Chapter 4

Generics

Concepts:
- ▷ Motivation for Parameterized Types
- ▷ Simple Parameterized Types
- ▷ Parameterization in `Vector`

Thank God that
the people running this world
are not smart enough
to keep running it forever.
—Arlo Guthrie

THE MAIN PURPOSE OF A LANGUAGE is to convey information from one party to another. Programming languages are no exception. Ideally, a language like Java lets the programmer express the logic of an algorithm in a natural way, and the language would identify errors of grammar (i.e., errors of *syntax*) and meaning (*semantics*). When these errors are flagged as the program is compiled, we call them *compile-time* errors. Certain logical errors, of course, do not appear until the program is run. When these errors are detected (for example, a division-by-zero, or an object reference through a null pointer) Java generates *runtime* errors.[1] It is generally believed that the sooner one can detect an error, the better. So, for example, it is useful to point out a syntax error ("You've forgotten to declare the variable, `cheatCode`, you use, here, on line 22.") where it occurs rather than a few lines later ("After compiling `Lottery.java`, we noticed that you mentioned 4 undeclared variables: `random`, `range`, `trapdoor`, and `winner`."). Because a runtime error may occur far away and long after the program was written, compile-time errors (that can be fixed by the programming staff) are considered to be preferable to runtime errors (that must be fixed by the public relations staff).

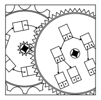
Java 5 provides some important language features that shift the detection of certain important errors from runtime to compile-time. The notion of a *generic class* is one of them. This chapter is about building generic classes.

[1] Even after considering compile-time and runtime errors, there are many errors that even a running system cannot detect. For example, undesirable infinite loops (some *are* desirable) are often the result of errors in logic that the computer cannot detect. The word "cannot" is quite strong here: it is impossible to build an environment that correctly identifies any program that loops infinitely (or even more broadly–fails to halt). This result is due to Alan Turing.

4.1 Motivation (in case we need some)

In Chapter 3 we constructed the `Vector` class, a general purpose array. Its primary advantage over arrays is that it is *extensible*: the `Vector` can be efficiently lengthened and shortened as needed. It suffers one important disadvantage: it is impossible to declare a `Vector` that contains exactly one type of object. Ponder this: we almost always declare variables and arrays to contain values of one particular type, but `Vector`s hold `Object`s. As we have seen, an `Object` can hold *any* reference type, so it is hardly a restriction. How can we construct a `Vector` restricted to holding just `String` references?

Of course, you could depend on your naturally good behavior. As long as the programs you write only add and remove `String`s from the `Vector`, life will be good. As an example, consider the following (silly) program:



LongWords

```
public static void main(String[] args)
{
    Vector longWords = new Vector();
    int i;
    for (i = 0; i < args.length; i++) {
        if (args[i].length() > 4) {
            longWords.add(args[i]);  // line 12
        }
    }
       ...
    for (i = 0; i < longWords.size(); i++) {
        String word = (String)longWords.get(i); // line 31
        System.out.println(word+", length "+word.length());
    }
}
```

This `main` method builds a `Vector` of long arguments—`String`s longer than 4 characters—and, sometime later, prints out the list of words along with their respective lengths. If we type

```
java LongWords Fuzzy Wozzie was a bear
```

we expect to get

```
Fuzzy, length 5
Wozzie, length 6
```

But programming is rarely so successful. Suppose we had instead written

```
for (i = 0; i < args.length; i++) {
    if (args[i].length() > 4) {
        longWords.add(args);  // line 12
    }
}
```

This mistake is silly, but most mistakes are.

On line 12 we are missing the index from the `args` array in the call to `add`. Instead of adding the `i`th argument, we have added *the entire argument array*, every time. Because we have no way to restrict the `Vector` class, the program compiles correctly. We only notice the mistake when we type

```
java LongWords Fuzzy Wozzie had no hair
```

and get

```
Exception in thread "main" java.lang.ClassCastException: [Ljava.lang.String;
        at LongWords.main(LongWords.java:31)
```

The problem is eventually recognized (on line 31) when what we thought was a `String` (but is actually an array) is removed as an `Object` (O.K.) and cast to a `String` (not O.K.). Notice if our broken program is run with only short words:

```
java LongWords aye but what a bear that Fuzz was
```

no runtime error occurs because no words are added to the `Vector`.

