CS1150
Principles of Computer Science
Final Review

Yanyan Zhuang

Department of Computer Science

http://www.cs.uccs.edu/~yzhuang

Numerical Data Types

Name	Range	Storage Size
`byte`	-2^7 to $2^7 - 1$ (-128 to 127)	8-bit signed
`short`	-2^{15} to $2^{15} - 1$ (-32768 to 32767)	16-bit signed
`int`	-2^{31} to $2^{31} - 1$ (-2147483648 to 2147483647)	32-bit signed
`long`	-2^{63} to $2^{63} - 1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: -3.4028235E+38 to -1.4E-45 Positive range: 1.4E-45 to 3.4028235E+38	32-bit IEEE 754
`double`	Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308	64-bit IEEE 754

- Reading for primitive data types:
  - https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Integer Division

+, -, *, /, and %

5 / 2 yields an integer **2**

5.0 / 2 yields a double value **2.5**

5 % 2 yields 1 (the remainder of the division) – often called modular operation

Java Identifiers

- An identifier
  - A sequence of characters that consist of letters, digits, underscores (_), and dollar signs ($)
  - No spaces
- Must start with a letter, an underscore (_), or a dollar sign ($)
  - It cannot start with a digit
- An identifier ***cannot*** be
  - A reserved word
  - true, false, or null
- An identifier can be of any length

Formatting decimal output

- Use DecimalFormat class
  - DecimalFormat df = new DecimalFormat("000.##");
  - System.out.println(df.format(celsius));
  - 0: a digit
  - #: a digit, zero shows as absent
    - 72.5 is shown as 072.5
    - 21.6666….. is shown as 021.67
  - More information https://docs.oracle.com/javase/tutorial/i18n/format/decimalFormat.html

Formatting decimal output

- Use System.out.format
  - System.out.format("the %s jumped over the %s, %d times", "cow", "moon", 2);
    - the cow jumped over the moon, 2 times
  - System.out.format("%.1f", 10.3456);
    - 10.3 // one decimal point
  - System.out.format("%.2f", 10.3456);
    - 10.35 // two decimal points
  - System.out.format("%8.2f", 10.3456);
    - 10.35 // Eight-wide, two decimal points

Variables and Constants

- Variable
  - Decimal numbers: to indicate float/double, use suffix f/d
    - Leaving off the suffix, the number defaults to a double
  - float floatValue = 71.71f;
    - If leave off “f” would get error: cannot convert double to float
  - double doubleValue = 12345.234d;
    - If you left off the “d” there is no issue
  - Use double (safe!)

Variables and Constants

- Constant
  - Used to store a value that will **NEVER** change
  - Constants follow certain rules
    - Must have a name (a meaningful name, like variables)
    - Name constants with all uppercase letters (Java convention)
    - Declared using the keyword **final**
      - Example: final double PI = 3.14159;
      - Let's look at code samples

Data Casting

- When you **explicitly** tell Java to convert a variable from one data type to another type
  - Think of data types as cups of different sizes
    - Can put the contents of a smaller variable (bottle) into a larger variable (bottle)
    - **Cannot** put the contents from a larger variable (bottle) into a smaller variable (bottle), **without losing information**
    - Cheat sheet: int (32 bits), double (64 bits)

Conversion Rules

When performing a binary operation involving two operands of different types, Java automatically converts the operand based on the following rules:

1. If one of the operands is double, the other is converted into double.
2. Otherwise, if one of the operands is float, the other is converted into float.
3. Otherwise, if one of the operands is long, the other is converted into long.
4. Otherwise, both operands are converted into int.

