Computer Science II
4003-232-01 (20073)

Week 1: Review and Inheritance

Richard Zanibbi
Rochester Institute of Technology

Review of CS-I

Hardware and Software

Hardware

- Physical devices in a computer system (e.g. CPU, busses, printers, etc.)
- The machine and attached devices

Software

- Computer programs (machine instructions)
- “Soft” because they can be easily replaced or altered

Syntax and Semantics of Formal (e.g. Programming) Languages

Syntax

The rules governing how statements of a formal language (e.g. Java) may be created and combined

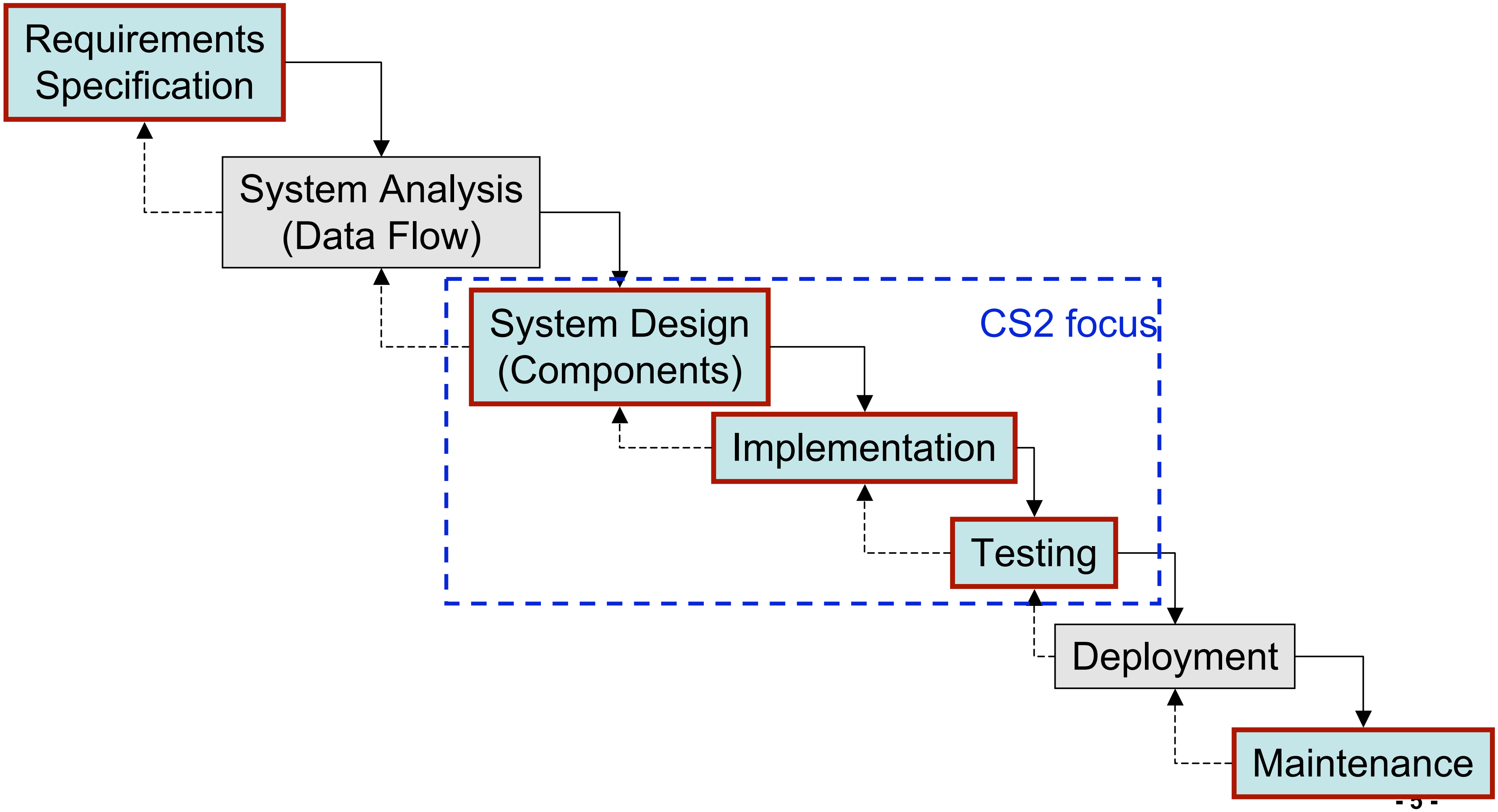
e.g. Rules for valid Rochester area phone numbers

Semantics

Given a language (code), the meaning of statements in the code (what statements *represent,* their *information*)

e.g. What *information* does the following represent:
"349-5313"

(Liang, p. 372): A "Waterfall" Model of Software Development



Requirements Specification

System Analysis (Data Flow)

System Design (Components)

CS2 focus

Implementation

Testing

Deployment

Maintenance

Object Oriented Programming

Paradigm:

Represent programs as a set of objects that encapsulate data and methods (state and behaviour) and pass messages between one another.

