Objectives

• You will learn/review:
  – Subsets of Java and Python...
  – That are appropriate for COS 333...
  – Through example programs
    • Example 1 in Java, Python
    • Example 2 in Java, Python
    • …
Agenda

• How do I do object-oriented programming?
• How do I implement “out” parameters?
“Fraction” Programs

• The job:
  – Define a general-purpose “Fraction” (alias “Rational”) module
  – Test it (minimally)
“Fraction” in Java

• See **Fraction.java**
  - Serializable **interface**
    • “Tagging” interface
    • Fraction objects can be serialized
  - **Comparable<Fraction>** **interface**
    • Fraction must define `compareTo(Fraction)` method
    • Fraction objects can be sorted by `Arrays.sort()`, `Collections.sort()`
“Fraction” in Java

– Private fields
– Constructors
– Method overloading (of constructors)
– Constructor chaining
– Private vs. public methods
– Overriding of methods inherited from Object
  • toString(), equals(), hashCode()
– Object creation via new operator
– Definition of main() method for unit testing
“Fraction” in Python

• See `frac1.py`
  – Inheritance from `object` class
  – Constructor: `__init__()`
    • Creation of fields within constructor
  – Explicit `self` parameter
    • Versus Java implicit `this` parameter
  – Default arguments
    • Versus Java overloaded methods
  – *Lack of* private vs. public members
    • Use of underscore to suggest private
Aside: Python Class Dangers

• Consider this client code:
  – `f1._num = 27`
    • Allowed
    • Corrupts f1
    • Lack of “private” is dangerous
  – `f1.num = 28  # Typo`
    • Allowed
    • f1 now contains three fields!!!
    • Lack of variable/field declarations is dangerous
“Fraction” in Python

• See **frac2.py**
  – Operator overloading
    • `__str__()` method
      – Inherited from class `object`
      – `str(f)` is same as `f.__str__()`
      – Implicitly called by `print` function
    • `__hash__()` method
      – Inherited from class `object`
      – `hash(frac1)` is same as `frac1.__hash__()`
      – Frac objects can be hashed
      – Frac objects reasonably can be keys in a hash table
“Fraction” in Python

– Operator overloading (cont)
  • `__eq__()` method
    – `f1 == f2` is same as `f1.__eq__(f2)`
  • `__ne__()` method
    – `f1 != f2` is same as `f1.__ne__(f2)`
  • `__lt__()` method
    – `f1 < f2` is same as `f1.__lt__(f2)`
  • And so forth
“Fraction” in Python

– Operator overloading (cont)
  • __neg__() method
    – \(-f1\) same as \(f1.__neg__()\)
  • __add__() method
    – \(frac1 + frac2\) same as \(frac1.__add__(frac2)\)
  • And so forth

• Very many operators can be overloaded
Aside: Object Ref & Object Equality

• See `Equality1.java`

```
s1 == s2      => false
s1.equals(s2) => true
```

```java
String s1 = "hello world";
String s2 = new String("hello world");
```
Aside: Object Ref & Object Equality

• In Java:
  – Strings are objects
  – s1 == s2
    • Tests object references for equality
    • Or, if you prefer…
    • Tests objects for identity
  – s1.equals(s2)
    • Tests objects for equality
Aside: Object Ref & Object Equality

• See `equality1.py`

```
s1 = "hello world"
s2 = "hello world"

s1 is s2                  => false
s1 == s2 => s1.__eq__(s2) => true
```

STACK                                            HEAP

```
s1 is s2            => false
s1 == s2 => s1.__eq__(s2) => true
```
Aside: Object Ref & Object Equality

• In Python:
  – Strings are objects
  – s1 is s2
    • Tests object references for equality
    • Or, if you prefer…
    • Tests objects for identity
  – s1 == s2
    • Abbreviation for s1.__eq__(s2)
    • Tests objects for equality
Aside: Object Ref & Object Equality

