Lecture 10: static

Sam McCauley
Data Structures, Spring 2026

Multiple Generic Types

So far, we have seen examples of classes with one generic type: a list where the generic type represents what kind of elements are in the list.

It is possible to have classes of multiple generic types in Java. The same notation with angle brackets extends to this case; we separate the types with commas.

Let's say we want a simple `Pair` class that stores two elements—think of this like a tuple in Python with two elements. We'll use the letters F and S for the two generic types (as opposed to E which we used for the elements of the list).

```java
public class Pair<F,S> {
    F first;
    S second;

    public Pair() {
        first = null;
        second = null;
    }
    public Pair(F first, S second) {
        this.first = first;
        this.second = second;
    }
    public F getFirst() {
        return first;
    }
    public S getSecond() {
        return second;
    }
    public void setFirst(F newFirst) {
        first = newFirst;
    }
    public void setSecond(S newSecond) {
        second = newSecond;
    }
}
```

Now, we can use it in our code. Let's say we want to pair an ID with a name:

```java
Pair<Integer, String> newPair = new Pair<Integer, String>(52534,
            "Antoinette");
System.out.println(newPair.getSecond() + " has ID number " + newPair.
    getFirst());
```

equals Method

Method Details. A little while ago, we mentioned that you should always compare objects with the `.equals()` method. We are ready to go into more detail of how these are used, and how they should be written.

Every `Object` (here we are referring to the Java class `Object`) has an `.equals()` method. Its declaration looks like this:

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

How equals() is Used. This method is used universally to test for equality of objects. For example, in the `ArrayList` class, the `contains()` and `indexOf()` methods traverse the list, using `.equals()` on each item in the list to determine where the item first occurs in the list. Here is the code from `ArrayList.java`:

```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;
}
```

The Java `ArrayList` library works exactly the same way.

This means that `indexOf()` and `contains()` *will not work* for elements of a class you write unless you implement your `.equals()` method. The problem is even bigger than these list methods: essentially every Java data structure will call `.equals()` to see if two objects of your class are equivalent.

Writing Your Own equals() Method. The first question is what makes two objects of your class equal. This is completely up to you: whatever you put in your `equals()` method will be used by other Java classes to determine if two objects of your class are considered to be the same.

Let's say we work at a museum, and we want to keep track of art stored at the museum. Each work of art has a title, artist, year, and reference number.

```java
public class WorkOfArt {
    private String title;
    private String artist;
    private int year;
    private int reference;

    public WorkOfArt() {
        title = "";
        artist = "";
        year = 0;
        reference = 0;
    }
    //assume standard getters and setters
```

How do we determine if two works of art are the same—in other words, what is our strategy for implementing `equals()`? Let's look at two reasonable suggestions. For each suggestion, let's look at what this means if we store our collection as an `ArrayList` of our `WorkOfArt` objects—bear in mind that while we are discussing an `ArrayList` specifically, many Java data structures call `equals()` and will be affected by this design decision.

In our first strategy, `equals()` returns `true` if both works of art have the same `title`. This means that if we call `contains()`, we will learn if a work in the collection has *the same title*.

In our second strategy, `equals()` returns `true` if both works of art have the same `reference number`. This means that if we call `contains()`, we will learn if a work in the collection has *the same reference number*.

Each of these strategies can be reasonable, depending on usage!

Let's say that we want to write a method that helps a museum visitor determine if a work of art they want is in the collection. They will be searching by title—so we want `equals()` to search items by title.

```java
//Assume we have an ArrayList<WorkOfArt> ourList already
//declared and instantiated, that contains all the museum's works of
   art
//Assume that .equals of WorkOfArt returns true if they have a matching
    title
System.out.println("Please enter the title of the art you want to
   search for: ");
String artTitle = reader.nextLine();
```

```java
//make an object with the input title so that we can search
WorkOfArt newWork = new WorkOfArt();
newWork.setTitle(artTitle);  //ensure our new object has the correct
    title
if(ourList.contains(newWork) ) {
    System.out.println("Work with correct title found!");
}
```

This works well to help a user search by title. But in other contexts, it could be a problem: if two works of art have the same title, the code we have written has no way to distinguish them. Notice that we did not have to fill in the other entries of `newWork`: we only need the title to find the object we want.

