FUNDAMENTALS

# CS62

DATA STRUCTURES AND ADVANCED PROGRAMMING

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

Alexandra Papoutsaki
she/her/hers

## 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

## 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.

## Typical structure of a Java project

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

## Package

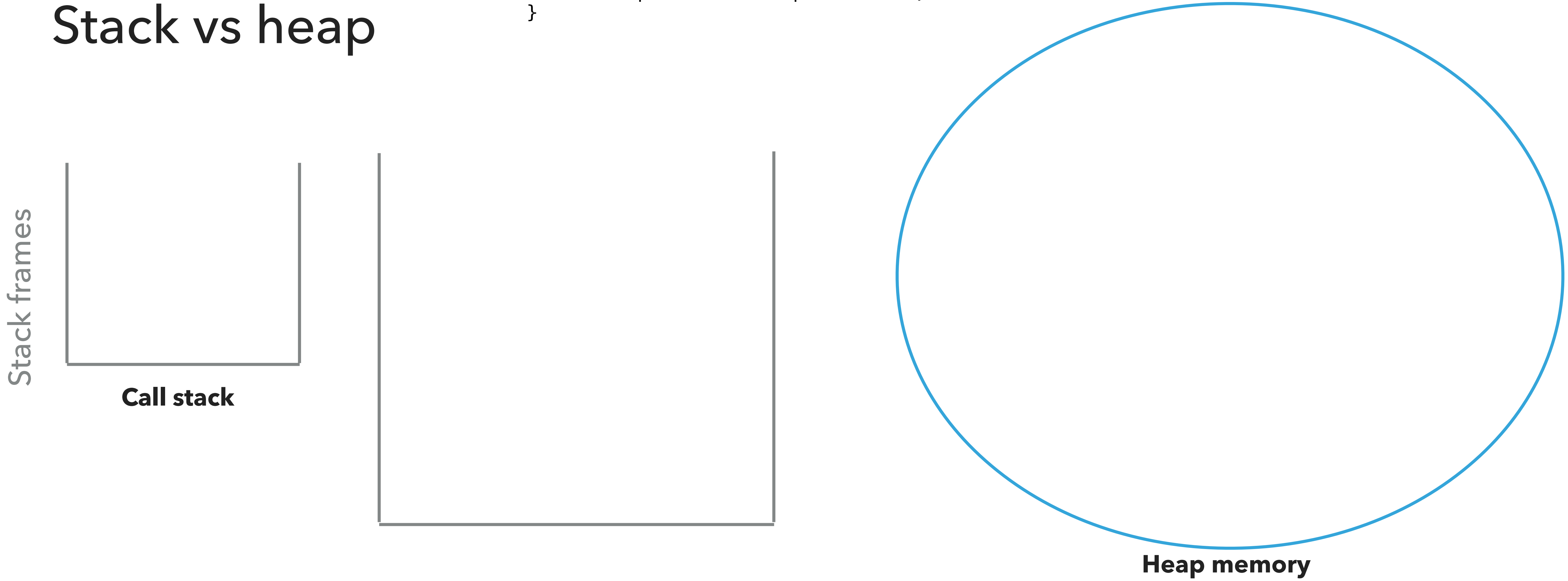
- A grouping of related classes that provides access protection and name space management. E.g.,
  - `java.lang` and `java.util` for fundamental classes or `java.io` for classes related to reading input and writing output.
- Packages correspond to folders/directories.
- Lower-case names. E.g.,
  - `package 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

```
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



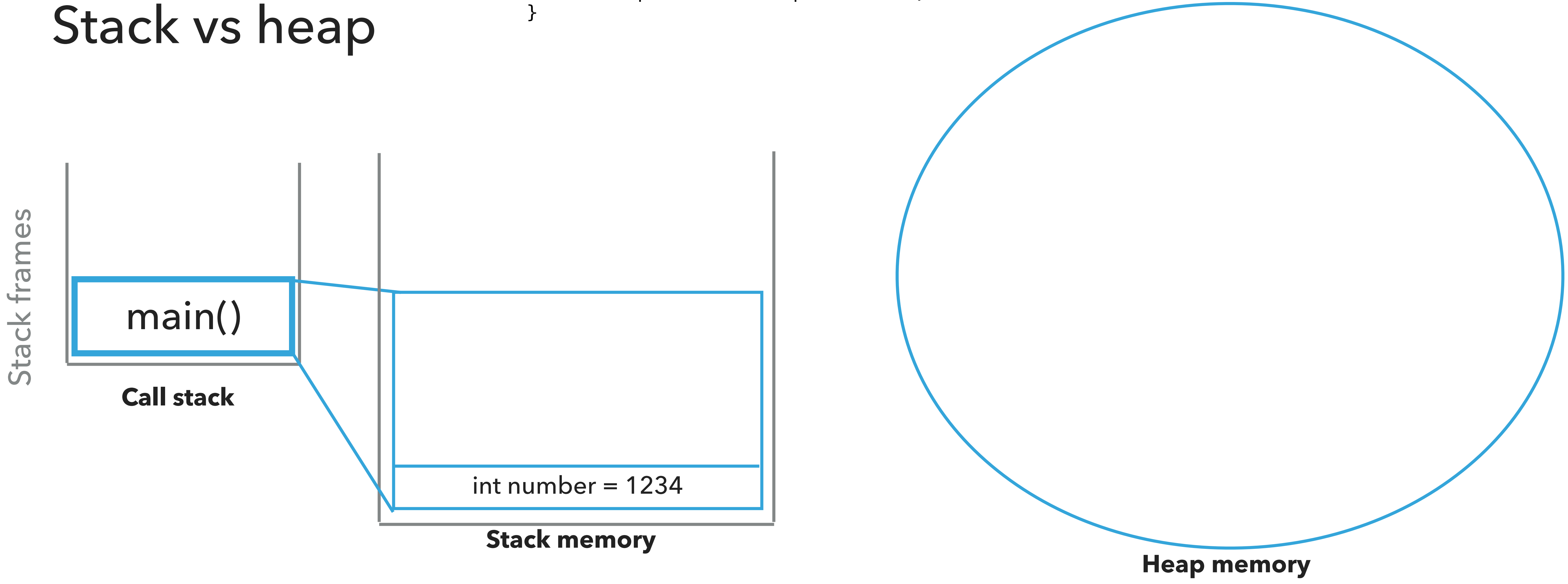
- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

- 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

