To the right is a simple class IntBox, an instance of which contains an int value. It could have many more methods, but it has just what we need to explain parametric poly­morphism.

/\*\* object contains a char. \*/

public class CharBox {

private char contents;

/\*\* Constr: box with '\u0000'. \*/

public CharBox() {}

/\*\* Put t into the box \*/

public void put(char t) {

contents= t;

}

/\*\* Return contents. \*/

public char get() {

return contents;

}

}

/\*\* object contains an int. \*/

public class IntBox {

private int contents;

/\*\* Constr: box with 0. \*/

public IntBox() {}

/\*\* Put t into the box \*/

public void put(int t) {

contents= t;

}

/\*\* Return contents. \*/

public int get() {

return contents;

}

}

Now suppose someone asks us to create a box class to hold a char value instead of an int value. We can copy class IntBox and change the copy to be a CharBox.

Now suppose someone asks us to create a class to hold a *third* kind of value. How many times are we going to have to create a new class, copy, paste, and edit? There *must* be a better way.

There *is* a better way, called *parametric polymorphism*, and it is implemented in Java using *generics.* As shown to the right, we give class Box a *type parameter* T enclosed in “<” and “>”, shown in red.

/\*\* object contains a T. \*/

public class Box<T> {

private T contents;

/\*\* Constr: box with   
 default value of T. \*/

public Box() {}

/\*\* Put t into the box \*/

public void put(T t) {

contents= t;

}

/\*\* Return contents. \*/

public T get() {

return contents;

}

}

Then, field contents has type T, parameter t of method put has type T, and the return type of method get is T. Also, when a new Box object is created, the value in it will be the default value for T.

To create a Box object that can contain an Integer value (pointer to an Integer object) and a Box object that can contain a Character value, use these two statements:

* Box<Integer> bi= new Box <>();
* Box <Character> bc= new Box <>();

In the first statement, Integer is the *type argument* given for type parameter T; similarly for Character in the second statement.

Think of this as giving us objects of classes like

* Class IntBox above, except that wrapper class Integer is used in place of type int.
* Class CharBox above, except that wrapper class Character is used in place of type char.

The wrapper classes have to be used because type arguments may only be class types and never primitive types. That’s OK; Java will automatically box and unbox between primitive type values and their wrapper classes. We don’t have to be concerned too much with that.

This, then, is *parametric polymorphism*. The use of a type parameter allows us to have many boxes, which differ only in the types of values they can hold.

Note the following. We generally write the new-expression in the assignment to bi above as shown below, with Integer between the < and >. But in this statement we can omit Integer because Java can infer what should be between < and > from the context.

Box<Integer> bi= new Box <>();

We have shown the simplest use of *generics* in Java. There is much more. Look at the entry for generics in JavaHyperText.