Now, consider the second strategy: we consider two objects equal if their reference numbers are equal. If we ensure that all works of art have unique reference numbers, our results are less ambiguous: we know if we have found exactly what we are looking for. But, we need to fill the reference number in correctly to be able to compare objects; this would not work for the previous example. Let's assume that we want to create lookup functionality for a museum curator. The following code asks the curator for a reference number, and gives the artist and year of the matching work.

```java
//Assume we have an ArrayList<WorkOfArt> ourList already
//declared and instantiated, that contains all the museum's works of
    art
//Assume that .equals of WorkOfArt returns true if they have a matching
     reference number
System.out.println("Please enter the reference number of the work you
    want: ");
int refNum = reader.nextInt();
WorkOfArt newWork = new WorkOfArt();
newWork.setReference(refNum);  //ensure our new object has the correct
    reference number
if (ourList.contains(newWork)) {
    int idx = ourList.indexOf(newWork);
    WorkOfArt found = ourList.get(idx);
    System.out.println("Artist: " + found.getArtist() + ", Year: " +
        found.getYear());
}
```

Overall, the second strategy is likely the better choice: two works of art can have the same name without being "equal." But it's worth going over both because there's no right or wrong answer: you should define equality for a class in a way that makes sense for the underlying data, and in a way that is useful for you to use.

Implementing equals() and `instanceof`. Let's implement both versions of `equals()` from the above discussion.

In the first strategy, we want to test if the two objects have the same `title`. Let's start filling in the method:

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

We have already run into a problem! We would like to write something like

```
    if(!title.equals(other.getTitle()) ) {
```

But `other` is an `Object`—so `other` does not have a `title` field, nor a `getTitle()` method. At the root of this problem is how Java stores objects. `other` really stores a reference to a place in memory, and the type of `other`—here, the type is `Object`—keeps track of how the memory is laid out when we follow that reference. Since an `Object` has no `getName()` method, the above code will not compile.

We should discuss briefly *why* `other` is an `Object` as opposed to a `WorkOfArt`. In Java, `equals` can compare *any* two objects, even if they are not of the same type—it can be beneficial that `equals` is so general. That said, when we get to inheritance, we will see that in fact it must be this way in Java. We can only have an `equals()` method universal to all objects if the parameter to `equals()` also accepts any object as a parameter.

Conceptually, we can handle this issue with a simple rule. Two works of art are equal if they have the same title, and a work of art is not equal to anything that is not a work of art.

```
WorkOfArt newWork = new WorkOfArt();
Student s1 = new Student();
System.out.println(newWork.equals(s1));  //false: a work of art is not
    "equal to" a student regardless of what data each stores
```

Java has a keyword that will help us here: `instanceof`. This keyword checks if an object is an instance of a given class.

With this, we have a plan. We will first check if the `Object` given to us by `equals()` is a `WorkOfArt`. If it's not, we just say `false` without checking its internal data. If they are both instances of `WorkOfArt`, then we compare their titles.

```
//Strategy 1:
public boolean equals(Object other) {
    if (!(other instanceof WorkOfArt)) {
        return false;
    }
    WorkOfArt otherWork = (WorkOfArt) other;
    return this.getTitle().equals(otherWork.getTitle());
}
```

Notice that we had to *cast* the other object to be a `WorkOfArt`, since `Objects` do not have a `getName()` method. We can do this cast confidently since we already ensured that `other` is a

`WorkOfArt`—we do not need to worry about a runtime error that would occur if `other` was not a `WorkOfArt`.

We can use a similar strategy to implement the `equals()` method to compare the reference numbers of the work of art. I should emphasize that this is a *choice*: you can only have one `equals()` method implemented for each class. If you want two different `equals()` methods, you will need to write two different classes.

```java
//Strategy 2:
public boolean equals(Object other) {
    if (!(other instanceof WorkOfArt)) {
        return false;
    }
    WorkOfArt otherWork = (WorkOfArt) other;
    return this.getReference() == otherWork.getReference();
}
```

Most `equals()` methods in Java follow this pattern. First, we ensure that the argument's type matches the type of our class. If not, we return `false`; if so, we compare the internal data of both objects using the class methods and return the result.

Static Variables

We have used the `static` keyword a number of times in Java: most notably, every time we write a `main` method. Let's look at static variables first, and then come back to `static` methods.

We've already seen that in Java, each object of a class has its own copy of each instance variable. So each `Student` has its own name, each `LinkedList` has its own `head` reference, and so on.

This is no longer the case if we mark a variable with `static`. There is only one copy of a static instance variable across all objects of the class. Every object of the class will see it as having the same value; if it is modified, it is modified for all objects of the class.

Note that these are no longer called "instance variables," since they are not associated with a specific instance of the class. Instead we will call them "static variables."

Let's look at two of the most common use cases for static variables.

Keeping Track of a Global Value. The first use case is when we have a variable that we specifically want to *share* between all objects of a class.

