Reading Assignment

- A Comparative Overview of C# (linked from class web page)

Java 1.5 (Tiger Java)

Recently, Java has been extended to include a variety of improvements, many drawn from functional languages.

Added features include:

- **Parametric polymorphism.**
  Classes and interfaces may be parameterized using a type parameter.
  ```java
class List<T> {
    T head;
    List<T> tail;
}
```
  Interfaces may also be parameterized.

- **Enhanced loop iterators.**
  ```java
  for (v : myArray) {
    // each element of myArray
    // appears as a value of v
  }
  ```

- **Automatic boxing and unboxing of wrapper classes.**
  Conversion from `int` to `Integer` or `Integer` to `int` is now automatic.

- **Typesafe enumerations.**
  ```java
  public enum Color {RED, BLUE, GREEN};
  ```

- **Static imports.**
  You may import all static members of a class and use them without qualification. Thus you may now write `out.println` rather than `System.out.println`.

- **Variable argument methods.**

- **Formatted output using `printf`:**
  ```java
  out.printf("Ans = %3d",a+b);
  ```

C#

C# is Microsoft’s answer to Java. In most ways it is very similar to Java, with some C++ concepts reintroduced and some useful new features.

Similarities to Java include:

- C# is object-based, with all object descended from class `Object`.
- Objects are created from classes using `new`. All objects are heap-allocated and garbage collection is provided.
- All code is placed within methods which must be defined within classes.
- Almost all Java reserved words have C# equivalents (many are identical).
- Classes have single inheritance.
C# generates code for a virtual machine to support cross-platform execution.

- Interfaces are provided to capture functionality common to many classes.
- Exceptions are very similar in form to Java’s.
- Instance and static data within an object must be initialized at point of creation.

C# Improves Upon Some Java Features

- Operators as well as methods can be overloaded:

```csharp
class Point {
    int x, y;
    static Point operator + (Point p1, Point p2) {
        return new Point(p1.x+p2.x, p1.y+p2.y);
    }
}
```

- Switch statements may be indexed by string literals.
- In a switch, fall throughs to the next case are disallowed (if non-empty).
- Goto’s are allowed.
- Virtual methods must be marked.

C# Adds Useful Features

- Events and delegates are included to handle asynchronous actions (like keyboard or mouse actions).
- Properties allow user-defined read and write actions for fields. You can add get and set methods to the definition of a field. For example,

```csharp
class Customer {
    private string name;
    public string Name {
        get { return name; }
    }
}
Customer c; ...
string s = c.Name;
```
Indexers allow objects other than arrays to be indexed. The [] operator is overloadable. This allows you to define the meaning of obj[123] or obj["abc"] within any class definition.

Collection classes may be directly enumerated:
foreach (int i in array) ...

Fields, methods and constructors may be defined within a struct as well as a class. Structs are allocated within the stack instead of the heap, and are passed by value. For example:

```csharp
struct Point {
    int x, y;
    void reset () {
        x=0; y=0; }
}
```

When an object is needed, a primitive (int, char, etc.) or a struct will be automatically boxed or unboxed without explicit use of a wrapper class (like Integer or Character). Thus if method List.add expects an object, you may write
```
List.add(123);
```
and 123 will be boxed into an Integer object automatically.

Enumerations are provided:
```
enum Color {Red, Blue, Green};
```

Rectangular arrays are provided:
```
int [,] multi = new int[5,5];
```

Reference, out and variable-length parameter lists are allowed.

Pointers may be used in methods marked unsafe.

---

**Reading Assignment**

- Pizza Tutorial (linked from class web page)

---

**Pizza**

Pizza is an extension to Java developed in the late 90s by Odersky and Wadler. Pizza shows that many of the best ideas of functional languages can be incorporated into a “mainstream” language, giving it added power and expressability.

Pizza adds to Java:

1. **Parametric Polymorphism**
   Classes can be parameterized with types, allowing the creation of “custom” data types with full compile-time type checking.

2. **First-class Functions**
   Functions can be passed, returned and stored just like other types.
3. Patterns and Value Constructors

Classes can be subdivided into a number of value constructors, and patterns can be used to structure the definition of methods.

Parametric Polymorphism

Java allows a form of polymorphism by defining container classes (lists, stacks, queues, etc.) in terms of values of type Object.

For example, to implement a linked list we might use:

```java
class LinkedList {
    Object value;
    LinkedList next;
    Object head() {return value;}
    LinkedList tail() {return next;}
    LinkedList(Object O) {
        value = O; next = null;
    }
    LinkedList(Object O, LinkedList L) {
        value = O; next = L;
    }
}
```

We use class Object because any object can be assigned to Object (all classes must be a subclass of Object).

Using this class, we can create a linked list of any subtype of Object.

But,

- We can't guarantee that linked lists are type homogeneous (contain only a single type).
- We must unbox Object types back into their "real" types when we extract list values.
- We must use wrapper classes like Integer rather than int (because primitive types like int aren't objects, and aren't subclass of Object).

For example, to use LinkedList to build a linked list of ints we do the following:

```java
LinkedList L =
    new LinkedList(new Integer(123));
int i =
    ((Integer) L.head()).intValue();
```

This is pretty clumsy code. We'd prefer a mechanism that allows us to create a "custom version" of LinkedList, based on the type we want the list to contain.

We can't just call something like LinkedList(int) or LinkedList(Integer) because types can't be passed as parameters.