Augmented Assignment Operators

- +, -, *, / and % operators
  - Each can be combined with the assignment operator (=)
  - a = a + 1; => a += 1;
  - Same as -=, *=, /= and %=

Increment and Decrement Operators

- Increment: ++ Decrement: --
  - Operator can be placed before or after variables (postfix)

```
int i = 1, j = 3;
i++;     // Same as i = i + 1;   i will become 2
j--;     // Same as j = j – 1;   j will become 2
```

  - Alternatively (prefix)

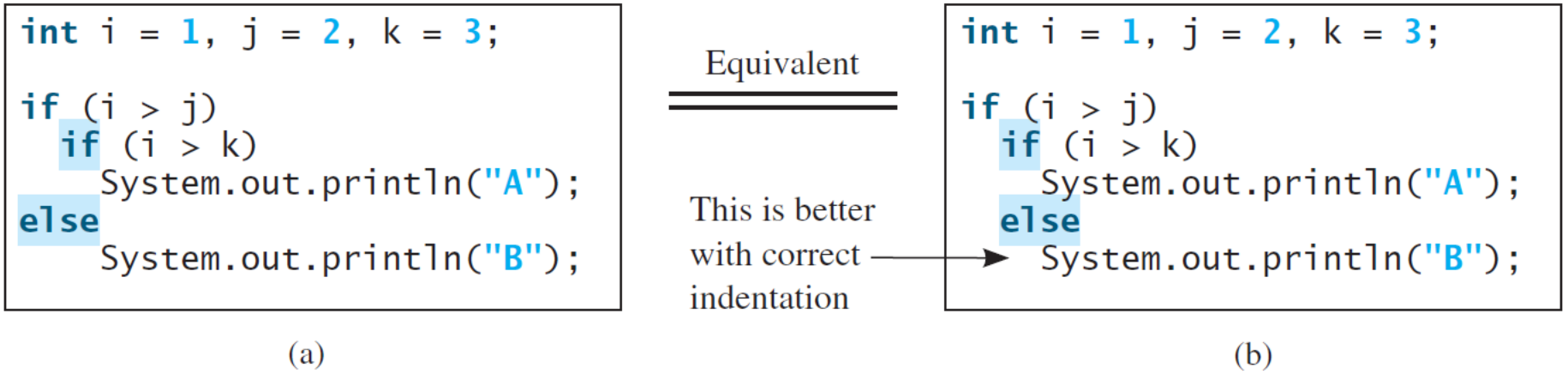
```
int i = 1, j = 3;
++i;     // Same as i = i + 1;   i will become 2
--j;     // Same as j = j – 1;   j will become 2
```

Order of Operators (Section 3.15)

- Anything in parentheses
- expr++ expr-- (postfix)
- + - ++expr --expr (unary plus/minus, prefix)
- (type) (Casting)
- **!** **(not)**
- * / % (multiplication, division, remainder)
- + - (binary addition, subtraction)
- < <= > >= (relational operators)
- == != (equality)
- **^** **(exclusive or)**
- **&&** **(and)**
- **||** **(or)**
- = += -= *= /= %= (assignment, augmented assignment)

If statement

- The else clause matches the most recent if clause



- An "else" always belongs with the most recent if

If statement

To force the else clause to match the first if clause, must add a pair of braces:

```
int i = 1, j = 2, k = 3;
if (i > j) {
  if (i > k)
    System.out.println("A");
}
else
  System.out.println("B");
```

`switch` Statement Notes

- switch expression
  - Must evaluate to a value of type char, byte, short, int
    - switch (x > 1) // Not allowed - evaluates to a boolean value
    - switch (x == 2) // Not allowed - another boolean expression

`switch` Statement Notes

- Case values
  - Are constants expressions
  - Cannot contain variables
    - case 0: system.out.println("......"); // valid
    - case (x+1): system.out.println("...."); // not valid
  - Though this is valid way to write the cases

```
int value = 3;
switch (value) {
    case 1:case 2:case 3: System.out.println("case 1, 2, and 3"); break;
    case 4: System.out.println("case 4"); break;
    default: System.out.println("default");
}
```

`switch` Statement Notes

- break statement

```
int day = 3;
switch (day) {
        case 1:
        case 2:
        case 3:
        case 4:
        case 5: System.out.println("Weekday"); break;
        case 0:
        case 6: System.out.println("Weekend");
}
```

Conditional Expressions

- Shortcut way to write a two-way if statement (if-else)
  - Consists of the symbols ? and : (aka the "ternary" operator)
  - result = expression ? value1 : value2
    - expression can be either a boolean value or a statement that evaluates to a boolean value
    - The conditional "expression" is evaluated
    - If the expression is true, value1 is returned
    - If the expression is false, value2 is returned