• See **Equality2.java**

<table>
<thead>
<tr>
<th>d2</th>
<th>123.45</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>123.45</td>
</tr>
</tbody>
</table>

STACK

d1 == d2 => true
d1.equals(d2)
Aside: Object Ref & Object Equality

• In Java:
  – Doubles are values, not objects
  – \( d_1 == d_2 \)
    • Tests values for equality
    \( d_1.equals(d_2) \)
    • Illegal
    • \( d_1 \) is not an object, so cannot receive messages
Aside: Object Ref & Object Equality

• In Java:
  – Bytes, chars, shorts, ints, longs, floats, and booleans are not objects

• Java is a hybrid object-oriented language
Aside: Object Ref & Object Equality

- See `equality2.py`

```
f1 is f2  => false
f1 == f2 => f1.__eq__(f2) => true
```
Aside: Object Ref & Object Equality

• In Python:
  – Floats are objects
  – f1 is f2
    • Tests object references for equality
    • Or, if you prefer…
    • Tests objects for identity
  – f1 == f2
    • Abbreviation for s1.__eq__(s2)
    • Tests objects for equality
Aside: Object Ref & Object Equality

• In Python
  – ints and bools also are objects

• *Python is a* **pure** *object-oriented language*
## Aside: Object Ref & Object Equality

### Java:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Compares</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x == y</code></td>
<td>Values</td>
</tr>
<tr>
<td></td>
<td>Object references</td>
</tr>
<tr>
<td><code>x.equals(y)</code></td>
<td>Objects</td>
</tr>
</tbody>
</table>

### Python:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Compares</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x is y</code></td>
<td>Object references</td>
</tr>
<tr>
<td><code>x == y</code></td>
<td>Objects</td>
</tr>
<tr>
<td><code>x.__eq__(y)</code></td>
<td></td>
</tr>
</tbody>
</table>
Agenda

• How do I do object-oriented programming?
• How do I implement “out” parameters?
“DivMod” Programs

• The job:
  – Define a function/method that accepts a dividend and a divisor, and must “return” a quotient and a remainder
  – Test the function/method (minimally)
“DivMod” in Java

See TestDivMod1Bad.java

(1) Before call

<table>
<thead>
<tr>
<th>rem</th>
<th>quo</th>
<th>STACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>???</td>
<td>???</td>
<td></td>
</tr>
</tbody>
</table>

(2) After call

<table>
<thead>
<tr>
<th>remainder</th>
<th>divisor</th>
<th>dividend</th>
<th>rem</th>
<th>quo</th>
</tr>
</thead>
<tbody>
<tr>
<td>???</td>
<td>3</td>
<td>11</td>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>

(3) Before return

<table>
<thead>
<tr>
<th>remainder</th>
<th>divisor</th>
<th>dividend</th>
<th>rem</th>
<th>quo</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>11</td>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>

(4) After return

<table>
<thead>
<tr>
<th>rem</th>
<th>quo</th>
</tr>
</thead>
<tbody>
<tr>
<td>???</td>
<td>3</td>
</tr>
</tbody>
</table>
“DivMod” in Java

• Notes on TestDivMod1Bad.java
  – `int` is passed by value
  – Fails
“DivMod” in Java

See TestDivMod2.java

(1) Before call

(2) After call

(3) Before return

(4) After return

See TestDivMod2.java
“DivMod” in Java

• Notes on TestDivMod2.java:
  – Custom wrapper class
  – Objects are passed by reference
  – Object references are passed by value
  – Works, but bulky
“DivMod” in Java

See TestDivMod3Bad.java

(1) Before call

STACK

rem
quo

HEAP

0

???

(2) After call

STACK

remainder
divisor
dividend

rem
quo

HEAP

3

11

0

???

(3) Before return

STACK

remainder
divisor
dividend

rem
quo

HEAP

???