For example, let's say we run a shipping company. We have a class for each shipment, that looks something like the following:

```java
public class Shipment{
    private static int total_num_shipments = 0;
    private int shipment_ID;
    private ArrayList<String> items;
```

This means that each `Shipment` has a `shipment_ID` and a list of `items`. Then, *all* shipments share a variable denoting how many shipments there have been.

A static variable can be accessed just like an instance variable:[1]

```java
public int getNumShipments() {
    return total_num_shipments;
}
```

Now, we want to update this variable every time we make a new `Shipment`. We can do this in the constructor, since the constructor is called every time a new `Shipment` is instantiated.

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

To emphasize, there is a *single* `total_num_shipments` variable shared by all `Shipment` objects. If one object increments this variable, all objects will see its value as being one larger. This means that this variable is incremented exactly as many times as the `Shipment()` constructor has run.

Storing Constants (And the final keyword). The second use case is to store *constants* common to all objects of the class. For example, recall our `WorkOfArt` class. It would be very useful to store data about what a "reference number" is: for example, we'll store the minimum and maximum reference number.

For this use case, we often also use the `final` keyword. This keyword means that the value of the variable may not be changed.[2] Note that while `final` is often used for `static` variables, it can be used for instance variables as well.

[1]This method could—and probably should—be a static method.
[2]The behavior of `final` is a bit more complicated than this: you're allowed to initialize it by setting its value once, oftentimes in the constructor, but you cannot update it again after that. You can apply the `final` keyword to classes or methods as well; we will not go over those usages in this course.

```java
public class WorkOfArt {
    private String title;
    private String artist;
    private int year;
    private int reference;

    public static final int min_reference_number = 0;
    public static final int max_reference_number = 999999;
```

There's no need to store a separate copy of these numbers for each object—if we have 1 million works of art, we don't need 1 million copies of the maximum reference number. The `static` keyword means that only one is stored.

Public Static Variables: When to Use, and Accessing. When `static` is used to store constants, it can be good code style to make the variable `public` (this is what we did in the example above). This is in contrast to instance variables, which should always be `private`.

Let's briefly discuss why. The reference numbers here are not data that needs to be "hidden" from other classes in any way: it's probably useful to allow the range of each reference number to be read by anybody. In this example in particular, the `final` keyword ensures that these values cannot be accidentally changed by another class.

Perhaps the most interesting aspect of public static variables is that they can be accessed directly, without using an object of the class. In fact, we can access these static variables even if no objects of the class exist! The notation for this is to first write the name of the *class*, then the `.` operator, then the name of the static variable.

In the following example, the main declaration is included to emphasize that there are no `WorkOfArt` objects created yet.

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

Static Methods

Static methods follow the same idea as static variables: rather than being associated with a specific object, they are associated with the class in general.

This is why `main` is a static method: we want to be able to call `main` without creating an object of the class.

The notation for calling a static method is similar to that of a static variable: we write the name of the *class* (rather than the object), then the dot operator, then the method.

Rules for Static Methods. Since a static method is not associated with a particular object, static methods can only access static variables. This makes sense: instance variables are only created when we instantiate an object, and a static method may be called before this occurs.

Looking at it another way, static methods are not associated with any particular object—so even if objects of the class exist, we would not know *which* instance variables should be returned.

For the same reason, static methods cannot call non-static methods without an object to call them on. So for example, inside `main()` you cannot call `get()` unless you call using an `ArrayList` object—writing `get(0)` alone would give an error, but `myList.get(0)` is perfectly fine.

However, non-static methods *can* call static methods.

Examples of Static Methods. Static methods are used when there is useful functionality for a class that does not rely on data for a specific class object.

One example we saw recently (before we went over autoboxing) is the following method:

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

`valueOf()` is a static method of the `Integer` class. It takes an `int` as argument, and returns an object of the wrapper class `Integer` that holds the same data. The code of this method belongs in the `Integer` class since it is a method for `Integers`, but the way it works does not depend on the value stored in a particular `Integer` object—in fact, we do not need an `Integer` object to call `valueOf()`.

The primitive type wrapper classes make use of quite a number of static methods. For example, the following code converts a `String str` into an `int`.

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

The `Math` library in Java, consisting of methods that do useful mathematical operations, consists entirely of static methods. For this usage, there's no need to create a `Math` object, and the way these methods work does not rely on any instance variables

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

Setting Values of Instance Variables

We've now seen a few examples where we declared and initialized or instantiated an instance variable or static variable at the same time. This is a bit unusual: when does this setting actually happen? So far, when Java code runs, we are always running code from some method, but this code is outside of any method.

The short answer is that for static variables, Java runs the code when setting up your program, and for instance variables, Java runs it right before the constructor. The longer answer can be found on the Oracle website here: https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html.