FUNDAMENTALS

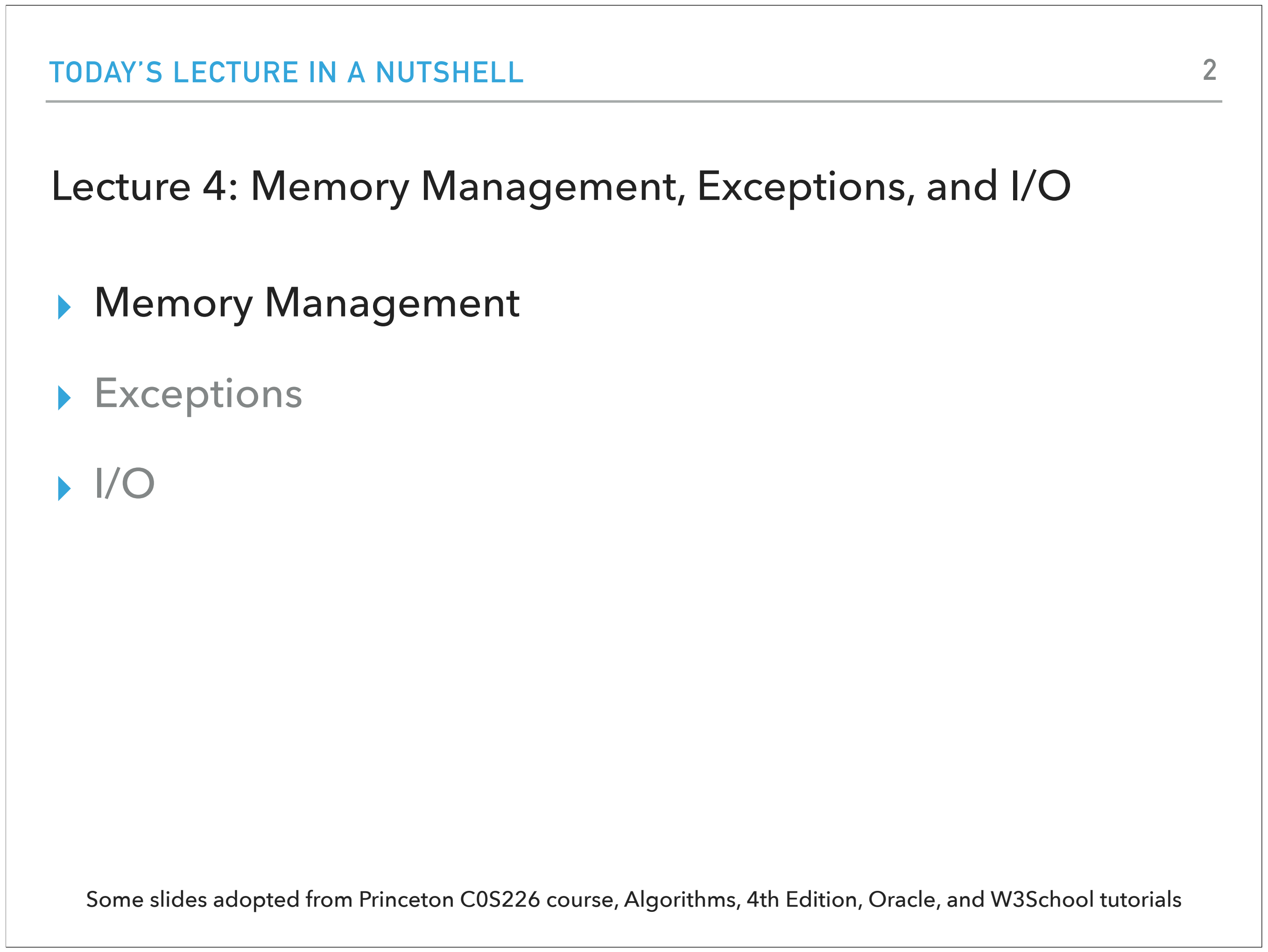
CS62

DATA STRUCTURES AND ADVANCED PROGRAMMING

4: Memory Management, Exceptions, and I/O

Alexandra Papoutsaki
she/her/hers

Today we will cover a number of different topics. Some will be directly useful for this week’s assignment. Others will give you a better understanding of what happens when you run your Java code.

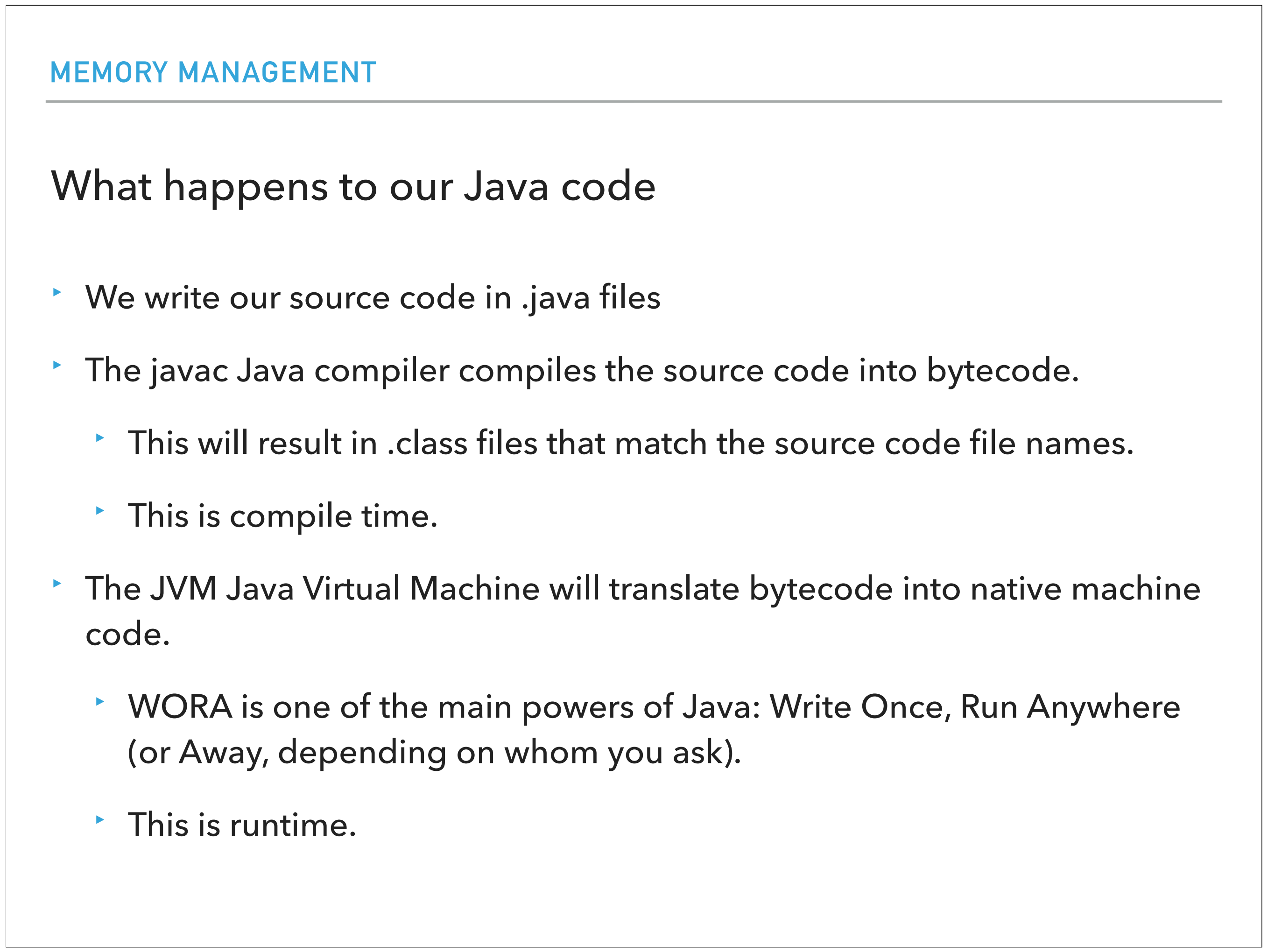


Lecture 4: Memory Management, Exceptions, and I/O

- Memory Management
- Exceptions
- I/O

Some slides adopted from Princeton C0S226 course, Algorithms, 4th Edition, Oracle, and W3School tutorials

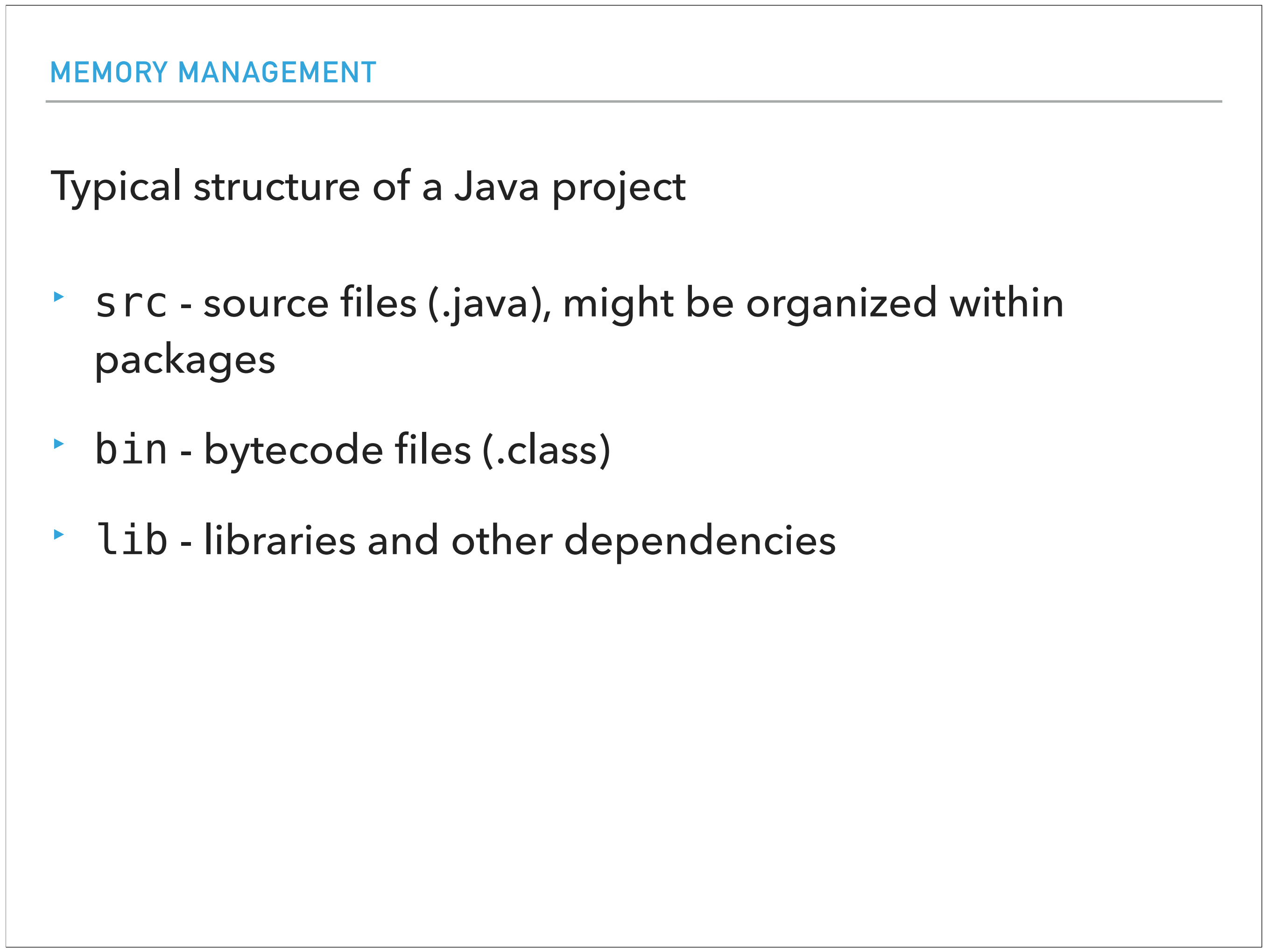
Let's start with talking about memory management in Java. But to do that, I want to establish some vocabulary that will be useful from now on.



What happens to our Java code

- We write our source code in .java files
- The javac Java compiler compiles the source code into bytecode.
  - This will result in .class files that match the source code file names.
  - This is compile time.
- The JVM Java Virtual Machine will translate bytecode into native machine code.
  - WORA is one of the main powers of Java: Write Once, Run Anywhere (or Away, depending on whom you ask).
  - This is runtime.

