CS2030 Programming Methodology II

Lecture 1

Shengdong Zhao
Spring 2019

Acknowledge: slides are adapted from Henry Chia

Same Problem, Different Approach

- From a list of people, select all names with more than 5 characters, sort the result alphabetically, and change all names to uppercase.

```
result = [];
for i = 0; i < length(people); i++ {
    p = people[i];
    if length(p.name)) > 5 {
        addToList(result, toUpper(p.name));
    }
}
return sort(result);
```

Same Problem, Different Approach

- From a list of people, select all names with more than 5 characters, sort the result alphabetically, and change all names to uppercase.

```
select upper(name)
from people
where length(name) > 5
order by name
```

Same Problem, Different Approach

- From a list of people, select all names with more than 5 characters, sort the result alphabetically, and change all names to uppercase.

```
sort(
    filter(λs. length s > 5,
        map(λp. to_upper(name p),
            people)))
```

Same Problem, Different Approach

- From a list of people, select all names with more than 5 characters, sort the result alphabetically, and change all names to uppercase.

```
result = []
for p in people {
    if p.name.length > 5 {
        result.add(p.name.toUpper);
    }
}
return result.sort;
```

Programming Paradigms

- **Imperative (procedural)**
  - Specifies how computation proceeds using statements that change program state

```
result = [];
for i = 0; i < length(people); i++ {
   p = people[i];
   if length(p.name)) > 5 {
      addToList(result, toUpper(p.name));
   }
}
return sort(result);
```

- **Declarative**
  - Specifies what should be computed, rather than how to compute it

```
select upper(name)
from people
where length(name) > 5
order by name
```

- **Functional**
  - A form of declarative programming and treats computation like evaluating mathematical functions

```
sort(
   filter(λs. length s > 5,
      map(λp. to_upper(name p),
         people)))
```

- **Object-oriented**
  - Supports imperative programming but organizes programs as interacting objects, following the real-world

```
result = []
for p in people {
   if p.name.length > 5 {
      result.add(p.name.toUpper);
   }
}
return result.sort;
```

Objectives of the course

- By using the latest release of Java, we shall aim to
  - revise imperative programming using a strongly-typed language
  - focus on object oriented modeling and programming
  - introduce functional-style and declarative programming so as to simplify programming tasks

Refresher on common programming concepts

- Data (Memory)
  - Primitive data-type: numerical, character, Boolean

    1, 1.2, … 'c', '#', … true, false
  - Composite data-type:
    - Homogeneous: array (multi-dimensional) [1,2,3,4,5]
    - Heterogeneous: record (or structure) {1, "hello", 1.2, foo}
- Process (Mechanism)
  - Input and output
  - Primitive operations: arithmetic, relational, logical, ...

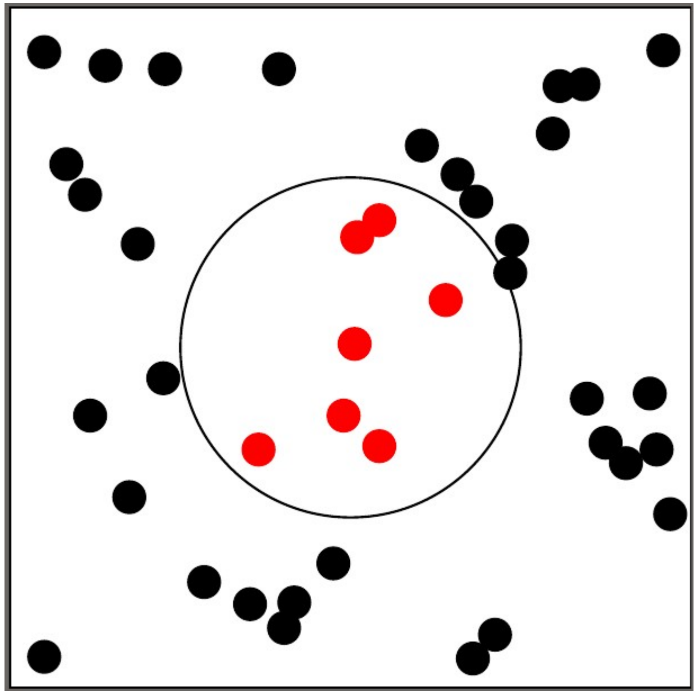
    +, -, *, /,… >=, >, … &&, ||, !
  - Control structures: sequence, selection, repetition

    if/else switch for(i=1, i<n, i++)
  - Modular programming: functions, procedures
  - Recursion

    fact(int n) { if (n == 0) **return** 1; else **return** n * fact(n - 1); }

Exercise: Disc Coverage Problem

- Given a set of points on the 2D Cartesian plane, find the number of points covering a unit disc (i.e. a circle of radius 1) centred at each point



Compilation vs. Interpretation

High level language:
C, C++, JavaScript, Java,

Compiler
(Compile **entire program**)

Interpreter
(Interpret **line by line**)

Compiler
(Compile)

Intermediate Language

Just in time Interpreter
(Interpret)

Execution of Machine Language

Java Compilation and Interpretation

- A class encompasses tasks common to a specific problem, e.g.

