CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP110

Lecture

Rev 5-5-22

DR JEFF SOFTWARE
INDIE APP DEVELOPER

Intro to Algorithms & Programming

LECTURES

Part 2

Dr Jeff Drobman

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP110

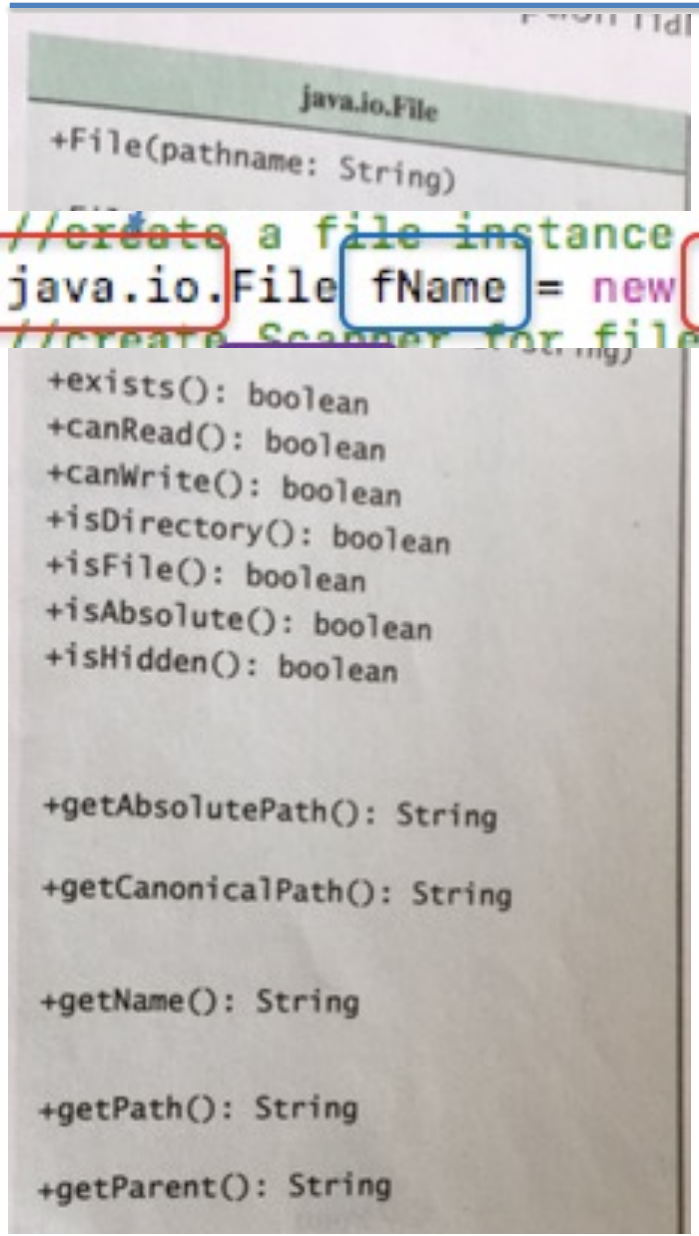
Slide Index

DR JEFF SOFTWARE
INDIE APP DEVELOPER

I/O

- File I/O

File Methods




```
//create a file instance
java.io.File fName = new java.io.File("myFile.txt");
//create Scanner for file
```

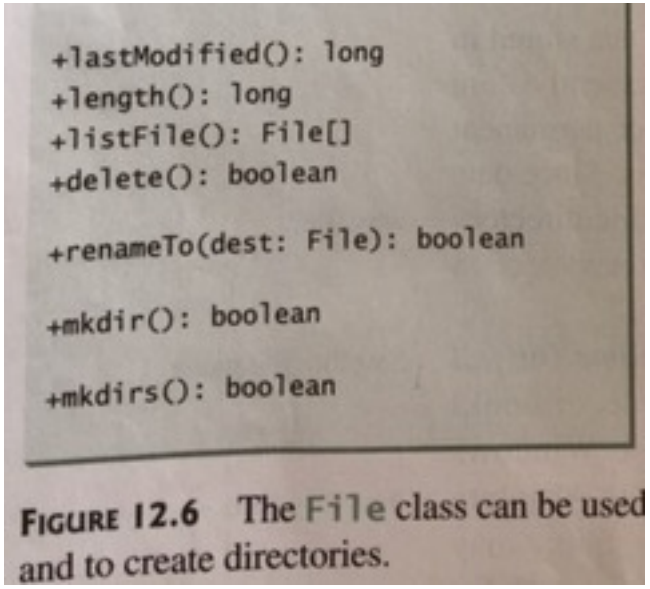


FIGURE 12.6 The File class can be used and to create directories.

File Methods Examples


```
public class File {
   static final boolean $DEBUG = true;
//main method
   public static void main(String[] args) {
      //debug
      if ($DEBUG) System.out.println("debug: starting code");
      //code starts here
      java.io.File fname = new java.io.File("homs.txt");
      System.out.println("Does file exist: " + fname.exists());
      System.out.println("File size in byte: " + fname.length());
      System.out.println("Can file be read: " + fname.canRead());
   } //end main method
} //end class
```

```
----jGRASP exec: java File
debug: starting code
Does file exist: true
File size in byte: 277
Can file be read: true

----jGRASP: operation complete.
```

File Methods Examples


```
// imports
import javax.swing.*;
import java.util.*;
import java.io.*;
// **main class**
public class File {
   static final boolean $DEBUG = true;
//main method
   public static void main(String[] args) {
      //debug
      if ($DEBUG) System.out.println("debug: starting code");
      //"is" methods
      java.io.File fname = new java.io.File("homs.txt");
      System.out.println("Does file exist: " + fname.exists());
      System.out.println("File size in byte: " + fname.length());
      System.out.println("Can file be read: " + fname.canRead());
      System.out.println("Can file be written: " + fname.canWrite());
      System.out.println("Is name a dir: " + fname.isDirectory());
      System.out.println("Is name a file: " + fname.isFile());
      System.out.println("Is path absolute: " + fname.isAbsolute());
      System.out.println("Is file hidden: " + fname.isHidden());
      //"get" methods
      System.out.println("Absolute path is: " + fname.getAbsolutePath());
      //date methods
      System.out.println("Date last modified: " + new Date(fname.lastModified
   } //end main method
} //end class
```

File Methods Examples


```
 ----jGRASP exec: java File
debug: starting code
Does file exist: true
File size in byte: 277
Can file be read: true
Can file be written: true
Is name a dir: false
Is name a file: true
Is path absolute: false
Is file hidden: false
Absolute path is: /Users/jhdphd/Documents/Classroom+ITT+CSUN/CSUN/J
Date last modified: Mon Mar 27 20:40:17 PDT 2017

 ----jGRASP: operation complete.
```

Text File Input

Listing 12.15
pp. 478-9

```
import java.util.*;
import javax.swing.*;
import java.io.*;

public class CS110 {
public static void main(String[], args) {

    //**text file input
    //create a file instance
    java.io.File fName = new java.io.File("myFile.txt");
    //create Scanner for file
    Scanner input = new Scanner(fName);
    //read data from file
    while (input.hasNext()) {
        String word = input.next();
        String pron = input.next();
        //alt:  String line = input.nextLine();
    }
    //when done (now) close file
    fName.close();
```

throws Exception?

input

```
public class cs110TextFile {
public static void main(String[] args) throws FileNotFoundException {
    //test I/O
```

Text File Read into Arrays


```
//INput method -- text file input
33    public static void readFile(String xName, String[] wArr,
String[] pArr) throws Exception {
34      //create a file instance
35      File fName = new File(xName);
36    //create Scanner for file
37      Scanner input = new Scanner(fName);
38    //read data from file
39      int i = 0, siz = wArr.length;
40      while (input.hasNext()) {
41          String word = input.next();
42          String pron = input.nextLine(); //clear cr/lf
43          wArr[i] = word;
44          pArr[i] = pron;
45          i++;
46          if ($DEBUG) System.out.println(word + pron);
47          if (i >= siz) { //don't overflow array
48              if ($DEBUG) System.out.println("array overflow");
49              break;}
```

ZyBook Ch 7

7.5 File input and output

Figure 7.5.1: Input from a file.

- Just use "File"

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

public class FileReadNums {
   public static void main (String[] args) throws IOException {
      FileInputStream fileByteStream = null; // File input stream
      Scanner inFS = null;                   // Scanner object
      int fileNum1;                          // Data value from file
      int fileNum2;                          // Data value from file

      // Try to open file
      System.out.println("Opening file myfile.txt.");
      fileByteStream = new FileInputStream("myfile.txt");
      inFS = new Scanner(fileByteStream);

      // File is open and valid if we got this far (otherwise exception thrown)
      // myfile.txt should contain two integers, else problems
      System.out.println("Reading two integers.");
      fileNum1 = inFS.nextInt();
      fileNum2 = inFS.nextInt();

      // Output values read from file
      System.out.println("num1: " + fileNum1);
      System.out.println("num2: " + fileNum2);
      System.out.println("num1+num2: " + (fileNum1 + fileNum2));

      // Done with file, so try to close it
      System.out.println("Closing file myfile.txt.");
      fileByteStream.close(); // close() may throw IOException if fails
   }
}
```

ZyBook Ch 7

7.5 File input and output

Figure 7.5.3: Reading a varying amount of data from a file.

- Just use "File"

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

public class FileReadVaryingAmount {
   public static void main(String[] args) throws IOException {
      FileInputStream fileByteStream = null; // File input stream
      Scanner inFS = null;                   // Scanner object
      int fileNum;                           // Data value from file

      // Try to open file
      System.out.println("Opening file myfile.txt.");
      fileByteStream = new FileInputStream("myfile.txt");
      inFS = new Scanner(fileByteStream);

      // File is open and valid if we got this far (otherwise exception thrown)
      System.out.println("Reading and printing numbers.");

      while (inFS.hasNextInt()) {
         fileNum = inFS.nextInt();
         System.out.println("num: " + fileNum);
      }

      // Done with file, so try to close it
      System.out.println("Closing file myfile.txt.");
      fileByteStream.close(); // close() may throw IOException if fails
   }
}
```

ZyBook Ch 7

7.5 File input and output

Figure 7.5.4: Sample code for writing to a file.

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

public class FileWriteSample {
   public static void main(String[] args) throws IOException {
      FileOutputStream fileByteStream = null; // File output stream
      PrintWriter outFS = null;               // Output stream

      // Try to open file
      fileByteStream = new FileOutputStream("myoutfile.txt");
      outFS = new PrintWriter(fileByteStream);

      // File is open and valid if we got this far (otherwise exception thrown)
      // Can now write to file
      outFS.println("Hello");
      outFS.println("1 2 3");
      outFS.flush();

      // Done with file, so try to close it
      fileByteStream.close(); // close() may throw IOException if fails
   }
}
```

Text File Output

Listing 12.16
pp. 480-1

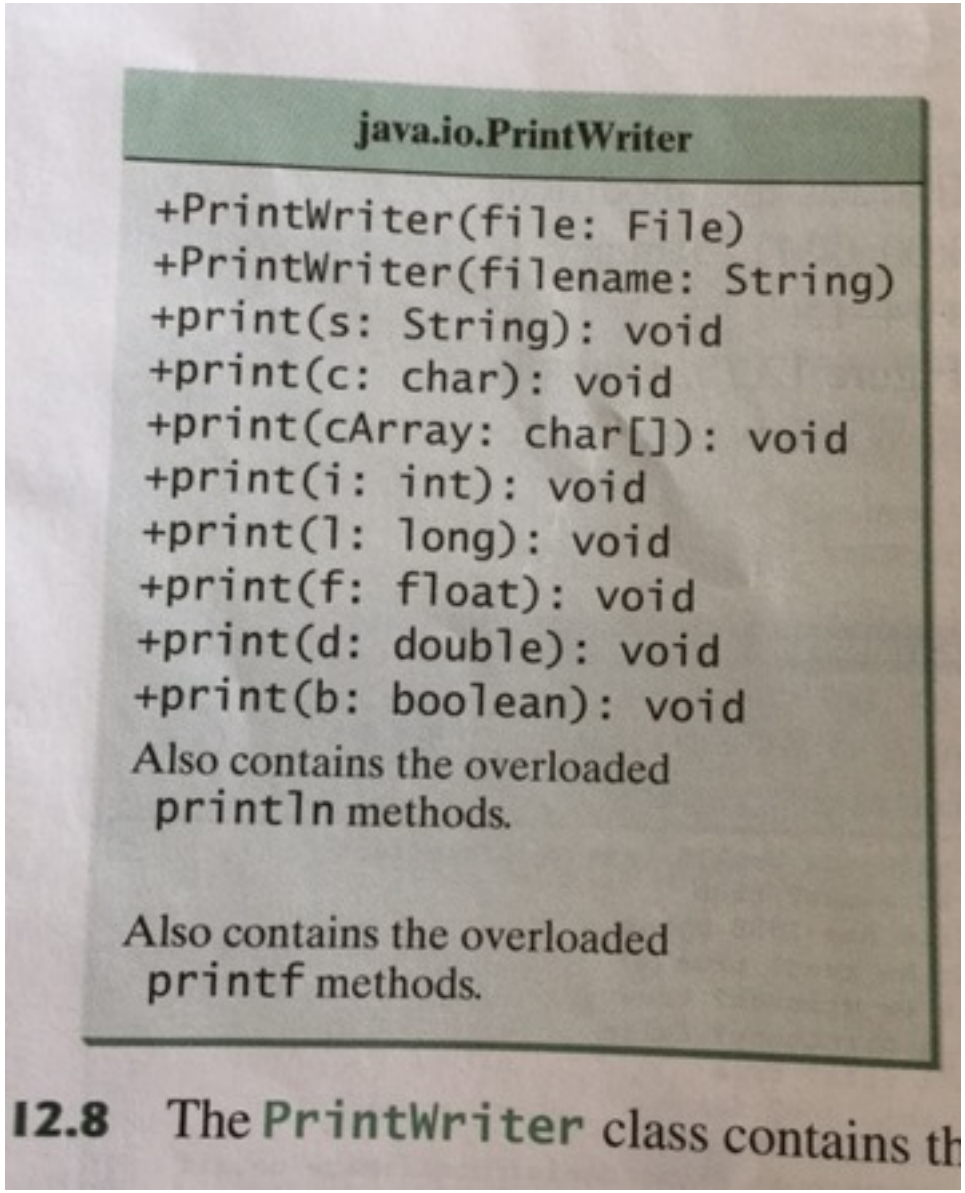
```
import java.io.*;

public class cs110TextFile {
public static void main(String[] args) throws FileNotFoundException {
    //test I/O

  //create OUTput (method later)
 File fName2 = new File("Documents/OutFile.txt");
 PrintWriter output = new PrintWriter(fName2);
 //create array for output data
 String[] outArr = new String[100];
 //write data TO file
 for (int i=0; i< outArr.length; i++){
 output.println(outArr[i]);
 }
```

- *PrintWriter*

in Windows:
("c: \\users\\jeff\\folder\\fName")

- *backslashes*

PrintWriter Methods



java.io.PrintWriter

```
+PrintWriter(file: File)
+PrintWriter(filename: String)
+print(s: String): void
+print(c: char): void
+print(cArray: char[]): void
+print(i: int): void
+print(l: long): void
+print(f: float): void
+print(d: double): void
+print(b: boolean): void
```

Also contains the overloaded `println` methods.

Also contains the overloaded `printf` methods.

12.8 The `PrintWriter` class contains th

- UML notation

Java

Exceptions

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP110

Exception Handling

Ch 12

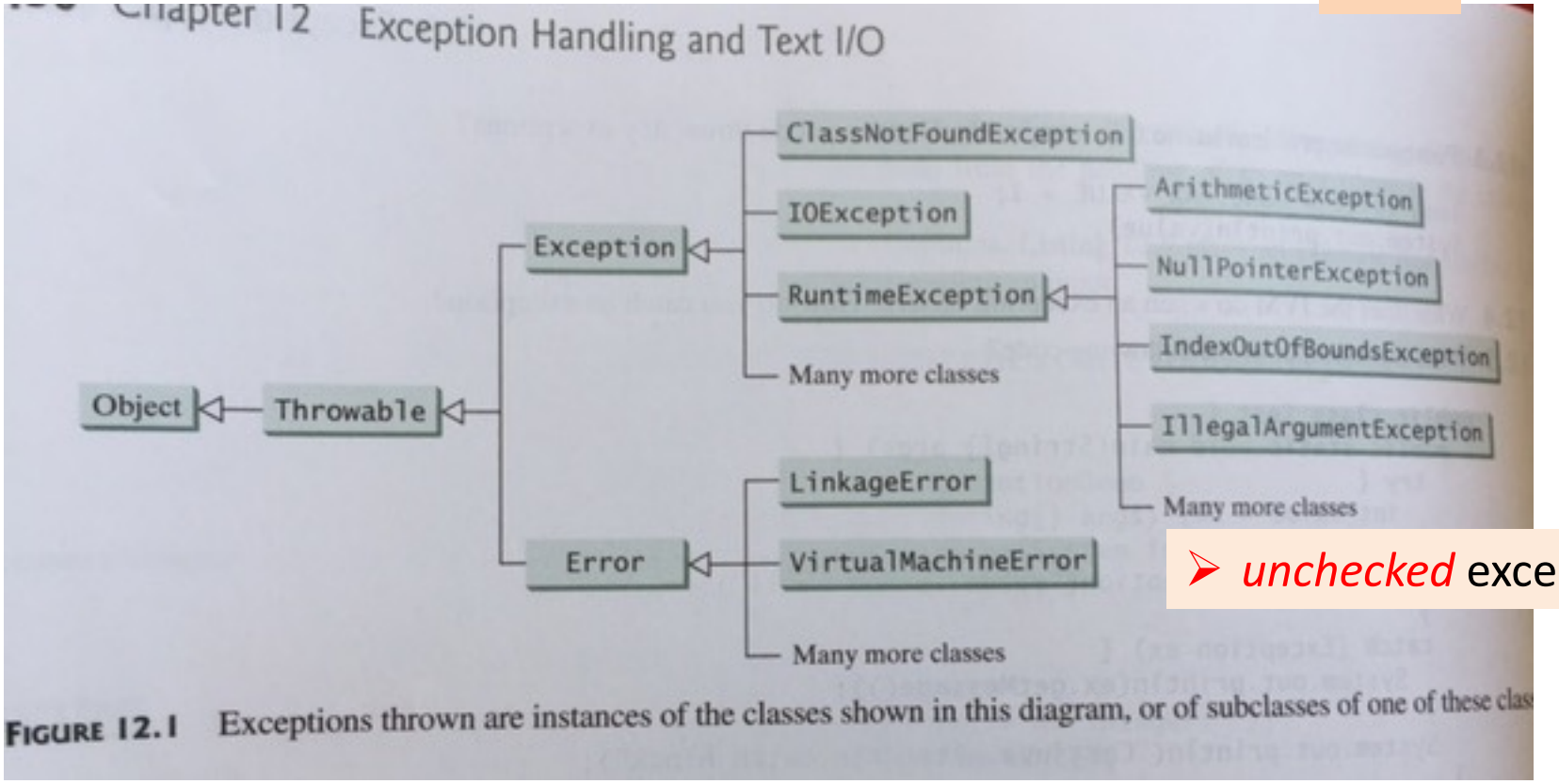


FIGURE 12.1 Exceptions thrown are instances of the classes shown in this diagram, or of subclasses of one of these clas

- *unchecked* exceptions

Exception Handling

- Exception is generic class
- Exception sub-classes
  - ClassNot FoundException
  - IOException

Tables 12.2-3
pp. 456-7

- RuntimeException sub-classes
  - ArithmeticException
  - NullPointerException
  - IndexOutOfBoundsException
  - IllegalArgumentException

➢ *unchecked* exceptions

➢ *example: File I/O*

➢ *checked* exceptions require declare *throws*
➢ *or use Try-Catch block*

Exception Handling


```
public class cs110Try {
public static void main(String[] args) {
    //*test I/O
    System.out.println("Hello World\n");

    //**test code here
try {
// code here
}
catch(Exception ex) {
//catch code here; "ex" is a parameter
System.out.println(ex);
}
    } //end main & class
}
```

Try-Catch


```
 9 public class cs110Try {
10 public static void main(String[] args) {
11     //*test I/O
12     System.out.println("Hello World\n");
13
14     //**test code here
15 try {
16 // code here
17 int x = 1/0; //create exception
18 }
19 catch(Exception ex) {
20 //catch code here; "ex" is a parameter
21 System.out.println(ex);
22 }
23 System.out.println("Past Catch block");
24
25     } //end main & class
```

Debugging

The code snippet below can print the exception stack using the **printStack()** method in the exception class:

```
public class MainClass {

  public static void main(String[] args) {
    // TODO Auto-generated method stub
    String test = null;
    try {
      int length = test.length();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

e.printStackTrace

```
java.lang.NullPointerException
    at com.example.MainClass.main(MainClass.java:9)
```

Try-Catch

12.4.3 Catching Exceptions

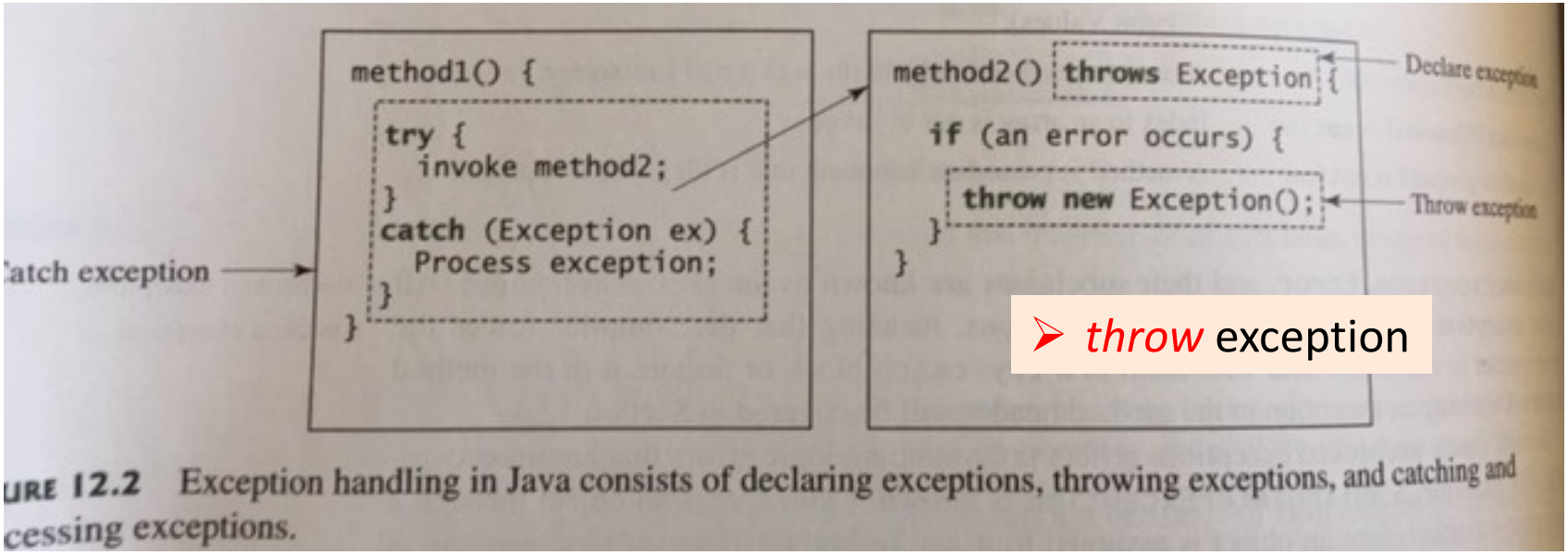
You now know how to declare an exception and
tion is thrown, it can be caught and handled in a

- *multiple* catches

```
try {
  statements; // Statements that may
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVarN) {
  handler for exceptionN;
}
```

Exception Handling

- *Throwing* exceptions
  - throw <exception name>;



URE 12.2 Exception handling in Java consists of declaring exceptions, throwing exceptions, and catching and cessing exceptions.

Try-Catch

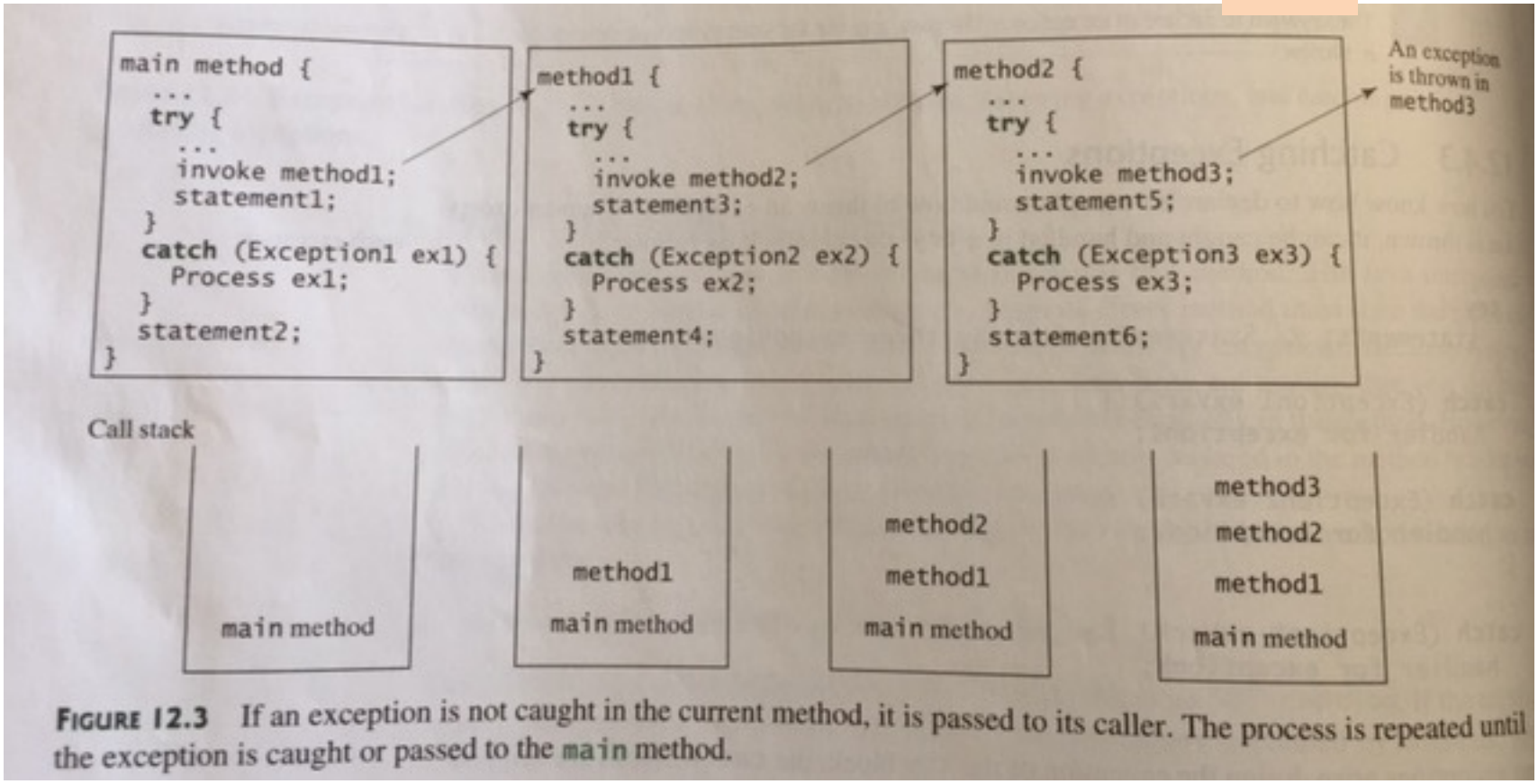


FIGURE 12.3 If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.

- *chaining* exception handling

Try-Catch-Finally


```
        //**test code here
try {
// code here
}
catch(Exception ex) {
//catch code here; "ex" is a parameter
System.out.println(ex);
}
```

```
finally {
<statements>
}
```

- *finally* block

Try-Catch-Finally

Ch 12

```
finally {
  finalStatements;
}
```

The code in the `finally` block is executed under all circumstances, regardless of whether an exception occurs in the `try` block or is caught. Consider three possible cases:

- If no exception arises in the `try` block, `finalStatements` is executed, and the next statement after the `try` statement is executed.
- If a statement causes an exception in the `try` block that is caught in a `catch` block, the rest of the statements in the `try` block are skipped, the `catch` block is executed, and the `finally` clause is executed. The next statement after the `try` statement is executed.
- If one of the statements causes an exception that is not caught in any `catch` block, the other statements in the `try` block are skipped, the `finally` clause is executed, and the exception is passed to the caller of this method.

The `finally` block executes even if there is a `return` statement prior to reaching the `finally` block.

➢ *finally* block

Note

The `catch` block may be omitted when the `finally` clause is used.

Putting It All Together


```
15 try {
16 // code here
17 int x = 3; //create exception
18 System.out.println("past x=1");
19 if (x==1) throw new ArithmeticException();
20 else if (x==2)throw new IndexOutOfBoundsException();
21 else if (x==3)throw new Exception();
22 //System.out.println("past throw");
23 }
24
25 catch(IndexOutOfBoundsException ex) {
26 //catch code here; "ex" is a parameter
27 System.out.println("catch1: " +ex);
28 }
29 catch(ArithmeticException ex) {
30 //catch code here; "ex" is a parameter
31 System.out.println("catch2: " +ex);
32 }
33 catch(Exception ex) {
34 //catch code here; "ex" is a para
35 System.out.println("catch3: " +ex
36 }
37 finally {
38 System.out.println("finally ..."
39 }
40 System.out.println("Past Catch & Finally block");
41     } //end main & class
```

```
----jGRASP exec: java cs110Try
start Main
past x=1
catch2: java.lang.ArithmeticException
finally ...
Past Catch & Finally block

----jGRASP: operation complete.
```

```
----jGRASP exec: java cs110Try
start Main
past x=1
catch3: java.lang.Exception
finally ...
Past Catch & Finally block

----jGRASP: operation complete.
```

```
----jGRASP exec: java cs110Try
start Main
past x=1
catch1: java.lang.IndexOutOfBoundsException
finally ...
Past Try, Catch & Finally blocks

----jGRASP: operation complete.
```

Putting It All Together


```
    //**test code here
try {
// code here
int x = 5; //create exception
System.out.println("past x=1");
if (x==1) throw new ArithmeticException();
else throw new IndexOutOfBoundsException();
//rest of code in block not executed
System.out.println("never gets here");
}
catch(IndexOutOfBoundsException ex) {
//catch code here; "ex" is a parameter
System.out.println("catch1: " +ex);
}
catch(ArithmeticException ex) {
System.out.println("catch2: " +ex);
}
catch(Exception ex) {
System.out.println("catch2: " +ex);
}
finally {
System.out.println("finally ...");
}
```

```
cs110Try.java:22: error: unreachable statement
System.out.println("never gets here");
^
1 error
```

Exception Handling Example


```
public void showDialog() {
    /*
     * "try" makes sure nothing goes wrong. If something does,
     * the interpreter skips to "catch" to see what it should do.
     */
    try {
        /*
         * The code below brings up a JOptionPane, which is a dialog box
         * The String returned by the "showInputDialog()" method is converted into
         * an integer, making the program treat it as a number instead of a word.
         * After that, this method calls a second method, calculate() that will
         * display either "Even" or "Odd."
         */
        userInput = Integer.parseInt(JOptionPane.showInputDialog("Please enter a number."));
        calculate();
    } catch (final NumberFormatException e) {
        /*
         * Getting in the catch block means that there was a problem with the format of
         * the number. Probably some letters were typed in instead of a number.
         */
        System.err.println("ERROR: Invalid input. Please type in a numerical value.");
    }
```

Java

Scope

Scope of Variables

- Scope of all identifiers (variables, methods)
  - LOCAL within block (for, while, case, method)

```
for (int i=0; i< wlen; i++) {
```

  - GLOBAL only when explicitly declared as such
    - static → declared in Class scope

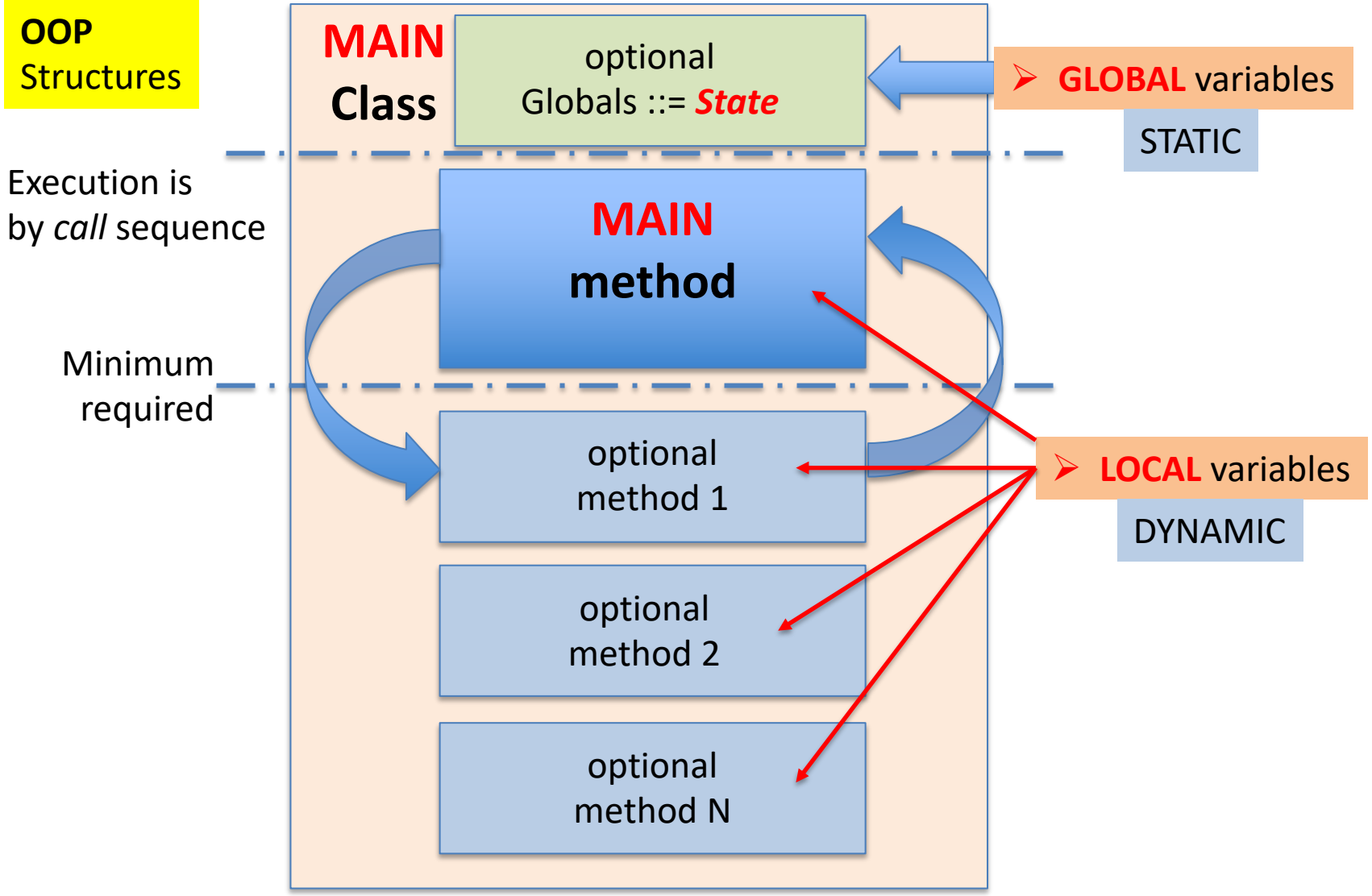
```
10 public class Lab4 {
11   static final boolean $DEBUG = true;
12   static final String spc = " ";
13   static String reason = "";
14 //main method
15   public static void main(String[] args) {
```