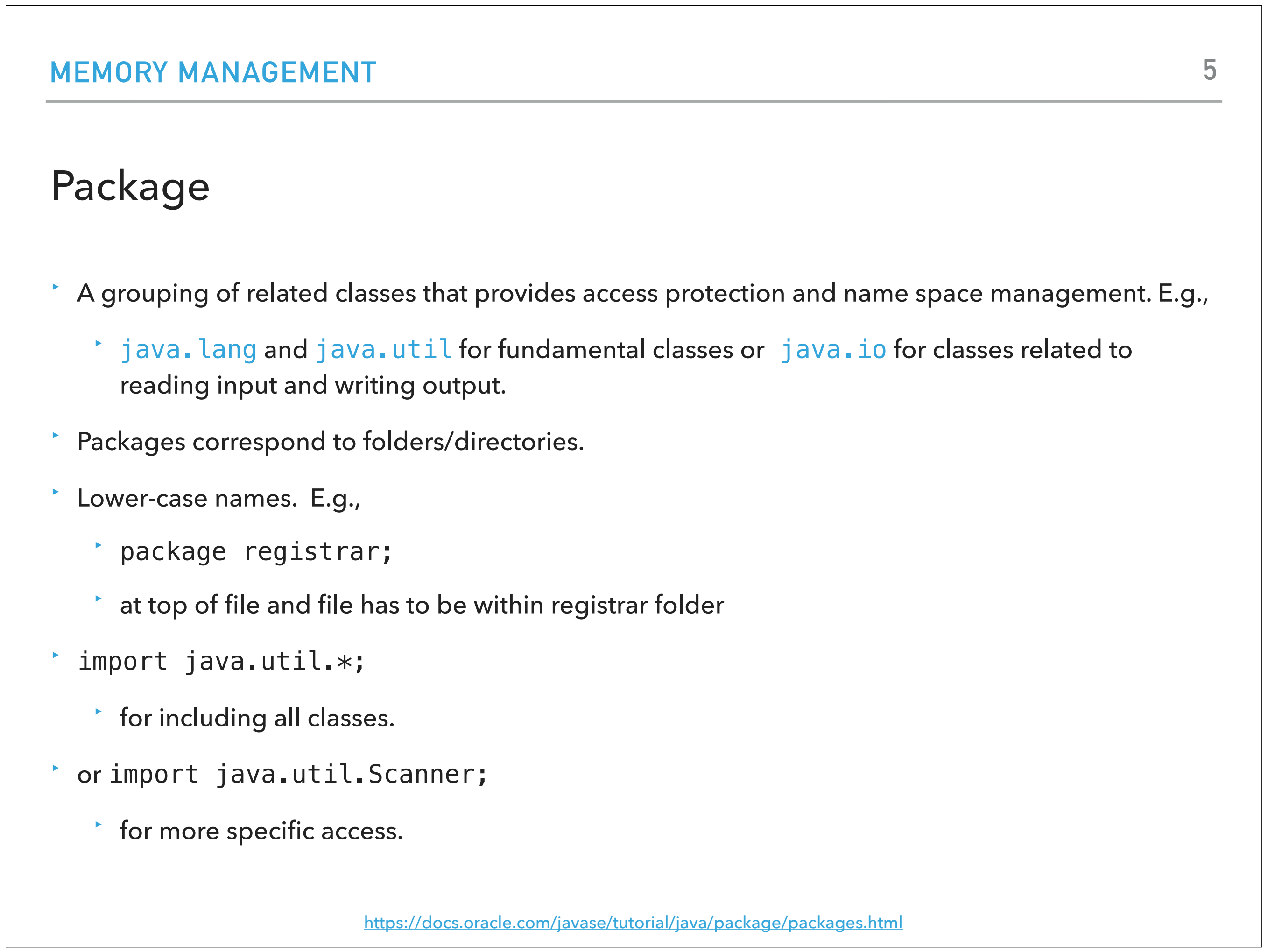
By now, you know that we write our Java programs in .java files. We call these source code. Since we use VS Code which is an IDE, we have been using the “Run Java” button. What that button does entails two steps. The first one, calls the javac, that is the Java compiler which takes our source code and compiles into bytecode. Bytecode is a special instruction set that is stored in .class files. The .class files match the names of our .java files. For example, if we had a PomonaStudent.java file, we would see that a PomonaStudent.class file would be generated after hitting the Run Java button. This first step is known as compile time. Once we have the bytecode, the second step is for the JVM, that is the Java Virtual Machine, to translate the bytecode into native machine code. This is a key feature of Java which allows it to run on any machine. This is known as WORA and it really changed how software was distributed since now we only have to have the right kind of JVM for our type of computer and it will take care of translating it into the appropriate machine code. This step is known as run time. Instead of using VS Code, you could have completed both steps through the terminal. In fact, you will see that whenever you hit the Run Java button, VS Code shows you the commands that have been executed (i.e. javac something.java and then java something which correspond to the compiler and JVM respectively).



Typical structure of a Java project

- `src` - source files (.java), might be organized within packages
- `bin` - bytecode files (.class)
- `lib` - libraries and other dependencies

If you open the directory system of your computer and navigate to the folder you have cloned the labs and assignment code we have given you, you will notice that your Java projects follow a consistent pattern. There is one folder called src where your source files (remember, that is the .java files we write) will reside. Within the src folder, there might be another level of directories which will correspond to packages (more soon), and where eventually we will find our source code.
The second type of folder, bin, will be created after you have run your code at least one. It will create the bytecode files (i.e. the .class files).
And finally, you might notice a lib folder that contains libraries and other dependencies we might need to run our code. Most of the times, VS Code takes care of this folder for us but we might occasionally need to manually update it.

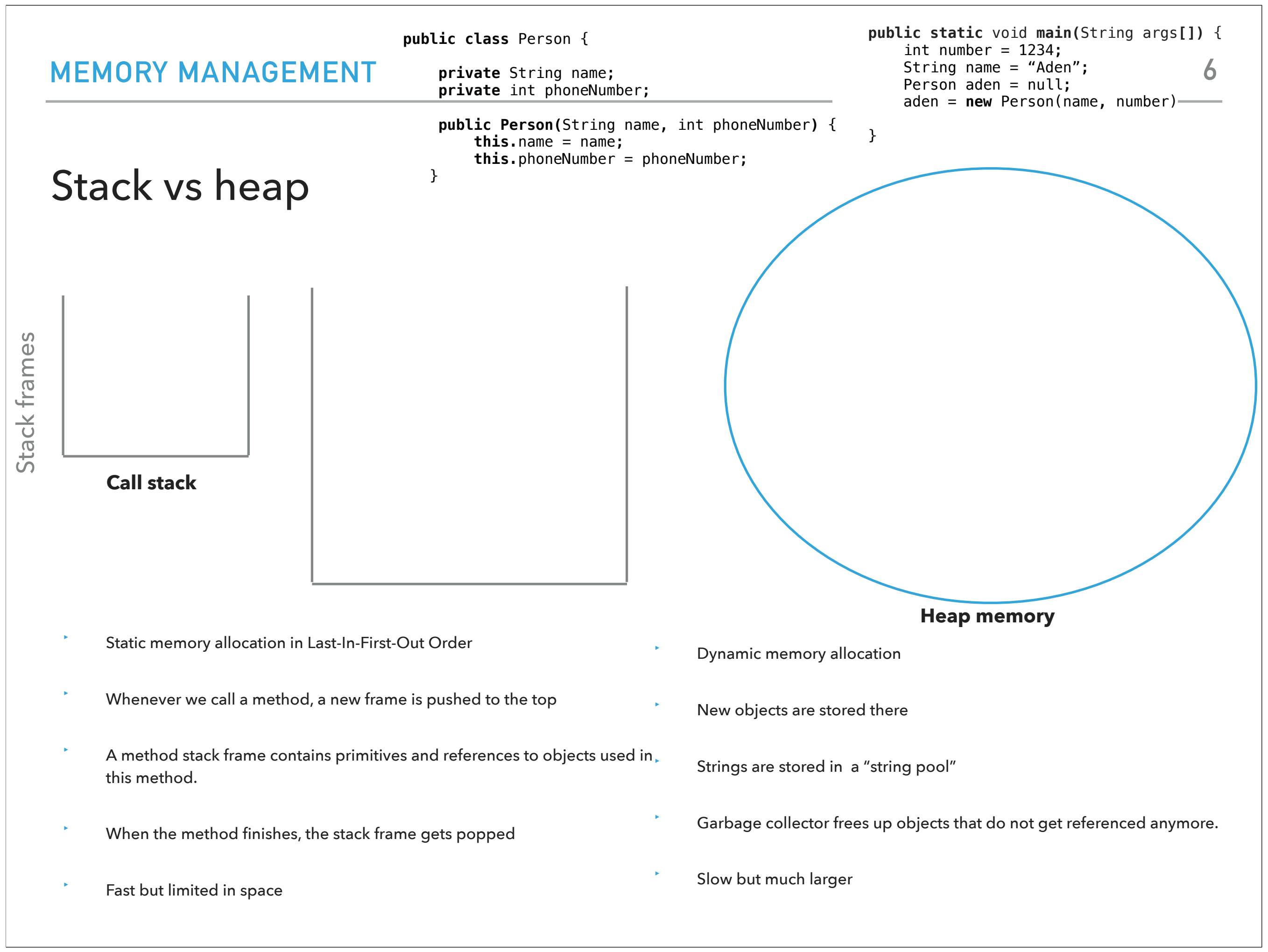


Package

- A grouping of related classes that provides access protection and name space management. E.g.,
  - `java.lang` and `java.util` for fundamental classes or `java.io` for classes related to reading input and writing output.
- Packages correspond to folders/directories.
- Lower-case names. E.g.,
  - `package registrar;`
  - at top of file and file has to be within registrar folder
- `import java.util.*;`
  - for including all classes.
- or `import java.util.Scanner;`
  - for more specific access.

https://docs.oracle.com/javase/tutorial/java/package/packages.html

I mentioned that your source code will reside in src but within it there might be packages. A package is a grouping of related classes (and later we will see interfaces as well) that provides access protection and name space management. For example, the Java developers created the java.lang and java.util packages for fundamental classes and the java.io package for classes related to reading input and writing output, as we will see today.
You can think of packages as corresponding to folders/directories. The convention is that their names are lower-cased. For example, if your package is called registrar, you would include registrar line package registrar; at the top of your java file AND your java file has to be within the whateverName folder in src.
If you want to have access to all classes in a package, you would import the package.*; or you can import a specific class which is better in terms of avoiding naming conflicts and having higher memory efficiency.
If you check the bag of tokens lab, you will see that the source code we gave you belongs to the package bag and that we imported the Random class to be able to randomly select a color and number for our tokens.



MEMORY MANAGEMENT

Stack vs heap

```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }
```

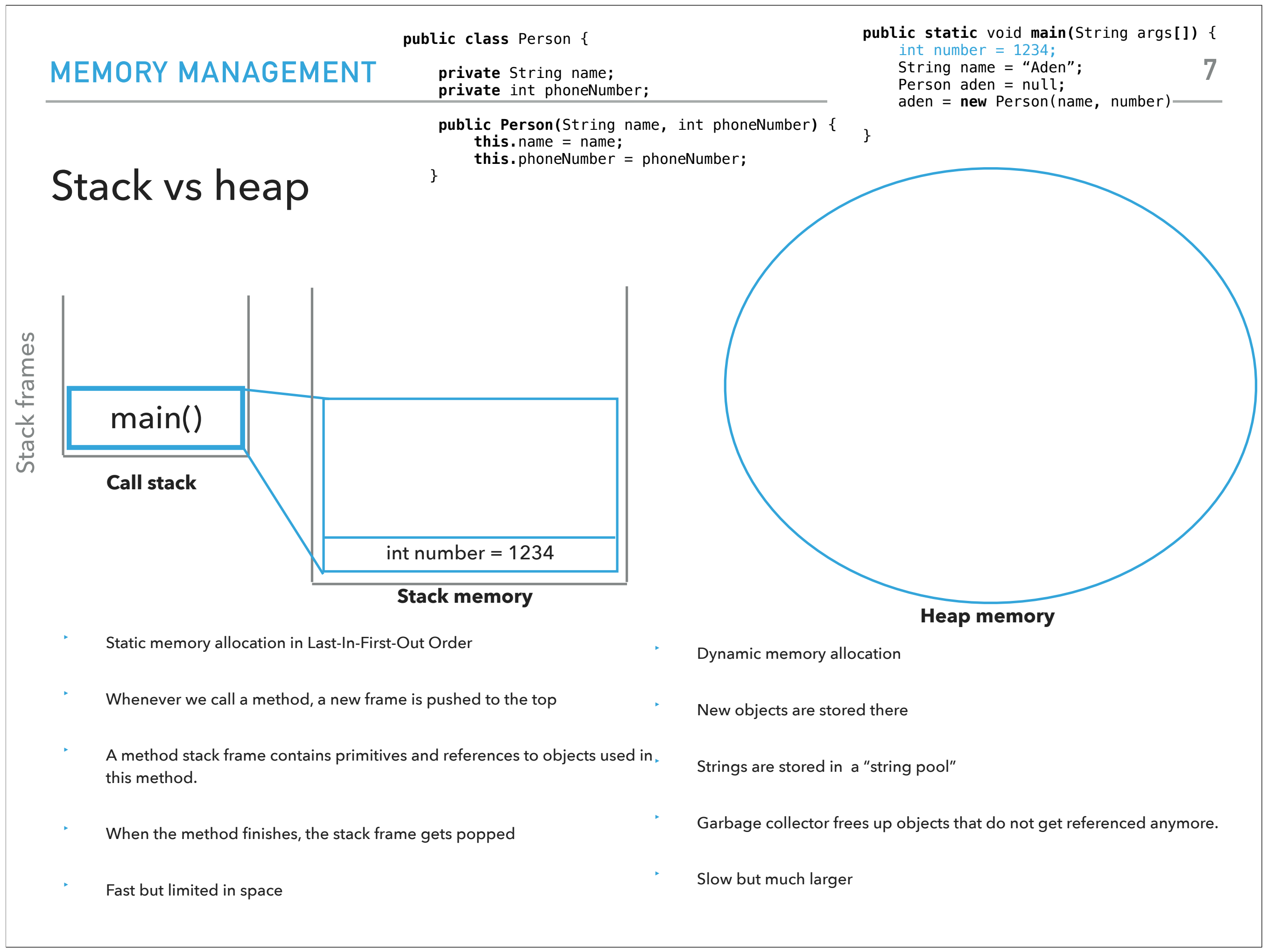
```
public static void main(String args[]) {
    int number = 1234;
    String name = "Aden";
    Person aden = null;
    aden = new Person(name, number)
}
```

Stack frames
Call stack
Heap memory

- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

- Dynamic memory allocation
- New objects are stored there
- Strings are stored in a "string pool"
- Garbage collector frees up objects that do not get referenced anymore.
- Slow but much larger

Now let's see what happens when it comes to how memory is managed. The are two types of memory allocation mechanisms: static and dynamic. Static memory allocation uses a data structure which we will encounter later in the semester and which is called a stack. Every time we call a method, a new frame is pushed to the top of the stack. A method stack frame contains primitives and references to methods used in this method. A stack follows the last-in-first-order, so once the code of a method finishes, the stack frame gets popped. The call stack is fast but is also limited in space.
On the other hand, we have dynamic memory allocation which uses a heap, another data structure we will encounter later on. Whenever we use the new keyword to instantiate a new object, the actual object (not the reference), will be stored there. Strings are a special type of object and Java has a string pool to reuse space. Once an object is no longer referenced by anyone, it is eligible for being cleaned by the garbage collector. The heap is slow but it is much larger than the stack.



```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }
```

```
public static void main(String args[]) {
    int number = 1234;
    String name = "Aden";
    Person aden = null;
    aden = new Person(name, number)
}
```

