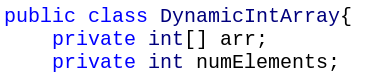
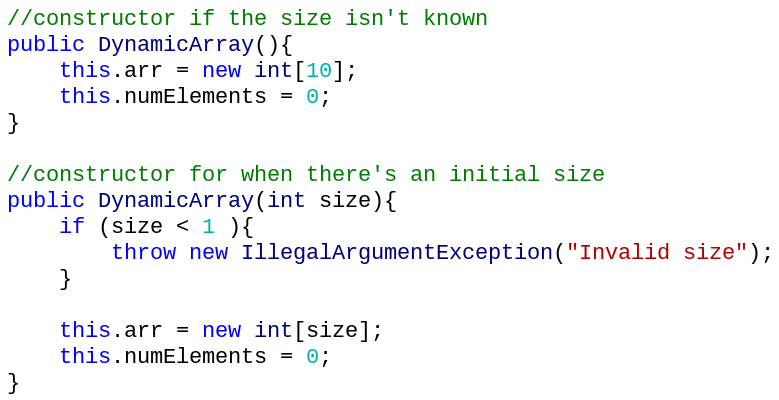
* Data structure

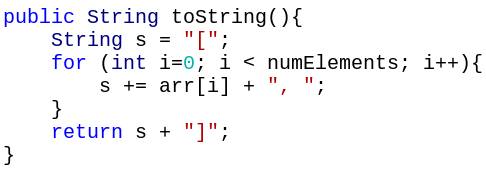
In computer science, a data structure is a specialized format for organizing and storing data.

* Dynamic-sized array class

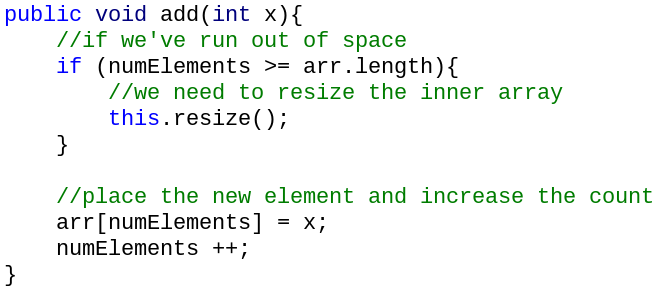






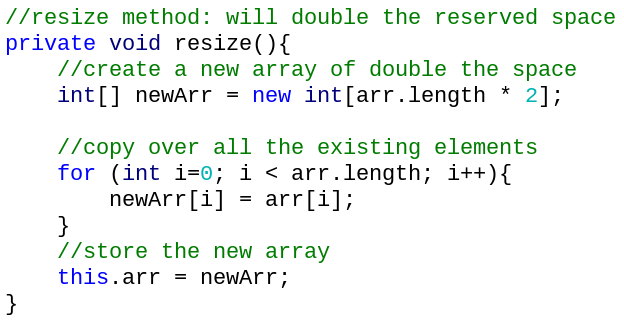


adding





resizing

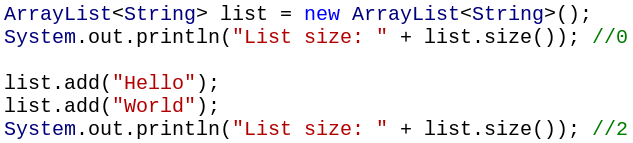


* ArrayList

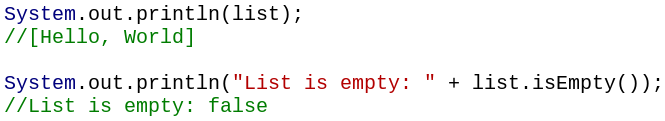
ArrayLists don’t have a fixed size

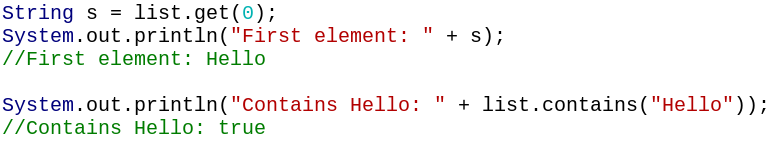
Can store any **reference** type

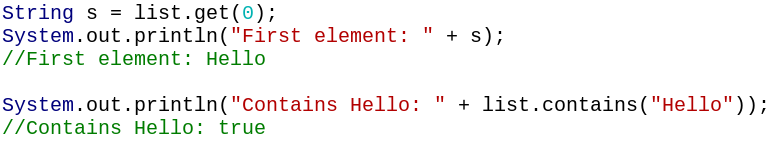
Can add, delete, and check if an object exists