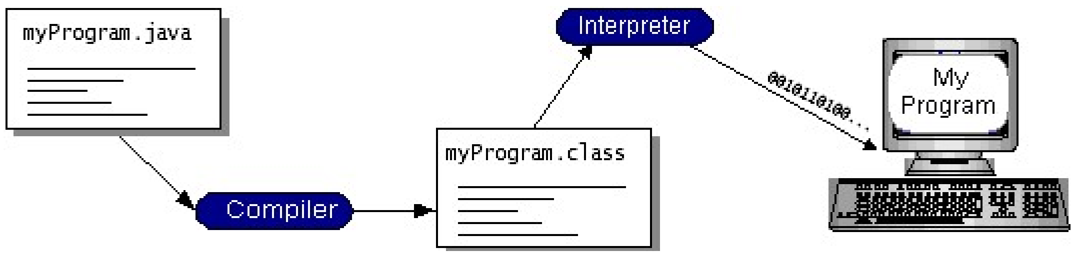
```
class DiscCoverage {
      public static void main(String[] args) {
      }
}
```

- To compile (assuming saved in DiscCoverage.java):

  $ javac DiscCoverage.java

- The above creates bytecode DiscCoverage.class which can be translated and executed on the java virtual machine using:

  $ java DiscCoverage



Input and Output

- Input/output via APIs (application programming interfaces):
- https://docs.oracle.com/javase/9/docs/api
- Import the necessary packages
  - Input: java.util.Scanner
  - Output: java.lang.System

  (java.lang.* imported by default)

```
import java.util.Scanner;
class DiscCoverage {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.println(scanner.next());
  }
}
```

Static vs Dynamic Typing

Dynamic (e.g. JavaScript)

```
var a;
var b = 5.0;
var c = "Hello";
b = "This?"; // ok
```

Static (e.g. Java)

```
int a;
double b = 5.0;
String c = "Hello";
b = "This?"; // error
```

- **Need to develop a sense of "type awareness" by maintaining type-consistency**
- **During compilation, incompatible typing throws off a compile-time error**

Static vs Dynamic Typing

```
import java.util.Scanner;
class DiscCoverage {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    double x;
    double y;
    x = scanner.nextDouble();
    y = scanner.nextDouble();
    System.out.println("(" + x + ", " + y + ")");
  }
}
```

- Another example of type sensitivity: + operator

https://docs.oracle.com/javase/specs/jls/se9/html/jls-15.html#jls-15.18.1

Input via File Re-direction

```java
import java.util.Scanner;
class DiscCoverage {
  public static void main(String[] args) {
    Scanner scanner;
    int numOfPoints;
    scanner = new Scanner(System.in);
    numOfPoints = scanner.nextInt();
    for (int i = 1; i <= numOfPoints; i++) {
      double x = scanner.nextDouble();
      double y = scanner.nextDouble();
      System.out.println("Point #" + i +
        ": (" + x + ", " + y + ")");
    }
  }
}
```

- Read input from data.in using the following command:

```
$ java DiscCoverage < data.in
```

Composite Data - Arrays

```
import java.util.Scanner;
class DiscCoverage {
  public static void main(String[] args) {
    Scanner scanner;
    double[][] points;
    scanner = new Scanner(System.in);
    points = new double[scanner.nextInt()][2];
    for (int i = 0; i < points.length; i++) {
      points[i][0] = scanner.nextDouble();
      points[i][1] = scanner.nextDouble();
      System.out.println("Point #" + (i + 1) + ": (" +
        points[i][0] + ", " +
        points[i][1] + ")");
    }
  }
}
```

- Number of elements defined in the array is given by *length*

Modularity

- Taking a complex program and breaking it up into dedicated sub-tasks to be solved
- The **main** method (object-oriented equivalent of function/procedure) describes the solution in terms of higher-level *abstractions*

```
import java.util.Scanner;
class DiscCoverage {
  public static void main(String[] args) {
    double[][] points;
    points = readPoints();
    printPoints(points);
  }
```