Stack vs heap

- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

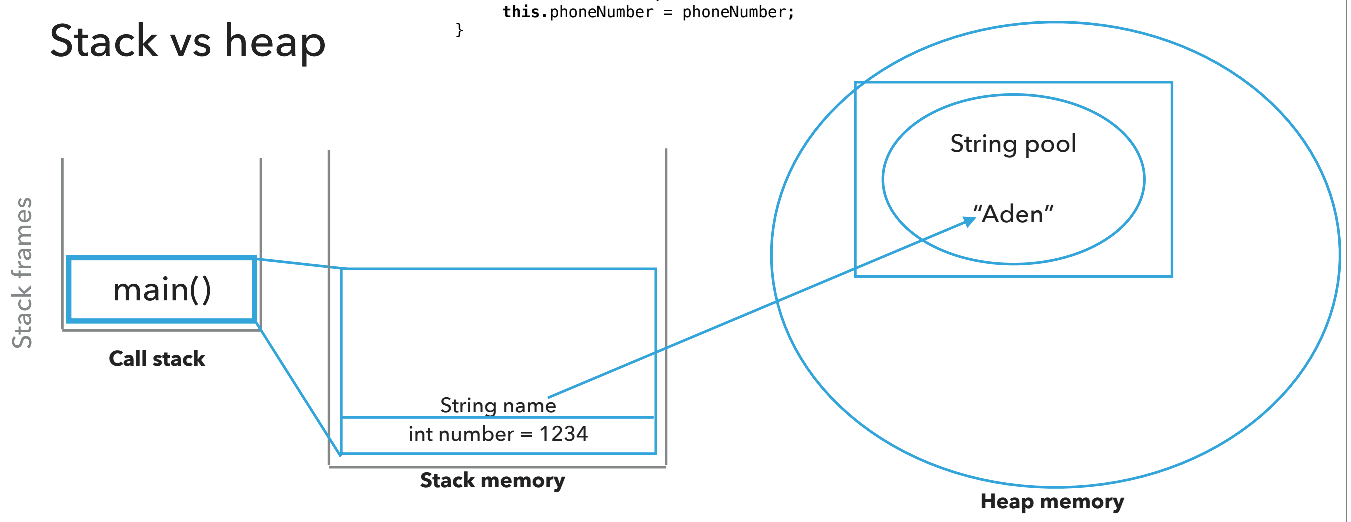
- Dynamic memory allocation
- New objects are stored there
- Strings are stored in a "string pool"
- Garbage collector frees up objects that do not get referenced anymore.
- Slow but much larger

Let's see how that plays out in the following scenario. We have a class Person with two instance variables, name and phone number, a constructor, and a main method. Once we run our code, the main method gets pushed to the call stack. Its first line, initializes a local variable number. Since number is a primitive it will be stored within the stack frame.

```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }
```

```
public static void main(String args[]) {
    int number = 1234;
    String name = "Aden";
    Person aden = null;
    aden = new Person(name, number)
}
```

Stack vs heap



- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

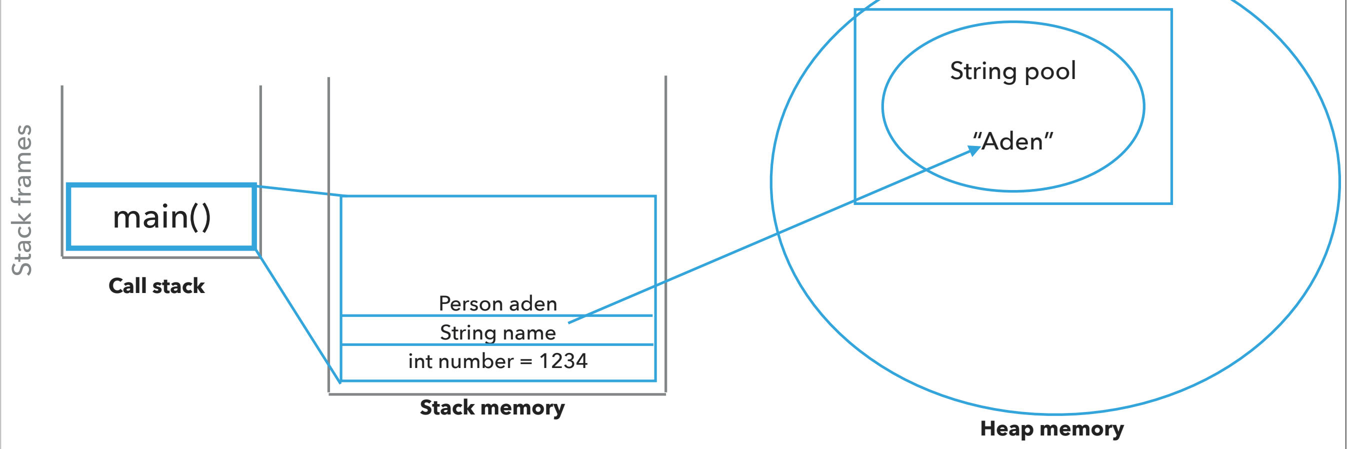
- Dynamic memory allocation
- New objects are stored there
- Strings are stored in a "string pool"
- Garbage collector frees up objects that do not get referenced anymore.
- Slow but much larger

The second line, initializes a String. Remember a String is an object. The reference is called name and it resides in the stack frame of main. The actual object will be created in the heap but since Strings are special objects, it will be created within the String pool.

```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }
```

```
public static void main(String args[]) {
    int number = 1234;
    String name = "Aden";
   Person aden = null;
    aden = new Person(name, number)
}
```

Stack vs heap



- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

- Dynamic memory allocation
- New objects are stored there
- Strings are stored in a "string pool"
- Garbage collector frees up objects that do not get referenced anymore.
- Slow but much larger

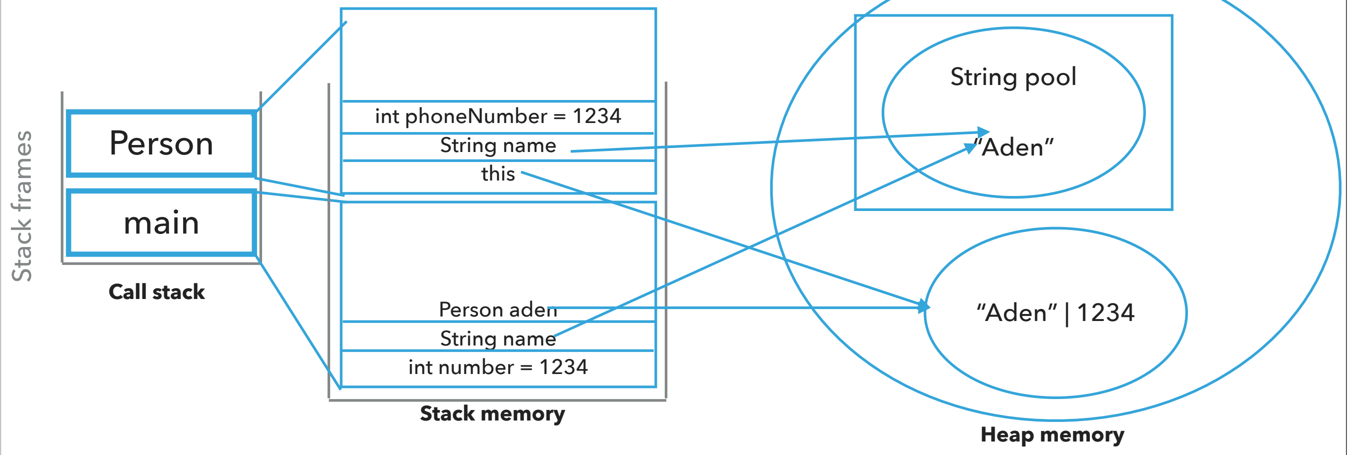
The third line creates a reference of type Person and sets it to null. Thus the reference is added to the stack frame of the main method.

MEMORY MANAGEMENT

```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }
```

```
public static void main(String args[]) {
    int number = 1234;
    String name = “Aden”;
    Person aden = null;
    aden = new Person(name, number)
}
```

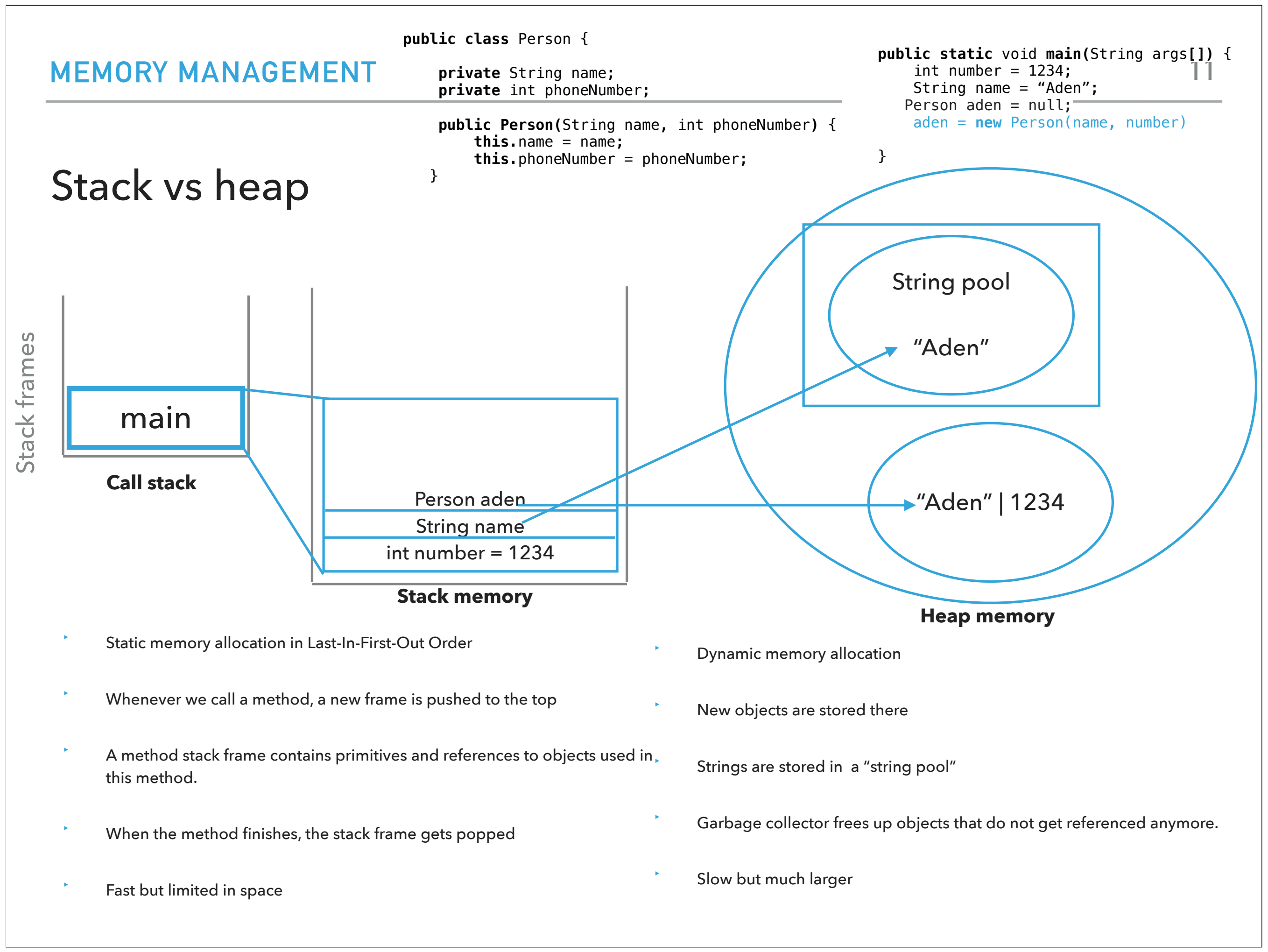
Stack vs heap



- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

- Dynamic memory allocation
- New objects are stored there
- Strings are stored in a “string pool”
- Garbage collector frees up objects that do not get referenced anymore.
- Slow but much larger

The final line of the main method, calls the constructor of Person. This will result to a new stack frame being added in the call stack. The stack frame will contain the local variable phoneNumber (which corresponds to the parameter) and a reference to the String Aden which is already in the String pool. Finally, a reference called this will point to the heap to the actual object of type Person that has all the information that a person comprises of. Eventually, aden, the reference in main, will also point to that object int the heap.



MEMORY MANAGEMENT

Stack vs heap

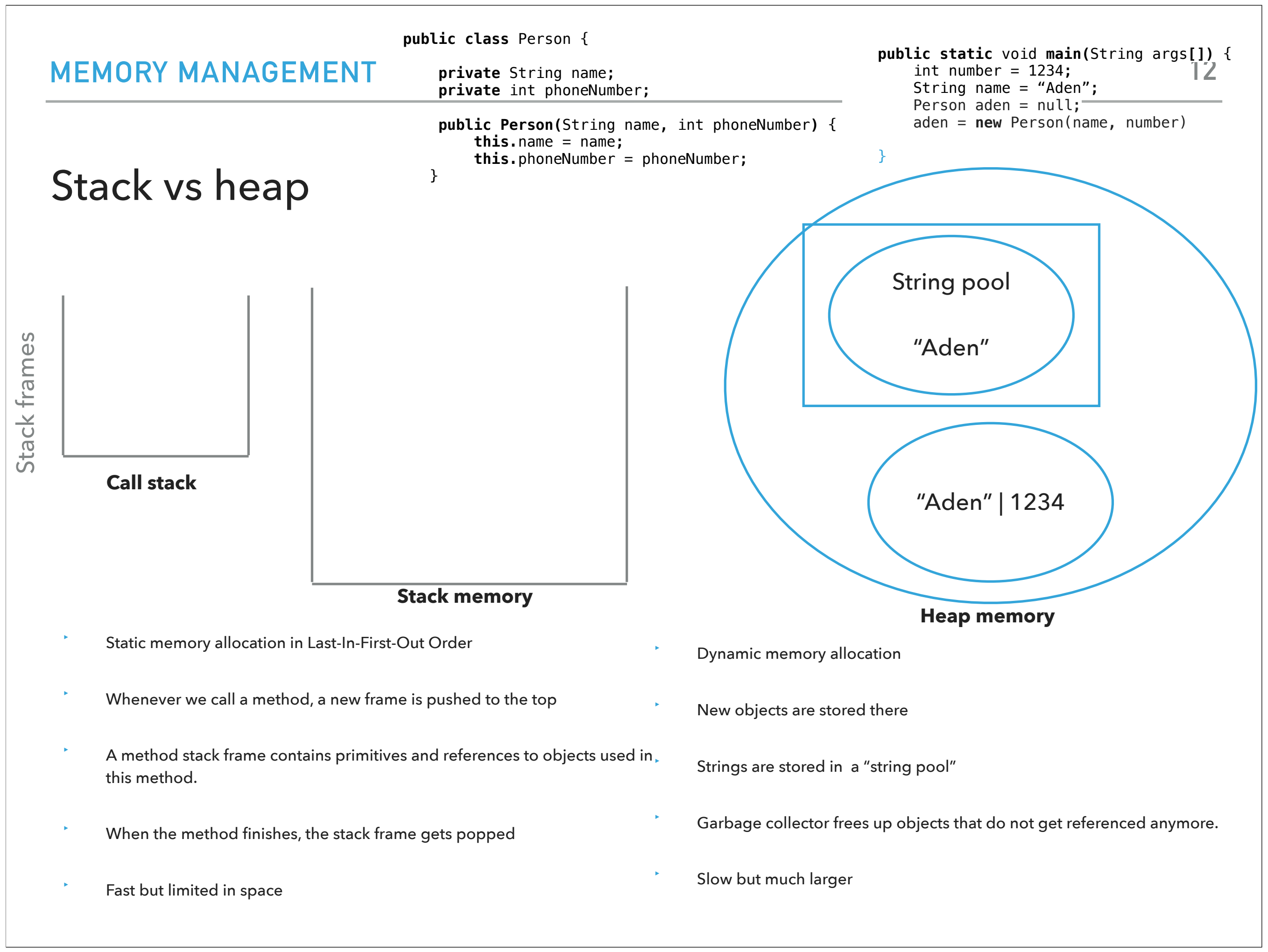
```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }
```

```
public static void main(String args[]) {
    int number = 1234;
    String name = "Aden";
   Person aden = null;
    aden = new Person(name, number)
}
```

- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

- Dynamic memory allocation
- New objects are stored there
- Strings are stored in a "string pool"
- Garbage collector frees up objects that do not get referenced anymore.
- Slow but much larger

Once the execution of the constructor completes, its stack frame will be popped from the call stack and all of its contents, including any references will be wiped. That leaves us with the stack frame of main.



```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }
```

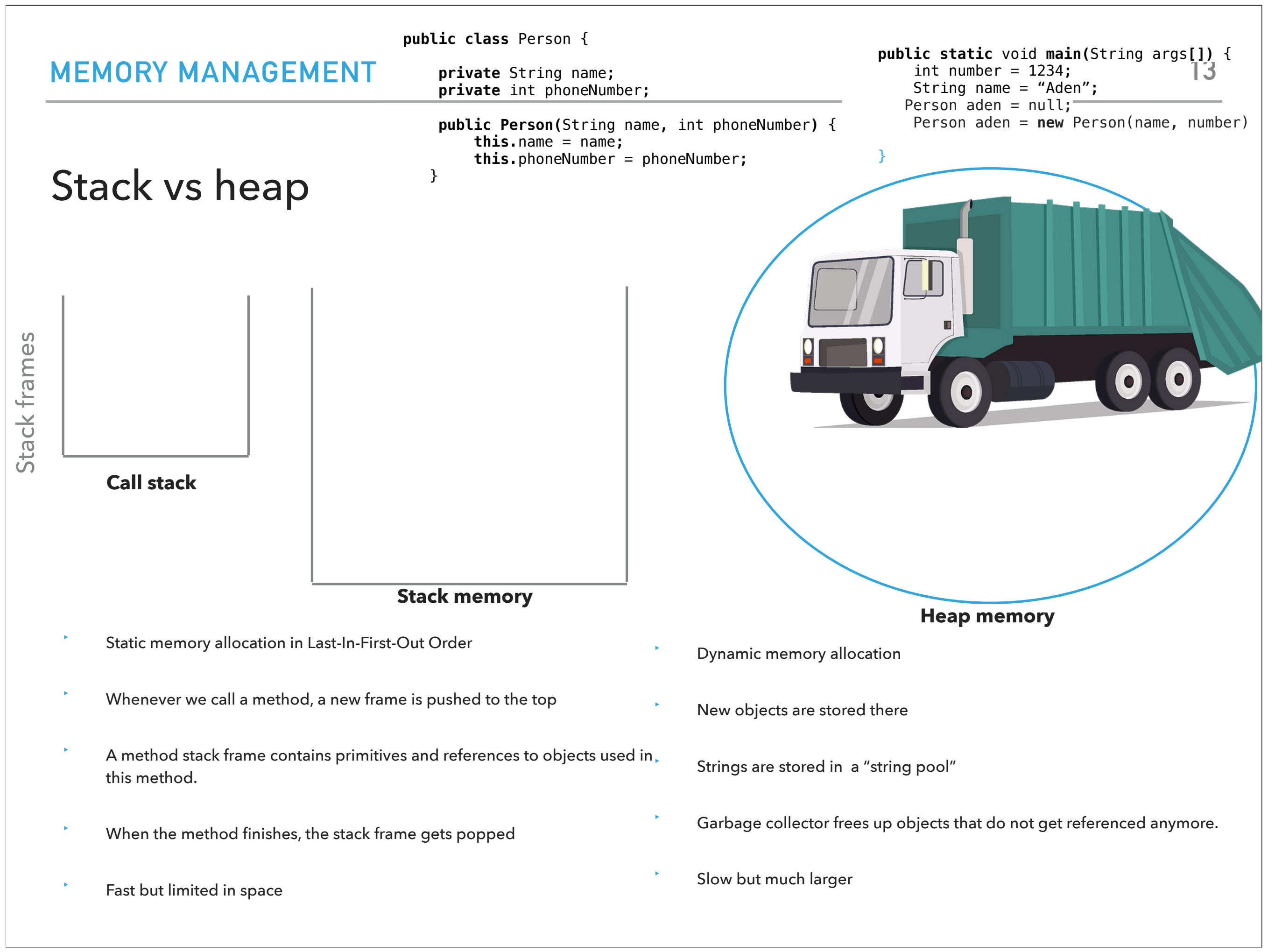
```
public static void main(String args[]) {
    int number = 1234;
    String name = “Aden”;
    Person aden = null;
    aden = new Person(name, number)
}
```

Stack vs heap

- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

- Dynamic memory allocation
- New objects are stored there
- Strings are stored in a “string pool”
- Garbage collector frees up objects that do not get referenced anymore.
- Slow but much larger

Once the entire main method gets executed, its stack frame will be also popped. That will remove the name and aden references to the heap string and Person objects.



MEMORY MANAGEMENT

```
public class Person {

    private String name;
    private int phoneNumber;

    public Person(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }
```

```
public static void main(String args[]) {
    int number = 1234;
    String name = "Aden";
   Person aden = null;
    Person aden = new Person(name, number)
}
```

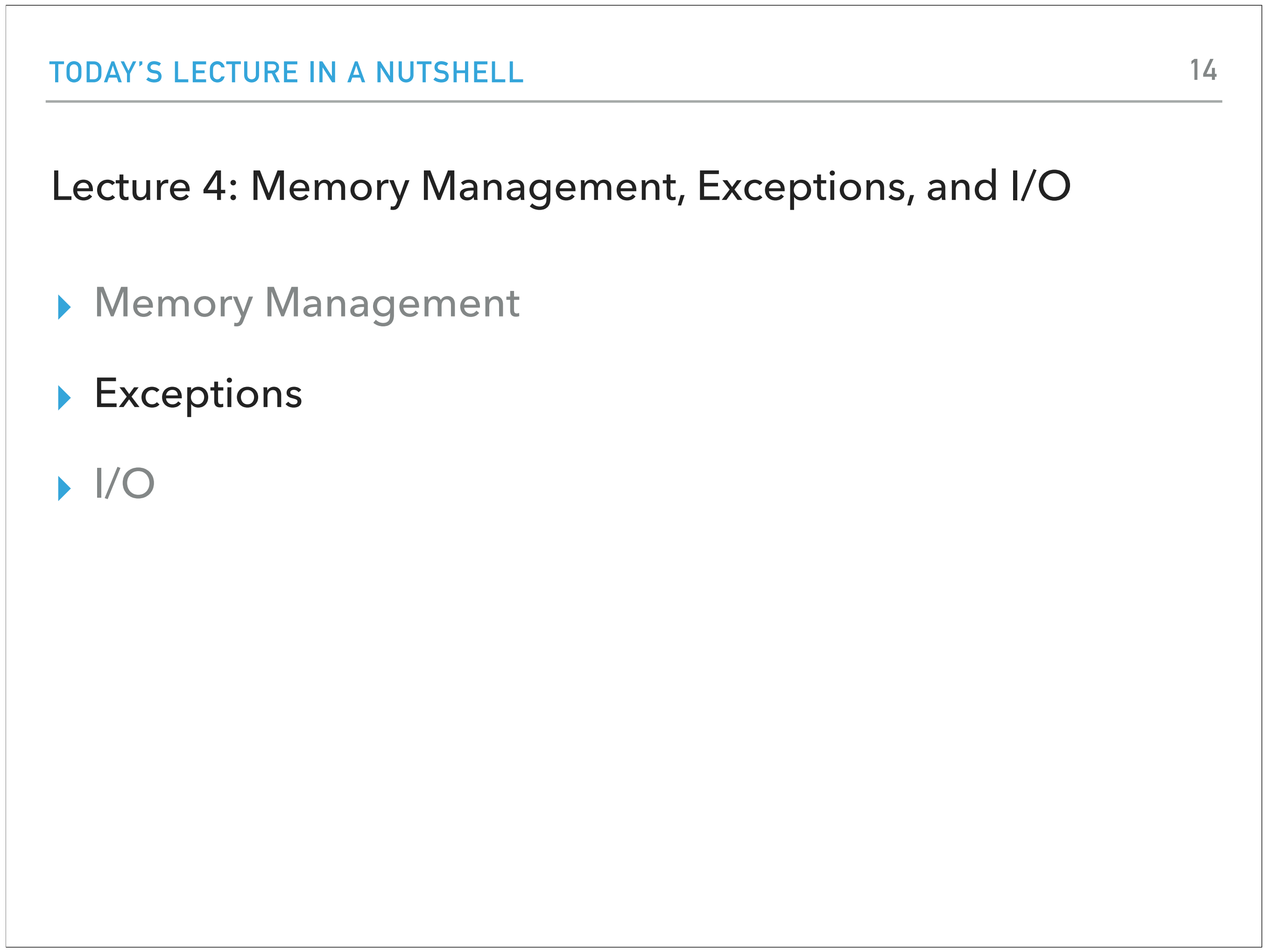
Stack vs heap

Stack frames
Call stack
Stack memory
Heap memory

- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

- Dynamic memory allocation
- New objects are stored there
- Strings are stored in a "string pool"
- Garbage collector frees up objects that do not get referenced anymore.
- Slow but much larger

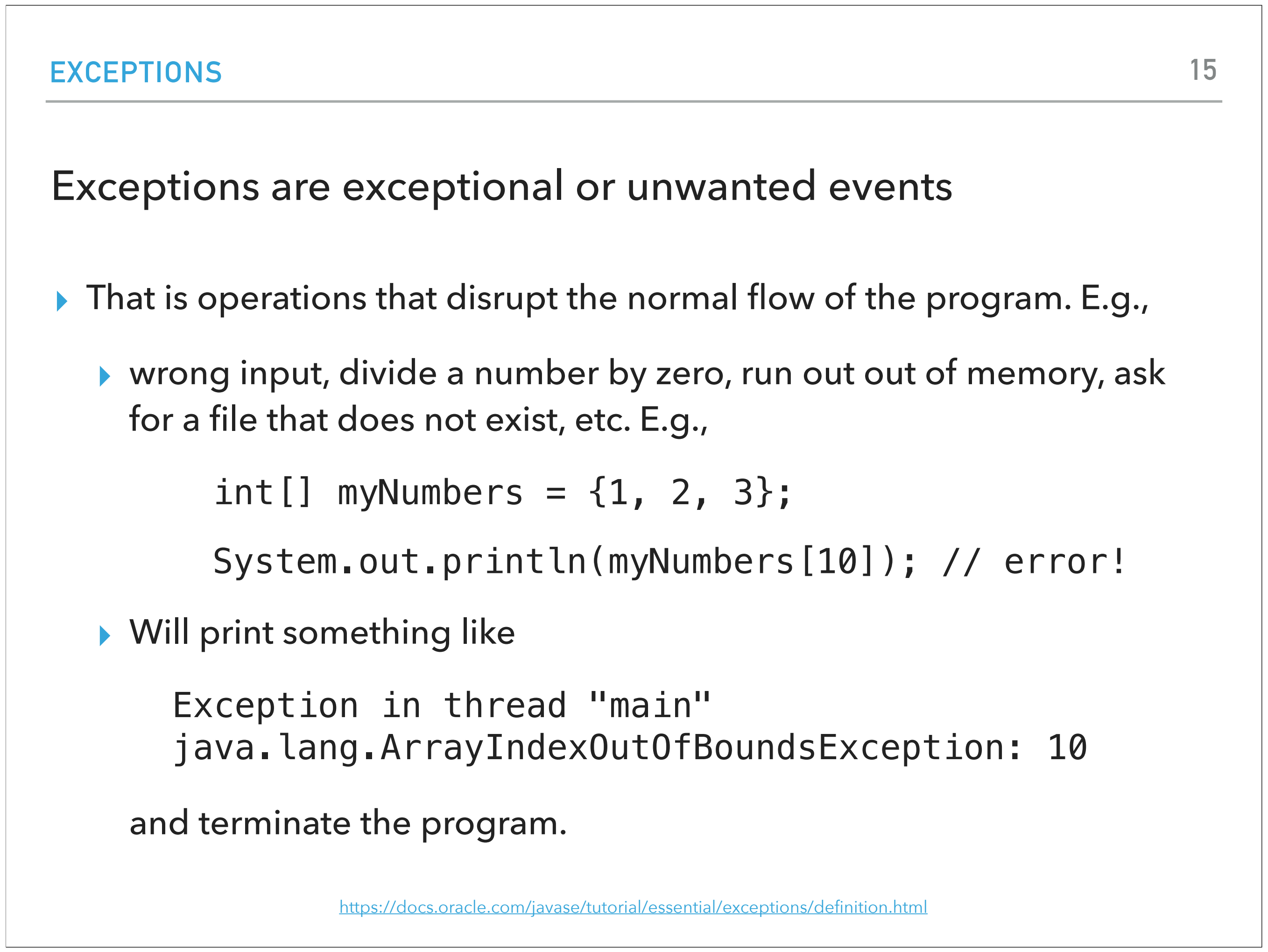
Which means that the garbage collector can come to wipe them out from the heap. In contrast to languages like C++, in Java we cannot really force the destruction of objects which annoys some programmers but is safer for beginner programmers.



Lecture 4: Memory Management, Exceptions, and I/O

- Memory Management
- Exceptions
- I/O

Hopefully that helped with getting a better sense of how Java works. Let's now see what happens when things go wrong with our code by talking about exceptions.