(4) After return

Immutable!!!
“DivMod” in Java

• Notes on TestDivMod3Bad.java
  – Fails
  – Objects of standard Integer wrapper class are immutable
  • Can be keys in a hash table
  • Can’t be used to accomplish call-by-reference
  • Same for Long, Float, Double, …
“DivMod” in Java

See TestDivMod4.java

(1) Before call

(2) After call

(3) Before return

(4) After return
“DivMod” in Java

• Notes on TestDivMod4.java
  – Works
  – But only because the two items to be returned are of the same type
“DivMod” in Python

See testdivmod1bad.py

(1) Before call
- STACK: rem, quo
- HEAP: 0

(2) After call
- STACK: remainder, divisor, dividend, rem, quo
- HEAP: 3, 11, 0

(3) Before return
- STACK: remainder, divisor, dividend, rem, quo
- HEAP: 2, 3, 11, 0

(4) After return
- STACK: rem, quo
- HEAP: 0, 3
“DivMod” in Python

• Notes on testdivmod1bad.py
  – Fails
  – Object references are passed by value

• Incidentally...
  – int objects with values -5 through 255 are created at Python startup
“DivMod” in Python

See `testdivmod2.py`

(1) Before call

STACK

rem

quo

0

0

HEAP

(2) After call

STACK

remainder

divisor

dividend

rem

quo

HEAP

3

11

0

0

(3) Before return

STACK

remainder

divisor

dividend

rem

quo

HEAP

2

11

2

0

(4) After return

STACK

rem

quo

HEAP

2

3
“DivMod” in Python

• Notes on testdivmod2.py
  – Works, but bulky
“DivMod” in Python

See `testdivmod3.py`

(1) Before call

(2) After call

(3) Before return

(4) After return
“DivMod” in Python

• Notes on testdivmod3bad.py
  – Works
  – *Tuple*: an immutable heterogeneous array
“DivMod” in Python

See testdivmod4.py

(1) Before call

(2) After call

(3) Before return

(4) After return

STACK

HEAP

STACK

HEAP

STACK

HEAP

STACK

HEAP

STACK

HEAP

STACK

HEAP

STACK

HEAP
“DivMod” in Python

• Notes on testdivmod4.py
  – Works
  – Same as testdivmod3.py, except…
  – A tuple is **iterable**
    • Caller can use unpacking assignment statement
“DivMod” in Python

• Generalizing
  – Examples of tuples:
    • (1, 'hi', 2.3, False, None)
    • (1,)
  – Non-example of a tuple
    • (1)
Summary

• We have covered the answers to these questions for Java and Python:
  – How do I do object-oriented programming?
  – How do I implement “out” parameters?
Appendix 2:
C Programming
“Fraction” in C

• See `fraction.h`
  – Definition of opaque pointer type
    • `Fraction` is an alias for `struct Fraction *`!
  – Declarations of functions
“Fraction” in C

- See `fraction.c`
  - Structure type definition
  - Function definitions; some `static`
  - `Fraction_print()` function
  - Parameter validation via `assert()`
  - Functions do not create result objects
  - Conditional def of `main()` for unit testing
“Fraction” in C

• Generalizing
  – Any function should be either:
    • **Declared** in module interface and defined **non-static**
    • **Not declared** in module interface and defined **static**
  – Requirement to explicitly free objects dramatically affects program design
  – Parameter validation...
C Parameter Validation

• “Private” (static) function
  – Validate parameters via assert
  – Enabled by default
  – Can disable at preprocess-time

  • $ gcc -D NDEBUG fraction.c readnum.c
    -o fraction
C Parameter Validation

• “Public” (non-static) function
  – Validate parameters via:
    • `assert()`?
      – Can disable
      – Could crash program!
    • `if` statement and return value?
      – What if return value has other use?
      – What if caller doesn't check?
  – No good choices
“DivMod” in C

- See `testdivmod1bad.c`
  - Fails!
  - `int` is passed by value

- See `testdivmod2.c`
  - Works
  - Uses call-by-reference via pointers