Polymorphic (Generic) Programming in Java

We have used the fact that Java classes are arranged as a tree with the built in class Object at the root to write "generic" or polymorphic code such as the following function to find an element in an array:

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length; i++){
      if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

What if we wanted to write a similar program to copy the contents of one array into another? A first attempt could be the following:

```
public void arraycopy (Object[] src, Object[] tgt){
  int i,limit;
  limit = Math.min(src.length,tgt.length); // No overflows!
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

However, notice that it is not enough for the two arguments to be compatible with `Object` for this function to work correctly: we also need the type of the array `tgt` to be a supertype of the array `src` for the assignment to work. In other words, assuming that we have a class `Ticket` with `ETicket` as a subclass, the following call is legal

```
Ticket[] ticketarr = new Ticket[10];
ETicket[] elecarr = new ETicket[10];
...
arraycopy(elecarr,ticketarr);
```

because it is legal to assign `ETicket` objects to variables of type `Ticket`. However, the converse call

```
arraycopy(ticketarr,elecarr);
```

would be flagged as a type error at runtime. In general, it is desirable to catch all type errors at compile time, but this kind of error cannot be detected because we have no mechanism to express dependencies across types in polymorphic code.

Let us look at another situation where our existing mechanism for typing is inadequate. Suppose we want to define a linked list that can hold values of any type.

We could begin with a private class to store a node of the list described as follows:

```
private class Node {
  public Object data;
  public Node next;
  ...
}
```

We can then define our list as follows:

```
public class LinkedList{

  private int size;
  private Node first;

  public LinkedList(){
    first = null;
  }

  public Object head(){  // return data at head of list if nonempty
    Object returnval = null;

    if (first != null){
      returnval = first.data;
      first = first.next;
    }

    return returnval;
  }

  public void insert(Object newdata){
    ...  // code to insert newdata in list
  }

  ...

  private class Node{
    ...
  }

}
```

There are two points to note about this style of implementing a polymorphic list:

1. The implementation loses information about the original type of the element. All data is stored and returned as an `Object`. This means we have to cast the output of head even when it is clear what the return type is, as shown below:

```
LinkedList list = new LinkedList();
Ticket t1,t2;

t1 = new Ticket();
list.insert(t1);
t2 = (Ticket)(list.head());  // head() returns an Object
```

2. This implementation does not capture our intuitive understanding that the underlying type of the list is uniform. There is nothing to stop us from constructing a heterogeneous list as follows:

   ```
   LinkedList list = new LinkedList();
   Ticket t = new Ticket();
   Date d = new Date();
   list.insert(t);
   list.insert(d);
   ...
   ```

   Once again, the bottleneck is our inability to specify dependency across types within the list: we would like to say that nodes can hold any type of data but the types across all nodes in a list are the same.

One way of addressing these deficiencies is to introduce explicit type variables. This has been incorporated in Java relatively recently (version 1.5) under the name "generics".

1 Generics in Java

We can now introduce type parameters for class definitions by using type variables. For instance, here is how we could modify our list definition to fix an arbitrary but uniform type `T` for the data stored in the list.

```
public class LinkedList<T>{

  private int size;
  private Node first;

  ...

  public T head(){
    T returnval;
    ...
    return returnval;
```

```
    }

    public void insert(T newdata){
      ...  // code to insert newdata in list
    }

    ...

    private class Node {
      public T data;
      public Node next;
      ...
    }

  }
```

The parameter `T` in angled brackets denotes a type variable for the class `LinkedList`. An instance of `LinkedList` would have `T` bound to a fixed class: for instance, `LinkedList<Ticket>` or `LinkedList<Date>`. All references to `T` inside the class definition are bound to the value of `T` used when the class was instantiated. Here is an example of how we would define and use the new type of linked list.

