Programming Languages and Techniques (CIS120e)

Lecture 21
Oct 29, 2010

Transition to Java III

More on Static vs. Dynamic Types

Review: Static vs. Dynamic Types

• The **dynamic type** of an object is the class that it was created from.

• The **static type** of a variable is an object type (class or interface) that describes what objects can be stored in that variable.

• Similarly, the **static type** of an expression is an object type that describes what we know *just from the text of the program* (without imagining running it) about the possible results of evaluating this expression.

Static vs. Dynamic

• In OCaml, there is no useful distinction between static and dynamic types

• In Java there is (because of subtyping)...
  – The dynamic type of a variable or expression will always be a *subtype* of its static type
```java
public Shape asShape (Shape s) { return s; }

Point p = new Point (5.0, 5.0);
Circle c = new Circle (0.0, 0.0, 10.0);
Shape s1 = p; // A
Shape s2 = c; // B
s2 = p; // C

/* What is the static type of s1 on line A?
What is the dynamic type of s1 when execution reaches A?
What is the static type of s2 on line B?
What is the dynamic type of s2 when execution reaches B?
What is the static type of s2 on line C?
What is the dynamic type of s2 when execution reaches C?
What is the static type of "asShape(p)"?
What is the dynamic type of its result?
What is the static type of "asShape(s1)"?
What is the dynamic type of its result?
Is the assignment "s1 = c" well typed?
Is the assignment "s1 = s2" well typed?
Is the assignment "p = c" well typed?
Is the assignment "p = s1" well typed? */
```

---

**Shapes in Java**

```java
interface Shape {
    public void move (double dx, double dy);
    public double getArea ();
}

class Point implements Shape {
    private double x;
    private double y;
    public Point (double initX, double initY) {
        x = initX; y = initY;
    }
    public void move (double dx, double dy) {
        x = x + dx; y = y + dy;
    }
    public double getArea () {
        return 0.0;
    }
}

class Circle implements Shape {
    private double x;
    private double y;
    private double r;
    public Circle (double initX, double initY, double initR) {
        x = initX; y = initY; r = initR;
    }
    public void move (double dx, double dy) {
        x = x + dx; y = y + dy;
    }
    public double getArea () {
        return 3.14159 * r * r;
    }
}
```

Slightly simplified from what we saw last time:

- just points and circles
- Circle objects have separate x and y fields, rather than a Point as their center
- getArea on a Point is defined to be 0

---

**Shapes in OCaml**

```ocaml
type double = float
(* "float" in OCaml means the same as "double" in Java. *)

type shape =
    | Point of double ref        (* x *)
        * double ref        (* y *)
    | Circle of double ref       (* x *)
        * double ref       (* y *)
        * double ref       (* radius *)

let new_point (initX:double) (initY:double) :
    shape =
    let x = ref initX in
    let y = ref initY in
    Point (x,y)

let new_circle (initX:double) (initY:double) (initR:double) :
    shape =
    let x = ref initX in
    let y = ref initY in
    let r = ref initR in
    Circle (x,y,r)
```

---

**More on Interfaces**
Abstraction

The shape type in both versions gives us a single name for all the possible kinds of shapes. This allows us to write code that manipulates arbitrary “shapes” (by invoking “shape operations”), without caring whether it’s dealing with points or circles.

Java:
```java
class DoStuff {
    public void moveItALot (Shape s) {
        s.move(3.0, 3.0);
        s.move(100.0, 1000.0);
        s.move(1000.0, 234651.0);
    }
    public void dostuff () {
        Shape s1 = new Point(5.0, 5.0);
        Shape s2 = new Circle(0.0, 0.0, 100.0);
        moveItALot(s1);
        moveItALot(s2);
    }
}
```

OCaml:
```ocaml
let moveItALot (s:shape) : unit =
    move s 3.0 3.0;
    move s 100.0 1000.0;
    move s 1000.0 234651.0
let test () =
    let s1 = shape = new_point 5.0 5.0 in
    let s2 = shape = new_circle 0.0 0.0 100.0 in
    moveItALot s1;
    moveItALot s2
```

(imagine that “moveItALot” is 10,000 lines of code...)