- Revisit for **Memory Mgt**

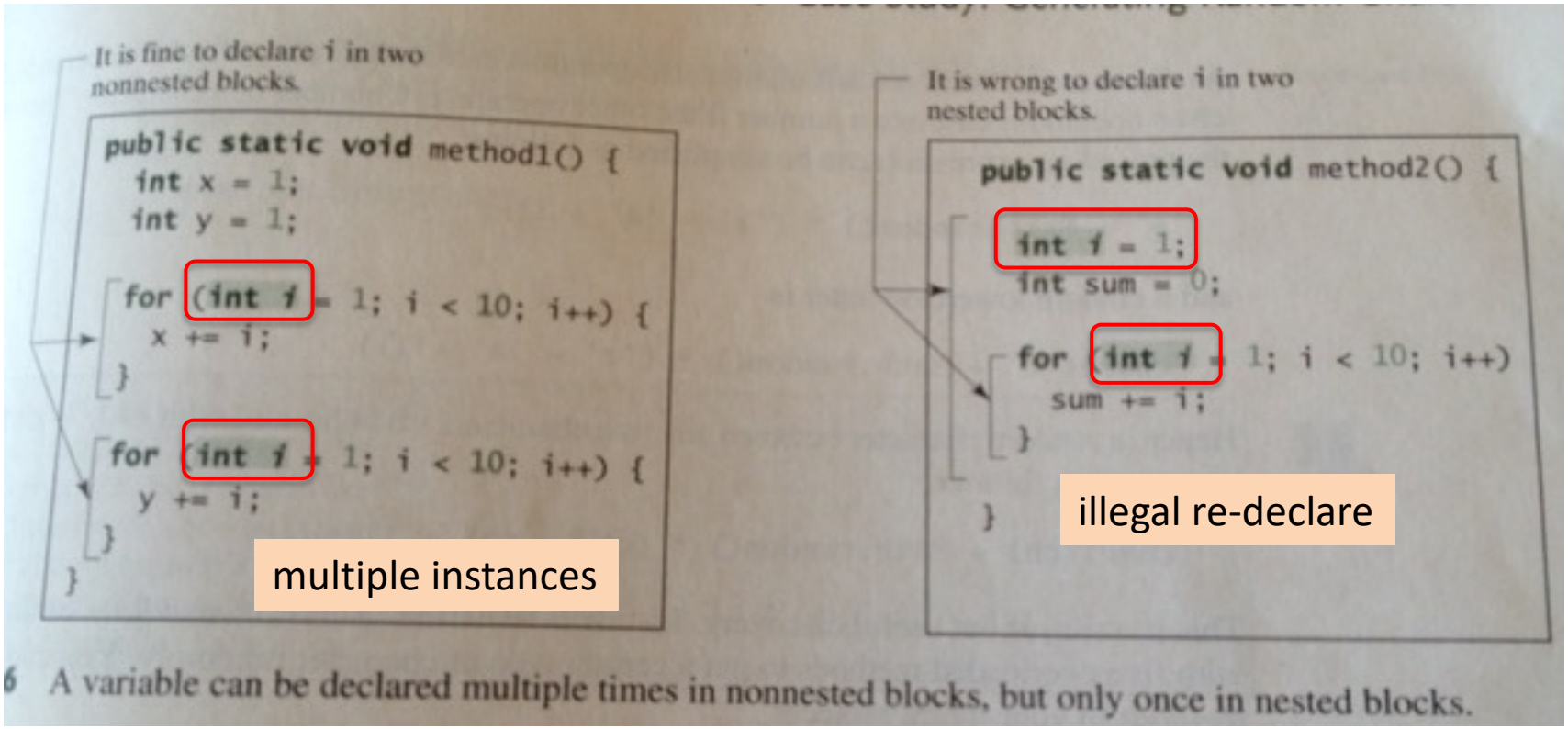
Main Class Structure

Java ⇔ OOP

OOP Structures



Scope of Variables



Java

Memory Mgt

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP110

Memory Mgt

DR JEFF SOFTWARE
INDIE APP DEVELOPER

10.1 Introduction to memory management

Present Note

ArrayLists

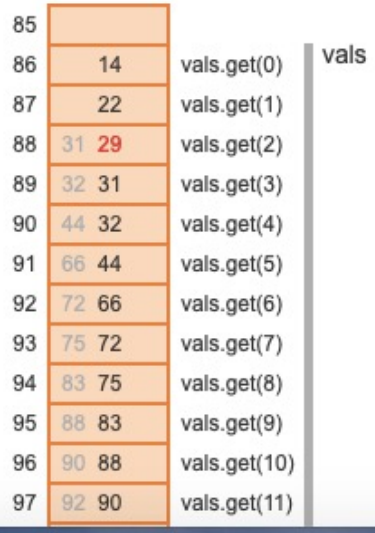
An ArrayList stores a list of items in contiguous mem ate access to any element at index i of ArrayList v by using the get() and set() methods — the program j he first element in v to arrive at the element. The methods `add(objRef)` and `add(i, objRef)` ap st, respectively. Now recall that inserting an item at locations other than the end of the ArrayList requir indexed items. Similarly, removing (via the `remove(i)` method) an item requires shifting higher-indexed items to fill the gap. Each shift of an item from one element to another element requires a few processor instructions. This issue exposes the ***ArrayList add/remove performance problem***.

For ArrayLists with thousands of elements, a single call to add() or remove() can require thousands of instructions, so if a program does many insert or remove operations on large ArrayLists, the program may run very slowly. The following animation illustrates shifting during an insertion operation.

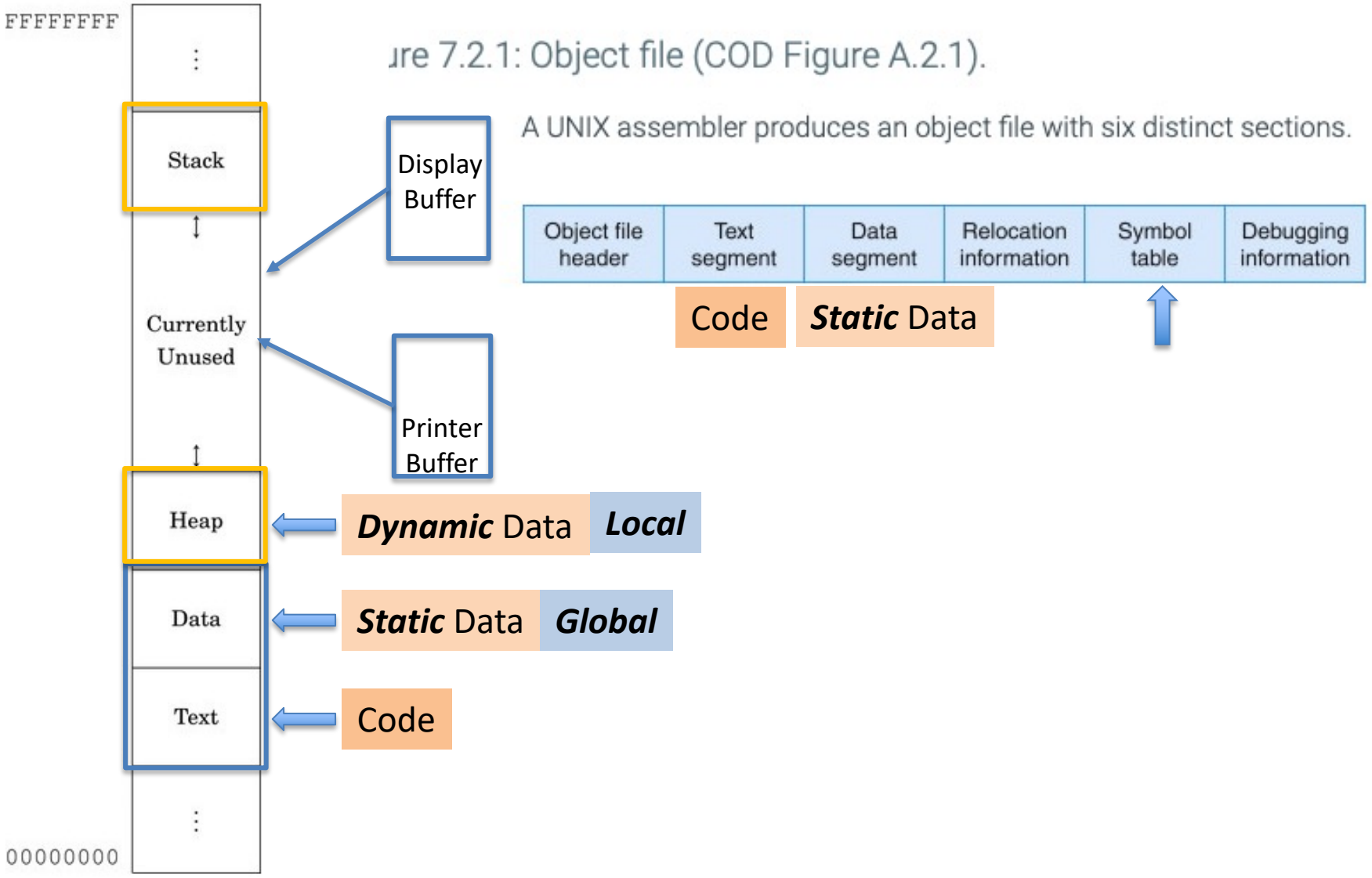
PARTICIPATION ACTIVITY 10.1.1: ArrayList add() performance problem.

Start 2x speed

```
...
vals.add(2, new Integer(29))
...
```



Memory Segments



ure 7.2.1: Object file (COD Figure A.2.1).

A UNIX assembler produces an object file with six distinct sections.

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information

Code **Static** Data

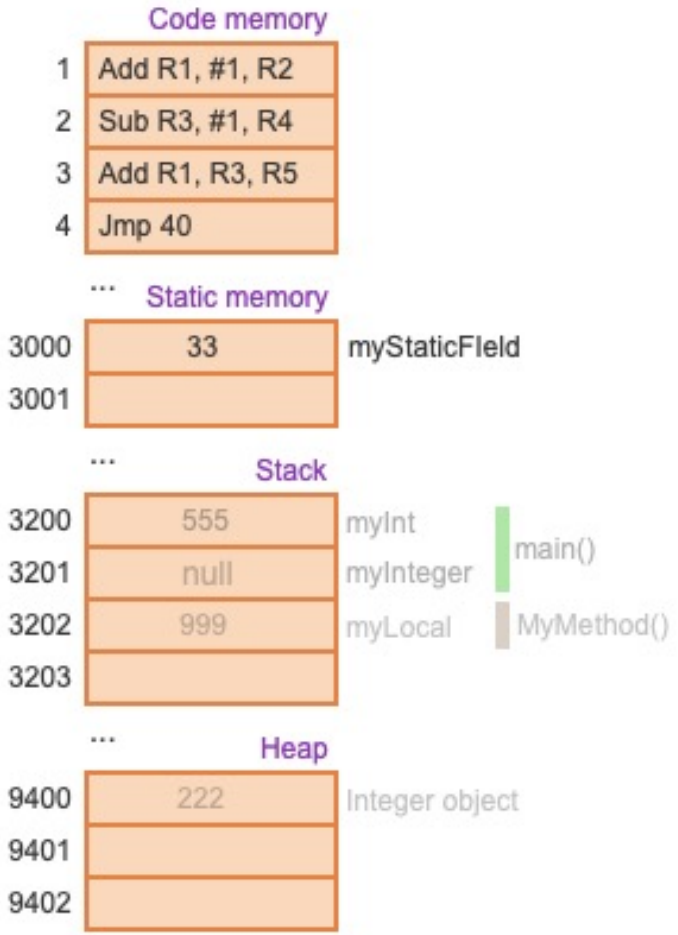
Typical memory layout for a program with a 32-bit address space.

Memory Segments

PARTICIPATION ACTIVITY 10.3.1: Use of the four memory regions.

Start 2x speed

```
// Program is stored in code memory
public class MemoryRegionEx {
   public static int myStaticField = 33;

   public static void myMethod() {
      int myLocal;        // On stack
      myLocal = 999;
      System.out.print(" " + myLocal);
   }

   public static void main(String[] args) {
      int myInt;                  // On stack
      Integer myInteger = null; // On stack
      myInt = 555;

      myInteger = new Integer(222); // In heap
      System.out.print(myInteger.intValue() +
                       " " + myInt);

      myInteger = null;

      myMethod(); // Stack grows, then shrinks
   } // Object deallocated automatically
}
```



Scope + Memory Mgt

- Persistence
  - Instance (default)
  - *Static*
    - Static vs. Dynamic
- Privacy
  - *Public*
  - *Private*
  - *Protected (Ch 11)*
  - Default

Memory Mgt

- Create: **new**
- Delete:
  - **= null** → refCount--
  - Pointer reassignment: **A = B;**

Memory Mgt

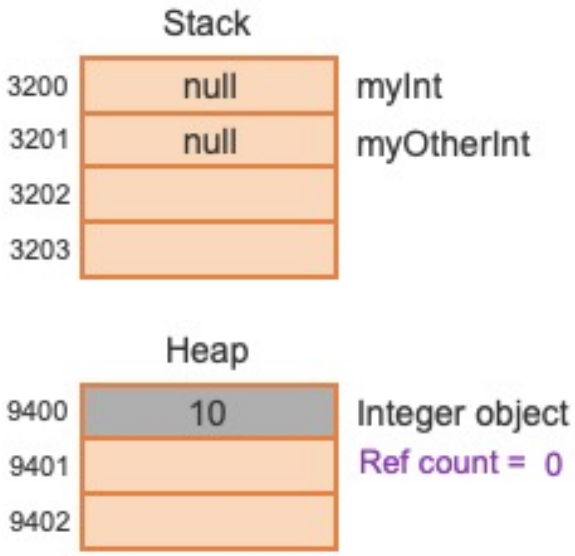
10.4 Basic garbage collection

In order to determine which allocated objects the program is currently using at runtime, the Java virtual machine keeps a count, known as a ***reference count***, of all reference variables that are currently referring to an object. If the reference count is zero, then the object is considered an ***unreachable object*** and is eligible for garbage collection, as no variables in the program refer to the object. The Java virtual machine marks unreachable objects, and deallocation occurs the next time the Java virtual machine invokes the garbage collector. The following animation illustrates.

PARTICIPATION ACTIVITY 10.4.1: Marking unused objects for deallocation.

Start ☐ 2x speed

```
Integer myInt = null;
Integer myOtherInt = null;

// Create object and assign reference
myInt = new Integer(10);

// Assign object reference
myOtherInt = myInt;

// Use object ...

// myInt does not refer to object
myInt = null;

// myOtherInt does not refer to object
myOtherInt = null;
```



CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

Memory Mgt

DR JEFF SOFTWARE
INDIE APP DEVELOPER

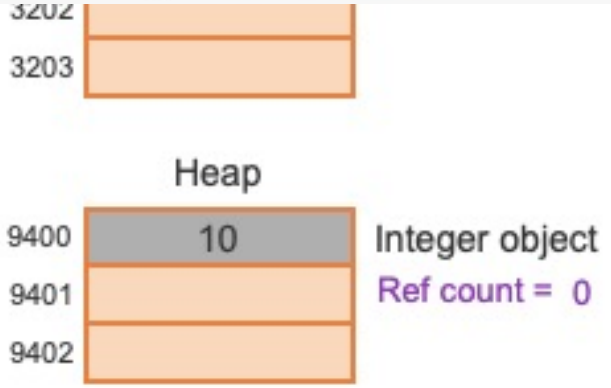
The program initially allocates memory for an Integer object and assigns a reference to the object's memory location to variables myInt and myOtherInt. Thus, the object's reference count is displayed as two at that point in the program's execution. After the object is no longer needed, the reference variables are assigned a value of *null*, indicating that the reference variables no longer refer to an object. Consequently, the object's reference count decrements to zero, and the Java virtual machine marks that object for deallocation.

PARTICIPATION ACTIVITY | 10.4.1: Marking unused objects for deallocation.

Start ☐ 2x speed

The program initially allocates memory for an Integer object and assigns a reference to the object's memory location to variables myInt and myOtherInt. Thus, the object's reference count is displayed as two at that point in the program's execution. After the object is no longer needed, the reference variables are assigned a value of *null*, indicating that the reference variables no longer refer to an object. Consequently, the object's reference count decrements to zero, and the Java virtual machine marks that object for deallocation.

```
// Create object and assign reference
myInt = new Integer(10);

// Assign object reference
myOtherInt = myInt;

// Use object ...

// myInt does not refer to object
myInt = null;

// myOtherInt does not refer to object
myOtherInt = null;

// Other instructions ...
```



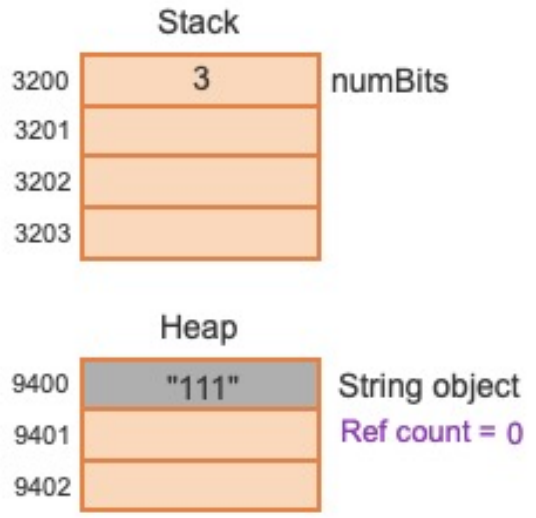
Memory Mgt

10.5 Garbage collection and variable scope

PARTICIPATION ACTIVITY 10.5.1: Marking unused objects in methods.

Start ☐ 2x speed

```
public class BitCounter {
   public static int countBits(int inNum) {
      int countInt;
      String binaryStr;

      binaryStr = Integer.toBinaryString(inNum);
      countInt = binaryStr.length();

      return countInt;
   }

   public static void main(String[] args) {
      int numBits;

      numBits = countBits(7); //Method call

      // Other instructions ...
   }
}
```



CSUN
CALIFORNIA
STATE UNIVERSITY
NORTHRIDGE
COMP110

Zy Chapter 11

DR JEFF
SOFTWARE
INDIE APP DEVELOPER

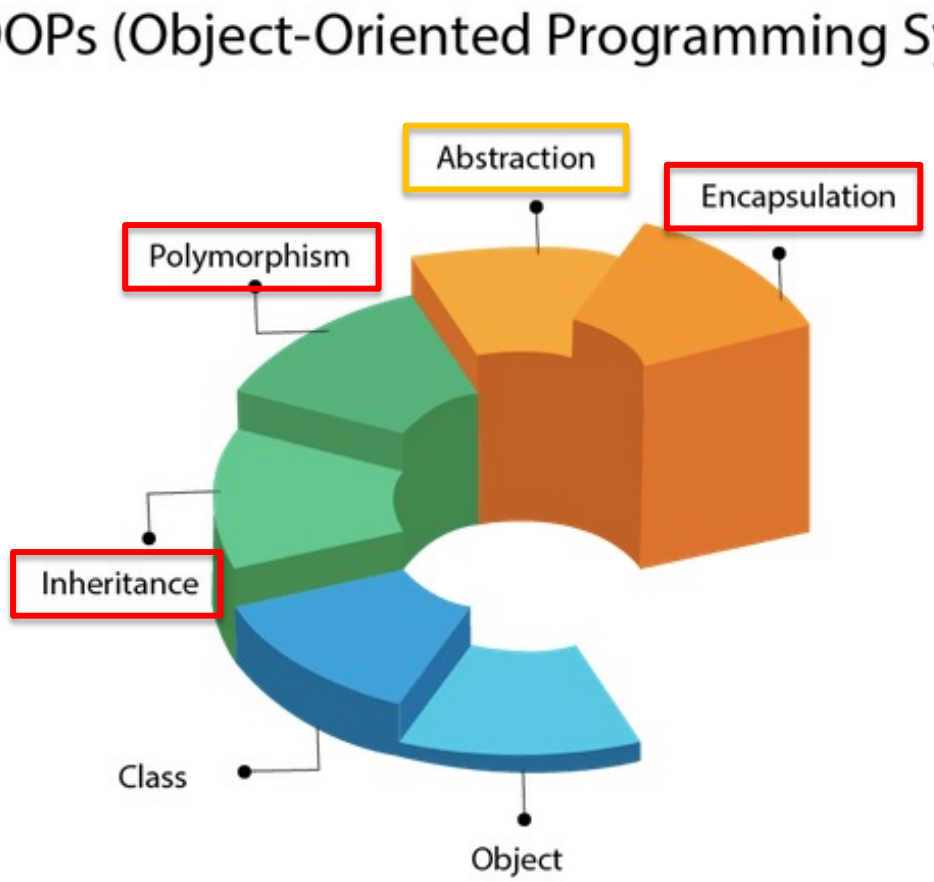
OOP:
Objects
& Classes

Hierarchy (Outline)

1. Main
    - 1.1 Main method
    - 1.2 2nd method
    - 1.3 3rd method
2. Class A
    - 2.1 1st method
    - 2.2 2nd method
3. Class B
    - 3.1 1st method
    - 3.2 2nd method

OOP Pillars

Concepts of OOPs(Object-Oriented Programming System):



CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP110

Classes & Objects

Overview

Classes *create* new → **Objects**

- Large *encapsulating* structure
  - **Collecting/grouping *Methods***
- Extends large programs by adding structures
- Supports *abstraction* as *Objects*
- Serve as *templates* for *Objects*

- *Objects* are *abstractions*
- *Objects* are *instances* of Classes

→ Scanner input = new Scanner

Name of new instance

Classes & Objects

Overview

- Built-in
  - Library functions (imports)
  - Wrappers (Integer, String)
- Programmer defined
- Special components
  - Constructors
  - Data Fields (global vars)

Object Oriented Design

3 *Pillars* of OOP

- **Encapsulation**
  - Objects
    - Classes as models
  - Classes
    - Properties/Data Fields
    - Constructors
    - Methods
- **Inheritance**
  - Class *extends*
- **Polymorphism**
  - Multiple Instantiations
    - Small changes to ***Data Fields***
    - Small changes to ***Methods***

```
Class Foo
   <decl vars (init)>
   <constructors>
   Fn 1{   }
   Fn 2{   }
End Class
```

- Declare class Foo
- Declare vars
- Define *constructors*
- Define *methods*

```
Fee extends Foo
   <decl vars2 (init)>
   Fn 3{   }
End Class
```

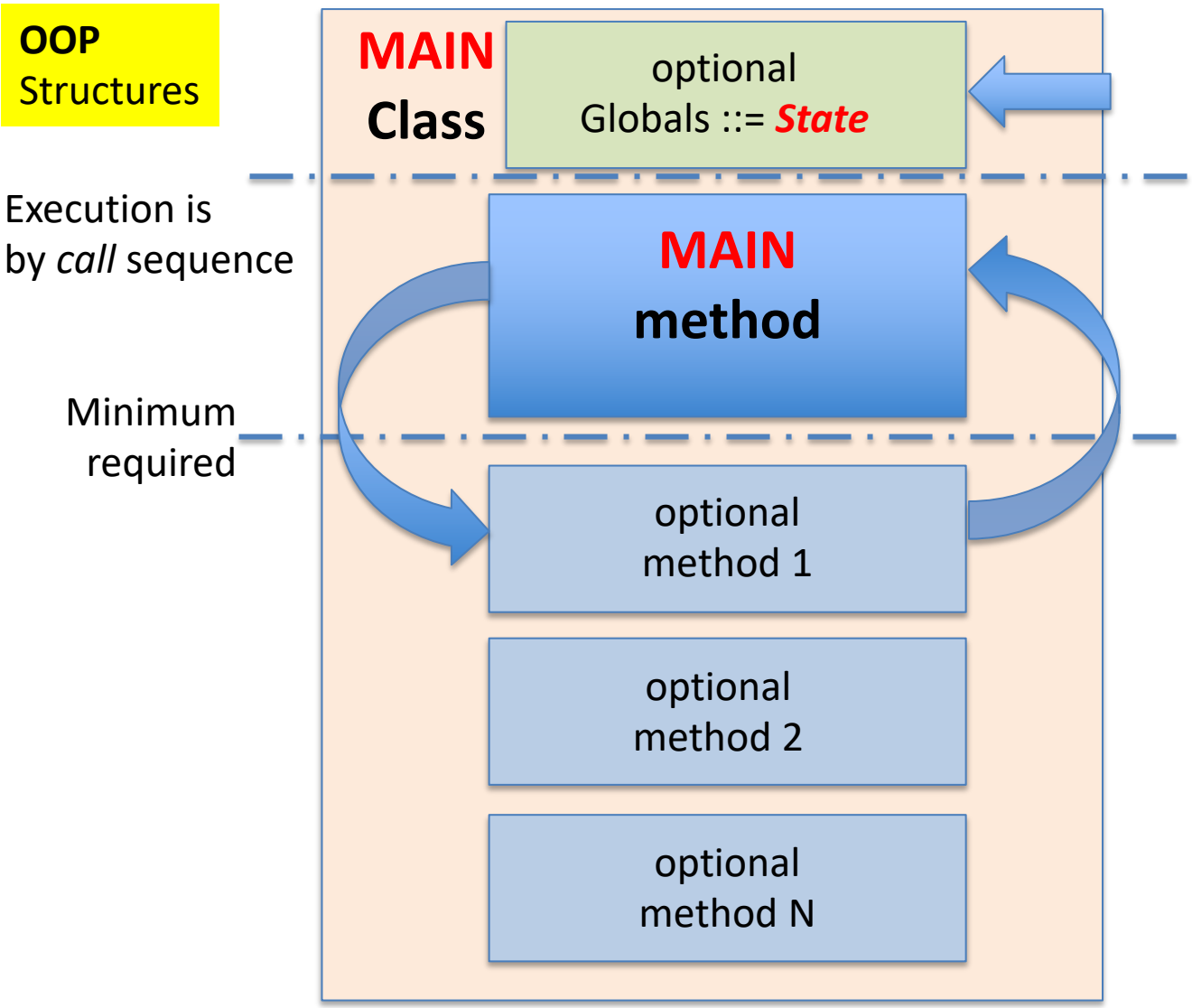
- Fee Instantiates Foo
- Fee Inherits Foo
- Fee Adds code to Foo

polymorphism

Main Class Structure

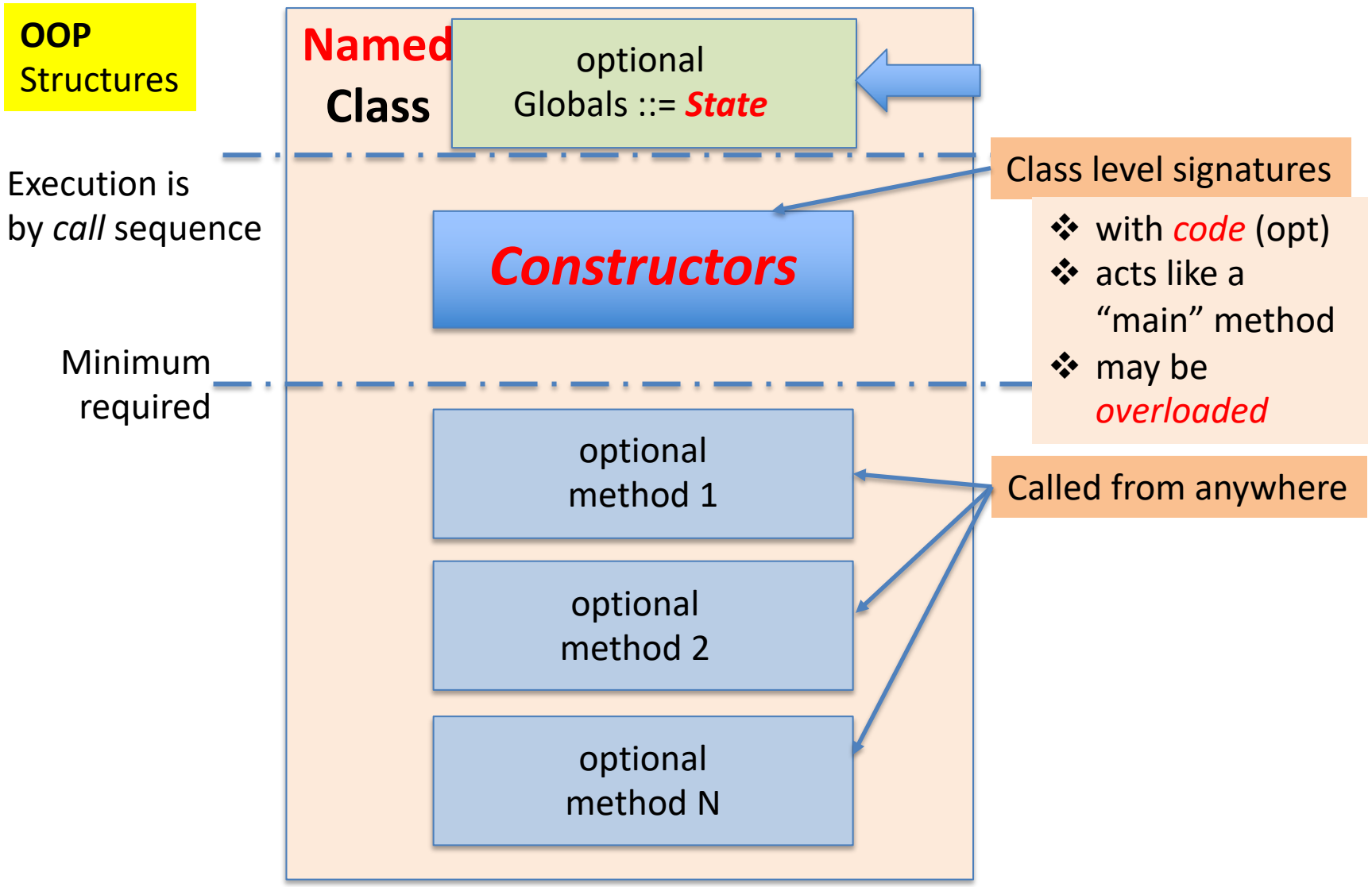
Java ⇔ OOP

OOP
Structures



Class Structure

Java ⇔ OOP



Method Structure

Java ⇔ OOP

Block Structures

Execution is sequential

Any method

Loose code

Conditional block
- IF-THEN-ELSE
- SWITCH-CASE

Loose code

Loop block
- FOR
- WHILE

Loose code

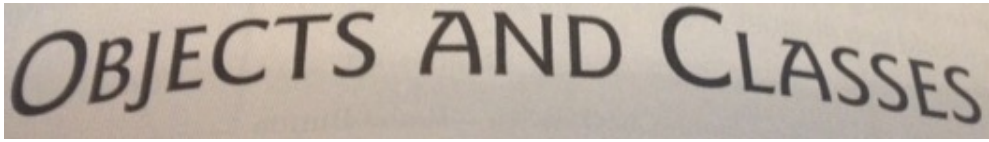
Loose code

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP110

Liang Chapter 9

DR JEFF SOFTWARE
INDIE APP DEVELOPER

Objects & Classes



1. Intro
2. Defining Classes for Objects
3. Examples: Circles, TVs
4. *Constructors*
5. *Reference* Variables
6. Java Class Library
7. Static Variables & Methods
8. Packages (Visibility)
9. Private Data Fields (Encapsulation)
10. Passing Objects to Methods (Args→Parms)
11. Array of Objects
12. Immutable Objects/Classes
13. Scope
14. ***this*** Reference

Example Objects

- Geometric shapes
  - Circles
  - Rectangles
- Fruits/veggies
  - Apples
  - Oranges
- Houses
- Animals/breeds
  - Dogs
  - Cats
- Games
  - Blackjack
  - Poker
  - Tic-Tac-Toe

➢ *Objects* have *Properties*

Example Object Properties

Examples: Fruit, Shapes, Houses

- Apple
  - Color
  - Size
  - Shape
  - Taste
  - Variety
- Orange
  - Color
  - Size
  - Shape
  - Variety
- Pear
- Circle
  - Radius
  - Area
  - Circumference
- Rectangle
  - Length
  - Width
  - Area
  - Perimeter
- Triangle
- House
  - # bedrooms
  - # bathrooms
  - size (sqft)
- Hotel
  - # rooms
  - Price range
  - # Parking spaces
- Bldg

