Primitive Java Generic Class

```java
public class PrimitiveMemoryCell {
    private Object storedValue;

    public Object read() {
        return storedValue;
    }

    public void write( Object x ) {
        storedValue = x;
    }
}
```

PrimitiveMemoryCell is an old-style Java "generic" class.

Used without a parameter, we obtain a very flexible storage unit… so flexible that it could hold anything at all…

```java
... PrimitiveMemoryCell raw = new PrimitiveMemoryCell ();
raw.write( new String("anodyne") );
System.out.println("Contents are " + raw.read());
raw.write( new Integer(100) );
System.out.println("Contents are " + raw.read());
```
Simple Formal Java Generic Class

GenericMemoryCell is a formal Java generic class.

Used without a parameter, we still obtain a very flexible storage unit… so flexible that it could hold anything at all…

```java
genericmemorycell<T> { 
    private T storedValue;
    public T read() { 
        return storedValue;
    }
    public void write( T x ) { 
        storedValue = x;
    }
}
```

```java
GenericMemoryCell raw = new GenericMemoryCell();
raw.write( new String("anodyne") );
System.out.println("Contents are " + raw.read());
raw.write( new Integer(100) );
System.out.println("Contents are " + raw.read());
```
But, used with a parameter, we can create a parameterized Java class:

```java
... 
GenericMemoryCell<String> MC = new GenericMemoryCell<String>();
MC.write( new String("anodyne") );
System.out.println("Contents are " + MC.read());
```

Operations on the object `MC` are type-checked at runtime to be sure we are only using `MC` to store objects of type `String`.

```java
MC.write( new Integer(100) );
```

```
exGMC.java:9: write(java.lang.String) in 
GenericMemoryCell<java.lang.String>
    cannot be applied to (java.lang.Integer)
    MC.write( new Integer(100) );
```
The `contains()` method can be used to search an array holding objects of any type.

```java
public static <T> boolean contains( T[] array, T x) {
    for ( T value : array ) {
        if ( x.equals(value) )
            return true;
    }
    return false;
}
```

```java
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

```java
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];
}
```

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

```java
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

```java
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…
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...
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.
Consequences of Type Erasure

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

```java
public class Foo<T> {
    private T[] array; // fine

    public Foo(int Sz) {
        T[] array = new T[Sz];
    }
}
```

Illegal:

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

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