- Abstractions can then be solved individually and incrementally

Modularity

how many rows want to put in

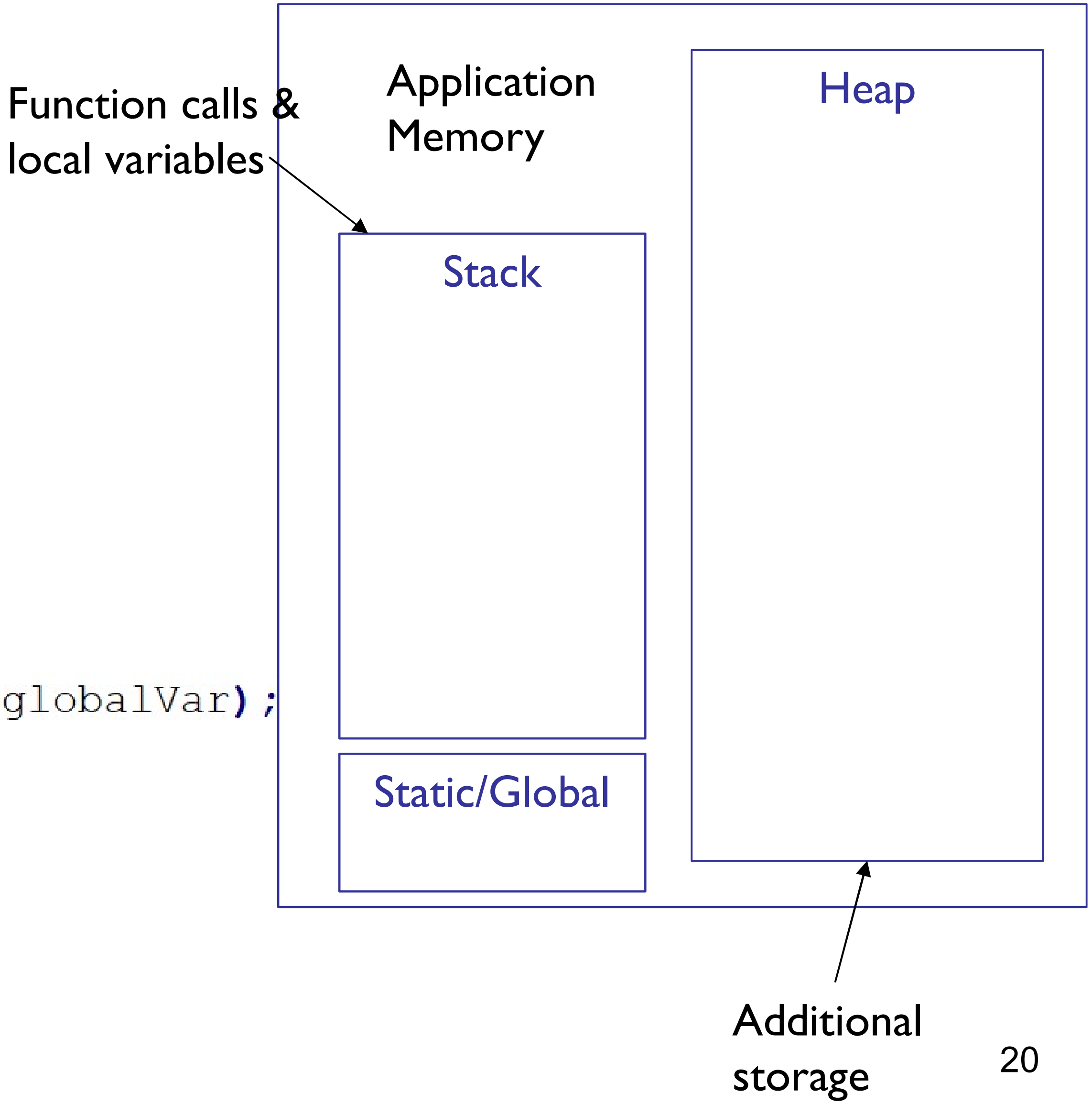
dimensional of array

for point in points, but need declare

```
  static double[][] readPoints() {
    Scanner scanner;
    double[][] points;
    scanner = new Scanner(System.in);
    points = new double[scanner.nextInt()][2];
    for (double[] point : points) {
      point[0] = scanner.nextDouble();
      point[1] = scanner.nextDouble();
    }
    return points;
  }
  static void printPoints(double[][] points) {
    int i = 0;
    for (double[] point : points) {
      System.out.println("Point #" + (i + 1) + ": (" +
        point[0] + ", " + point[1] + ")");
      i++;
    }
  }
}
```