4.1.1 Possible Solution: Specialization

Another solution to our problem would be to create a specialized class, a `String-Vector` that was similar to the `Vector` class in every way, except that `Objects` are replaced with `Strings`. The `add` method, for example, would be declared:

```
public class StringVector implements Cloneable
{
    protected String elementData[];     // the data
    protected int elementCount;         // number of elements in vector
    ...
    public void add(String obj)
    {
        ensureCapacity(elementCount+1);
        elementData[elementCount] = obj;
        elementCount++;
    }
}
```

StringVector

Compare this with the code that appears in Chapter 3 on page 51.

There are several things to note about this approach. First, it works. For fun, we create a similar (erroneous) program, `LongWords2`, that makes use of the `StringVector`:

```
public static void main(String[] args)
{
    StringVector longWords = new StringVector();
    int i;
    for (i = 0; i < args.length; i++) {
        if (args[i].length() > 4) {
            longWords.add(args);  // line 12
        }
    }
       ...
    for (i = 0; i < longWords.size(); i++) {
        String word = longWords.get(i);  // line 31
        System.out.println(word+", length "+word.length());
    }
}
```

LongWords2

Instead of using `Vector` we use `StringVector`. The compiler knows that the `add` method must take a `String`, so passing an array of `String`s to the add method generates the error:

```
LongWords2.java:12: cannot find symbol
symbol  : method add(java.lang.String[])
location: class StringVector
                longWords.add(args);
                         ^
```

The success of this technique leads us to the following principle:

Principle 7 *Write code to identify errors as soon as possible.*

Here, (1) the actual source of the logical error was identified, and (2) it was identified at compile-time.

To accomplish this, however, we had to write a new class and modify it appropriately. This is, itself, an error-prone task. Also, we must rewrite the class for *every* every new contained type. Soon we would be overwhelmed by special-purpose classes. These classes are not related in any way and, it should be noted, our original class, `Vector`, would never be used. Imagine the difficulty of adding a new method to the `Vector` class that you wished to make available to each of your specialized versions! A better solution is to create a *generic class* or a class *parameterized by the type of data it holds*.

4.2 Implementing Generic Container Classes

Ideally, we would like to be able to construct *container types*, like `Associations` and `Vectors`, that hold objects of one or more specific types. At the same time, we would like to write each of these classes once, probably without prior knowledge of their client applications. If this were possible, we could provide much of the polymorphic utility of these classes as they were implemented in Chapters 1 and 3, and still identify errors in data types at compile-time. In Java 5, *generic* or *parameterized data types* provide the programmer the necessarily flexibility.

4.2.1 Generic `Association`s

For simplicity, we first consider the signatures of the protected data and several of the methods of the `Association` class, declared using parameterized types:[2]

```
package structure5;
public class Association<K,V>
{
```

Association

[2] The remainder of this text describes classes of the `structure5` package. The `structure` package provides `Object`-based implementation, for backward comparability with pre-Java5 compilers. Both the `structure` and `structure5` packages are available in a single `jar` file, `bailey.jar`, that may be inserted in your `CLASSPATH`.

```
        protected K theKey; // the key of the key-value pair
        protected V theValue; // the value of the key-value pair

        /*
          for example:
          Association<String,Integer> personAttribute =
             new Assocation<String,Integer>("Age",34);
         */
        public Association(K key, V value)
        // pre: key is non-null
        // post: constructs a key-value pair

        public V getValue()
        // post: returns value from association

        public K getKey()
        // post: returns key from association

        public V setValue(V value)
        // post: sets association's value to value
    }
```

At the time of their declaration, parameterized class name are followed by a list of comma-separated *type parameters* in angle brackets. This book follows the common convention that type parameters are indicated by single capital Latin letters.[3] In this example, K is a place holder for the type of the association's key, while V will be used to represent the actual type of the association's value. Within the class definition, these type variables may be used wherever class names are used: declaring data variables and declaring method parameters and return values. The one constraint is that

Generic types may not appear in array allocations.

The reason is technical and obviously not of concern in our declaration of Associations.

The code associated with the generic implementation of the Association class is, at this point, straightforward. For example, the setValue method would be written:

```
    public V setValue(V value)
    {
        V oldValue = theValue;
        theValue = value;
        return oldValue;
    }
```

[3] There is some dispute about this convention. See your instructor. Professional driver, closed course. Your mileage may vary. Danger, Will Robinson. Some identifiers too small for three year olds.

Notice that there are no casts in this code: since the value is declared to be of type `V`, and since the return value for `setValue` is the same, no cast is necessary. The removal of casts is a sign that type checking will happen in the compiler and not while the program is running.

