Notes on Types in Java

(notes mostly stolen[1] from the Java/Pizza tutorials on the Pizza web site).

Class Extension in Java:

```
class X extends Superclass
       [implements Interface ...]
```

No multiple inheritance.

Every class (except java.lang.Object) has exactly one superclass.

If no extends clause is given, Object is assumed.

Extension has two effects:

- It creates a subtype
- It includes all declarations of the extended class in the extending class, unless they are overridden.

1. "Lesser artists borrow; great artists steal." Igor Stravinsky. (Quote stolen from the Latex manual.)

Subtyping

If class X extends class Y, X is said to be a subclass of Y.

In this case, an instance of class X can be used whereever an instance of class Y is expected.

```
class X extends Y { ...

X anX;
Y aY;

aY = anX;       // OK;

anX = (X)aY;
// explicit type conversion required.
```

Because of subclassing, the static, declared type of a variable might differ from the dynamic type of the run-time value it contains. More precisely: The dynamic type can be a subtype of the static type.

Typecasts and typetests

- A type conversion `(C)expr` checks whether the dynamic type of expr is (a subtype of) class C. If not, a ClassCastException is thrown.
- A type test `expr instanceof C` tests whether the dynamic type of expr is (a subtype of) class C. The result of this operation is a boolean.

Example:

```
class String {
  ...
  boolean equals(Object other) {
    if (other instanceof String) {
      String that = (String)other;
      if (this.len != that.len) return false;
      for (int i = 0; i < this.len; i++) {
        if (this.chars[i] != that.chars[i])
          return false;

      return true;
    }
}
```


Java Arrays and Subtyping

Subtyping translates to arrays:
if X extends Y then also X[] extends Y[]

This is actually unsound! (remember the contravariant rule ...)

Example:

```
String[] a = {"hello", "world!"};

Object[] b = a;
// OK because of array subtyping rule

b[0] = new Integer(1);
// will throw an ArrayStoreException ...

String s = a[0];
// ... to avoid having to assign an Integer
// to a String
```

Moral: Subtyping should not extend to types with mutable fields.

Interfaces

Interfaces specify supported operations without implementing them.

Componets of interfaces:

- methods
- constants
- but no method implementations, variables.

```
interface Storable {
    int READ = 0;
    int WRITE = 1;

    byte[] get();
    put(byte[] data);

    int lastOp(); // returns READ or WRITE;
}
```


Pizza: A Java Extension

Key additions

- parametric polymorphism
- higher-order functions
- algebraic data types

Parametric Polymorphism

```
class StoreSomething<A> {
        A something;

        StoreSomething(A something) {
            this.something = something;
        }

        void set(A something) {
            this.something = something;
        }

        A get() {
            return something;
        }
}
```


Using StoreSomething

```
StoreSomething<String> a =
        new StoreSomething("I'm a string!");

StoreSomething<int> b =
     new StoreSomething(17+4);

b.set(9);

int i = b.get();

String s = a.get();
```

Pairs

```
public class Pair<A, B> {

    public A fst;
    public B snd;

    public Pair(A fst, B snd) {
        this.fst = fst;
        this.snd = snd;
    }
}
```

Using this class to return two values:

```
Pair<String, int> getValues() {
    return new Pair(aString, anInt);
}
```


Interfaces

All the things we did to classes can also be done to interfaces.

Interface to sets of elements of a parameterised type:

```
interface Set<A> {
      boolean contains(A x);
      Set<A> include(A x);
 }
```


Linked Lists

```
abstract class LinkedList<A> implements Set<A>
{

  public abstract boolean contains(A x);
  public LinkedList<A> insert(A x) {
    if (contains(x)) return this;
    else return new NonEmptyList(x, this);
  }

  public Set<A> include(A x) {
     return insert(x);
    }
}

class EmptyList<A> extends LinkedList<A> {

  public boolean contains(A x) {
    return false;
    }
  }
```



```
class NonEmptyList<A> extends LinkedList<A> {

  private A elem;
  private LinkedList<A> rest;

  public NonEmptyList(A elem,
        LinkedList<A> rest) {
    this.elem = elem;
    this.rest = rest;
  }

  public boolean contains(A x) {
    return ((Object)elem).equals((Object)x)
      || rest.contains(x);
    }
  }
```

List Example

Test whether the first command line argument is repeated in subsequent arguments:

```
public class TestForDuplicates {

  public static void main(String[] args) {
    Set<String> set = new EmptyList();
    for (int i = 1; i < args.length; i++)
        set = set.include(args[i]);
    System.out.println(set.contains(args[0]));
  }
}
```


Bounded Polymorphism - Binary Tree Example

Set up an interface for the tree types:

```
interface Comparable {
    boolean equals(Object x);
    boolean less(Object x);
}
```