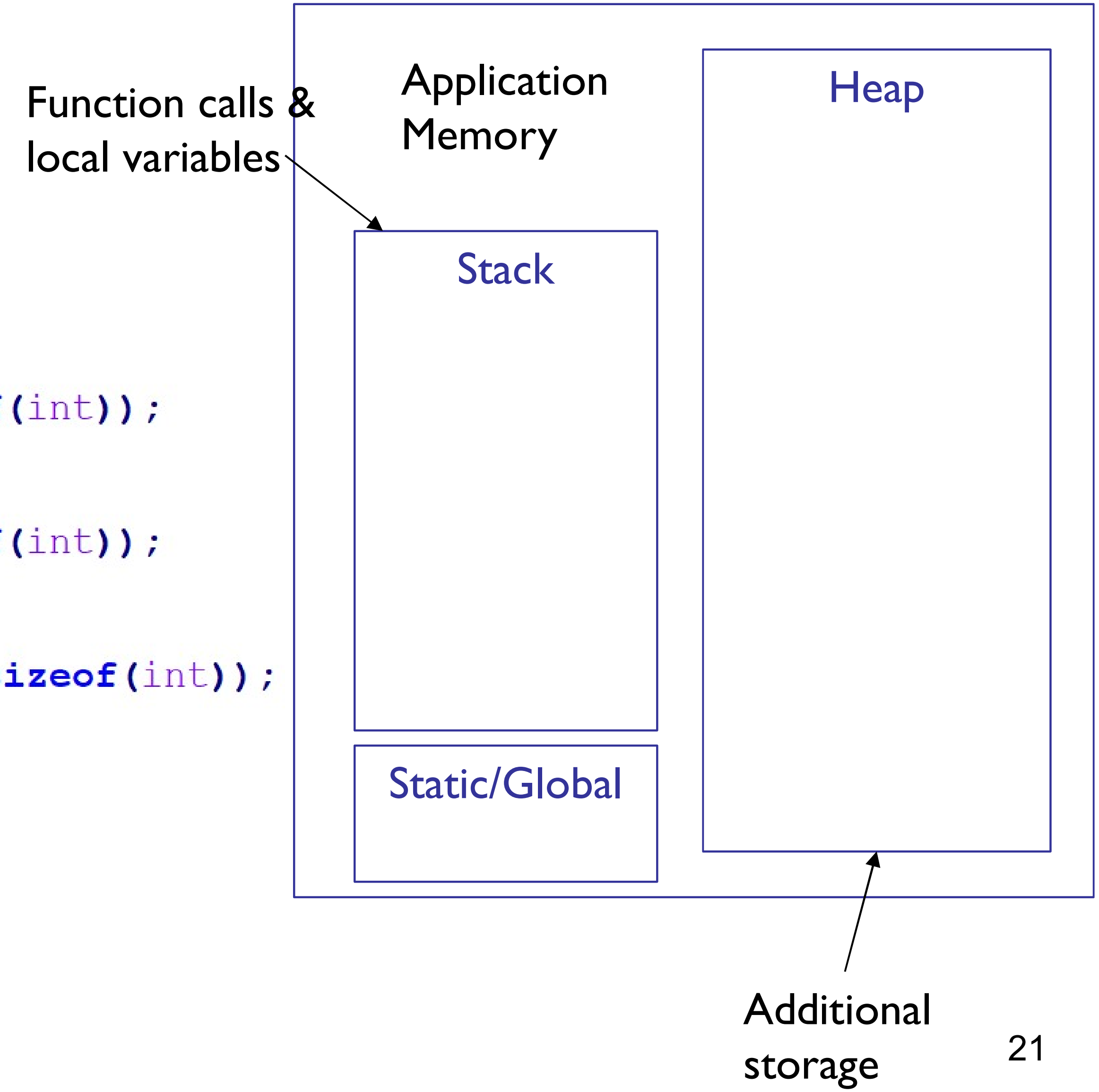
Mental Modeling (Stack)

```c
#include <stdio.h>
int globalVar;

int B(int x){
    return x;
}
int A(int x, int y){
    int z = B(x)+y;
    return z;
}
int main(){
    int a = 4, b=8;
    globalVar = A(a,b);
    printf("output = %d", globalVar);
}
```



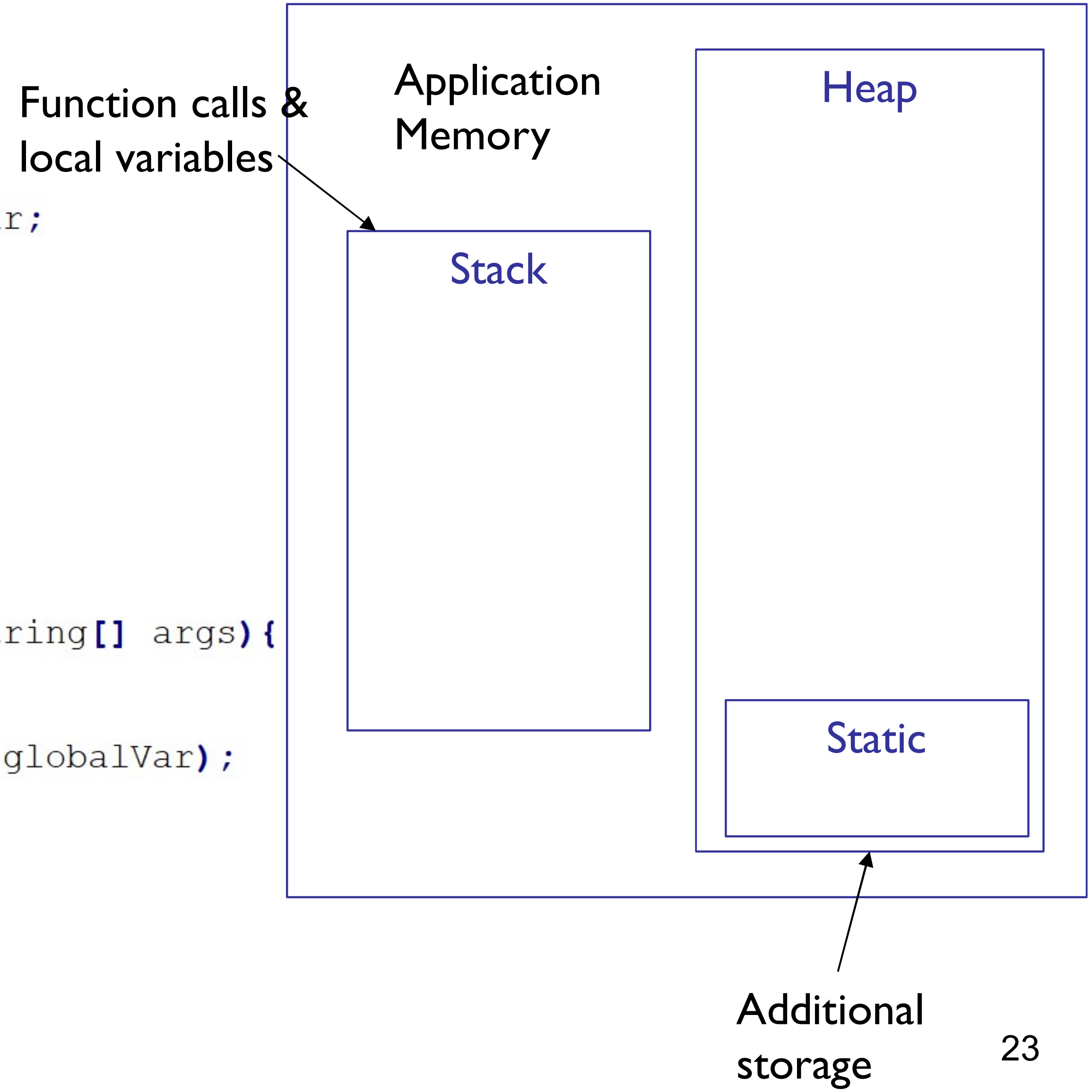
Mental Modeling (Heap - C)