To make use of the generic version of a class, we must specify the actual parameter types associated with the formal types demanded by `Association`. For example, a new `Association` between a `String` and an `Integer` would be declared:

```
Association<String,Integer> personAttribute =
   new Assocation<String,Integer>("Age",34);
```

(Hotdoggers: the 34 here is, yes, *autoboxed*[4] into an `Integer`.) Of course, if the `Association` is declared within a class that is itself parameterized, formal type parameters may be specified. We will see this on many occasions in future chapters.

4.2.2 Parameterizing the `Vector` Class

Yeah, and if we eat our spinach now, we won't get E. coli *later.*

We now consider the parameterization of a significant container class, `Vector`. Because of the way that `Vector`s are implemented, there will be special technical details (associated with arrays) that pertain, for the most part, only to the `Vector` class. Most other structures will avoid these difficulties because they will make use of parameterized `Vector`s, and therefore build on the work we do here. Still, for the purposes of efficiency, it may be useful to declare structures that make direct use of the array-based techniques described here.

(The reader may find it useful to review the implementation of the non-generic `Vector` as declared in Chapter 3 before following along here, where the technical details of the `Vector` implementation will not be reviewed.)

One thing seems fairly obvious: the parameterized `Vector` class should be defined in terms of a single type—the type of each element within the `Vector`. The declaration of the major method signatures of the `Vector` class is as follows:

`Vector`

```
public class Vector<E> extends AbstractList<E> implements Cloneable
{
    private Object elementData[];       // the data
    protected int elementCount;         // number of elements in vector
    protected int capacityIncrement;    // the rate of growth for vector
    protected E initialValue;   // new elements have this value
    protected final static int defaultCapacity = 10; // def't capacity, must be>0

    public Vector()
    // post: constructs a vector with capacity for 10 elements
```

[4] Two syntactic features of Java 5 include *autoboxing* and its inverse, *-unboxing*. With autoboxing, primitive types are automatically "boxed" into their object equivalents if doing so would fix a type incompatibility. In addition, the corresponding object types will be converted to primitive equivalents, if necessary.

```
        public Vector(int initialCapacity)
        // pre: initialCapacity >= 0
        // post: constructs an empty vector with initialCapacity capacity

        public Vector(int initialCapacity, int capacityIncr)
        // pre: initialCapacity >= 0, capacityIncr >= 0
        // post: constructs an empty vector with initialCapacity capacity
        //    that extends capacity by capacityIncr, or doubles if 0

        public Vector(int initialCapacity, int capacityIncr, E initValue)
        // pre: initialCapacity, capacityIncr >= 0
        // post: constructs empty vector with capacity that begins at
        //       initialCapacity and extends by capacityIncr or doubles
        //       if 0.  New entries in vector are initialized to initValue.

        public void add(E obj)
        // post: adds new element to end of possibly extended vector

        public E remove(E element)
        // post: element equal to parameter is removed and returned

        public boolean contains(E elem)
        // post: returns true iff Vector contains the value
        //       (could be faster, if orderedVector is used)

        public E get(int index)
        // pre: 0 <= index && index < size()
        // post: returns the element stored in location index

        public void insertElementAt(E obj, int index)
        // pre: 0 <= index <= size()
        // post: inserts new value in vector with desired index,
        //   moving elements from index to size()-1 to right

        public E remove(int where)
        // pre: 0 <= where && where < size()
        // post: indicated element is removed, size decreases by 1

        public E set(int index, E obj)
        // pre: 0 <= index && index < size()
        // post: element value is changed to obj; old value is returned
    }
```

First, it is important to note that the `Vector` holding objects of type `E`, the `Vector<E>` class, implements `AbstractList<E>`. Anywhere where an `Abstract-List<E>` is required, `Vector<E>` may be provided.[5] The `E` in the `Vector<E>` identifies the *formal type parameter*, while the `E` of `AbstractList<E>` is used

[5] This reads like a law journal. A technical point important to note here is that a `Vector<E>` *does not* have any relation to `Vector<S>` if `S` is a subclass of `E`. For example, you cannot assign a

to provide an *actual type parameter* to the AbstractList class. This, of course, presumes that the AbstractList class has been implemented using a single generic type parameter.

At the top of the declaration, we see the instance variables and constants that are found in a Vector. The initialValue variable holds, for each object, the value that is used to fill out the Vector as it is extended. Typically (and, in most constructors, by default) this is null. Clearly, if it is non-null, it should be a value of type E. The values elementCount, capacityIncrement, and the constant defaultCapacity are all ints, unconstrained by the choice of the parameterized type.

