Lec 9: Generics

Sam McCauley

February 27, 2026

Admin

- Midterm next Friday
  - Practice midterm posted soon

- Any questions?

Holding Different Kinds of Items in a List

What we've done so far

- We made three data structures to store a list of `int`s
  - `ArrayListInt`, `LinkedListInt`, `DoublyLinkedListInt`
- What if we wanted a list of something else?
- Our motivation for classes was a list of `Student`s. How can we do that?

List of Students

- Let's create `LinkedListStudent.java` to store a list of `Students`
- We are changing the type of the data held in each node. That's it!
- Let's look at what we need to change in the code.
- Can make similar changes for `DoublyLinkedListStudent` or `ArrayListStudent`.
- Can also make data structures for other types.

This is a bad strategy

- We're wasting our time here! All of this code does *exactly* the same thing
- The *only* thing we changed was the type of what we are passing around
- Surely we don't need a new data structure just to tell Java that the data we're passing around has a new type?

What we Want to Do.

- Write our methods to actually handle the back end of our data structure: things like `getNode()` or `ensureCapacity()`
- Tell Java: there will be some type of data in each node. You should do the same thing whatever that type is.
  - When we need a list, we'll tell Java what type of list we need
  - Java will fill in the appropriate type. In short, it will build the `LinkedListStudent` (etc.) for us!
- Java can do this using "Generics"

Generics

Generics

- Goal: we will tell Java “the data in this Linked List wil be of some type; we’ll tell you what it is when we actually make the list”
- Notation for this: the type goes in angle brackets at the beginning of the class declaration
- Generic types in Java should *always* be single capital letters. (Nearly-universal style choice)
- Let’s build `Node.java` and `LinkedList.java` to use Generic types.
- Similar strategy works for Doubly Linked Lists (code is posted).

Generics and Primitive Types

Using Generics with Primitive Types

- Let's say we want to store a Linked List of `doubles`

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

- This gives an error! In Java, we can only use a *class* for a generic type; primitive types are not allowed.

- Easy way around this: "wrapper classes"

Wrapper Classes

- Each primitive type has a corresponding wrapper class: a class whose sole purpose is to hold a value of that primitive type

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

- Use the wrapper class in place of the primitive type when using generics

Converting Between Primitives and Wrapper Classes

We can use the following notation to convert between a primitive class and the corresponding wrapper class. (Don't worry about memorizing this: we will momentarily see an *easier way*.)

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

```
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
```

Converting Between Primitives and Wrapper Classes

Now, we can make a linked list of `doubles`—or should I say, `Doubles`.

```
1 LinkedList<Double> listOfDoubles = new LinkedList<Double>();
2 listOfDoubles.add(Double.valueOf(2.3));
3 double y = listOfDoubles.get(0).doubleValue(); // 2.3
```

Autoboxing

- Manually converting between `double` and `Double` is tedious:

```
1 listOfDoubles.add(Double.valueOf(y));          // annoying!
2 double z = listOfDoubles.get(3).doubleValue(); // annoying!
```

- Java's autoboxing feature handles the conversion automatically
- You can use primitive values and wrapper class objects interchangeably:

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

```
1 LinkedList<Double> listOfDoubles = new LinkedList<Double>();
2 listOfDoubles.add(10.3);          // Java converts to Double
    automatically
3 double y = listOfDoubles.get(0);  // Java converts back
    automatically
4 listOfDoubles.add(10.3);          // Java converts to Double
    automatically
```

Autoboxing: Care with Casts

```
1 LinkedList<Double> listOfDoubles = new LinkedList<Double>();
2 listOfDoubles.add(3.0);      //OK!
3 listOfDoubles.add(3);  //Error!  (Java cannot cast and
     autobox automatically)
```

- Note when autoboxing: the wrapper class *is* being used; you just do not need to name it explicitly.

- Java converts these values to `Doubles` on the back end

ArrayLists, Generics, and Objects

ArrayLists with Generics

- I've been carefully sidestepping what happens with `ArrayLists`
- Basic idea is the same: we want to replace all of our items with generic types
- However, we're going to run into an issue for `ArrayLists` specifically

First Attempt at `ArrayList<E>`

- Let's try to rewrite `ArrayListInt` with generics

```
1 public class ArrayList<E> {
2     private E[] arr;
3     private int numElems;
4
5     public ArrayList() {
6         arr = new E[1]; // ERROR: can't create array of
              generic type!
7         numElems = 0;
8     }
9 }
```

- Java does *not* allow creating arrays of generic type
- We need a workaround

Objects in Java

- Java has a special built-in class called `Object`
- Every object in Java is also an `Object`
  - Every `String`, every `Student`, every `LinkedList`, …
  - Think of it like: “every triangle is a shape”—`Object` is the most general class
- The `Object` class has two methods we’ve seen: `.toString()` and `.equals()`
- Note: primitive types are *not* `Objects`
  - But their wrapper classes are!

Using `Object[]` to Store Elements

- Since we can't create an array of type `E[]`, let's use `Object[]` instead

```
1 public class ArrayList<E> {
2     private Object[] arr;
3     private int numElems;
4
5     public ArrayList() {
6         arr = new Object[1]; // OK!
7         numElems = 0;
8     }
9 }
```

- In pairs: how does this affect the rest of our methods? What do `get()` and `set()` look like?

get() and set()

- What should their return type be?
    - Still type E
- When we access the array, we get an `Object`
- Need to cast it to type E before we return

`get()` and `set()` with Casting

```java
public E get(int index) {
    checkInBounds(index);
    return (E) arr[index]; // cast from Object to E
}
```

```java
public E set(int index, E newElement) {
    checkInBounds(index);
    E ret = (E) arr[index]; // cast when retrieving
    arr[index] = newElement; // no cast needed when storing
    return ret;
}
```

- Java warns: `ArrayList.java uses unchecked or unsafe operations`
- Why do you think this is “unsafe?”

Unchecked or Unsafe Operations

- Java is worried about casting an `Object` to an `E`, since not all `Objects` have type `E`
- Can this ever be an issue for *us*? Can we ever be casting something that's not of type `E`?
    - No! We only put objects of type `E` into our array, so we will only get objects of type `E` out
- In this case we can safely ignore this warning. (In fact, the posted code tells Java to ignore this warning.)
- In general, you *should not* ignore this warning

Raw Types

- Omitting the generic type creates a “raw type”—avoid this!

```
1  LinkedList newList = new LinkedList(); //no type specified
```

- Acts like a `LinkedList<Object>`: Java can’t do type checking for you
- In this course: raw types are not allowed—always specify a type
- Java will warn you; you can (and *should*) get details with the `-Xlint:unchecked` flag:

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

```
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);
                              ^
```