Lecture 9: Generics

Sam McCauley
Data Structures, Spring 2026

Different Kinds of Lists

We've seen three ways to store a list of `int`s: an `ArrayListInt`, a `LinkedListInt`, and a `DoublyLinkedListInt`.

What if we want to store a list of other things? One could imagine a list of type `double` or `char`, or even a list of objects. In fact, our original motivation when we began discussing objects was storing a list of `Students`. This will be the topic of discussion today. On the way, we'll encounter a few additional features of Java.

Extending our Code. Let's start with a simple example. Let's say we want to write `LinkedListStudent`, a class which has largely the same functionality as `LinkedListInt`, but stores `Students`. As it turns out, this works out very smoothly. First, we update our `NodeInt` class to a `NodeStudent` class that has data of type `Student`:

```
public class NodeStudent{
    private Student data;
    private NodeStudent next;

    //most methods are unchanged except getData and setData:
    public Student getData(){
        return data;
    }
    public void setData(Student newData) {
        data = newData;
    }
}
```

Notice that we didn't really change much here: `getData()` and `setData()` do exactly the same thing as they did in `NodeInt`. All we did was change data types from `int` to `Student`.

The `LinkedListStudent` class works with similar changes.

```
public class LinkedListStudent {
    private NodeStudent head;
    private int numElems;

    // most methods are the same, except we need to change the
    // return type of any methods that return list elements
```

```
7
8       // returns the Student stored in slot index
9       public Student get(int index) {
10          checkInBounds(index);
11          NodeStudent current = getNode(index);
12          return current.getData();
13      }
14
15      // sets slot index to store newElement
16      // returns the Student previously stored at index
17      public Student set(int index, Student newElement) {
18          checkInBounds(index);
19          NodeStudent current = getNode(index);
20          Student oldValue = current.getData();
21          current.setData(newElement);
22          return oldValue;
23      }
```

Why This is Not a Good Strategy. The internal structure of a Linked List is the same no matter what it stores. All we changed to get from `LinkedListInt` to `LinkedListStudent` is some parameter types and return types—in all cases, we just changed the type to be whatever the type of each element is (from `int` to `Student`).

This extra work makes no sense! We shouldn't have to make a new linked list each time we have a new type we want to store in our list. We're not doing anything different; all of this work is just to make Java happy with types.

What We Want to Do. What we'd really like to do is set up our methods independent of types. Basically, tell Java: "this linked list is going to store elements that are all of some type. When the user calls `get()`, the return type will be whatever the type of these elements are."

This is exactly what Generics do in Java. We will write code that will work for *any* type. When the user creates a list, they will specify the type of items in the list. Java will then automatically "replace" all relevant types. So when the user creates a list of `Students`, then `data` in each node will be of type `Student`; the return type of `get()` will be `Student`; and the return type and parameter of `set()` will be of type `Student`. Or, the user can create a list of `doubles`, and the exact same replacements will be made.[1]

[1] We will see that there are some caveats when using primitive types with generics: we'll have to do a little extra work to really get a list of `doubles`.

Generics

Let's build a class `LinkedList.java` that stores a singly linked list. We'll also need a class `Node.java` to store nodes for our list.

The first thing to do is to tell Java that we are using generics in the class (Java needs to know about this up front). The following code tells Java that we are going to use E to denote a generic type; E will be filled in later with an actual type. Let's look at the two class declarations

```
1 public class Node<E> {
```

```
1 public class LinkedList<E> {
```

These "angle brackets" tell Java that we are using a generic type E throughout the class description. You technically can name these generic types whatever you want, but you should always name them a single capital letter (this is a nearly-universal style choice).

Let's look ahead a bit. When we create a `LinkedList` object, we need to explicitly tell Java what type we want it to use. To make a linked list of `Students`, we would write

```
1 LinkedList<Student> myList = new LinkedList<Student>();
```

Then, Java would run the code as if all instances of E were replaced with `Student`.

Now, we can use E as if it were a type when writing our class.

```
1  public class Node<E> {
2      private E data; //create instance variable of type E
3      private Node<E> next; //points to another Node<E>
4
5      public E getData() {
6          return data;
7      }
8
9      public void setData(E newData) {
10         data = newData;
11     }
12     ...
13 }
```

We can change the rest of our linked list similarly. Let's look at a couple interesting methods.

```
1  public class LinkedList<E> {
2      private Node<E> head;
3      private int numElems;
4
5      // sets slot index to store newElement
6      // returns the element previously stored at index
```

```
7       public E set(int index, E newElement) {
8           checkInBounds(index);
9           Node<E> current = getNode(index);
10          E oldValue = current.getData();
11          current.setData(newElement);
12          return oldValue;
13      }
14
15      // returns node at position index in the list
16      // assumes that index satisfies 0 <= index < numElems
17      private Node<E> getNode(int index) {
18          Node<E> current = head;
19          for (int skips = 0; skips < index; skips++) {
20              current = current.getNext();
21          }
22          return current;
23      }
24  }
```

In all cases, we've just replaced the return type with E.