Key Object Oriented Concepts:

Class (template for a set of objects)

–Class (‘static’) variables that belong to a class

–Class (‘static’) methods that belong to a class

Instances (*objects),* each with state and behavior

–Instance variables that belong to *individual objects*

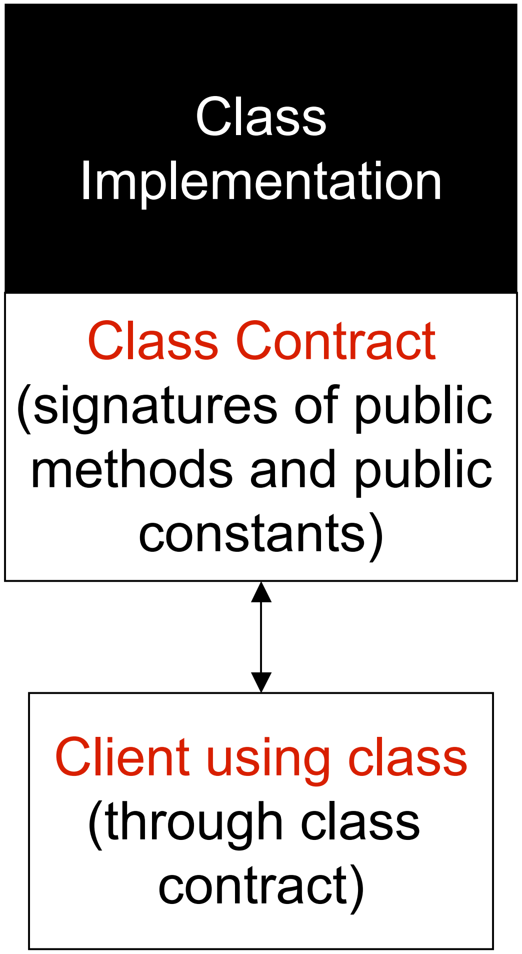
–Instance methods that are associated with *individual objects*

Main Elements of a Java Class

1. **Class signature**
   - Name, access modifiers (public, private, etc.), relationships with other classes, etc.
2. **Class (‘static’) properties**
   - Data members (variables, constants)
   - Methods: accessors, mutators, other methods
     - *cannot* reference (use) instance variables
3. **Instance properties**
   - Data members (variables, constants)
   - Methods: accessors, mutators, other methods
     - can reference (use) static and instance variables
     - ‘this’: refers to specific instance executing a method at run-time; all direct references to instance variables and methods implicitly refer to ‘this’
4. **(Instance) Constructors**
   - Used by the ‘new’ operator to initialize constructed instances
   - Constructors may invoke other constructors using ‘this’ (must be first statement if this is the case)

```
public class MyClass { // CLASS SIGNATURE
        private static int numberObjects = 0;    // CLASS DATA
        private int instanceVariable;            // INSTANCE DATA

        public MyClass(int value){              // CONSTRUCTOR
              instanceVariable = value;
              numberOfObjects++;
        }

        public int getInstanceVariable() { // INSTANCE METHOD
              return instanceVariable;
        }

        public static int getNumberObjects() {   // CLASS METHOD
              return numberObjects;
        }

        public static void main(String[] args) { // CLASS METHOD
              MyClass instance = new MyClass(5);
              MyClass instance2 = new MyClass(6);
              System.out.println(numberObjects + “: “ +
                     instance.getInstanceVariable() +
                     instance2.getInstanceVariable() );
        }
}
```

Class Contract



Class Implementation

Class Contract (signatures of public methods and public constants)

Client using class (through class contract)

- Collection of methods and data members accessible outside of a class

- Includes description of data members and method signatures

Method Signature

Name, return type, and parameter types for a method

e.g. boolean isDaytime(int seconds)

Miscellaneous Java...

- Declaring, Initializing, and Assigning Variables
- Floating point vs. Integer Arithmetic
- Type conversions (widening and narrowing), and casting
- Operator Precedence and Associativity (see Liang, p. 86-88 and Appendix C)
- Constants (‘final’)

- Class definition syntax
- Method definition syntax: constructors, void methods (procedures), non-void methods (functions)
- The new operator (instantiates objects from classes)
- Visibility modifiers (public and private)
- Arguments (pass-by-value), and local variables

Review: State (*Data/Variables*)

Variable Properties

1. **Location (in memory)**
2. **Name**
   - A symbol representing the location
3. **Type (of *encoding* used for stored data)**
   - Primitive (e.g. int, boolean), or
   - Reference (address in memory of a class instance (object))
4. **Value**
   - The primitive value or reference (for objects) stored at the variable location

Memory Diagrams: Illustrating Variable Properties

Variable Storage (Memory Locations)

Represented using a box

Variable Names and Types

Indicated using labels outside the box (e.g. x : int)

For static variables, indicate ‘static’ and class name

- e.g. x : int (static Widget)

Variable Values

– Primitive types: show value in the box (e.g. for integers, show decimal value)

