SIENAcollege
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2019

Topic Notes: Generic Classes

General Purpose Classes

The few custom classes we have seen so far have been developed for very specific situations. The `Ratio` class stores a ratio, but isn't useful for much else. The `PurchasedItem` class in the purchase tracker example contains a `String`, a `double`, and an `int`, all used for very specific purposes.

Part of a good object-oriented design is to find places where we can write some software (in our case, a Java class) that might be useful in situations beyond the one at hand. It turns out there are many simple and no-so-simple structures that arise in many contexts, and rather than developing a new Java class each time we need one, we strive to re-use ones that already exist. Or if one does not exist, develop one that will satisfy needs of future programmers as well.

Let's consider one common *data structure*: the *pair*. To begin, let's assume that we want a pair of floating-point values. These might be used to represent a coordinate in two dimensions, a latitude and longitude, or perhaps even a pair of corresponding data values from a science experiment such as an object's volume at a given temperature.

Without knowing the reason someone might want to use our pair of `double` values, we can write a class that encapsulates them:

See Example: DoublePair

Notice that we have only very general-purpose code here. While it might make sense, for example, to add up the two numbers in the pair in some cases, we don't add that to our general purpose class.

Before we move on, we notice a couple of other items of interest.

- We provide a method `equals` that returns whether this `DoublePair` is the same as another `DoublePair`. `equals` is another of those methods (like `toString`) that are provided by Java for any object type. But like `toString`, we normally will want to provide our own `equals` method that determines the equality or equivalence of two objects of the class we are defining in some meaningful way. Here, we define `equals` so that two `DoublePair` objects are equal only if each of the numbers this one contains are equal to the corresponding numbers in the other.

  Note also that the method signature of the `equals` method includes a formal parameter of type `Object`, even though we know that for this to make any sense, it should be passed as an object of type `DoublePair`. It is necessary to define it with `Object`, however, as `equals` methods must have this same method signature for all classes to conform to Java's rules.

The main complication of this fact is that we need to tell Java that we expect this `Object` to be a `DoublePair`, so we can make the meaningful comparison. The first line of the body of our `equals` method is a *cast*, which tells Java that the object we passed in as `o` should subsequently be treated as a `DoublePair` called `other`. The main drawback of this is that if we pass in some other type of object, the program will crash when the cast is attempted, with a `ClassCastException`. Uncomment the last line of the provided `main` method to see this happen.

- We provide a `main` method that tests our class. This method would not be used by a "user" of this class – it is provided only as a convenient way to test the class. It would work exactly the same way if we placed the `main` into a separate class.

Making it More General Purpose

While it's nice to have a pair of `double` values, what if we need a pair of `int`s, or `String`s, or `PurchasedItem`s? Or maybe a pair where the "first" is a `String` and the "second" is a `double`?

Java's `Object` class, which we just saw as a parameter to the `equals` method, provides a mechanism we can use for this.

See Example: ObjectPair

The code is nearly identical, except now our "first" and "second" can be any object type we wish.

In the `main` method, we actually pass primitive types as well, but that is being facilitated by Java's *autoboxing* functionality.

Since primitive types, like `int`, `double`, and `boolean` are not objects, they do no qualify as a valid item to be stored in an `Object` variable. But, since we often do want to treat such values as objects, Java provides a set of classes, including `Integer`, `Double`, and `Boolean`, that are objects whose sole purpose is to include one `int`, `double`, or `boolean`, respectively. In old Java versions, we would have had to create such objects explicity (*e.g.*, "`new Double(9.1)`") but the Java compiler will now insert code to do this for us automatically.

Making it Generic

Java also allows 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. But once we "bind" to a type, we have to stick with that type (unlike how our `ObjectPair`'s "first" was initially a `String` and later a `Double`).

Here is the generic version of our pair class:

See Example: GenericPair

The important feature here is the use of *type parameters* to specify the actual data types we want to use for the first and second entries of our pair. These are specified throughout in this example as

`U` and `V`. We assume (and in fact, require) that all places we refer to `first` are of type `U` and of `second` of type `V`. This includes declarations of instance variables, formal parameters, and return types.

Only when we construct an actual `GenericPair` object do we specify the types we wish to use. See in the provided `main` method how this is done.

As an example of a place we could use the generic pair, let's look at a program that reads in a list of *Harry Potter* spells into a pair of parallel arrays and allows the user to look up the spell actions by name.

See Example: SpellsArrays

Rather than having two separate arrays of `String` values, we could group them in `GenericPair` objects. Each will have the spell incantation in its `first` and the spell's action in its `second`.

See Example: SpellsArrayGenericPair

Associations

Let's consider a very simple example of a data structure that we'll call an *association*.

As the name suggests, an association is a way to associate pairs of objects, one of which is the *key* and one of which is the *value*. Unlike the "pair" structures we just considered, once created, the key of an association cannot be changed, only the value can.

This is another "general purpose" structure, but one with the above restriction.

This is the first of many situations where we will study and use a data structure that is a restricted version of one we already have. This structure does everything a pair does, except it does not allow the key to be modified. We will later that we can use this restriction to our advantage.

When developing any such structure, there are some questions to be answered first.

- What should such a structure look like?
- What instance variables will it need?
- What constructors should be provided?
- What methods will it need?

As we have seen, we have two main options when developing a generic class. We can develop our structures to hold references to `Objects` and then use them to store instances of any Java class, or use type parameters. The latter is the usual approach in modern Java programs.

An implementation of this structure is in a modified version of our spells example:

See Example: SpellsArrayAssociation

The actual implementation of the `Association` class is pretty straightforward. A couple of quick notes:

- We require a `key` to construct a new `Association`, but the `value` is optional. If not provided, the `value` part defaults to `null`.
- Two `Association`s are considered equal (by the `equals` method) if their `key`s are the same, regardless of their `value`s.
- We have an accessor for the `key` (`getKey`) but no mutator. Once created, the `key` of an `Association` may not be modified.
- For the `value`, we have both an accessor (`getValue`) and a mutator (`setValue`).