```
LinkedList<Ticket> ticketlist = new LinkedList<Ticket>();
LinkedList<Date> datelist = new LinkedList<Date>();
Ticket t = new Ticket();
Date d = new Date();
ticketlist.insert(t);
datelist.insert(d);
...
```

When we invoke `ticketlist.insert(t)`, the type of `t` is compared by the compiler to the instantiation of the type `T` used when defining `ticketlist`. In this case, since `ticketlist` was created as an object of type `LinkedList<Ticket>`, `T` is bound to `Ticket` and so it is legal to invoke `ticketlist.insert(t)` for a `Ticket` object `t`. On the other hand, `ticketlist.insert(d)` for a `Date d` would be a type error. It is important to note that this type compatibility can be checked at compile time. [1]

Notice that all instances of `T` within the parameterized class definition are automatically bound once the class is created with a specific value of `T`. Thus, the return type `T` in the function head and the type `T` for the field data in the private class `Node` both get instantiated correctly to the actual type provided for `T` when the class is instantiated.

[1] We are not constrained to use `T` to denote a type variable. We can use any name we want, but it is helpful to have a standard convention to avoid confusing type variable and normal data variables.

We can also define type parameters for functions within a class. For instance, suppose we want to rewrite `arraycopy` in a very strict form that insists that both the source and target arrays are of the same type. We could specify this as follows:

```
public <T> void arraycopy (T[] src, T[] tgt){
  int i,limit;
  limit = min(src.length,tgt.length); // No overflows!
  for (i = 0; i < limit; i++){
      tgt[i] = src[i];
  }
}
```

The type parameter comes before the return type. Since `T` is substituted uniformly by the same type value, both `src` and `tgt` must be arrays of the same type.

It is important to note the difference between the definitions of `arraycopy` and `head` (from `LinkedList`). In the function `head`, we used a type variable `T` without defining it as part of the function definition. This is because the `T` that appears in `head` is derived from the parameter `T` supplied to the surrounding class. If we write, instead

```
public <T> T head(){
  T returnval;
  ...
  return returnval;
}
```

then the type variable `T` referred to inside `head` is a new, private variable which hides the outer `T` and is completely independent. In other words, we should think of `<T>` like a logical quantifier with a natural scope. If we reuse `<T>` within the scope of another `<T>`, the outer `T` is hidden by the inner `T`.

2 Type dependencies and type variables

For `arraycopy`, it is sufficient for the type of the target array `tgt` to be a supertype of the source array `src`. We can specify that the two arrays are of different types by using multiple type variables, say `S` and `T`. As a first step, we write

```
public <S,T> void arraycopy (S[] src, T[] tgt){
  ...
}
```

This, however, does not define any constraints on the types assigned to `S` and `T`. We want `S` to be a subtype of `T`, so we can make our definition more precise by writing[2]:

[2]In this context, Java does not distinguish between `S` implementing an interface `T` and `S` extending a class `T`. For both cases, we just write `S extends T`.

```
public <S extends T,T> void arraycopy (S[] src, T[] tgt){
  ...
}
```

As another example, suppose we want to relax our typing constraint for lists to allow a list of type T to store elements of any subtype of T. We could then define insert as follows:

```
public class LinkedList<T>{

  ...

  public <S extends T> void insert(S newdata){
    ...  // code to insert newdata in list
  }

}
```

Now we cannot be sure, in general, about the specific type of each element of the list. However, we do still know that all values in the list are compatible with type T. Thus, the definition and type of the function head remain the same as before:

```
  public T head(){
    T returnval;
    ...
    return returnval;
  }
```

3 Type variables and the class hierarchy

For conventional types, we can lift the hierarchy on underlying types to more complex objects. For instance, if S is a subtype of T then S[] is also a subtype of T[]. This is crucial, for instance, to write the generic find function that we began with.

```
public int find (Object[] objarr, Object o){...}
```

If we did not have that S[] is compatible with Object[] for every type S, this function would not work in the polymorphic manner that we expect.

However, consider the following situation, where (as usual) ETicket is a subclass of Ticket.