- *Objects* have *Properties*
- *State* = *{Properties}*

- *Objects* are created from *Classes*

instantiated

Objects Example

- Apple
  - Color
  - Size
  - Shape
  - Taste
  - Variety

- 3 **Apple** objects created
- From 1 template = "Class"

Apple1

Apple2

Apple3

Objects Example

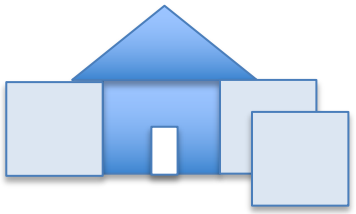
- House
  - # bedrooms
  - # bathrooms
  - size (sqft)

- 4 **House** objects created
- From 1 template = "Class"

House1

House2

House3



House4

House *polymorphisms*

polymorphism | ˌpälēˈmôrfizəm |

noun

the condition of occurring in several different forms: *the complexity and polymorphism of human cognition.*

Example Object Hierarchy

Example: Biology Taxononmy

- Kingdom
- Phylum
- Class
- Order
  - **Mammals**
  - Reptiles
  - Amphibians
  - Birds
- Family (Mammals)
  - Primates
  - Canines
  - Felines
- Genus
- Species

- *Class hierarchy*
  - *Super classes*

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP110

Object Oriented Design

Ch 9

➢ 3 "Pillars" of OOP

❖Encapsulation

- ❑ Objects
  - ✧ Classes as models
- ❑ Classes
  - ✧ Properties
  - ✧ Constructors
  - ✧ Methods

```
Class Foo
   <state vars>
   meth1
   meth2
Obj1 = new Fee
End Class
```

- ❖ Declare class Foo
- ❖ Declare state vars
- ❖ Define *methods*
- ❖ Add code

❖Inheritance

➢ *extends*

- ❑ Class Instantiations

```
Class Fee extends Foo
   <state vars>
   meth3
End Class
```

polymorphism

- ❖ Foo Instantiates Fee
- ❖ Foo Inherits Fee
- ❖ Fee Adds code to Foo

❖Polymorphism

- ❑ Multiple Instantiations
  - ✧ Additional Methods and/or State
- ❑ Method overloading

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP110

Objects: Data + Code

Ch 9

DATA

- *Objects* have *Properties*
- *State* = *{Properties}*

state ::= {properties}

aka "data fields"

- Data (Apple)
  - Color
  - Size
  - Shape
  - Taste
  - Variety

CODE

- *Objects* have *Methods*

- Code
  - *Constructors*
  - Methods
    - *Getters*
    - *Setters*
    - Other methods

OOP – Class Structure


```
public class <classname> {
    <class state:  properties>
    //constructors
    <classname> ( )  {  //no args
    }
    void meth1( ) {
    <code 1>
    }
    int meth2(parm) {
    <code 2>
    }
//end class
}
```

- Classes may (optional) include ***constructors***

<classname>.meth1()

- *Classes **do** include methods*

<classname>.meth2(arg)

- ***Main*** Class
  - includes ***main*** *method*
  - has no constructors

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP110

OOP – Class Structure

Ch 9

- Class.property
- Class.method(args…)

```
public class <className> {
    <class state>
    <class constructors>
    <main method>*
    <other methods>
}
```

*may be empty

*only in "main" class

<state> ::= {<properties> }
<properties> ::= <attributes> ::= <data fields>
< constructors> ::= <ClassName>( ) |
| <className>(<parms…>)
| <default>

❖ Constructors

- templates (blueprints)
- *signatures* like methods
  - any number of parameters (>=0)
- *overloading*
  - any number of parameters (>=0)
  - called with any signature (# args)
- code (optional)
- default

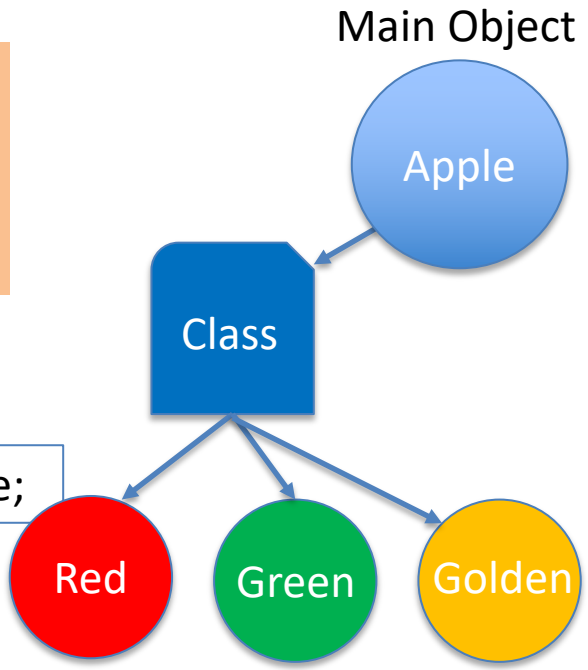
Objects vs. Classes

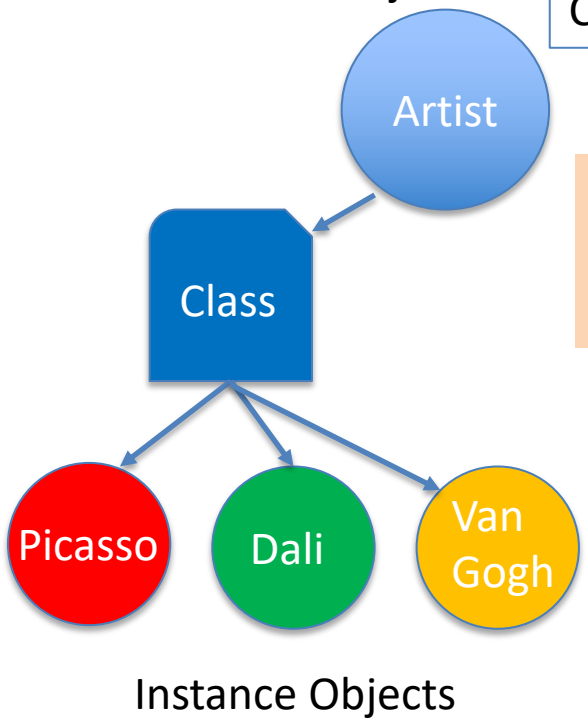
- ***Classes*** are ***templates*** for *objects*
- ***Objects*** are ***instances*** of *Classes*

```
class ClassName {
}
```

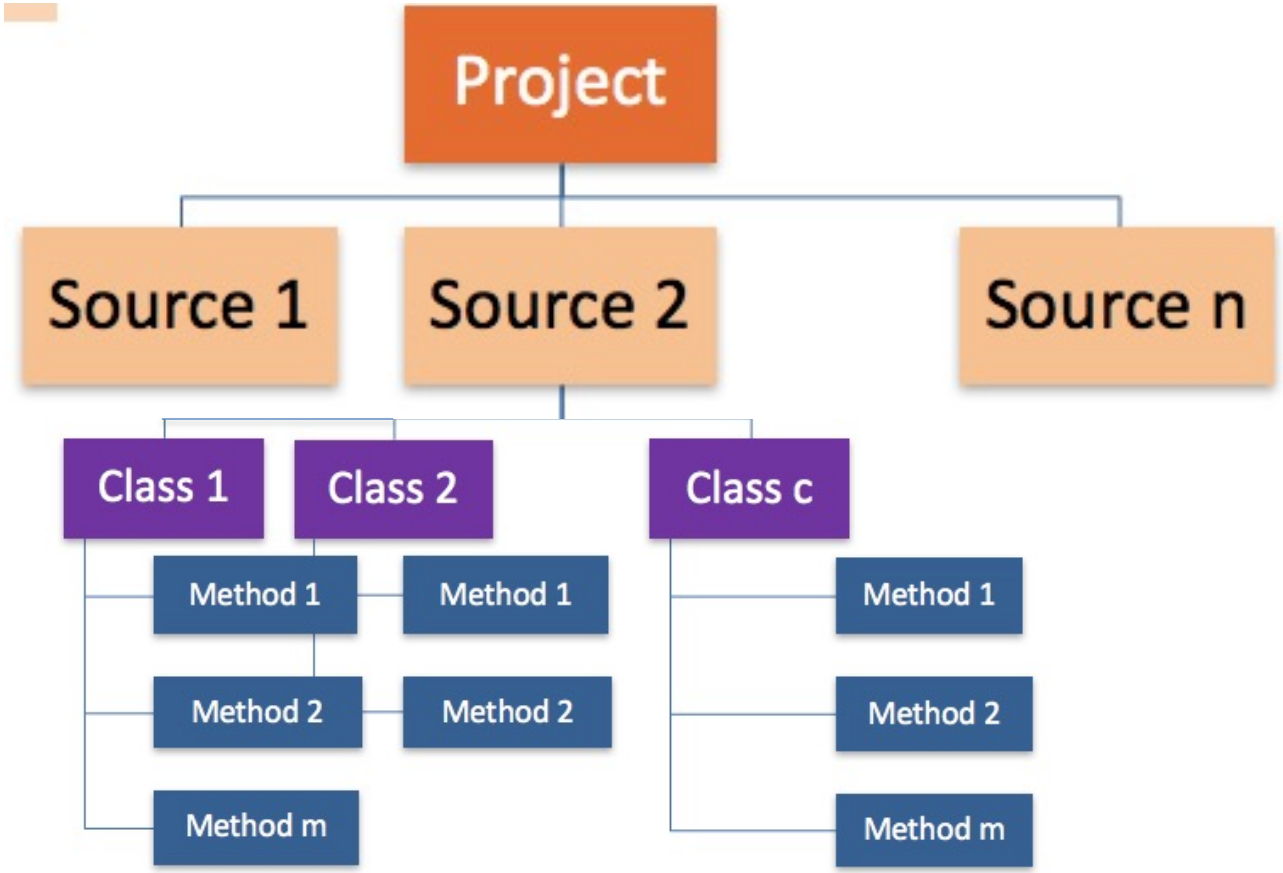
```
ClassName Cname = new ClassName;
```

- ClassName is a ***Class***
- Cname is an ***Object*** (an ***instance*** of ClassName)





UML Diagrams



TV Class UML

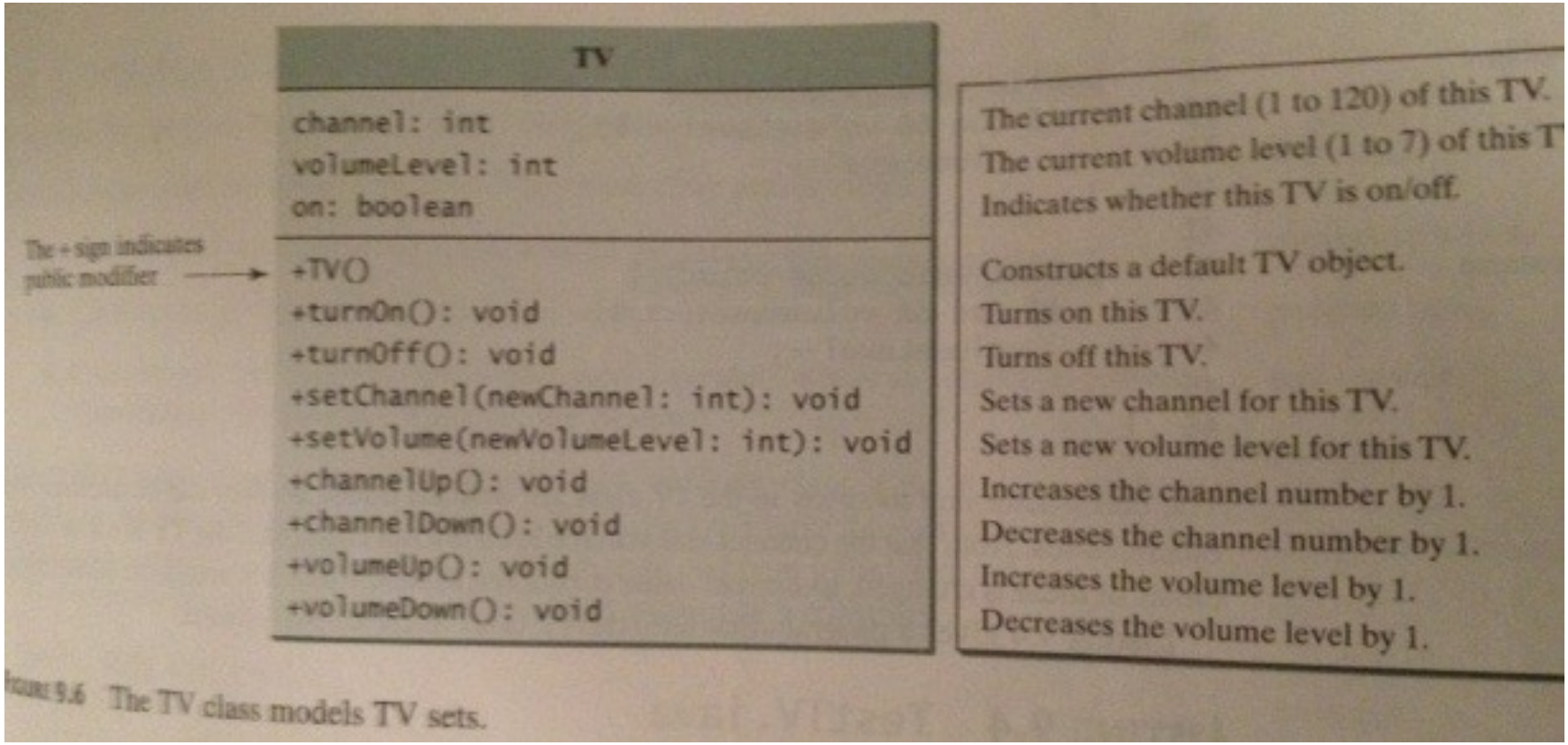


FIGURE 9.6 The TV class models TV sets.

HDTV

Artist Class UML

❖ Class

❖ Objects

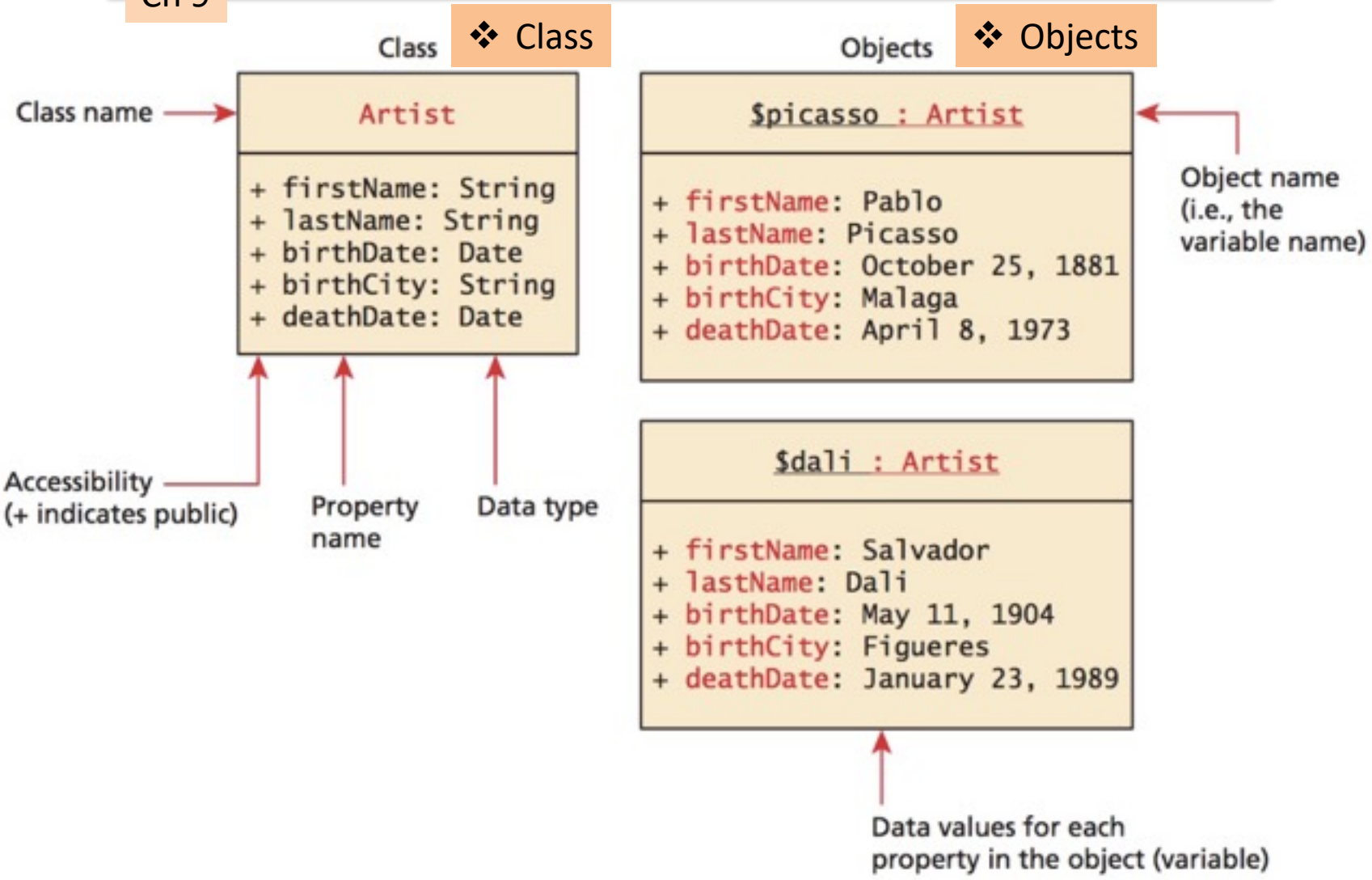


FIGURE 10.2 Relationship between a class and its objects in UML

OOP – Constructors

Ch 9

```
public class <classname> {
        <class state:  properties>
        //constructors
        <classname> ( )  {  //no args
        <code 1>*
        }
        <classname> (arg )  {  //1 arg
        <code 2>*
        }
        <classname> (arg1,arg2)  {  //2 args
        <code 3>*
        }
        <methods>
//end class
}
```

no type

*code (if any) gets executed upon each instantiation

❖ 0 or more methods

OOP – Simple Example


```
 9 public class classes {
10 public static boolean $DEBUG = true; //global
11 public static void main(String[] args) {
12     //*test/debug
13     if ($DEBUG) System.out.println("Hello World\n");
14     class1 clx = new class1();
15     } //end main
16 } //end class
17
18 class class1 {
19 public static boolean $DEBUG = true;
20 //constuctor
21 class1() {
22     if ($DEBUG) System.out.println("Hello CLASS\n");
23     } //end class1
24 }
```

```
----jGRASP exec: java classes
Hello World

Hello CLASS
```

OOP – Simple Example


```
19 class class1 {
20    public static boolean $DEBUG = true;
21 //constuctor
22 class1() {
23      if ($DEBUG) System.out.println("Hello CLASS\n");
24      }
25 class1(int arg) {
26      if ($DEBUG) System.out.println("Hello CLASS No. " + arg + "\n");
27      }
28 //loose code (not in a constructor or method)
29 System.out.println("code in class1");
30 }
```

```
classes.java:29: error: <identifier> expected
System.out.println("code in class1");
                   ^

classes.java:29: error: illegal start of type
System.out.println("code in class1");
                   ^

2 errors
```

OOP – Simple Example

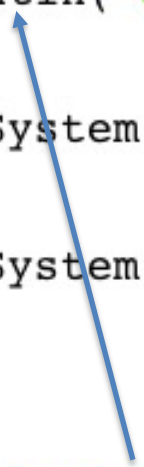

```
12        //*test/debug
13        if ($DEBUG) System.out.println("Hello World\n");
14        class1 cl1 = new class1();
15        class1 cl2 = new class1(5);
16        class1.test(); //test code in method
17    } //end main
18 } //end class
19
20 class class1 {
21    public static boolean $DEBUG = true;
22 //constuctor
23 class1() {
24       if ($DEBUG) System.out.println("Hello CLASS\n");
25       }
26 class1(int arg) {
27       if ($DEBUG) System.out.println("Hello CLASS No. " + arg + "\n");
28       }
29       //loose code in a method
30       void test() {
31    System.out.println("code in class1");
32    }
```

static

```
classes.java:16: error: non-static method test() cannot be referenc
        class1.test(); //test code in method
```

OOP – Simple Example


```
19 class class1 {
20    public static boolean $DEBUG = true;
21    //loose code (not in a constructor or method)
22    System.out.println("code in class1");
23 //constuctor
24 class1() {
25      if ($DEBUG) System.out.println("Hello CLASS\n");
26      }
27 class1(int arg) {
28      if ($DEBUG) System.out.println("Hello CLASS No. " + arg + "\n");
29      }
30 }
```



```
classes.java:22: error: <identifier> expected
System.out.println("code in class1");
                   ^

classes.java:22: error: illegal start of type
System.out.println("code in class1");
                   ^
```

OOP – Simple Example


```
13        if ($DEBUG) System.out.println("Hello World\n");
14        class1 cl1 = new class1();
15        class1 cl2 = new class1(5);
16     } //end main
17  } //end class
18
19  class class1 {
20     public static boolean $DEBUG = true;
21  //constuctor
22  class1() {
23        if ($DEBUG) System.out.println("Hello CLASS\n");
24        }
25  class1(int arg) {
26        if ($DEBUG) System.out.println("Hello CLASS No. " + arg + "\n");
27        }
28  //loose code in a method
29     void test() {
30     System.out.println("code in class1");
31     }
```

did not run

```
Hello CLASS

Hello CLASS No. 5


  ----jGRASP: operation complete.
```

OOP – Simple Example


```
15        class1 cl2 = new class1(5);
16        class1.test(); //test code in method
17      } //end main
18  } //end class
19
20  class class1 {
21      public static boolean $DEBUG = true;
22  //constuctor
23  class1() {
24        if ($DEBUG) System.out.println("Hello CLASS\n");
25        }
26  class1(int arg) {
27        if ($DEBUG) System.out.println("Hello CLASS No. " + arg + "\n");
28        }
29        //loose code in a method
          static void test() {
31    System.out.println("code in class1");
32      }
```

did run

```
Hello CLASS

Hello CLASS No. 5

code in class1

----jGRASP: operation complete.
```

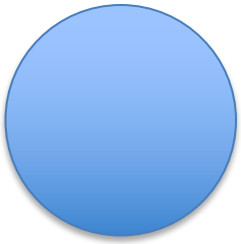
Objects Example: Circles

- Circle
  - Radius

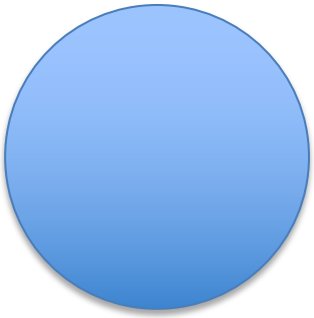
- 3 **Circle** objects created
- From 1 template = "Class"

- Methods
  - **Get**
    - Perimeter
    - Area
  - **Set**
    - Radius

Getters and *Setters*



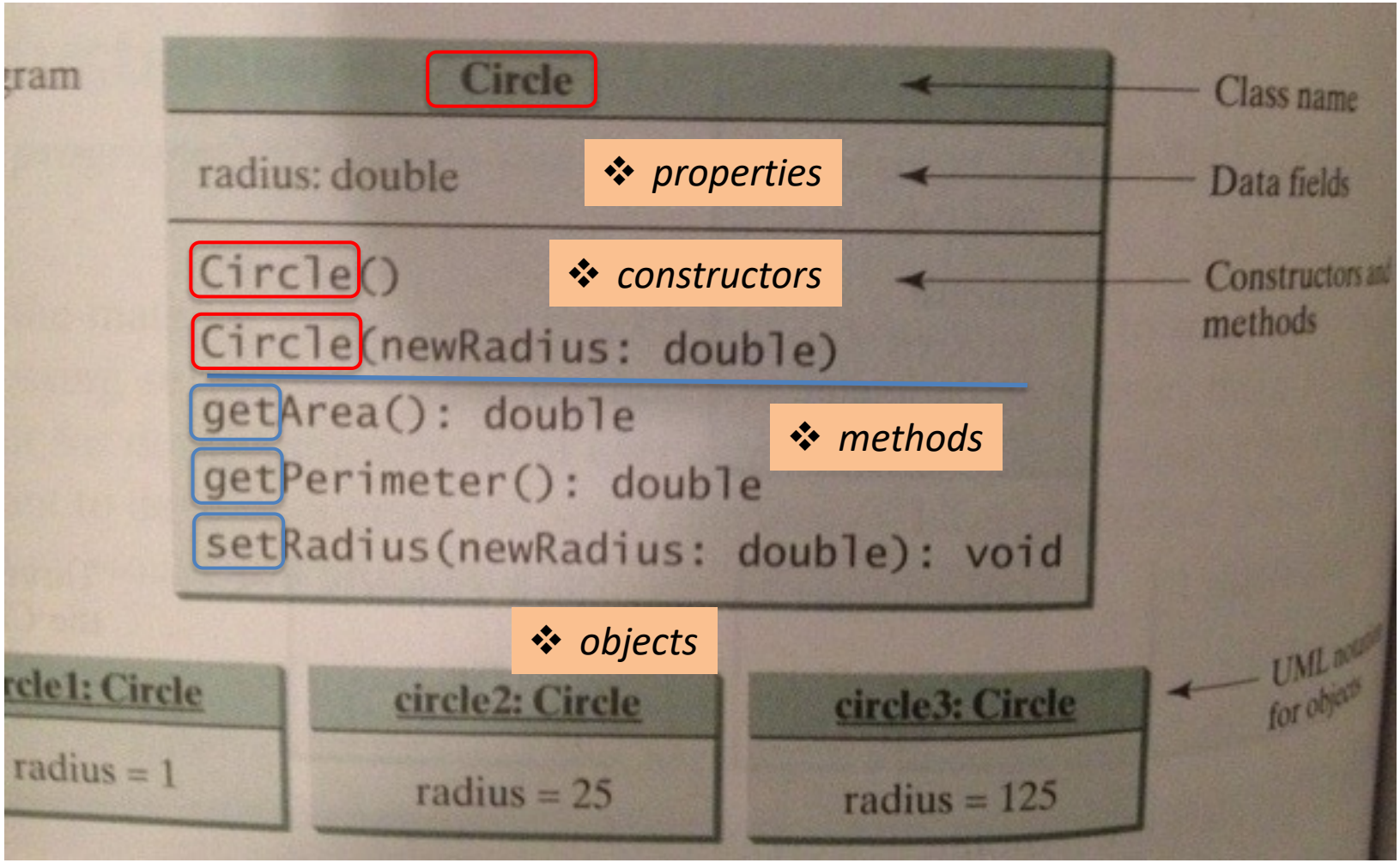
Circle1



Circle2

Circle3

Circle Class UML

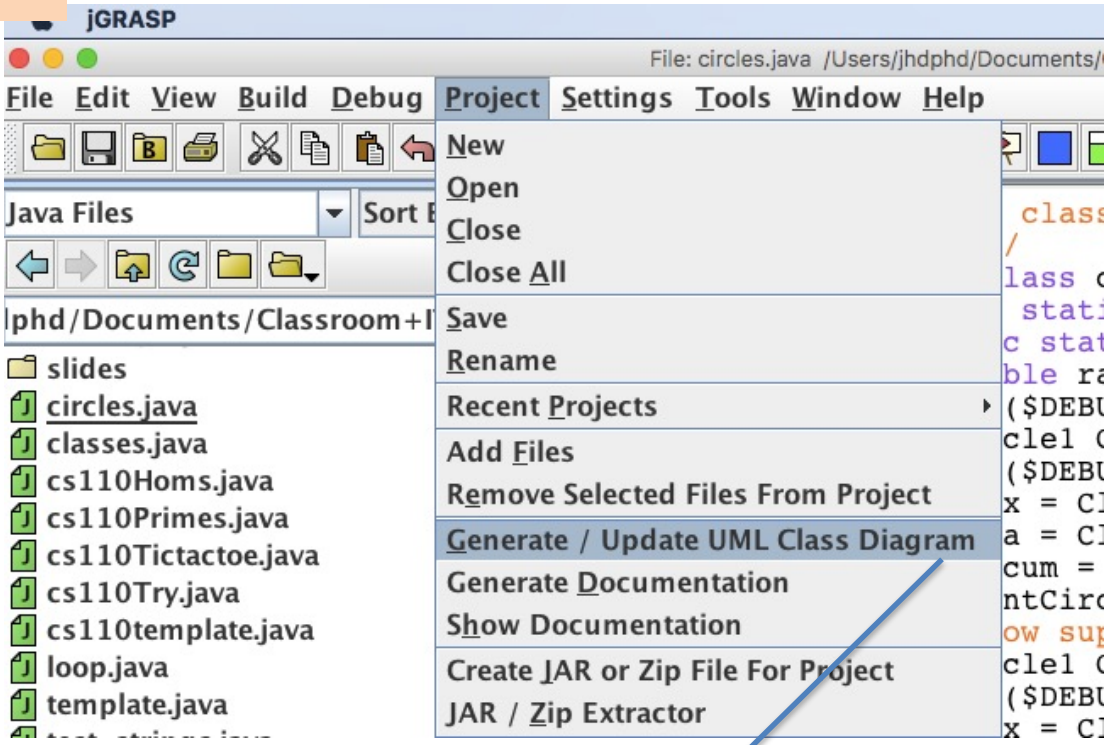


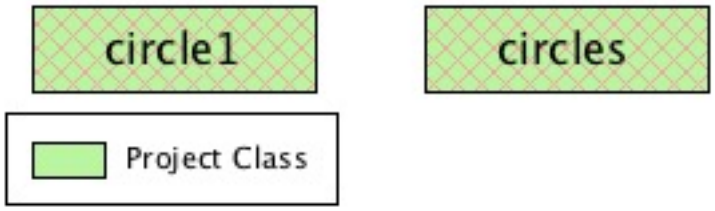
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP110

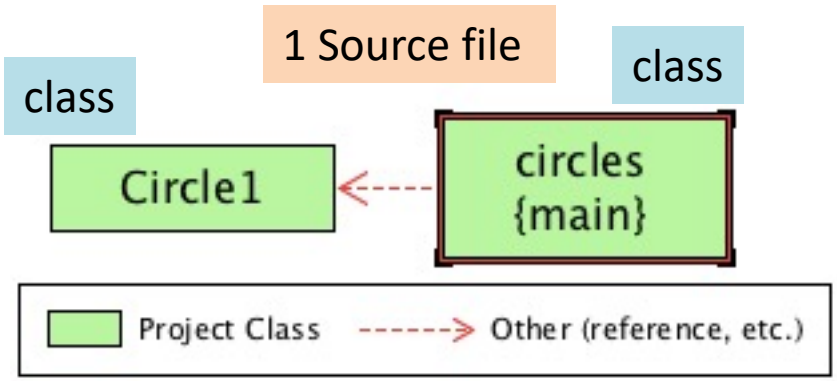
Circle UML in jGRASP

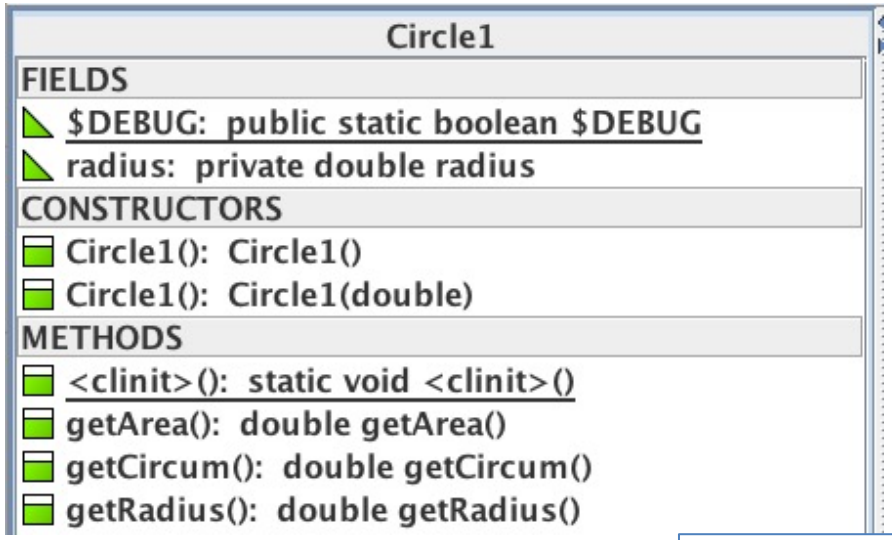
Ch 9

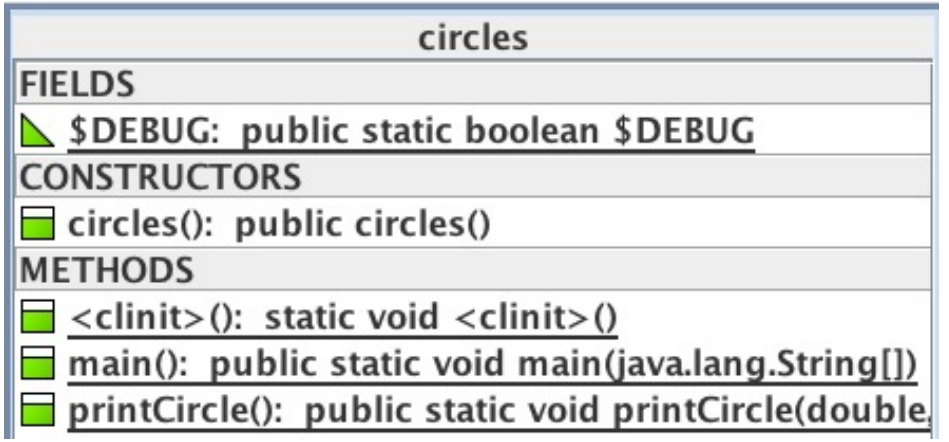




UML in jGRASP







\+ → public

\- → private

Underline → static

OOP – Circle Code

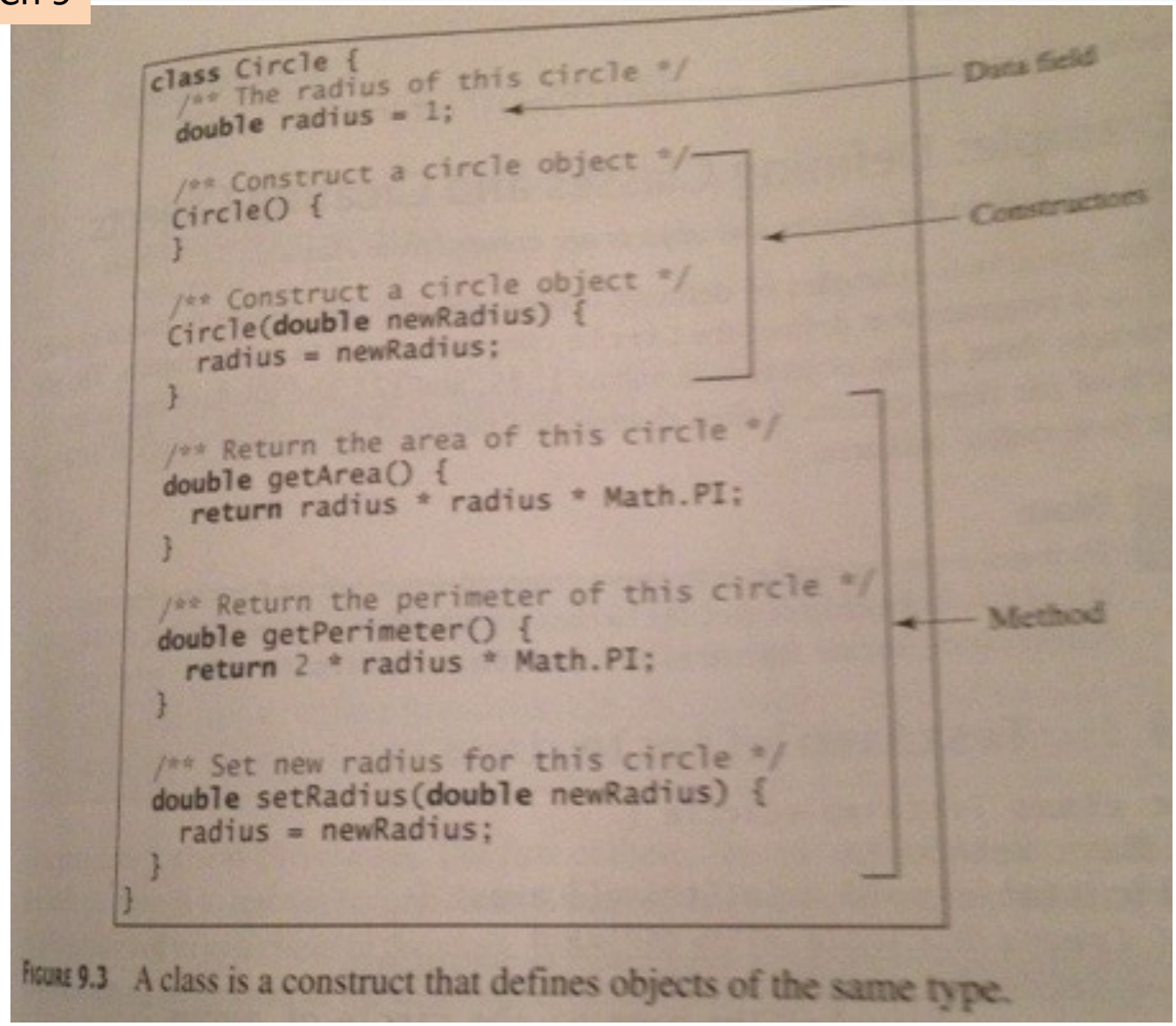


FIGURE 9.3 A class is a construct that defines objects of the same type.

