

# CS062

## DATA STRUCTURES AND ADVANCED PROGRAMMING

### 4: The Catch-All Java Lecture

---



**Alexandra Papoutsaki**  
she/her/hers

## Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Memory Management
- ▶ Exceptions
- ▶ Assertions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

## What is a package?

- ▶ A grouping of related classes and interfaces that provides access protection and name space management.
- ▶ e.g., `java.lang` for fundamental classes or `java.io` for classes related to reading input and writing output.
- ▶ Packages correspond to folders/directories.
- ▶ Lower-case names.
- ▶ `package` whatevername; at top of file.
- ▶ `import` graphics.\*; for including all classes/interfaces.
- ▶ or `import` graphics.Circle; for more specific access.

## Access modifiers

Modifier	Class	Package	Subclass	World
<code>public</code>	Y	Y	Y	Y
<code>protected</code>	Y	Y	Y	N
No modifier	Y	Y	N	N
<code>private</code>	Y	N	N	N

## Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Memory Management
- ▶ Exceptions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events



# Java Documentation Generation System

- ▶ Reads JavaDoc comments and gives HTML pages
- ▶ JavaDoc comment = description written in HTML + tags
- ▶ Enclosed in `/**` `*/`
- ▶ Must precede class, variable, constructor or method declaration
- ▶ For class:
  - ▶ **@author** author name - classes and interfaces
  - ▶ **@version** date - classes and interfaces
- ▶ For method:
  - ▶ **@param** param name and description - methods and constructors
  - ▶ **@return** value returned, if any - methods
  - ▶ **@throws** description of any exceptions thrown - methods

## Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ **Memory Management**
- ▶ Exceptions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

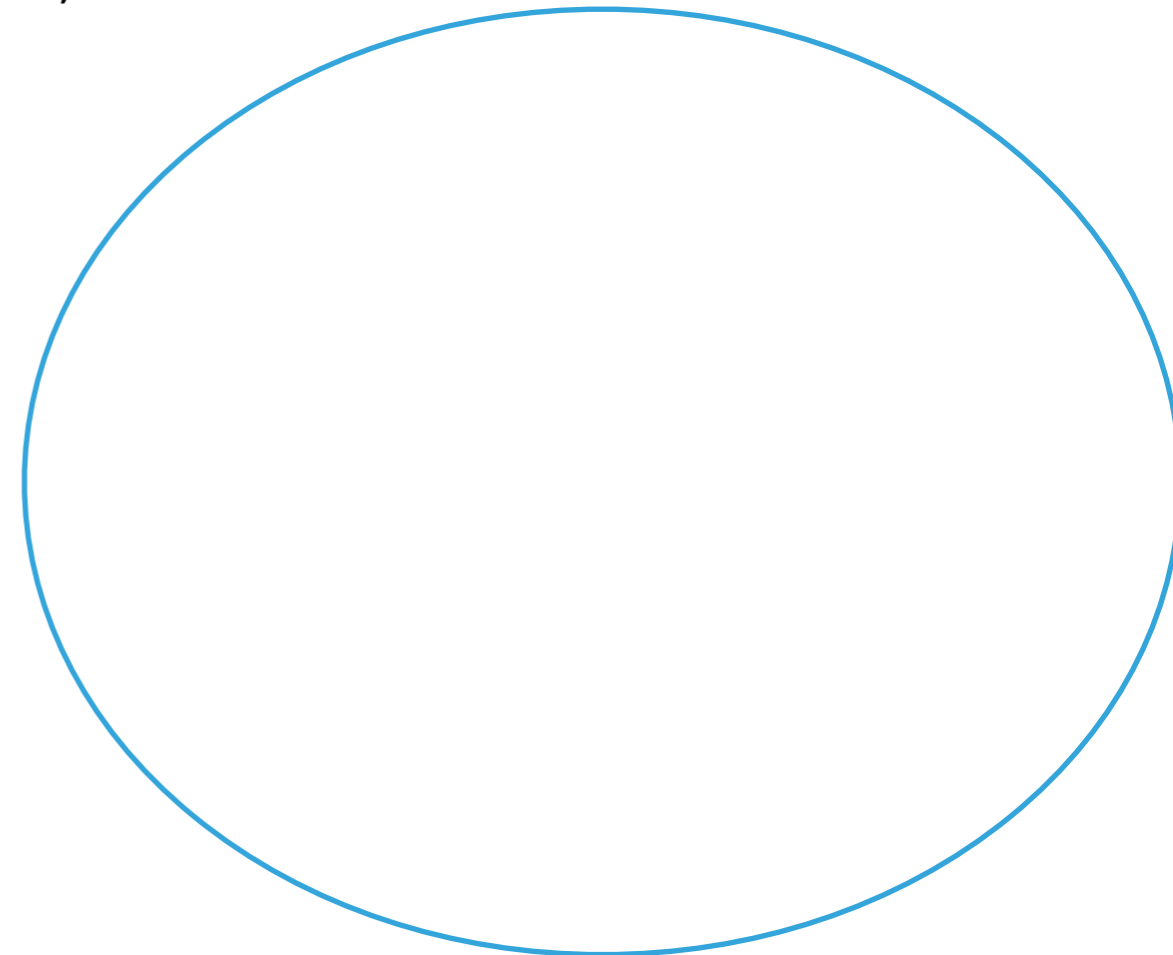
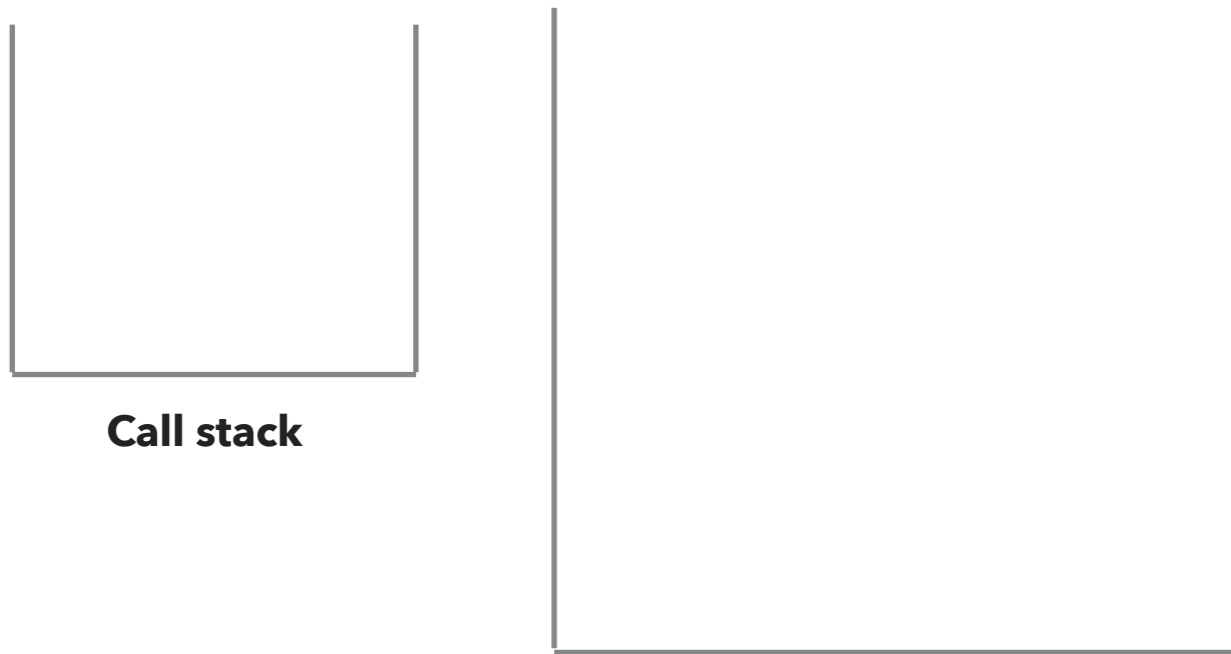
# MEMORY MANAGEMENT

```
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

Stack frames



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

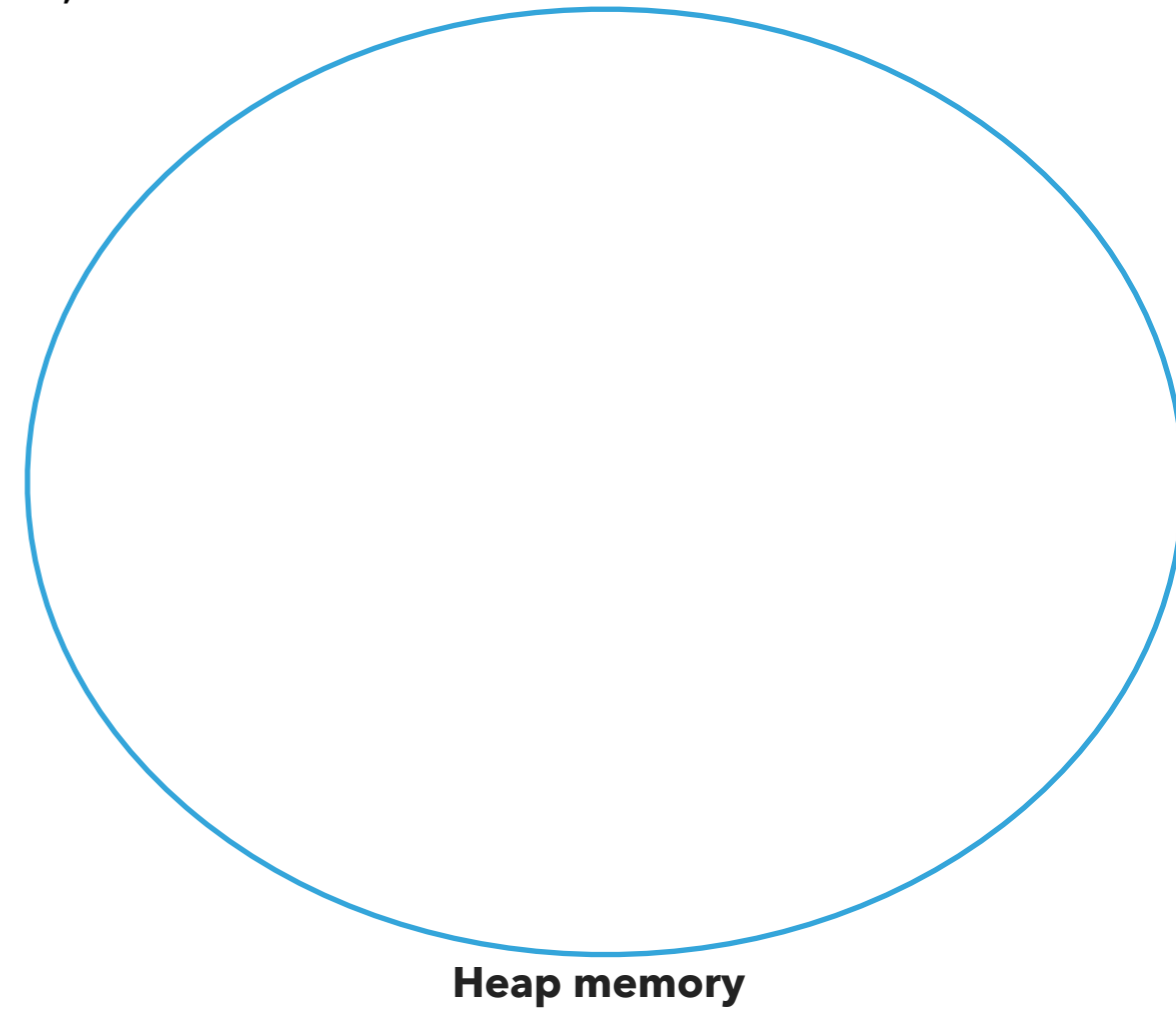
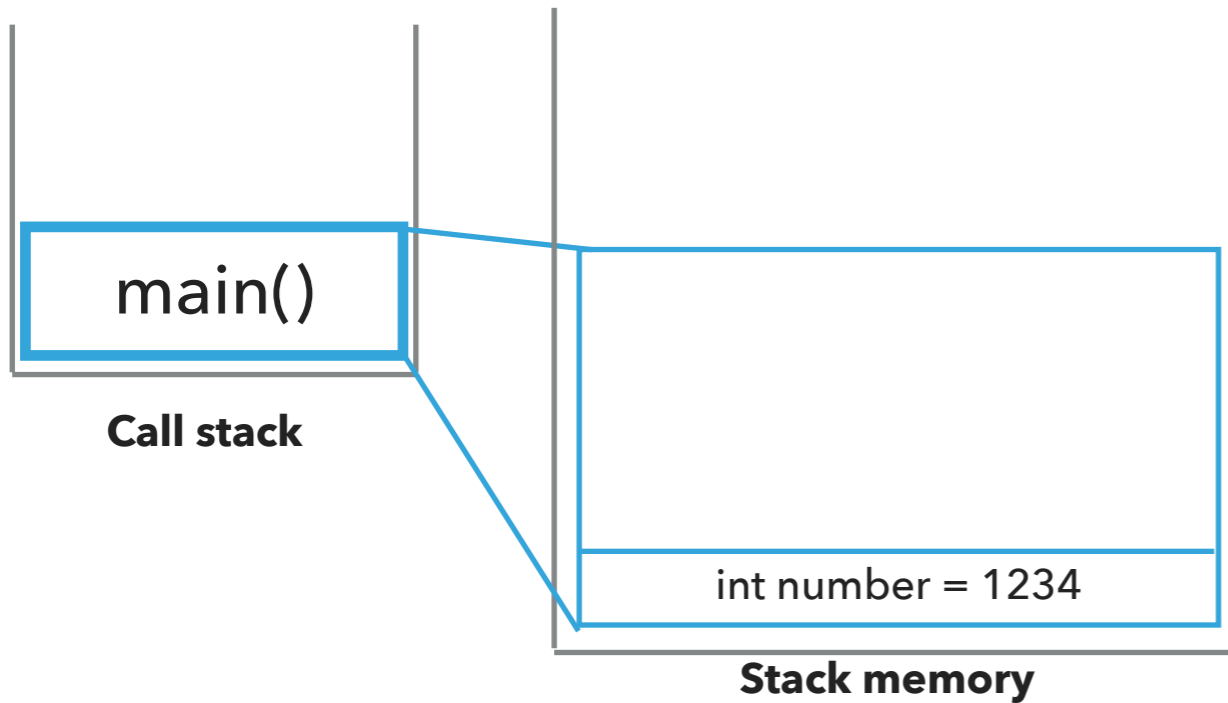


