**Oriette Ramos**

**Lesson Plan 22**

**ISTA-220**

1. What is a type parameter?  
   **A type parameter is a generic type that can be used in place of a real type.**
2. What does a type parameter do?  
   **A type parameter acts as a placeholder for a type in the code that is replaced by a real type at compile time based on the types used. For example, List<T> myList = new List<T>(); will create a new generic list. If the code also has myList.Add("Jason"); then when the program is compiled it will automatically convert the previous code to List<string> myList = new List<string>();.**
3. How many type parameters can a generic class have?  
   **A generic class does not have any limit on the number of type parameters it can have.**
4. What is the difference between a generic class and a generalized class?  
   **A generic class has its type parameters converted to real types (referred to as constructed types) at compile time, so boxing and unboxing are unnecessary. A generalized class can hold any object, so the data being used must be cast to the correct type anytime the programmer wants to use it.**
5. What is a constraint? How do you specify a constraint?  
   **By using a constraint, you can limit the type parameters of a generic class to those that implement a particular set of interfaces and therefore provide the methods defined by those interfaces. For example, if the IEdible interface implemented the Eat method, you could create the EdibleCollection like this: public class EdibleCollection<T> where T : IEdible.**
6. What is a generic method? How do you define a generic method?  
   **With a generic method, you can specify the types of the parameters and the return type by using a type parameter in a manner similar to that used when you define a generic class. In this way, you can define generalized methods that are type safe and avoid the overhead of casting (and boxing, in some cases). You define generic methods by using the same type parameter syntax you use when you create generic classes. (You can also specify constraints.) For example, the following code implements the generic Trade method:**
7. static void Trade<T>(ref T thing1, ref T thing2)
8. {
9. T temp = thing1;
10. thing1 = thing2;
11. thing2 = temp;

}

1. What do we mean when we say that a generic type interface is invariant?  
   **You cannot assign an IWrapper<A> object to a reference of type IWrapper<B>, even if type A is derived from type B. By default, C# implements this restriction to ensure the type safety of your code.**
2. What do we mean when we say that a generic type interface is covariant?  
   **In situations where the type parameter occurs only as the return value of the methods in a generic interface, you can inform the compiler that some implicit conversions are legal and that it does not have to enforce strict type safety. You do this by specifying the out keyword when you declare the type parameter. This feature is called covariance. You can assign an IRetrieveWrapper<A> object to an IRetrieveWrapper<B> reference as long as there is a valid conversion from type A to type B, or type A derives from type B. Only interface and delegate types can be declared as covariant. You do not specify the out modifier with generic classes.**
3. Does covariance work with value types? Does it work with reference types?  
   **Covariance works only with reference types. This is because value types cannot form inheritance hierarchies.**
4. What do we mean when we say that a generic type interface is contravariant?  
   **Contravariance follows a similar principle to covariance except that it works in the opposite direction; it enables you to use a generic interface to reference an object of type B through a reference to type A as long as type B derives from type A. The in keyword tells the C# compiler that you can either pass the type T as the parameter type to methods or pass any type that derives from T. You cannot use T as the return type from any methods. Essentially, this makes it possible for you to reference an object either through a generic interface based on the object type or through a generic interface based on a type that derives from the object type. Basically, if type A exposes some operations, properties, or fields, in that case if type B derives from type A, it must also expose the same operations (which might behave differently if they have been overridden), properties, and fields. Consequently, it should be safe to substitute an object of type B for an object of type A.**