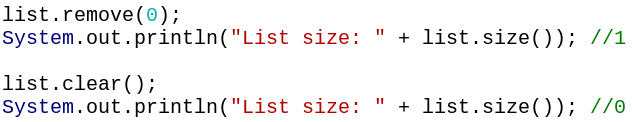


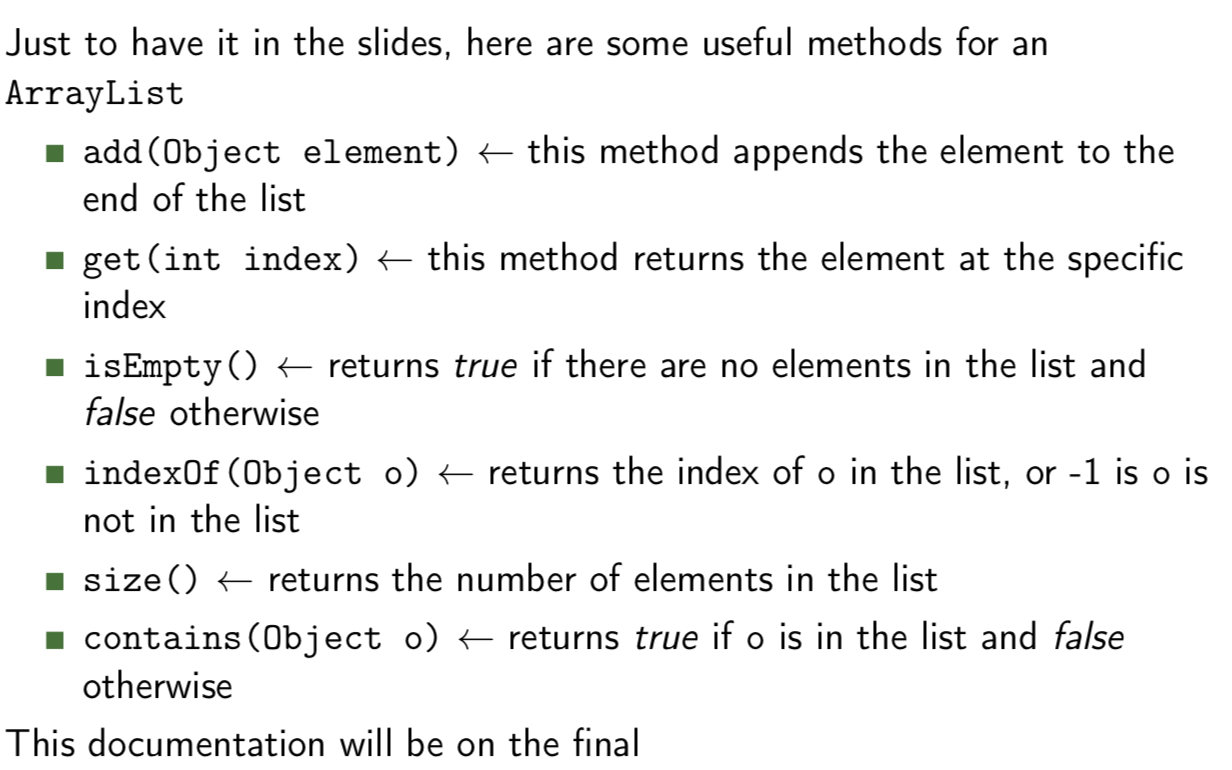
There’s a toString method in ArrayLists

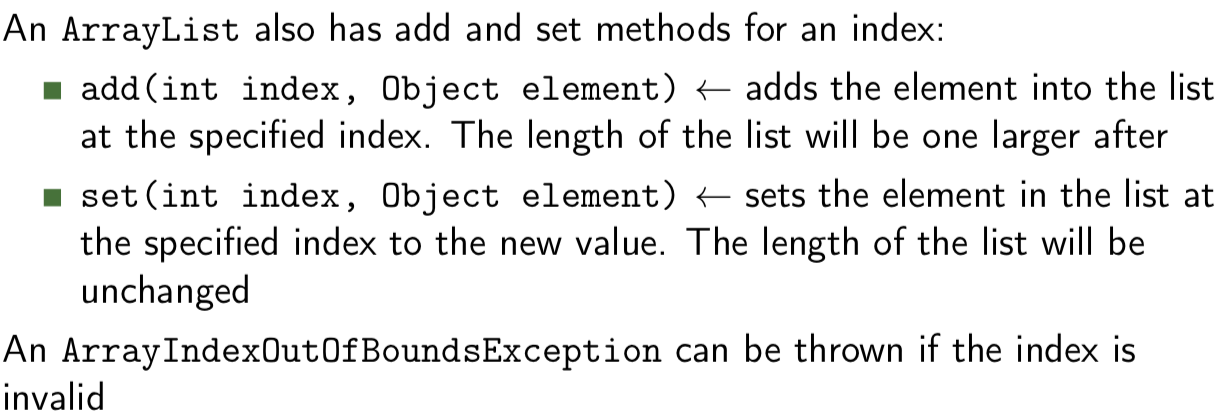












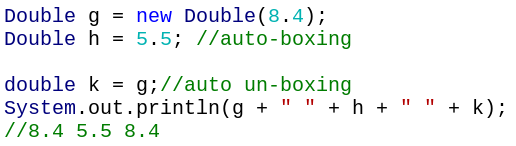
**We can’t store int and double in ArrayList, because they are primitive types, and ArrayLists only**

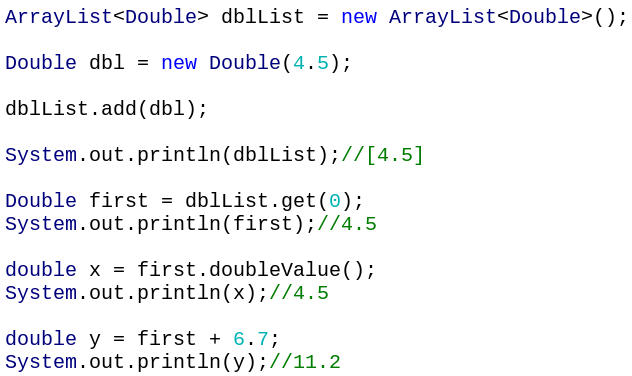
**store reference types/Objects**

* Wrapper class

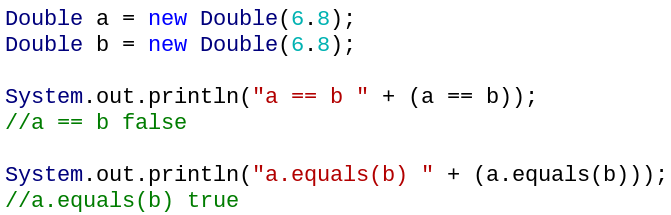
Wraps primitive values in a class

Only use wrapper classes if you have to place values in an ArrayList



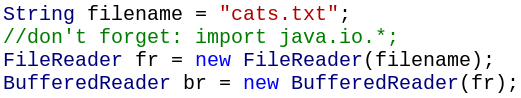


We can’t forget that they are **reference types**. Specifically, aliasing and checking for equality

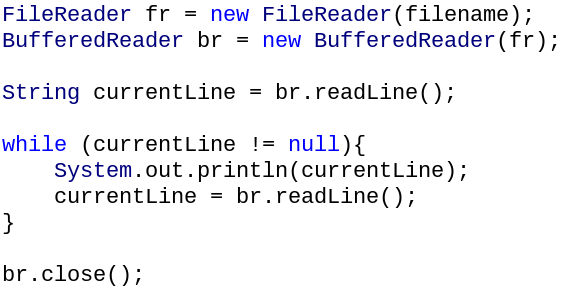


* File

A file is a named collection of 1s and 0s.