Exceptions are exceptional or unwanted events

- That is operations that disrupt the normal flow of the program. E.g.,
  - wrong input, divide a number by zero, run out out of memory, ask for a file that does not exist, etc. E.g.,

```
int[] myNumbers = {1, 2, 3};

System.out.println(myNumbers[10]); // error!
```

  - Will print something like

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 10
```

  and terminate the program.

https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

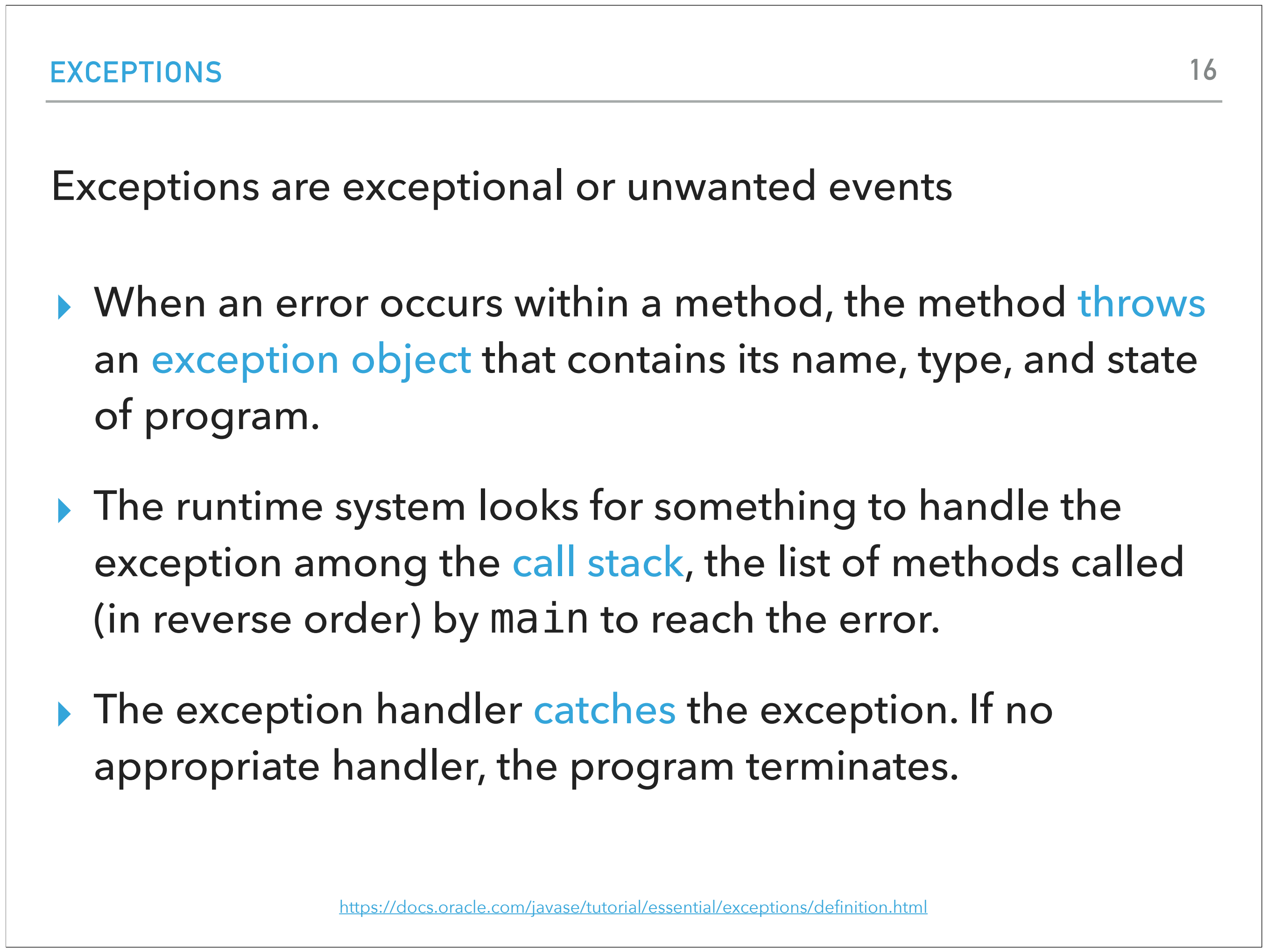
An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things. E.g., the code

```
   int[] myNumbers = {1, 2, 3};
   System.out.println(myNumbers[10]); // error!
```

Will print something like

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10

and terminate the program.

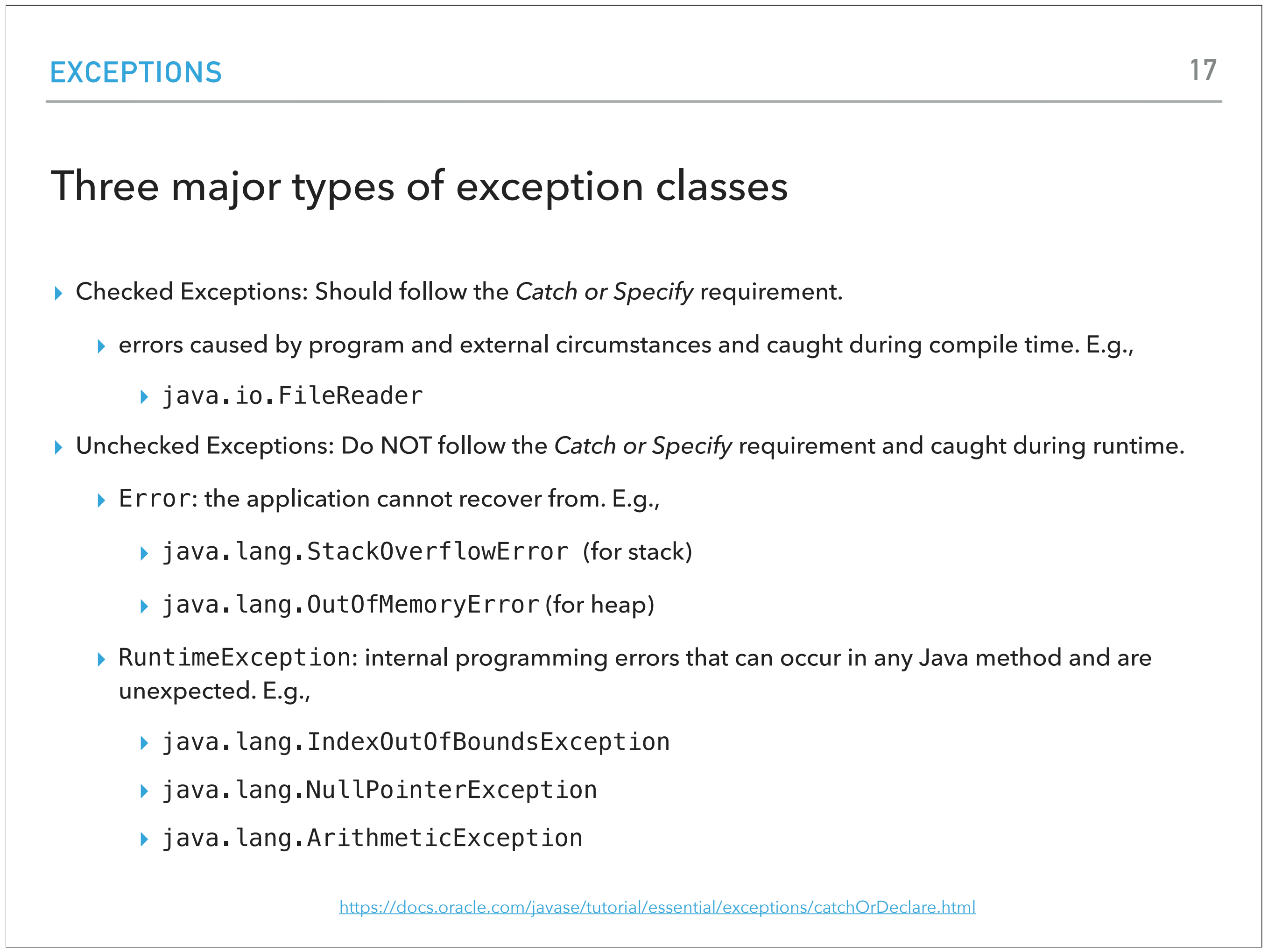


Exceptions are exceptional or unwanted events

- When an error occurs within a method, the method throws an exception object that contains its name, type, and state of program.
- The runtime system looks for something to handle the exception among the call stack, the list of methods called (in reverse order) by `main` to reach the error.
- The exception handler catches the exception. If no appropriate handler, the program terminates.

https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

This is achieved by Java throwing an exception object (throw an error). The object will contain an exception name and type, and information about the state of the program. The runtime system looks for something to handle the exception among the call stack, the list of methods called (in reverse order) by main to reach the error. The exception handler catches the exception. If no appropriate handler, the program terminates.

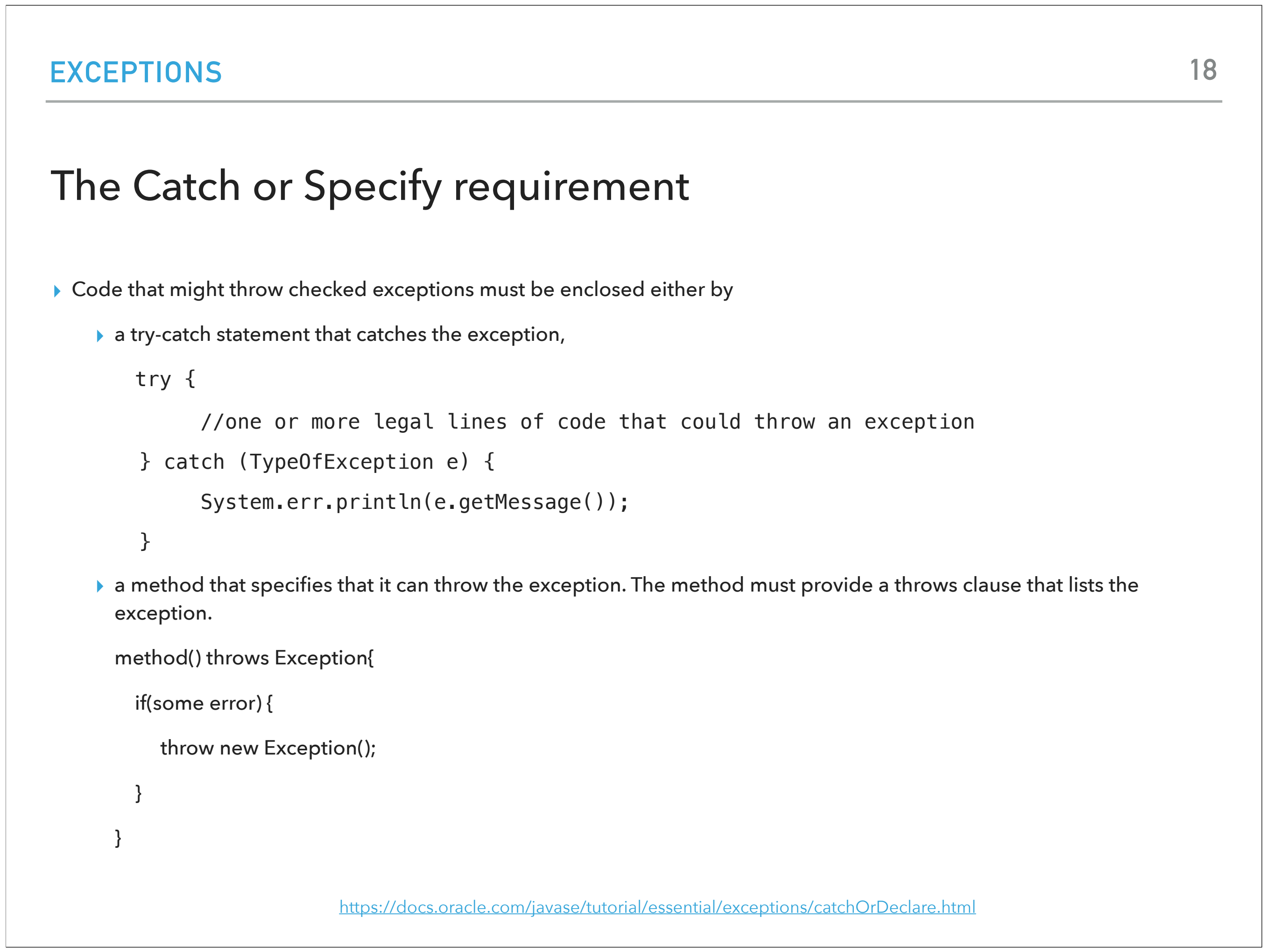


Three major types of exception classes

- Checked Exceptions: Should follow the *Catch or Specify* requirement.
  - errors caused by program and external circumstances and caught during compile time. E.g.,
    - `java.io.FileReader`
- Unchecked Exceptions: Do NOT follow the *Catch or Specify* requirement and caught during runtime.
  - `Error`: the application cannot recover from. E.g.,
    - `java.lang.StackOverflowError` (for stack)
    - `java.lang.OutOfMemoryError` (for heap)
  - `RuntimeException`: internal programming errors that can occur in any Java method and are unexpected. E.g.,
    - `java.lang.IndexOutOfBoundsException`
    - `java.lang.NullPointerException`
    - `java.lang.ArithmeticException`

https://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html

There are three major types of exception classes. The first belongs in checked exceptions and follow the catch or specify requirement which we'll see in a moment. Checked exceptions are caused by the program and external circumstances and are caught during compile time. For example, if our code is supposed to open and read a file and the file doesn't exist in the directory we specified, then we would get a java.io.FileReader
exception.
Unchecked exceptions reflect programming logic errors that are unrecoverable and which do not follow the catch or specify requirement and are caught during runtime. They include Errors and RuntimeExceptions. The application cannot recover from errors. e.g., we might run out of stack or heap memory, for example, because we have an infinite loop or because our objects are too many and too big. Runtime exceptions are internal programming errors that are unexpected. e.g., we try to access an index out of the bounds of an array, or we forgot to instantiate an object and try to use a reference that points to null, or we tried to divide with zero.



The Catch or Specify requirement

- Code that might throw checked exceptions must be enclosed either by
  - a try-catch statement that catches the exception,

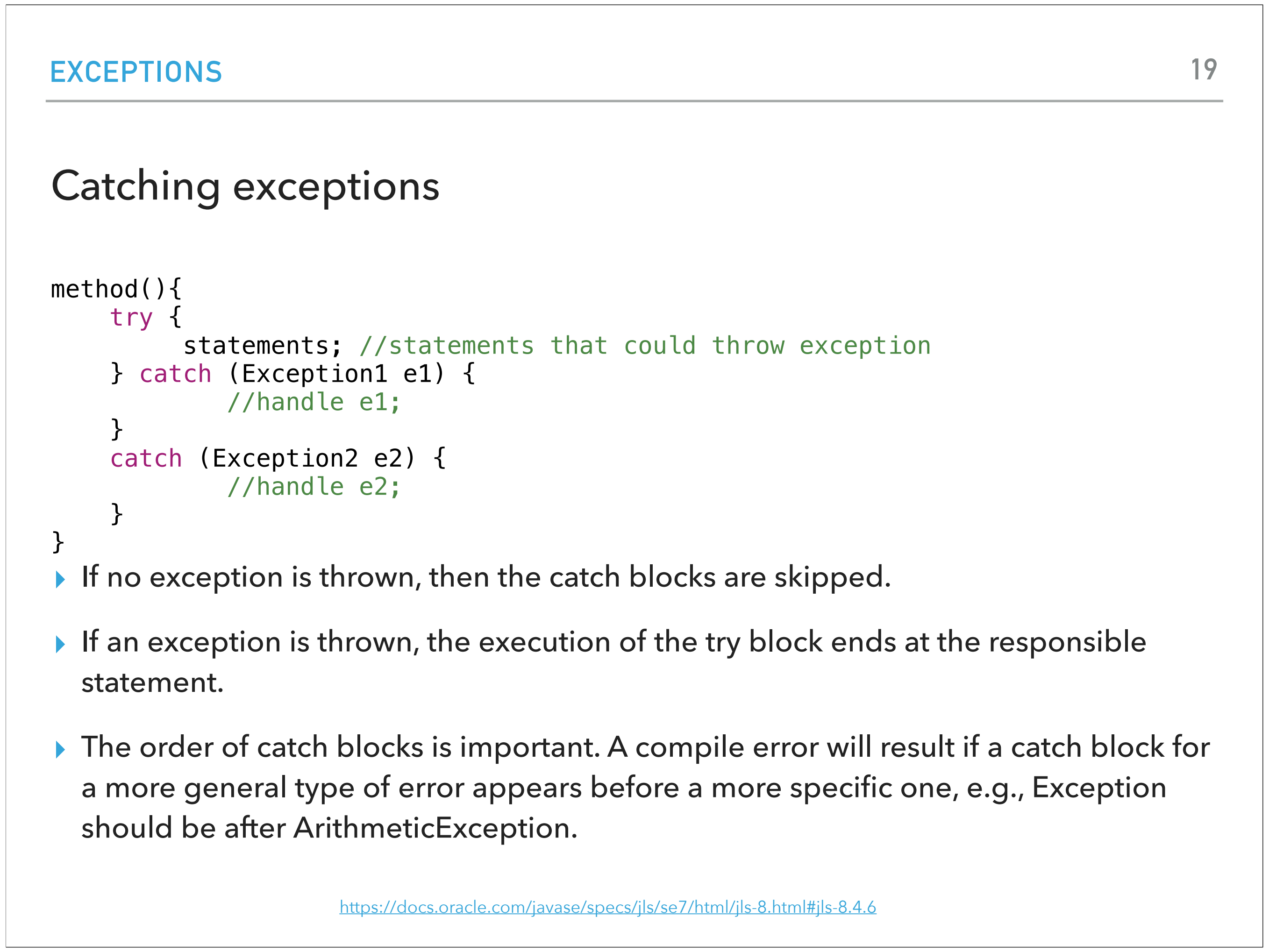
```
try {
     //one or more legal lines of code that could throw an exception
 } catch (TypeOfException e) {
     System.err.println(e.getMessage());
 }
```

  - a method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception.