Rules for While/Do..while Loops

- The loop condition must be a boolean expression
  - Boolean expression must be in parentheses
  - Boolean expressions are formed using relational or logical operators
- Loop condition
  - Usually a statement **before** the while loop "initializes" the loop condition to true
  - Some statement within the loop body eventually change the condition to false
- If the condition is never changed to false, the program is forever in the loop
  - This is called an "infinite loop"
- Curly braces are not necessary if only one statement in loop
  - But best practice is to always include curly braces

Rules of for loops

- The control structure of the for-loop needs to be in parentheses
  - for **(i=0; i<= 2; i++)** { statements; }
- The loop condition (i <= 2) must be a boolean expression
- The control variable (i): not recommended to be changed within the for-loop body
- Curly braces are not necessary if only one statement in loop
  - Best practice is to always include curly braces

Using `break` and `continue`

- Break in loops
  - Used "break" in switch statements to end a case
  - Can be used in a loop to **terminate** a loop
  - Breaks out of loop
- Continue in loops
  - Used to end **current iteration** of loop
  - Program control goes to end of loop body

Note

- You may declare the control variable outside/within the for-loop

```
for (int j = 0; j <= 5; j++) {
  System.out.println ("For loop iteration = " + j);
}
 int j;
 for (j = 0; j <= 5; j++) {
  System.out.println ("For loop iteration = " + j);
}
```

- Note on variable scope (the area a variable can be referenced)
  - Declaring control variable **before the for loop** cause its **scope** to be inside and outside for-loop
  - Declaring the control variable **in the for-loop** causes its **scope** to be only inside the for loop
    - If I tried to use the variable j outside the for-loop - error

Note

- Mistakes to avoid
  - Infinite loops
  - Off-by-one error
- Nested loops
  - Know how to trace them

The `Math` Class

- Class constants:
  - PI (3.14159…)
  - E (2.71828…base of natual log)
- Class methods:
  - Trigonometric Methods
  - Exponent Methods
  - Rounding Methods
  - min, max, abs, and random Methods

Random Numbers

- Math.random()
  - How to generate a random integer between [lower, upper)?
    - Example: int lower=100, upper=120;
    - randomDouble = Math.random(); // [0.0, 1.0)
    - randomDouble = randomDouble * (upper-lower); // [0.0, 20.0)
    - randomDouble = lower + randomDouble; // [100.0, 120.0)
    - randomInt = (int) randomDouble; // cast double → int
  - Or in one step
    - randomInt = (int) (lower + Math.random() * (upper-lower));

Character Data Type

- Values: one single character
  - Use single quote ‘’ to represent a character (doubles quotes “” are for Strings)
    - char middleInitial = 'M';
    - char numCharacter = '4'; // Assigns digit character 4 to numCharacter
    - System.out.println(numCharacter); // Displays 4
  - Placing a character in “” it is no longer a char: it is a String
    - char middleInitial = "M"; // Error - cannot convert String to char

ASCII Code for Commonly Used Characters

Characters	Code Value in Decimal	Unicode Value
`'0'` to `'9'`	48 to 57	\u0030 to \u0039
`'A'` to `'Z'`	65 to 90	\u0041 to \u005A
`'a'` to `'z'`	97 to 122	\u0061 to \u007A

More information: http://kunststube.net/encoding/

Casting between char and Numeric Types

```
int i = 'a'; // Same as int i = (int)'a';

System.out.println (”i = " + i); // i = 97
```

all numeric operators can be applied to the char operands

```
char c = 97; // Same as char c = (char)97;

System.out.println ("c = " + c);  // c = a
```

Increment and decrement can be used on char variables to get the next or preceding ASCII/Unicode character.

```
char ch = 'a';
System.out.println(++ch); //shows character b
```

Comparing and Testing Characters

```
if (ch >= 'A' && ch <= 'Z')
  System.out.println(ch + " is an uppercase letter");
else if (ch >= 'a' && ch <= 'z')
  System.out.println(ch + " is a lowercase letter");
else if (ch >= '0' && ch <= '9')
  System.out.println(ch + " is a numeric character");
```

all numeric operators can be applied to the char operands

How to generate a random character?