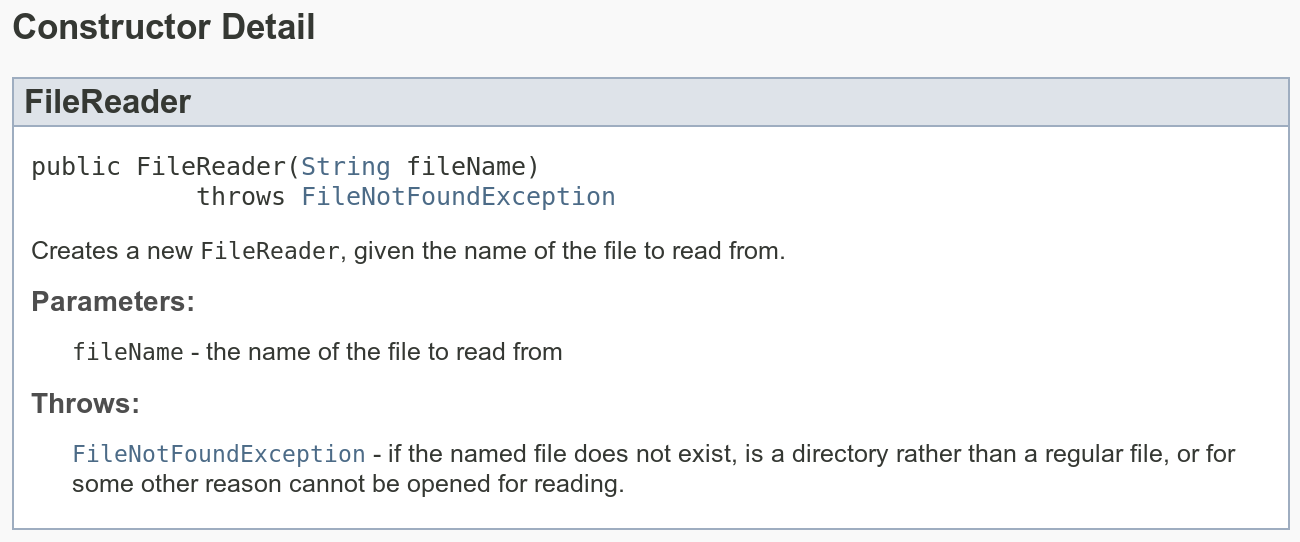


We use the readLine() method on the BufferedReader to get the next line from the file.

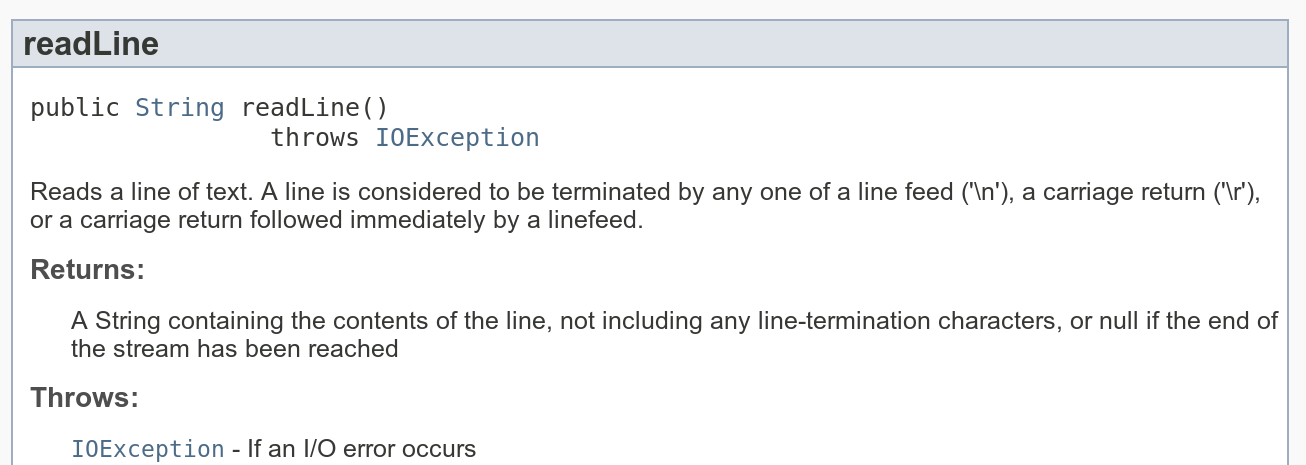


* Java forces us to handle some Exceptions when we read from or write to a file.

