Iterators, Exceptions, and Try/Catch

Abhinav Ashar
CS 61B: Data Structures and Algorithms

February 25, 2019

1 Interfaces

Interfaces in Java are a special class that operate a bit differently from normal classes. Since the intuition behind this topic can be sometimes confusing, it is best to first understand the what are iterators, why do we use them, and how can we implement them.

What Are Interfaces?

- A special Java class
- Specifies what a class must do, but not how to do it. In a sense, the iterator is a blueprint for other classes. (Ex. A Dog interface might have a function bark(). This means that all Dogs can bark. However, the bark() of a Poodle, which is a subclass of Dog, is probably very different than the bark() of a German Shepard, another subclass of Dog. Thus, the Dog interface provided a blueprint for Poodle and German Shepherd by stating that they should have a bark method, but it did not specify exactly how bark works. That is left up to Poodle and German Shepard.
- Classes that utilize interfaces need to specify the code for methods in the interface

Why Do We Use Interfaces?

- Provides a layer of abstraction
- Creates natural groupings
- Adheres to polymorphism (related objects with distinct implementation) and encapsulation (putting related things in one module)
- Creates an is-a relationship. If a class A utilizes interface B, A is a B.
- From the example in the last section, imagine if we didn't have a Dog interface. Then, if we wanted to refer to all dogs, we would have to refer to Poodles and German Shepherds and Golden Retrievers and Pitbulls etc. Rather than trying to refer to all these different types of classes, which can get complicated, we instead create a Dog interface and put all these types of dogs under this over-arching Dog interface. Thus, we can just refer to Dog and it will work for any specific type of Dog.

How Do We Use Interfaces?

- When creating the interface class, use the **interface** keyword
- Use final, static variables. Interface variables are public, final, and static by default if you do not specify.
- Methods are public by default if you do not specify
- Create unspecified functions with no code inside of them (Ex. public void bark();)
- When a class wants to utilize an interface, it needs to use the **implements** keywords
- In Java, classes can implement multiple interfaces, but they can only extend one class

2 Iterators

Iterators often utilize interfaces. An iterator is an object which allows us to traverse a data structure in a linear fashion.

a) Iterators always have the following two methods: hasNext() and next()

b) hasNext() lets you check if there are any more items left to look at

c) next() lets you get the next item

d) Always check hasNext() before you use next()

e) If there is nothing next in the iterator and you use next(), the program will likely error

f) You often will have to implement hasNext() and next(). Understand what is the object you are trying to iterate over, and what does it mean to ”iterate” over the object. Then consider what tools you need to be able to iterate, such as a pointer indicating where you are. For example, if you are iterating over an array, it makes sense to use an *index* variable that keeps track of the index you are at in the array. Then, you will just increment the index as you call next to go to the next element in the array.

3 Exceptions and Try/Catch

An exception in Java is an unwanted event that interrupts the normal flow of a program. When an exception occurs, program execution gets terminated and we get an error from the system. By handling exceptions, we can do two primary things: **create insightful messages for debugging or deal with the error and let the program continue running**. For the first point, we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user. For the latter point, if we did not want to have the program stop exception, we catch the exception, deal with it accordingly, and let the program continue running. Try/Catch is a technique used to handle exceptions.

a) There are two types of exceptions: Checked Exceptions and Unchecked Exceptions

b) **Checked Exceptions** - compiler checks them before the program runs to see whether the programmer has handled them or not; will not compile if these exceptions have not been handled (Ex. IOException, ClassNotFoundException)

c) **Unchecked Exceptions** - these exceptions are not checked at compile-time, and it is up to the programmer to handle these exceptions and provide a safe exit (Ex. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException)

d) Try/Catch blocks are used to handle exceptions. One or many catch blocks normally come after the try block. Sometimes, you will also see the finally block at the very bottom.

```
try {
    doSomething();
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("caught array index exception");
} catch (Exception e) {
    System.out.println("caught an exception");
    throw e;
} catch (NullPointerException e) {
    System.out.println("caught null pointer exception");
} finally {
    System.out.println("in finally block");
}
```

If doSomething() throws an error, it could be caught by one of the catch blocks. Look at what error doSomething() throws, and see if that error is caught by one of the catch blocks. If not, see if a parent/ancestor of that error is caught by one of the catch blocks. For example, an ArithmeticException is a type of Exception, so we will execute the second catch block.

e) The order of catch blocks should be from more specific exceptions to more general exceptions. Thus, NullPointerExceptions and ArrayIndexOutOfBoundsExceptions should occur before Exception because they are types of Exceptions. If you don't follow this rule, then Java will throw a **compile-time error**. In the above example, you will see that if doSomething() throws a NullPointerException, then it will be caught by the second catch block, not the third, because NullPointerException is a type of Exception. Thus, there is no instance where the third catch block will be utilized. Java doesn't like this, and that's why it will throw a compile-time error.

f) Once an error is caught by a catch block, it will not be re-caught by any other catch block. If the catch block itself throws an error (like in second catch block in the example above), the error will not be caught unless it has its own try/catch block where that error is part of a try section inside of that catch block.

g) **ALWAYS** run the finally block at the end, which is either right before it executes the line of code after the try/catch block or it errors. A runtime error will terminate the execution of the program, but you must run the finally block before you throw that error. For example, the second catch block in the example throws an error that will not be handled. Right before the error is thrown and the program execution is stopped, go to the finally block and run it. Then, throw the error. I repeat: **ALWAYS** run the finally block before continuing on or terminating the execution of the program due to an error.