OOP – Circle Code

Ch 9

main

```
3  public class circles {
4  public static boolean $DEBUG = true; //global
5  public static void main(String[] args) {
6      double radx, area, circum;
7      if ($DEBUG) System.out.println("Hello World\n");
8      Circle1 Cls1 = new Circle1();
9      if ($DEBUG) System.out.println("print circle 1");
10     radx = Cls1.radius;
11     area = Cls1.getArea();
12     circum = Cls1.getCircum();
13     printCircle(radx, area, circum)
14     //now supply a new radius=25
15     Circle1 Cls2 = new Circle1(25);
16     if ($DEBUG) System.out.println("print circle 2");
17     radx = Cls2.radius;
18     area = Cls2.getArea();
19     circum = Cls2.getCircum();
20     printCircle(radx, area, circum);
21     } //end main
```

Circle1: radius=1

method to print

Circle2: radius=25

method to print

OOP – Circle Code


```
32 class Circle1 {
33 public static boolean $DEBUG = true;
34 static double radius;
35 //constructors
36 Circle1() {
37 radius = 1;
38 }
39 Circle1(double newRadius) {
40 radius = newRadius;
41 }
42 //methods
43 double getArea() {
44    return radius*radius *Math.PI;
45 }
46 double getCircum() {
47    return (radius+radius) *Math.PI;
48 }
49 } //end Circle1
```

Circle1: **NO arg**, radius=1

Circle1: **1 arg**, radius=value passed

OOP – Circle Code

method to print

```
22      //method to print
23      public static void printCircle(double rr, double aa, double cc) {
24      System.out.println("area of circle with radius=" +
25      rr + " is " + aa);
26      System.out.println("circumference of circle with radius=" +
27      rr + " is " + cc);
28      System.out.println("");//blank line
29  } //end printCircle
30  } //end circles
31
```

```
print circle 1
area of circle with radius=1.0 is 3.141592653589793
circumference of circle with radius=1.0 is 6.283185307179586

print circle 2
area of circle with radius=25.0 is 1963.4954084936207
circumference of circle with radius=25.0 is 157.07963267948966
```

OOP – Reference Vars

- Garbage Collection

by the Java Virtual Machine (this process is called *garbage collection*).

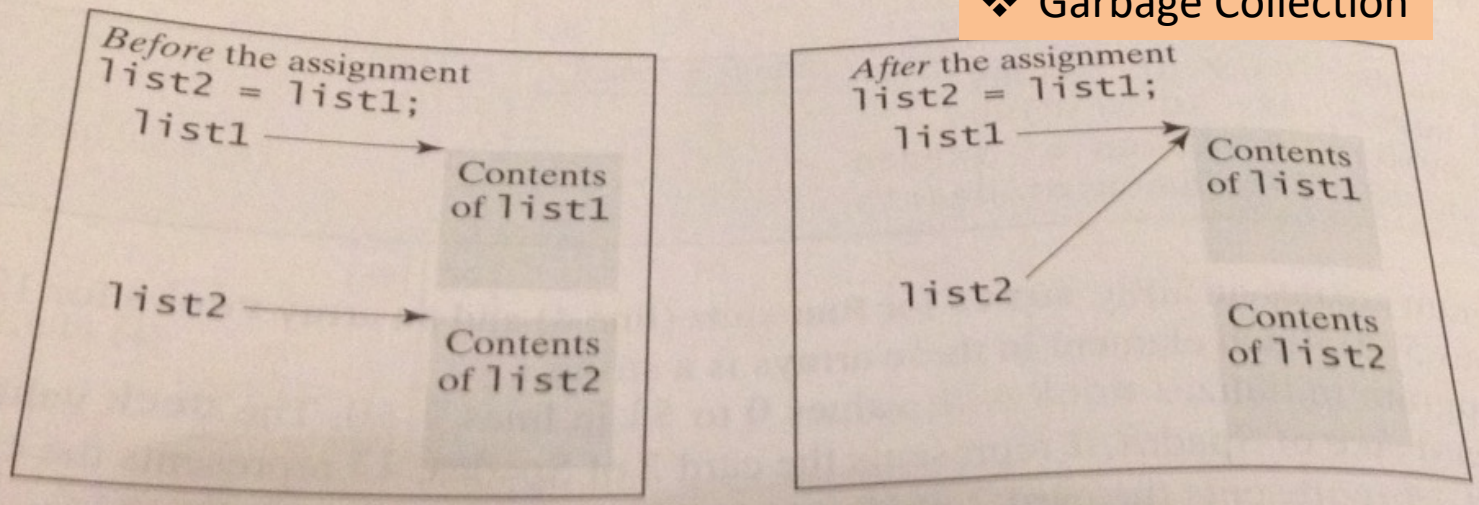


FIGURE 7.4 Before the assignment statement, `list1` and `list2` point to separate memory locations. After the assignment, the reference of the `list1` array is passed to `list2`.

OOP – Reference Vars

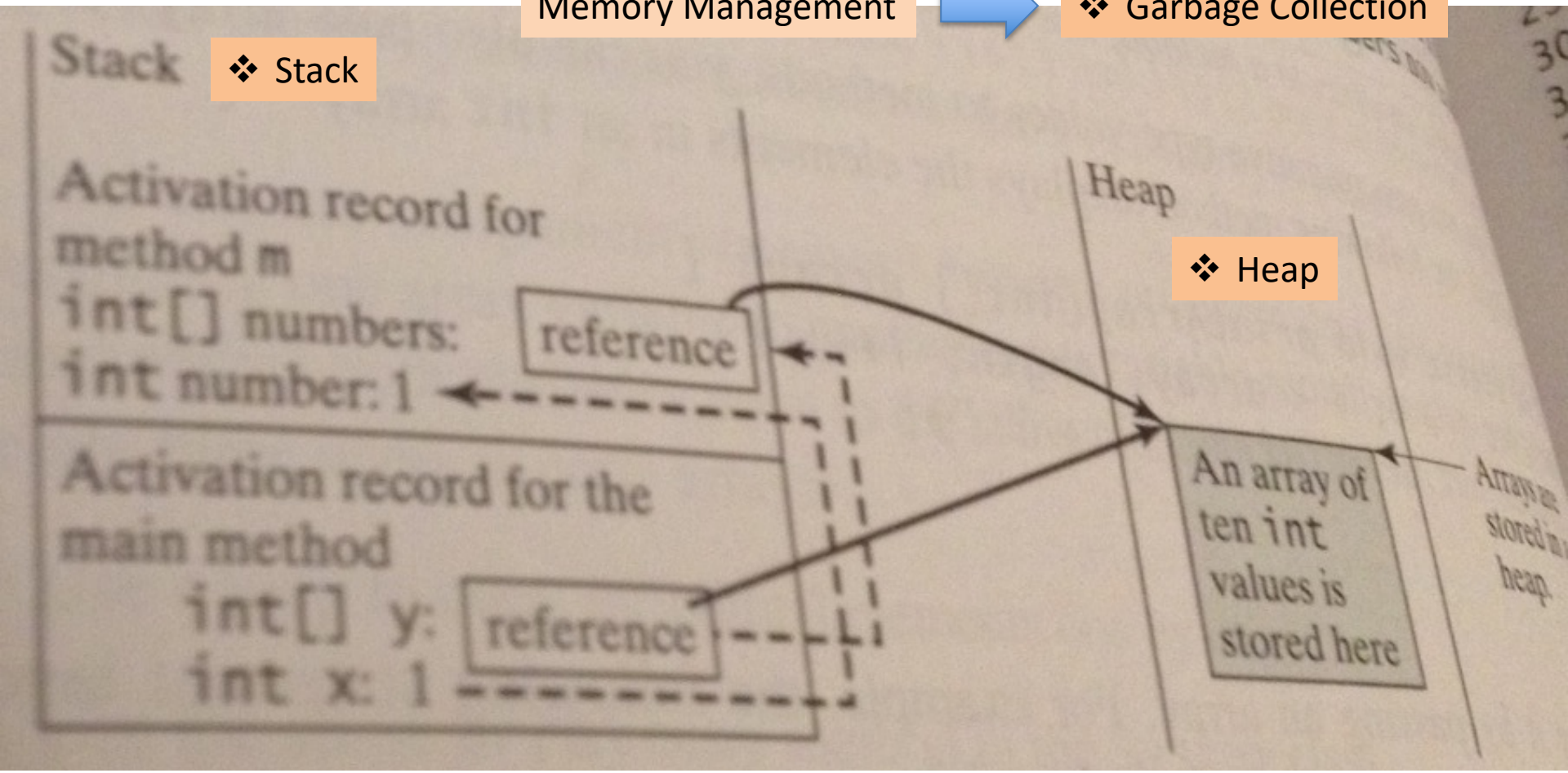
Sec 9.5

Arrays as a Class

Memory Management → ❖ Garbage Collection

❖ Stack

❖ Heap



OOP – Reference Vars

- Garbage Collection

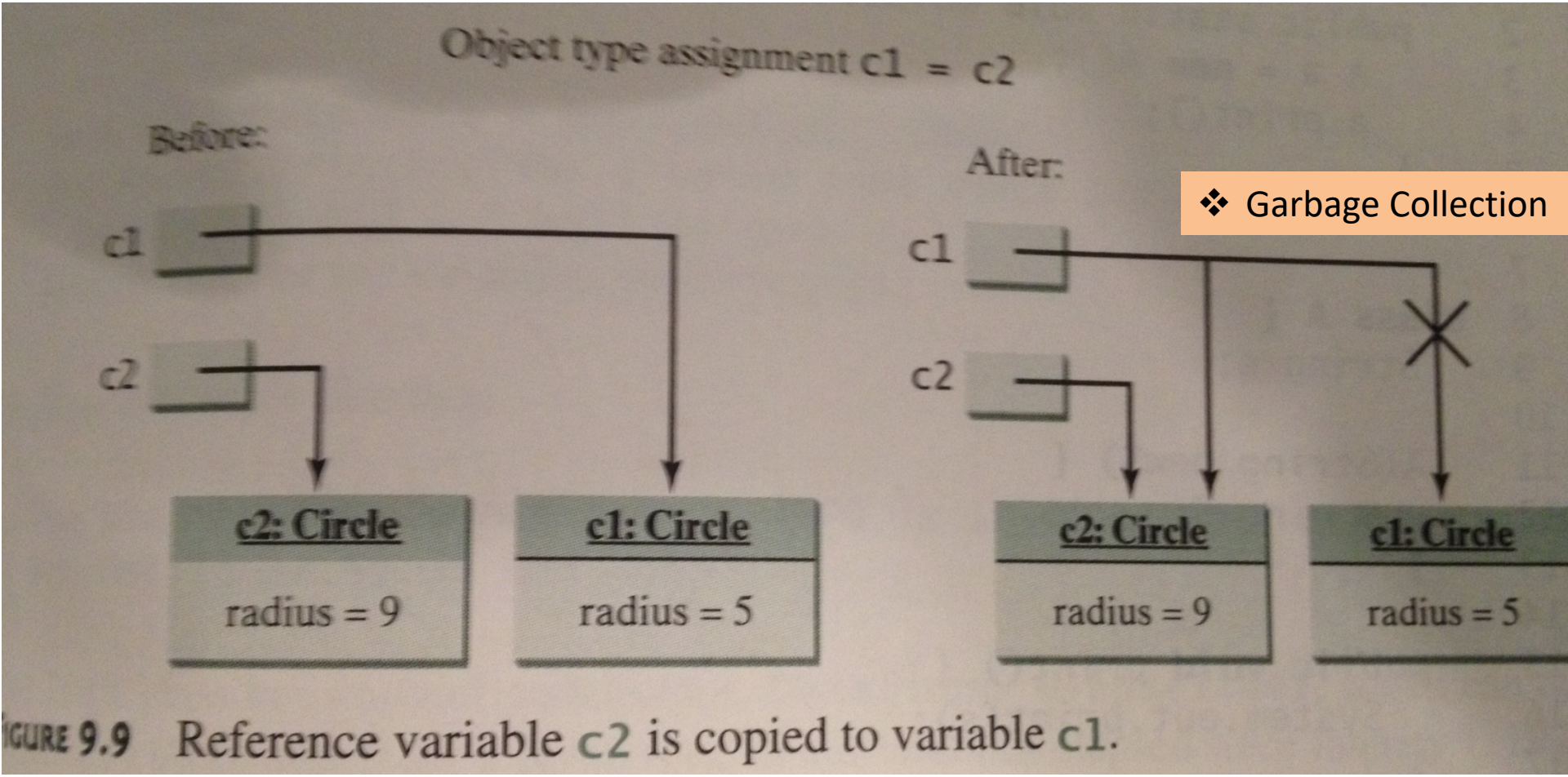


FIGURE 9.9 Reference variable c2 is copied to variable c1.

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP110

Java Class Library: Math

DR JEFF SOFTWARE
INDIE APP DEVELOPER

Sec 9.6

https://docs.oracle.com/javase/9/docs/api/java/lang/Math.html

OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES SEAR

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Field Summary

Fields

Modifier and Type	Field	Description
static double	E	The double value that is closer than any other to *e*, the base of the natural logarithms.
static double	PI	The double value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Method Summary

All Methods | **Static Methods** | **Concrete Methods**

Modifier and Type	Method	Description
static double	abs(double a)	Returns the absolute value of a double value.
static float	abs(float a)	Returns the absolute value of a float value.
static int	abs(int a)	Returns the absolute value of an int value.
static long	abs(long a)	Returns the absolute value of a long value.
static double	acos(double a)	Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*.
static int	addExact(int x, int y)	Returns the sum of its arguments, throwing an exception if the result overflows an int.
static long	addExact(long x, long y)	Returns the sum of its arguments, throwing an exception if the result overflows a long.
static double	asin(double a)	Returns the arc sine of a value; the returned angle is in the range -*pi*/2 through *pi*/2.

Random *Class*

Sec 9.6

java.util.Random	
+Random()	Constructs a Random object with the current time as its seed.
+Random(seed: long)	Constructs a Random object with a specified seed.
+nextInt(): int	Returns a random int value.
+nextInt(n: int): int	Returns a random int value between 0 and n (excluding n).
+nextLong(): long	Returns a random long value.
+nextDouble(): double	Returns a random double value between 0.0 and 1.0 (excluding 1.0).
+nextFloat(): float	Returns a random float value between 0.0F and 1.0F (excluding 1.0F).
+nextBoolean(): boolean	Returns a random boolean value.

FIGURE 9.11 A Random object can be used to generate random values.

When you create a Random object, you have to specify a seed or use the default seed. A seed is a number used to initialize a random number generator. The no-arg constructor creates a Random object using the current elapsed time as its seed. If two Random objects have the same seed, they will generate identical sequences of numbers. For example, the following code creates two Random objects with the same seed, 3:

```
Random generator1 = new Random(3);
System.out.print("From generator1: ");
for (int i = 0; i < 10; i++)
  System.out.print(generator1.nextInt(1000) + " ");

Random generator2 = new Random(3);
System.out.print("\nFrom generator2: ");
for (int i = 0; i < 10; i++)
  System.out.print(generator2.nextInt(1000) + " ");
```

The code generates the same sequence of random int values:

```
From generator1: 734 660 210 581 128 202 549 564 459 961
From generator2: 734 660 210 581 128 202 549 564 459 961
```

Random *Class*


```
18      //Random number generation using "random" method
19      int N = 10;
20      while(run) {
21         double randFlt = Math.random();
22         System.out.println("Float number= " + randFlt);
23         double rand10Flt = N * randFlt;
24         System.out.println("10x Float number= " + rand10Flt);
25         int rand10 = (int)rand10Flt;
26         System.out.println("10x Integer= " + rand10);
27      //Random number generation using "Random" Class
28         long seed = 3;
29         Random Rand1 = new Random();//default seed
30         int num1 = Rand1.nextInt(10);//use as seed
31         Random Rand2 = new Random(seed);//using seed=3
32         Random Rand3 = new Random(num1);//using random seed
33         int num2 = Rand2.nextInt(10);
34         int num3 = Rand3.nextInt(10);
35         int num4 = Rand1.nextInt(10);//2nd random number
36         System.out.println("Class NO seed=" +num1);
37         System.out.println("Class with seed of 3=" +num2);
38         System.out.println("Class with random seed=" +num3);
39         System.out.println("Class NO seed=" +num4);
40         int cont = JOptionPane.showConfirmDialog(null, "continue?");
41            switch (cont) {
42               case 0: System.out.println("keep going...");
43                  break;
44               default: System.out.println("good-bye!");
45                  run = false; //terminate loop
46            }//end switch
```

Random *Class*


```
----jGRASP exec: java Rand
debug: starting code
Float number= 0.7008979867065662
10x Float number= 7.008979867065662
10x Integer= 7
Class NO seed=6
Class with seed of 3=4
Class with random seed=1
Class NO seed=2
good-bye!
```

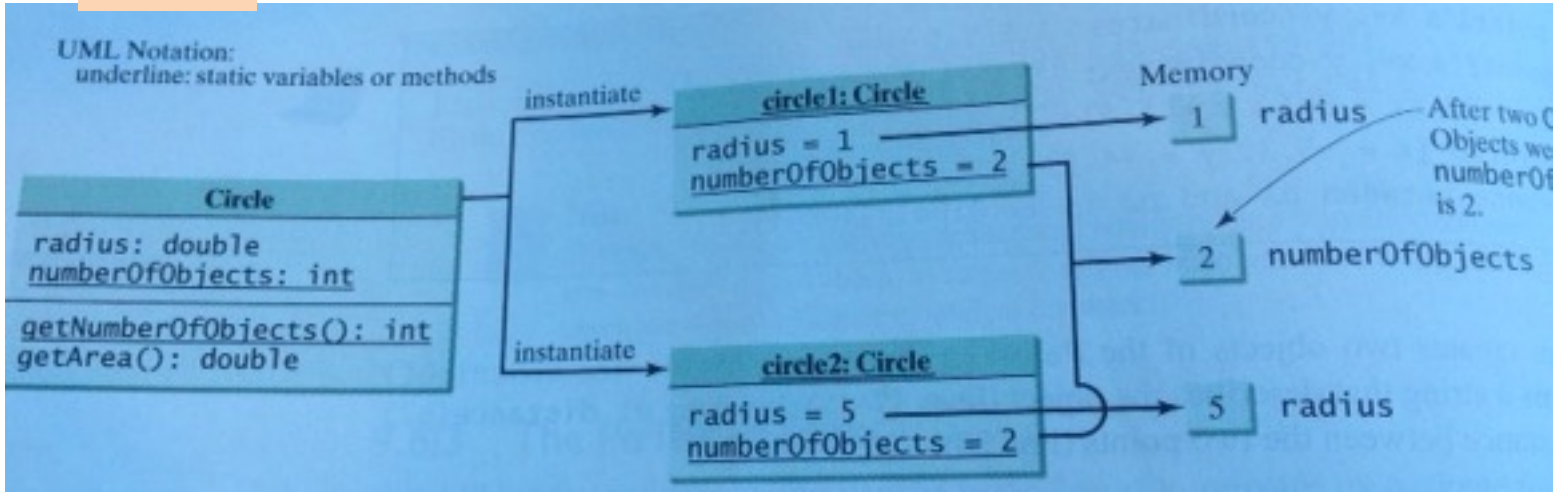
```
----jGRASP exec: java Rand
debug: starting code
Float number= 0.1290705780344708
10x Float number= 1.290705780344708
10x Integer= 1
Class NO seed=5
Class with seed of 3=4
Class with random seed=7
Class NO seed=7
good-bye!
```

Scope

- Persistence
  - Instance (default)
  - *Static*
- Privacy
  - *Public*
  - *Private*
  - *Protected (Ch 11)*
  - Default

Static Scope

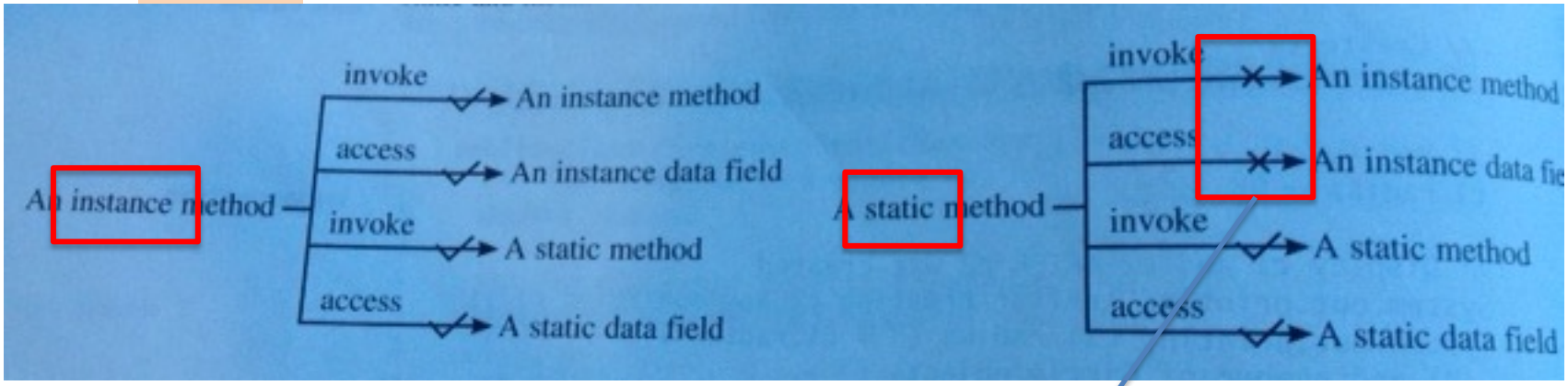
Sec 9.7




```
Before creating objects
The number of Circle objects is 0
After creating c1
c1: radius (1.0) and number of Circle objects (1)
After creating c2 and modifying c1
c1: radius (9.0) and number of Circle objects (2)
c2: radius (5.0) and number of Circle objects (2)
```

Static v Instance

Sec 9.7




```
1  public class A {
2    int i = 5;
3    static int k = 2;
4
5    public static void main(String[] args) {
6      A a = new A();
7      int j = a.i; // OK, a.i accesses the object's instance variable
8      a.m1(); // OK. a.m1() invokes the object's instance method
9    }
10
11   public void m1() {
12     i = i + k + m2(i, k);
13   }
14
15   public static int m2(int i, int j) {
16     return (int)(Math.pow(i, j));
17   }
18 }
```

- Refutes diagram in book

OOP – Circles

Sec 9.7

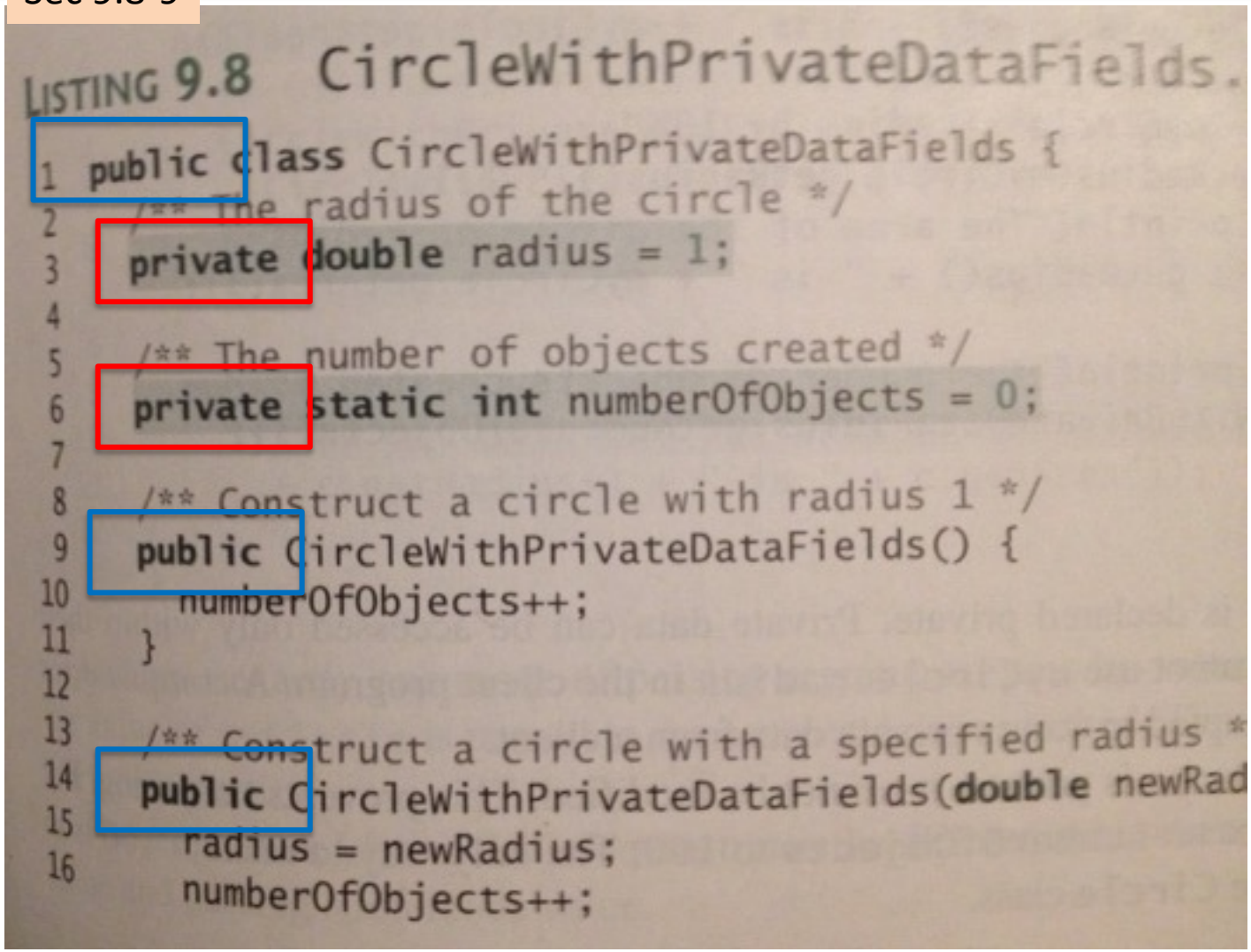
```
1  /*circles classes
2  Dr Jeff */
3  public class circles {
4  public static boolean $DEBUG = true; //global
5  public static void main(String[] args) {
6      //*test/debug
7      if ($DEBUG) System.out.println("Hello World\n");
8      circle1 c11 = new circle1();
9      System.out.println("area of circle with radius=" +
10     circle1.radius + "is " + circle1.getArea());
11     //now supply a new radius=25
12     circle1 c12 = new circle1(25);
13     System.out.println("area of circle with radius=" +
14     circle1.radius + "is " + circle1.getArea());
15
16     } //end main
17 } //end circles
```

OOP – Circles


```
19 class circle1 {
20 public static boolean $DEBUG = true;
21 static double radius;
22 //constructors
23 circle1() {
24 radius = 1;
25 }
26 circle1(double newRadius) {
27 radius = newRadius;
28 }
29 //methods
30 double getArea() {
31    return radius*radius *Math.PI;
32 }
33 double getCircum() {
34    return (radius+radius) *Math.PI;
35 }
36 } //end circle1
```

OOP – Private

Sec 9.8-9



LISTING 9.8 CircleWithPrivateDataFields.

```
1  public class CircleWithPrivateDataFields {
2    /** The radius of the circle */
3    private double radius = 1;
4
5    /** The number of objects created */
6    private static int numberOfObjects = 0;
7
8    /** Construct a circle with radius 1 */
9    public CircleWithPrivateDataFields() {
10     numberOfObjects++;
11   }
12
13   /** Construct a circle with a specified radius *
14   public CircleWithPrivateDataFields(double newRad
15     radius = newRadius;
16     numberOfObjects++;
```

Private in Circles


```
3  public class circles {
4     public static boolean $DEBUG = true; //global
5     public static void main(String[] args) {
6        double radx, area, circum;
7        if ($DEBUG) System.out.println("Hello World\n");
8        Circle1 Cls1 = new Circle1();
9        if ($DEBUG) System.out.println("print circle 1");
10       radx = Cls1.radius;
11       area = Cls1.getArea();
```

```
32 class Circle1 {
33    public static boolean $DEBUG = true;
34    private double radius;
```

```
circles.java:10: error: radius has private access in Circle1
      radx = Cls1.radius;
                 ^
circles.java:17: error: radius has private access in Circle1
      radx = Cls2.radius;
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP110

OOP – Getters & Setters

DR JEFF SOFTWARE
INDIE APP DEVELOPER

Sec 9.9

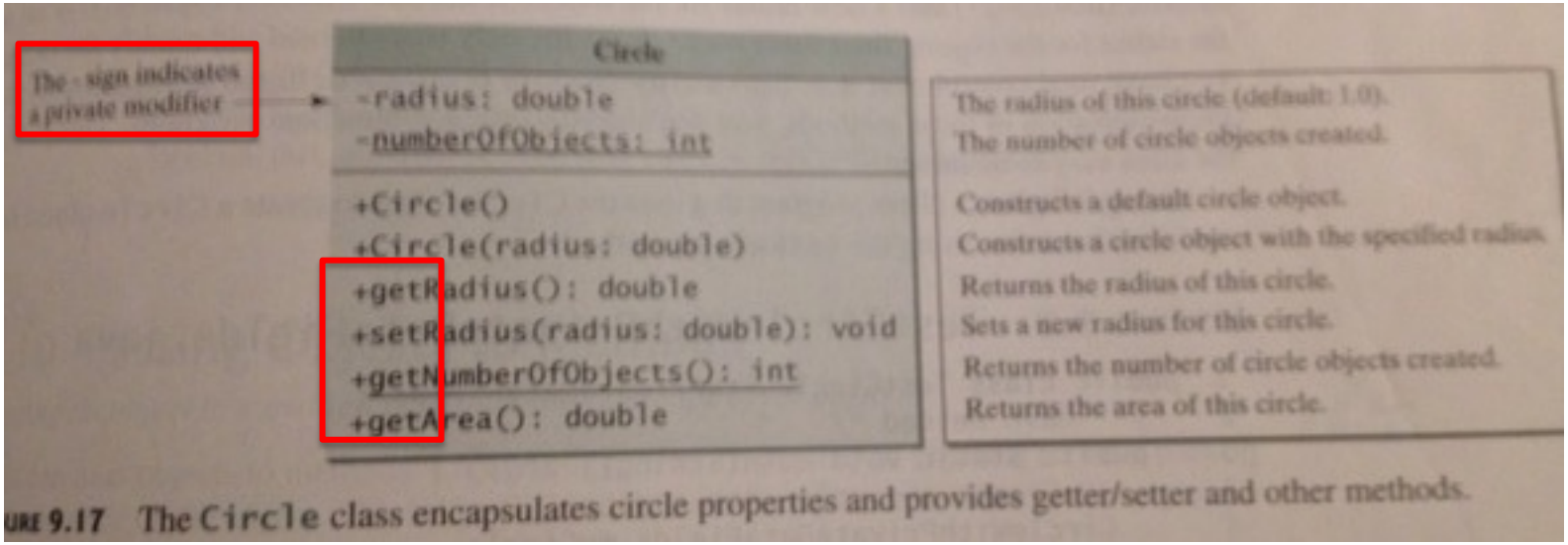
```
public returnType getPropertyName()
```

If the returnType is boolean, the getter method should be defined as