*FileNotFoundException* - if the file was missing when reading

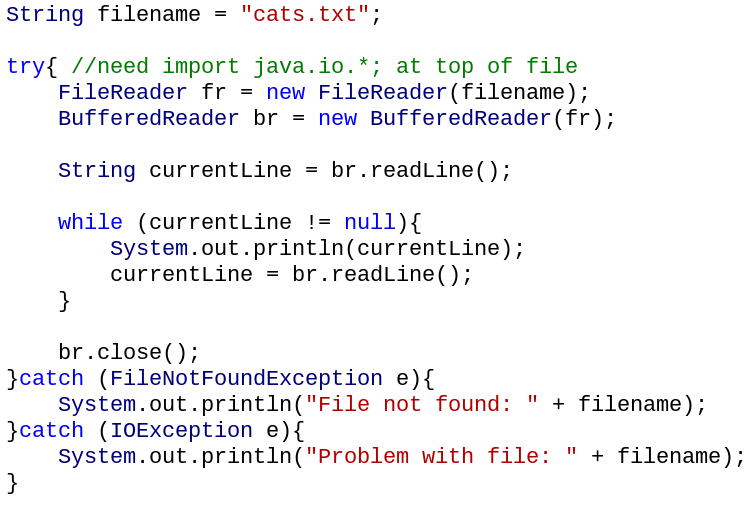
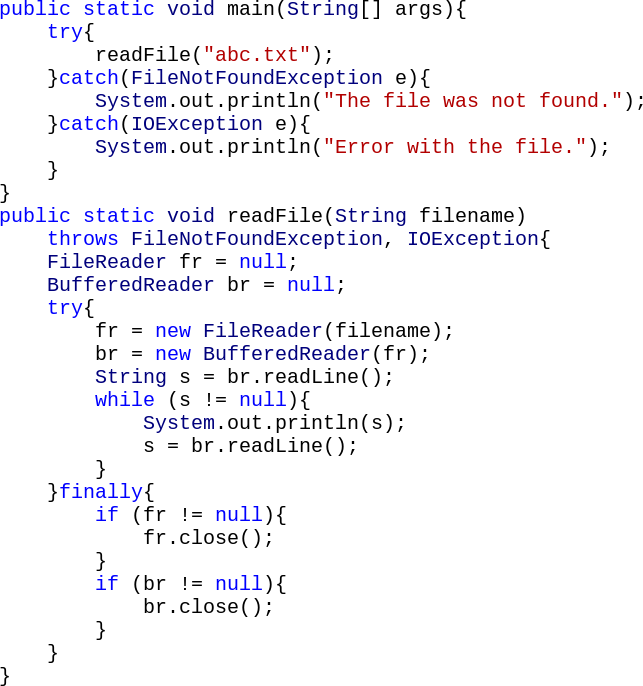


*IOException* - any other problem



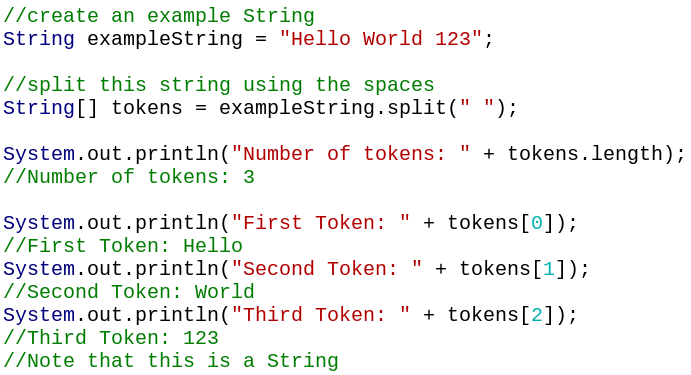
Put all File IO operations inside a try-block and have a catch block for the exceptions

Or, pass on the exceptions by using throws in the header of the method

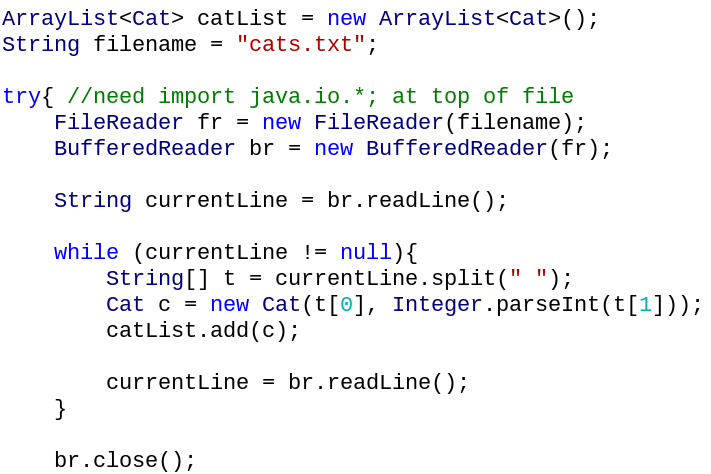
 

* Instance creation
* *spilt method*

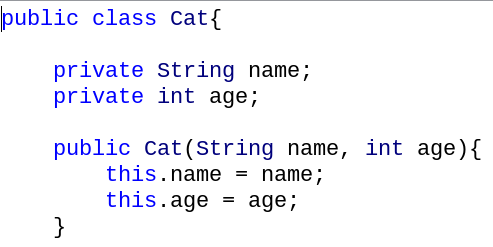
Take a String and a delimiter, and returns a String[] where the String is split on that delimiter.



Create cat

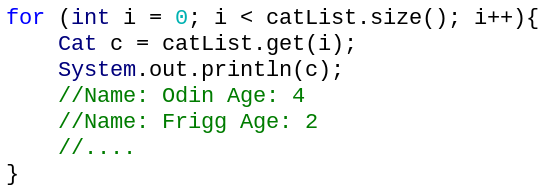


Cat class

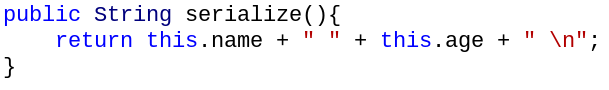


Print out the cats.

