



Topic Notes: Generics

We saw in our examples using `Vector` that the items stored within our `Vector`s are treated as instances of `class Object`.

This is quite convenient in that we can store whatever type of objects we wish, and until version 1.4 of the Java Development Kit, this was the preferred approach.

This approach does have a few disadvantages:

1. When we retrieve an item from our `Vector`, we need to use a cast before we can treat it as an instance of its own specific type.
2. If we make a programming error and mistakenly place an object of one type into the `Vector` but then cast it to a different type upon retrieval, your program will crash with a *runtime error*. Ideally, we would be able to detect such errors sooner – when we compile.

One approach to dealing with these disadvantages is to implement a *specialized* version of our `Vector` (or whatever other) data structure that holds exactly the type of items we wish, much like we can declare arrays of any type.

To implement, for example, a `Vector` that holds `Integers` (we could call it `class IntegerVector`), we could take the `Vector` implementation, and instead of using `Objects` as the type for our internal array and for the method parameter and return types, we would use `Integer`.

This would take care of both disadvantages we noted in the original `Vector` implementation. Casts are no longer needed because the return type of methods such as `get` would be `Integer`. And perhaps more importantly, if we attempted to write code that stored anything other than an `Integer` (or a subclass of `Integer`), the Java compiler would flag the error (a *compile-time error*), which is much more convenient time to detect an error than at run time.

But unfortunately, this “solution” means writing a brand new specialized `Vector`-like class for each data type we need to store.

Starting with JDK 1.5 (Java 5), Java was extended to allow class definitions to include *generic*, or *parameterized data types*. This means that we can write a definition of the structure using data types that are unspecified (much like the value of a method parameter is unspecified) until we create an instance of the class.

We can see generic versions of `Association` and `Vector` in use:

See Example:

`/home/jteresco/shared/cs211/examples/Spells/SpellsVectorT.java` **See Example:**

`/home/jteresco/shared/cs211/examples/PocketChange/PocketChangeT.java:`

As you can see in the examples, we specify the data type of the items we will be storing in the generic data structure in angle brackets after the structure type. For example, the `Vector` of `Integer`:

```
Vector<Integer> intVec = new Vector<Integer>();
```

With this declaration, any attempt to store an item which is not of type `Integer` or any treatment of an item retrieved as a non-`Integer` type will result in a compile-time error.

The generic data types, including `Vector` and `Association` are provided in the `bailey.jar` library, but you will need to `import structure5.*;` instead of `import structure.*;` at the top of your program.

Note: we would like to be able to use a primitive type as a type parameter:

```
Vector<int> intVec = new Vector<int>();
```

but this is not permitted – the type parameters must be object types. Fortunately, with autoboxing, this is not much of an inconvenience to programmers.

From here on, we will make use of the generic classes.

Generic Association Implementation

See Structure Source:

```
/home/jteresco/shared/cs211/src/structure5/Association.java
```

In the `Association` class, we in fact make use of two type parameters, one for the key and one for the value. Rather than treating both as `Objects` as in the non-generic `Association` implementation, we use `K` for the type of the key and `V` for the type of the value.

Everywhere we refer to the key and value in type declarations for variables, parameters, or return types, we can use `K` and `V`, respectively.

The rest of the code for the class remains unchanged from the non-generic `Association`.

Generic Vector Implementation

The generic `Vector` class is parameterized on the type of the items (elements) it will contain. The implementation uses `E` as the type.

For the most part, the `Vector` implementation is straightforward. However, a technical problem comes into play when we declare the `Vector`'s internal array. This is not something we will concern ourselves with at this point, but the description of the problem and of its solution within `structure` is worth reading in the text.