– Reference variables: draw arrow from box to object data

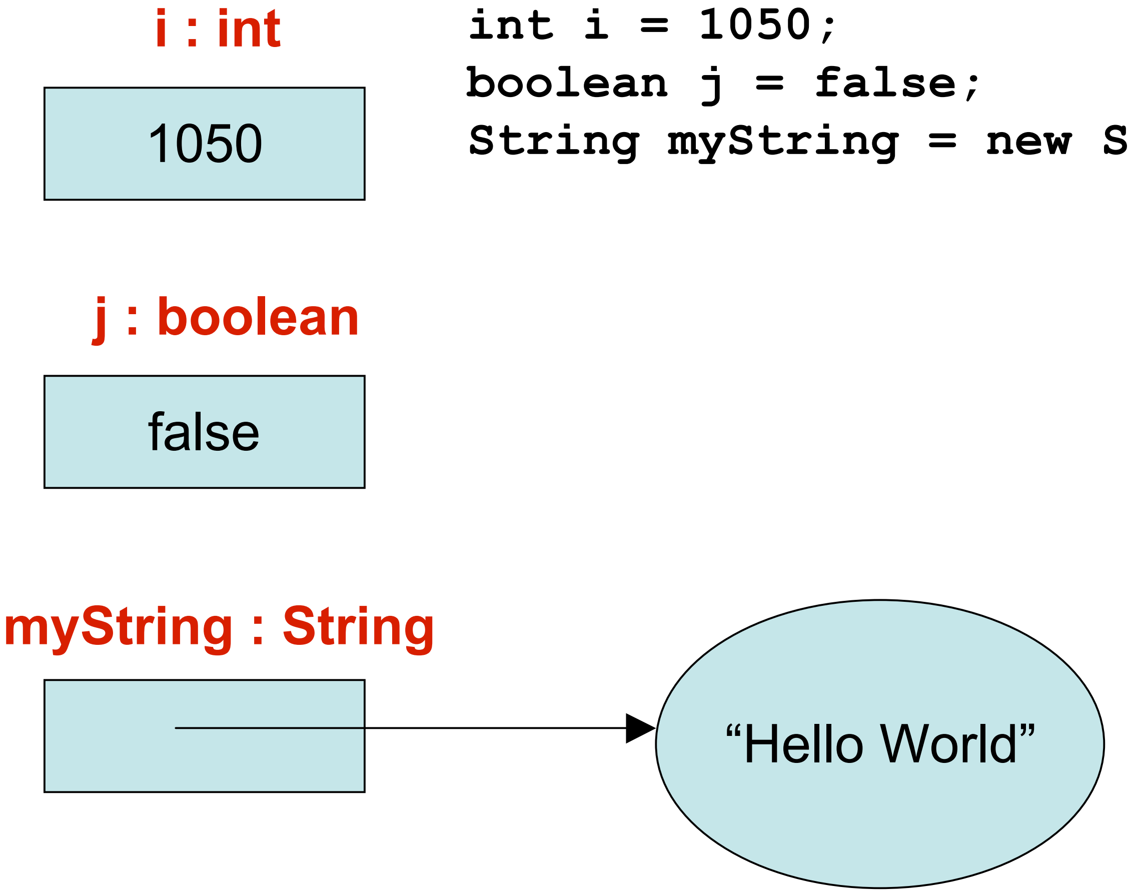
Objects

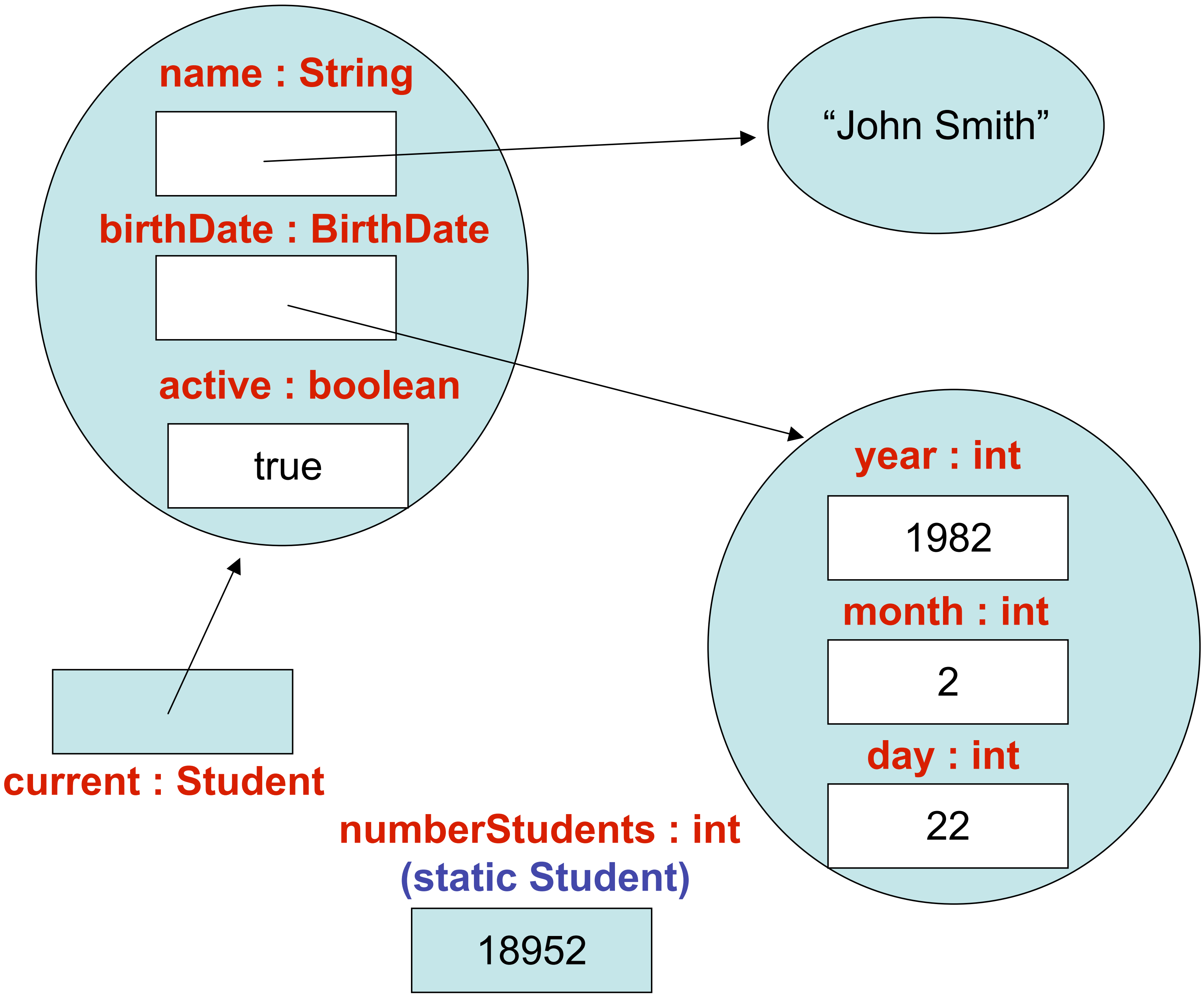
Drawn as circles, with internal boxes to represent data members

