

- User can trigger JavaScript event handler
- JavaScript can trigger new page load

**Scope in Server-Side Scripts**

- Application
  - `$_COOKIE`
  - `$_SESSION`
- Session
  - `$_SESSION` variables
- Request
  - `$_GET`, `$_POST`
- Global variables
  - `$g`
Relational Databases

- Database = collection of tables
- Table = relation = set of rows w/ same columns
- Row = tuple = one value per column
- Column = attribute = name+primitive type
- Only store primitive values, never nest tables

<table>
<thead>
<tr>
<th>Relig</th>
<th>Ing</th>
<th>qty</th>
<th>unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>cake</td>
<td>flour</td>
<td>2.5</td>
<td>cup</td>
</tr>
<tr>
<td>cake</td>
<td>milk</td>
<td>3</td>
<td>tbsp</td>
</tr>
<tr>
<td>gravy</td>
<td>salt</td>
<td>2</td>
<td>tsp</td>
</tr>
<tr>
<td>gravy</td>
<td>sugar</td>
<td>10</td>
<td>oz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Volume</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>cup</td>
<td>lb</td>
</tr>
<tr>
<td>tbsp</td>
<td>oz</td>
</tr>
<tr>
<td>tsp</td>
<td>g</td>
</tr>
</tbody>
</table>

SQL Injection

- Untrusted input, may be malicious:
  - Drop Table y;

Security breach: e.g., database damaged

XSS (Cross-Site Scripting)

- Untrusted input, may be malicious:
  - name=<script>…</script>

Security breach: e.g., victim’s password sent back to attacker

Downgrading

- Information-security policy can establish that:
  - Certain parts of secret information can be declassified and revealed to public listeners. E.g.:
    - Last 4 digits of SSN can be revealed to bank teller
    - Result of a password check can be revealed to anyone
  - Certain parts of untrusted input can be endorsed and used in certain trusted computations. E.g.:
    - Untrusted user input can be used in web application if it is properly formatted ("sanitized")

<table>
<thead>
<tr>
<th>Integrity</th>
<th>Confidentiality</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Secret</td>
</tr>
<tr>
<td>Low</td>
<td>Public</td>
</tr>
<tr>
<td>Downgrading</td>
<td>Endorsement</td>
</tr>
</tbody>
</table>

Outline

- How to Describe a Language
- Dynamic Typing
- Text Processing
- Data Manipulation
- Closures and Objects
- Web Programming
- Security
- Debugging
Defects, Infections, Failures

- **Sane State**: The correct state of the system.
- **Infection**: A state that is incorrect but can become correct under certain circumstances.
- **Failure**: An incorrect state that is observed by the user.

Zeller avoids the word "bug", since it could mean any of the above.

Defect = wrong code that turns sane state into infection
Infection spreads.
Failure = wrong output observed by user

Space

The Scientific Method

- **Hypothesis**: E.g., from looking at code
- **Experiment**: E.g., run with certain input
- **Observation**: E.g., using print statement
- **Conclusion**: E.g., step of infection chain

Repeat

Delta Debugging Example

```perl
sub sort_ref_to_array { # buggy!
  my @sorted = sort @{$_[0]};
  return $sorted[0];
}

sub test_sort {
  my @arrayref = sort_ref_to_array($@[0]);
  for (my $i=0; $i+1 < @$arrayref; $i++) {
    if ($arrayref->[0] > $arrayref->[1]) {
      return 'fail';
    } else {
      return 'pass';
    }
  }
  return 'pass';
}

our $min = ddmin([1,3,5,2,4,6], &test_sort);
print "minimized to ", @min, "\n";
```

Software Vise

- Vise holds an artifact firm for working on it
- Perl makes it easy to build vise for batch application
- VBA allows you to build vise for GUI application

Last Slide

- hw10 due tomorrow = Friday 1 August
- Final exam Thursday 7 August
  - During regular class hours = 6pm-8:20pm
  - In the regular class room = CIWW 109
- Now: Course evaluations
- Today’s lecture • Next week
  - Review • Final exam