```
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



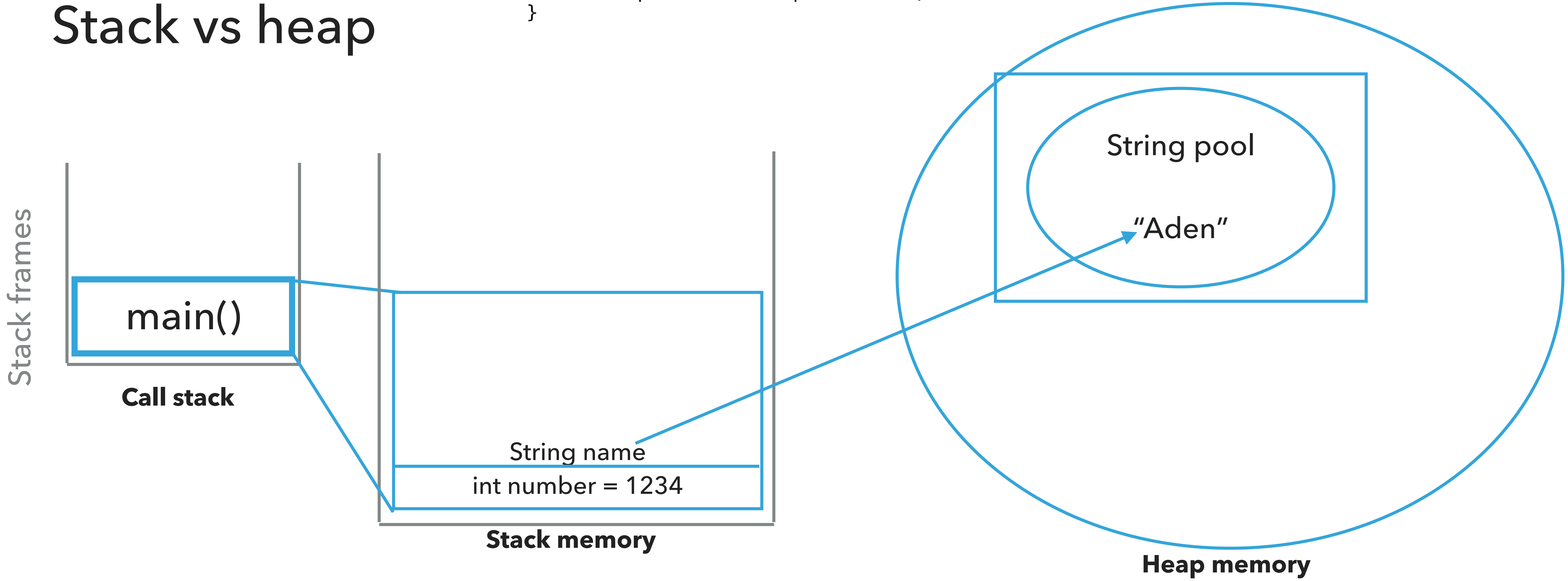
- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

