

Typing in Java and Python - A Comparative Analysis

Author: Mikael Jansson <tic@dtek.chalmers.se>
Course: Topics in Programming Languages (TDA260), Chalmers University of Technology
Date: January 17th, 2006

Typing in General

There are two main classes of type systems in programming languages, namely monomorphic and polymorphic. The former means that a function can only be applied to values of specific types, whereas the latter means that a function can in some way be applied to values of more than one type. Polymorphism can be divided into two main classes: universal and ad-hoc, where universal can be either parametric or inclusion-based, and ad-hoc can be implemented through overloading or coercion ([Cardelli 85], pp. 4).

Parametric polymorphism means that a function parametrizes its type, such that it could be used for any value if instantiated for that particular type first. Inclusion polymorphism is what object-oriented languages utilize through inheritance: a function that can be applied to a more general type can also be applied to the more specific type (i.e., lower in the inheritance chain).

Ad-hoc polymorphism is implemented either by means of overloading function names, or by coercing values. The name itself means that a function may seem to have universal polymorphism, while in fact it might not hold for all types in a specific type class. In the case of overloading, a function can have more than one different type signature, but is in fact, only syntactic sugar for having the same name for different functions -- an implicit dispatch table, if you may. There is nothing that guarantees the same behaviour for overloaded functions, and they may in fact do the opposite of any other function with the same name. Static typing enforces that a function cannot be applied to a value if there is no function with the correct type signature. Coersion is when there are rules for transforming a type into another type for which there might be a function suitable, e.g. the `+` operator which might only be defined for real numbers, and in order to make an integer addition a coersion to real has to be made.

Java

Versions previous to 1.5 have monomorphic functions, but achieves universal polymorphism through inclusion. In the latest iteration, however, types can be parameterized through the concept of generics. Type variables were also introduced. Furthermore, there

are two kinds of types: primitives (such as integers and booleans) and Objects. The Java class library -- especially Collections -- operate on the primitives' object counterpart, e.g. Integer. In 1.5, there's a new feature, *autoboxing*, to let the compiler deal with the problem.

Generics

Quantification on the type variable:

```
int sum(Collection<Int> xs) {
    int s = 0;
    for (Iterator<Int> i = xs.iterator(); i.hasNext();) {
        s += i.next();}
    return s;}
```

Eliminates unsafe type casting; in previous versions there was no way of being certain a collection would only contain elements of a specific type, short of explicitly iterate over all items and verify their types. This makes code easier to use, as it imposes less pre-conditions on the user of the functions -- types are checked at compile time instead of run time, i.e. by manual type casting.

The type information in generics is only available at compile time, after which the object only has the instantiated type. This for backward compatibility with code not utilizing or knowing about generics; as such, generalized types can be treated (in fact, at run time, are) "raw" Java types. Legacy code can, however, in the case with the above `sum` function, insert non-integers into the collection, and only when executing `sum` will there be a failure with a `ClassCastException`. There exists special Java classes for attaching run-time checks that no invalid values can even be inserted to the collection, which can be used where needed.

Generics in Java can also contain wildcards for universal quantification:

```
int length(Collection<?> xs) {
    int l = 0;
    for (Iterator<Int> i = xs.iterator(); i.hasNext(); i.next()) {
        l += 1;}
    return l;}
```

The quantification (both wildcard and regular types) can have a lower or upper bound, through the use of the `super` and `extends` keywords, where the former imposes a lower bound and the latter an upper bound. We modify the `sum()` function to give an example of `extends`:

```
int sum(Collection<? extends Int> xs) {
    int s = 0;
    for (Iterator<Int> i = xs.iterator(); i.hasNext();) {
        s += i.next();}
    return s;}
```

Lower bounds are useful for e.g. operating on generic data structures. The following function declaration would let the user add an object *x* of type *T* into any queue structure that supports adding a super-type of *T*:

```
int addToQ(T x, Q<? super T> q) {  
    ...  
}
```

Classes and Interfaces

Classes in Java utilize existential quantification in that the public methods of the class represent the general interface, immediately instantiated by the code in the methods themselves. This achieves information hiding, by not exposing all of the actual implementation of the code, allowing for private methods and state to be used internally as helpers.

Interfaces allows for i) inclusion polymorphism and ii) existential quantification as it enables functions to parameterize on the any sub-type of the interface while hiding the the actual type/implementation of the interface. Any given class can extend zero or more classes, but only implement the behaviour of zero or one interface.

Python

Unlike Java, there are no primitives -- everything is an object. Python has classes, but no interfaces. Instead of a static type system, Python utilizes *late binding* for attribute access, causing a run-time exception should the object not support the given operation. In fact, that's a common idiom for checking for the existence of variable names:

```
try:  
    foo  
except NameError, x:  
    # name 'foo' not in use  
    pass
```

In Python (like Java), variables are reference, but the name itself does not have a type attached to it -- all it does is point to an object. Links has to be followed to give the return value. Because of the late binding, Python does not use interfaces, but instead relies on naming conventions, even if there's some work (Zope's `zope.interface`, most notably) to get a more formalized interface. For example, given the following function which counts the occurrence of the literal '1' in a file:

```
def countOnes(f):  
    return len([i for i in f.read() if i == '1'])
```

`lineCount` can be used with *file-like* objects supporting the operation `.read()`. We could therefore define a class that's not a file at all, but still supports the read operation:

```
class RandomSequence:
    def __init__(self, count):
        self.count = count

    def read(self):
        return [str(random.randint(0, 9)) for i in range(self.count)]

rs = RandomSequence(100)
print "Out of 100 numbers,", countOnes(rs), "were ones."
```

Objects can be dynamically altered at run time; adding and removing attributes to objects is a common task in Python. This is form of universal polymorphism, achieved through the *overloading* ad-hoc polymorphism. Magic names are used for supporting operators in the language. For example, the operator + is called `__add__` and can be added to any class definition. Python also supports coercion for certain types, such as addition with integer and real values producing a real value.