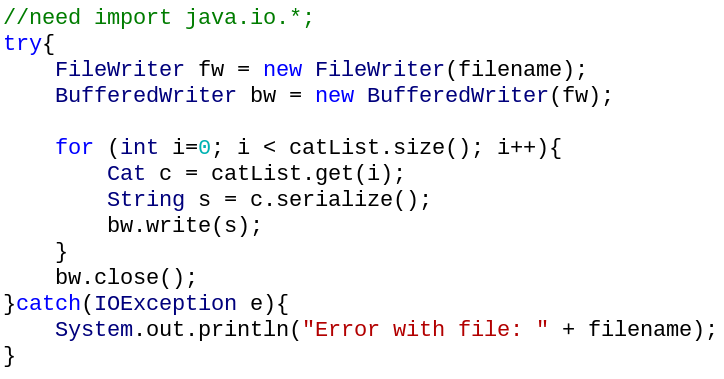
This calls the toString method



* Write to a file



*FileWriter and BufferedWriter*



* Exception
* ***Unchecked*----NullPointerException**

**----ArrayIndexOutOfBoundsException**

**----IllegalArgumentException**

These exceptions are not checked at compile-time and they can cause your code to crash at run-time

You are not forced by the compiler to handle these exceptions It is up to the programmer to decide to catch the exceptions

* ***Checked*----Exception**

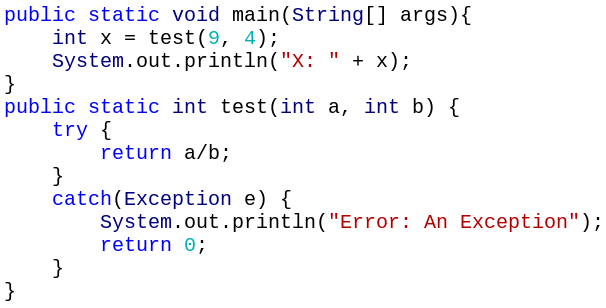
**----IOException**

These exceptions are checked at compile-time!

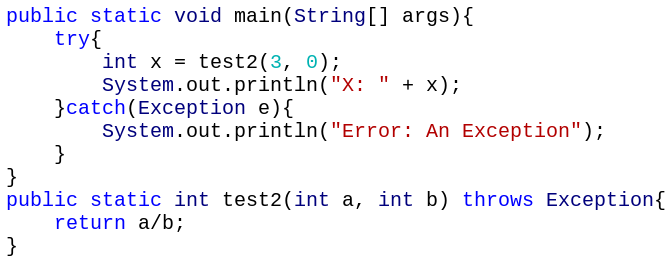
The programmer is forced to handle these exceptions

* Handle exception

i) Surround the code that might throw an exception with a try/catch block.

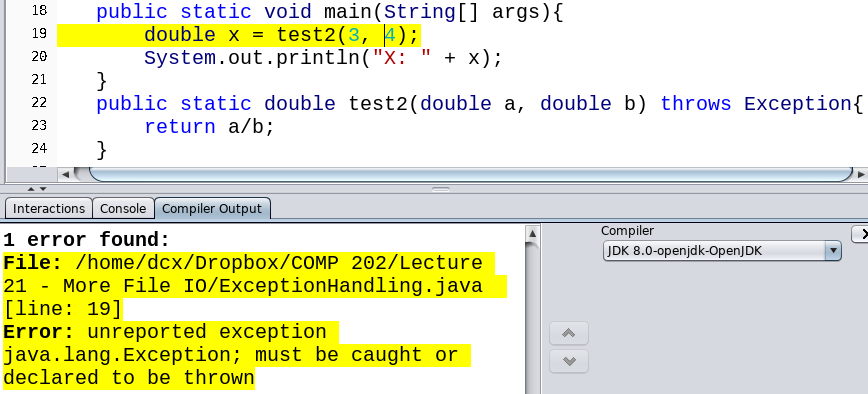


ii) Specify in the method header that there’s an exception using the **throws** keyword followed by the type of the exception. The method call then needs to be caught.



throw arithmeticException :

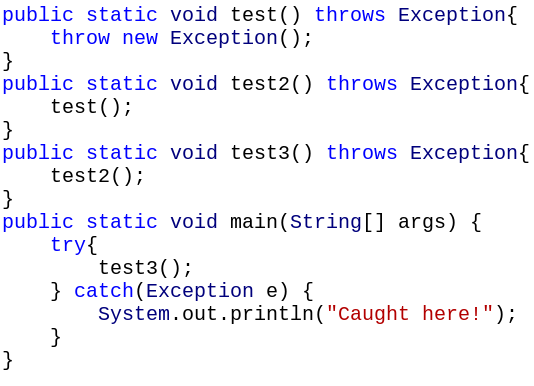
Don’t need to catch the call



* Chaining

We can keep throwing the Exception to the calling method

We force the programmer to decide what to do when there’s an error



In this course, you’re not allowed to throw Exceptions from the main method

* Finally

The *finally* block is attached to a try block, and it always executes.

Even if ----an unexpected exception occurs in the try block

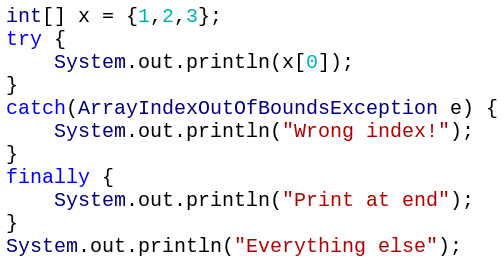
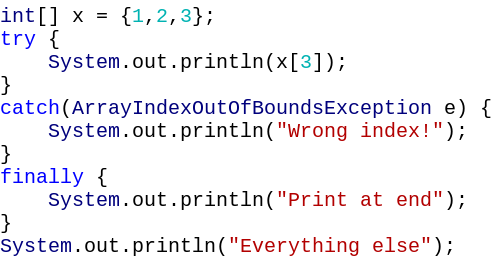
an exception occurs in the catch block

There’s a return/continue/break statement in the try/catch block.

You can have a *finally* even with just a try block (and no catch).

