Object-Oriented Programming Languages

Read: Scott, Chapter 10.1-10.4
Lecture Outline

- Object-oriented programming
- Encapsulation and inheritance
- Initialization and finalization
- Subtyping and dynamic method binding

- Polymorphism
Benefits of Object Orientation

- **Abstraction**
  - Classes bridge the gap between concepts in the application domain and software
  - E.g., domain concept of *Customer* maps to class *Customer*

- **Encapsulation**
  - Classes provide interface but hide data representation
  - Easier to understand and use
  - Can be changed internally with minimal impact

- **Reuse**
  - Inheritance and composition provide mechanisms for reuse

- **Extensibility**
Encapsulation and Inheritance

- Access control modifiers – public, private, and others
  - What portion of the class is visible to users?
  - Public, protected or private visibility
  - Java: Has package as default; protected is slightly different from C++
  - C++: Has friend classes and functions
  - Smalltalk and Python: all members are public

- With inheritance
  - What control does the superclass have over its fields and methods? There are different choices
  - C++: a subclass can restrict visibility of superclass members
  - C#, Java: a subclass can neither increase nor restrict visibility of superclass members
Initialization and Finalization

- Reference model for variables used in Java, Smalltalk, Python
  - Every variable is a reference to an object
  - Explicit object creation: foo b = new foo();
- Value model for variables used in C++, Modula-3, Ada-95
  - A variable can have a value that is an object
  - Object creation may be implicit: e.g. foo b;
- How are objects destroyed?
Question

Consider the following code:

```cpp
A a; // a is a local variable of type A
a.m(); // We call method m on a
```

What happens in C++?
What happens in Java?
More on Implicit Creation in C++

- C++ requires that an appropriate constructor is called for every object implicitly created on the stack, e.g., `A a;`

- What happens here: `foo a;`
  - Compiler calls zero-argument constructor `foo::foo()`

- What happens here: `foo a(10, 'x');`
  - Calls `foo::foo(int, char)`
More on Implicit Creation in C++

What happens here:

```cpp
foo a;
foo c = a;
```

- Calls `foo::foo()` at `foo a`; calls copy constructor `foo::foo(foo&)` at `foo c = a`;
- `=` operator here stands for initialization, not assignment!
More on Implicit Creation in C++

What happens here:

```cpp
foo a, c;  // declaration

// assignment
```

- Calls `foo::foo()` twice at `foo a, c;`
- calls assignment operator `foo::operator=(foo&)` at `c = a;`
- `= operator here stands for assignment!`
Lecture Outline

- Object Oriented programming
- Encapsulation and inheritance
- Initialization and finalization
- Subtyping and dynamic method binding
- Polymorphism
Subtyping and Dynamic Method Binding

- Subtyping and subtype polymorphism - the ability to use a subclass where a superclass is expected
- Thus, dynamic method binding (also known as dynamic dispatch) - the ability to invoke a new refined method, even in a context where an earlier version is expected
  - E.g., class B is a Java subclass of A
  - A a; … a.m();
Subtyping and Dynamic Method Binding

- Advantages?
- Disadvantages?

- C++: static binding is default, dynamic binding is specified with keyword `virtual`
- Java: dynamic binding is default, static binding is specified with `final`
Benefits of Subtype Polymorphism

- Covered extensively in Principles of Software
- Enables extensibility and reuse
  - E.g., we can extend a type hierarchy with no modification to the client of hierarchy
  - Reuse through inheritance or composition
- Subtype polymorphism enables the Open/closed principle (credited to Bertrand Meyer)
  - Software entities (classes, modules) should be open for extension but closed for modification
Example

- Application draws shapes on screen
- Possible solution in C

```c
enum ShapeType { circle, square };
struct Shape { ShapeType t };
struct Circle
    { ShapeType t; double radius; Point center; };
struct Square
    { ShapeType t; double side; Point topleft; };
```
Example

```c
void DrawAll(struct Shape *list[], int n) {
    int i;
    for (i = 0; i < n; i++) {
        struct Shape *s = list[i];
        switch (s->t) {
            case square: DrawSquare(s); break;
            case circle: DrawCircle(s); break;
        }
    }
}
```

What problems do you see here?
Example

- OO Solution in Java

```java
abstract class Shape { public void draw(); }  
class Circle extends Shape { ... } 
class Square extends Shape { ... } 

void DrawAll(Shape[] list) {
    for (int i=0; i < list.length; i++) {
        Shape s = list[i];
        s.draw();
    }
}
```
Benefits of Subtype Polymorphism

abstract class Shape { public void draw(); }
class Circle extends Shape { ... }
class Square extends Shape { ... }
class Triangle extends Shape { ... }

Extending the Java code requires no changes in \texttt{DrawAll}! Thus, it is closed for modification.

Extending the C code triggers modifications in \texttt{DrawAll} (and throughout the code)!
Benefits of Subtype Polymorphism

- “Science” of software design teaches Design Patterns
- Design patterns promote design for extensibility and reuse
- Nearly all design patterns make use of subtype polymorphism!
Lecture Outline

- Object-oriented programming
- Encapsulation and inheritance
- Initialization and finalization
- Subtyping and dynamic method binding

- Polymorphism
Polymorphism

- Generally, refers to the mechanisms that a programming language provides, to allow for the same piece of code to be used with objects or values of multiple types
- Poly = many and morph = form
- Examples of polymorphism
  - Generic functions in C++ and Haskell
  - Templates in C++, generics in Java
  - Implicitly polymorphic foldl/foldr in Scheme
  - Other
Varieties of Polymorphism

- Subtype polymorphism
  - What we just discussed… Code can use a subclass $B$ where a superclass $A$ is expected
  - Standard in object-oriented languages