```
public class Person {  
    private String name;  
    private int phoneNumber;  
  
    public Person(String name, int phoneNumber) {  
        this.name = name;  
        this.phoneNumber = phoneNumber;  
    }  
}
```

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

## Stack vs heap

Stack frames



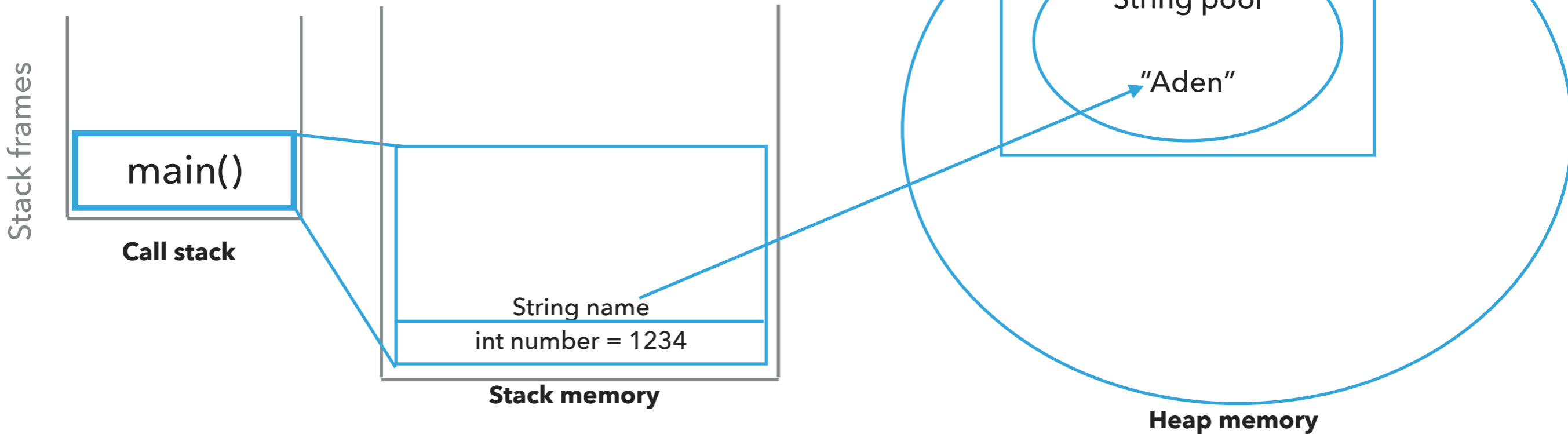
- ▶ Static memory allocation in Last-In-First-Out Order
- ▶ Whenever we call a method, a new frame is pushed to the top
- ▶ A method stack frame contains primitives and references to objects used in this method.
- ▶ When the method finishes, the stack frame gets popped
- ▶ Fast but limited in space

- ▶ 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

# MEMORY MANAGEMENT

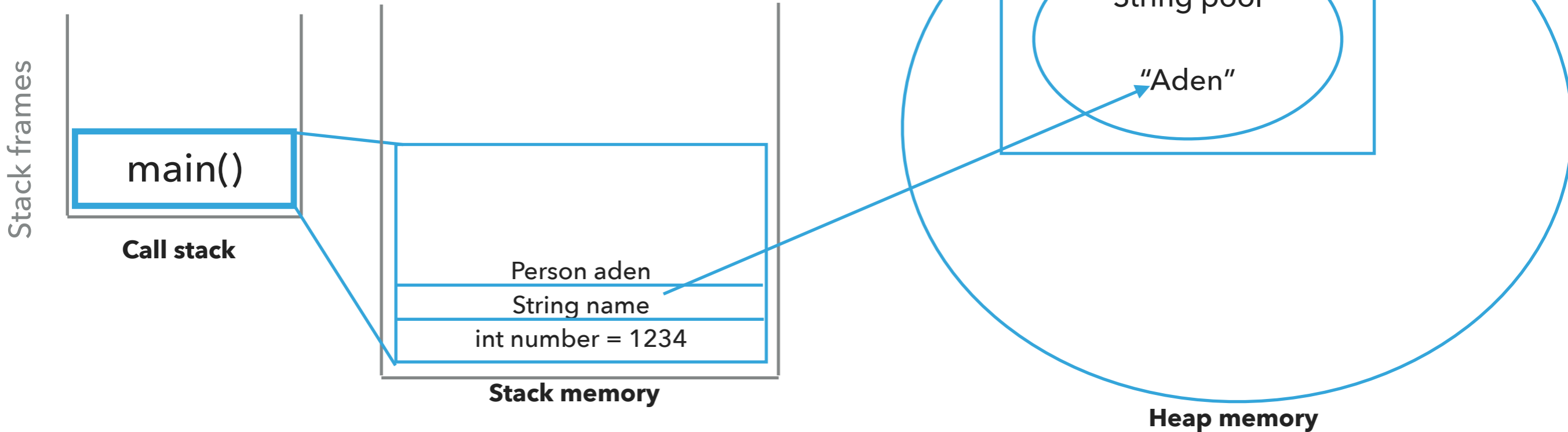
```
public class Person {
```