Strings are a ‘special case’ (see next slide)

Program:

```
int i = 1050;
boolean j = false;
String myString = new String(“Hello World”);
```




name : String
"John Smith"
birthDate : BirthDate
active : boolean
true
year : int
1982
month : int
2
day : int
22
current : Student
numberStudents : int
(static Student)
18952

Testing Reference Variable Values (==) vs. Object States (*.equals()*)

Equivalent References (==)

Tests whether the memory location referred to by reference variables is identical

(A == B): Does String variable A refer to the same memory location as String variable B?

Equivalent Object States (*.equals()*)

A method defined for the Object class, and overwritten by other Java classes (e.g. String) that normally tests for identical object states

(A.equals(B)): Does String variable A have the same state (characters) as String variable B?

WARNING: for Object, equals() and == *are the same*

Variable Scope (*another* property)

Definition

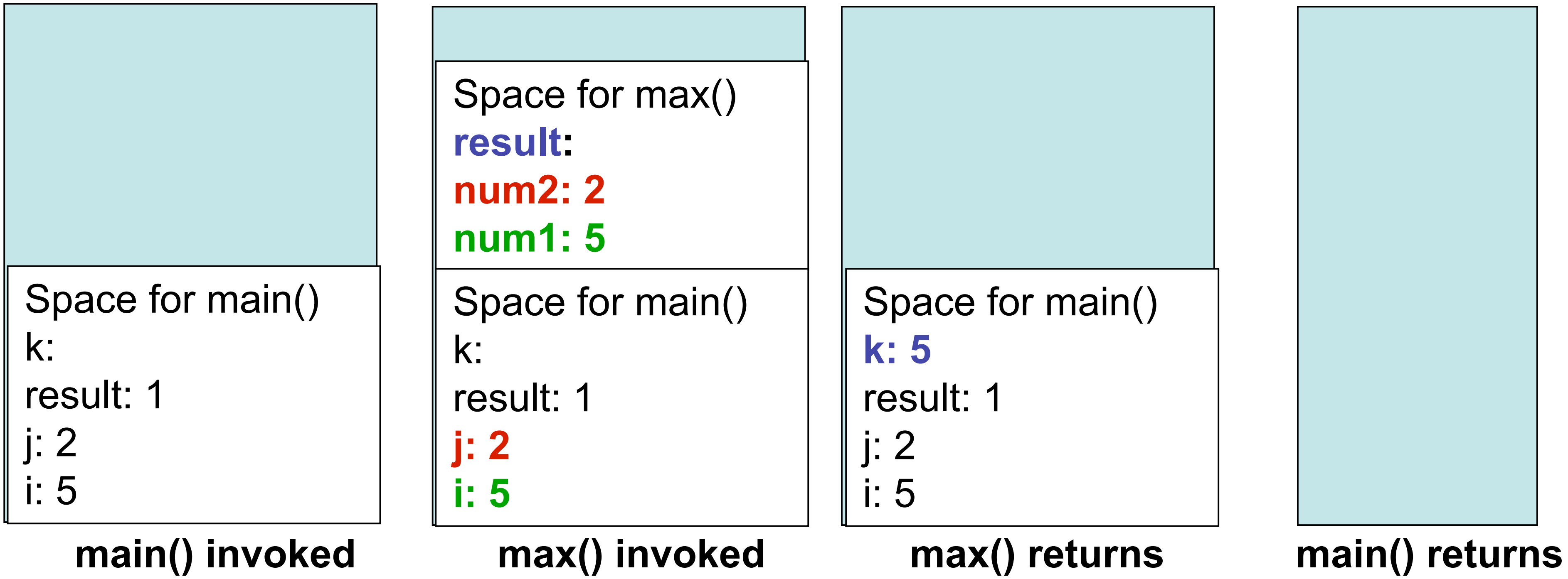
The program statements from which a variable may be referenced

Local variable

- A variable declared (and thus defined) within a given block (e.g. a loop index variable *i* within the outer block of a method)
- Local variables may be referenced only within the block in which they are declared (*locally,* by statements in the block)
- Formal method parameters are local variables that may be referenced within the body of a method
  - Actual parameters (arguments) provide initial value for formal parameters (Java has a pass-by-value semantics for parameters)

Masking or Shadowing

- Local variable has same name as variable in the outer scope; references in the local scope are to the *locally declared* variable
- Local variables may also mask instance or static variables in a method



Space for main()
k:
result: 1
j: 2
i: 5

main() invoked

Space for max()
result:
num2: 2
num1: 5

Space for main()
k:
result: 1
j: 2
i: 5

max() invoked

Space for main()
k: 5
result: 1
j: 2
i: 5

max() returns

main() returns

```
public class TestMax {
  public static int max(int num1, int num2) { int result; ...   ;
       return result;}

  public static void main(String[] args) {
    int i = 5, j = 2;
    int result = 1;
    int k = max(i, j);
    System.out.println(“Max is " + k + “ , result = “ + result);
  }
}
```

Arrays (which are *objects* in Java)

Use for Arrays

- Allow us to organize variables in a structure, rather than have a large set of unique names for every variable

Multi-dimensional Arrays:

We might represent 150 marks as:

- A one-dimensional array of 150 (double) floating point numbers

  -
    ```
    double[] marks = new double[150]; marks[0] = 95.1;
    ```
- A 2-D array of 15 (students) x 10 ((double) marks per student)

