Object Orientation

Definitions

- Give definitions for the following:
  - Object-oriented language
  - Object-oriented programming
- Then again, why bother?

Observations

- Object-oriented programming is not the same as programming in an object-oriented language
- Object-oriented languages are not all like Java

Outline

- 16.2 Object-oriented programming
  - OO in ML
  - Non-OO in Java
- 16.3 Object-oriented language features
  - Classes
  - Prototypes
  - Inheritance
  - Encapsulation
  - Polymorphism

```java
public class Node {
  private String data;
  private Node link;
  public Node(String theData, Node theLink) {
    data = theData;
    link = theLink;
  }
  public String getData() {
    return data;
  }
  public Node getLink() {
    return link;
  }
}
```

A previous Java example: a node used to build a stack of strings

Node Class

- Two fields, **`data`** and **`link`**
- One constructor that sets **`data`** and **`link`**
- Two methods: **`getData`** and **`getLink`**
- In the abstract, an object takes a message ("get data", "get link") and produces a response (a String or another object)
- An object is a bit like a function of type **`message->response`**

```
datatype message =
    GetData
  | GetLink;

datatype response =
    Data of string
  | Object of message -> response;

fun node data link GetData = Data data
  | node data link GetLink = Object link;
```

Same OO idea in ML.

We have a type for messages and a type for responses.

To construct a node we call **`node`**, passing the first two parameters.

Result is a function of type **`message->response`**.

Node Examples

```
- val n1 = node "Hello" null;
val n1 = fn : message -> response
- val n2 = node "world" n1;
val n2 = fn : message -> response
- n1 GetData;
val it = Data "Hello" : response
- n2 GetData;
val it = Data "world" : response
```

- Objects responding to messages
- **`null`** has to be something of the object type (**`message->response`**); we could use

```
fun null _ = Data "null";
```

Stack Class

- One field, **`top`**
- Three methods: **`hasMore`**, **`add`**, **`remove`**
- Implemented using a linked list of **`node`** objects

```
datatype message =
    IsNull
  | Add of string
  | HasMore
  | Remove
  | GetData
  | GetLink;

datatype response =
    Pred of bool
  | Data of string
  | Removed of (message -> response) * string
  | Object of message -> response;

fun root _ = Pred false;
```

Expanded vocabulary of messages and responses, for both **`node`** and **`stack`**

Root class handles all messages by returning **`Pred false`**

```
fun null IsNull = Pred true
  | null message = root message;

fun node data link GetData = Data data
  | node data link GetLink = Object link
  | node _ _ message = root message;

fun stack top HasMore =
      let val Pred(p) = top IsNull
      in Pred(not p) end
  | stack top (Add data) =
      Object(stack (node data top))
  | stack top Remove =
      let
        val Object(next) = top GetLink
        val Data(data) = top GetData
      in
        Removed(stack next, data)
      end
  | stack _ message = root message;
```

```
- val a = stack null;
val a = fn : message -> response
- val Object(b) = a (Add "the plow.");
val b = fn : message -> response
- val Object(c) = b (Add "forgives ");
val c = fn : message -> response
- val Object(d) = c (Add "The cut worm ");
val d = fn : message -> response
- val Removed(e,s1) = d Remove;
val e = fn : message -> response
val s1 = "The cut worm " : string
- val Removed(f,s2) = e Remove;
val f = fn : message -> response
val s2 = "forgives " : string
- val Removed(_,s3) = f Remove;
val s3 = "the plow." : string
- s1^s2^s3;
val it = "The cut worm forgives the plow." : string
```

Inheritance, Sort Of

- Here is a **`peekableStack`** like the one in Java from Chapter Fifteen:

```
fun peekableStack top Peek = top GetData
  | peekableStack top message = stack top message;
```

- This style is rather like a Smalltalk system
  - Message passing
  - Messages not statically typed
  - Unhandled messages passed back to superclass

Thoughts

- Obviously, not a good way to use ML
  - Messages and responses not properly typed
  - No compile-time checking of whether a given object can handle a given message
- (Objective CAML is a dialect that integrates OO features into ML)
- The point is: it's possible
- OO programming is not the same as programming in an OO language