Parametric polymorphism is the solution. Using this mechanism, we can use type parameters to build a
“custom version” of a class from a general purpose class.
C++ allows this using its template mechanism. Pizza also allows type
parameters.
In both languages, type parameters are enclosed in “angle brackets” (e.g.,
Linkedlist<T> passes T, a type, to the LinkedList class).
In Pizza we have
class LinkedList<T> {
    T value; LinkedList<T> next;
    T head() {return value;}
    LinkedList<T> tail() {
        return next;
    }
    LinkedList(T O) {
        value = O; next = null;
    }
    LinkedList(T O, LinkedList<T> L) {
        value = O; next = L;
    }
}

When linked list objects are created (using new) no type qualifiers are
needed—the type of the constructor's parameters are used. We can create
LinkedList<int> L1 =
    new LinkedList(123);
int i = L1.head();
LinkedList<String> L2 =
    new LinkedList("abc");
String s = L2.head();
LinkedList<LinkedList<int> > L3 =
    new LinkedList(L1);
int j = L3.head().head();

Bounded Polymorphism
In Pizza we can use interfaces to bound the type parameters a class
will accept.
Recall our Compare interface:
interface Compare {
    boolean lessThan(Object o1,
        Object o2);
}
We can specify that a parameterized class will only takes types that
implement Compare:
class LinkedList<T implements
    Compare> { ... }

In fact, we can improve upon how interfaces are defined and used.
Recall that in method lessThan we had to use parameters declared as
type Object to be general enough to match (and accept) any object type.
This leads to clumsy casting (with run-time correctness checks) when
lessThan is implemented for a particular type:
class IntCompare implements Compare {
    public boolean lessThan(Object i1,
        Object i2) {
        return ((Integer)i1).intValue() <
            ((Integer)i2).intValue();
    }
}
Pizza allows us to parameterize class definitions with type parameters, so why not do the same for interfaces? In fact, this is just what Pizza does. We can now define `Compare` as

```java
interface Compare<T> {
    boolean lessThan(T o1, T o2);
}
```

Now we define class `LinkedList` as

```java
class LinkedList<T extends Compare<T>> {
    // ...
}
```

Given this form of interface definition, no casting (from type `Object`) is needed in classes that implement `Compare`:

```java
class IntCompare implements Compare<Integer> {
    public boolean lessThan(Integer i1, Integer i2) {
        return i1.intValue() < i2.intValue();
    }
}
```

First-class Functions in Pizza

In Java, functions are treated as constants that may appear only in classes.

To pass a function as a parameter, you must pass a class that contains that function as a member. For example,

```java
class Fct {
    int f(int i) { return i+1; }
}
class Test {
    static int call(Fct g, int arg) {
        return g.f(arg);
    }
}
```

Changing the value of a function is even nastier. Since you can’t assign to a member function, you have to use subclassing to override an existing definition:

```java
class Fct2 extends Fct {
    int f(int i) { return i+111; }
}
```

Computing new functions during executions is nastier still, as Java doesn’t have any notion of a lambda-term (that builds a new function).

Pizza makes functions first-class, as in ML. You can have function parameters, variables and return values. You can also define new functions within a method.

The notation used to define the type of a function value is

```
(T_1, T_2, \ldots) \rightarrow T_0
```

This says the function will take the list `(T_1, T_2, \ldots)` as its arguments and will return `T_0` as its result.

Thus

```
(int) \rightarrow int
```

represents the type of a method like

```java
int plus1(int i) {return i+1;}
```
The notation used by Java for fixed functions still works. Thus
static int f(int i){return 2*i;};
denotes a function constant, \( f \).

The definition
static (int)->int g = f;
defines a field of type \((\text{int})\rightarrow\text{int}\) named \( g \) that is initialized to the value of \( f \).

The definition
static int call((int)->int f, int i)
{return f(i);};
defines a constant function that takes as parameters a function value of type \((\text{int})\rightarrow\text{int}\) and an \( \text{int} \) value. It calls the function parameter with the \( \text{int} \) parameter and returns the value the function computes.

Pizza also has a notation for anonymous functions (function literals), similar to \( \text{fn} \) in ML and \( \text{lambda} \) in Scheme. The notation
fun \((T_1 \ a_1, T_2 \ a_2, \ldots) \rightarrow T_0 \)
(Body)
defines a nameless function with arguments declared as \( (T_1 \ a_1, T_2 \ a_2, \ldots) \) and a result type of \( T_0 \). The function's body is computed by executing the block \{Body\}.

For example,
static (int)->int compose(
(int)->int f, (int)->int g){
  return fun (int i) \rightarrow int
  {return f(g(i));};
}
defines a method named \( \text{compose} \). It takes as parameters two functions, \( f \) and \( g \), each of type \( (\text{int})\rightarrow\text{int} \).

With function parameters, some familiar functions can be readily programmed:

```java
class Map {
  static int[] map((int)->int f, int[] a){
    int[] ans = new int[a.length];
    for (int i=0; i<a.length; i++)
      ans[i]=f(a[i]);
    return ans;
  }
}
```

The function returns a function as its result. The type of the result is \((\text{int})\rightarrow\text{int}\) and its value is the composition of functions \( f \) and \( g \):
return \( f(g(i)) \);

Thus we can now have a call like
compose(\( f_1, f_2 \))(100)
which computes \( f_1( f_2(100) ) \).
And we can make such operations polymorphic by using parametric polymorphism:

class Map<T> {
    private static T dummy;
    Map(T val) {dummy=val;};
    static T[] map((T)->T f, T[] a) {
        T[] ans = (T[]) a.clone();
        for (int i=0; i<a.length; i++)
            ans[i]=f(a[i]);
        return ans;
    }
}