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int a;
    int *p;
    p = (int*) malloc(sizeof(int));
    *p = 10;

    p = (int*) malloc(sizeof(int));
    *p = 20;

    p = (int*) malloc(20 * sizeof(int));
}
```



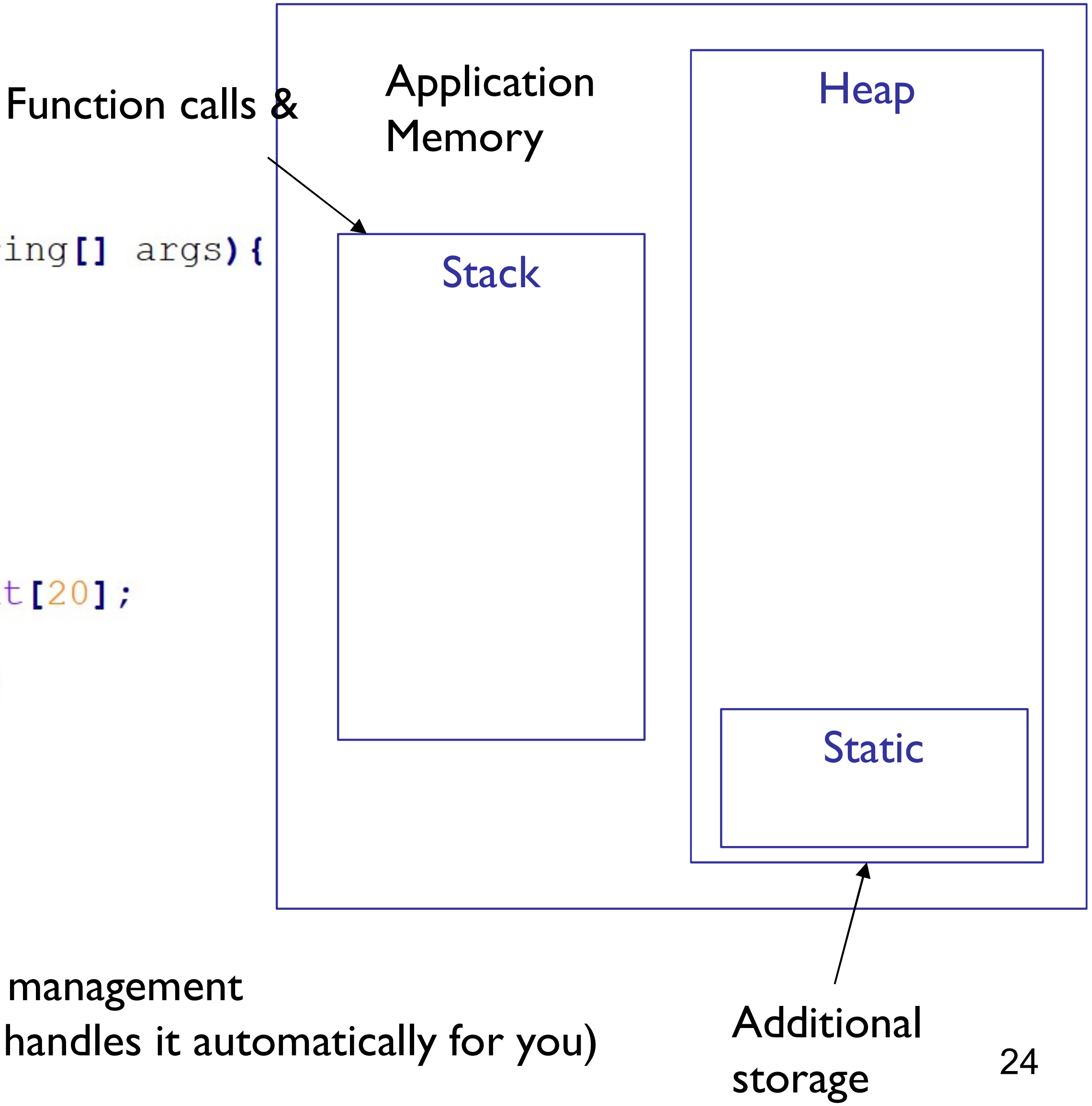
Mental Modeling (Java)