Why use *finally*?

Close files readers/writers and scanners whether or not there was an Exception

Wrong index!

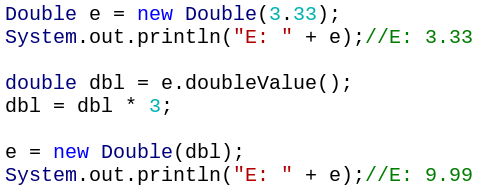
Print at end

Everything else

1

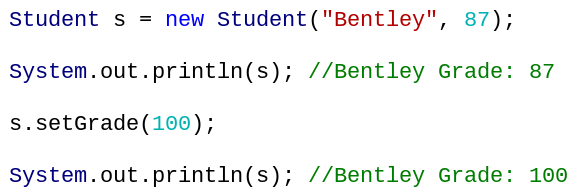
Print at end

Everything else

Immutable: Values can’t change

* Mutable versus Immutable

Mutable: Values in a class can change



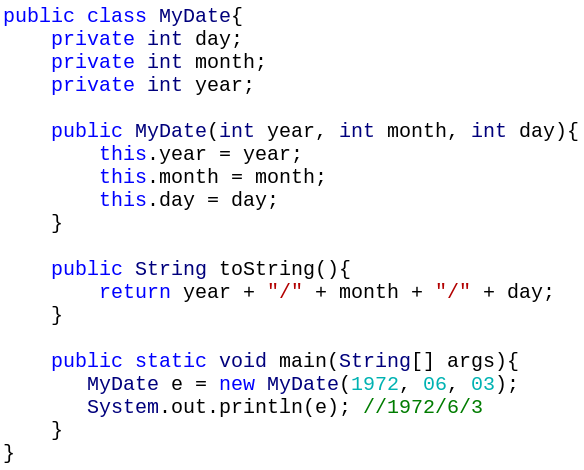
There is no setter to change the value stored.

If you want a Double instance with a different value, you must create a new instance

* Writing an Immutable Class

Make the variables private, and don’t provide setter methods

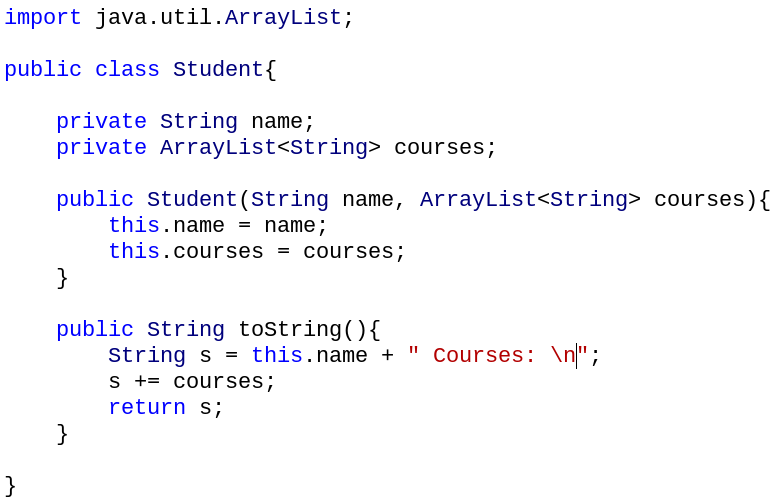
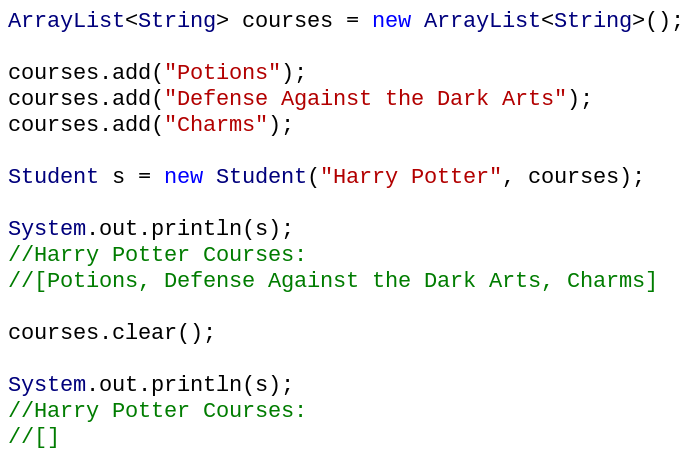
**Only way to set attribute values is through the constructor**



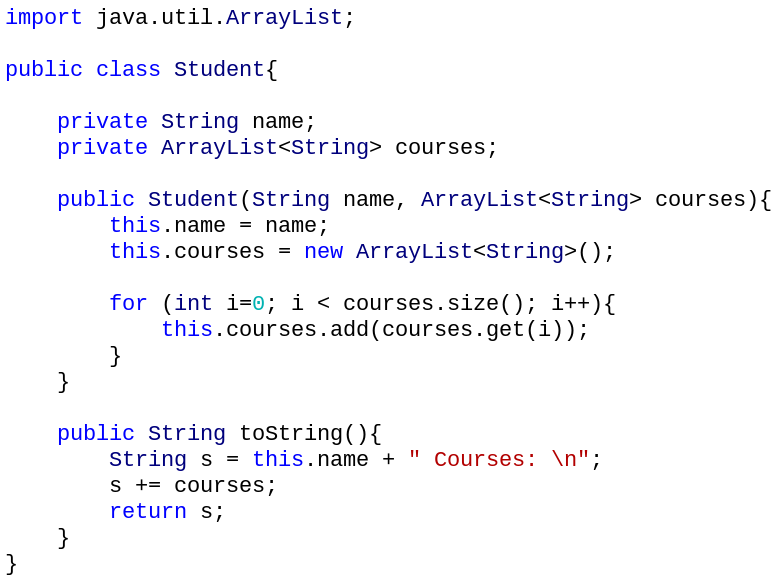
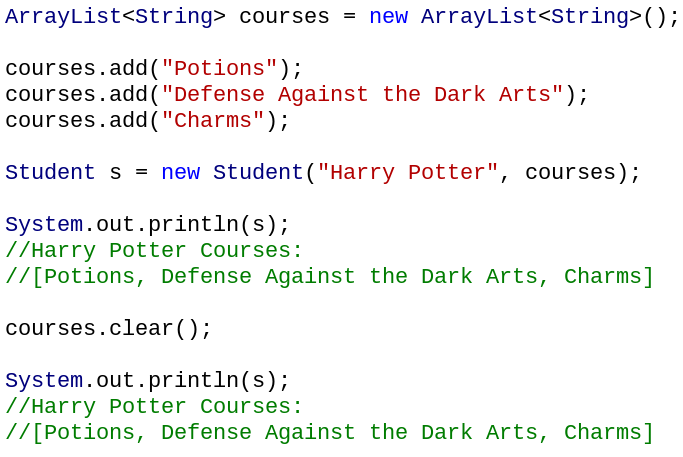
If we have an instance of MyDate, then we know that another part of our program can’t accidentally change the information