```
public boolean isPropertyName()
```

A setter method has the following signature:

```
public void setPropertyName(dataType propertyValue)
```



URE 9.17 The Circle class encapsulates circle properties and provides getter/setter and other methods.

❖ To access Private variables

Circles: Private + Getter

Sec 9.9

```
3  public class circles {
4    public static boolean $DEBUG = true; //global
5    public static void main(String[] args) {
6      double radx, area, circum;
7      if ($DEBUG) System.out.println("Hello World\n");
8      Circle1 Cls1 = new Circle1();
9      if ($DEBUG) System.out.println("print circle 1");
0      radx = Cls1.getRadius(); //changed to getter
```

```
32 class Circle1 {
33   public static boolean $DEBUG = true;
34   private double radius; //private,add getter
42   //methods
43   double getRadius() { //added getter
44     return radius;
```

```
print circle 1
area of circle with radius=1.0 is 3.141592653589793
circumference of circle with radius=1.0 is 6.283185307179586

print circle 2
area of circle with radius=25.0 is 1963.4954084936207
circumference of circle with radius=25.0 is 157.07963267948966
```

Best Practices

Classes

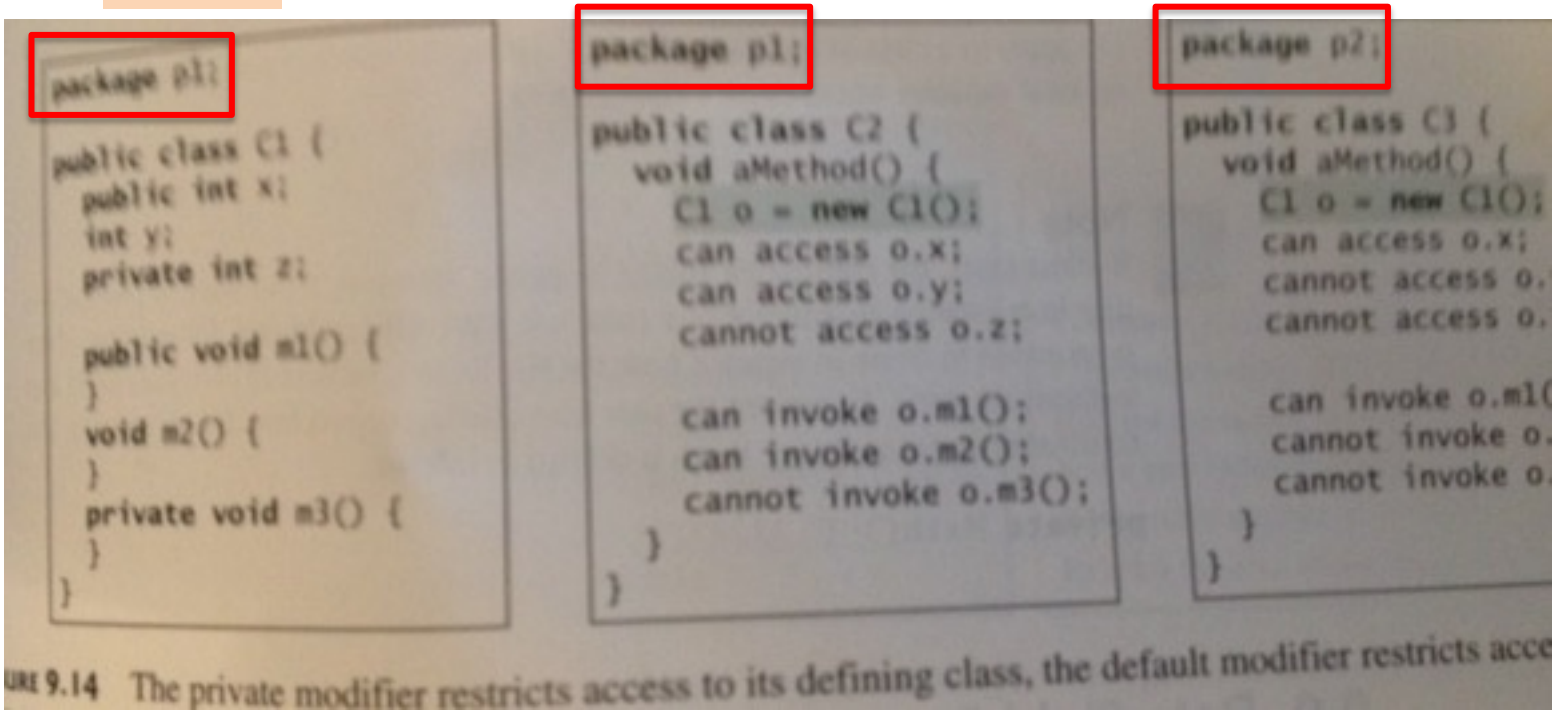
```
/*
 * Optional class specific comment
 *
 */
public class SomeClass {
  // Static variables in order of visibility
  public static final Integer PUBLIC_COUNT = 1;
  static final Integer PROTECTED_COUNT = 1;
  private static final Integer PRIVATE_COUNT = 1;


  // Instance variables in order of visibility
  public String name;
  String postalCode;
  private String address;


  // Constructor and overloaded in sequential order
  public SomeClass() {}


  public SomeClass(String name) {
    this.name = name;
  }


  // Methods
  public String doSomethingUseful() {
    return "Something useful";
  }


  // getters, setters, equals, hashCode and toString at the end
}
```

Immutable


```
public class ImmutableObject {
    private final int primitiveField;
    private final Object objectField;

public ImmutableObject(int p, Object o) {
        primitiveField = p;
        objectField = new Object(); // Clone
        objectField.setField(o.getField()); // Copy
    }

}
```

OOP – Packages

Sec 9.8



```
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
package p1;

public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

```
package p2;

public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.
    cannot access o.

    can invoke o.m1(
    cannot invoke o.
    cannot invoke o.
  }
}
```

URE 9.14 The private modifier restricts access to its defining class, the default modifier restricts acce

- Packages ::= Groups of Classes
  - Scope encapsulation

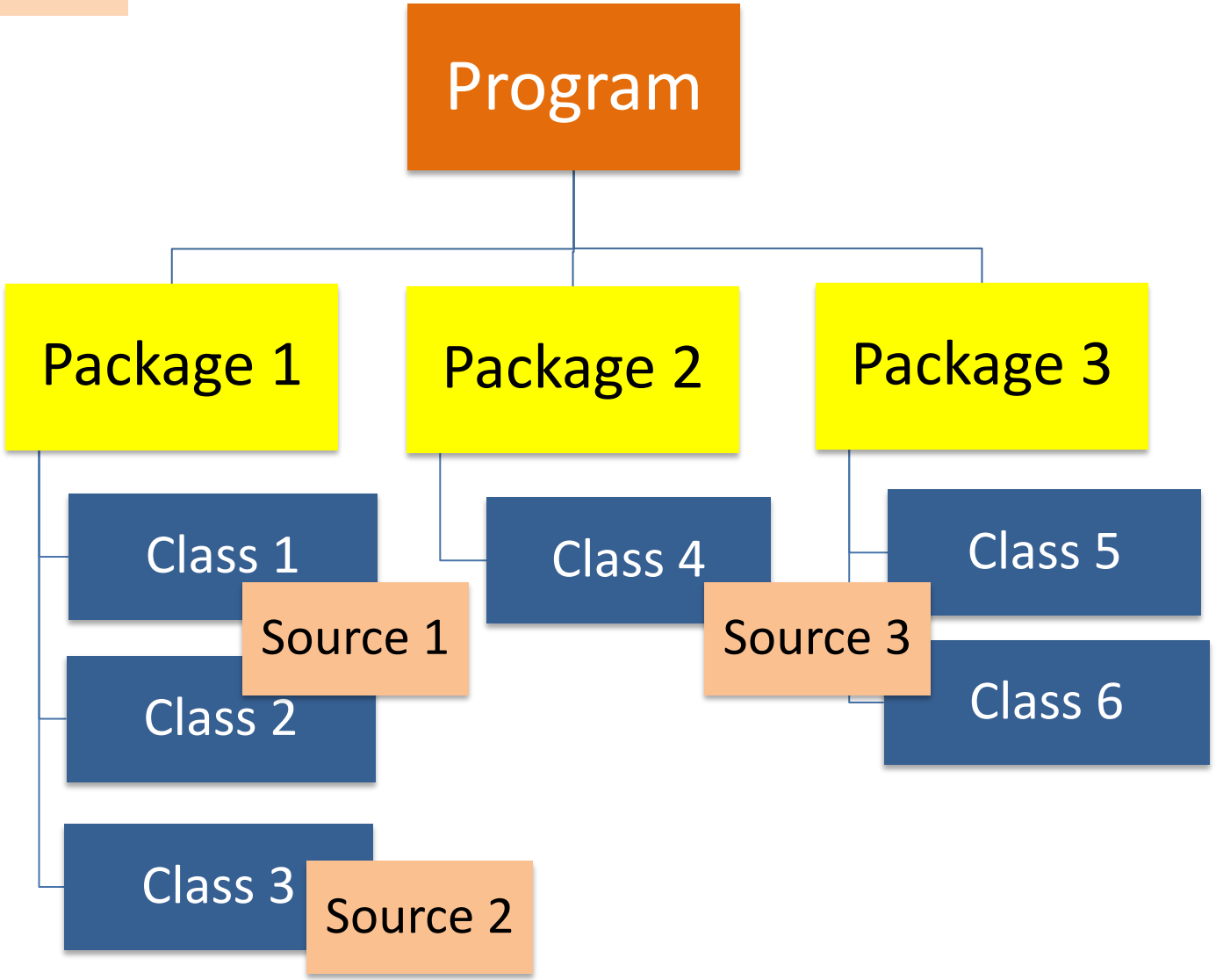
Scope Rules for Packages

TABLE 11.2 Data and Methods Visibility

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass in a different package	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

```
package p1;

public class C1 {
  public int x;
  protected int y;
  int z;
  private int u;

  protected void m() {
  }
}

public class C2 {
  C1 o = new C1();
  can access o.x;
  can access o.y;
  can access o.z;
  cannot access o.u;

  can invoke o.m();
}

package p2;

public class C3
      extends C1 {
  can access x;
  can access y;
  can access z;
  cannot access u;

  can invoke m();
}

public class C4
      extends C1 {
  can access x;
  can access y;
  cannot access z;
  cannot access u;

  can invoke m();
}
```

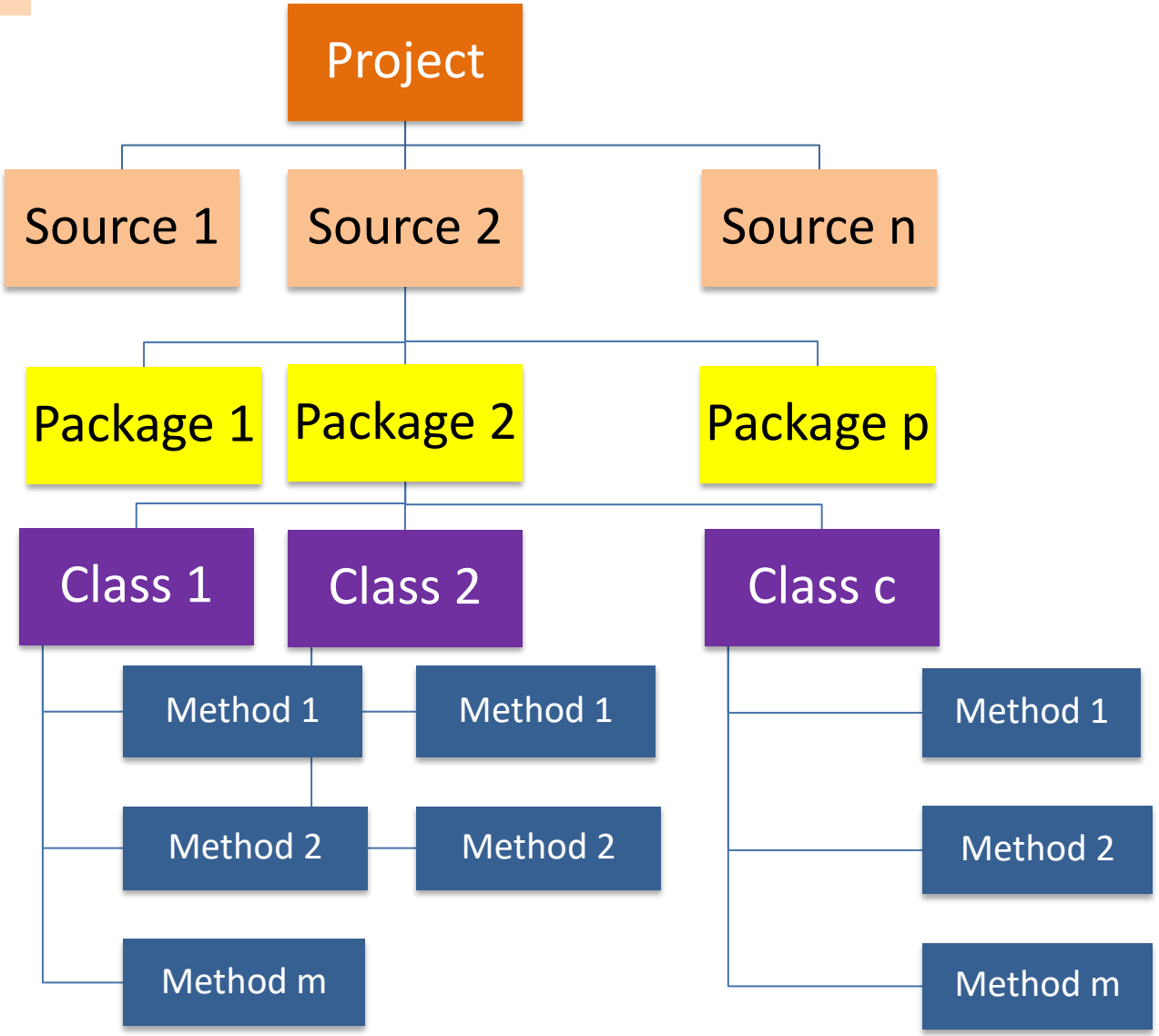
OOP – Packages



CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP110

OOP – Packages

Sec 9.8



Public Classes


```
11    //main
12      public static void main(String[] args) {
13        //debug
14        if ($DEBUG) System.out.println("starting code");
15        //**test code here
16
17      } //end main & class
18    }
19    //test class
20    public class F {
21      private int i = 5; //instance
22      private static double k = 0; //static
23
24      public void setI(int x) {
25        this.i = x;
26      }
27      public static void setK(double x) {
28        F.k = x;
29      }
30    }
```

- ***public*** classes have to be placed into a separate file (.java)

```
testThis.java:20: error: class F is public, should be declared in a
public class F {
       ^
                                                    in a file named F.java
1 error
```

Objects as Arguments


```
method1(int xxx) {
ClassName Cname = new ClassName ( );
method2(Cname);
}
void method2(ClassName parm1) {
// Cname object passed to method2 as parm1
```

- classes are *objects* and can be passed as arguments
- *Ref variable* (class pointer) passed "by value" is really passed "by reference"

Arrays of Objects


```
method1(int xxx) {
//instantiate array of objects
   ClassName [ ] Cname = new ClassName [ 5];

//initialize array of objects
   for (int i=0; i< Cname.length; i++)
      Cname[ i] = new ClassName( );
```

- Arrays can hold *objects* (as well as *primitive* data types)

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP110

Immutable Objects/Classes

DR JEFF SOFTWARE
INDIE APP DEVELOPER

Sec 9.12

- ***Immutable*** means variables (state) cannot be modified
- both ***Classes*** and their ***Objects*** can be ***Immutable***

➢ ***Immutable*** is a *property*, not a *keyword* (declaration)

Requirements

- All state variables are ***private***
- NO ***setters***
- NO references to ***functions***

CSUN CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP110

The *this* Reference

Sec 9.14

- ***this*** refers to the *calling* object (instance)

```
public class F {
  private int i = 5; //instance
  private static double k = 0; //static

  public void setI(int x) {
    i = x;
  }
public static void setK(double x) {
    k = x;
  }
```

```
2    CSUN class CS110
3    file:  testThis.java
4    */
5    import java.util.*;
6    import javax.swing.*;
7    import java.io.*;
8    //class
9    public class testThis {
10     static final boolean $DEBUG = true;
11   //main
12     public static void main(String[] args) {
13       //debug
14       if ($DEBUG) System.out.println("starting code");
15       int i=9;
16       double k=66;
17       //instantiate 2 classes/objects
18       F Fobj1 = new F();
19       Fobj1.setI(1);
20       System.out.println("main i=" + i);
21       Fobj1.setI(6);
22       F Fobj2 = new F();
23       Fobj2.setK(6);
24     } //end main & class
```

```
26   //test class
27   class F {
28     private int i = 5; //instance
29     private static double k = 0; //static
30
31       public void setI(int x) {
32         i = x;
33         System.out.println("setI.i=" + i);
34         //System.out.println("this.i=" + this.i);
35       }
36       public static void setK(double x) {
37         k = x;
38         System.out.println("F.k=" + k);
39       }
```

```
starting code
setI.i=1
private i=9
setI.i=6
F.k=6.0
```

The *this* Reference

- ***this*** refers to the *calling* object (instance)

```
public class F {
  private int i = 5; //instance
  private static double k = 0; //stat

  public void setI(int x) {
    i = x;
  }
public static void setK(double x) {
    k = x;
  }
```

```
17      //instantiate class/object
18      F Fobj1 = new F();
19      Fobj1.setI(1);
20      System.out.println("main i=" + i);
21      System.out.println("F.i=" + F.i);
22      Fobj1.setI(6);
23      System.out.println("setI.i=" + i);
24      //instantiate another class/object
25      F Fobj2 = new F();
26      Fobj2.setK(6);
27    } //end main & class
```

```
testThis.java:21: error: i has private access in F
        System.out.println("F.i=" + F.i);
                                     ^
testThis.java:22: error: i has private access in F
        System.out.println("Fobj1.i=" + Fobj1.i);
                                           ^
```

```
26    //test cl
27    class F
28      privat
29      privat
30
31      publi
32        i =
33        Sys
34        //S
35      }
36      public static void setK(double x) {
37        k = x;
38        System.out.println("F.k=" + k);
39      }
```

The *this* Reference

- ***this*** refers to the *calling* object (instance)

```
public class F {
  private int i = 5; //instance
  private static double k = 0; //

  public void setI(int x) {
    i = x;
  }
public static void setK(double x
    k = x;
  }
```

```
11 //main
12    public static void main(String[] args) {
13     //debug
14       if ($DEBUG) System.out.println("starting code");
15       int i=9;
16       double k=66;
17     //instantiate class/object
18     F Fobj1 = new F();
19     Fobj1.setI(4);
20     System.out.println("main i=" + i);
21     //System.out.println("F.i=" + F.i);
22     //System.out.println("Fobj1.i=" + Fobj1.i);
23     Fobj1.setK(33);
24     System.out.println("setK.k=" + Fobj1.k);
25     System.out.println("main k=" + k);
```

```
26  //test class
27  class F {
28    private int i = 5; //instance
29    private static double k = 0; //static
30
31      public void setI(int x) {
32        i = x;
33        System.out.println("setI.i=" + i);
34        //System.out.println("this.i=" + this.i);
35      }
36      public static void setK(double x) {
37        k = x;
38        System.out.println("F.k=" + k);
39      }
```

```
starting code
main i=9
F.k=33.0
setK.k=33.0
main k=66.0
F.k=6.0
```

CSUN CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP110

The *this* Reference

Sec 9.14

- ***this*** refers to the *calling* object (instance)

```
public class F {
  private int i = 5; //instance
  private static double k = 0; //s
  public void setI(int x) {
    i = x;
  }
public static void setK(double x
```

```
11 //main
12    public static void main(String[] args) {
13     //debug
14       if ($DEBUG) System.out.println("starting code");
15       int i=9;
16       double k=66;
17     //instantiate class/object
18     F Fobj1 = new F();
19     Fobj1.setI(4);
20     System.out.println("main i=" + i);
21     //System.out.println("F.i=" + F.i);
22     System.out.println("Fobj1.i=" + Fobj1.i);
```

```
32 //test class
33 class F {
34    int i = 5; //instance: this
35    static double k = 88; //static
36
37    public void setI(int x) {
38       System.out.println("setI.i=" + i);
39       int i=66; //local
40       System.out.println("setI.i=" + i);
41       System.out.println("this.i=" + this.i);
42       this.i = 77;
43    }
44    public static void setK(double x) {
45       k = x;
46       System.out.println("F.k=" + k);
47    }
```

this

```
starting code
setI.i=5
setI.i=66
this.i=5
main i=9
Fobj1.i=77
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP110

The *this* Reference

DR JEFF SOFTWARE
INDIE APP DEVELOPER

zyBook 11.11

Figure 11.11.1: Using 'this' to refer to an object's members.

ShapeSquare.java:

```
public class ShapeSquare {
   // Private fields
   private double sideLength;

   // Public methods
   public void setSideLength(double sideLength) {
      this.sideLength = sideLength;
      // Field member    Parameter
   }

   public double getArea() {
      return sideLength * sideLength; // Both refer to field
   }
}
```

ShapeTest.java:

```
public class ShapeTest {
   public static void main(String[] args) {
      ShapeSquare square1 = new ShapeSquare();

      square1.setSideLength(1.2);
      System.out.println("Square's area: " + square1.getArea());
   }
}
```

The *this* Reference

- ***this*** refers to the *calling* object (instance)
- ***this*** applied to *Construcotrs*

9.14.2 Using `this` to Invoke a Constructor

The `this` keyword can be used to invoke another constructor of the same class. For you can rewrite the `Circle` class as follows:

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }
  ...
}
```

The **this** keyword is used to reference the hidden data field radius of the object being constructed.

The **this** keyword is used to invoke another constructor.

The line `this(1.0)` in the second constructor invokes the first constructor with a value argument.

The *this* Reference

- ***this*** applied to *Constructors*

Figure 11.11.2: Calling overloaded constructor using this keyword.

```
public class ElapsedTime {
   private int hours;
   private int minutes;

   // Overloaded constructor definition
   public ElapsedTime(int timeHours, int timeMins) {
      hours   = timeHours;
      minutes = timeMins;
   }

   // Default constructor definition
   public ElapsedTime() {
      this(0, 0);
   }

   // Other methods ...
}
```

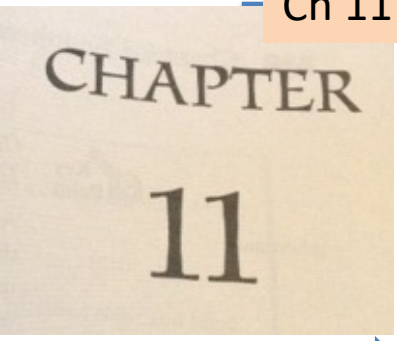
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP110

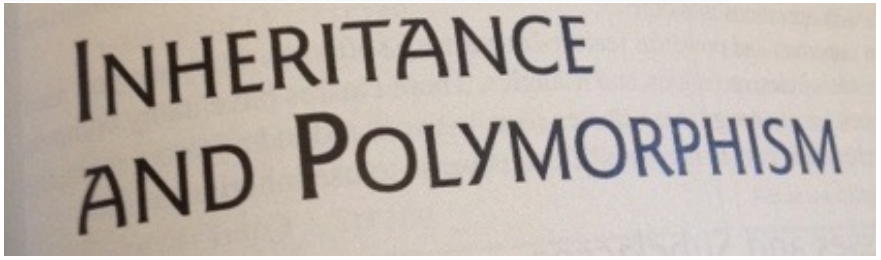
Chapter 11

DR JEFF SOFTWARE
INDIE APP DEVELOPER

Ch 11



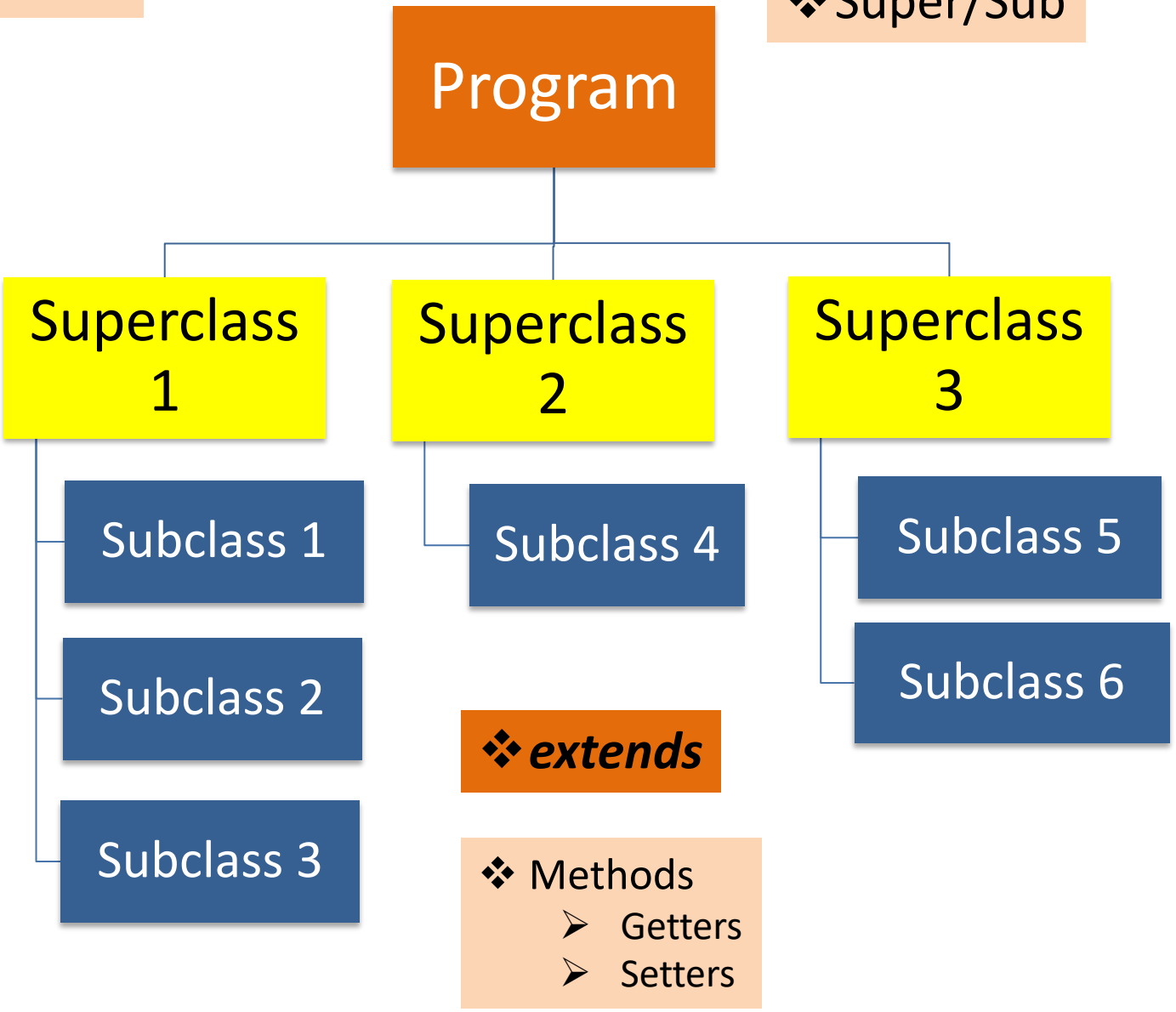
More Objects & Classes



1. Intro
2. Super & Sub Classes
3. ***super*** keyword
4. *Overriding* Methods
5. *Overriding* vs. *Overloading*
6. ***Object*** Class & ***toString( )*** Method
7. Polymorphism
8. Dynamic Binding
9. Casting Objects & ***instanceof*** Operator
10. Object's ***equals*** Method
11. ArrayList Class
12. Useful Methods for Lists
13. Case Study: Custom Stack Class
14. ***Protected*** Data & Methods
15. Preventing Extending & Overriding