- 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

```
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



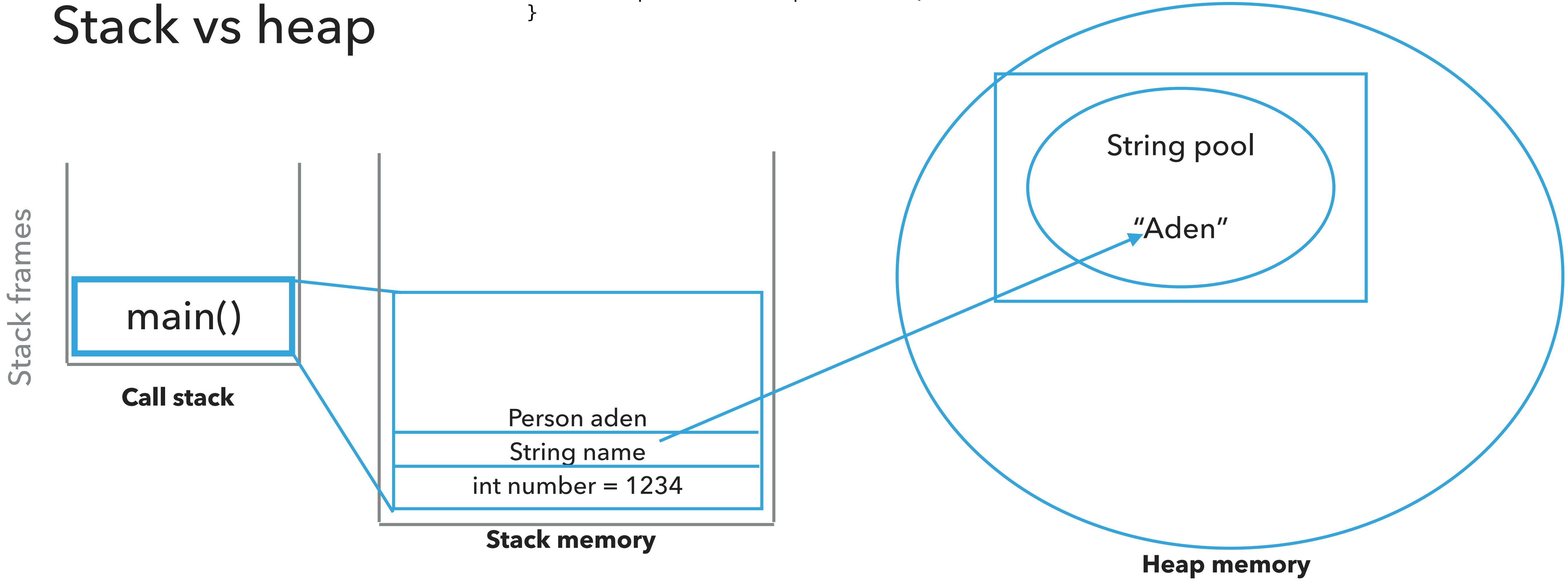
- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

- 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