Could be useful in a database program, where transactions shouldn’t be deleted or modified

* Copying reference

The other class has access to the list, so it can change the data

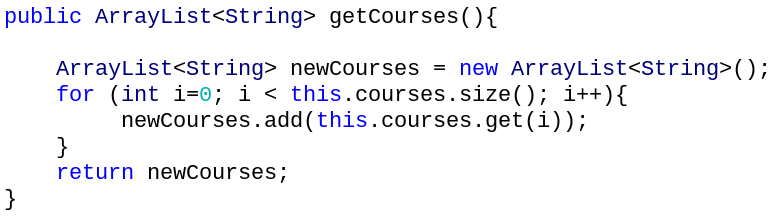
 

Copying References in Getter

Make sure that when a private ArrayList is returned, a copy is made

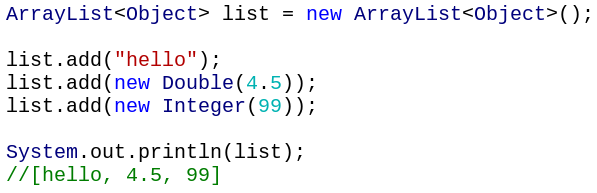
Otherwise, another class can obtain the ArrayList and change the data

This code creates a new ArrayList, and copies the elements into the new list



* Storing Objects

To store multiple object types in an ArrayList, we write ArrayList<Object> list = new ArrayList<Object>(); **Note that we can’t store primitive types, only reference types!**

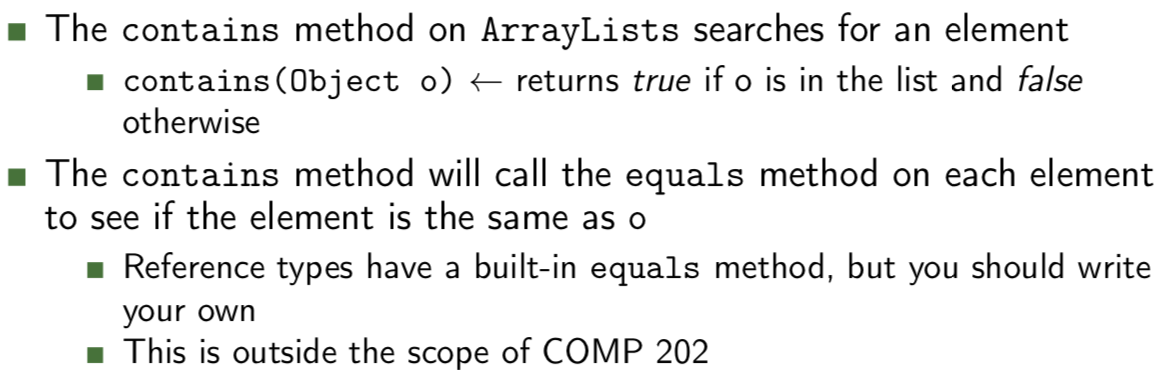


All reference types (like Strings and Students) have some default methods, such as **toString and equals**

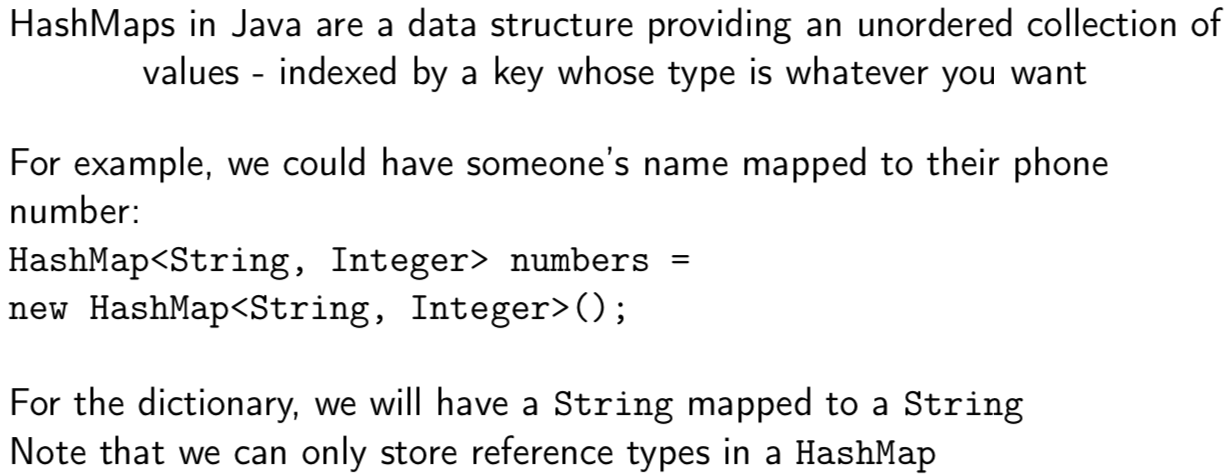
We say that all classes inherit these methods from the parent class