  -
    ```
    double[][] marks = new double[15][10]; marks[0][0]= 95.1;
    ```
- A 3-D array of 15 (students) x 5 (quizzes) x 2 ((double) mark for each section of each quiz, e.g. programming and short answer)

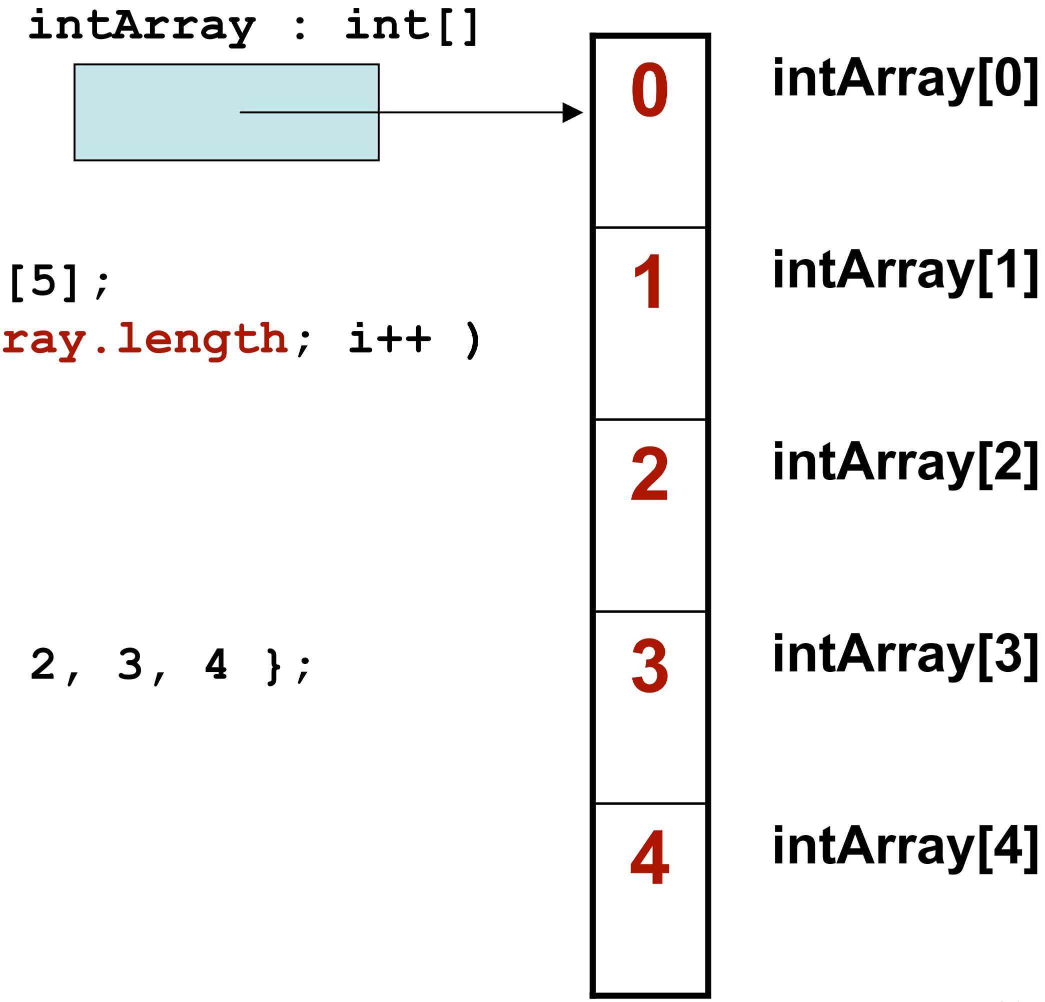
  -
    ```
    double[][][] marks = new double[15][5][2];
    marks[0][0][0] = 95.1; marks[0][0][1]=85.0;
    ```

Ragged Array (see pages 194-195)

- Array of 15 (students) x *different* sized arrays for each student, to represent the case where some students miss quizzes

  -
    ```
    double[][] marks = new double[15][];
     marks[0] = new double[2]; marks[1] = new double[3];
    ```
  - Possible because java implements 2D and higher dimensional arrays as *arrays of arrays*