```
    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";  
    String name2 = "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

# MEMORY MANAGEMENT

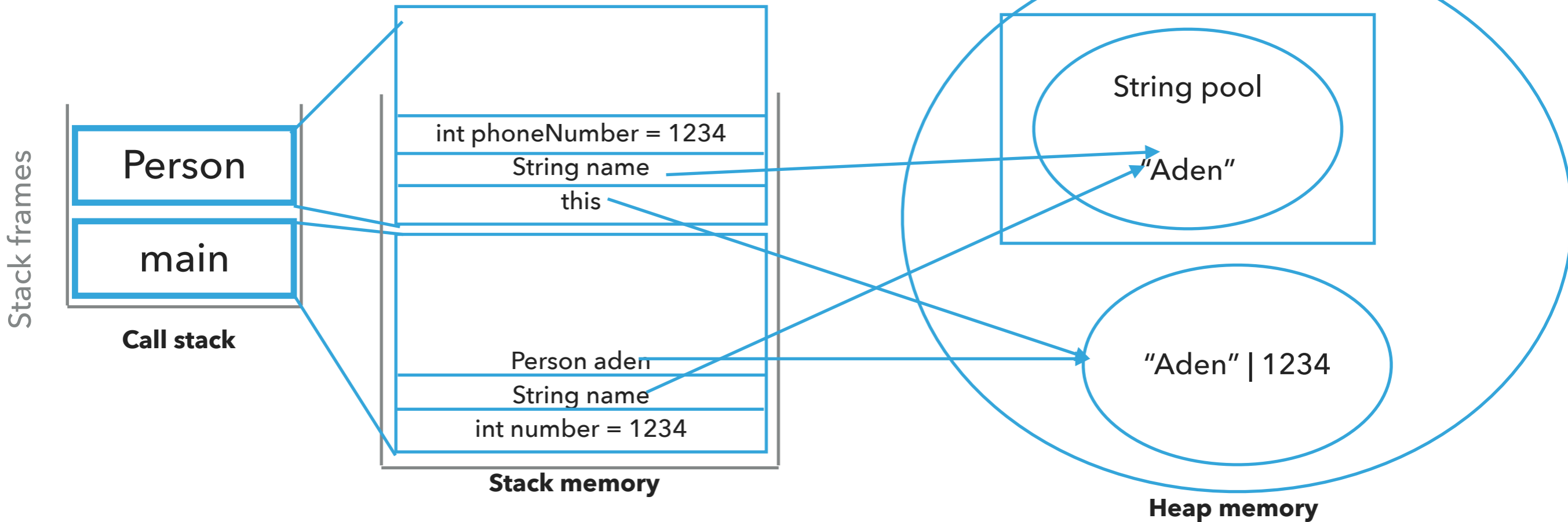
```
public class Person {
```

```
    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";  
    String name2 = "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

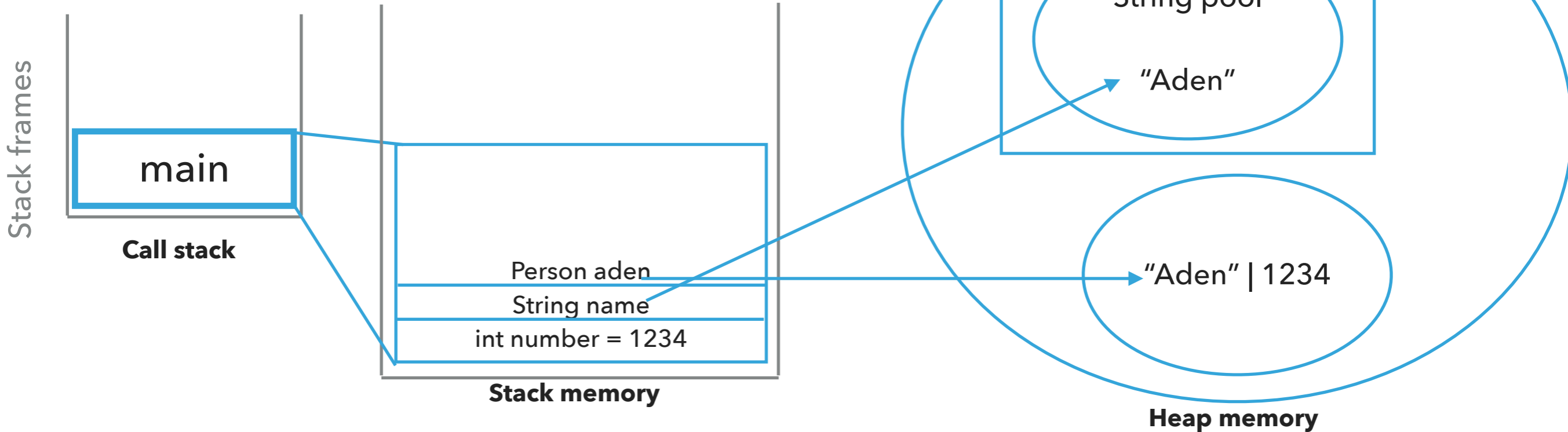
- ▶ Dynamic memory allocation
- ▶ New objects are stored there
- ▶ Strings are stored in a "string pool"
- ▶ Garbage collector frees up objects that do not get referenced anymore.
- ▶ Slow but much larger

# 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";  
    String name2 = "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

# MEMORY MANAGEMENT