```
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



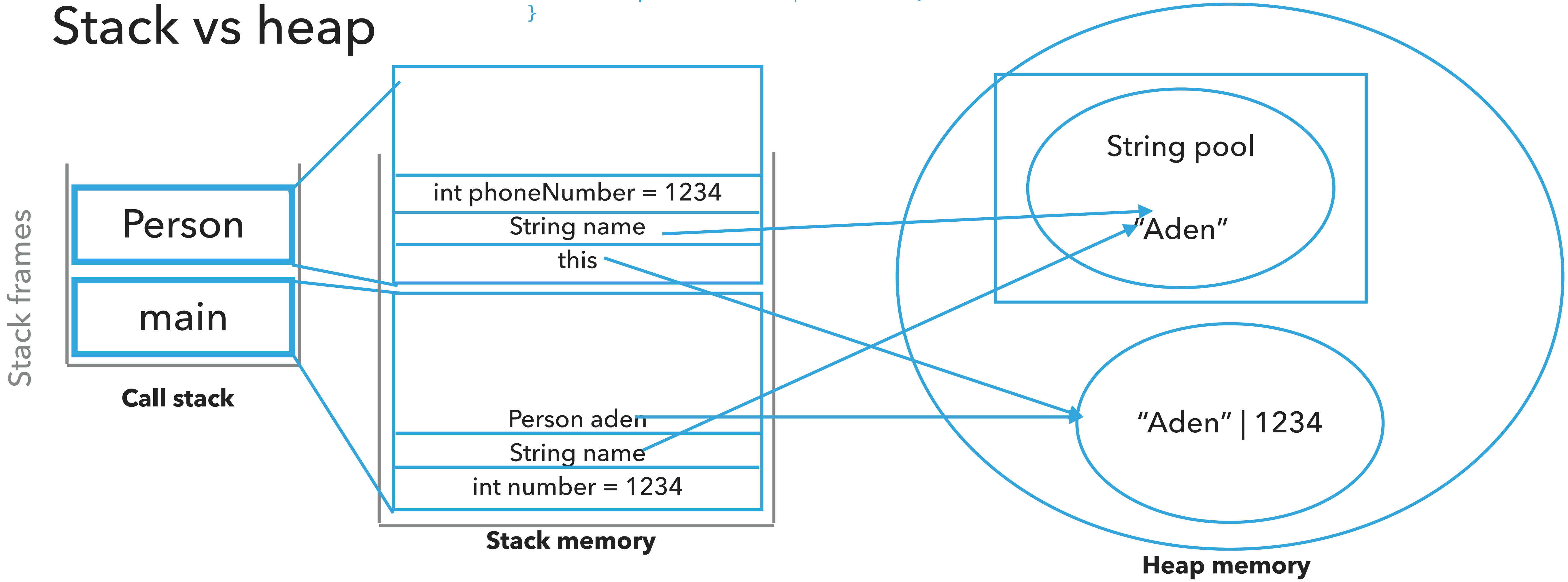
- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