1D Array: Example Memory Diagram



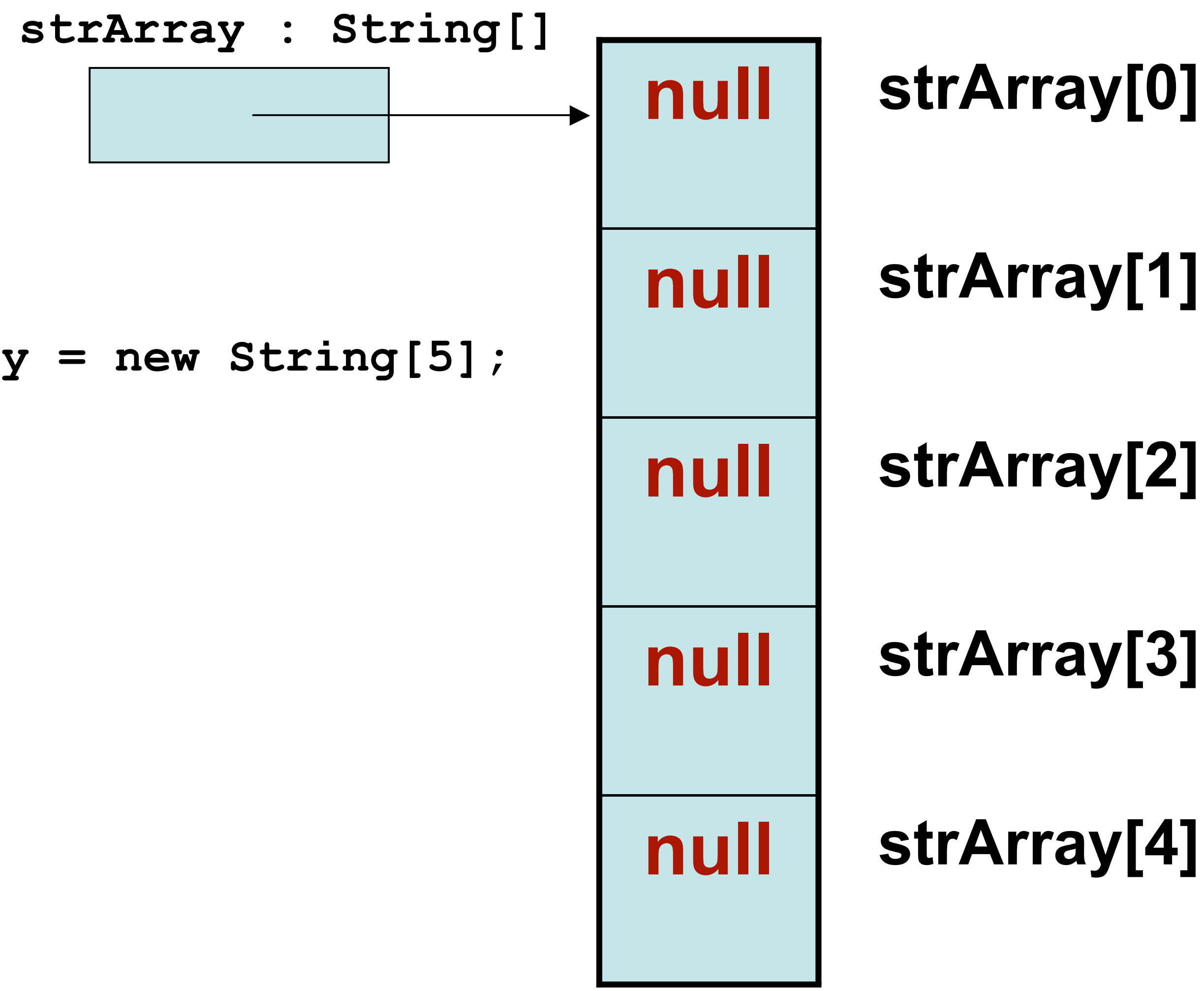
intArray : int[]

0	intArray[0]
1	intArray[1]
2	intArray[2]
3	intArray[3]
4	intArray[4]

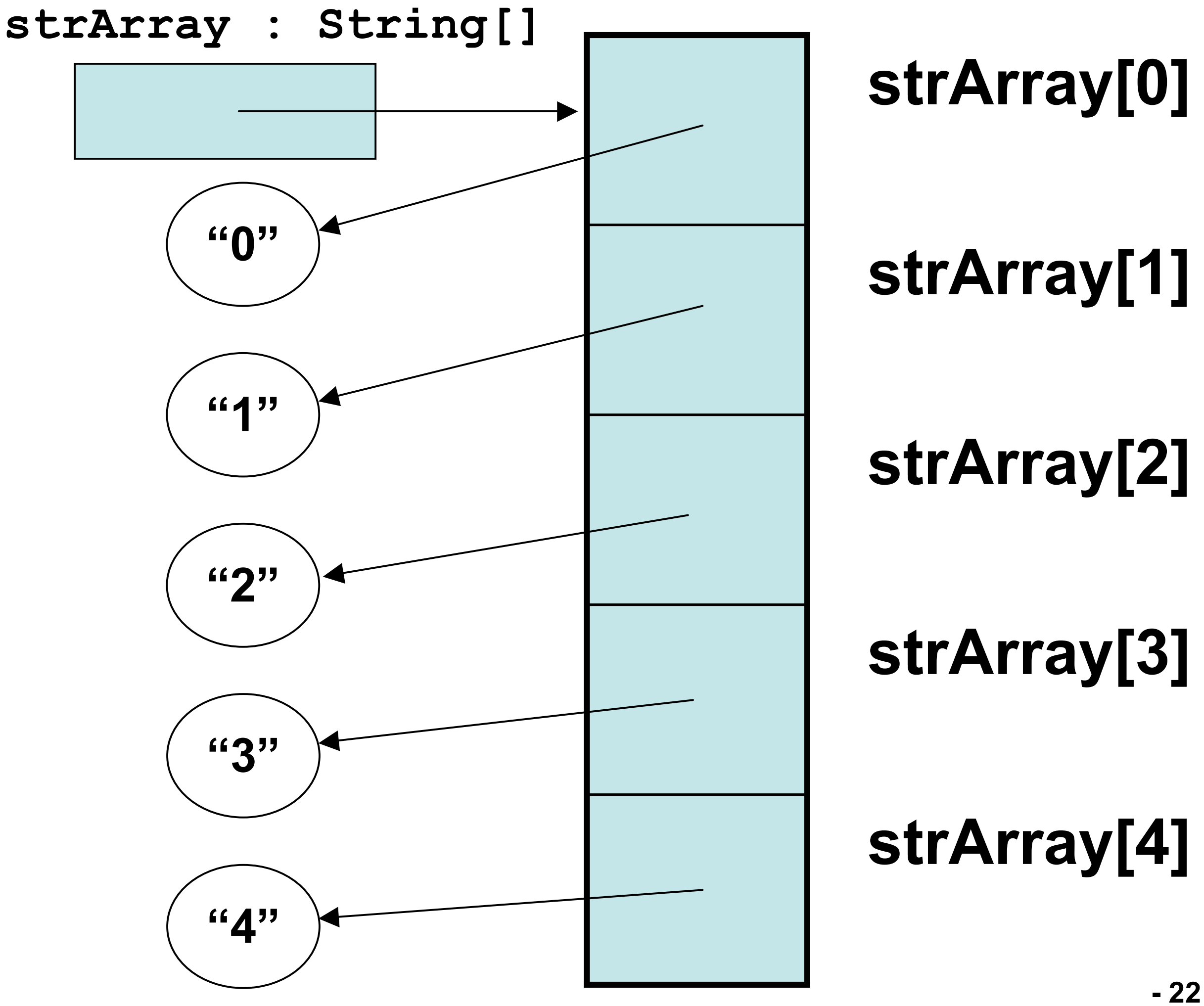
```
int intArray[] = new int[5];
for ( int i=0; i < intArray.length; i++ )
    intArray[i] = i;

      vs.

int intArray[] = { 0, 1, 2, 3, 4 };
```