- A random character between any two characters ch1 and ch2 with ch1 < ch2 can be generated as:
  - (char)(ch1 + Math.random() * (ch2 – ch1 + 1))
  - Example: random upper case letter
    - (char)('A' + Math.random() * ('Z'– 'A' + 1))
  - Example: random numeric character
    - (char)('0' + Math.random() * ('9'– '0' + 1))

How to convert a numeric int character to its int value?

- Converting ‘0’ to 0, etc.
- Example: how to convert ‘0’ to 0?
  - ‘0’ - ‘0’ is 0
  - ‘1’ - ‘0’ is 1
  - ‘2’ - ‘0’ is 2
  - ……

The String Type

- A **char** is in single quotes and a **String** is in double quotes
  - char middleInitial = "M"; // Error - can't convert String to char
  - char middleInitial = 'M'; // Correct

  - string studentName = "Max" // Error - uppercase "String"
  - String studentName = 'Max'; // Error - double quotes
  - String studentName = "Max"; // Correct

Strings and chars

- String methods (length, get char from String)
- Read Strings/chars from console
- Concatenate/compare Strings
- Converting between numbers and Strings
- Finding substrings
- Formatting output (%s, %d etc.)

Rules for Methods

- A method may or may not return a value
- A method must declare a return type!
  - If a method returns a value
    - Return type is the data type of the value being returned
    - The **return** statement is used to return the value
  - If a method does not return a value
    - Return type in this case is **void**
    - No return statement is needed (look at max no return example)
- The values you pass in must match the **order and type** of the parameters declared in the method

Overloading Methods -- Rules

- To be considered an overloaded method
  - Name - must be the same
  - Return type - can be different - but you cannot change ***only*** the return type
  - Formal parameters - must be different
- Java will determine which method to call based on the parameter list
  - Sometimes there could be several possibilities
  - Complier will pick the "best match"
- It is possible that the methods are written in way that the complier cannot decide best match
  - This is called ***ambiguous invocation***
  - This results in an error

Scope of Local Variables

- A local variable: a variable defined inside a method/block
- Scope: the part of the program where the variable can be referenced
- The scope of a local variable starts **from its declaration** and continues **to the end of the block** that contains the variable
  - A local variable must be declared before it can be used.

Scope of Local Variables, cont.

- **Can** declare a local variable with the same name multiple times in **different non-nesting** blocks in a method
- **Cannot** declare a local variable twice in nested blocks
- Formal parameters are considered local variables

Creating Arrays

Cannot do anything with an array variable until after the array has been constructed with the **new** operator:

```
arrayRefVar = new datatype[arraySize];
```

Example:

(1) creates an array with 10 double (i.e. it allocates memory)

```
myList = new double[10]; //use new to give a size
                         //allocate memory for array
```

(2) assigns the reference of the array to the variable myList

`myList[0]` references the first element in the array.

`myList[9]` references the last element in the array.

Accessing arrays

- For loops generally used with arrays since we know how many times the loop will occur
  - Example: assign the numbers 0 to 4 to numberList

```
// Assign the numbers 0 to 4 to numberList array
int[] numberList = new int[5];
for (int i = 0; i < 5; i++) {
        numberList[i] = i;
        System.out.println("numberList[" + i + "] = " + numberList[i]);
}
```

