Lec 10: static

Sam McCauley

March 2, 2026

Admin

- Midterm Friday
  - Practice midterm posted as soon as possible
- Wednesday: finish lectures and fill in larger code file
- Lab time: optional review/office hours
  - Bring questions!

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
```

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);
                              ^
```

Multiple Generic Types

Multiple Generic Types in One Class

- Same idea works when class has multiple generic types
- Separate the types (in the angle brackets) with a comma
- Let's make a `Pair` class; works like a two-element tuple in Python class

equals() Method

equals() method

- In Java, every time you compare objects you should use the `equals()` method

- Every `Object` has one

```
1 public boolean equals(Object other) {
```

How `equals()` is Used

- Used to see if two objects are equivalent
- Example: in `indexOf()` and `contains()` of `ArrayList.java`
- We did this in `LinkedList.java` and `DoublyLinkedList.java` as well

How `equals()` is Used

```java
// find the first occurrence of element and return its index
// if there is no occurrence, return -1
// Note: uses .equals() instead of == to compare objects
public int indexOf(E element) {
    for (int index = 0; index < numElems; index++) {
        if (element.equals(arr[index])) {
            return index;
        }
    }
    return -1;
}
// returns true if element is in the list; false otherwise
public boolean contains(E element) {
    return indexOf(element) != -1;
}
```

- Essentially all Java library data structures use `equals` in the same way

Writing an `equals()` method

- Let’s say we work at a museum
- Want to keep track of works of art
- Each has a title, artist, year, and reference number
- How do we determine if two works of art are equal?

Writing an `equals()` method

```
1  public class WorkOfArt {
2      private String title;
3      private String artist;
4      private int year;
5      private int reference;
6
7      public WorkOfArt() {
8          title = "";
9          artist = "";
10         year = 0;
11         reference = 0;
12     }
13     //assume standard getters and setters
```

What should `equals()` do?

Any suggestions?

1. Check if `title` is the same
2. Check if `reference` is the same

- Are there use cases where the first strategy is better?
- Are there use cases where the second strategy is better?

Checking Title Makes Sense When...

- Let's say the purpose of our `WorkOfArt` class is for users to search for a specific work
- They always search by title
- Then `equals()` lets us search in an `ArrayList<WorkOfArt>` directly

Checking Reference Number Makes Sense When...

- Let's say the purpose of our `WorkOfArt` class is for back-end museum curators to search for more information on a specific piece
- They always search by reference number
- Then `equals()` lets us search in an `ArrayList<WorkOfArt>` directly

Comparing the Strategies

- Takeaway: the `equals()` method depends on *how the method is used;* it's not a matter of right or wrong
- That said, in a vaccuum the second strategy is probably a better default
    - I'd say two "unequal" works of art can have the same name
- Let's implement the second strategy

equals() method parameter

- Its type is `Object`
- This is because the method is common to all `Objects`
- How do we compare (say) a `WorkOfArt` to an `Object`?
- Idea: if the other `Object` is not a `WorkOfArt`, we will always say `false`; otherwise we compare their reference
- Use `instanceof` keyword to check

static

static variables

```
1 public class Shipment {
2     private static int total_num_shipments;
```

- In Java, we have seen that each object of a class has its own copy of every instance variable
- If we use `static`, it is a “static variable” instead of an instance variable
- There is only *one* copy shared across all objects of the class
- Can be accessed just like instance variables
- First question: why?

First Common Use Case

- Keep track of a “global” value (common to all objects)
- Let’s say we run a shipping company. We’ll create a `Shipment` object to keep track of the data from each shipment
- Also want to track the total number of shipments so far. We’ll use a `static` variable

Global Value Example

```
1  public class Shipment{
2      private static int total_num_shipments = 0;
3      private int shipment_ID;
4      private ArrayList<String> items;
5
6      public int getNumShipments() {
7          return total_num_shipments;
8      }
9  }
```

Continuing Example

- Each time we make a new `Shipment`, we want to update the tracker of how many there have been
- We'll do this in the constructor

```
1 public Shipment(int newID) {
2     shipment_ID = newID;
3     items = new ArrayList<String>();
4     total_num_shipments++;  //we have a new Shipment!
          Increment the counter
5 }
```

Second Usage Example: Storing Constants

- Sometimes we want to store constants common to all objects of the class
- In our `WorkOfArt` class, it's likely useful to store the maximum possible reference number
- If we have `1000` works of art, we don't need `1000` copies of the maximum reference number! We'll use the `static` keyword so there's only one copy

`final` keyword

- We'll also use the `final` keyword in this example

- If a variable is `final`, then it cannot be changed after it is initialized

Second Example in Code

```
1  public class WorkOfArt {
2      private String title;
3      private String artist;
4      private int year;
5      private int reference;
6
7      public static final int min_reference_number = 0;
8      public static final int max_reference_number = 999999;
```

public static variables

- Unlike instance variables, it is OK if static variables are `public`
    - Especially if they are `final`
- We don't need the max reference number to be "hidden" from other classes, and `final` means it cannot be changed
- You may want to make it `private` for the sake of a "clean API": in short, static variables have the same rules as methods in terms of access modifiers

Accessing static variables directly

- Since static varibles don’t need an object, we can access them without using a specific object

- Notation: name of the class, then ., then the name of the variable

```
1 public static void main(String[] args) {
2     System.out.println(WorkOfArt.max_reference_number); //
          999999
3 }
```

Static Methods

Overview

- Same idea as static variables
    - Not associated with a specific *object*, associated with the class in general
- This is why `main` is `static`: we don't need an object to run the `main` method
- Can call a `static` method with the same notation: using class name + `.` + method name

Static method rules

- You cannot access instance variables from a static method. Why?
- You cannot call a non-static method from a static method. Why?
- It is OK to call a static method from a non-static method.

Static method Examples

- Used when functionality is associated with a specific class, but does not rely on data from a specific object

- Example from earlier today:

```
1  Integer y = Integer.valueOf(10);
```

Static method Examples

- Primitive type wrapper classes have quite a few staic methods

- Example: this method converts a `String str` into an `int`

```
1 int number = Integer.parseInt(str);
```

Static Method: Second Example

- The `Math` library consists entirely of static methods

- We don't need a `Math` object, or any stored data, to do math! But it's useful to have these methods grouped together.

```
1  double power = Math.pow(8, 2.5); //raise 8 to the 2.5th power
2  double root = Math.sqrt(2);  //store square root of 2 in root
```

Setting Values Early

- This code is a little unusual when you think about it:

Setting Values Early

- So is this (maybe even more so). When is `= 0` run?

```
1  public class Shipment{
2      private static int total_num_shipments = 0;
3      private int shipment_ID;
4      private ArrayList<String> items;
```

Setting Values Early

- Long story short:
    - For instance variables, Java sets them at the beginning of the constructor.
    - For static variables, Java sets them when setting up the program