1D Array: Another Example

```
String[] strArray = new String[5];
```



```
String[] strArray = new String[5];
for ( int i=0; i < strArray.length; i++)
    strArray[i] = new Integer(i).toString();
```



Numeric Type Casting

Type Casting

Changes the type *(representation or encoding)* of a variable or constant

Narrowing Conversion

Convert from a larger range of values to a smaller range of values

- e.g. int x = (int) 5.32;

Widening Conversion

Convert from a smaller to a larger range of values

- e.g. double x = (double) 5;

Exercise: Variables

A. What are the four variable properties we discussed? Name and define each in point form.

B. Draw memory diagrams for the following.

1. int x = 5; double y = 2.0; y = x;
2. String s1 = “first”; String s2 = “second”; s1 = s2;
3. String strArray[ ] = { “a”, “b”, “c”, “d” };
4. boolean f[ ][ ] = new boolean[3][2];
5. int y[ ][ ] = { { 1, 2, 3 }, { 4, 5, 6, 7} };

C. Are the following legal? Why?

1. double y = (double) 5;
2. int x = 5.0;
3. int x = (int) 5.0;

Exercise: Variables (Cont'd)

What is output by the following program? How do the definitions for x and y differ? What kind of parameter and variable is args?

```
class SimpleExample {
    static int x = 5;
    int y = 2;

    public static void main(String args[ ]) {
        int x = 4;
        y = 9;
        System.out.println(x + y);
    }
}
```

Review: Control Flow and Methods

Control Flow

Definition

- – The order in which statements in a program are executed
- – “Simple” control flow: sequential execution of statements
  - “do a, then b, then c”

Conditional Statements (if, switch)

Change control flow by defining different *branches* of execution followed depending on Boolean conditions (expressions)

- “if C is true then do a, else do b”
- “if C is true then do {a, then b, then c}, else do {d, then e, then f}”

Iteration Statements (while, do...while, for)

Change control flow by repeating a statement or block (*compound statement*) while a Boolean condition holds

- “while C is true, do {a, then b, then c }”

Method Invocation

Produces a “jump” to the instructions in a method invocation

(also changes context (e.g. defined local variables): see earlier “TestMax” example on slide 15))

Method Polymorphism (*Overriding)* vs. Method Overloading

Method Polymorphism (Overriding)

– Method redefines (‘overrides’) a method of the same name in the parent class (e.g. toString() is often overridden)

– Note similarity of *overriding* to *variable masking*

Method Overloading

Methods with different parameter lists but the same name.

```
public static int max(int num1, int num2)
public static double max(double num1, double num2)
```

overloaded methods must have different parameter types; you cannot overload methods based on modifiers or return types

What is ‘`this`’ ?

Definition

- In java, this is used within an instance method to refer to the object invoking the method
- Roughly: a reference to ‘me’ for an object
- All instance variable references and method invocations implicitly refer to ‘this’
  - (x = 2 *same as* this.x = 2; toString() *same as* this.toString())

Some Uses

1. Prevent masking of variables (e.g. formal params. And instance variables in a constructor:

   ```
   public MyClass(int x){ this.x = x;}
   ```

2. Invoke other constructors within a class
   - Note: this(arg-list) must be first statement in constructor definition

   ```
   public MyClass(int x){ this(); this.x = x;}
   ```

3. Have object pass itself as a method argument

   ```
   someClass.printFancy(this);
   ```

Exercise: Methods

A. What is produced as output by the following?

```
int a = 2;
switch (a) {
    case 2: System.out.println(“Case 2”);
    case 1: System.out.println(“Case 1”); break;
    default: System.out.println(“Default Case”); }
```

B. Answer the following in 1-2 sentences each.

1. In what way are an *if* statement and a *while* statement the same?
2. How do an *if* statement and a *while* statement differ?
3. What extra elements are added to the conditional test in a *while loop* to produce a *for loop*?

Exercise: Methods, cont’d

C. What is wrong with the following?

```
class MethodExample {
    private int x = 5;
    static private int y = 3;

    public int methodOne() {return methodTwo();}
    public int methodOne(int x) {this.x = x; return x;}
    static public int methodTwo() {return y + methodOne(2); }
    static public int methodTwo(int x) { this.x = x; return x;}
}
```

New Material: Inheritance

(Ch. 9 of Liang)

In OOP, What is Inheritance?

Definition

- A new class taking the definition of an existing class as the starting point for it's own definition.
- Represents 'is-a' relationship between derived and existing classes.

