.NET Generics

Introduction

C# *generics* were modeled after *templates* in C++[1].

Although Bjarne Stroustrup (the inventor of C++) dislikes referring to C++ templates as macros, it's close enough so I'm going to describe them as a type of macro.

But note that templates are much more powerful than generics, although the advanced features of templates often requires an expert to create and maybe even use them. However the designers of C# wanted a language that didn't require rocket scientists to code in, so they opted to support just the major benefits of templates.

The First Benefit of Generics

One goal is to reduce the amount of code written by the programmer. Consider a *min* function that takes two int's and returns the smaller of the two.

```
int min(int x, int y) {
       if (x < y) {
              return x;
       } else {
              return y;
       }
}
```

Now consider a *min* function that takes two single precision (*float*) arguments

```
float min(float x, float y) {
       if (x < y) {
              return x;
       } else {
              return y;
       }
}
```

Note that, except for the data types, the source code is identical[2]. And the same comment holds true for other data types, such as *double*, *long*, *short*, etc. And since the "<" operator is defined on, say, *strings*[3], again, the same code would work on strings.

So we could write…

```
T min<T>(T x, T y) {
       if (x < y) {
              return x;
```

[1] I don't know where C++ got the idea from.
[2] The *object* code would be different, but that's the compiler's problem, not the programmer's.
[3] Or, if your user-defined class defines operator <, it could even work in that case.

```
        } else {
              return y;
        }
}
```

The min<*T*> syntax tells the compiler that this is (sort of) a macro with a parameter *T*[4], and we would invoke it as

```
int int_mini = min<int>(3, 6);
var float_mini = min<float>(1.2f, -4.75f);// "f" suffix means single precision
double dbl_mini = min<double>(1.2, Math.PI);
```

params

Nothing to do with generics (although it can be used with them), but this is a reasonable place to introduce the *params* keyword.

Suppose you want the minimum of, not 2 values, but 3. Sure, we can write a method with parameters *x*, *y*, and *z*. And, if necessary, write yet another method that finds the minimum of 4 values. And so on. But where do you stop? 10 parameters? 20? Where?

Well, one approach would be to define a method with a different parameter signature, taking an array of values. For example,

```
int[] nums = new int[] {3, 6, 27};
int int_mini = min<int>(nums);

T min<T>(T[] vals) {
       if (vals.Length = 0) {
              return int.MinValue;
       }
       int min = vals[0];
       for (int i = 1; i < vals.Length; ++i) {
              if (vals[i] < min) {
                     min = vals[i];
              }
       }
}
```

[4] *T* being, by convention, short for *Type*. However, longer names are allowable and even encouraged. For example, a hash table (which .NET calls a *Dictionary*) that takes a *string* and returns a*n* object might be declared as *Dictionary<TKey, TValue>*.