```
method() throws Exception{
  if(some error) {
     throw new Exception();
  }
}
```

https://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html

The catch or specify requirement says that if we have code that might throw an exception, then we are required to either enclose it within a try catch statement that will catch the exception or specify that this method throws an exception and someone else will handle it.



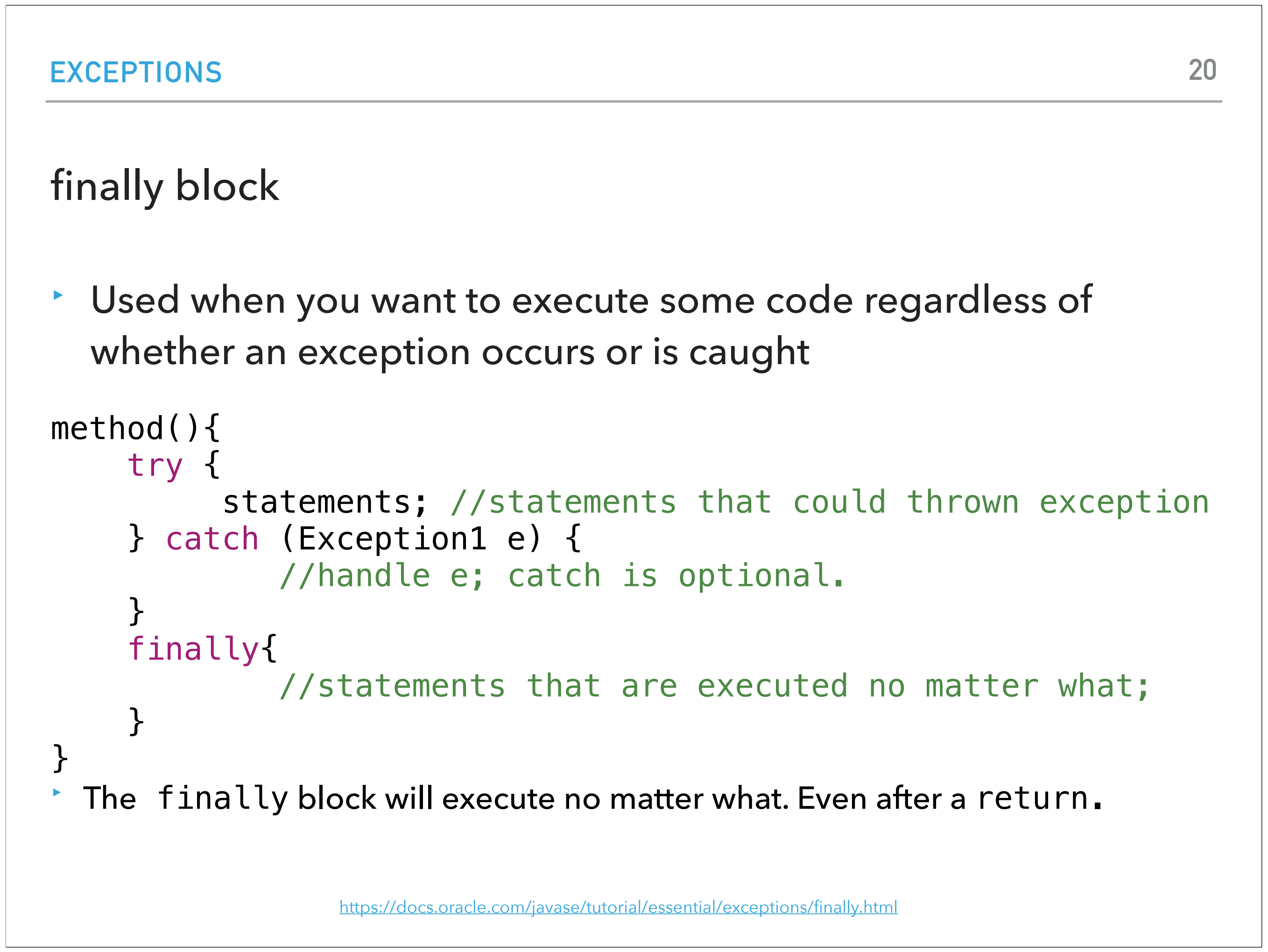
Catching exceptions

```
method(){
    try {
         statements; //statements that could throw exception
    } catch (Exception1 e1) {
            //handle e1;
    }
    catch (Exception2 e2) {
            //handle e2;
    }
}
```

- If no exception is thrown, then the catch blocks are skipped.
- If an exception is thrown, the execution of the try block ends at the responsible statement.
- The order of catch blocks is important. A compile error will result if a catch block for a more general type of error appears before a more specific one, e.g., Exception should be after ArithmeticException.

https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.4.6

Let's say we have some statements in our method that might throw an exception. To catch an exception, we would need to enclose these potentially offending statements in a try catch statement. If no exception is thrown, then the catch blocks are skipped. If an exception is thrown, then we stop at the responsible statement and jump to the catch blocks. If we have. More than one catch block, java will get in the first one that matches the type of exception that was thrown. What that means, is that the order of catch blocks is important. If we have a more general type of exception, it needs to follow a more specific one, e.g., first ArithmeticException, and then Exception.



finally block

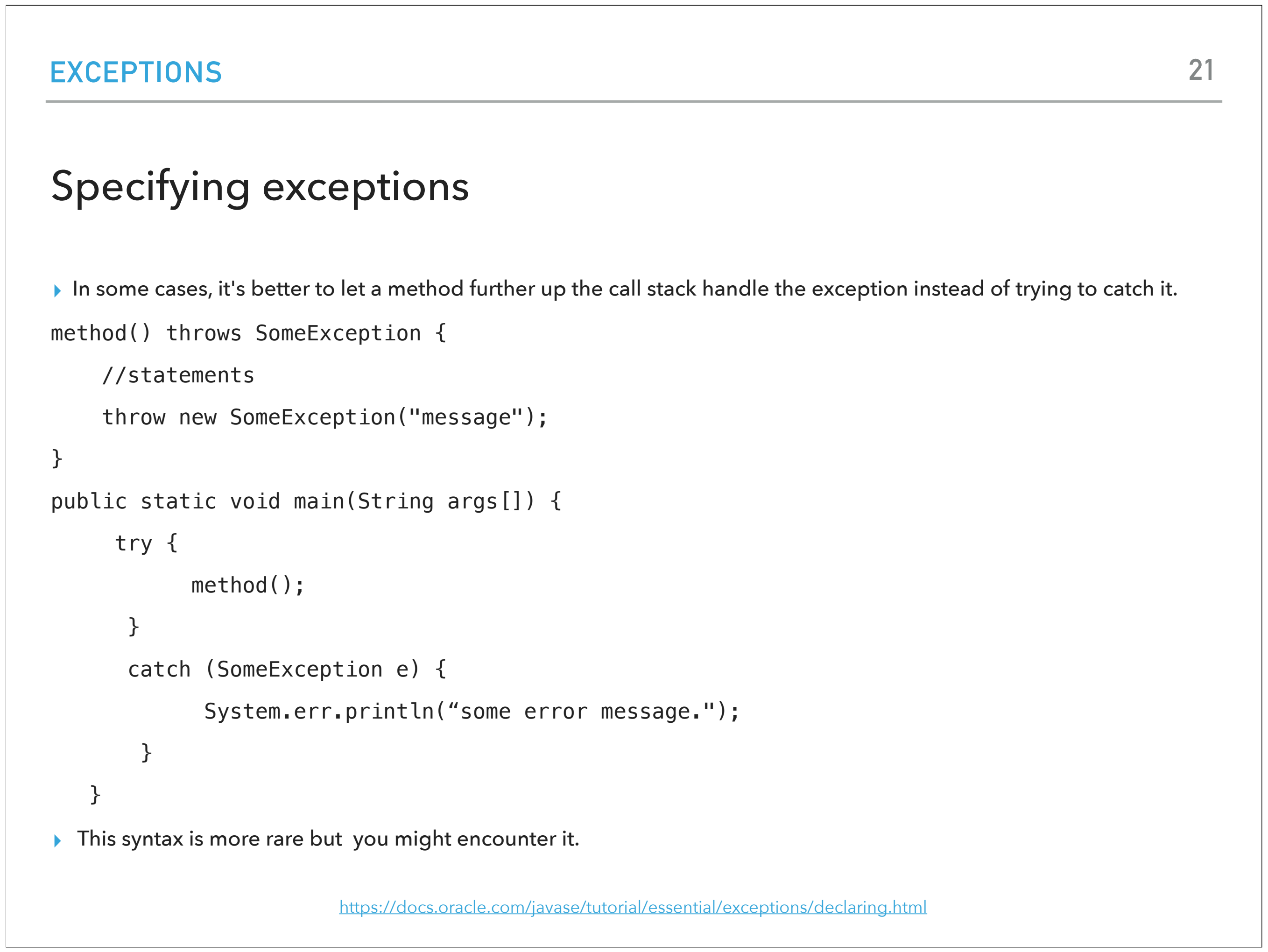
- Used when you want to execute some code regardless of whether an exception occurs or is caught

```
method(){
    try {
         statements; //statements that could thrown exception
    } catch (Exception1 e) {
            //handle e; catch is optional.
    }
    finally{
            //statements that are executed no matter what;
    }
}
```

- The `finally` block will execute no matter what. Even after a `return`.

https://docs.oracle.com/javase/tutorial/essential/exceptions/finally.html

The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.



Specifying exceptions

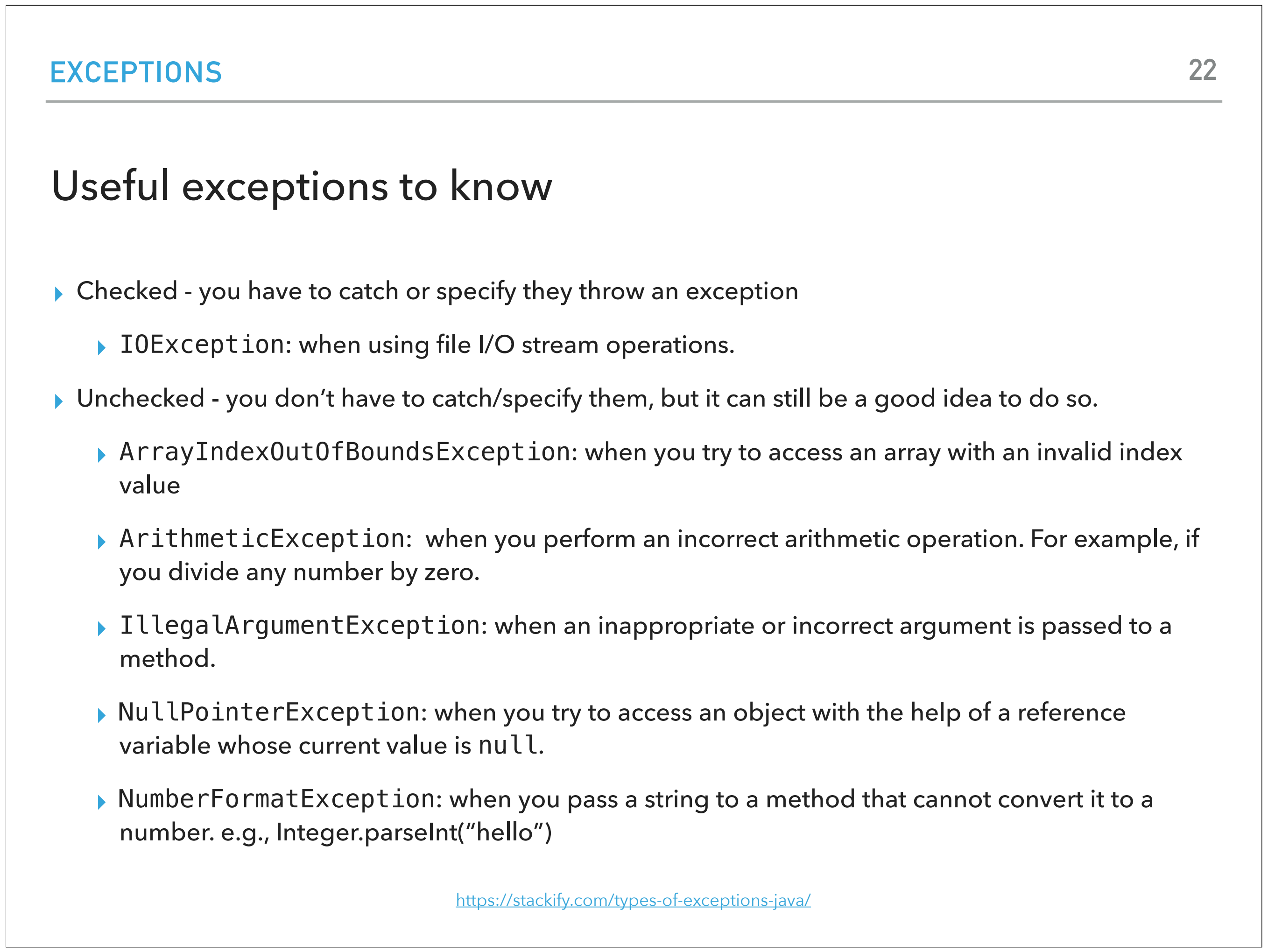
- In some cases, it's better to let a method further up the call stack handle the exception instead of trying to catch it.