- 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

```
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



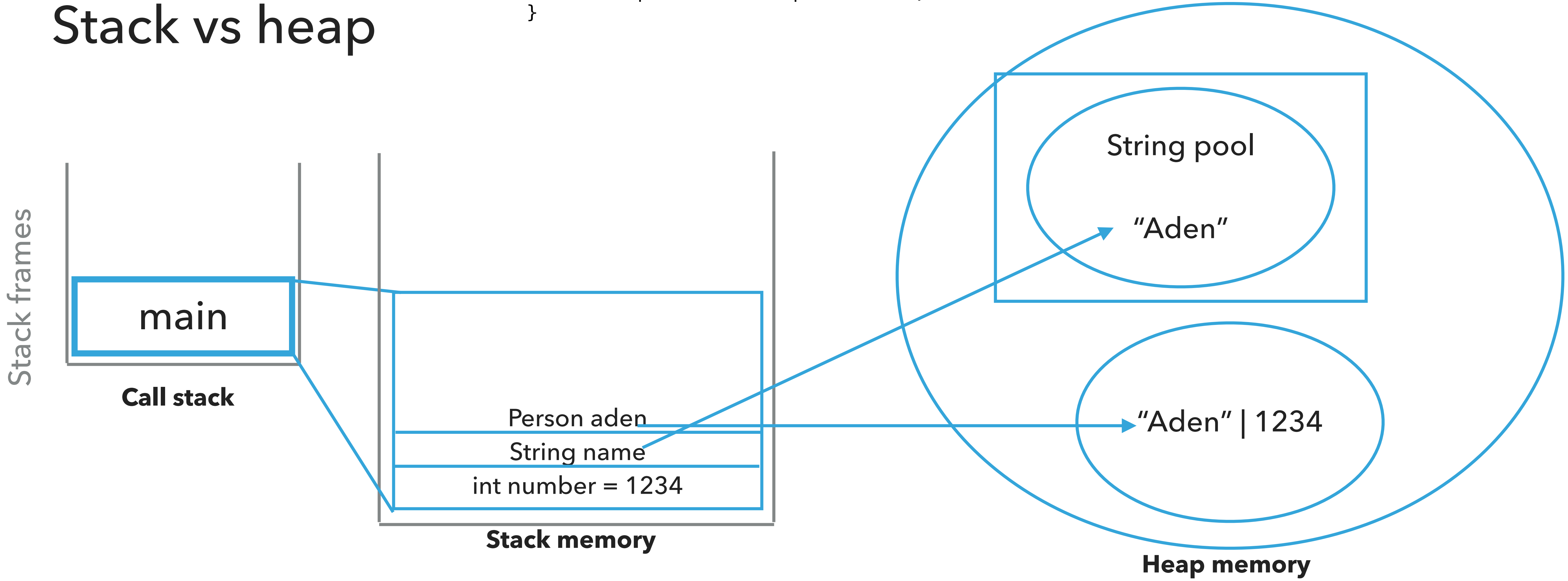
- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

- 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

```
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



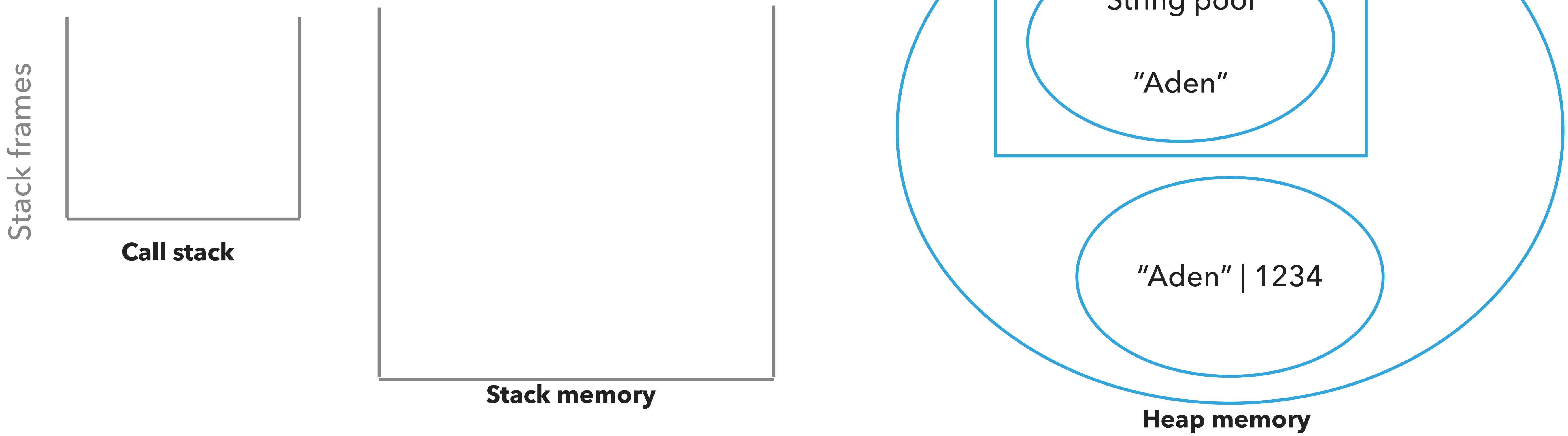
- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