Superclass

The existing *("parent")* class providing the initial definition for the new *"derived"* or *"child"* class

Subclass

A class derived from an existing class (*"child class")*

In Java

Only accessible (e.g. non-private) data members and methods are `inherited' by a subclass. Constructors are also not inherited.

Inheritance is a formalized type of 'code-reuse'

Inheritance: A Simple Example

Superclass: OnlineStore

– Private Data: versionNumber, ...
– **Public Data: cash, inventoryValue, ...**
– Private Methods: computeInterest(), ...
– **Public Methods: getCash(), sale(String item, int quantity), ...**

Subclass 1: OnlineBookStore extends OnlineStore

– **Inherited Data: cash, inventoryValue**
– New Public Data: bookTitles, bookDistributors....
– **Inherited Methods: getCash(), sale(String item, int quantity), ...**
– New Public Methods: findISBN(String title), ...

Subclass 2: OnlineMusicStore extends OnlineStore

– **Inherited Data: cash, inventoryValue**
– New Public Data: albumTitles, musicDistributors, ...
– **Inherited Methods: getCash(), sale(String item, int quantity), ...**
– New Public Methods: getArtist(String albumTitle), ...

Subclass 3: OnlineScifiBookStore extends OnlineBookStore

– **Inherited Data: cash, inventoryValue, bookTitles, bookDistributors...**
– New Public Data: scifiOrganizations, scifiConferenceDates, ...
– **Inherited Methods: getCash(), sale(String item, int quantity), findISBN(String title), ...**
– New Public Methods: conferencesOn(Date day), ...

Inheritance in Java

Syntax

Use “extends” keyword

- (e.g. class NewClass **extends** AnotherClass { ... })

‘Object’ as the “Parent of them all”

All classes in Java extend (inherit from) the object class.

```
public class NewClass{} = public class NewClass extends Object{}
```

Multiple Inheritance

A class inheriting from more than one parent class

- Not permitted in Java
- Is permitted in other languages such as C/C++

See Figure 9.1: Geometric Object superclass, Circle and Rectangle subclasses (in course text)

UML diagram: + represents public, - private

The Java ‘super’ keyword

Purpose

Provides a reference to the superclass of the class in which it appears

Uses

1. Invoke a superclass constructor
   - Constructors are not inherited in Java
   - Similar to using ‘this,’ the call to ‘super(arg1, arg2, ...)’ must be the first statement in a constructor if present.
2. Invoke a superclass method that has been overridden
   - e.g. we can use super.toString() to invoke the toString() method of the superclass rather than that in the current class
   - Similar to ‘this,’ it is possible but not necessary to use super to invoke all inherited methods from the superclass (implicit)
   - **Warning:** we cannot ‘chain’ super, as in super.super.p()

The Inheritance Hierarchy and *Constructor Chaining*

Calling a constructor

Normally invokes default constructors for each class from root of the inheritance hierarchy, starting with *Object*

- This is necessary to ensure that all inherited data is properly initialized according to the class definitions.

e.g. public A() { } = public A() { super(); }

Example

Faculty class (see code, pg. 307 of Liang)

A Warning About Constructor Chaining in Java...

Default Constructor ("no-arg constructor")

Is automatically defined if no constructor is given by the programmer, otherwise it must be explicitly defined to exist

This Means...

That an error occurs if we go to construct an object and one of its ancestor classes in the inheritence hierarchy does not have a *default* constructor defined.

Fix:

If a class may be extended, explicitly define a default constructor to avoid this situation.

More naïve approach: always define a default constructor.

Overriding (Polymorphic) Methods

public String toString()

Defined in Object, normally overridden to give text summary of object state

- default output is “ClassName@HexAddress”

```
Loan loan = new Loan();
System.out.println(loan.toString())
====>(output) Loan@15037e5
```

Implementing Overriding in Java

Achieved by redefining an inherited method in a child class. Method signature must be the same.

e.g. in Circle, redefine toString() method inherited from Object:

```
public String toString() {
 return “A Circle with color: “ + color +
    “and is filled: “ + filled;}
```

Example:
Overriding (left) vs. Overloading (right)

See Section 9.5 in the course textbook for a comparison of these two concepts.

Inheritance and the Class Hierarchy (What happens if Class A inherits from Class B?)

Effect on Types

Objects from a class possess:

- The type (incl. data + methods) of the class itself
- The type (incl. data + methods) of the superclass
- The type (incl. data + methods) of the superclass' superclass
- ... and so on, up to the Object class in the class inheritance hierarchy.

Reference Variables

May invoke accessible methods of an object for the reference variable type, and any types that precede that type in the class inheritance hierarchy

String x = "Hi there."; // *String* and *Object* methods usable on x
Object a = x;   // Only *Object* methods may be invoked on a.

The `protected` access modifier

Decreasing Visibility:

1. public
2. protected
3. (default)   (no modifier)
4. private

See Figure 9.9 for an example of visibility across packages (roughly, directories containing class files)

Exercise

A. What is the printout of running class C?

```
class A {
    public A() {
        System.out.println(“Constructor A()”);
    }
}

class B extends A { }

public class C {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

B. What is wrong with the following?

```
class A {
    public A(int x) { }
}

class B extends A {
    public B() { }
}

public class C {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

C. True or False:

1. A subclass is a subset of a superclass
2. When invoking a constructor from a subclass, it's superclass's no-arg constructor is always invoked.
3. You can override a private method defined in a superclass

D. What is the difference between method overloading and method overriding?

E. Does every class have a toString() method? Why or why not?

F. What is wrong with this class? Also, draw a UML class diagram for class B (Circle may be represented using just a labeled box).

```
class B extends Circle {
    private double length;

    B(double radius, double length) {
        Circle(radius);
        length = length;
    }
}
```