- Parametric polymorphism
  - Code takes a type as parameter
  - Explicit parametric polymorphism
  - Implicit parametric polymorphism
  - Standard in functional programming languages

- Ad-hoc polymorphism (overloading)
Explicit Parametric Polymorphism

- Occurs in Ada, Clu, C++, Java, Haskell (type classes)
- There is an explicit type parameter
- Explicit parametric polymorphism is also known as genericity
- E.g. in C++:

```cpp
template<class V>
class list_node {
    list_node<V>* prev;
    ...
};
```

```cpp
template<class V>
class list {
    list_node<V> header;
    ...
};
```
Explicit Parametric Polymorphism

- Usually (but not always!) implemented by creating **multiple copies** of the generic code, one for each concrete type

```c
typedef list_node<int> int_list_node;
typedef list<int> int_list;
```

- Object-oriented languages usually provide both **subtype polymorphism** and **explicit parametric polymorphism**, which is referred to as generics
Generics are tricky...

Consider this C++ code (uses the STL):

```cpp
list<int> l;
sort(l.begin(), l.end());
```

Compiler produces around 2K of text of error messages, referring to code in the STL.

The problem here is that the STL’s `sort` requires a `RandomAccessIterator`, while the `list` container provides only a `Bidirectional Iterator`
On Concepts in C++ and Much More

- Thriving in a Crowded and Changing World: C++ 2006–2020
- By Bjarne Stroustrup

In Java, Bounded Types Restrict Instantiations by Client

- Generic code can perform operations permitted by the bound

class MyList1<E extends Object> {
    void m(E p) {
        p.intValue(); //compile-time error; Object
        //does not have intValue()
    }
}

class MyList2<E extends Number> {
    void m(E p) {
        p.intValue(); //OK. Number has intValue()
    }
}

Programming Languages CSCI 4430, A/ Milanova (modified from example by Michael Ernst)
In Java, Bounded Types Restrict Instantiations by Client

- Instantiations respect the bound

```java
class MyList2<E extends Number> {
    void m(E arg) {
        arg.intValue(); // OK. Number has intValue()
    }
}
MyList2<String> ls = new MyList2<String>(); // compile-time error; String is not within //bounds of E
MyList2<Integer> li = ... // OK. Integer is subtype of Number
```
In Haskell, Type Predicates Restrict Instantiation of Generic Functions

\[
\text{sum :: (Num } a) \Rightarrow a \rightarrow \text{List } a \rightarrow a
\]

\[
\text{sum } n \text{ Nil } = n
\]

\[
\text{sum } n \text{ (Cons } x \text{ xs) } = \text{sum } (n+x) \text{ xs}
\]

- \( a \) is an explicit type parameter
- \( \text{(Num } a) \) is a predicate in type definition
- \( \text{(Num } a) \) constrains the types we can instantiate the generic function with
Implicit Parametric Polymorphism

- Occurs in Scheme, Python and others
- There is no explicit type parameter, yet the code works on many different types

- Usually, there is a single copy of the code, and all type checking is delayed until runtime
  - If the arguments are of type as expected by the code, code works
  - If not, code issues a type error at runtime
Implicit Parametric Polymorphism

twice in Scheme: \((\text{define} \ (\text{twice} \ f \ x) \ (f \ (f \ x)))\)

\((\text{twice} \ (\lambda (x) (+ 1 \ x)) \ 1)\) yields ?

; twice :: (int -> int) -> int -> int

---> \((\lambda (x) (+ 1 \ x)) \ ((\lambda (x) (+ 1 \ x)) \ 1)\)

---> \((\lambda (x) (+ 1 \ x)) \ 2\)

---> yields 3
Implicit Parametric Polymorphism

twice in Scheme: (define (twice f x) (f (f x)))

(twice (lambda (x) (cons ‘a x)) ‘(b c)) yields ?

; twice = ([sym] -> [sym]) -> [sym] -> [sym]

yields (a a b c)
Implicit Parametric Polymorphism

twice in Scheme: `(define (twice f x) (f (f x)))`

`(twice 2 3)` yields ?

--> 2 (2 3)

--> bombs, 2 is not a function value

map, foldl, length are all implicitly parametric
def intersect(seq1, seq2):
    res = []
    for x in seq1:
        if x in seq2:
            res.append(x)
    return res

As long as arguments for seq1 and seq2 are of iterable type, intersect works
Let Polymorphism

- A form of explicit parametric polymorphism
- Occurs in Haskell and in ML
  - Also known as ML-style polymorphism

let f = \x -> x in if (f True) then (f 1) else 0
--- f is a polymorphic function
--- At (f True) instantiates to bool->bool function
--- At (f 1) instantiates to int->int function
Let Polymorphism

\[
\text{let } f = \lambda x \rightarrow x \text{ in if (f True) then (f 1) else 0}
\]

- Informally, let polymorphism restricts polymorphism to functions defined at \texttt{let} bindings
- Disallows functions that take polymorphic functions as arguments
- Formally defined by Hindley Milner system
- Allows for type inference
Let Polymorphism

Let \( f = \lambda x \rightarrow x \) in if \( (f \ True) \) then \( (f \ 1) \) else 0

- Allows for a natural form of type inference
  - Inference “sees” the function definition at let binding before the call (use) of the function
  - Inference “generalizes” the type of the function
  - At each call in let expression body, inference replaces explicit type parameter with fresh var
- Cannot be done with a function argument
Let Polymorphism

- Contrast

(1) let $f = \lambda x . x$ in if (f True) then (f 1) else 0
vs.

(2) ($\lambda f . \text{if (f True) then (f 1) else 0}$) ($\lambda x . x$)

- Let-bound vs. Lambda-bound polymorphism
The End