```
method() throws SomeException {
    //statements
    throw new SomeException("message");
}
public static void main(String args[]) {
     try {
           method();
      }
      catch (SomeException e) {
            System.err.println(“some error message.");
       }
   }
```

- This syntax is more rare but you might encounter it.

https://docs.oracle.com/javase/tutorial/essential/exceptions/declaring.html

Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception. For example, you might write a method that new that it might throw some sort of exception, let’s say of type SomeException. Whoever called this method, would need to specify that it either throws SomeException or enclose it in a try catch statement as seen here. This syntax is more rate but you might encounter it.



Useful exceptions to know

- Checked - you have to catch or specify they throw an exception
    - `IOException`: when using file I/O stream operations.
- Unchecked - you don't have to catch/specify them, but it can still be a good idea to do so.
    - `ArrayIndexOutOfBoundsException`: when you try to access an array with an invalid index value
    - `ArithmeticException`: when you perform an incorrect arithmetic operation. For example, if you divide any number by zero.
    - `IllegalArgumentException`: when an inappropriate or incorrect argument is passed to a method.
    - `NullPointerException`: when you try to access an object with the help of a reference variable whose current value is `null`.
    - `NumberFormatException`: when you pass a string to a method that cannot convert it to a number. e.g., Integer.parseInt("hello")

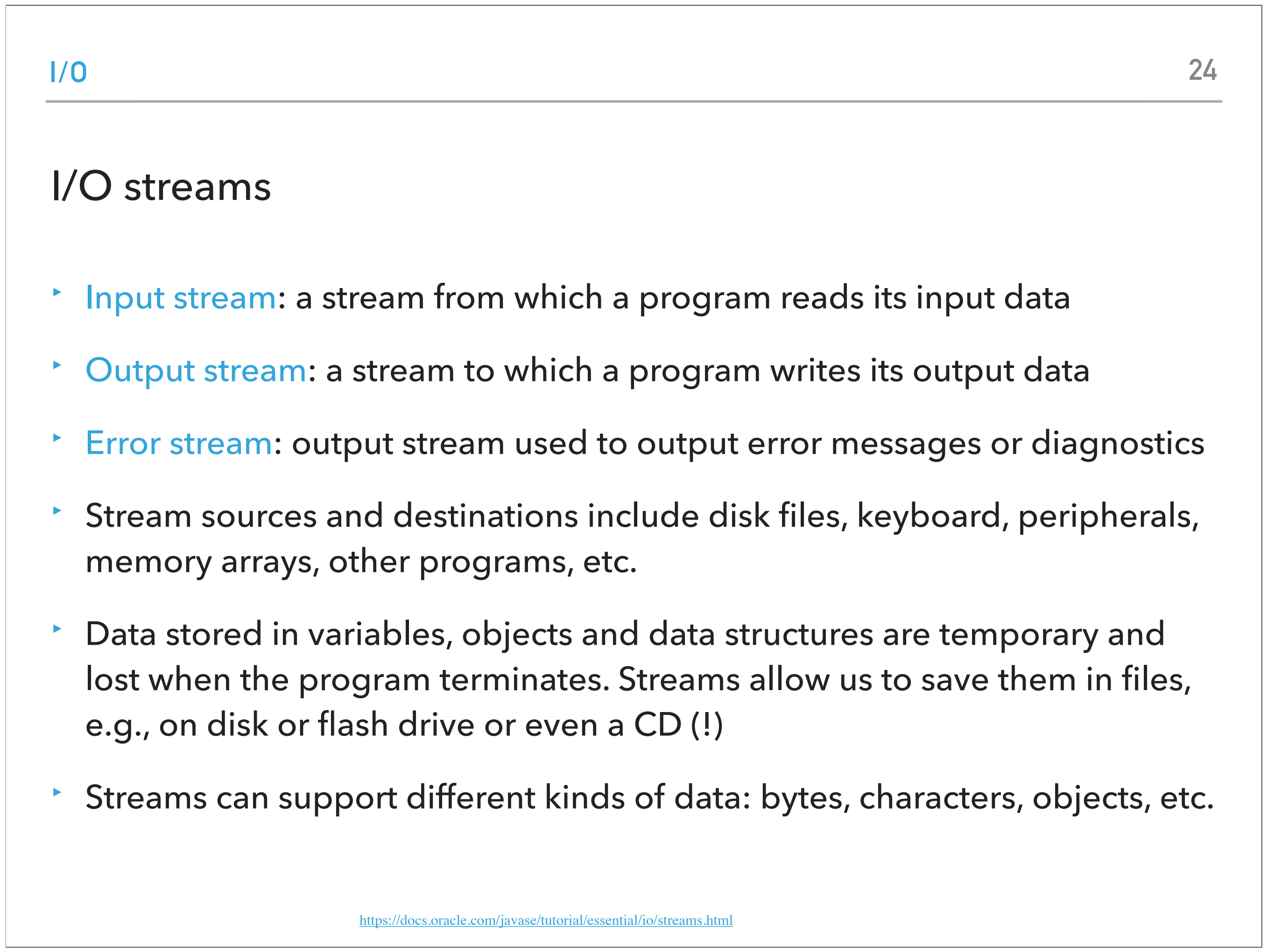
https://stackify.com/types-of-exceptions-java/

There are some useful types of exceptions to know divided by checked and unchecked. Checked
IOException: when using file I/O stream operations.
Unchecked
ArrayIndexOutOfBoundsException: when you try to access an array with an invalid index value
ArithmeticException: when you perform an incorrect arithmetic operation. For example, if you divide any number by zero.
IllegalArgumentException: when an inappropriate or incorrect argument is passed to a method.
NullPointerException: when you try to access an object with the help of a reference variable whose current value is null.
NumberFormatException: when you pass a string to a method that cannot convert it to a number. e.g., Integer.parseInt("hello")

Lecture 4: Memory Management, Exceptions, and I/O

- Memory Management
- Exceptions
- I/O

Now that we know how to handle errors, let's see how we can read input and write output (I/O) with our programs. We will focus on textual I/O but Java also supports working with binary files.



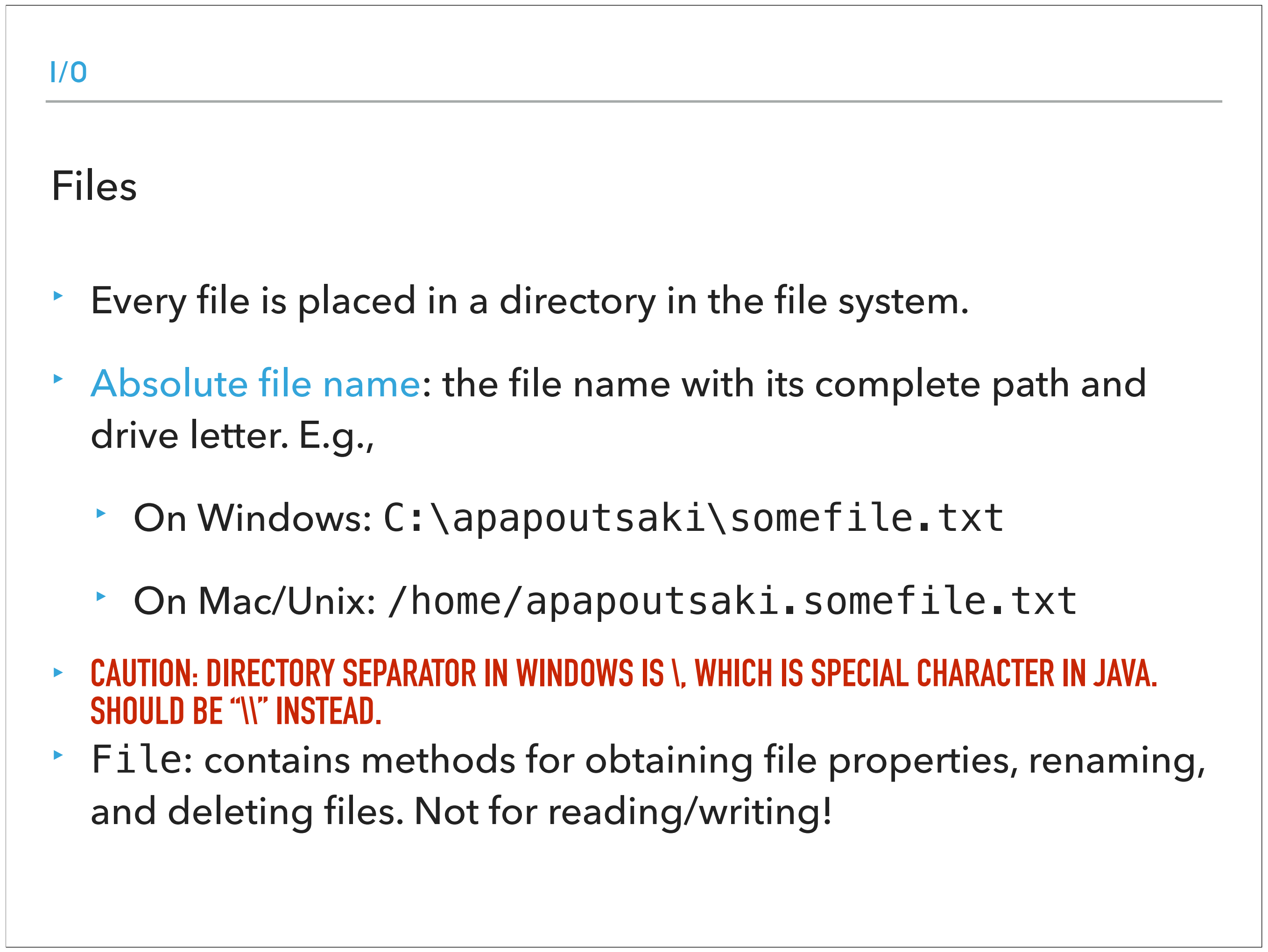
I/O streams

- Input stream: a stream from which a program reads its input data
- Output stream: a stream to which a program writes its output data
- Error stream: output stream used to output error messages or diagnostics
- Stream sources and destinations include disk files, keyboard, peripherals, memory arrays, other programs, etc.
- Data stored in variables, objects and data structures are temporary and lost when the program terminates. Streams allow us to save them in files, e.g., on disk or flash drive or even a CD (!)
- Streams can support different kinds of data: bytes, characters, objects, etc.

https://docs.oracle.com/javase/tutorial/essential/io/streams.html

There are three primary streams of data in a program. The input stream is a stream from which a program reads its input data. The output stream is a stream to which a program writes its output data. And finally the error stream is another output stream used to output error messages or diagnostics. It is a stream independent of standard output and can be redirected separately. Common stream sources and destinations include disk files, keyboard, peripherals, memory arrays, other programs, etc. Data stored in variables, objects and data structures are temporary and lost when the program terminates. Streams allow us to save them in files, e.g., on disk or flash drive or even a CD (!) Streams can support different kinds of data

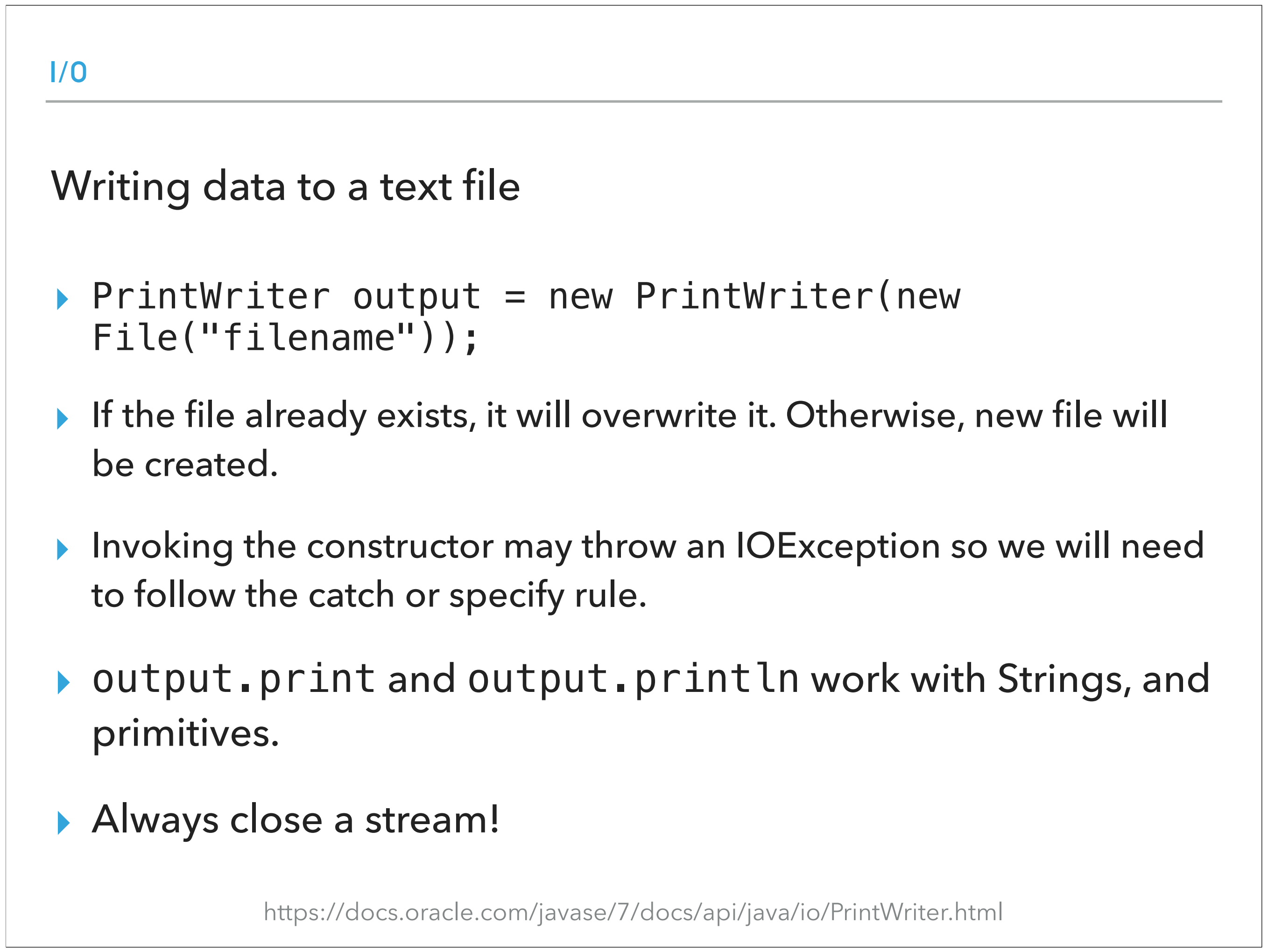
ASCII stands for American Standard Code for Information Interchange



Files

- Every file is placed in a directory in the file system.
- Absolute file name: the file name with its complete path and drive letter. E.g.,
  - On Windows: `C:\apapoutsaki\somefile.txt`
  - On Mac/Unix: `/home/apapoutsaki.somefile.txt`
- CAUTION: DIRECTORY SEPARATOR IN WINDOWS IS \, WHICH IS SPECIAL CHARACTER IN JAVA. SHOULD BE "\\" INSTEAD.
- `File`: contains methods for obtaining file properties, renaming, and deleting files. Not for reading/writing!

All of the files in your computer are organized in directories in the file system. A file has an absolute name which is the name of the file plus the complete path and an identifier of what drive it is stored. e.g., on windows that would be the C disk. On Mac and other Unix-based system that is typically in /home. Please note that the directory system for paths in Windows is a backward slash which is a special character in Java. You should put two \\ instead. Information about files in Java is handled through the class File. This class contains methods for obtaining file properties, renaming, and deleting files. It is not responsible for reading and writing files.



Writing data to a text file

- `PrintWriter output = new PrintWriter(new File("filename"));`
- If the file already exists, it will overwrite it. Otherwise, new file will be created.
- Invoking the constructor may throw an IOException so we will need to follow the catch or specify rule.
- `output.print` and `output.println` work with Strings, and primitives.
- Always close a stream!

https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html

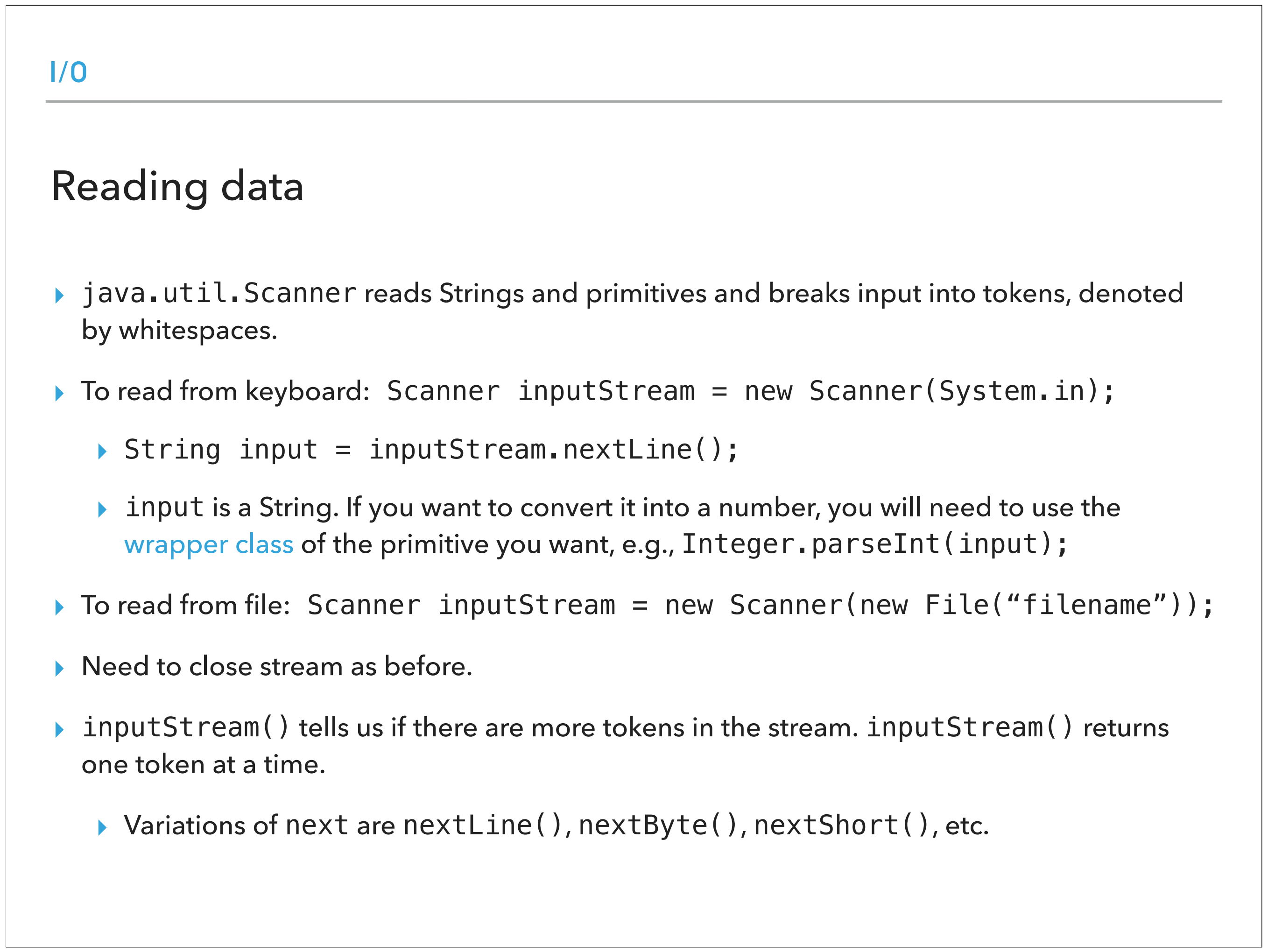
To write data to a text file, we will use the PrintWriter class. You will call its constructor and within it you will pass an object of type File. The constructor of File will take as an argument the name of the file you want to write to. If the file already exists, it will overwrite it. Otherwise, a new file will be created. Invoking the constructor may throw an IOException so we will need to follow the catch or specify rule. Once we have the PrintWriter reference, we can use the print and println methods to pass whatever Strings or primitives we want to write to the file. And don’t forget, we always need to close a stream at the end.

I/O