- Trying to access an element outside the range of an array it is an **error**

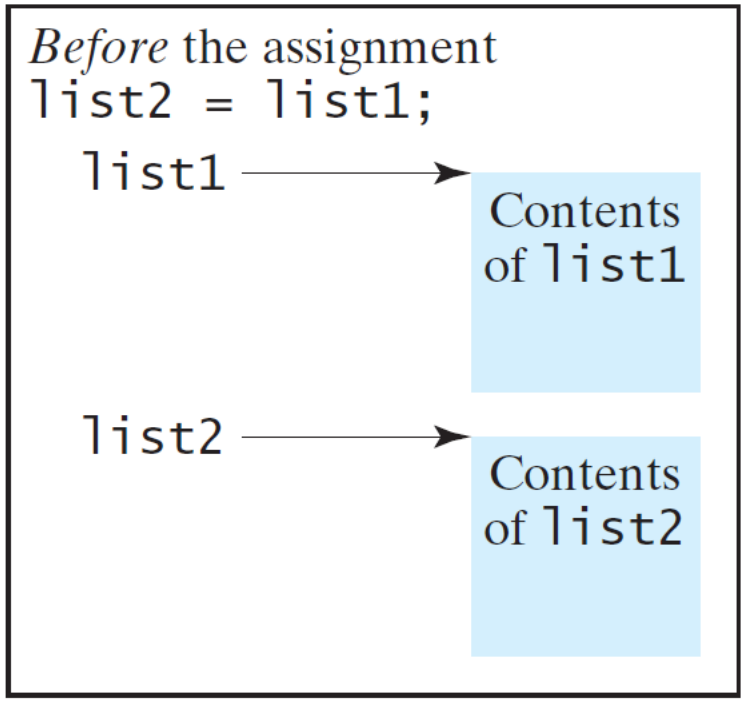
Processing Arrays

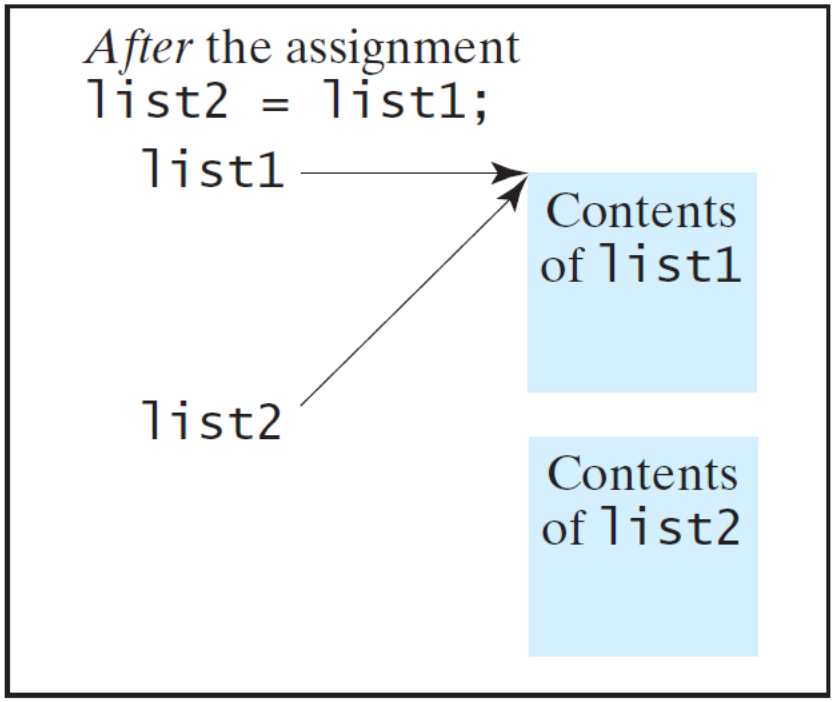
1. **(Initializing arrays with input values)**
2. **(Initializing arrays with random values)**
3. **(Printing arrays)**
4. **(Summing all elements)**
5. **(Finding the largest element)**
6. **(Finding the smallest index of the largest element)**
7. (*Random shuffling*)
8. (*Shifting elements*)

Copying Arrays

Often, in a program, you need to duplicate an array or a part of an array. In such cases you could attempt to use the assignment statement (=), as follows:

list2 = list1;





Copying Arrays

Using a loop:

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new
  int[sourceArray.length];


for (int i = 0; i < sourceArrays.length; i++)
    targetArray[i] = sourceArray[i];
```

Passing Arrays to Methods