Outline

- Object-oriented programming
  - OO in ML
  - Non-OO in Java
- Object-oriented language features
  - Classes
  - Prototypes
  - Inheritance
  - Encapsulation
  - Polymorphism

Java

- Java is better than ML at supporting an object-oriented style of programming
- But using Java is no guarantee of object-orientation
  - Can use static methods
  - Can put all code in one big class
  - Can use classes as records—public fields and no methods, like C structures

Classes Used As Records

```
public class Node {
  public String data; // Each node has a String...
  public Node link;   // ...and a link to the next Node
}

public class Stack{
  public Node top;  // The top node in the stack
}
```

A Non-OO Stack

```
public class Main {
  private static void add(Stack s, String data) {
    Node n = new Node();
    n.data = data;
    n.link = s.top;
    s.top = n;
  }
  private static boolean hasMore(Stack s) {
    return (s.top!=null);
  }
  private static String remove(Stack s) {
    Node n = s.top;
    s.top = n.link;
    return n.data;
  }
  …
}
```

Note direct references to public fields—no methods required, data and code completely separate

Polymorphism

- In Chapter Fifteen: **`Worklist`** interface implemented by **`Stack`**, **`Queue`**, etc.
- There is a common trick to support this kind of thing in non-OO solutions
- Each record starts with an element of an enumeration, identifying what kind of **`Worklist`** it is…

A Non-OO Worklist

```
public class Worklist {
  public static final int STACK = 0;
  public static final int QUEUE = 1;
  public static final int PRIORITYQUEUE = 2;
  public int type; // one of the above Worklist types
  public Node front; // front Node in the list
  public Node rear; // unused when type==STACK
  public int length; // unused when type==STACK
}
```

The **`type`** field says what kind of **`Worklist`** it is.

Meanings of other fields depend on **`type`**.

Methods that manipulate **`Worklist`** records must branch on **`type`**…

Branch On Type

```
private static void add(Worklist w, String data) {
   if (w.type==Worklist.STACK) {
     Node n = new Node();
     n.data = data;
     n.link = w.front;
     w.front = n;
   }
   else if (w.type==Worklist.QUEUE) {
     the implementation of add for queues
   }
   else if (w.type==Worklist.PRIORITYQUEUE) {
     the implementation of add for priority queues
   }
 }
```

Every method that operates on a **`Worklist`** will have to repeat this branching pattern

Drawbacks

- Repeating the branching code is tedious and error-prone
- Depending on the language, there may be no way to avoid wasting space if different kinds of records require different fields
- Some common maintenance tasks are hard—like adding a new kind of record

OO Advantages

- When you call an interface method, language system automatically dispatches to the right implementation for the object
- Different implementations of an interface do not have to share fields
- Adding a new class that implements an interface is easy—no need to modify existing code

Thoughts

- OO programming is not the same as programming in an OO language
  - Can be done in a non-OO language
  - Can be avoided in an OO language
- Usually, an OO language and an OO programming style do and should go together
  - You *usually* get a worse ML design by using an OO style
  - You *usually* get a better Java design by using an OO style (hint: avoid enumerations)

Outline

- 16.2 Object-oriented programming
  - OO in ML
  - Non-OO in Java
- 16.3 Object-oriented language features
  - Classes
  - Prototypes
  - Inheritance
  - Encapsulation
  - Polymorphism

Classes

- Most OO languages, including Java, have some kind of class construct
- Classes serve a variety of purposes, depending on the language:
  - Group fields and methods together
  - Are *instantiable*: the running program can create as many objects of a class as it needs
  - Serve as the unit of inheritance: derived class inherits from base class or classes

Classes

- More purposes:
  - Serve as a type: objects (or references to them) can have a class or superclass name as their static type
  - House *static* fields and methods: one per class, not one per instance
  - Serve as a labeled namespace; control the visibility of contents outside the class definition

Without Classes

- Imagine an OO language with no classes
- With classes, you create objects by instantiating a class
- Without classes, you could create an object from scratch by listing all its methods and fields on the spot
- Or, you could clone an existing *prototype* object and then modify parts of it

```
x = new Stack();
```

With classes: instantiation

```
x = {
  private Node top = null;
  public boolean hasMore() {
    return (top!=null);
  }
  public String remove() {
    Node n = top;
    top = n.getLink();
    return n.getData();
  }
  …
}
```