Limit the elements of our set to it:

```
interface Set<A implements Comparable> {
    boolean contains(A x);
    Set<A> include(A x);
}
```



```
abstract class Tree<A implements Comparable>
      implements Set<A> {
  public abstract boolean contains(A x);
  public abstract Tree<A> insert (A x);
  public Set<A> include(A x) {
    return insert(x);
    }
  }

class EmptyTree<A implements Comparable>
      extends Tree<A> {
    public boolean contains(A x) {
      return false;
      }
    public Tree<A> insert(A x) {
      return new Branch(x, this, this);
      }
    }
```



```
class Branch<A implements Comparable>
        extends Tree<A> {
  A elem;
  Tree<A> left;
  Tree<A> right;

  public Branch(A elem, Tree<A> left,
       Tree<A> right) {
    this.elem = elem;
    this.left = left;
    this.right = right;
    }

  public boolean contains(A x) {
    if (x.less(elem))
       return left.contains(x);
      else if (elem.less(x))
       return right.contains(x);
    else
    // we assume total ordering, therefore
    // x.equals(elem)
    return true;
    }
```

```
public Tree<A> insert(A x) {
  if (x.less(elem))
    return new Branch(elem,
         left.insert(x), right);
  else if (elem.less(x))
    return new Branch(elem, left,
         right.insert(x));
  else return this;
    }
  }
```



```
class ComparableString implements Comparable {
  private String s;

  ComparableString(String s) {
    this.s = s;
    }

  public boolean less(Object other) {
    if (other instanceof ComparableString)
      return s.compareTo
          (((ComparableString)other).s) < 0;
    else throw new Error("can't compare
         to non-strings");
    }
  }
```


To be able to include argument strings in sets, we need to wrap them in a ComparableString.

The test-for-duplicates program again:

```
public class TestForDuplicates {

public static void main(String[] args) {
  Set<ComparableString> set = new EmptyTree();
  for (int i = 1; i < args.length; i++)
   set.include(new ComparableString(args[i]));
  System.out.println(set.contains(new
       ComparableString(args[0])));
  }
}
```


Using F-bounded Polymorphism

```
interface Comparable<A> {
  boolean less(A x);
}

interface Set<A implements Comparable<A>> {
  boolean contains(A x);
  Set<A> include(A x);
}
```

Now ComparableString can be defined as follows:

```
class ComparableString implements
Comparable<ComparableString> {
  private String s;
  ComparableString(String s) { this.s = s; }
  public boolean less(ComparableString other) {
    return s.compareTo(other.s) < 0;
  }
}
```

Note that the type variable A appears recursively in its own bound in the class definition of Set.

Implementing Parametric Types

Two translations are possible

- The homogeneous translation maps type variables to Objects and inserts appropriate casts.
- The heterogeneous translation creates many instances of a parameterised class, one for each instance type.

The homogeneous translation effectively erases type information.

First Class Functions

The syntax for function types is:

(argtype, ..., argtype) -> resulttype

These are implemented by compiling extra classes with methods that implement the function. (See paper for details.)

Consider the compareTo() method of the Java class String. This method compares two Strings lexically and case sensitve.

Change the behaviour of the sorting by providing the function that compares to Strings:

```
class PrintSortedStrings {
  static void print(String[] args,
       (String, String) -> int compare) {
    ....
    if (compare(args[i], args[j]) > 0)
    ....
    }
}
```



```
int myCompare(String s1, String s2) {
  return s1.compareTo(s2);
}

int myCompareNoCase(String s1, String s2) {
  return s1.toLowerCase().compareTo(
         s2.toLowerCase());
}

String[] myStrings = { "These", "Strings",
"will", "be", "sorted", "!"};

PrintSortedStrings.print(myStrings,
                   myCompare);

PrintSortedStrings.print(myStrings,
                   myCompareNoCase);
```


Anonymous Functions

```
void <A> traverse(Tree<A> t, (A)->void proc) {
  if (t instanceof Branch) {
    Branch b = (Branch)t;
    traverse(b.left, proc);
    proc(b.elem);
    traverse(b.right, proc);
  }
}
```

We can use this method to sum up all elements of a tree of integers:

```
int sum(Tree<int> t) {
  int n = 0;
  traverse(t, fun(int x) -> void { n += x; });
  return n;
}
```

Algebraic Data Types

Can use to avoid complicated abstract class/concrete cases

In Pizzaa class can contain declarations of these forms:

```
case ident(arguments);
case ident;
```

Tree class:

```
class Tree<A> implements Set<A> {
  case Empty;
  case Branch(A elem, Tree<A> left,
          Tree<A> right);
}
```

A tree with three elements:

```
Tree<int> t = Tree.Branch(2,
  Tree.Branch(1, Tree.Empty, Tree.Empty),
  Tree.Branch(3, Tree.Empty, Tree.Empty));
```


Pattern Matching

Complicated:

```
Tree<String> t = ... ;
String s;
if (t instanceof Branch) {
  s = (Branch)t.elem;
} else {
  s = "(empty)";
}
```

Simpler if we use pattern matching.

Pattern matching:

- selects appropriate case
- assigns variables in pattern

Better:

```
class Tree<A> {
  case Empty;
  case Branch(A elem, Tree<A> left,
      Tree<A> right);
  public boolean contains(
       (A, A) -> boolean less, A x) {
    switch (this) {
    case Empty: return false;
    case Branch(A elem, Tree<A> l, Tree<A> r):
      if (less(x, elem))
        return l.contains(less, x);
      else if (less(elem, x))
        return r.contains(less, x);
      else return true;
    }
  }
```



```
public Tree<A> insert(
       (A, A) -> boolean less, A x) {
  switch (this) {
  case Empty: return Branch(x, Empty, Empty);
  case Branch(A elem, Tree<A> l, Tree<A> r):
    if (less(x, elem))
      return Branch(elem, l.insert(less, x),r);
    else if (less(elem, x))
      return Branch(elem, l,
           r.insert(less, x));
    else return this;
    }
  }
}
```

Special case: Enums

By defining a class with only constant subcases we get enumeration types, which are missing in Java(compared to C):

```
class Color {
    case Red;
    case Green;
    case Blue;

    String toString() {
        switch (this) {
        case Red: return "Red";
        case Green: return "Green";
        case Blue: return "Blue";
        }
    }
}
```