```
class MyClass {

    public static int globalVar;

    int B(int x){
        return x;
    }
    int A(int x, int y){
        int z = B(x)+y;
        return z;
    }

    public static void main(String[] args){
        int a = 4, b=8;
        globalVar = A(a,b);
        printf("output = %d", globalVar);
    }
}
```



Mental Modeling (Java)

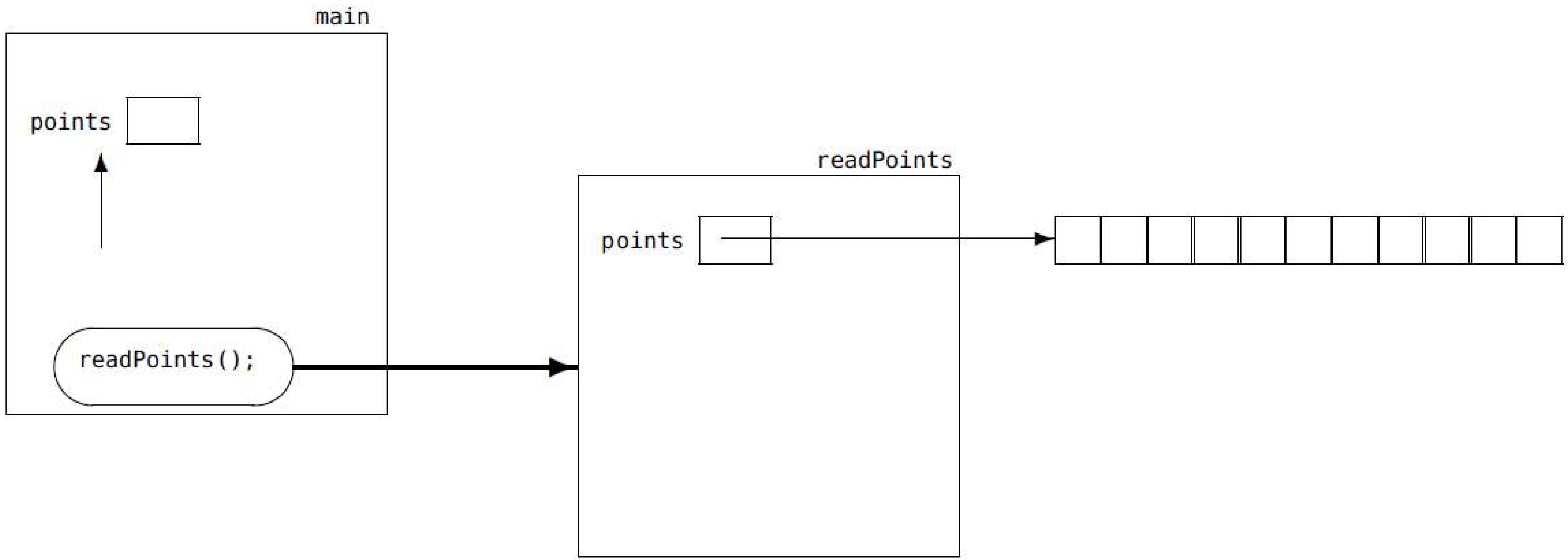
```
class MyClass {

    public static void main(String[] args){
        //Approach in C++
        //int *p;
        //p = new int[20];
        //delete[] p;
        //p = new int[30];

        //Approach in Java
        int[] intArray = new int[20];

        intArray = new int[30];
    }
}
```



No more pointer, de-referencing, management
No need to remember delete (it handles it automatically for you)

Mental Modeling

- Establish a mental model of program execution that is **correct**, **consistent** and **complete**
- Consider modeling the following statement:
  points = readPoints()



Mental Modeling

- Method readPoints with return type double [ ][ ] returns, the reference to the array is assigned to points in main

- While **stack** memory allocated for the readPoint method is flushed (together with the local variable point) upon return, the **heap** memory associated with the array remains intact

Revisit Static vs Dynamic Typing

Dynamic (e.g. JavaScript)

```
var a;
var b = 5.0;
var c = "Hello";
b = "This?"; // ok
```

Static (e.g. Java)

```
int a;
double b = 5.0;
String c = "Hello";
b = "This?"; // error
```

- **Remember we said that static typing languages are usually compiled while dynamic typing languages are usually interpreted, do you understand why now?**

Imperative Solution for Disc Coverage

```java
/**
* Determines if <code>point</code> is contained within the unit
* disc centred at <code>centre</code>.
*
* @param centre is the centre of the unit disc
* @param point is the other point
* @return true if <code>point</code> is contained within unit
* disc centred at <code>centre</code>; false otherwise
*/
static boolean isInside(double[] centre, double [] point) {
```

Imperative Solution for Disc Coverage

