Primitive Java Generic Class

A buffer pool is a data structure that caches records retrieved from a disk file, in order to improve an application's performance.

Typically, the pool stores some sort of data object together with the offset where that record occurs in the disk file.

The data objects could be of just about any type, and we hate to write duplicate classes, so we'd like to write a single class to encapsulate the file offset and the data object:

```
public class BPEntry {

   private Long   Offset;
   private Object Value;   // could be anything!

   public BPEntry(Long offset, Object value) {
      Offset = offset;
      Value  = value;
   }
```

Well... that's flexible... maybe too flexible....

Primitive Java Generic Class

Suppose that we have a file of records that could be represented by `String` objects:

```
. . .
Long   offset = raf.getFilePointer();
String record = raf.readLine();

BPEntry bp = new BPEntry(offset, record);    // OK
```

But suppose we made the following mistake:

```
. . .
BPEntry bp = new BPEntry(offset, offset);    // ??
```

This will compile and run since `Long` is a subtype of `Object`...

... using `Object` eliminates any type-checking at compile time...

Simple Formal Java Generic Class

```
public class BPEntry<T> {

   private Long Offset;
   private T    Value;

   public BPEntry(Long offset, T value) {
      Offset = offset;
      Value  = value;
   }
   ...
```

Now...

```
   ...
   Long   offset = raf.getFilePointer();
   String record = raf.readLine();

   BPEntry<String> bp = new BPEntry(offset, record);    // OK
```

But this will not compile:

```
   ...
   bp = new BPEntry<String>(offset, offset);   // error!!
```

Improved Version?

The current version limits us to using `Long`s for the offset field, but if the file is small then the offset could be an `Integer` (and save memory).

We can improve the flexibility:

```
public class BPEntry<K, T> {

   private K Offset;
   private T Value;

   public BPEntry(K offset, T value) {
      Offset = offset;
      Value  = value;
   }
   . . .
```

But now, a careless user could specify a non-numeric type for the offset:

```
   . . .
   bp = new BPEntry<String, Integer>(record, offset);
```

Final Version

We can place a restriction on the type parameter (a *bound* in Java terms):

```
public class BPEntry<K extends Number, T> {

   private K Offset;
   private T Value;

   public BPEntry(K offset, T value) {
      Offset = offset;
      Value  = value;
   }
   ...
```

Now the user cannot use a non-numeric type for the offset.

But a user could still specify a non-integer type, which would be illogical...

Simple Generic Method

The `contains()` method can be used to search an array holding objects of any type.

```
public static <T> boolean contains( T[] array, T x) {

   for ( T value : array ) {
      if ( x.equals(value) )
         return true;
   }
   return false;
}
```

```
   Integer[] array = new Integer[10];
   for (int pos = 0; pos < 10; pos++) {
      array[pos] = pos * pos;
   }

   if ( contains( array, new Integer(15) ) ) {
      System.out.println("Found value in array.");
   }
   else {
      System.out.println("Could not find value in array.");
   }
```

The Need for Type Bounds

```
public static <T> T findMax( T[] array) {

   int maxIndex = 0;

   for ( int i = 1; i < array.length; i++) {

      if ( array[i].compareTo(array[maxIndex]) > 0 )
         maxIndex = i;
   }

   return array[maxIndex];
}
```

```
D:\Code\Generics>javac exFindMax1.java
exFindMax1.java:20: cannot find symbol
symbol  : method compareTo(T)
location: class java.lang.Object
         if ( array[i].compareTo(array[maxIndex]) > 0 )
                      ^
1 error
```

Problem:

There is no way for the Java compiler to know that the generic type `T` will represent an actual type that implements the method `compareTo()` used in the test within the loop.

So, this will not do…

Applying a Type Bound

```
public static <T extends Comparable<T> > T findMax( T[] array) {

   int maxIndex = 0;

   for ( int i = 1; i < array.length; i++) {

      if ( array[i].compareTo(array[maxIndex]) > 0 )
         maxIndex = i;
   }

   return array[maxIndex];
}
```

This restricts the type parameter `T` to be a type that implements the interface `Comparable<T>`, guaranteeing that the call to `compareTo()` is valid.

Problem with the Fix

```
public static <T extends Comparable<T> > T findMax( T[] array) {

   int maxIndex = 0;

   for ( int i = 1; i < array.length; i++) {

      if ( array[i].compareTo(array[maxIndex]) > 0 )
         maxIndex = i;
   }

   return array[maxIndex];
}
```

Problem:

Suppose that `Shape` implements `Comparable<Shape>`, and that `Square` extends `Shape`, so that we know `Square` implements `Comparable<Shape>`.

Then `Square` would not satisfy the condition used above, even though the necessary method is, in fact, available.

So, this will not do… in all cases…

A Better Fix

```
public static <T extends Comparable<? super T> > T findMax( T[] array) {

   int maxIndex = 0;

   for ( int i = 1; i < array.length; i++) {

      if ( array[i].compareTo(array[maxIndex]) > 0 )
         maxIndex = i;
   }

   return array[maxIndex];
}
```

We need a restriction that allows `T` to be derived from a superclass that itself implements `Comparable()`…

The bound used here does so…

The Fix Explained

```
public static <T extends Comparable<? super T> > T findMax( T[] array) {

   . . .
}
```

Wildcards:

The symbol `'?'` is a *wildcard*.

A wildcard represents an arbitrary class, and is followed by a restriction.

In this case, the restriction is that the arbitrary class must be a superclass of `T`.

So, this says that `T` must extend a base class `X` which is-a `Comparable<X>`.

So, `T` is-a `Comparable<X>`.

So, `T` implements the required method and all is well.

Limitation: Type Erasure

The compiler translates generic and parameterized types by a technique called *type erasure*.

Basically, it elides all information related to type parameters and type arguments.

For instance, the parameterized type `List<String>` is translated to type `List` , which is the so-called *raw type*.

The same happens for the parameterized type `List<Long>` ; it also appears as `List` in the bytecode.

After translation by type erasure, all information regarding type parameters and type arguments has disappeared.

As a result, all instantiations of the same generic type share the same runtime type, namely the raw type.

Angelika Langer's FAQ

Consequences of Type Erasure

The use of type erasure limits the usefulness* of formal Java generics. For example:

```
public class Foo<T> {

   private T[] array;          // fine

   public Foo(int Sz) {

      array = new T[Sz];
   }
}
```

Illegal:

When the code is compiled, `T` will be replaced by its bound (which may be merely `Object`).

The compiler also auto-generates a typecast for the return value from `new`.

The typecast will fail because `Object[]` is-not-a `T[]`.

***vs C++ templates**

How to Cheat...

```
public class HashTable<T extends Hashable> {

   T[] Table;
   ...

   public HashTable(int Sz, probeOption Opt) {
      ...
      Table = (T[]) new Hashable[Sz];
      ...
```

Allocate array of an appropriate "concrete" type

Typecast to the generic type...

The price is that this typecast will generate a warning since the compile cannot know that this will be safe.