```
ETicket[] elecarr = new ETicket[10];
Ticket[] ticketarr = elecarr;       // OK. ETicket[] is a
                                    // subtype of Ticket[]
...
ticketarr[5] = new Ticket();        // Not OK.  ticketarr[5]
                                    // refers to an ETicket!
```

This generates a type error at runtime. The compiler cannot find fault with the assignment

```
ticketarr[5] = new Ticket();
```

because `ticketarr` is defined to be of type `Ticket[]`!

With type variables, we can implement polymorphic functions without requiring this troublesome property that the class hierarchy on basic types is extended uniformly to more complex structures built on that type. Thus, for classes defined with type variables, we no longer inherit the hierarchy. For instance, though `ETicket` extends `Ticket`, `LinkedList<ETicket>` does not extend `LinkedList<Ticket>`.

Suppose we want to write a function `printlist` that handles arbitrary lists. We might be tempted to write this as follows:

```
public static void printlist(LinkedList<Object> list){...}
```

However, since the hierarchy on base types is not inherited by `LinkedList`, `LinkedList<Object>` is not the "most general" type of `LinkedList`. Instead, if we want a function that works on all variants of `LinkedList`, we should introduce a type variable:

```
public static <T> void printlist(LinkedList<T> list){...}
```

4 Type variables and reflection

The class `Class` whose objects represent information about Java classes is now a generic class that takes a type parameter. Thus, the type `String` is represented by an object of type `Class<String>`, type `ETicket` is represented by an object of type `Class <ETicket>` etc.

Unfortunately, type variables cannot be used in all contextss where a type name is expected. As we have seen, type variables can be used to specify the types of arguments and return values to functions and also to specify the types of variables within a function or a class. However in expressions such as

```
if (s instanceof Shape){ ... }
```

we cannot replace `Shape` by a type variable and write

```
if (s instanceof T){ ... } // T a type variable
```

This means that the following version of the function `classequal` (see page 83 in the older notes) would not work:

```
public static <S,T> boolean classequal(S o1, T o2){
  return (o1 instanceof T) && (o2 instance of S); // Illegal!
}
```

However, we can fruitfully exploit the fact that `Class` now has a type parameter. Suppose we want to write a reflective function that takes as input a class and creates a new object of that type. In the normal setup, we would have written:

```
public Object createinstance(Class c){
   return c.newInstance();
}
```

The return type of this function is `Object` because we cannot infer any information about the nature of the class `c` which is received as the argument.

In the revised framework, we if `c` is the `Class` object corresponding to a class `T`, then it is actually of type `Class<T>`. Thus, we can rewrite this function as follows:

```
public T createinstance(Class<T> c){
   return c.newInstance();
}
```

Now, notice that we are able to extract the underlying type `T` from `c` and use it to specify a more precise return value, thus avoiding the need to use a cast when calling this function.

One important point about Java's parameterized classes is that they are more sophisticated than macros or templates. It is not correct to think of a definition like

```
public class LinkedList<T>{...}
```

as a skeleton that is instantiated into a "real" Java class each time it is actually used with a concrete value of `T`. There is truly only one class `LinkedList<T>` and all specific instantiations of `LinkedList<T>` have the same underlying object. In other words, if we declare

```
LinkedList<String> stringlist = new LinkedList<String>();
LinkedList<Integer> intlist = new LinkedList<Integer>();
```

then

```
classequal(stringlist,intlist)
```

returns `true`.

This seems to have some unfortunate implications. For instance, suppose we write:

```
LinkedList<String> sl = new LinkedList<String>();
LinkedList<String> newsl = createinstance(sl.getClass());
```

Since all instantiations `LinkedList<T>` are of the same type, the value returned by `sl.getClass()` is of the "generic" type `LinkedList<T>` and the Java compiler is unable to reconcile this with the specific type `LinkedList<String>` defined for `newsl`. (Try this for yourself and see.)