```
/**
* Determines the number of points within the <code>points</code>
* array that is covered by a unit disc centred at
<code>centre</code>
*
* @param centre is the centre of the unit disc
* @param points is the array of points
* @return the number of points covered
*/
static int discCover(double[] centre, double[][] points) {
```

Imperative Solution for Disc Coverage

```
/**
* Outputs the unit disc coverages centred at each point.
*
* @param points list of points
*/
  static void printCoverage(double[][] points) {
    for (double[] point : points) {
      int numOfPoints = discCover(point, points);
      System.out.println("Disc centred at (" +
        point[0] + ", " + point[1] +
        ") contains " + numOfPoints + " points.");
    }
  }
  public static void main(String[] args) {
   double[][] points;
   points = readPoints();
   printCoverage(points);
  }
}
```

Modeling an Object-Oriented (OO) Solution

- An object-oriented model based on interacting objects:
  - What are the different types of object in the problem?
    - * Circle (or unit disc) * Point
  - A circle has a point as its centre and a radius; these are **attributes / properties / fields** of the circle
  - Likewise a point has two double attributes representing the x- and y-coordinates of the point
  - To determine if a circle contains a point,
    - the circle takes a point to check for containment; this is a method (or behaviour)
    - the circle's centre (i.e. a point) needs a method to check its distance with respect to another point

OOP Principle #1: Abstraction

- Establish an abstraction level relevant to the task at hand and ignore lower level details
  - Data abstraction: abstract away low level data items
  - Functional abstraction: abstract away control flow details

```
public class Circle {
  private Point centre;
  private double radius;
  public Circle(Point centre) {
    this.centre = centre;
    this.radius = 1.0;
  }
  public Circle(Point centre, double radius) {
    this.centre = centre;
    this.radius = radius;
  }
  public boolean contains(Point point) {
    return centre.distance(point) <= radius;
  }
}
```

OOP Principle #1: Abstraction

