Parameterized types

ML has **parametric** polymorphic types:
```
'a * 'b -> ('a * 'b) list
```

Java has **subtype** polymorphism:
a variable of type `POINT` can hold values of any subtype of `POINT`

Java does not have parameteric polymorphism,
except for arrays

```
POINT[] pointA = new POINT[2];
pointA[0] = new CartPoint(3,4);        // OK
pointA[1] = new CartPoint3D(3,4,5);    // OK
POINT p1 = pointA[0].add(pointA[1]);   // OK

Vector pointV = new Vector();
pointV.add(new CartPoint(3,4));        // OK
pointV.add(new CartPoint3D(3,4,5));    // OK
POINT p2 = pointV.get(0);              // NOT OK
POINT p3 = (POINT)pointV.get(0);       // OK
```

Both classes and methods would benefit from allowing parametric polymorphism

Parameterized types in Java

Pizza: a Java extension with parameterized types, first-class functions, and ML-like datatypes

GJ (Generic Java): a version of Pizza's parameterized types

- to go into next major version of Java

Example:

```
public class Vector<Elem>
    extends Collection<Elem> {
  protected Elem[] elementData;
  ...
  public Elem get(int i) { ... };
  public void set(int i, Elem data) { ... };
  public void add(Elem data) { ... };
  ...
};

Vector<POINT> pointV = new Vector<POINT>();
pointV.add(new CartPoint(3,4));        // OK
pointV.add(new CartPoint3D(3,4,5));    // OK
POINT p2 = pointV.get(0);              // OK
```

Bounds on type parameters

ML's type parameters (e.g. `'a`) are unconstrained

+ can be instantiated by any type
– values of a type parameter can't have anything "interesting" done to them

Pizza's type parameters can be constrained
to be a subtype of some bound

+ allows interesting operations on values of type parameters

```
public interface Printable {
  public void print();
};
public class PrintableVector
                  <Elem implements Printable>
    extends Vector<Elem> implements Printable {
  public void print() {
    Enumeration e = elements();
    while (e.hasMoreElements()) {
      Elem elem = e.nextElement();
      elem.print();              // OK
    }
  };
};
```

A client

```
// assume String implements Printable too

PrintableVector<String> names =
  new PrintableVector<String>();  // OK
names.add("bob"); ...             // OK
names.print();                    // OK

// assume POINT doesn't implement Printable

PrintableVector<POINT> points =
  new PrintableVector<POINT>();   // NOT OK
```

Eliminating equality types, and more

An ML-style equality type:

```
public interface Eq<Elem> {
  public boolean equals(Elem arg);
};
```

An interface for types that also are ordered:

```
public interface Ord<Elem> extends Eq<Elem> {
  public boolean less_than(Elem arg);
};
```

A way to say `String` is ordered:

```
public class String extends ...
                   implements Ord<String> {
  ...
  public boolean equals(String arg) { ... };
  public boolean less_than(String arg) { ... };
};
```

A binary tree

```
public class
    BinTree<Elem implements Ord<Elem>> {
  protected Elem value;
  protected BinTree<Elem> leftSubtree;
  protected BinTree<Elem> rightSubtree;
  ...
  public void insert(Elem elem) {
    if (elem.equals(value)) return;
    if (elem.less_than(value)) {
      leftSubtree.insert(elem);
    } else {
      rightSubtree.insert(elem);
    }
  };
  public boolean member(Elem elem) {
    if (elem.equals(value)) return true;
    if (elem.less_than(value)) {
      return leftSubtree.member(elem);
    } else {
      return rightSubtree.member(elem);
    }
  };
};
```

Beats ML!