Java identifiers

- Variable, class and method names are identifiers
- Alphanumeric characters or _ starting with a letter or _
  ```
  size
  myName
  MILES_PER_GALLON
  A1
  the_end
  ```
- Interpretation depends on context: variables and classes can have the same name

Identifier abuse

Class, instance variable, constructor, and method with the same name...

```java
public class Turtle {
    private Turtle Turtle;
    public Turtle() { }
}
```

```ocaml
public Turtle Turtle (Turtle Turtle) { 
    return Turtle;
}
```
### Primitive Types

- **int** standard integers (32 bits)
- **byte, short, long** other flavors of integers
- **char** characters (unicode)
- **float, double** floating-point numbers
- **boolean** true / false

### Arithmetic Operators

- **=** assignment (can be used in expressions, but don’t!)
- **==** equality
- **!=** inequality
- **>, >=, <, <=** comparisons
- **+** addition (and string concatenation)
- **-** subtraction (and unary minus)
- *** ** multiplication
- **/ ** division
- **% ** remainder (modulus)
- **!** logical “not”
- **& &** logical “and” (short-circuiting)
- **||** logical “or” (short-circuiting)

### Operator Overloading

- The meaning of an operator is determined by what it operates on
  - Integer division
    - $4/3 \equiv 1$
  - Floating point division
    - $4.0/3.0 \equiv 1.3333333333333333$
  - Automatic conversion
    - $4/3.0 \equiv 1.3333333333333333$
- Overloading is a much more general mechanism in Java — we’ll see more of it later

### Strings

- Built-in reference type **String**
- Strings are sequences of characters
  - "Java" "3 Stooges" "富士山" string literals
- + means concatenation
  - "3" + " " + "Stooges" $\Rightarrow$ "3 Stooges"
- Automatic conversion of numbers to strings
  - $3 + " " + "Stooges" \Rightarrow "3 Stooges"
- Text in a **String** is immutable (like OCaml)
**Equality**

- like OCaml, Java has two ways of testing reference types for equality:
  - “pointer equality”
    - `o1 == o2`
  - “deep equality”
    - `o1.equals(o2)`

- `=` is the assignment operator in Java
  - behaves like `:=` in OCaml

- Normally, you should use `==` to compare primitive types and `.equals` to compare objects

---

**Accumulation operators**

- State change and iteration often involve *accumulating* into some variable
  - accumulate interest into a savings account balance
  - accumulate distance traveled into an odometer

- Java (like C and C++) provides concise accumulation statements:
  
  ```
  sum += increment;
  sum -= decrement;
  ```

---

**Increment and Decrement Statements**

```java
count += 1;
```

```java
count++;
```

---

**Increment/Decrement Expressions**

<table>
<thead>
<tr>
<th>this...</th>
<th>means...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x = (++y)*2;</code></td>
<td><code>y += 1; x = y*2;</code></td>
</tr>
<tr>
<td><code>x = (y++)*2;</code></td>
<td><code>x = y*2; y += 1;</code></td>
</tr>
<tr>
<td><code>x = (--y)*2;</code></td>
<td><code>y -= 1; x = y*2;</code></td>
</tr>
<tr>
<td><code>x = (y--)*2;</code></td>
<td><code>x = y*2; y -= 1;</code></td>
</tr>
</tbody>
</table>

Can also be used as expressions. Not recommended — confusing!
### Calling methods

- Calling a method of another object that returns void:
  
  ```java
  o.m();
  ```

- Calling a method of another object that returns a number:
  
  ```java
  System.out.println(o.m() + 5);
  ```

- Calling a method of another object that returns another object:
  
  ```java
  System.out.println(o.m().n());
  System.out.println(o.m().n().x().y().z().a().b().c().d().e());
  ```

- Calling a method of the same object:
  
  ```java
  m();
  m() + 5
  ```

### Iteration

Repeat an action, stopping when some condition is satisfied

```java
while (condition) {
  statements
}
```

```java
while (!bag.isEmpty()) {
  bag.eatMM();
}
```

```java
while (shelf.hasNextBook()) {
  Book book = shelf.getNextBook();
  catalogue.addInfo(book);
  numBooks = numBooks+1;
}
```

### Conditional

```java
public String transport() {
  if (isFrigid())
    return "skis";
  else if (isTropical())
    return "surfboard";
  else
    return "bicycle";
}
```