```java
public class Point {
  private double x;
  private double y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public double distance(Point otherpoint) {
    double dispX = this.x - otherpoint.x;
    double dispY = this.y - otherpoint.y;
    return Math.sqrt(dispX * dispX + dispY * dispY);
  }
  @Override
  public String toString() {
    return "(" + this.x + ", " + this.y + ")";
  }
}
```

- How should the Main driver class be adapted?

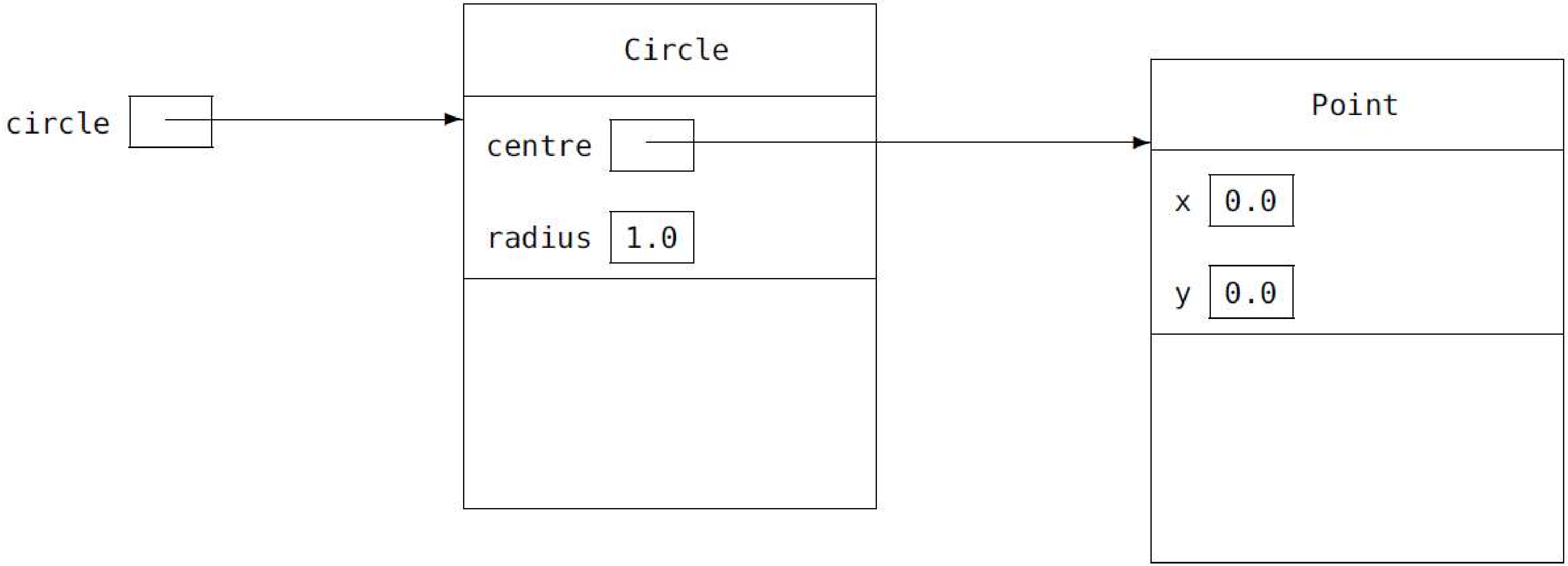
OOP Principle #2: Encapsulation

- Abstraction barrier
  - Separation between implementer and client
- Having established a particular high-level abstraction,
  - Implementer defines the data/functional abstractions using lower-level data items and control flow
  - Client uses the high-level data-type and methods
- Encapsulation
  - To protect implementation against inadvertent use
  - Packaging data and related behaviour together into a self-contained unit
  - Hiding information/data from the client, restricting access using methods/interfaces

Object-Oriented Mental Model

- Extending our mental model to include objects
- Example, when instantiating a Circle object

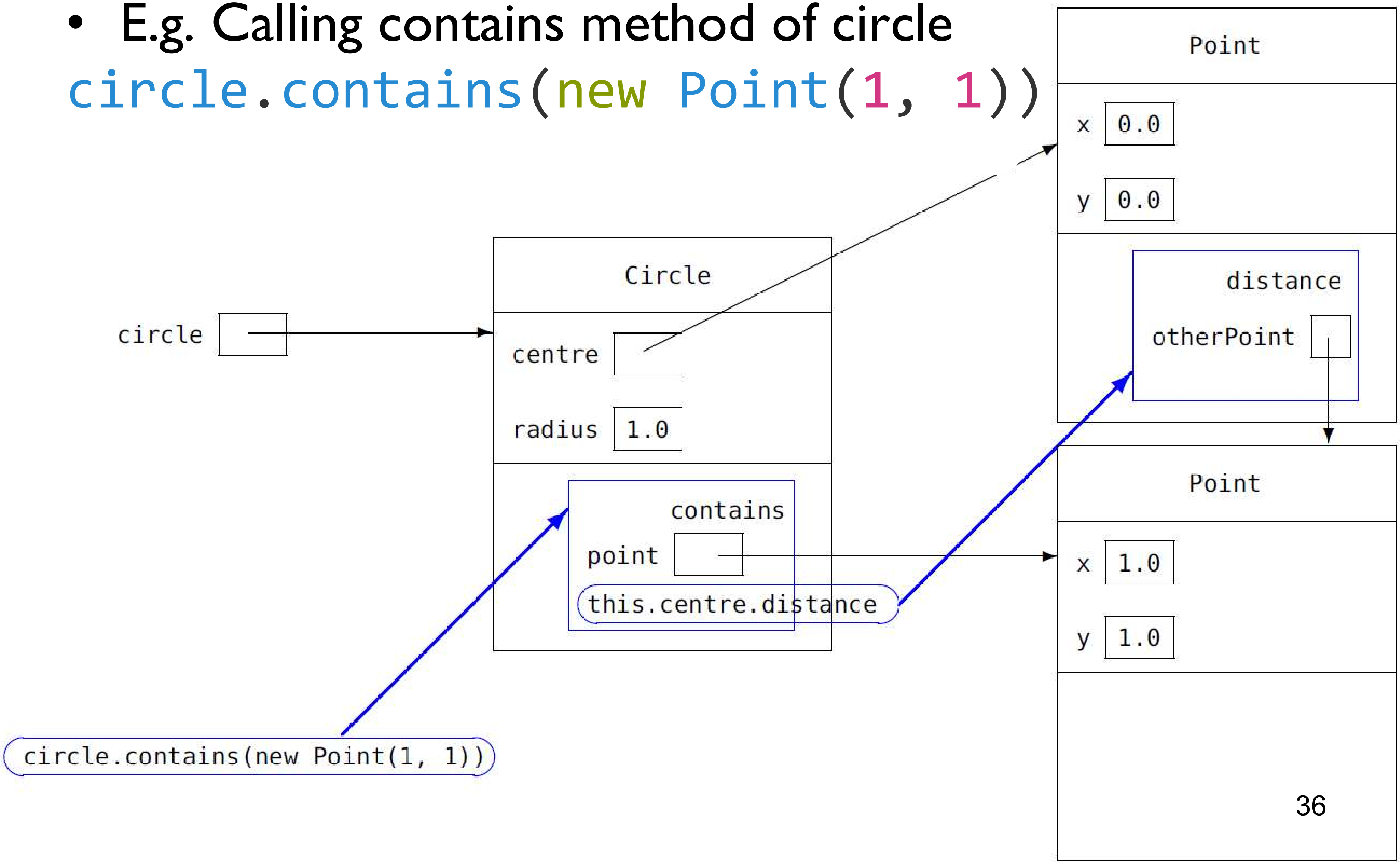
```
Circle circle = new Circle(new Point(0, 0), 1);
```



Object-Oriented Mental Model

- E.g. Calling contains method of circle

```
circle.contains(new Point(1, 1))
```



Lecture Summary

- Appreciate the different programming paradigms
- Appreciate java compilation and interpretation
- Develop a sense of type awareness when developing programs
- Able to employ object-oriented modeling to convert an imperative solution to OO
- Understand the OO principles of abstraction and encapsulation
- Develop and apply a mental model of program execution