```
public class Person {
```

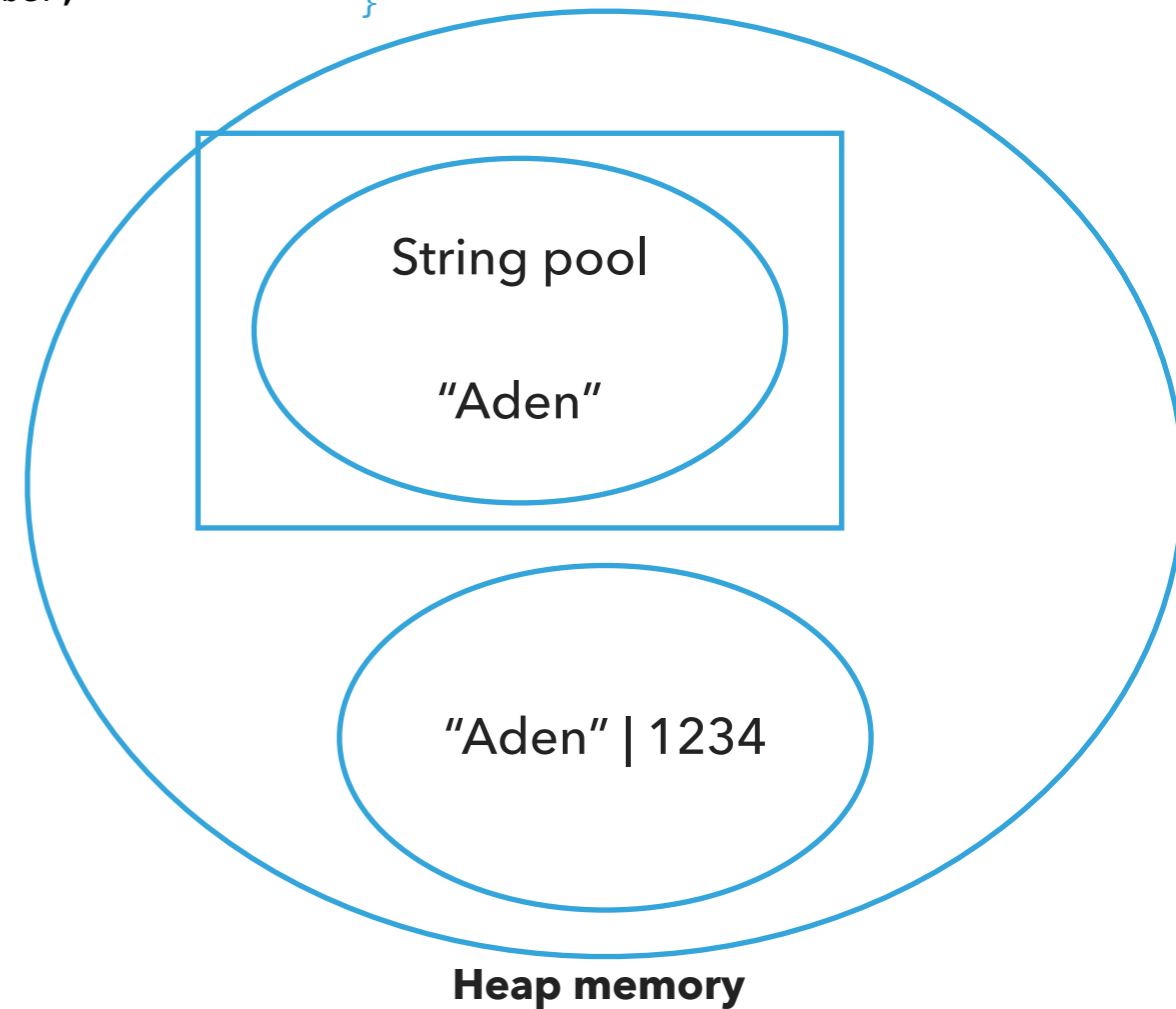
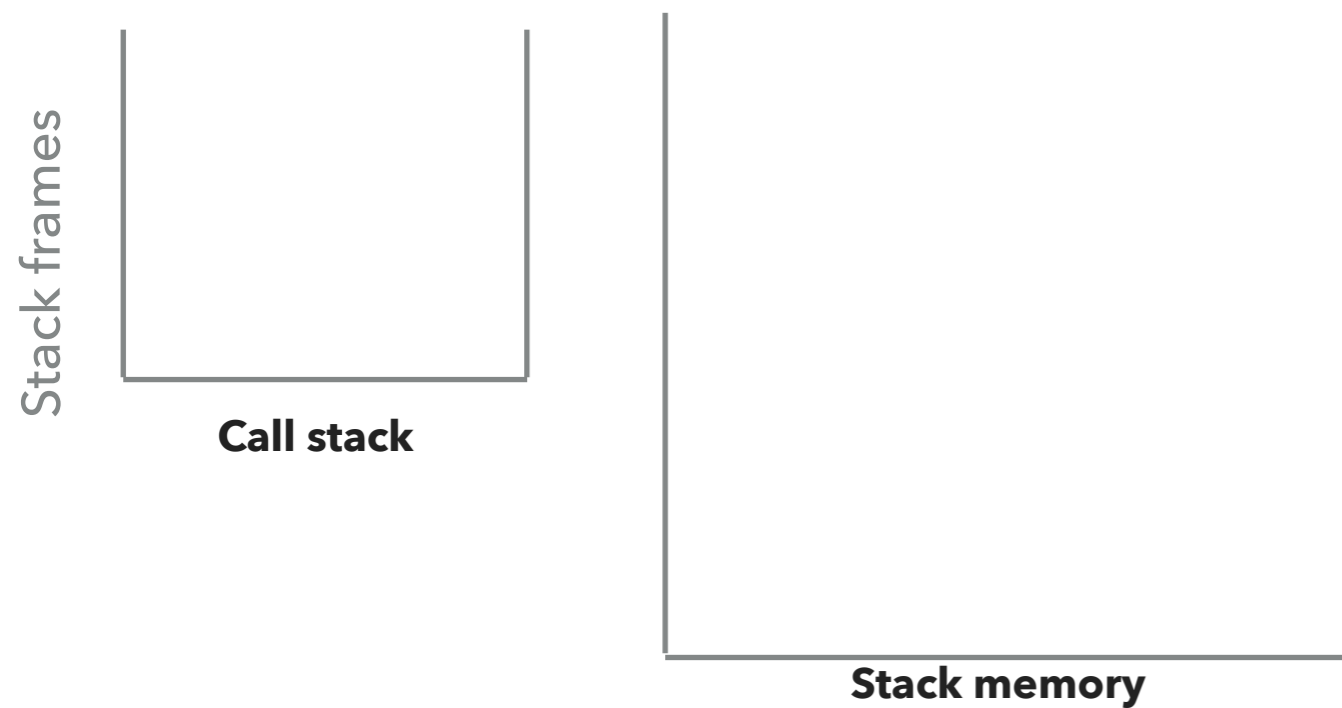
```
    private String name;  
    private int phoneNumber;
```

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

```
public static void main(String args[]) {  
    int number = 1234;  
    String name = "Aden";  
    String name2 = "Aden";  
    Person aden = null;  
    aden = new Person(name, number)  
}
```

14

## 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

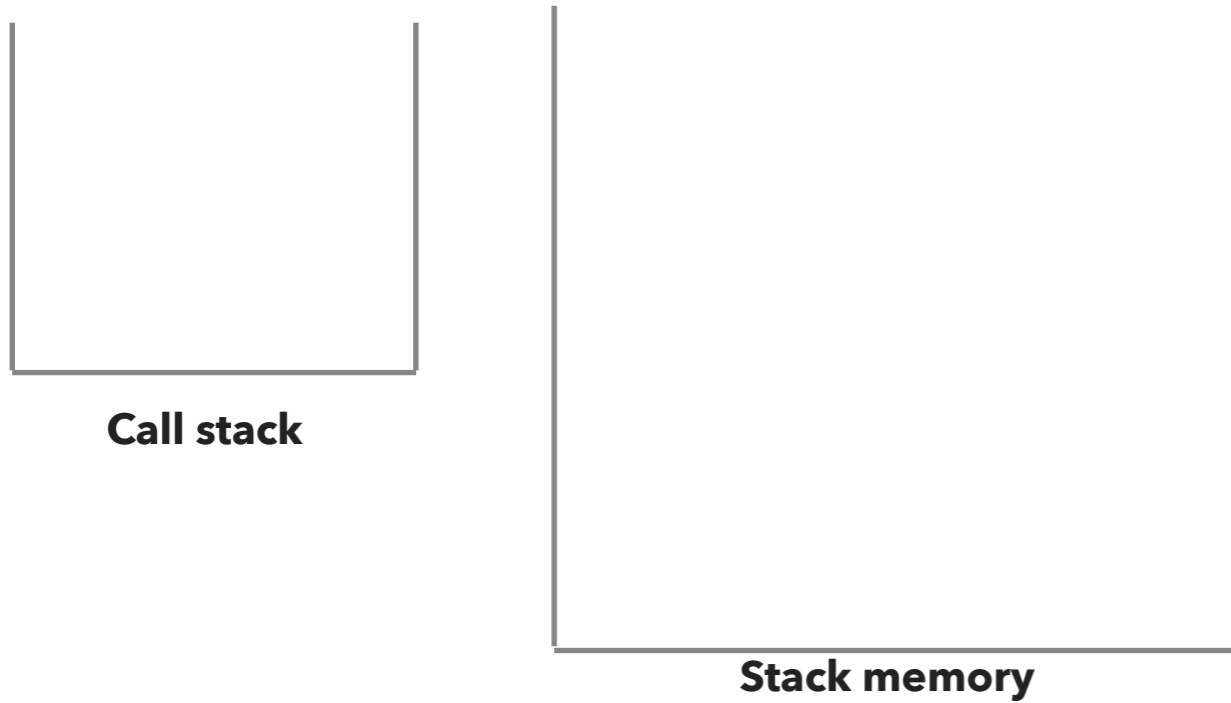
# MEMORY MANAGEMENT