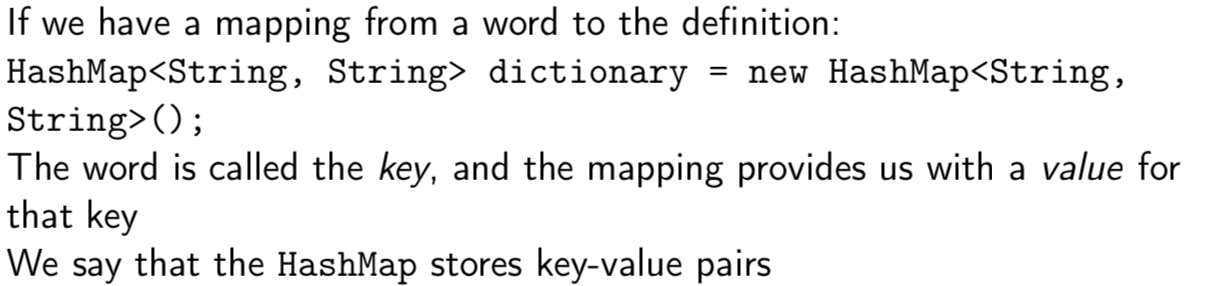
For example, the toString method of an ArrayList can just call toString on all the elements

All reference types have a toString method, so this will always work



* HashMaps





**Note: There is no ordering in a HashMap.**

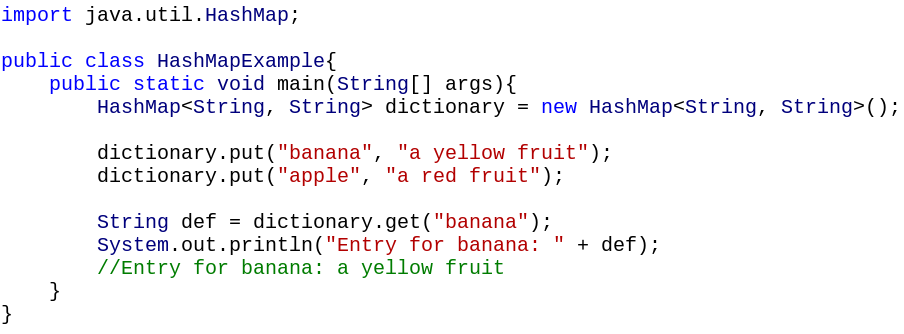
This means that keys might not be sorted alphabetically, and not by the order you inserted them in.

* Basic HashMap methods:

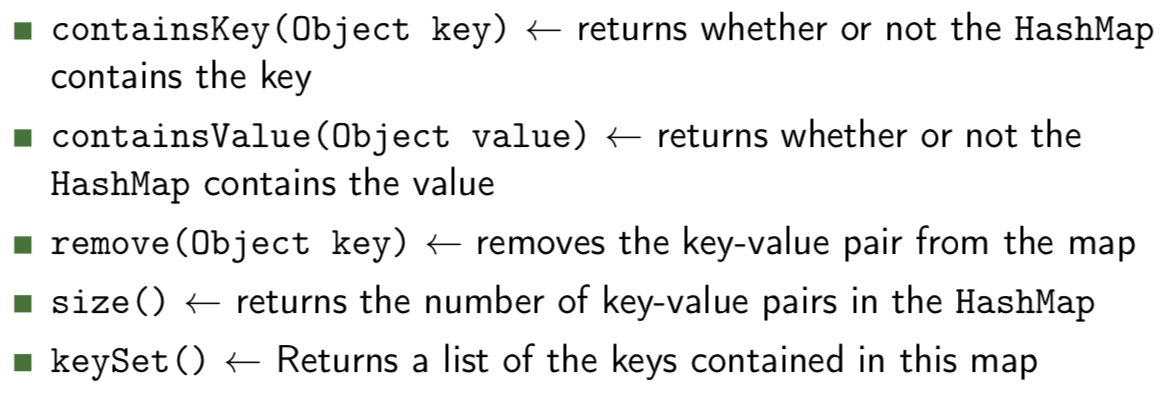
**put(Object key, Object value)** ← adds a value to the HashMap and associates it with that key

**get(Object key)** ← returns the value associated with that key

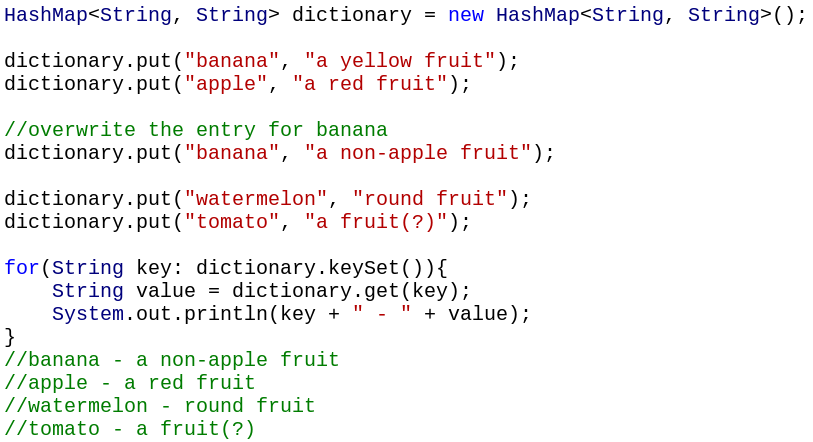
Note: the types must match what the HashMap expects



* Other method



* for-each loop



* Get count