Class Hierarchy

❖ Super/Sub



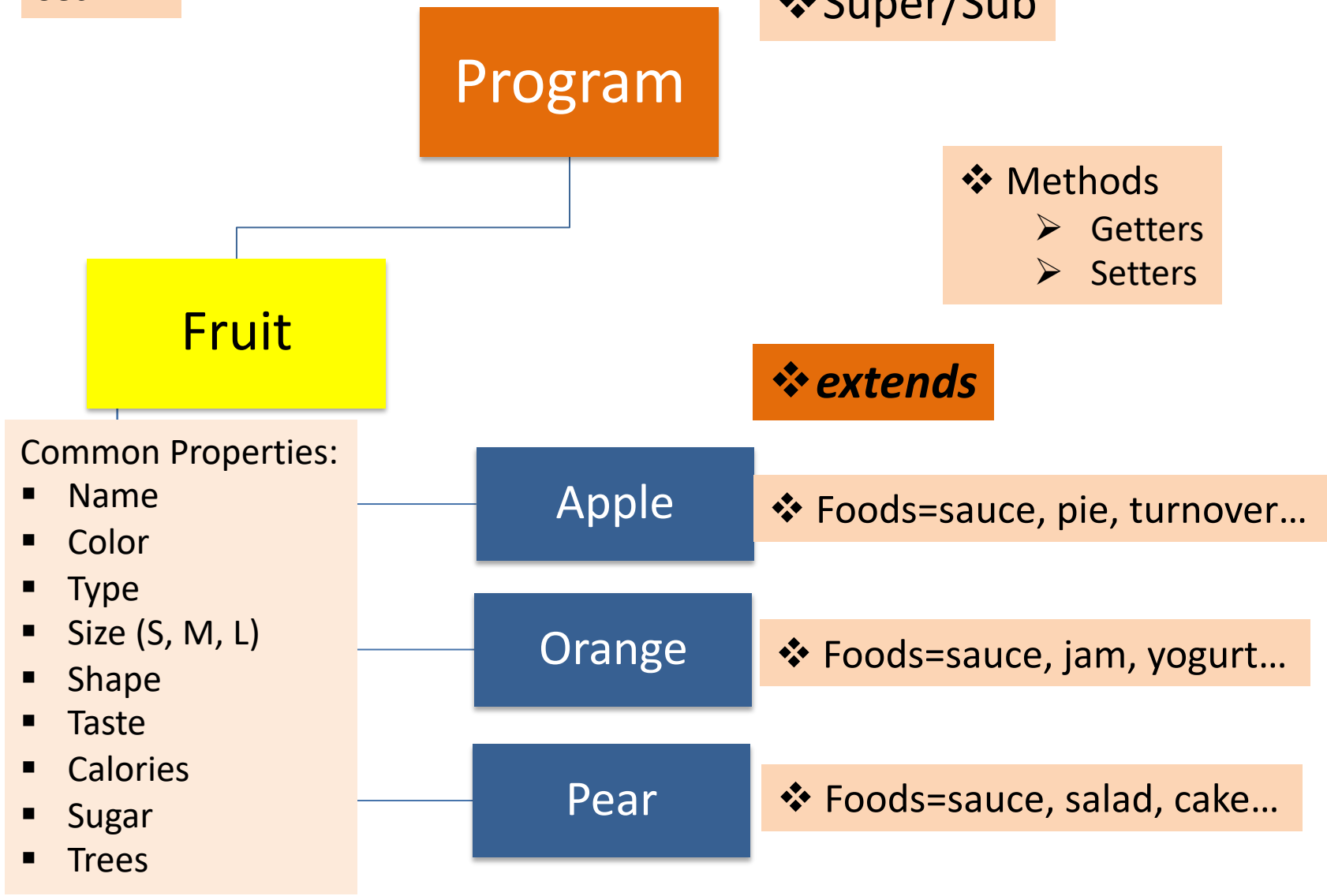
❖ ***extends***

❖ Methods
- ➢ Getters
- ➢ Setters

Class Example: Fruit

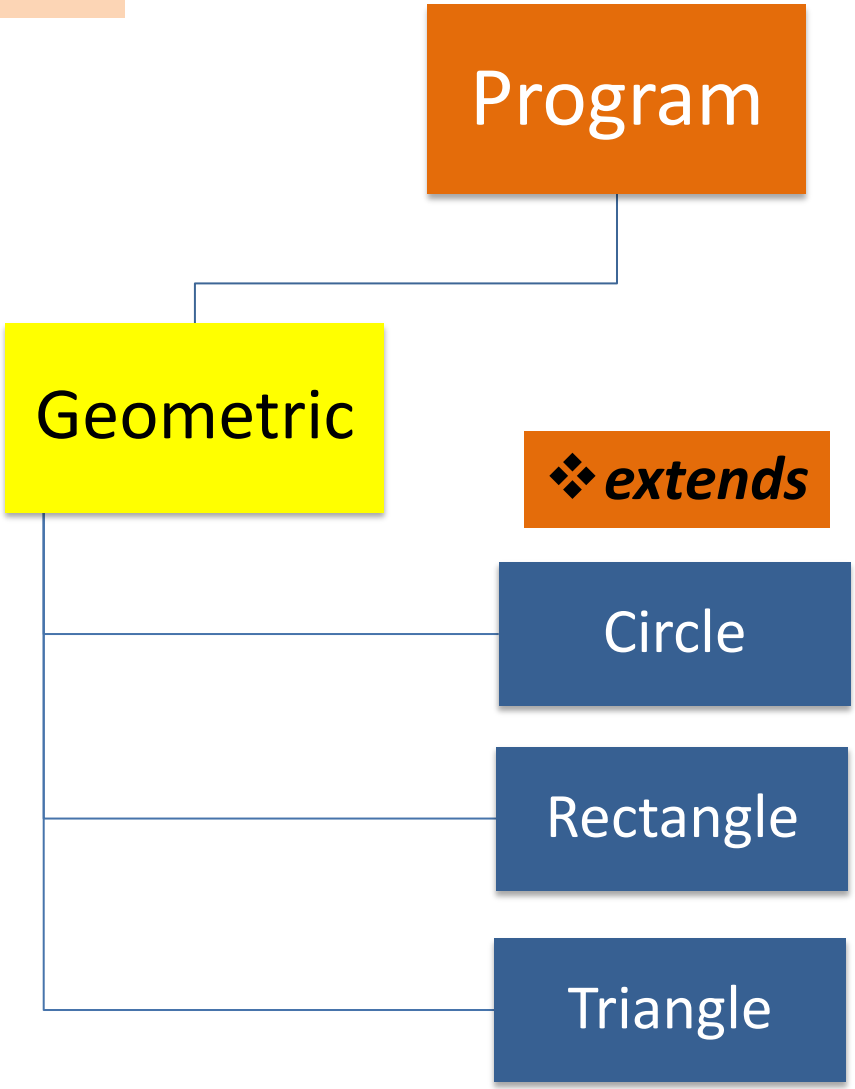
Sec 11.2

❖ Super/Sub



Class Example: Geometric

❖ Super/Sub

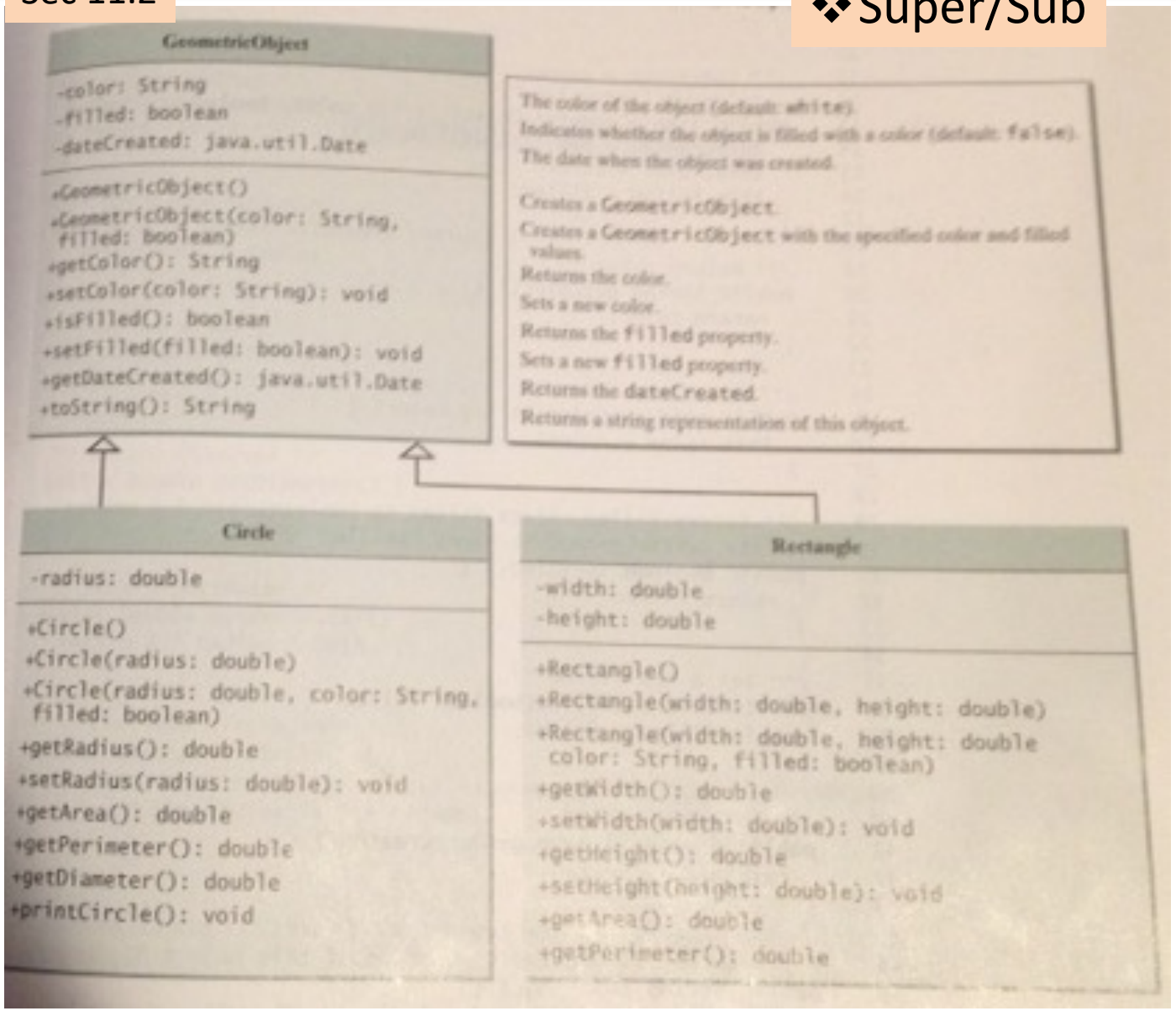


❖ Methods
- ➢ Getters
- ➢ Setters

Class Hierarchy

Sec 11.2

❖Super/Sub



Object Instantiation

❖ **circle** object

LISTING 11.2 CircleFromSimpleGeometricObject.java

```
1  public class CircleFromSimpleGeometricObject
2      extends SimpleGeometricObject {
3    private double radius;
4
5    public CircleFromSimpleGeometricObject() {
6    }
7
8    public CircleFromSimpleGeometricObject(double radius) {
9      this.radius = radius;
10   }
11
12   public CircleFromSimpleGeometricObject(double radius,
13       String color, boolean filled) {
14     this.radius = radius;
15     setColor(color);
16     setFilled(filled);
```

Object Instantiation

- **circle** object

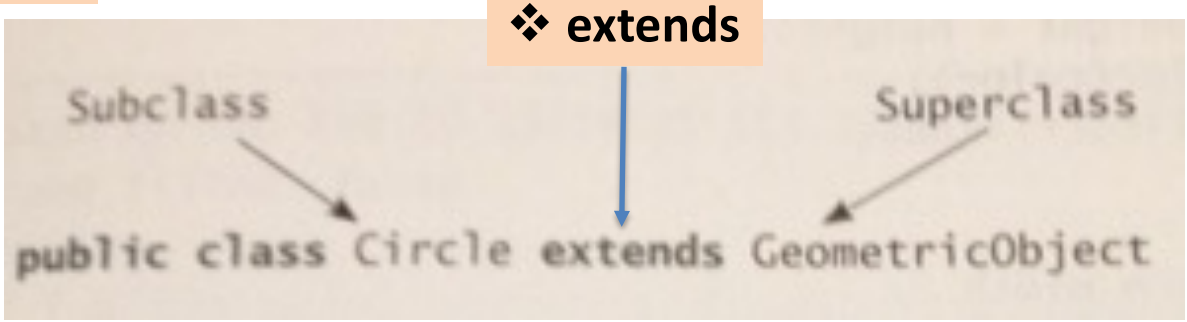
LISTING ... TestCircleRectangle.java

```
1  public class TestCircleRectangle {
2    public static void main(String[] args) {
3      CircleFromSimpleGeometricObject circle =
4       new CircleFromSimpleGeometricObject(1);
5      System.out.println("A circle " + circle.toString());
6      System.out.println("The color is " + circle.getColor());
7      System.out.println("The radius is " + circle.getRadius());
8      System.out.println("The area is " + circle.getArea());
9      System.out.println("The diameter is " + circle.getDiameter());
10
11     RectangleFromSimpleGeometricObject rectangle =
12       new RectangleFromSimpleGeometricObject(2, 4);
13     System.out.println("\nA rectangle " + rectangle.toString());
14     System.out.println("The area is " + rectangle.getArea());
15     System.out.println("The
16       rectangle.getPerimeter
```

```
A circle created on Thu Feb 10 19:54:25 EST 2011
color: white and filled: false
The color is white
The radius is 1.0
The area is 3.1415926535897931
The diameter is 2.0
A rectangle created on Thu Feb 10 19:54:25 EST 2011
color: white and filled: false
The area is 8.0
The perimeter is 12.0
```

Inheritance

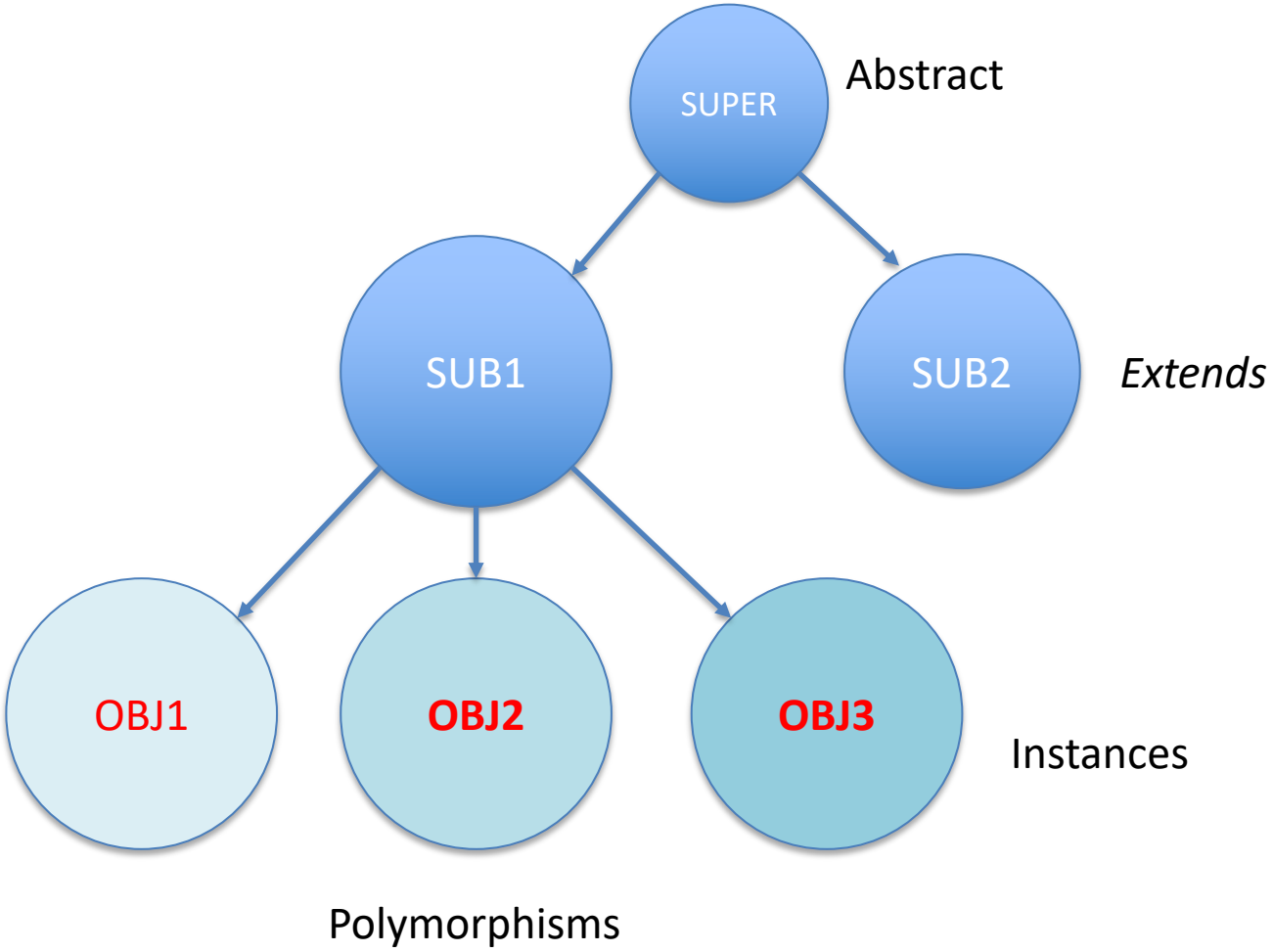
❖ **extends**



- ❖ Properties
- ❖ Methods

➢ but NOT constructors

Inheritance

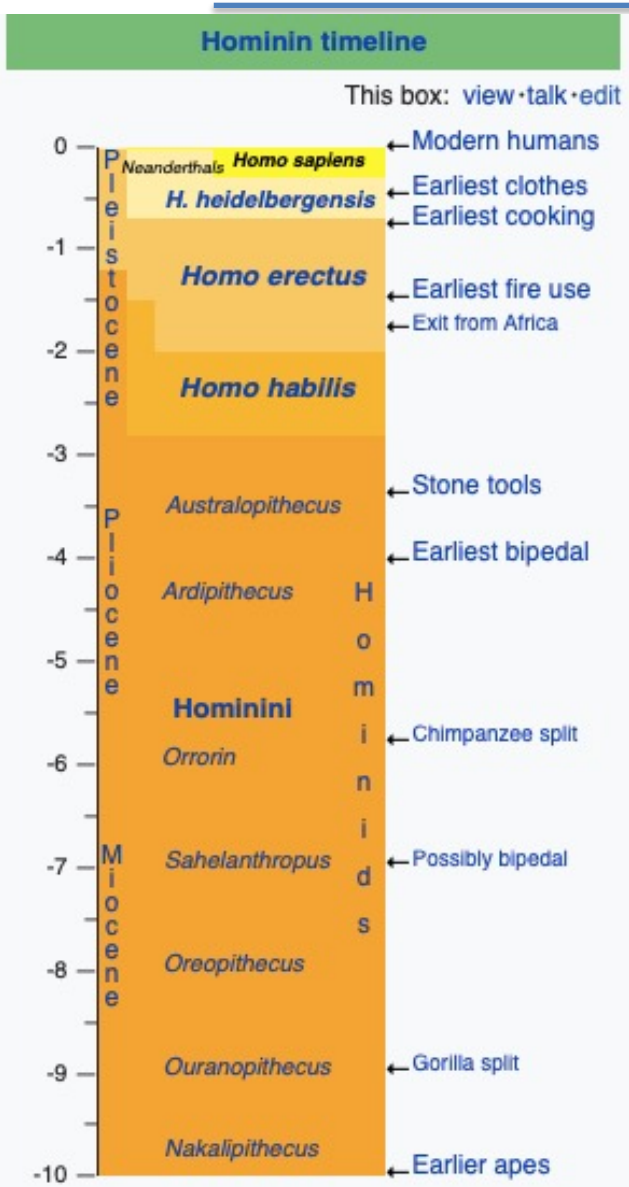


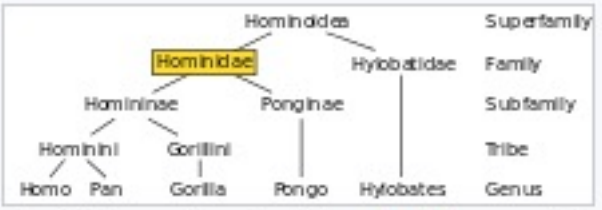
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP110

Human Biologic Taxonomy

DR JEFF SOFTWARE
INDIE APP DEVELOPER

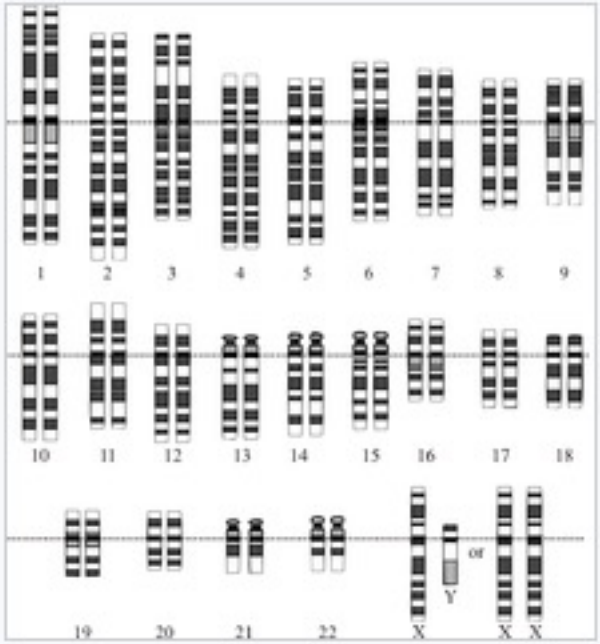




Family tree showing the extant hominoids: humans (genus *Homo*), chimpanzees and bonobos (genus *Pan*), gorillas (genus *Gorilla*), orangutans (genus *Pongo*), and gibbons (four genera of the family Hylobatidae: *Hylobates*, *Hoolock*, *Nomascus*, and *Symphalangus*). All except gibbons are hominids.

Inheritance in Evolution

- Human genome (DNA)
  - 3B base pairs (nucleotides)
    - A, C, G, T (base 4 code)
  - 23 chromosomes (2 pairs)
  - 22,000 genes
- *Polymorphisms*
  - SNP's
    - 0.1% (1/1000)
    - 3M SNP's

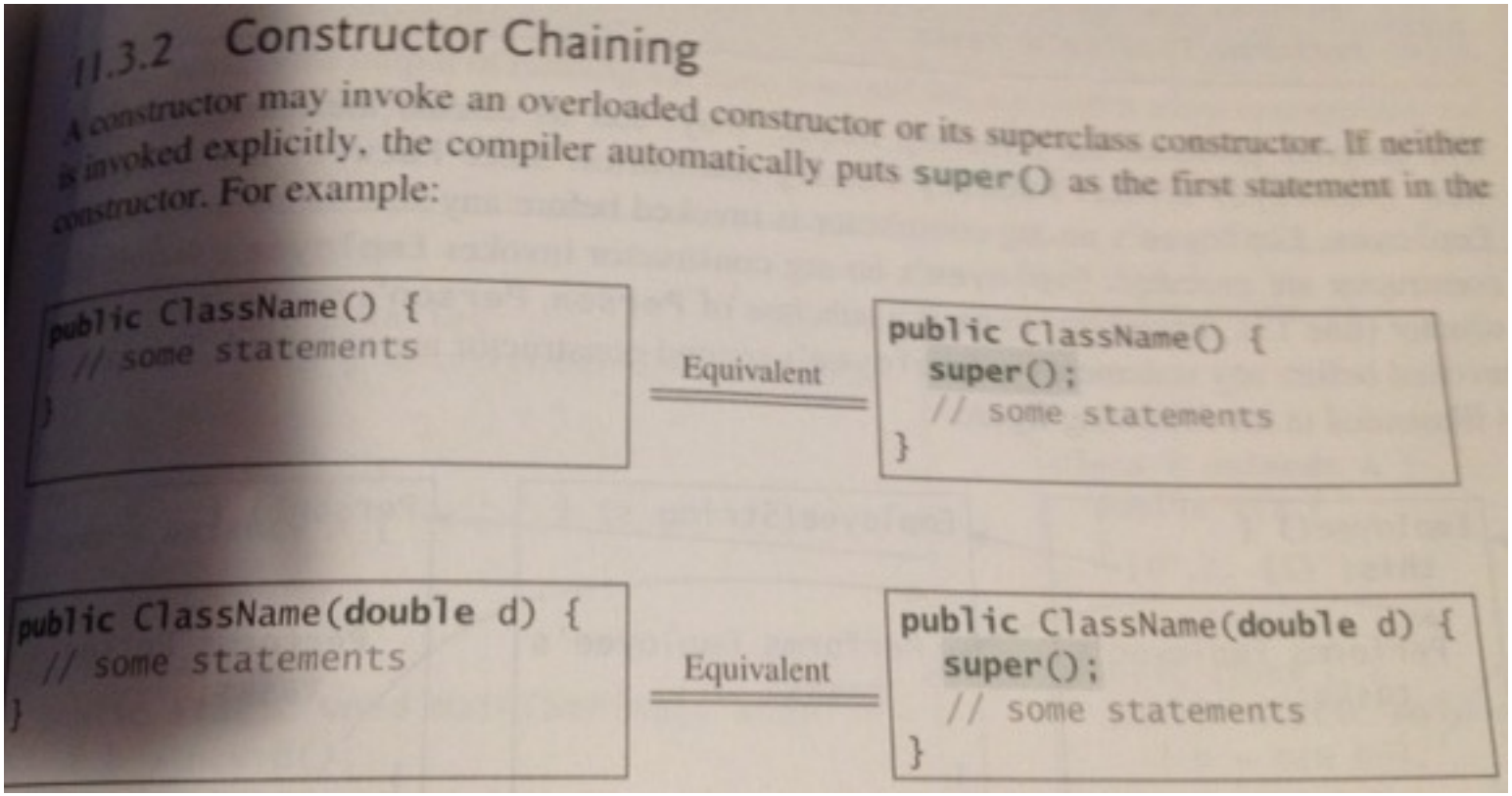


A graphical representation of the standard human karyotype, including both the male (XY) and female (XX) sex chromosomes.

Constructor Chaining

Sec 11.3

❖ **super**( ) constructor



11.3.2 Constructor Chaining

A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically puts `super()` as the first statement in the constructor. For example:

```
public ClassName() {
  // some statements
}
```

Equivalent

```
public ClassName() {
  super();
  // some statements
}
```

```
public ClassName(double d) {
  // some statements
}
```

Equivalent

```
public ClassName(double d) {
  super();
  // some statements
}
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP110

Overriding Methods

Sec 11.4

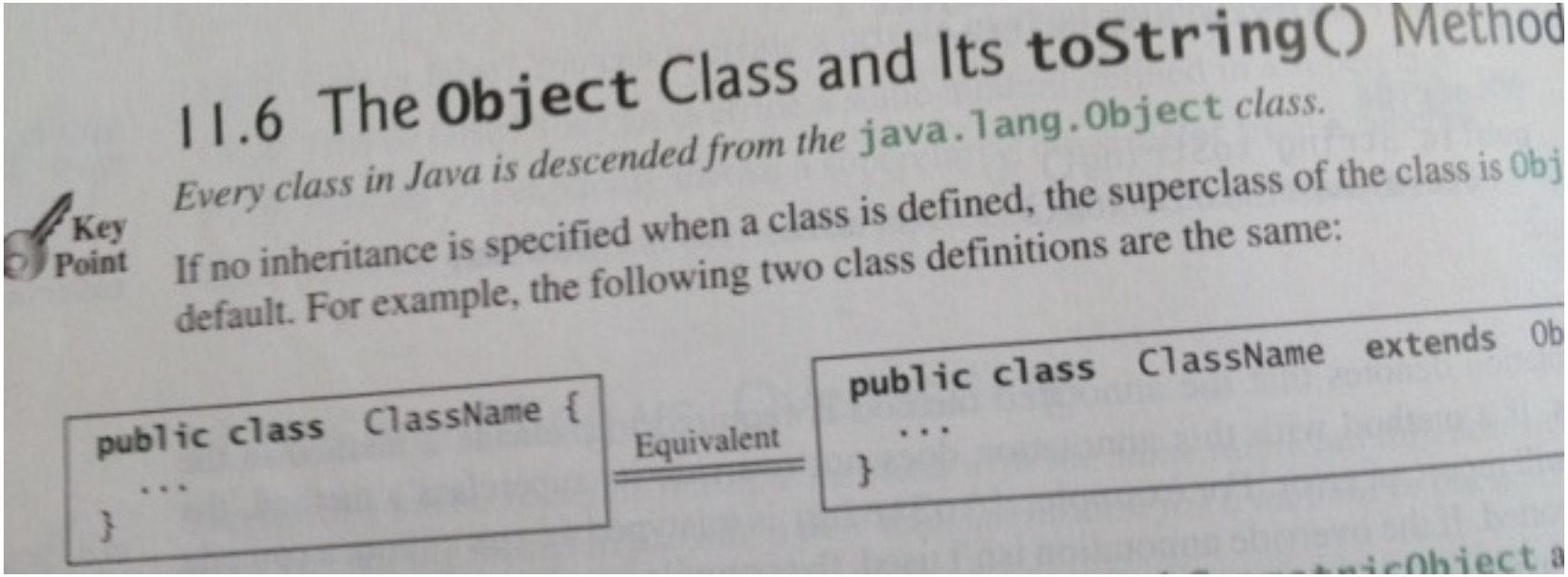
- Methods in sub-classes
  - Define changed method with same name

toString( ) Method

pubic String toString( ) {

- returns a descriptive string of object

❖ ***object*** class is implicit/default *superclass*



11.6 The Object Class and Its toString() Method

Every class in Java is descended from the `java.lang.Object` *class.*

Key Point

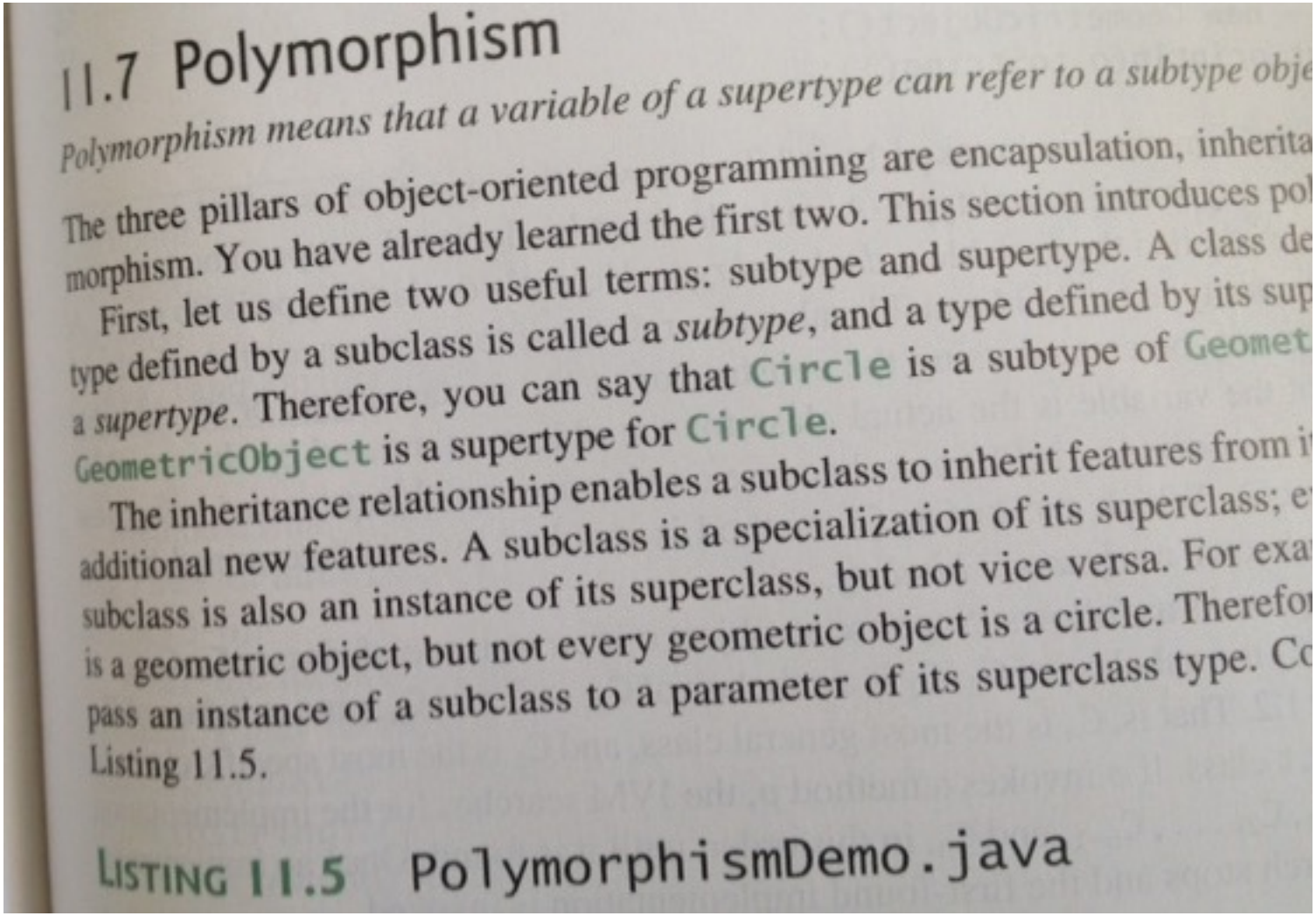
If no inheritance is specified when a class is defined, the superclass of the class is Obj default. For example, the following two class definitions are the same:

```
public class ClassName {
   ...
}
```

Equivalent

```
public class ClassName extends Ob
   ...
}
```

Polymorphism



11.7 Polymorphism

Polymorphism means that a variable of a supertype can refer to a subtype obje

The three pillars of object-oriented programming are encapsulation, inherita morphism. You have already learned the first two. This section introduces pol

First, let us define two useful terms: subtype and supertype. A class de type defined by a subclass is called a *subtype*, and a type defined by its sup a *supertype*. Therefore, you can say that `Circle` is a subtype of `Geomet` `GeometricObject` is a supertype for `Circle`.

The inheritance relationship enables a subclass to inherit features from it additional new features. A subclass is a specialization of its superclass; e subclass is also an instance of its superclass, but not vice versa. For exa is a geometric object, but not every geometric object is a circle. Therefo pass an instance of a subclass to a parameter of its superclass type. Co Listing 11.5.

LISTING 11.5 PolymorphismDemo.java

Optimizing by *Refactoring*

➢ Wikipedia

Here are some examples of micro-refactorings; some of these may only apply to certain languages or language types. A longer list can be found in Martin Fowler's refactoring book[2][*page needed*] and website.[6] Many development environments provide automated support for these micro-refactorings. For instance, a programmer could click on the name of a variable and then select the "Encapsulate field" refactoring from a context menu. The IDE would then prompt for additional details, typically with sensible defaults and a preview of the code changes. After confirmation by the programmer it would carry out the required changes throughout the code.

- Techniques that allow for more abstraction
  - Encapsulate field – force code to access the field with getter and setter methods
  - Generalize type – create more general types to allow for more code sharing
  - Replace type-checking code with state/strategy[7]
  - Replace conditional with polymorphism [8]
- Techniques for breaking code apart into more logical pieces
  - Componentization breaks code down into reusable semantic units that present clear, well-defined, simple-to-use interfaces.
  - Extract class moves part of the code from an existing class into a new class.
  - Extract method, to turn part of a larger method into a new method. By breaking down code in smaller pieces, it is more easily understandable. This is also applicable to functions.
- Techniques for improving names and location of code
  - Move method or move field – move to a more appropriate class or source file
  - Rename method or rename field – changing the name into a new one that better reveals its purpose
  - Pull up – in object-oriented programming (OOP), move to a superclass
  - Push down – in OOP, move to a subclass

❖ Re-Encapsulation

❖ Factoring

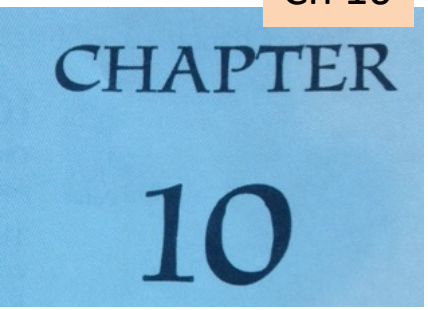
❖ Move

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

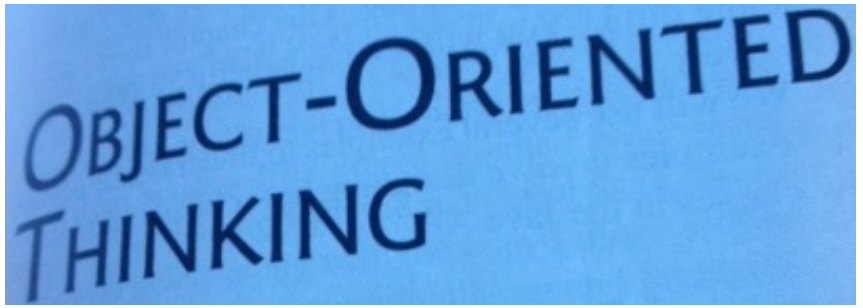
COMP110

Chapter 10

Ch 10



More Objects & Classes

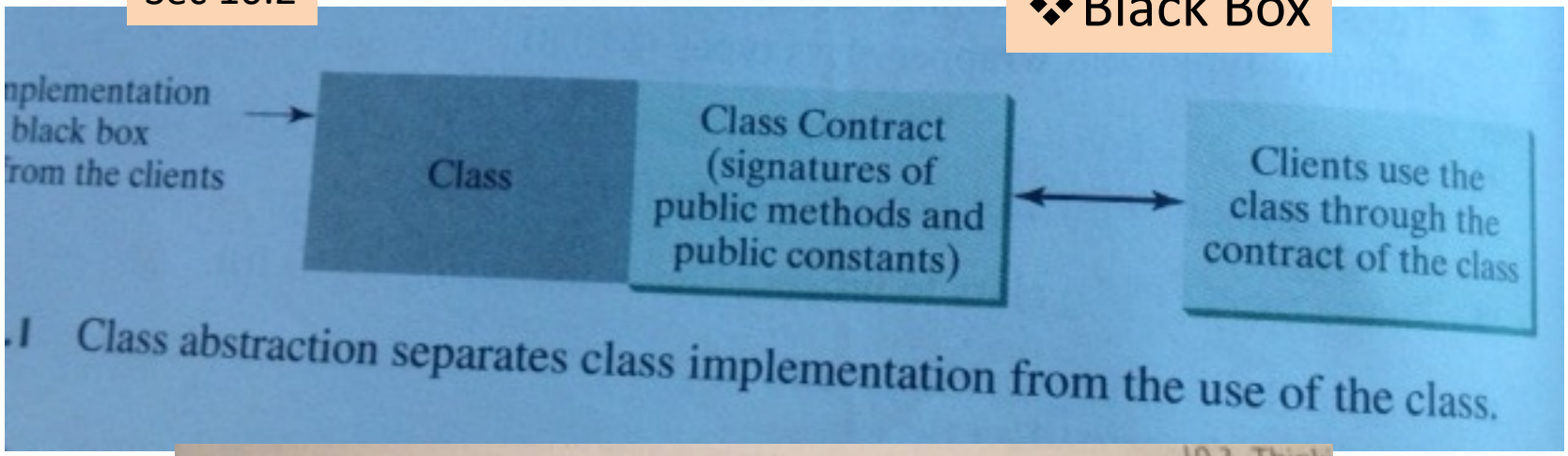


1. Intro
2. Abraction/Encapsulation
3. Thinking in Objects
4. Class Relationships
5. Case Study – Course
6. Case Study – Stacks
7. Primitive Data Types as Objects
8. Wrapper Class Types (conversion)
9. BigInteger & BigDecimal classes
10. String class
11. StringBuilder & StringBuffer classes
    a. Palindromes revisited (Listing 10.10)

Abstraction/Encapsulation

Sec 10.2

❖ Black Box



.1 Class abstraction separates class implementation from the use of the class.

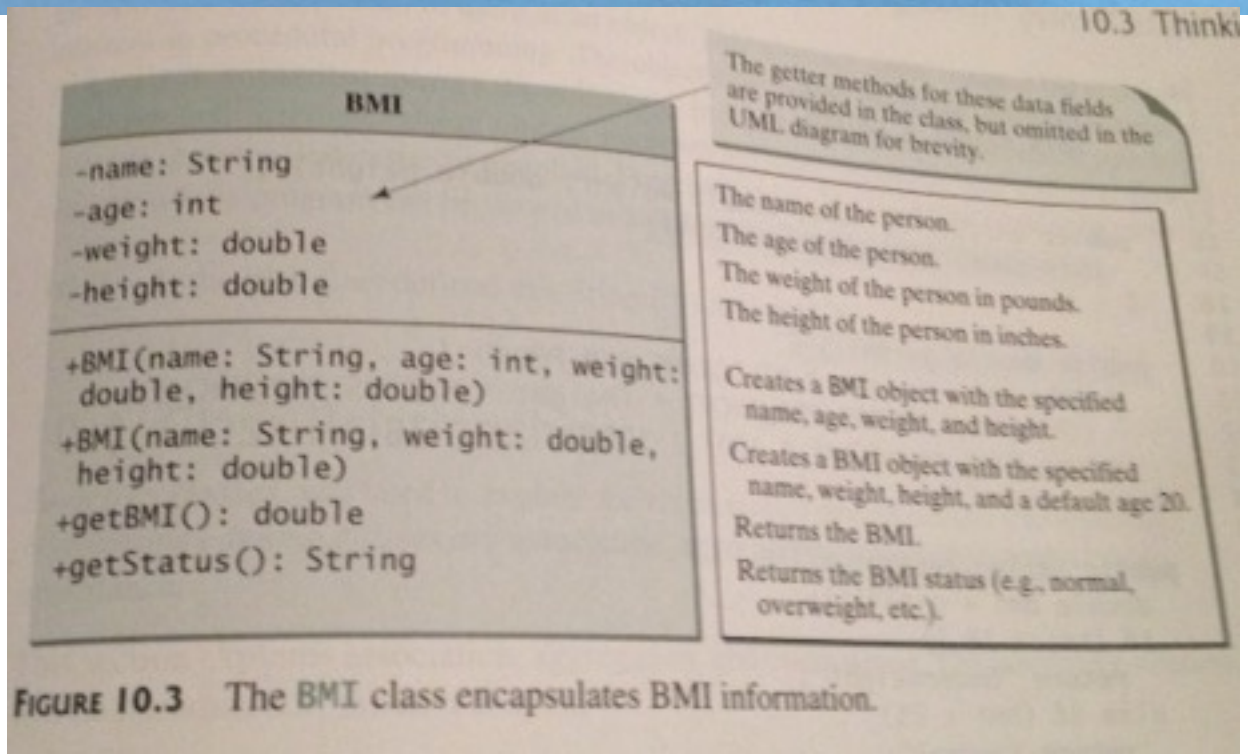
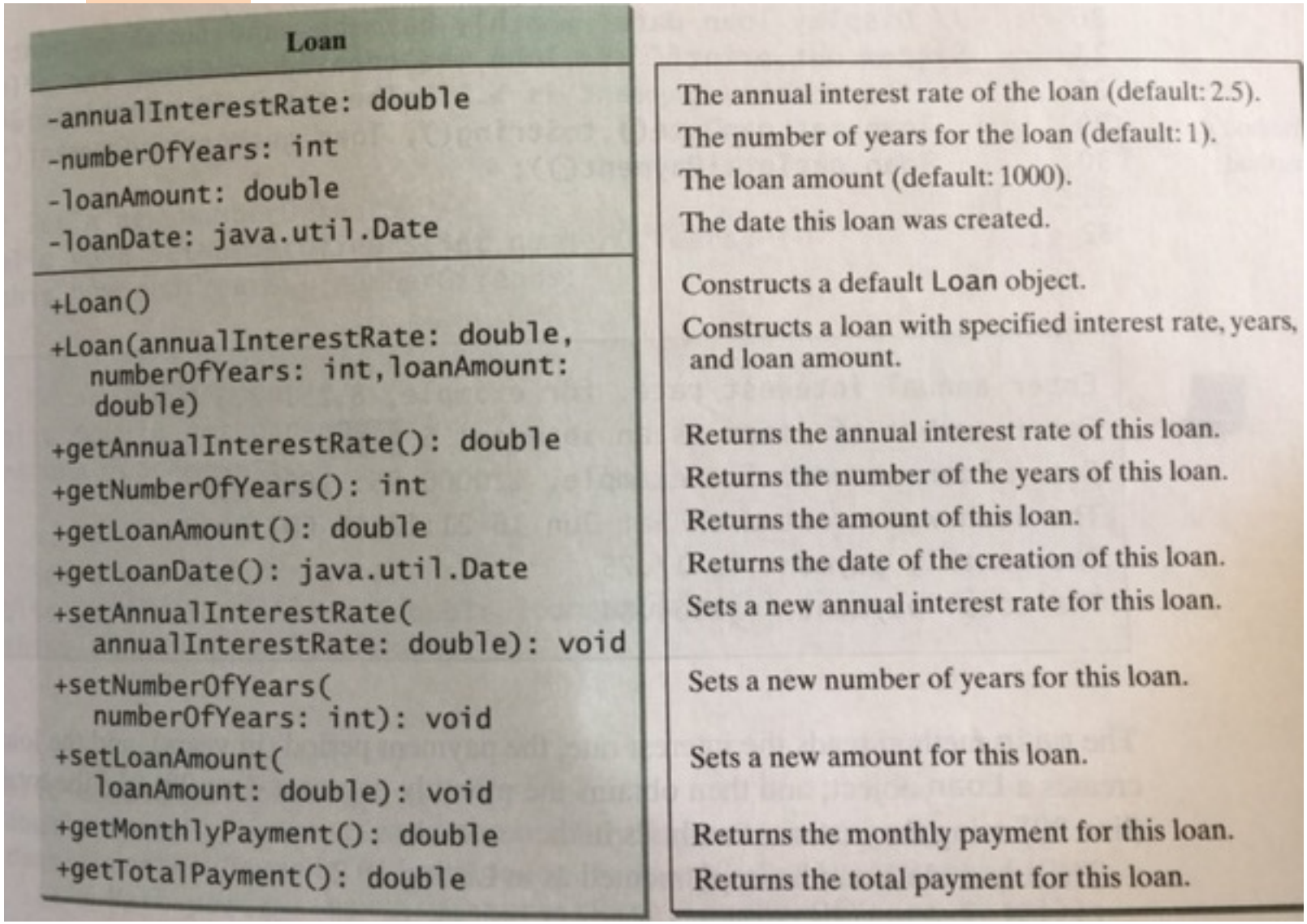


FIGURE 10.3 The BMI class encapsulates BMI information.

Thinking in Objects

Sec 10.3



Loan	
-annualInterestRate: double	The annual interest rate of the loan (default: 2.5).
-numberOfYears: int	The number of years for the loan (default: 1).
-loanAmount: double	The loan amount (default: 1000).
-loanDate: java.util.Date	The date this loan was created.
+Loan()	Constructs a default Loan object.
+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)	Constructs a loan with specified interest rate, years, and loan amount.
+getAnnualInterestRate(): double	Returns the annual interest rate of this loan.
+getNumberOfYears(): int	Returns the number of the years of this loan.
+getLoanAmount(): double	Returns the amount of this loan.
+getLoanDate(): java.util.Date	Returns the date of the creation of this loan.
+setAnnualInterestRate(annualInterestRate: double): void	Sets a new annual interest rate for this loan.
+setNumberOfYears(numberOfYears: int): void	Sets a new number of years for this loan.
+setLoanAmount(loanAmount: double): void	Sets a new amount for this loan.
+getMonthlyPayment(): double	Returns the monthly payment for this loan.
+getTotalPayment(): double	Returns the total payment for this loan.

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP110

Class Associations

Sec 10.4

- UML

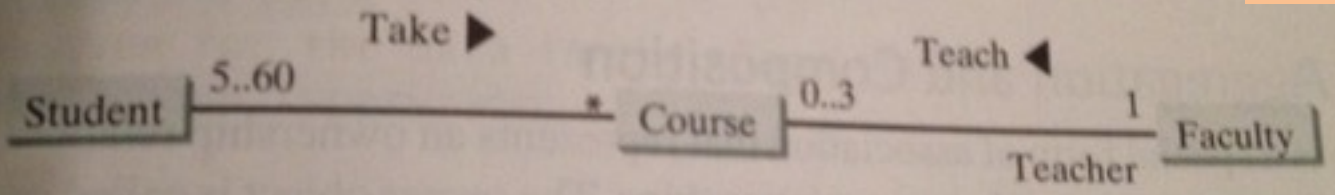


FIGURE 10.4 This UML diagram shows that a student may take any number of courses. A faculty member may teach at most three courses, a course may have from five to sixty students, and a course is taught by only one faculty member.

- Java

```
public class Student {
  private Course[]
    courseList;

  public void addCourse(
    Course s) { ... }
}
```

```
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  public void addStudent(
    Student s) { ... }

  public void setFaculty(
    Faculty faculty) { ... }
}
```

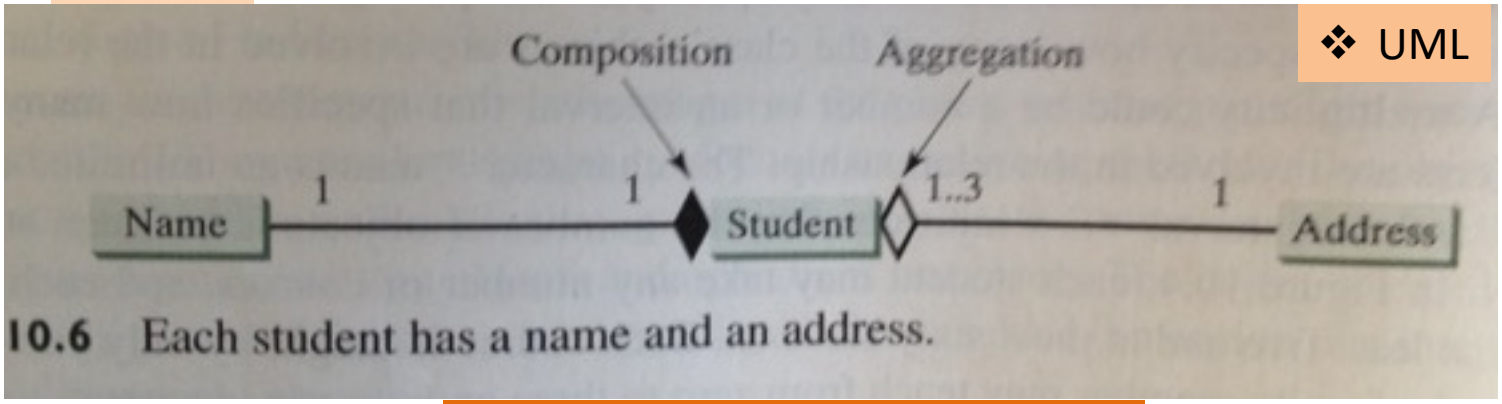
```
public class Faculty {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```

IGURE 10.5 The association relations are implemented using data fields and methods in classes.

Class Associations

Sec 10.4

❖ UML



10.6 Each student has a name and an address.

➢ Composition & Aggregation

❖ Java

```
public class Name {
   ...
}
```

Aggregated class

```
public class Student {
   private Name name;
   private Address address;
   ...
}
```

Aggregating class

```
public class Address {
   ...
}
```

Aggregated class

FIGURE 10.7 The composition relations are implemented using data fields in classes.

Case: Course

- Course class
- UML

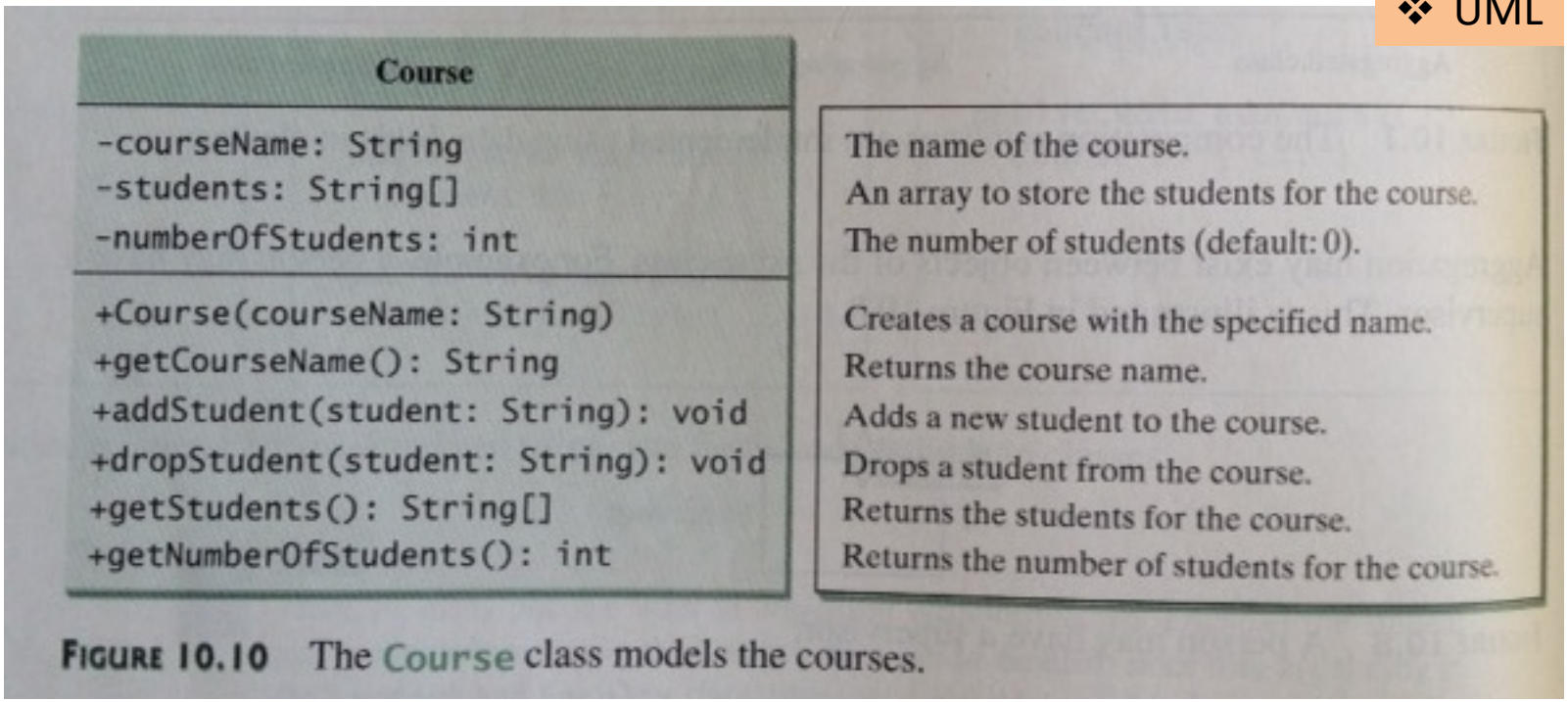


FIGURE 10.10 The Course class models the courses.

Case: Course

- Course class

- Java: main class

LISTING 10.5 TestCourse.java

```
1  public class TestCourse {
2    public static void main(String[] args) {
3      Course course1 = new Course("Data Structures");
4      Course course2 = new Course("Database Systems");
5
6      course1.addStudent("Peter Jones");
7      course1.addStudent("Kim Smith");
8      course1.addStudent("Anne Kennedy");
9
10     course2.addStudent("Peter Jones");
11     course2.addStudent("Steve Smith");
12
13     System.out.println("Number of students in course1: "
14       + course1.getNumberOfStudents());
15     String[] students = course1.getStudents();
16     for (int i = 0; i < course1.getNumberOfStudents(); i++)
17       System.out.print(students[i] + ", ");
18
19     System.out.println();
20     System.out.print("Number of students in course2: "
21       + course2.getNumberOfStudents());
22   }
23 }
```

Case: Course

- Course class
- Java: Course class

properties (state)

constructor

methods

LISTING 10.6 Course.java

```
1  public class Course {
2    private String courseName;
3    private String[] students = new String[100];
4    private int numberOfStudents;
5
6    public Course(String courseName) {
7      this.courseName = courseName;
8    }
9
10   public void addStudent(String student) {
11     students[numberOfStudents] = student;
12     numberOfStudents++;
13   }
14
15   public String[] getStudents() {
16     return students;
17   }
18
19   public int getNumberOfStudents() {
20     return numberOfStudents;
21   }
22
23   public String getCourseName() {
24     return courseName;
25   }
26
27   public void dropStudent(String student) {
28     // Left as an exercise in Programming Exercise 10.9
29   }
30 }
```

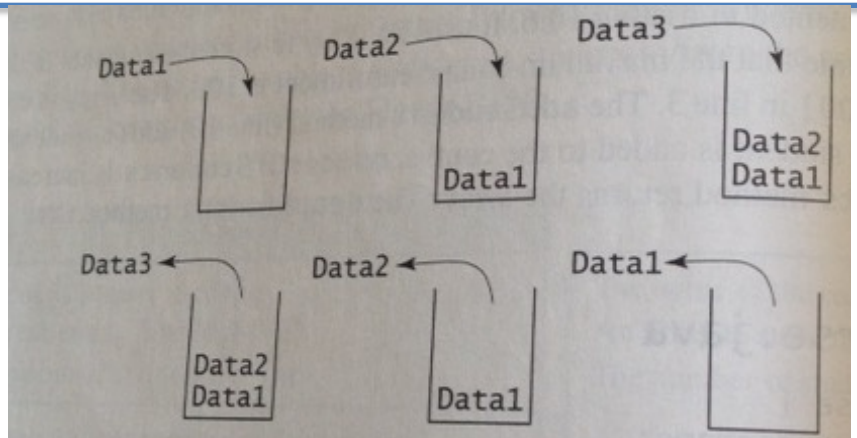
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP110

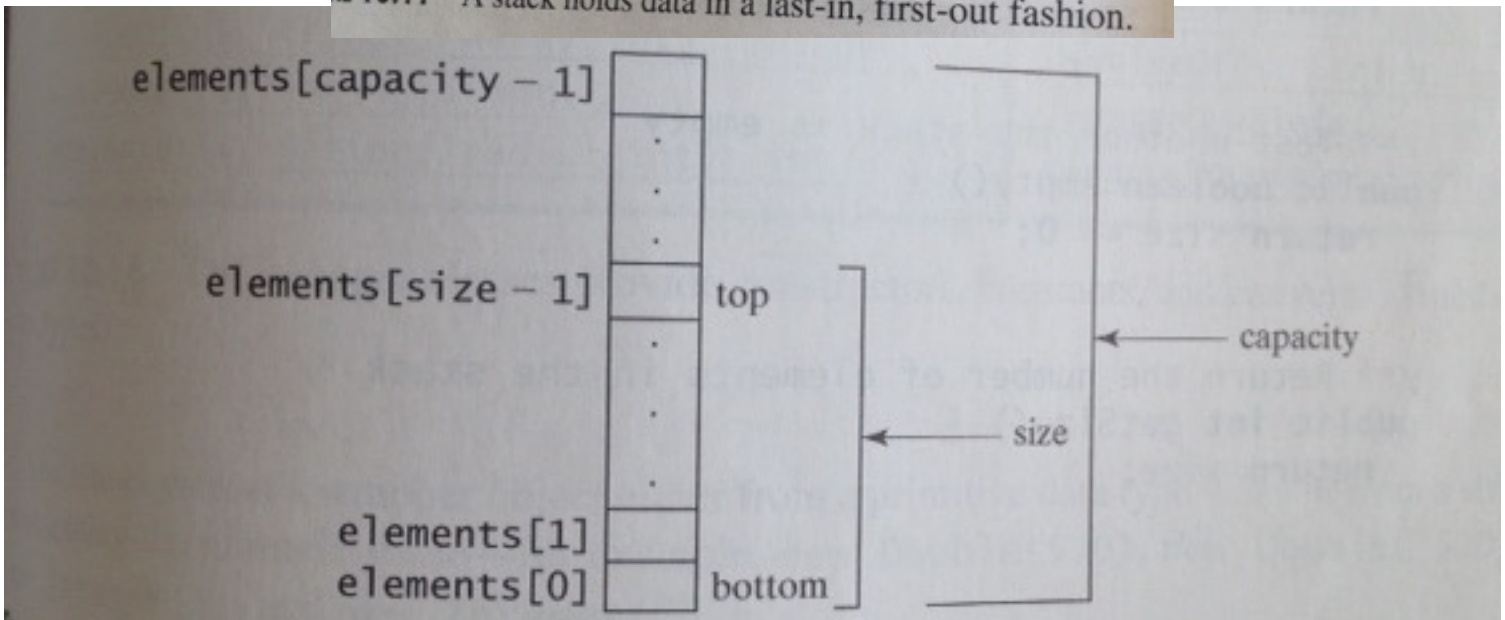
Case: Stacks

Sec 10.6

DR JEFF SOFTWARE
INDIE APP DEVELOPER



RE 10.11 A stack holds data in a last-in, first-out fashion.



IGURE 10.13 The StackOfIntegers class encapsulates the stack storage and provides
e operations for manipulating the stack.

Case: Stacks

Sec 10.6

❖ Stack UML

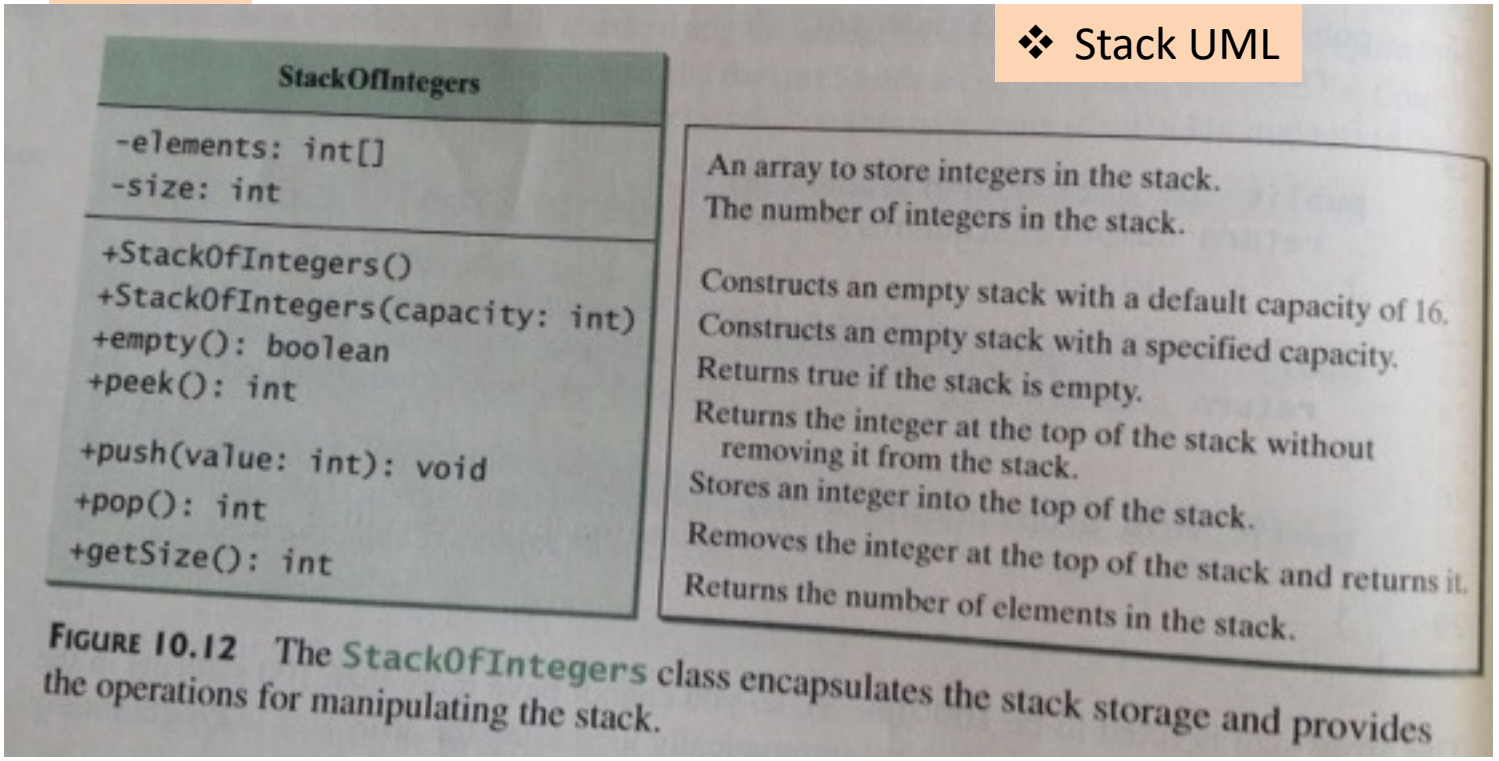


FIGURE 10.12 The `StackOfIntegers` class encapsulates the stack storage and provides the operations for manipulating the stack.

Case: Stacks

- Java: main class

LISTING 10.7 TestStackOfIntegers.java

```
1  public class TestStackOfIntegers {
2    public static void main(String[] args) {
3      StackOfIntegers stack = new StackOfIntegers();
4
5      for (int i = 0; i < 10; i++)
6        stack.push(i);
7
8      while (!stack.empty())
9        System.out.print(stack.pop() + " ");
10   }
11 }
```

Case: Stacks

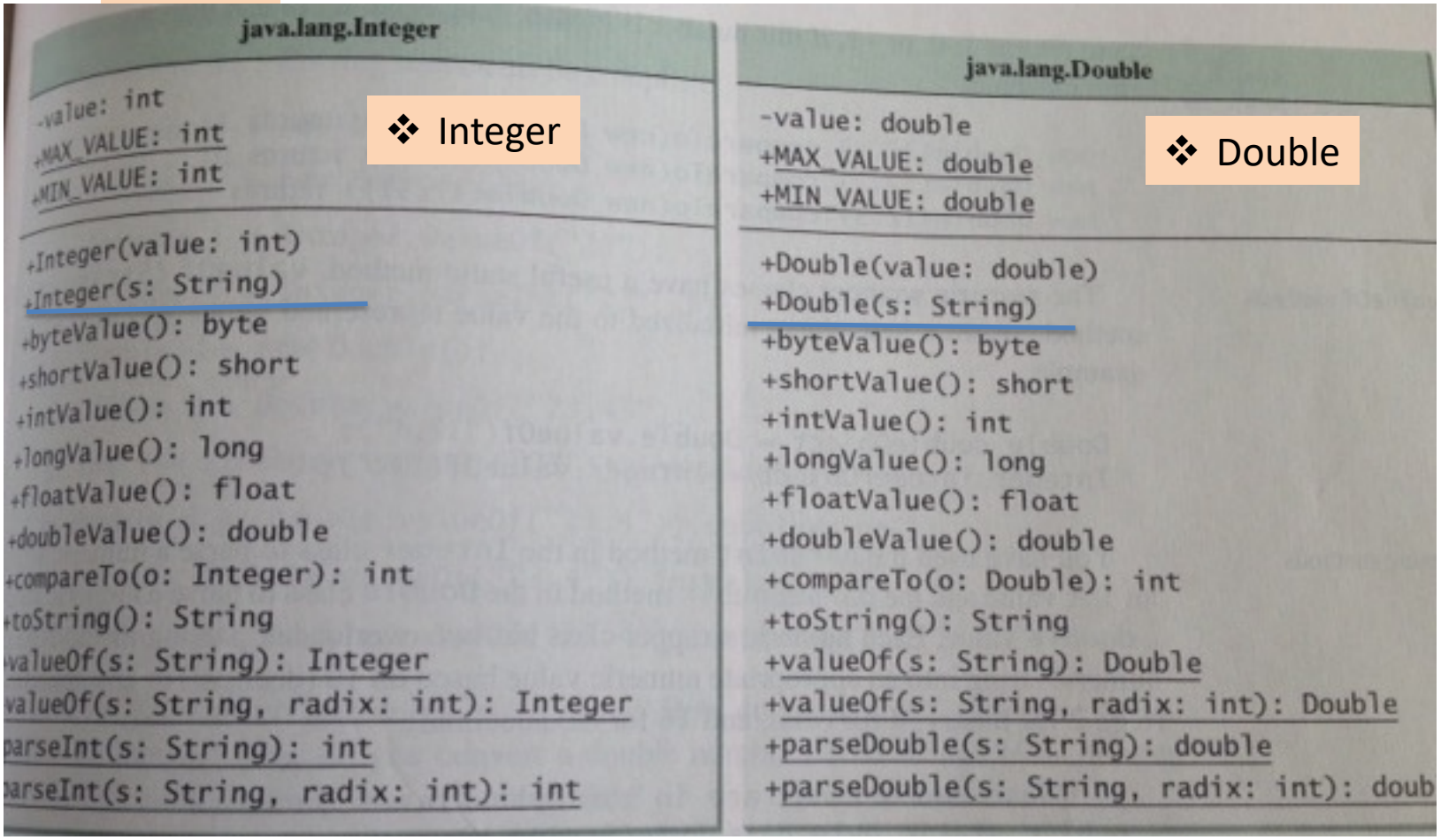
LISTING 10.8 StackOfIntegers.java

```
1  public class StackOfIntegers {
2    private int[] elements;
3    private int size;
4    public static final int DEFAULT_CAPACITY = 16;
5
6    /** Construct a stack with the default capacity 16 */
7    public StackOfIntegers() {
8      this (DEFAULT_CAPACITY);
9    }
10
11   /** Construct a stack with the specified maximum capa
12   public StackOfIntegers(int capacity) {
13     elements = new int[capacity];
14   }
15
```

- Java: Stack class

```
17   public void push(int value) {
18     if (size >= elements.length) {
19       int[] temp = new int[elements.length * 2];
20       System.arraycopy(elements, 0, temp, 0, elements.length);
21       elements = temp;
22     }
23
24     elements[size++] = value;
25   }
26
27   /** Return and remove the top element from the stack */
28   public int pop() {
29     return elements[--size];
30   }
31
32   /** Return the top element from the stack */
33   public int peek() {
34     return elements[size - 1];
35   }
36
37   /** Test whether the stack is empty */
38   public boolean empty() {
39     return size == 0;
40   }
41
42   /** Return the number of elements in the stack */
43   public int getSize() {
44     return size;
45   }
46 }
```

Wrapper Class Types



java.lang.Integer

-value: int
+MAX_VALUE: int
+MIN_VALUE: int

+Integer(value: int)
+Integer(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Integer): int
+toString(): String
+valueOf(s: String): Integer
+valueOf(s: String, radix: int): Integer
+parseInt(s: String): int
+parseInt(s: String, radix: int): int

❖ Integer

java.lang.Double

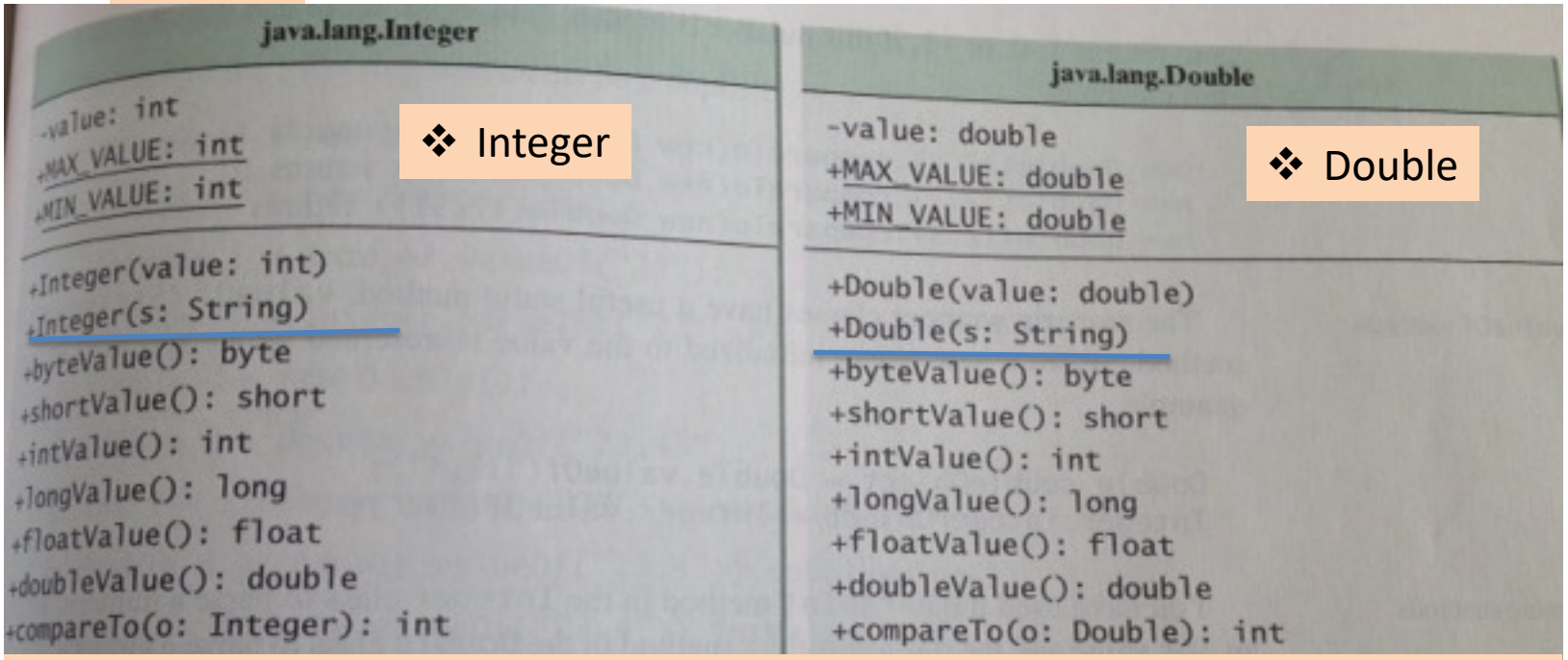
-value: double
+MAX_VALUE: double
+MIN_VALUE: double

+Double(value: double)
+Double(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Double): int
+toString(): String
+valueOf(s: String): Double
+valueOf(s: String, radix: int): Double
+parseDouble(s: String): double
+parseDouble(s: String, radix: int): doub

❖ Double

- new Double(12.4).intValue( ) → returns 12
- new Integer(12).doubleValue( ) → returns 12.0

Wrapper Class Types



- Integer
- Double

- new Double(12.4).compareTo (new Double(12.3)) → returns **1**
- new Double(12.3).compareTo (new Double(12.3)) → returns **0**
- new Double(12.3).compareTo (new Double(12.51)) → returns **-1**

Automatic Conversion

- convert **PRIMITIVE** to **WRAPPER** → ***boxing***
- convert **WRAPPER** to **PRIMITIVE** → **un*boxing***

BigInteger & BigDecimal

Sec 10.9

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

- max long
- Using ***Strings*** to represent large numbers

The output is 18446744073709551614.