```
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";  
    String name2 = "Aden";  
    Person aden = null;  
    Person aden = new Person(name, number);  
}
```

## Stack vs heap

Stack frames



- ▶ 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: The Catch-All Java Lecture

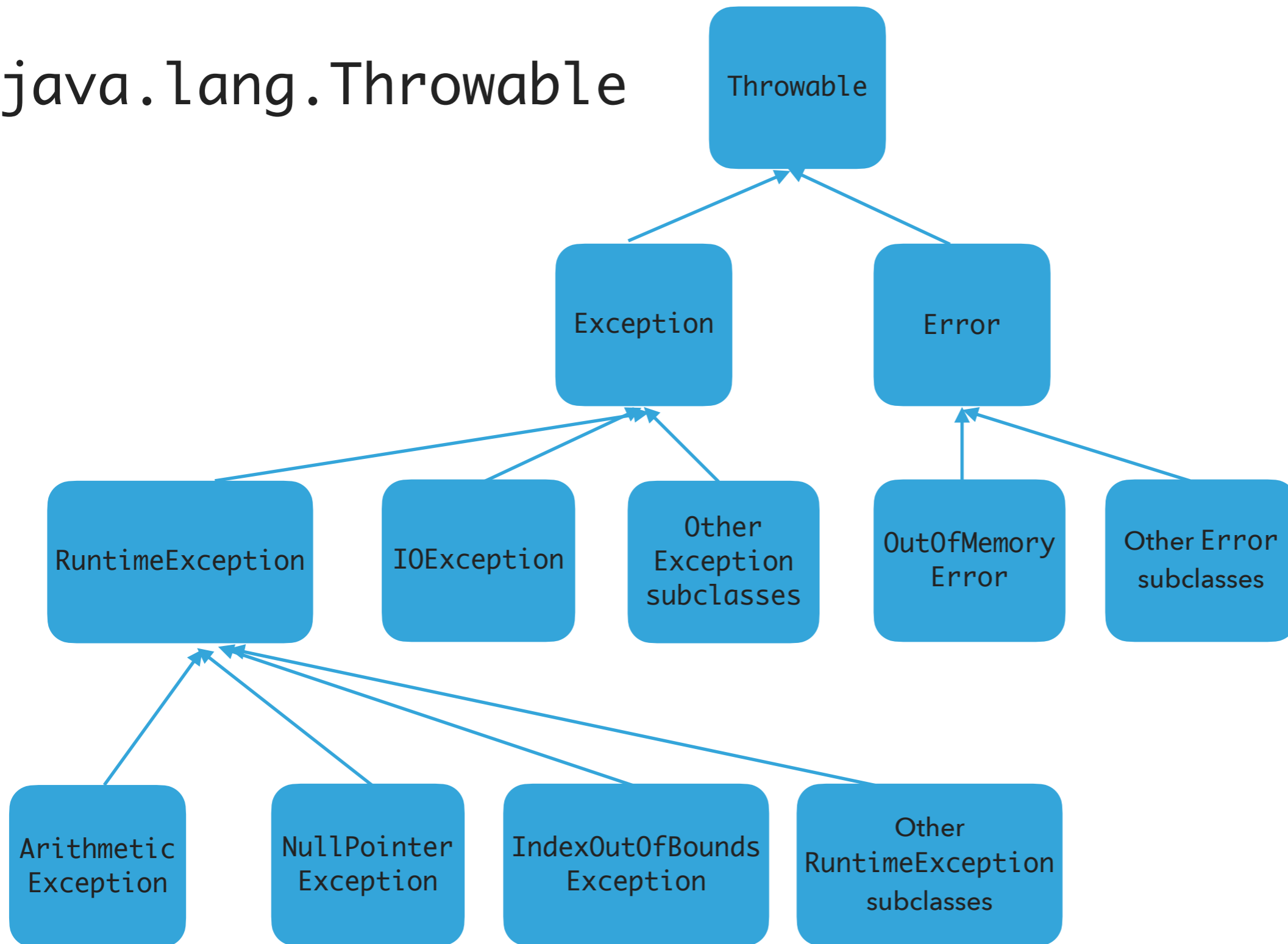
- ▶ Packages
- ▶ JavaDoc
- ▶ Memory Management
- ▶ **Exceptions**
- ▶ Assertions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events



## Exceptions are exceptional or unwanted events

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

# java.lang.Throwable



## Three major types of exception classes

- ▶ **Error**: rare internal system errors that an application cannot recover from.
  - ▶ Typically not caught and program has to terminate.
  - ▶ e.g., `java.lang.StackOverflowError` (for stack) or `java.lang.OutOfMemoryError` (for heap)
- ▶ **Exception**: errors caused by program and external circumstances.
  - ▶ Can be caught and handled.
  - ▶ e.g., `java.io.IOException`
- ▶ **RuntimeException**: programming errors that can occur in any Java method.
  - ▶ Method not required to declare that it throws any of the exception.
  - ▶ e.g., `java.lang.IndexOutOfBoundsException`, `java.lang.NullPointerException`, `java.lang.ArithmeticException`
- ▶ **Unchecked exceptions**: programming logic errors that are unrecoverable. Error and RuntimeException and subclasses, e.g., `java.lang.NullPointerException`, `java.lang.IndexOutOfBoundsException`
- ▶ **Checked exceptions**: All other exceptions - programmer has to check and deal with them.

# Handling exceptions

- ▶ Three operations:
  - ▶ Declaring an exception
  - ▶ Throwing an exception
  - ▶ Catching an exception

```
method1(){
```

```
    try {  
        method2();  
    } catch (Exception e) {  
        System.err.println(e.getMessage());  
    }  
}
```

**CATCH EXCEPTION**

```
}
```

```
method2() throws Exception{
```

```
    if(some error) {
```

```
        throw new Exception();  
    }  
}
```

**DECLARE EXCEPTION**

**THROW EXCEPTION**

```
}
```

## Declaring exceptions

- ▶ Every method must state the types of *checked* exceptions it might throw in the method header so that the caller of the method is informed of the exception.
  - ▶ System errors and runtime exceptions can happen to any code, therefore Java does not require explicit declaration of unchecked exceptions.
- ▶ `public void exceptionalMethod() throws IOException{`
- ▶ `throws`: the method might throw an exception. Can also throw multiple exceptions, separated by comma.

## Throwing exceptions

- ▶ If an error is detected, then the program can throw an exception.
  - ▶ e.g., you have asked for age and the user gave you a string. You can throw an `IllegalArgumentException`.
- ▶ `throw new IllegalArgumentException("Wrong format for age");`
  - ▶ The argument in the constructor is called the exception message. You can access it by invoking `getMessage()`.
- ▶ `throws` **FOR DECLARING AN EXCEPTION,**
- ▶ `throw` **TO THROW AN EXCEPTION**

## Catching exceptions

- ▶ An exception can be caught and handled in a try-catch block.

```
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 superclass type appears before a catch block for a subclass. E.g., `catch(Exception ex)` followed by `catch(RuntimeException ex)` won't compile.
- ▶ If a method declares a checked exception (e.g., `void p1() throws IOException`) and you invoke it, you have to enclose it in a try catch block or declare to throw the exception in the calling method (e.g., `try{ p1();} catch (IOException e){...}`).

## 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 throw 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.



```
/**
 * Illustrates try,catch, finally blocks
 * @author https://docs.oracle.com/javase/tutorial/essential/exceptions/putItTogether.html
 *
 */
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class ListOfNumbers {
    // Note: This class will not compile yet.

    private List<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers() {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }

    public void writeList() {
        PrintWriter out = null;

        try {
            System.out.println("Entering" + " try statement");

            out = new PrintWriter(new FileWriter("OutFile.txt"));
            for (int i = 0; i < SIZE; i++) {
                out.println("Value at: " + i + " = " + list.get(i));
            }
        } catch (IndexOutOfBoundsException e) {
            System.err.println("Caught IndexOutOfBoundsException: " + e.getMessage());
        } catch (IOException e) {
            System.err.println("Caught IOException: " + e.getMessage());
        } finally {
            if (out != null) {
                System.out.println("Closing PrintWriter");
                out.close();
            } else {
                System.out.println("PrintWriter not open");
            }
        }
    }
}
```

## Practice Time

- ▶ 1. Is there anything wrong with this exception handler?

```
try {  
  
} catch (Exception e) {  
  
} catch (ArithmeticException a) {  
  
}
```

## Answers

- ▶ 1. The ordering matters! The second handler can never be reached and the code won't compile.

## Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Memory Management
- ▶ Exceptions
- ▶ **Assertions**
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

## Pre and post conditions

- ▶ **Pre-condition:** Specification of what must be true for method to work properly, typically in terms of its parameters.
- ▶ **Post-condition:** Specification of what must be true at end of method if precondition was met before execution.

# Assertions

- ▶ **Assertion** is a statement in the Java programming language that enables you to test your assumptions about your program.
- ▶ Each assertion contains a boolean expression that you believe will be true when the assertion executes. If it is not true, the system will throw an error. By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors.
- ▶ By convention, preconditions on public methods are enforced by explicit checks that throw particular, specified exceptions. You can use them for testing parameters of non-public methods.
- ▶ `assert Expression1 [: Expression2];`
  - ▶ When the system runs the assertion, it evaluates the boolean `Expression1` and if it is `false` throws an `AssertionError` with no detail message. If the optional `Expression2` is included and assertion fails, it passes `Expression2` in the constructor of `AssertionError`.

## Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Memory Management
- ▶ Exceptions
- ▶ Assertions
- ▶ **Text I/O**
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

## I/O streams

- ▶ **Input stream**: a sequence of data into the program.
- ▶ **Output stream**: a sequence of data out of the program.
- ▶ 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 CD (!)
- ▶ Streams can support different kinds of data: bytes, principles, characters, objects, etc.



### 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`
- ▶ `File`: contains methods for obtaining file properties, renaming, and deleting files. Not for reading/writing!
- ▶ **CAUTION: DIRECTORY SEPARATOR IN WINDOWS IS \, WHICH IS SPECIAL CHARACTER IN JAVA. SHOULD BE “\” INSTEAD.**

## TEXT I/O

---

```
/**
 * Demonstrates File class and its operations.
 * @author https://liveexample.pearsoncmg.com/html/TestFileClass.html
 *
 */

import java.io.File;
import java.util.Date;

public class TestFile {
    public static void main(String[] args) {
        File file = new File("some.text");
        System.out.println("Does it exist? " + file.exists());
        System.out.println("The file has " + file.length() + " bytes");
        System.out.println("Can it be read? " + file.canRead());
        System.out.println("Can it be written? " + file.canWrite());
        System.out.println("Is it a directory? " + file.isDirectory());
        System.out.println("Is it a file? " + file.isFile());
        System.out.println("Is it absolute? " + file.isAbsolute());
        System.out.println("Is it hidden? " + file.isHidden());
        System.out.println("Absolute path is " + file.getAbsolutePath());
        System.out.println("Last modified on " + new Date(file.lastModified()));
    }
}
```

### Writing data to a text file

- ▶ `PrintWriter output = new PrintWriter(new File("filename"));`
- ▶ New file will be created. If already exists, discard.
- ▶ Invoking the constructor may throw an I/O Exception...
- ▶ `output.print` and `output.println` work with Strings, and primitives.
- ▶ Always close a stream!