One thing to notice: in `LinkedList.java`, our nodes are *themselves* of generic type: a `LinkedList<E>` maintains its structure using a `Node<E>`. We can see that every time we create a node, its type is `Node<E>`.

Doubly Linked List. We can translate `DoublyLinkedListInt.java` to be able to handle arbitrary types using a similar strategy.

First, the `DLLNode.java` file.

```
1  public class DLLNode<E> {
2      E data;
3      DLLNode<E> next;
4      //...and so on
5  }
```

Then, `DoublyLinkedList.java`.

```
1  public class DoublyLinkedList<E> {
2      private DLLNode<E> head;
3      private DLLNode<E> tail;
4      private int numElems;
5
6      private DLLNode<E> getNode(int index) {
7      //... and so on
8  }
```

See the files `DoublyLinkedList.java` and `DLLNode.java` for details.

Generics and Primitive Types

The following code gives an error when we attempt to compile it.

```
1  LinkedList<double> listOfDoubles = new LinkedList<double>();
```

In Java, a generic type can only be replaced by a class, not a primitive type.

Fortunately, there is an easy way around this. Each primitive type in Java has a "wrapper class:" basically, it is a class whose sole purpose is to hold a variable of a primitive type.

Primitive Type	Wrapper Class
int	Integer
double	Double
boolean	Boolean
char	Character

Each of these classes has a way to translate a value of the primitive type into the corresponding wrapper class object. This allows us to easily translate between the primitive type, and an object that contains the same information. (We will see soon that there is another, easier way to accomplish this—this example is here to highlight what these classes do.)

```
1  int x = 0;
2  double y = 10.3;
3  Integer xObject = Integer.valueOf(x);
4  Double yObject = Double.valueOf(y);
```

These wrapper classes also have methods to do the reverse:

```
1  Integer xObject = Integer.valueOf(7);
2  Double yObject = Double.valueOf(4.2);
3  int x = xObject.intValue(); //7
4  double y = yObject.doubleValue(); //4.2
```

With this in mind, we can fix our issue: rather than a list of `doubles`, we can create a list of `Doubles`.

```
1  LinkedList<Double> listOfDoubles = new LinkedList<Double>();  //OK!
```

Autoboxing. As described so far, these lists are very annoying to use. Let's say I do make a list of `Doubles`. Each time I interact with it, I would need to translate from a `double` to a `Double`; something like the following:

```
1    //add y (a double) to the end of our list of Doubles
2    listOfDoubles.add(Double.valueOf(y));
3
```

```
4    //store the entry in slot 3 of our list into a new variable z
5    double z = listOfDoubles.get(3).doubleValue();
```

Fortunately, we do *not* need to do this in Java. There is a feature called "autoboxing" which allows us to skip this step. Java handles all relevant translation on the back end.

Autoboxing means that you can freely use values of primitive types, and objects of the corresponding wrapper class, as being equivalent. Java will convert between them automatically.

```
1 Integer x = 10; //works!
2 int y = x - 10; //also works!
```

This means that we can make a `LinkedList<Double>` object, and then store and retrieve values as if they were `doubles`.

```
1 LinkedList<Double> listOfDoubles = new LinkedList<Double>();
2 listOfDoubles.add(10.3);
3 listOfDoubles.add(-1.0);  // -1 won't work: Java cannot cast and
     autoboxsimultaneously
4 double x = 1.2;
5 listOfDoubles.set(0, x);
6 double y = listOfDoubles.get(0);
```

Note that the wrapper class *is* being used: Java is converting 10.3, -1, and 1.2 all to `Doubles` on the back end.

ArrayList with Generics, and Objects in Java

You may have noticed that we have not yet mentioned translating our `ArrayListInt` class to work with generic types. That's because there is a wrinkle here.

First Attempt, and the Problem. Let's start using generic types to rewrite `ArrayListInt`. It would probably start like this.

```
1 public class ArrayList<E> {
2     private E[] arr;
3     private int numElems;
4
5     public ArrayList() {
6         arr = new E[1];
7         numElems = 0;
8     }
```

Unfortunately, we have already run into an issue: in Java, you cannot create an array of generic type.

