Two simple generic classes

To the right, we define a class Wrapper1. Each instance wraps one value. Use procedure set to store a value in a Wrapper instance, and function get to get a value out.

```
public class Wrapper1 {
   private Object object;
   public void set(Object ob) {
      object= ob;
   }
   public Object get() {
      return object;
   }
}
```

Suppose we create a Wrapper1 instance and store a String in it:

```
Wrapper1 w= new Wrapper1(); w.set("bcd");
```

When getting the value out, it must be cast to String:

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

Further, we have to be very careful with each instance of Wrapper1 to remember what type of value it wraps. It's easy to make mistakes in trying to keep track of these things.

Instead, to the right, we create a *generic class*, Wrapper, giving it a *type parameter* E within braces, "<E>". Throughout the body of the class, we use E as a type.

```
public class Wrapper<E> {
   private E object;
   public void set(E ob) {
      object= ob;
   }
   public E get() {
      return object;
   }
}
```

Then, we declare a Wrapper variable and create a Wrapper instance, giving a *type argument* to replace the type parameter, as in

```
Wrapper<String> s= new Wrapper<String>();
Wrapper<Integer> h= new Wrapper<Integer>();
```

Now, field s.object has type String and field h.object has type Integer. Similarly the return types of s.get and h.get are String and Integer, respectively, so that we can write the following without having to cast s.get() to String.

```
String s1= s.get();
```

Diamond notation <>

We can abbreviate a declaration-assignment like this, omitting the type argument:

```
Wrapper<String> s2= new Wrapper<>();
```

That's because the type to be placed within <> can be inferred from the type of s2. Whenever the missing type can be inferred, we can write <>.

More than one type parameter

The class to the right shows how to write a generic class with several type parameters; they are separated by commas. In this case the parameters are E and F. This special class is written to make it easy to wrap two values in an object. The fields are public, so no getters and setters are needed. We need only two constructors and function toString.

```
/** An instance contains an ordered pair. */
public class Pair<E, F> {
   public E first;      // First element
   public F second; // Second element

   /** Constructor: a null pair */
   public Pair() {}

/** Constructor: a pair e, f */
   public Pair(E e, F f) {
      first= e;
      second= f;
   }

   /** return a representation of this pair. */
   public @Override String toString() {
      return "(" + first + ", " + second + ")";
   }
}
```

The examples below show how to create and use instances of class Pair. Note how the second argument of type Pair is itself a Pair.

```
Pair<Integer, Pair<String, Boolean>> p2= new Pair<>();
p2.first= 5;
p2.second= new Pair<>("a", true);
System.out.println(p2);
```

The three constants 5, "a", and true are autoboxed.

This code prints the string: (5, (a, true)) .