Without classes: raw object creation

```
x = y.clone();
x.top = null;
```

Without classes: prototype cloning

Prototypes

- A *prototype* is an object that is copied to make similar objects
- When making copies, a program can modify the values of fields, and can add or remove fields and methods
- Prototype-based languages (like Self) use this concept instead of classes

Without Classes

- Instantiation is only one use of classes
- Other things prototype-based languages must do without:
  - Classes as types: most prototype-based languages are dynamically typed
  - Inheritance: prototype-based languages use a related dynamic technique called *delegation*

Inheritance

- Simple enough in outline
  - Set up a relationship between two classes: a derived class and a base class
  - Derived class gets things from the base class
- But what a derived class gets from the base class (or classes) depends on the language…

Inheritance Questions

- More than one base class allowed?
  - Single inheritance: Smalltalk, Java
  - Multiple inheritance: C++, CLOS, Eiffel
- Forced to inherit everything?
  - Java: derived class inherits all methods, fields
  - Sather: derived class can rename inherited methods (useful for multiple inheritance), or just undefine them

Inheritance Questions

- Universal base class?
  - A class from which all inherit: Java's **`Object`**
  - No such class: C++
- Specification inherited?
  - Method obligations, as in Java
  - More specification: invariants, as in Eiffel
- Types inherited?
  - Java: all types of the base class

Inheritance Questions

- Overriding, hiding, etc.?
  - Java, roughly (skipping many details):
    - Constructors can access base-class constructors with **`super`**; implicit call of no-arg super constructor
    - New instance method of the same name and type overrides inherited one; overridden one can be called using **`super`**
    - New field or static method hides inherited ones; still accessible using **`super`** or base class static types
- Languages differ considerably

Encapsulation

- Found in virtually all modern programming languages, not just OO ones
- Encapsulated program parts:
  - Present a controlled interface
  - Hide everything else
- In OO languages, objects are encapsulated
- Different languages do it differently

Visibility Of Fields And Methods

- Java: four levels of visibility
  - **`private`**: only within class
  - Default access: throughout package
  - **`protected`**: package + derived classes
  - **`public`**: everywhere
- Some OO languages (Smalltalk, LOOPS, Self) have less control: everything public
- Others have more: in Eiffel, features can be exposed to a specific set of client classes

Polymorphism

- Found in many languages, not just OO ones
- Special variation in many OO languages:
  - When different classes have methods of the same name and type, like a stack class and a queue class that both have an **`add`** method
  - When language permits a call of that method in contexts where the class of the object is not known statically

Example: Java

```
public static void flashoff(Drawable d, int k) {
  for (int i = 0; i < k; i++) {
    d.show(0,0);
    d.hide();
  }
}
```

- Here, **`Drawable`** is an interface
- Class of object referred to by **`d`** is not known at compile time

Dynamic Dispatch

- In Java, static type of the reference may be a superclass or interface of the actual class
- At runtime, the language system must find the right method for the actual class
- That's *dynamic dispatch*: the hidden, implicit branch-on-class to implement method calls
- Optional in C++; always used in Java and most other OO languages

Implementation And Type

- In Java, two mechanisms:
  - A class inherits both types and implementation from its base class
  - A class gets additional types (but no implementation) by implementing interfaces
- Partially separates inheritance of implementation and inheritance of type
- Other OO languages differ in how much they separate these two

Implementation And Type

- In C++, no separation:
  - One mechanism for general inheritance
  - For inheriting type only, you can use an abstract base class with no implementations
- In Sather, complete separation:
  - A class can declare that it *includes* another class, inheriting implementation but not type
  - A class can declare that it is a subclass of an abstract class, inheriting type but not implementation (like Java interfaces)

About Dynamic Typing

- Some OO languages use dynamic typing: Smalltalk, Self
- An object may or may not be able to respond to a particular message—no compile-time check (like our ML trick)
- Total freedom: program can try using any method for any object
- Polymorphism is not relevant here

Conclusion

- Today, a cosmopolitan perspective:
  - Object-oriented programming is not the same as programming in an object-oriented language
  - Object-oriented languages are not all like Java
- There is no single OO programming style or set of OO language features: they are often debated and they are evolving
- Be skeptical of definitions!