The reasons for this restriction involve the details of how generics are implemented; we won't go over this in this course. In any case, this code will not compile.

Objects in Java. In Java, there is a special class called `Object` that is built into the language.

Every object in Java is also an `Object`: this means that every `String` is an `Object`, every `Student` is an `Object`, every `CoinStrip` is an `Object`, every `LinkedList` is an `Object`, and so on.

When we say "every `Student` is an `Object`" we mean it in the same way that one would say "every triangle is a shape." "Shape" is more general than "triangle."

The `Object` class in Java has two methods that we've seen: `.toString()` and `.equals()`. This is why we mentioned these methods yesterday as special methods.

When we say that every object is an `Object`, bear in mind that primitive types are not `Objects`. An `int` is not an `Object` (though an `Integer` is).

Using Objects to Help. We aren't allowed to create arrays of generic type in Java. But we do know something about our array: whatever is contained, it is an `Object`. (Remember that we'll use the wrapper class for any primitive types.) What if we make an array of `Objects`?

```
1  public class ArrayList<E> {
2      private Object[] arr;
3      private int numElems;
4
5      // creates an array with 1 slot, sets number of elements to 0
6      public ArrayList() {
7          arr = new Object[1];
8          numElems = 0;
9      }
10 }
```

Bear in mind that our class still uses generics: we want to enforce that every element of our `ArrayList<E>` has type E. We will bear this in mind when creating our methods.

Let's look at `get()`. We find the array element at `index`. However, this element has type `Object`. Therefore, we *cast* it to type E before returning it.

(Remember that "casting" means converting a value of one type into another. We saw this before casting between, say, `int`s and `double`s, as in `double y = (double)x/2;`.)

The code for `get` is as follows.

```
1  public E get(int index) {
2      checkInBounds(index);
3      return (E) arr[index];
```

```
4 }
```

Note that this conversion is "unsafe." Usually, the Java compiler checks to make sure that your types always match. Here, it can't: you are taking an `Object`, and casting it to type E. This won't work for all `Objects`—if cast an `Object` that is not of type E, we will have a runtime error.

For this reason, Java gives a warning:

```
Note: ArrayList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

In general, you should take this warning seriously, and try to remove the underlying issue (see the discussion below on "raw types").

In our case, however, it is not actually a problem. We will only ever place items into the array using the `add()` methods and `set()` method; both of which only accept an element of type E. We can be confident that this cast will work for this data structure. In the source code, there is a special command that tells the Java compiler to ignore this warning (you do not need to know this command, and should not use it in your code in this class).

Let's continue with `set`. Again, we use casting to retrieve the element. Note that we do *not* need to cast when storing `newElement` in our array, since `newElement` is already an `Object`.

```
1 // sets slot index to store newElement
2 // returns the element previously stored at index
3 public E set(int index, E newElement) {
4     checkInBounds(index);
5     E ret = (E) arr[index];
6     arr[index] = newElement;
7     return ret;
8 }
```

The rest of the methods are similar. You can see the full code in the posted `ArrayList.java`.

Raw Types If you leave out the generics, you are creating what is called a "raw type."

```
1 LinkedList newList = new LinkedList();
```

This will, in short, act like a `LinkedList` of `Objects`. This is not a critical bug, but it is not ideal—it means that Java will not be able to do type checking for you.

In this course, raw types are not allowed: you must always specify a type when using generics.

Oftentimes, if you do accidentally forget to specify a type, Java will give a warning. It is the same warning we saw above about unchecked types.

If we run the following code:

```
1 LinkedList listOfDoubles = new LinkedList();
2 listOfDoubles.add(1.1);
```

We see the following message:

```
> Note: LinkedList.java uses unchecked or unsafe operations.
> Note: Recompile with -Xlint:unchecked for details.
```

The second line of that message gives a recommendation. It's actually quite a useful recommendation: this flag tells the compiler to tell you *where* the potential issues are. Let's run the compiler with this flag on, we see the following:

```
> javac -Xlint:unchecked LinkedList.java
  LinkedList.java:139: warning: [unchecked] unchecked call to add(E) as a
     member of the raw type LinkedList
                 listOfDoubles.add(1.1);
                                   ^
```

It points to exactly the problem: we added 1.1 (a `Double`) to a `LinkedList` rather than a `LinkedList<Double>`.

If you see this warning in this course, you should run the compiler again with the `-Xlint:unchecked` flag, as in the above. You should find where the issue is, and the majority of the time you should fix the issue so there is no warning.