- 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

```
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



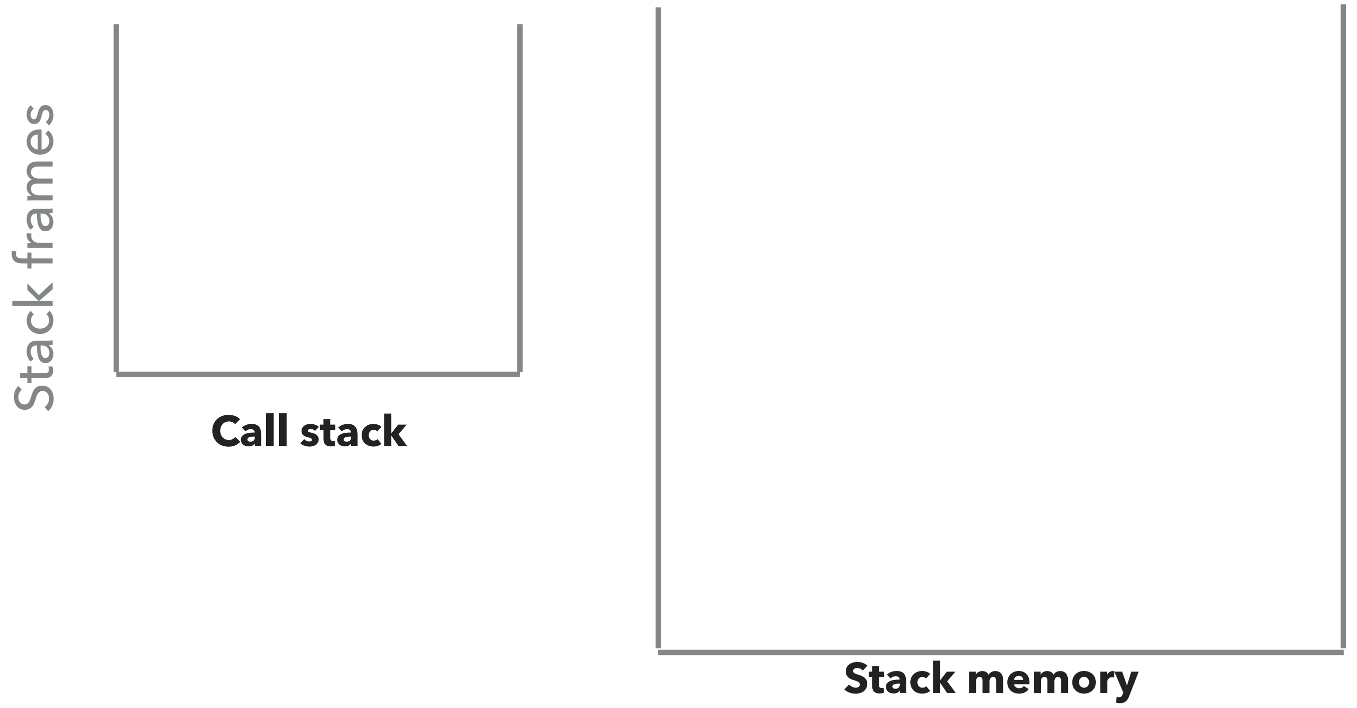
- Static memory allocation in Last-In-First-Out Order
- Whenever we call a method, a new frame is pushed to the top
- A method stack frame contains primitives and references to objects used in this method.
- When the method finishes, the stack frame gets popped
- Fast but limited in space

- 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

```
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



- 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

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

- Memory Management
- Exceptions
- I/O

## 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

## 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

## 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

## 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

## 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

## 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

## 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

## 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/

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

- Memory Management
- Exceptions
- I/O

## 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

## 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!

## 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

## I/O

```java
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

## 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.

## 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

## 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!

## ANSWER - Worksheet

- [https://github.com/pomonacs622024sp/code/blob/main/Lecture4/FileIOExample.java](https://github.com/pomonacs622024sp/code/blob/main/Lecture4/FileIOExample.java)

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

- Memory Management
- Exceptions
- I/O

## 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