- Using ***methods*** for arithmetic

```
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

- Round mode

The output is 0.33333333333333333334.

BigInteger & BigDecimal

Sec 10.9

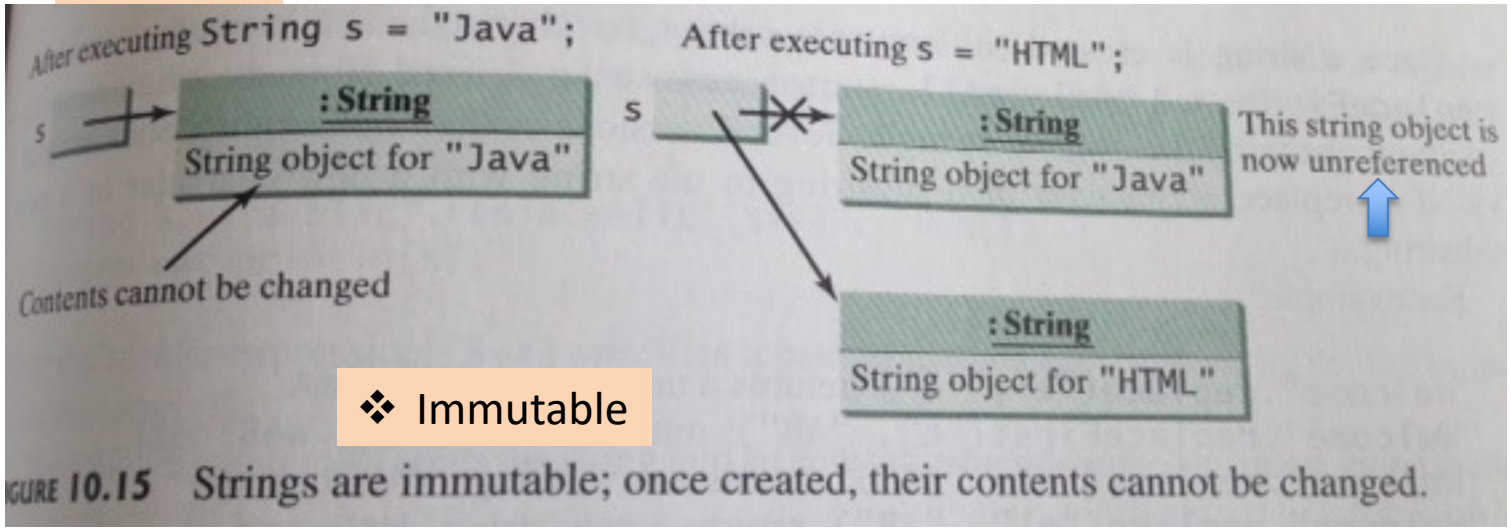
LISTING 10.9 LargeFactorial.java

```
1  import java.math.*;
2
3  public class LargeFactorial {
4    public static void main(String[] args) {
5      System.out.println("50! is \n" + factorial(50));
6    }
7
8    public static BigInteger factorial(long n) {
9      BigInteger result = BigInteger.ONE;
10     for (int i = 1; i <= n; i++)
11       result = result.multiply(new BigInteger(i + ""));
12
13     return result;
14   }
15 }
```

```
50! is
30414093201713378043612608166064768844377641568960512000000000000
```

String Class

Sec 10.10



❖ Immutable

GURE 10.15 Strings are immutable; once created, their contents cannot be changed.

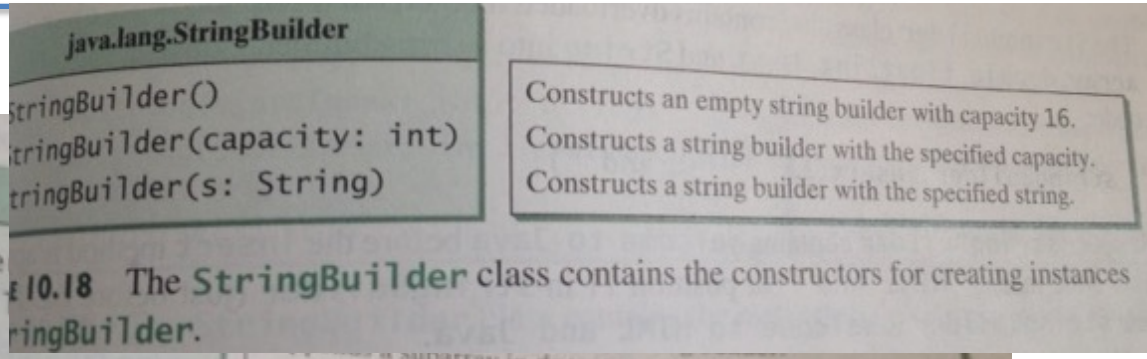
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP110

StringBuilder & StringBuffer

Sec 10.11

DR JEFF SOFTWARE
INDIE APP DEVELOPER



java.lang.StringBuilder

StringBuilder()
StringBuilder(capacity: int)
StringBuilder(s: String)

Constructs an empty string builder with capacity 16.
Constructs a string builder with the specified capacity.
Constructs a string builder with the specified string.

10.18 The StringBuilder class contains the constructors for creating instances ringBuilder.

java.lang.StringBuilder

append(data: char[]): StringBuilde
append(data: char[], offset: int, StringBuilder
append(v: aPrimitiveType): StringBuilder
append(s: String): StringBuilder
delete(startIndex: int, endIndex: int): StringBuilder
deleteCharAt(index: int): StringBuilder
insert(index: int, data: char[], offset: int, len: int): StringBuilder
insert(offset: int, data: char[]): StringBuilder
insert(offset: int, b: aPrimitiveType): tringBuilder
sert(offset: int, s: String): StringBuilder
place(startIndex: int, endIndex: int, s: ring): StringBuilder
erse(): StringBuilder
CharAt(index: int, ch: char): void

subarray in data into this string builder.
Appends a primitive type value as a string to this builder.
Appends a string to this string builder.
Deletes characters from startIndex to endIndex-1.
Deletes a character at the specified index.
Inserts a subarray of the data in the array into the builder at the specified index.
Inserts data into this builder at the position offset.
Inserts a value converted to a string into this builder.
Inserts a string into this builder at the position offset.
Replaces the characters in this builder from startIndex to endIndex-1 with the specified string.
Reverses the characters in the builder.
Sets a new character at the specified index in this builder.

19 The StringBuilder class contains the methods for modifying string builders.

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP110

Palindromes Revisited

Sec 10.11

String methods

- stringBuilder
- isLetterOrDigit

LISTING 10.10 PalindromeIgnoreNonAlphanumeric.java

```
1  import java.util.Scanner;
2
3  public class PalindromeIgnoreNonAlphanumeric {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Prompt the user to enter a string
10     System.out.print("Enter a string: ");
11     String s = input.nextLine();
12
13     // Display result
14     System.out.println("Ignoring nonalphanumeric characters, \nis "
15       + s + " a palindrome? " + isPalindrome(s));
16   }
17
18   /** Return true if a string is a palindrome */
19   public static boolean isPalindrome(String s) {
20     // Create a new string by eliminating nonalphanumeric chars
21     String s1 = filter(s);
22
23     // Create a new string that is the reversal of s1
24     String s2 = reverse(s1);
25
26     // Check if the reversal is the same as the original string
27     return s2.equals(s1);
28   }
29
30   /** Create a new string by eliminating nonalphanumeric chars */
31   public static String filter(String s) {
32     // Create a string builder
33     StringBuilder stringBuilder = new StringBuilder();
34
35     // Examine each char in the string to skip alphanumeric char
36     for (int i = 0; i < s.length(); i++) {
37       if (Character.isLetterOrDigit(s.charAt(i))) {
38         stringBuilder.append(s.charAt(i));
39       }
```

Abstract Classes

Why not use an abstract class?

- Although this update in Java 8 does make it seem as though interfaces and abstract classes are the same… that is not the case. An abstract class can define a constructor. They can be objects with a state associated with them, in contrast to an interface which simply defines a contract. Methods in an abstract class can modify both method arguments as well as fields of their class, whereas default methods in an interface can only access its arguments because interfaces do not have any state. Both are really used for different purposes.

Threads

Then, when you want to execute this code, you construct an instance of the `Worker` class. You can then submit the instance to a thread pool, or keep it simple and start a new thread:

```
1  Worker w = new Worker();
2    new Thread(w).start();
```

CSUN
CALIFORNIA
STATE UNIVERSITY
NORTHRIDGE
COMP110

Data Structures

DR JEFF
SOFTWARE
INDIE APP DEVELOPER

Array Lists

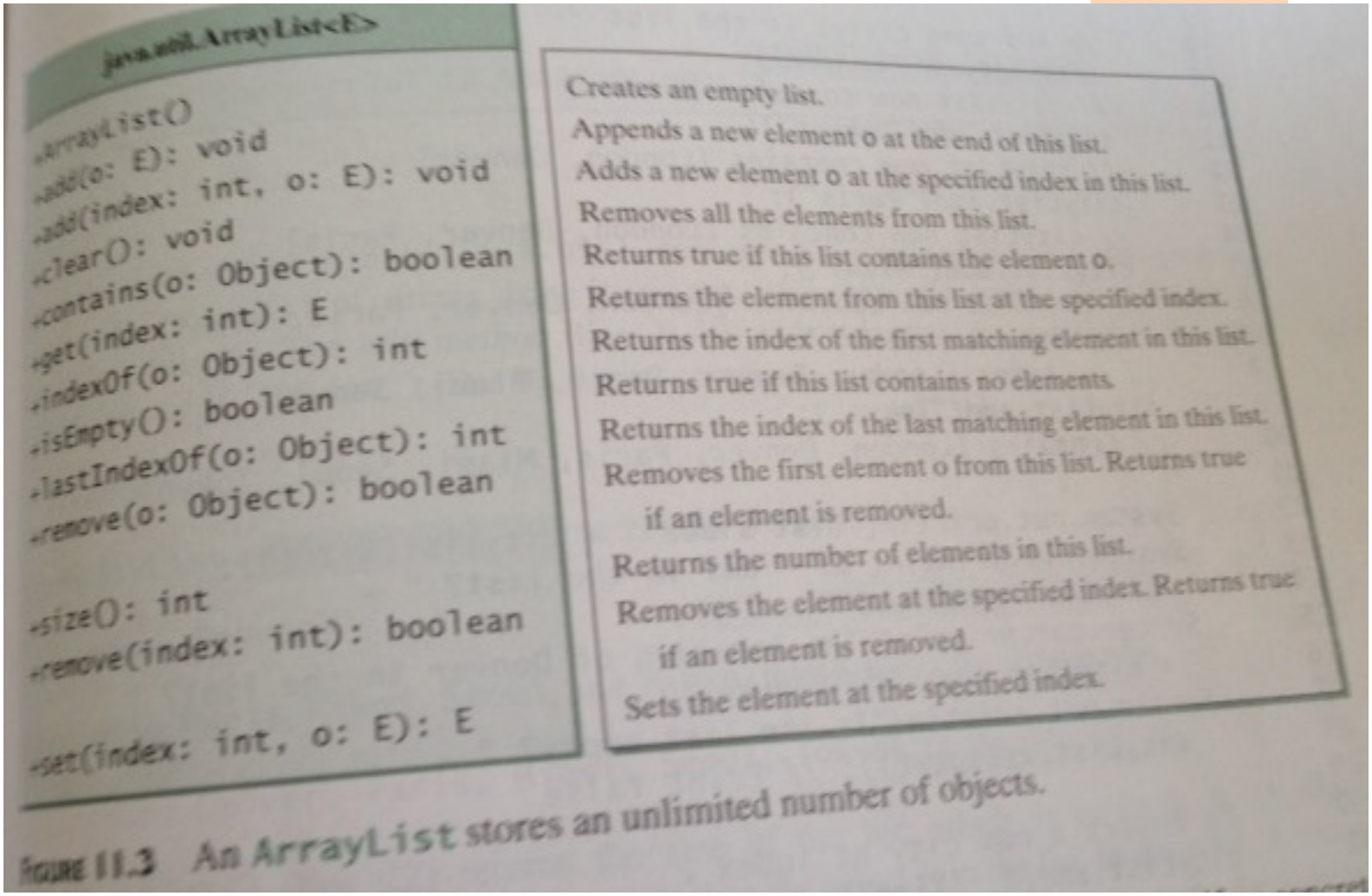
Sec 11.11
pp. 432-438

Array List Class

ArrayList Class

```
ArrayList<E> list = new ArrayList<E>();
list.add(object);
list.add(index, object);
list.clear();
Object o = list.get(index);
boolean b = list.isEmpty();
boolean b = list.contains(object);
int i = list.size();
list.remove(index);
list.set(index, object);
int i = list.indexOf(object);
int i = list.lastIndexOf(object);
```

ArrayList



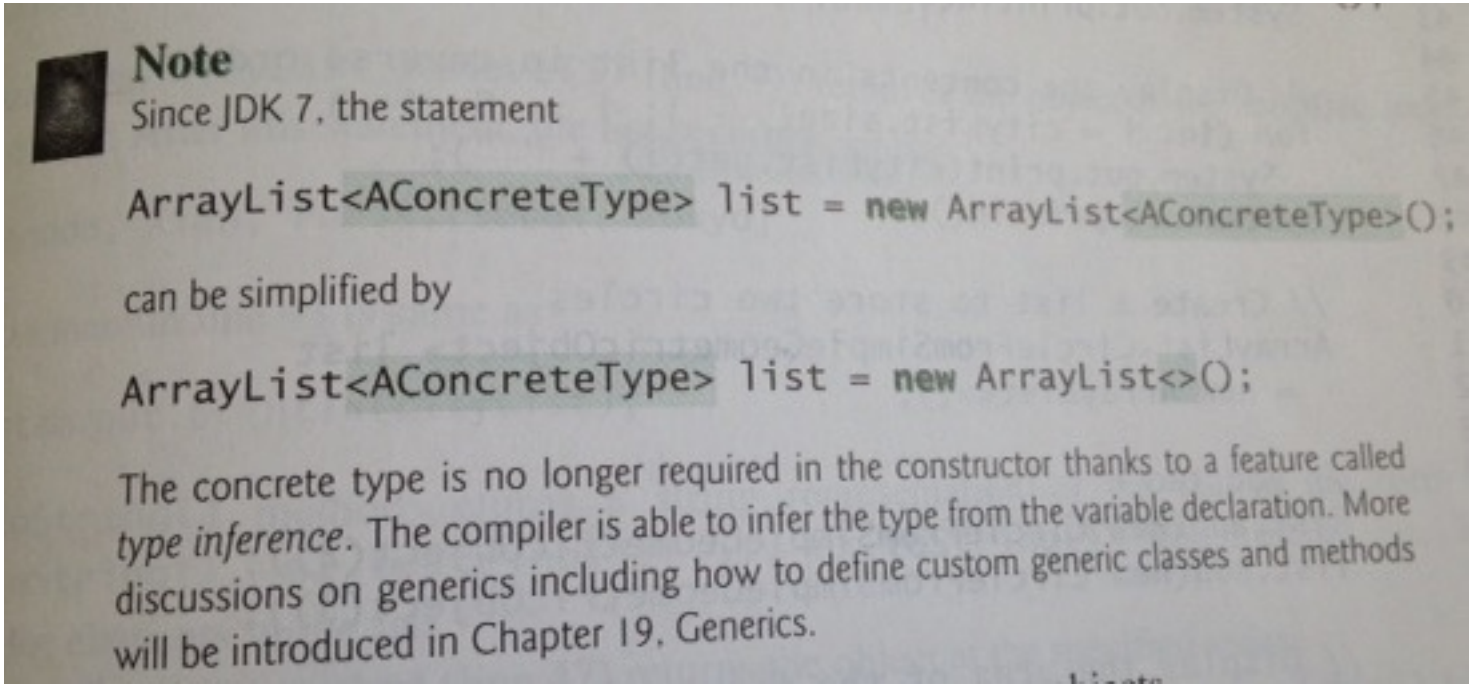
java.util.ArrayList<E>

```
+ArrayList()
+add(o: E): void
+add(index: int, o: E): void
+clear(): void
+contains(o: Object): boolean
+get(index: int): E
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean

+size(): int
+remove(index: int): boolean

+set(index: int, o: E): E
```

Creates an empty list.
Appends a new element o at the end of this list.
Adds a new element o at the specified index in this list.
Removes all the elements from this list.
Returns true if this list contains the element o.
Returns the element from this list at the specified index.
Returns the index of the first matching element in this list.
Returns true if this list contains no elements.
Returns the index of the last matching element in this list.
Removes the first element o from this list. Returns true if an element is removed.
Returns the number of elements in this list.
Removes the element at the specified index. Returns true if an element is removed.
Sets the element at the specified index.

FIGURE 11.3 An ArrayList stores an unlimited number of objects.

ArrayList



Note

Since JDK 7, the statement

```
ArrayList<AConcreteType> list = new ArrayList<AConcreteType>();
```

can be simplified by

```
ArrayList<AConcreteType> list = new ArrayList<>();
```

The concrete type is no longer required in the constructor thanks to a feature called *type inference*. The compiler is able to infer the type from the variable declaration. More discussions on generics including how to define custom generic classes and methods will be introduced in Chapter 19, Generics.

ArrayList

Sec 11.11

TABLE 11.1 Differences and Similarities between Arrays and ArrayList

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

CSUN
CALIFORNIA
STATE UNIVERSITY
NORTHRIDGE

COMP110

COMP182

Data Structures

DR JEFF
SOFTWARE
INDIE APP DEVELOPER

- Collections
- Maps

Data Structures

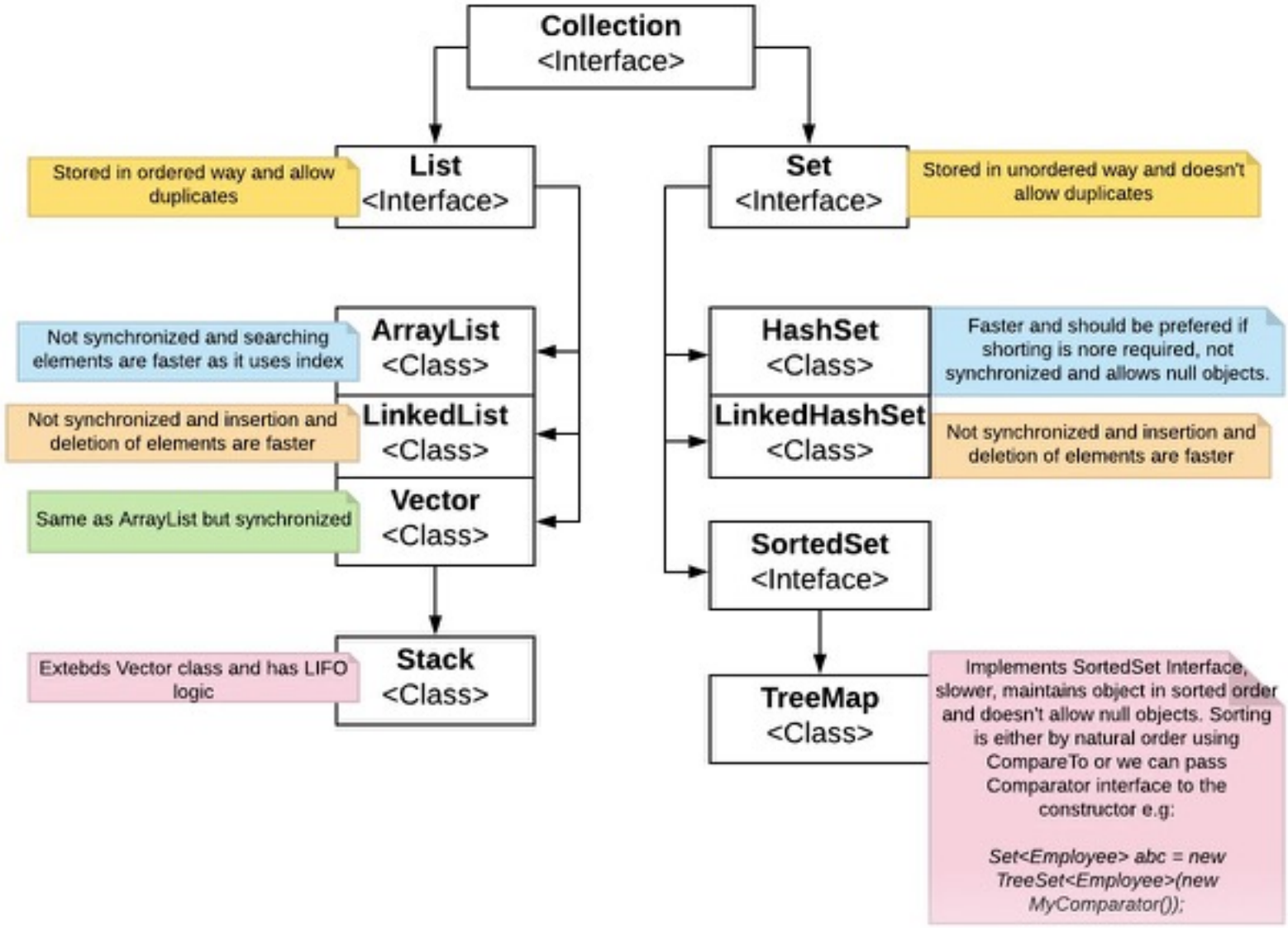
Big-O Worst-Case Run-Time Performance

Access: O(1), Search: O(1), Insertion: O(1), Deletion: O(1)

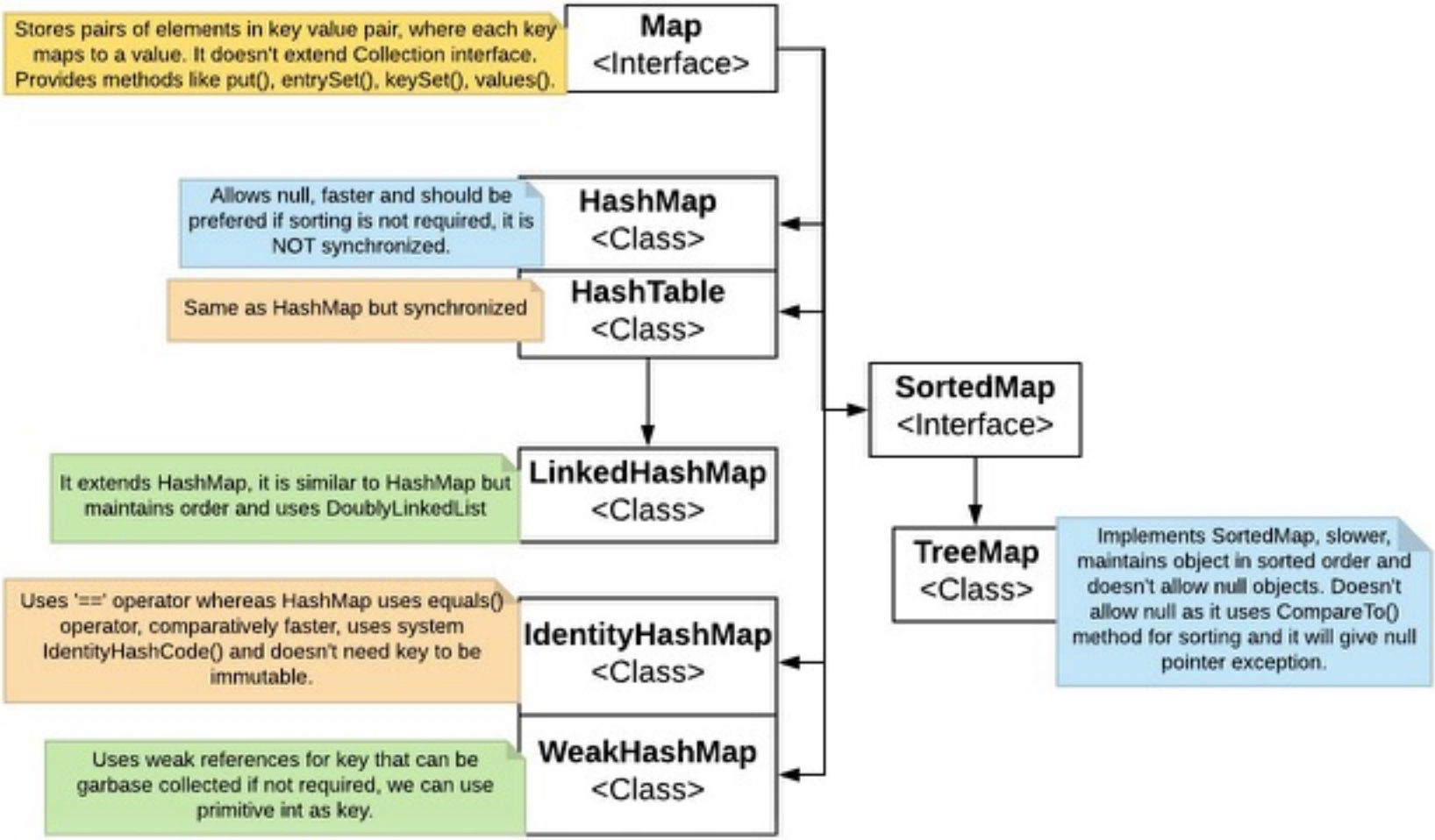
Data Structures

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion	
Basic Array	O(1)	O(n)	-	-	O(1)	O(n)	-	-	O(n)
Dynamic Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Collections



Maps



Summary

Extras

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP110

Lambda

DR JEFF SOFTWARE
INDIE APP DEVELOPER

Anonymous function

From Wikipedia, the free encyclopedia
(Redirected from Lambda (programming))

In computer programming, an **anonymous function** (**function literal**, **lambda abstraction**, or **lambda expression**) is a function definition that is not bound to an identifier. Anonymous functions are often arguments being passed to higher-order functions, or used for constructing the result of a higher-order function that needs to return a function.[1] If the function is only used once, or a limited number of times, an anonymous function may be syntactically lighter than using a named function. Anonymous functions are ubiquitous in functional programming languages and other languages with first-class functions, where they fulfill the same role for the function type as literals do for other data types.

Anonymous functions originate in the work of Alonzo Church in his invention of the lambda calculus, in which all functions are anonymous, in 1936, before electronic computers.[2] In several programming languages, anonymous functions are introduced using the keyword *lambda*, and anonymous functions are often referred to as lambdas or lambda abstractions. Anonymous functions have been a feature of programming languages since Lisp in 1958, and a growing number of modern programming languages support anonymous functions.

Lambda

Lambda expressions are converted to "functional interfaces" (defined as interfaces that contain only one abstract method in addition to one or more default or static methods),[13] as in the following example:

```
public class Calculator {
    interface IntegerMath {
        int operation(int a, int b);

        default IntegerMath swap() {
            return (a, b) -> operation(b, a);
        }
    }

    private static int apply(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    public static void main(String... args) {
        IntegerMath addition = (a, b) -> a + b;
        IntegerMath subtraction = (a, b) -> a - b;
        System.out.println("40 + 2 = " + apply(40, 2, addition));
        System.out.println("20 - 10 = " + apply(20, 10, subtraction));
        System.out.println("10 - 20 = " + apply(20, 10, subtraction.swap()));
    }
}
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP110

Lambda

DR JEFF SOFTWARE
INDIE APP DEVELOPER

Java [edit]

Java supports anonymous functions, named *Lambda Expressions*, starting with JDK 8.[12]

A lambda expression consists of a comma separated list of the formal parameters enclosed in parentheses, an arrow token (->), and a body. Data types of the parameters can always be omitted, as can the parentheses if there is only one parameter. The body can consist of one statement or a statement block.[13]

```
// with no parameter
() -> System.out.println("Hello, world.")

// with one parameter (this example is an identity function).
a -> a

// with one expression
(a, b) -> a + b

// with explicit type information
(long id, String name) -> "id: " + id + ", name:" + name

// with a code block
(a, b) -> { return a + b; }

// with multiple statements in the lambda body. It needs a code block.
// This example also includes two nested lambda expressions (the first one is also a closure).
(id, defaultPrice) -> {
    Optional<Product> product = productList.stream().filter(p -> p.getId() == id).findFirst();
    return product.map(p -> p.getPrice()).orElse(defaultPrice);
}
```

Lambdas

Like Macros for HLL

The Syntax of Lambda Expressions

Consider the previous sorting example again. We pass code that checks whether one string is shorter than another. We compute

```
1  Integer.compare(first.length(), second.length())
```

What are first and second? They are both strings! Java is a strongly typed language, and we must specify that as well:

```
1  (String first, String second)
2       -> Integer.compare(first.length(), second.length())
```

You have just seen your first lambda expression! Such an expression is simply a block of code, together with the specification of any variables that must be passed to the code.

Why the name? Many years ago, before there were any computers, the logician Alonzo Church wanted to formalize what it means for a mathematical function to be effectively computable. (Curiously, there are functions that are known to exist, but nobody knows how to compute their values.) He used the Greek letter lambda (λ) to mark parameters. Had he known about the Java API, he would have written:

```
1  λfirst.λsecond.Integer.compare(first.length(), second.length())
```

Why the letter λ? Did Church run out of other letters of the alphabet? Actually, the venerable *Principia Mathematica* used the ˆ accent to denote free variables, which inspired Church to use an uppercase lambda (Λ) for parameters. But in the end, he switched to the lowercase version. Ever since, an expression with parameter variables has been called a "lambda expression."

Code Guidelines

- Scope
  - **Local** – best to use – **Private**
  - **Global** – *be very careful* – **Public**
- Type casting
  - Use *explicit* types (avoid implicit casting & *overloading*)
- Procedure parameter passing
  - Use "By Value" for variables
  - Use "By Reference" for objects
- Condition codes
  - Set "CC" binary var (T/F) on action completion
  - Test "CC" before continuing with next action
- Error trapping & handling
  - TRY & CATCH blocks – use generously
  - Catch exception descriptions
  - Add as much pertinent info as possible (esp. location)
  - Report via "alert boxes"
  - Never allow un-trapped errors – they cause program interruption (that is what "beta testing" is for)

Tradeoffs

- Memory
  - Code (KB-MB)
    - Static
    - Lines of code
    - Verbosity
  - Data (MB-GB-TB)
    - Small files (CSV)
    - Databases (SQL)
    - Big data (data mining)

VS

- Performance (Speed)
  - Total execution time (sec)
    - Small tasks (compute only)
    - Big simulations (e.g., weather)
    - Verbosity
  - User response (msec)
    - Clicks
    - Text characters
    - Forms
  - Embedded control (msec)
    - Real-time response
    - Interrupts

Math Lib

Theory: Computability

Uncomputable?

Are there any problems in computer programming, that are seen as impossible?

Thomas Cormen, I've been teaching Computer Science at Dartmouth College since 1992.

Written Nov 5 · Upvoted by Timothy Johnson, PhD student in CS, UC Irvine and Jeff Nelson, Invented Chromebook, #Xoogler

Yes, indeed. The best-known one is the Halting Problem, shown to be uncomputable by Alan Turing. Suppose you want to know whether a program, call it P, given an input, call it x, would run to completion. Not whether P produces a correct answer, and not whether P produces an answer at all. Just: does program P, running on input x, run to completion? That is, does P running on x halt? Turing proved that it is impossible to write a computer program that takes two inputs, P and x, and correctly tells you *every time* whether P running on x halts.

❖ Halting

Once you have one uncomputable problem, you can find others. For example, Post's Correspondence Problem. Here, we are given two lists of n strings, say $A_1, A_2, \ldots, A_n$ and $B_1, B_2, \ldots, B_n$. The problem is to determine whether there exists a sequence of indices $i_1, i_2, \ldots, i_m$ such that $A_{i_1}, A_{i_2}, A_{i_3}, \ldots, A_{i_m}$ (the strings $A_{i_1}, A_{i_2}, A_{i_3}, \ldots, A_{i_m}$ concatenated together) gives the same string as $B_{i_1}, B_{i_2}, B_{i_3}, \ldots, B_{i_m}$. Using an example I wrote in *Algorithms Unlocked*, suppose that $n = 5$, and $A_1 = \text{ey}$, $A_2 = \text{er}$, $A_3 = \text{mo}$, $A_4 = \text{on}$, $A_5 = \text{h}$ and $B_1 = \text{ym}$, $B_2 = \text{r}$, $B_3 = \text{oon}$, $B_4 = \text{e}$, $B_5 = \text{hon}$. Then the sequence $\langle 5, 4, 1, 3, 4, 2 \rangle$ works, since both $A_5A_4A_1A_3A_4A_2$ and $B_5B_4B_1B_3B_4B_2$ form honeymooner.

❖ Strings