## TEXT I/O

---

```
/**
 * Demonstrates how to write to text file.
 * @author https://liveexample.pearsoncmg.com/html/WriteData.html
 *
 */

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();
        }
    }
}
```

### Reading data from a text file

- ▶ `java.util.Scanner` reads Strings and primitives.
- ▶ Breaks input into tokens, demoted by whitespaces.
- ▶ To read from keyboard: `Scanner input = new Scanner(System.in);`
- ▶ To read from file: `Scanner input = new Scanner(new File("filename"));`
- ▶ Need to close stream as before.
- ▶ `hasNext()` tells us if there are more tokens in the stream. `next()` returns one token at a time.
  - ▶ Variations of `next` are `nextLine()`, `nextByte()`, `nextShort()`, etc.

## TEXT I/O

---

```
/**
 * Demonstrates how to read data from a text file.
 * @author https://liveexample.pearsoncmg.com/html/ReadData.html
 *
 */

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();
        }
    }
}
```

## Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Memory Management
- ▶ Exceptions
- ▶ Assertions
- ▶ Text I/O
- ▶ **Java GUIs**
- ▶ Graphics
- ▶ Events

## GUIs

- ▶ **AWT**: The Abstract Windowing Toolkit is found in the package `java.awt`
  - ▶ Heavyweight components.
  - ▶ Implemented with native code written for that particular computer.
  - ▶ The AWT library was written in six weeks!
- ▶ **Swing**: Java 1.2 extended AWT with the `javax.swing` package.
  - ▶ Lightweight components.
  - ▶ Written in Java.



## JFrame

- ▶ `javax.swing.JFrame` inherits from `java.awt.Frame`
- ▶ The outermost container in an application.
- ▶ To display a window in Java:
  - ▶ Create a class that extends `JFrame`.
  - ▶ Set the size.
  - ▶ Set the location.
  - ▶ Set it visible.

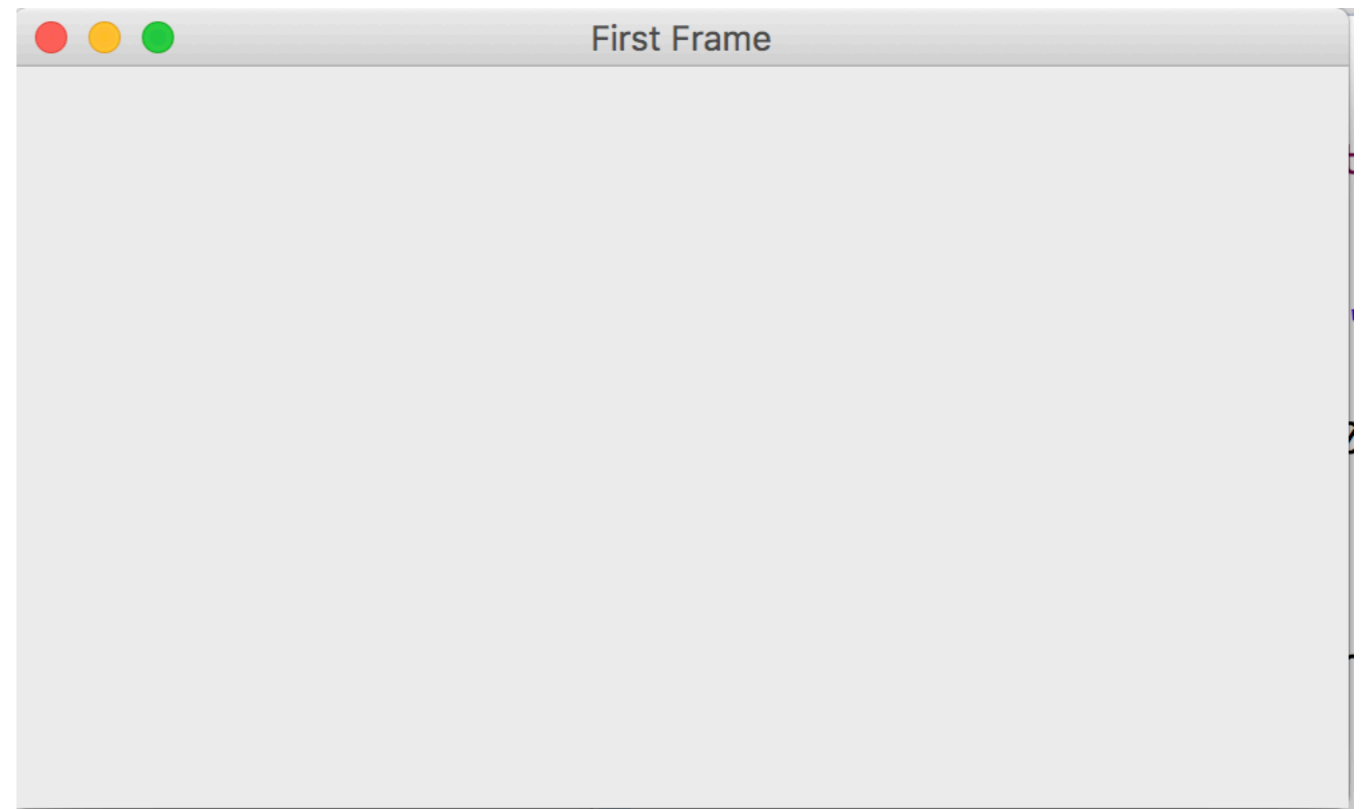
# JFrame

```
import javax.swing.JFrame;

public class MyFirstGUI extends JFrame {

    public MyFirstGUI() {
        super("First Frame");
        setSize(500, 300);
        setLocation(100, 100);
        setVisible(true);
    }

    public static void main(String[] args) {
        MyFirstGUI mfgui = new MyFirstGUI();
    }
}
```



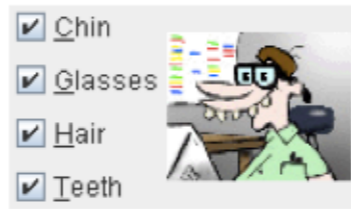
## Closing a GUI

- ▶ The default operation of the quit button is to set the visibility to false. The program does not terminate!
- ▶ `setDefaultCloseOperation` can be used to control this behavior.
- ▶ `mfgui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`
- ▶ More options (hide, do nothing, etc).

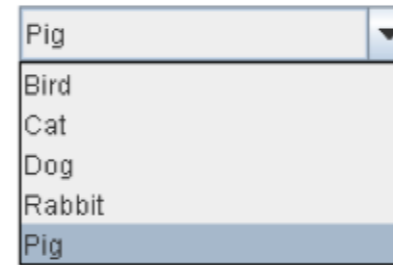
# Basic components



[JButton](#)



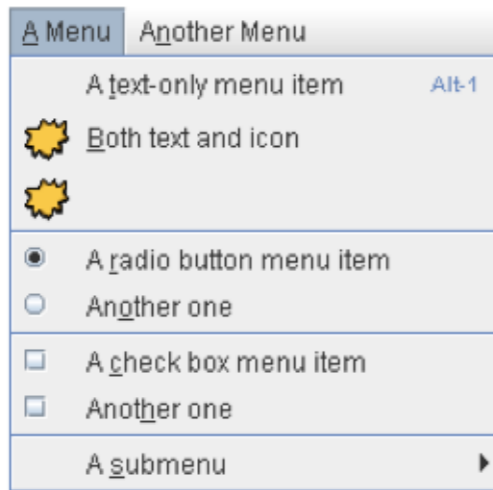
[JCheckBox](#)



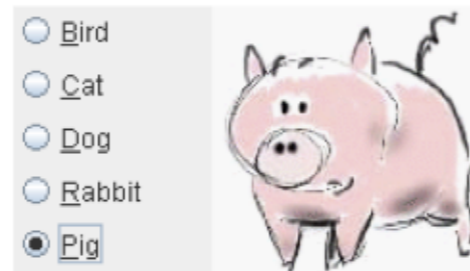
[JComboBox](#)



[JList](#)



[JMenu](#)



[JRadioButton](#)



[JSlider](#)



[JSpinner](#)