The variable elementData is an extensible array (the array the supports the Vector's store): it should contain only objects of type E. Again, because of a technical constraint in Java[6], it is impossible to construct new arrays of any type that involves a type parameter. We must know, at the time the class is compiled, the *exact* type of the array constructed.

But this goes against our desire to construct a parameterized array of type E. Our "workaround" is to construct an array of type Object, limit the type of the elements to E with *appropriate casts within mutator methods*, and to declare the array private eliminating the possibility that subclasses would be able to store non-E values within the array.[7] When values are retrieved from the array, we can be confident that they are of the right type, and using casts (that we know are trivial) we return values of type E.

Let's look at the details. Here is the implementation of the most general Vector constructor:

```
public Vector(int initialCapacity, int capacityIncr, E initValue)
{
    Assert.pre(initialCapacity >= 0, "Nonnegative capacity.");
    capacityIncrement = capacityIncr;
    elementData = new Object[initialCapacity];
    elementCount = 0;
    initialValue = initValue;
}
```

The Vector constructor does not mention the <E> type parameter in its name. Why? The type parameter is obvious because the constructor declaration appears within the class Vector<E>. When the value is constructed, of course,

Vector<Integer> to a Vector<Number>, even though Integer values *may* be assigned to Numbers. You might think about why this is the case: Why is this the case?

[6] This constraint is related to the fact that every assignment to the array at runtime forces a runtime check of the data type. This check is necessitated by the fact that, over time, alternative arrays—arrays of different base types—can be assigned to a single array reference. Unfortunately Java 5 does not transmit parameterized type information to the running program. This means that parameterized types cannot be exactly checked at runtime, but they should be. The easiest way to enforce correct checking is to make it impossible to construct arrays of parameterized types. We expect future implementations of Java will address this issue, and that David Ortiz will be the American League Most Valuable Player, but good things take time.

[7] See more about privacy rights in Appendix B.8.

the actual parameter appears within the new statement (see the example on page 78). Within the constructor, the supporting array is allocated as a logically empty array of Object, and the initial value (type E) is cached away within the type. Is there any way that a non-E value can be stored in elementData, using this method? No. The method ensures the safety of the types of Vector's values. If we ask this question for each mutator method, and the answer is uniformly *No*, we can be sure the Vector contains only the correct types.

Now, consider a method that adds a value to the Vector:

```
public void add(E obj)
// post: adds new element to end of possibly extended vector
{
    ensureCapacity(elementCount+1);
    elementData[elementCount] = obj;
    elementCount++;
}
```

The Vector is expanded and the new value is appended to the end of the array. Is there a possibility of a type error, here? In a more extensive review, we might check to make sure ensureCapacity allows only type E values in elementData. (It does.) The second and third statements append an object of type E to the array, so there is no opportunity for a non-E value to be assigned. Incidentally, one might wonder what happens if the add method were called with a non-E type. It is precisely what happens in the StringVector example at the beginning of the chapter: an exception is raised at the site of the call, *during compilation*. We'll review this in a moment.

Next, we consider the get method. This accessor method retrieves a value from within the array.

```
public E get(int index)
{
    return (E)elementData[index];
}
```

By assumption (and eventually through proof) we believe the safety of the array is intact to this point of access. Every element is of type E, even though it appears in a more general array of Objects. The cast of the value, while required, will always hold; runtime type errors will never occur at this point. The method, then, returns a value of type E.

The set method is a combination of the prior two methods. It is useful to convince yourself that (1) the method is type-safe, and (2) the cast is necessary but always successful.

```
public E set(int index, E obj)
{
    Assert.pre(0 <= index && index < elementCount,"index is within bounds");
    E previous = (E)elementData[index];
    elementData[index] = obj;
    return previous;
}
```

It is, of course, possible to make type mistakes when using the `Vector` class. They fall, however, into two distinct categories. `Vector` implementation mistakes may allow elements of types other than `E` into the array. These mistakes are always possible, but they are the responsiblity of a single programmer. Good design and testing will detect and elminate these compile-time and runtime errors quickly. The second class of errors are abuses of the `Vector<E>` class by the client application. These errors are unfortunate, but they *will be reported at compile-time at the appropriate location*, one that focuses the programmer's attention on the mistake.

