As you can see, the output is similar to the previous version of the program.

There are several things of interest in this version. First, notice that **NonGen** replaces all uses of **T** with **object**. This makes **NonGen** able to store any type of object, as can the generic version.

However, this approach is bad for two reasons.

First, explicit casts must be employed to retrieve the stored data.

Second, many kinds of type mismatch errors cannot be found until runtime.

Let’s look closely at each problem.

We will begin with this line:

int v = (int) iOb.GetOb();

Because the return type of **GetOb( )** is now **object**, the cast to **int** is necessary to enable the value returned by **GetOb( )** to be unboxed and stored in **v**. If you remove the cast, the

program will not compile.

In the generic version of the program, this cast was not needed

because **int** was specified as a type argument when **iOb** was constructed.

In the non-generic

version, the cast must be employed.

This is not only an inconvenience, but a potential

source of error.

Now, consider the following sequence from near the end of the program:

// This compiles, but is conceptually wrong!

iOb = strOb;

// The following line results in a runtime exception.

// v = (int) iOb.GetOb(); // runtime error!

Here, **strOb** is assigned to **iOb**. However, **strOb** refers to an object that contains a string, not an integer. This assignment is syntactically valid because all **NonGen** references are the same,

and any **NonGen** reference can refer to any other **NonGen** object.

However, the statement is semantically wrong, as the commented-out line shows. In that line, the return type of **GetOb( )** is cast to **int** and then an attempt is made to assign this value to **v**. The trouble is that **iOb** now refers to an object that stores a **string**, not an **int**. Unfortunately, without generics, the

compiler won’t catch this error.

Instead, a runtime exception will occur when the cast to **int** is attempted.

To see this for yourself, try removing the comment symbol from the start of

the line and then compiling and running the program.

A runtime error will occur.

The preceding sequence can’t occur when generics are used. If this sequence were attempted in the generic version of the program, the compiler would catch it and report an error, thus preventing a serious bug that results in a runtime exception.

The ability to create type-safe code in which type-mismatch errors are caught at compile time is a key advantage of generics. Although using **object** references to create “generic” code has always been possible in C#, that code was not type-safe and its misuse could result in runtime exceptions.

Generics prevent this from occurring. In essence, through generics, what

were once runtime errors have become compile-time errors. This is a major benefit. There is one other point of interest in the **NonGen** program. Notice how the type of the

**NonGen** instance variable **ob** is obtained by **ShowType( )**:

Console.WriteLine("Type of ob is " + ob.GetType());

Recall that **object** defines several methods that are available to all data types.

One of these methods is **GetType( )**, which returns a **Type** object that describes the type of the invoking object at runtime. Thus, even though the type of **ob** is specified as **object** in the program’s source code, at runtime, the actual type of object being referred to is known.

This is why the CLR will generate an exception if you try an invalid cast during program execution