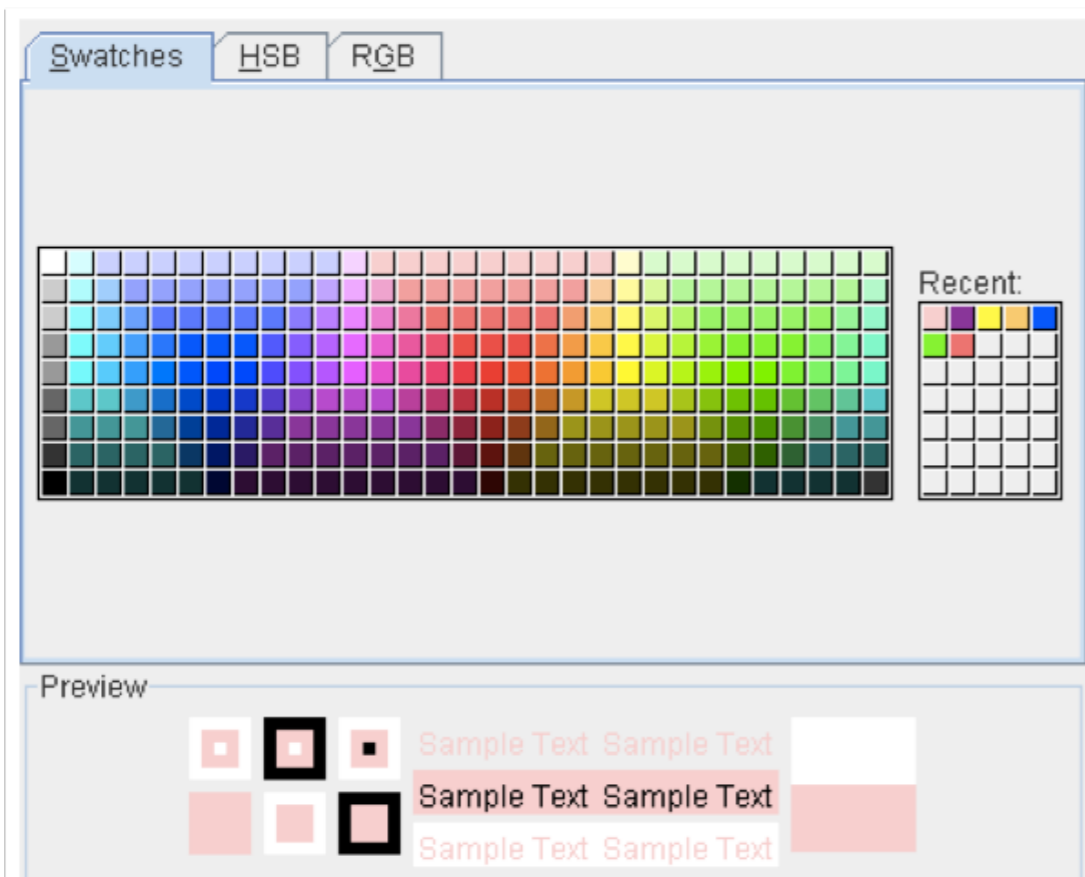


[JTextField](#)

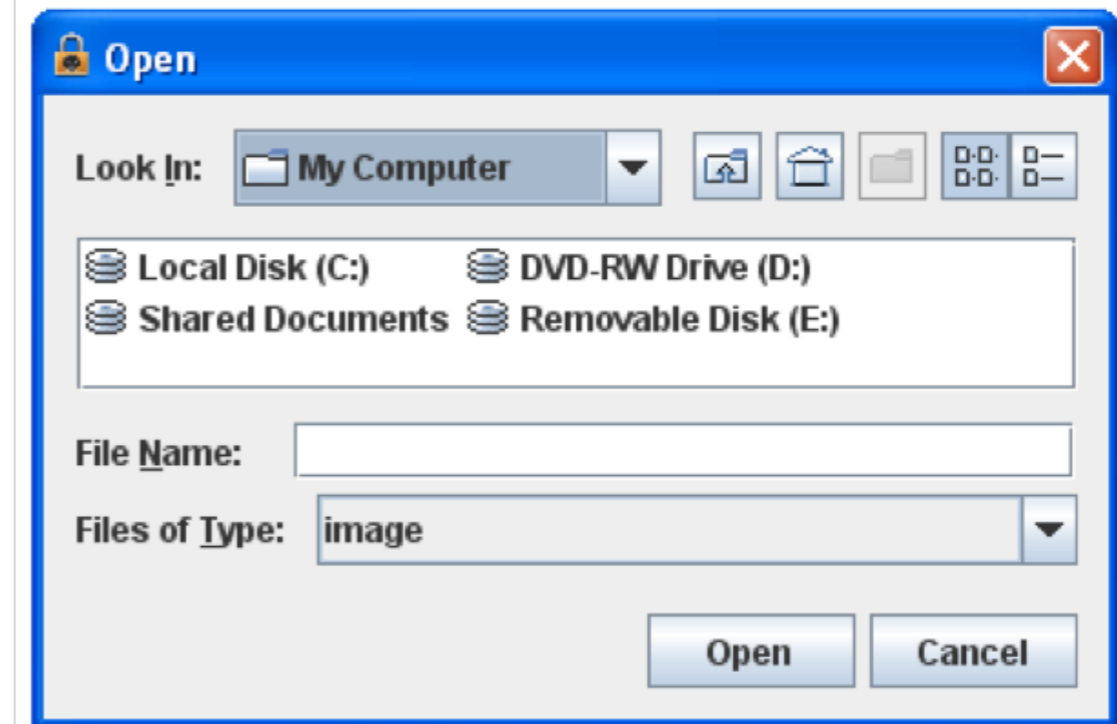


[JPasswordField](#)

# Interactive displays



[JColorChooser](#)



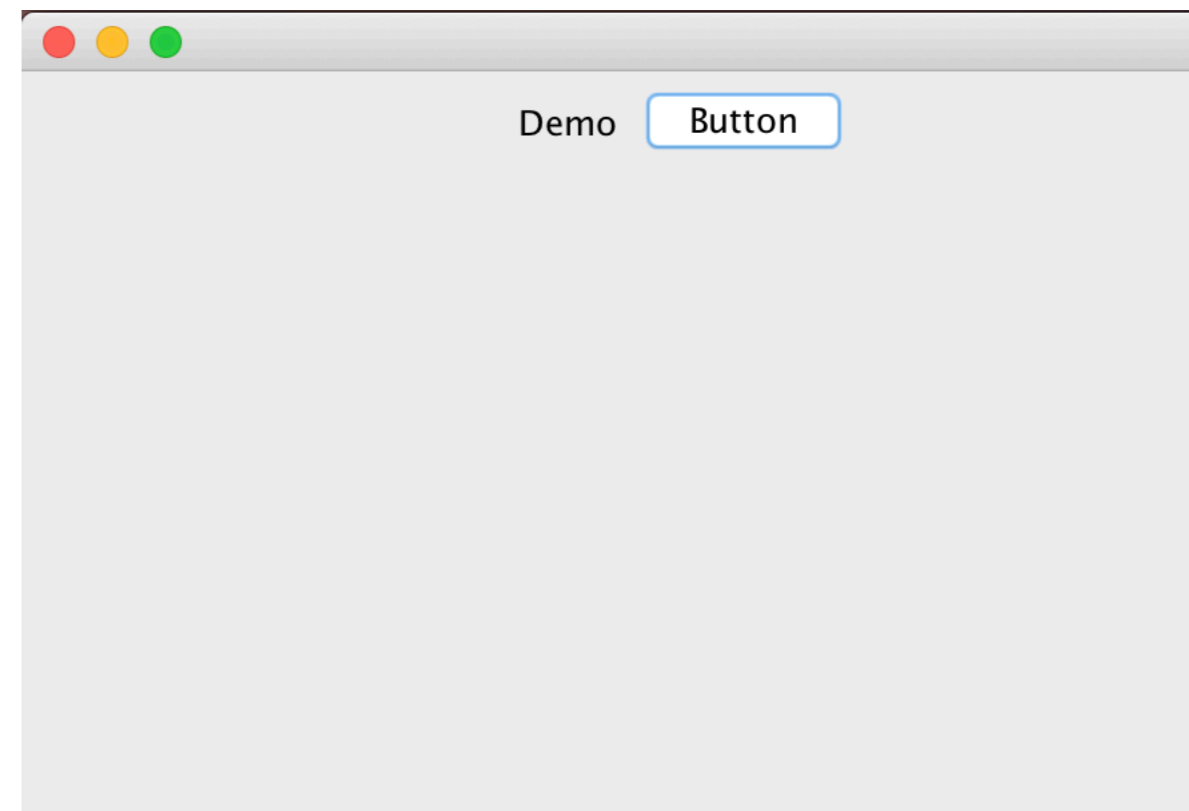
[JFileChooser](#)