```
import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;

public class WriteData {
    public static void main(String[] args) {

        PrintWriter output = null;
        try {
            output = new PrintWriter(new File("addresses.txt"));
            // Write formatted output to the file
            output.print("Alexandra Papoutsaki ");
            output.println(222);
            output.print("Tzu-Yi Chen ");
            output.println(221);

        } catch (IOException e) {
            System.err.println(e.getMessage());
        } finally {
            if (output != null)
                output.close();
        }
    }
}
```

https://liveexample.pearsoncmg.com/html/WriteData.html

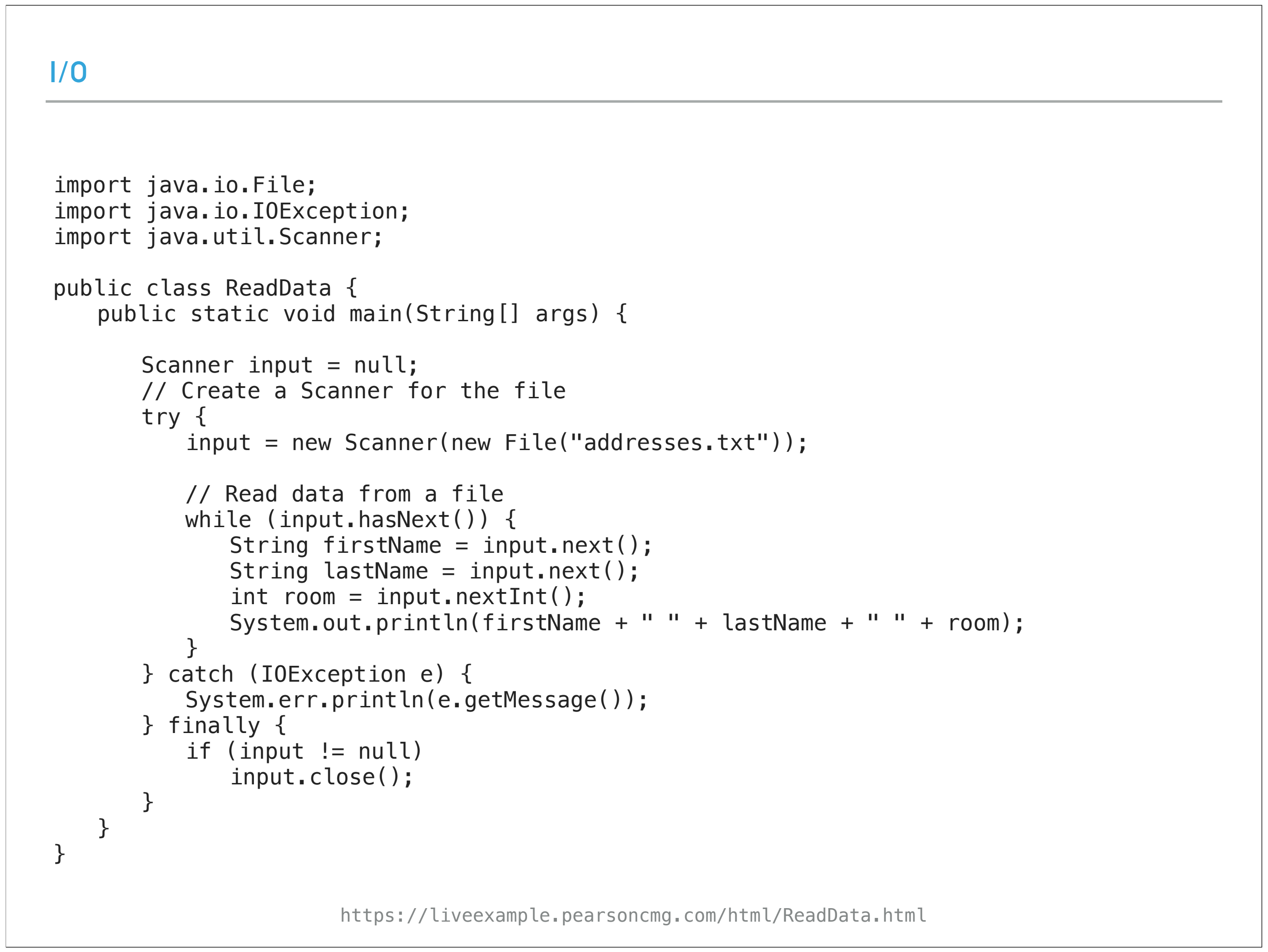
Here is an example of a class with a main method that writes some some string and integers to a file called addresses.txt. Notice that we have enclosed the code in a try catch statement that handles the potential IOException. To ensure that the output stream is closed, we enclose it within a finally statement which is executed even if an exception happens.



Reading data

- `java.util.Scanner` reads Strings and primitives and breaks input into tokens, denoted by whitespaces.
- To read from keyboard: `Scanner inputStream = new Scanner(System.in);`
  - `String input = inputStream.nextLine();`
  - `input` is a String. If you want to convert it into a number, you will need to use the wrapper class of the primitive you want, e.g., `Integer.parseInt(input);`
- To read from file: `Scanner inputStream = new Scanner(new File("filename"));`
- Need to close stream as before.
- `inputStream()` tells us if there are more tokens in the stream. `inputStream()` returns one token at a time.
  - Variations of `next` are `nextLine()`, `nextByte()`, `nextShort()`, etc.

If you want to read data, you will want to use the java.util.Scanner class which allow us to read Strings and primitives by breaking them into input tokens denoted by whitespaces by default (this can be changed). If you want to read input from the user's keyboard, you will want to pass system.in to the Scanner constructor. If you want to read from a text file, you will want to pass a File reference. As before, we need to close our input stream. Once we have a Scanner reference, we can use the hasNext method to ensure there are more tokens in the stream and the next or variations of next to read one token at a time.



I/O

```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class ReadData {
   public static void main(String[] args) {

      Scanner input = null;
      // Create a Scanner for the file
      try {
         input = new Scanner(new File("addresses.txt"));

         // Read data from a file
         while (input.hasNext()) {
            String firstName = input.next();
            String lastName = input.next();
            int room = input.nextInt();
            System.out.println(firstName + " " + lastName + " " + room);
         }
      } catch (IOException e) {
         System.err.println(e.getMessage());
      } finally {
         if (input != null)
            input.close();
      }
   }
}
```

https://liveexample.pearsoncmg.com/html/ReadData.html

Here is an example of a class that has a main method that reads from the file that we created. Note that we follow again the try/catch/finally structure. The while loop is very handy to keep reading as long as there are more tokens. Since we know that the file was structures as String String int, we can use the next next and nextInt methods.

PRACTICE TIME - Worksheet

- Write a Java class called FileIOExample
- It will contain a main method that will prompt the user for a String corresponding to a text file in their directory and a number for how many lines of text they want to read from that file.
- Use these two pieces of information to open the file, read the specified number of lines, and write them into a new file called output.txt.
- You can add whatever checks for exceptions you think are appropriate.
- Don't forget to close the input and output streams!

Let's put everything we learned today together. Write a Java class FileIOExample, which contains a main method that will prompt the user for a String corresponding to a text file in their directory and a number for how many lines of text they want to read from that file. Use these two pieces of information to open the file, read the specified number of lines, and write them into a new file called output.txt. You can add whatever checks for exceptions you think are appropriate. Don't forget to close the input and output streams!

ANSWER - Worksheet

- https://github.com/pomonacs622024sp/code/blob/main/Lecture4/FileIOExample.java

Lecture 4: Memory Management, Exceptions, and I/O

- Memory Management
- Exceptions
- I/O

And that's all for today: we've seen how Java uses a stack for static memory allocation and a heap for dynamic memory allocation. We learned about exceptions and the most important types to know. And we also learned how to read from the standard input or a file and how to write to text files.

Readings:

- Exceptions: https://docs.oracle.com/javase/tutorial/essential/exceptions/
- I/O: https://docs.oracle.com/javase/tutorial/essential/io

Code

- Lecture 4 code

Worksheet

- Lecture 4 worksheet