CECS 277 – Lecture 12 – Generics

Generics allow you to specify the types of objects that a class or method will use. They use the type parameter to represent the types that will be passed in and used to construct the object. The type parameter is placed in angle brackets <> to specify it, and then used throughout the class whenever that type needs to be used.

Example: Class declaration for a generic type

```
public class Pair <T> {
     private T first;
     private T second;
     public Pair( T f, T s ) {
          setPair( f, s );
     }
     public T getFirst( ) {
          return first;
     }
     public T getSecond( ) {
          return second;
     }
     public void setPair( T f, T s ) {
          first = f;
          second = s;
     }
}
```

Creating an instance of the Pair class is now the same as creating an ArrayList, the type is specified in angle brackets when declared.

```
Pair <String> a = new Pair <String>( "Cat", "Meow" );

Pair <Integer> n = new Pair <Integer>( 3, 3 );

Color r = new Color( Color.RED );
Color g = new Color( Color.GREEN );
Pair <Color> c = new Pair <Color>( r, g );

Pair s = new Pair( "Apple", .67 );
```

Each of the above lines are creating a pair of objects of different types. All but the last line are creating pairs of the same type. This line will return a warning by the compiler, but it will still run. This is because the compiler will give the default type of Object to both of these values, effectively making them the same type. But now, if there are any type specific manipulations or methods that were needed to be used on those objects, they will need to be downcasted to those types first.

Using the Generic Object – When using generics, you do not know anything about the object except that it is an Object. This means that you can always use methods from the Object class, such as, equals(), toString(), hashCode(), etc., but you cannot use any methods from any classes that might be passed in as type T.

```
public class Pair <T> {
     ...
     public void method( T t ) {
          t.getName(); //<- does not work
     }
}
```

This doesn't work since the compiler does not know if the generic type T has the method getName() defined in its class. However, there are ways to fix this.

Constraining the Generic Type – if we know some information about the types of classes that T might be, we can constrain the generic type so we can access the methods of that class.

```
public class Pair <T extends Animal> {
     ...
     public void method( T t ) {
          t.getName(); //<- can use methods in Animal
     }
}
```

The above example is considered as giving the generic type a constraint. This technique also works for interfaces as well (still uses extends). If all of the classes you are going to use in your generic class are derived from a common superclass or interface, then you can specify this when defining your generic parameter, this way you can use any of the methods that are in the super class in your generic class.

Downcasting – if we know that the generic type might be only one of a few possibilities, we can test the object to see if it is of one of those types.

```
public class Pair <T> {
     ...
     public void method( T t ) {
          if( t instanceof Dog) {
               Dog d = (Dog) t;
               d.getName(); <-- can use methods in Dog
          }
     }
}
```

This is usually frowned upon and it should be avoided whenever possible. The point of using generics is so that it works for all types. Downcasting requires you to know each type that it might be (including future types that haven't been created yet, which means that it will have to be consistently updated).

Generic Methods – Similar to classes, methods can also be made to be generic. Since the generic type T may be ambiguous (it could be a type of object made from a class named T), the generic type <T> needs to be specified before the return type. Local variables of type T may be created in the method as needed. The following examples are using generic parameters and return types.

```
public static <T> void displayArr( T [] arr ) {
     for( int i = 0; i < arr.length; i++ ) {
          T val = arr[i];
          System.out.print( val + " " );
     }
     System.out.println( );
}

public static <T> T replace( T [] arr, int i, T new) {
     T old = arr[i];
     arr[i] = new;
     return old;
}
```

When the method is called, the type does not need to be specified like a class does when creating a new object. With methods, the type is inferred based on the data type of the parameter passed in to the method.

```
Integer [] intArr = {1, 2, 3, 4, 5};
String [] strArr = {"A", "B", "C"};

displayArr( strArr ); // A B C
displayArr( intArr ); // 1 2 3 4 5

System.out.println( replace( strArr, 0, "Z" ) ); // A
System.out.println( replace( intArr, 2, 7 ) );   // 3

displayArr( strArr ); // Z B C
displayArr( intArr ); // 1 2 7 4 5
```

Just like classes, generic methods can also be constrained and use multiple generic types.