We can now test our new implementation on working code and then attempt to break it. In `LongWords3`, we consider, yet again, the implementation of our program that checks for long argument words. The implementation makes use of a `Vector<String>` as follows:

LongWords3

```
public static void main(String[] args)
{
    Vector<String> longWords = new Vector<String>();
    int i;
    for (i = 0; i < args.length; i++) {
        if (args[i].length() > 4) {
            longWords.add(args[i]);  // line 12
        }
    }
       ...
    for (i = 0; i < longWords.size(); i++) {
        String word = longWords.get(i);  // line 31
        System.out.println(word+", length "+word.length());
    }
}
```

Because the `Vector<String>` only contains `String` elements, there is no need for a cast to a `String` on line 31. This makes the running program safer and faster.

If the `add` statement were incorrectly written as:

```
longWords.add(args);  // line 12: woops!
```

the compiler would be confused—we're passing an *array* of `Strings` to a class that has only methods to add `Strings`. The resulting error is, again, a missing symbol (i.e. a missing method) error:

```
LongWords3.java:12: cannot find symbol
symbol  : method add(java.lang.String[])
location: class java.util.Vector<java.lang.String>
                longWords.add(args);
                         ^
```

The compiler cannot find an appropriate `add` method.

By the way, you can get the behavior of an unparameterized `Vector` class by simply using `Vector<Object>`.

4.2.3 Restricting Parameters

There are times when it is useful to place restrictions on type parameters to generic classes. For example, we may wish to implement a class called a `NumericalAssociation`, whose keys are always some specific subtype of `java.lang.Number`. This would, for example, allow us to make an assumption that we could compute the integer value by calling `intValue` on the key. Examples of declarations of `NumericalAssociations` include:

"numerical" means "numeric"

```
NumericalAssociation<Integer,String> i;
NumericalAssociation<Double,Integer> d;
NumericalAssociation<Number,Object> n;
NumericalAssociation<BigInteger,Association<String,Object>> I;
```

We declare the `NumericalAssociation` class as follows:

```
public class NumericalAssociation<K extends Number,V>
       extends Association<K,V>
{
    public NumericalAssociation(K key, V value)
    // pre: key is a non-null value of type K
    // post: constructs a key-value pair
    {
        super(key,value);
    }

    public int keyValue()
    // post: returns the integer that best approximates the key
    {
        return getKey().intValue();
    }
}
```

The `extends` keyword in the type parameter for the class indicates that type `K` is any subclass of `Number` (including itself). Such a constraint makes it impossible to construct an class instance where `K` is a `String` because `String` is not a subclass of `Number`. Since the `Number` class implements the `intValue` method, we can call `intValue` on the key of our new class. This type restriction is called a *type bound*.

Realize, of course, that the type bound bounds the type of class constructed. Once the class is constructed, the particular types used may be more restrictive. The class `NumericalAssociation<Double,Object>` supports keys of type `Double`, but not `Integer`. The `NumericalAssociation<Number,Object>` is the most general form of the class.

By the way, the same `extends` type bound can be used to indicate that a type variable *implements* a specific interface. An `Association` whose key is a `Number` and whose value implements a `Set` of keys is declared as

```
public class NSAssociation<K extends Number, V extends Set<K>>
       extends Association<K,V>
```

We have only touched on the complexities of parametric type specification.

4.3 Conclusions

When we write data structures, we often write them for others. When we write for others, we rarely know the precise details of the application. It is important that we write our code as carefully as possible so that semantic errors associated with our data structures occur as early as possible in development and as close as possible to the mistake.

We have seen, here, several examples of generic data types. The `Association` class has two type parameters, and is typical in terms of the simplicity of writing generalized code that can be tailored for use in specific applications. Formal type parameters hold the place for actual type parameters to be specified later. Code is written just as before, but with the knowledge that our structures will not sacrifice the type-safety of their client applications.

There is one "fly in the ointment": we cannot construct arrays of any type that involves a formal type parameter. These situations are relatively rare and, for the most part, can be limited to the some carefully crafted code in the `Vector` class. When array-like storage is required in the future, use of the extensible `Vector` class has one more advantage over arrays: it hides the difficulties of manipulating generic arrays.

Type bounds provide a mechanism to constrain the type parameters that appear in generic class specifications. These bounds can be disturbingly complex.

The remainder of this text demonstrates the power of the generic approach to writing data structures. While the structure of these data types becomes increasingly technical, the nature of specifying these structures in a generic and flexible manner remains quite simple, if not natural.