## Adding JComponents to JFrame

```
import java.awt.Container;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class GUIDemo extends JFrame {
    public GUIDemo() {
        // Container cp = getContentPane();
        // cp.setLayout(new FlowLayout());
        // cp.add(new JLabel("Demo"));
        // cp.add(new JButton("Button"));
        JPanel mainPanel = new JPanel(new FlowLayout());
        mainPanel.add(new JLabel("Demo"));
        mainPanel.add(new JButton("Button"));
        setContentPane(mainPanel);
        setSize(500, 300);
        setVisible(true);
    }

    public static void main(String[] args) {
        GUIDemo gd = new GUIDemo();
        gd setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```



## Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Memory Management
- ▶ Exceptions
- ▶ Assertions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ **Graphics**
- ▶ Events

## Java Graphics

- ▶ Create arbitrary objects you want to draw:
  - ▶ `Rectangle2D.Double`, `Line.Double`, etc.
  - ▶ Constructors take `x`, `y` coordinates and dimensions, but don't actually draw items.
- ▶ All drawing takes place in `paint` method using a "graphics content".
- ▶ Triggered implicitly by uncovering window or explicitly by calling the `repaint` method.
  - ▶ Adds repaint event to draw queue and eventually draws it.



## Graphics context

- ▶ All drawing is done in `paint` method of component.
- ▶ `public void paint (Graphics g)`
- ▶ `g` is a graphics context provided by the system.
- ▶ “pen” that does the drawing.
- ▶ You call `repaint()` not `paint()`.
- ▶ Need to import classes from `java.awt.*`, `java.geom.*`, `javax.swing.*`
- ▶ See `MyGraphicsDemo`.

## General graphics applications

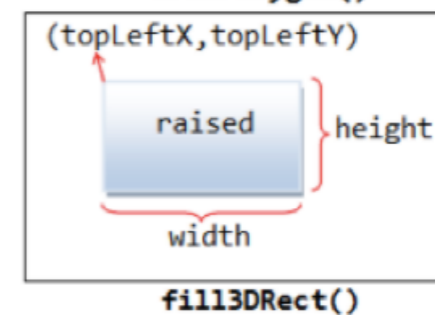
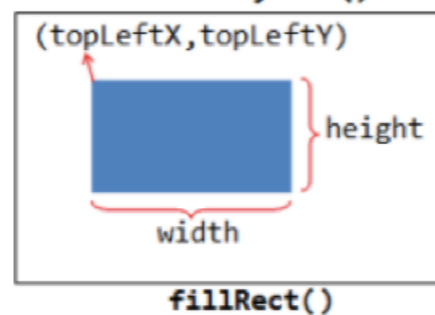
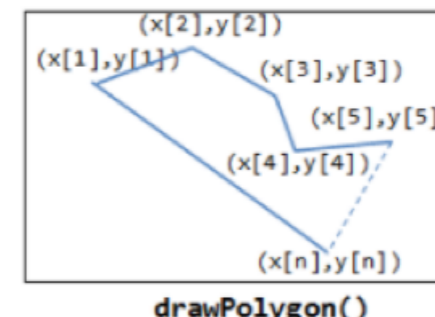
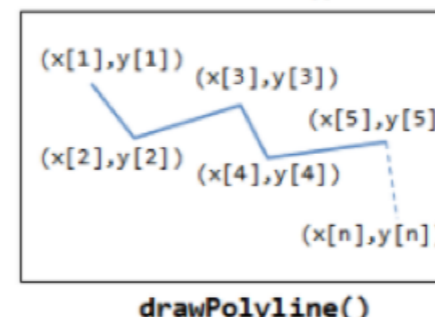
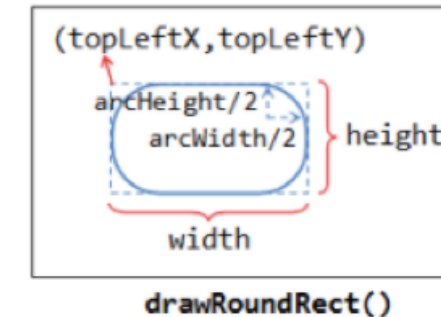
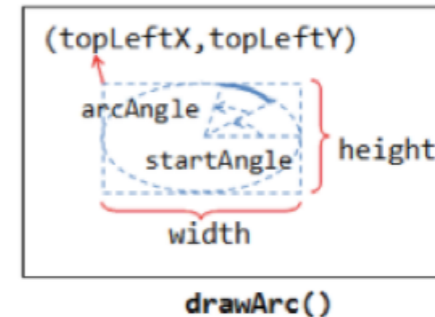
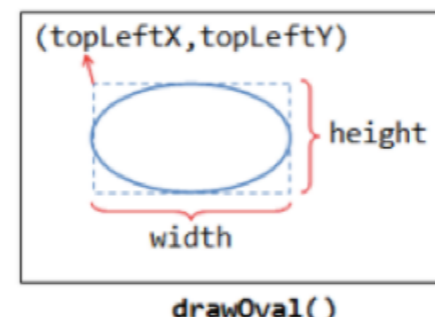
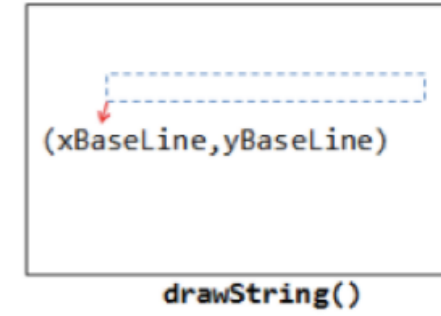
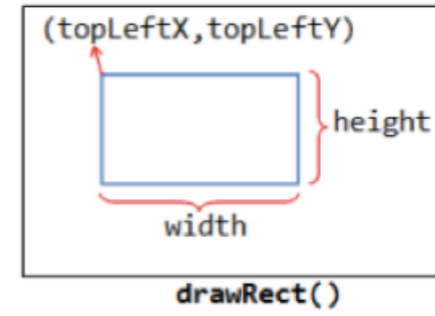
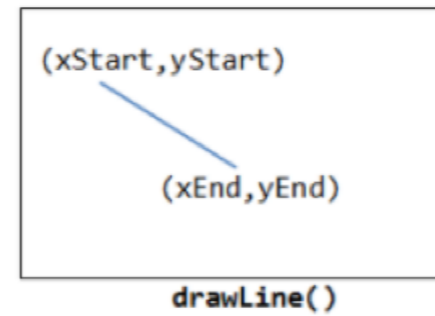
- ▶ Create an extension of component (JPanel or JFrame) and implement `paint` method in subclass.
- ▶ At start of `paint()` method cast `g` to `Graphics2D`.
- ▶ Call `repaint()` every time you want the component to be redrawn.

## Geometric objects

- ▶ Objects from classes `Rectangle2D.Double`, `Line2D.Double`, etc. from `java.awt.geom`
- ▶ Constructors take parameters `x`, `y`, `width`, `height` but don't draw object.
- ▶ `Rectangle2D.Double`
- ▶ `Ellipse2D.Double`
- ▶ `Arc2D.Double`
- ▶ etc.

# Drawing

- ▶ `myObj.setFrame(x, y, width, height)`: moves and sets size of component
- ▶ `g2.draw(myObj)`: gives outline
- ▶ `g2.fill(myObj)`: gives filled version
- ▶ `g2.drawString("a string", x, y)`: draws string



# java.awt.Color



## Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Memory Management
- ▶ Exceptions
- ▶ Assertions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

## Action listeners

- ▶ Define what should be done when a user performs certain operations.
  - ▶ e.g., clicks a button, chooses a menu item, presses Enter, etc.
- ▶ The application should implement the [ActionListener](#) interface.
- ▶ An instance of the application should be registered as a listener on one or more components.
- ▶ Implement the `actionPerformed` method.

```
public class MultiButtonApp implements ActionListener {
    ...
    //where initialization occurs:
    button1.addActionListener(this);
    button2.addActionListener(this);

    ...
    public void actionPerformed(ActionEvent e) {
        if(e.getSource() == button1){
            //do something
        }
    }
}
```

## Mouse listeners

- ▶ Define what should be done when a user enters a component, presses or releases one of the mouse buttons.
- ▶ The application should implement the [MouseListener](#) interface
  - ▶ Implement methods `mousePressed`, `mouseReleased`, `mouseEntered`, `mouseExited`, and `mouseClicked`.
- ▶ **Or** extend the [MouseListenerAdapter](#) class
  - ▶ Which has default implementations of all of them.

```
public class MouseEventDemo ... implements MouseListener {
    //where initialization occurs:
    //Register for mouse events on blankArea and the panel.
    blankArea.addMouseListener(this);
    addMouseListener(this);
    ...

    public void mousePressed(MouseEvent e) {
        saySomething("Mouse pressed; # of clicks: "
            + e.getClickCount(), e);
    }
}
```



## Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Memory Management
- ▶ Exceptions
- ▶ Assertions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

## Readings:

- ▶ Oracle's guides:
  - ▶ JavaDoc: <https://www.oracle.com/technetwork/articles/java/index-137868.html>
  - ▶ Exceptions: <https://docs.oracle.com/javase/tutorial/essential/exceptions/>
  - ▶ Assertions: <https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>
  - ▶ I/O: <https://docs.oracle.com/javase/tutorial/essential/io>
  - ▶ Writing Event Listeners: <https://docs.oracle.com/javase/tutorial/uiswing/events/index.html>
- ▶ Java Graphics: <https://github.com/pomonacs622023fa/Handouts/blob/master/graphics.md>
- ▶ Programming with GUIs: <https://github.com/pomonacs622023fa/Handouts/blob/main/JavaGUI.pdf>
- ▶ Swing/GUI Cheat Sheet: <https://github.com/pomonacs622023fa/Handouts/blob/master/swing.md>
- ▶ Recommended Textbook:
  - ▶ Chapter 1.2 (Page 107)

## Code

- ▶ [Lecture 4 code](#)