```
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
```

Invoke the method

```
int[] list = {3, 1, 2, 6, 4, 2};
printArray(list);
```

Invoke the method

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Anonymous array

Pass By Value

Java uses *pass by value* to pass arguments to a method. There are important differences between passing a value of variables of primitive data types and passing arrays.

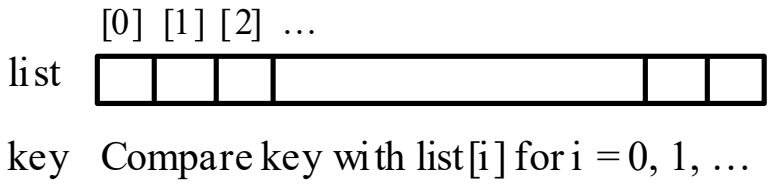
- For a parameter of a **primitive type** value, the actual value is passed. Changing the value of the local parameter inside the method *does not affect the value of the variable outside* the method.

- For a parameter of an **array type**, the value of the parameter contains a reference to an array; this reference is passed to the method. Any changes to the array that occur inside the method body *will affect the original array* that was passed as the argument.

Searching Arrays

Searching is the process of looking for a specific element in an array; for example, discovering whether a certain score is included in a list of scores. Searching is a common task in computer programming. There are many algorithms and data structures devoted to searching. In this section, two commonly used approaches are discussed, *linear search* and *binary search*.

```
public class LinearSearch {
  /** The method for finding a key in the list */
  public static int linearSearch(int[] list, int key) {
    for (int i = 0; i < list.length; i++)
      if (key == list[i])
        return i;
    return -1;
  }
}
```



Sorting Arrays

Sorting, like searching, is also a common task in computer programming. Many different algorithms have been developed for sorting. This section introduces a simple, intuitive sorting algorithms: *selection sort*.

Object state and behavior

- An object has two important pieces: **state** and **behavior**
- State
  - The properties (data fields) that define an object: **things an object knows!**
  - A "dog" object may have properties such as color, size, gender, etc.
- Behavior
  - The methods that define an object: **things an object does!**
  - A "dog" object may have behaviors such as sleep, fetch, rollover, bark, sit, etc.

Classes

Classes are constructs that define objects of the same type.

A Java class uses variables to define data fields and methods to define behaviors.

Additionally, a class provides a special type of methods, known as **constructors**, which are invoked to construct objects from the class.

Constructors, cont.

A constructor with no parameters is referred to as a *no-arg constructor*.

- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors are invoked using the **new** operator when an object is created. Constructors play the role of initializing objects.

Constructors cont.

- A constructor can be **overloaded**
  - public StudentD (){
    }
  - public StudentD (String lastName, String firstName){
    this.lastName = lastName;
    this.firstName = firstName;
    }
  - public StudentD (int ID, int level){
    this.studentID = ID;
    this.academicLevel = level;
    }

Accessing Object's Members

- Referencing the object's properties (array's length):

```
objectRefVar.data
```

e.g., `myCircle.radius`

- Invoking the object's method (String's length()):

```
objectRefVar.methodName(arguments)
```

e.g., `myCircle.getArea()`

Static Variables, Constants, and Methods

Static *variables* are shared by all the instances of the class. Static *constants* are final variables shared by all the instances of the class.

Static *methods* are not tied to a specific object.

To declare static variables, constants, and methods, use the **static** modifier.

The this Keyword

- The this keyword is the name of a reference that refers to an object itself. One common use of the this keyword is reference a class's *hidden data fields*.
- Another common use of the this keyword to enable a constructor to invoke another constructor of the same class.

Calling Overloaded Constructor

```
public class Circle {
  private double radius;

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

  public Circle() {
    this(1.0);
  }

  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
}
```

this must be explicitly used to reference the data field radius of the object being constructed

this is used to invoke another constructor

Every instance variable belongs to an instance represented by this, which is normally omitted

Scope of Variables

- The scope of instance and static variables is the entire class. They can be declared anywhere inside a class.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be initialized explicitly before it can be used.

Visibility Modifiers

By **default**, a class, variable, or method can be accessed by any class in the same package.

- `public`

  The class, data, or method is visible to any class in any package.

- `private`

  The data or methods can be accessed only by the declaring class.

The getter and setter methods are used to read and modify private properties.

Inheritance

- When the definition of a class is based on an existing class (called the **superclass**)
- The class that is inheriting (**subclass**) can use accessible date fields and methods from superclass

Using the Keyword `super`

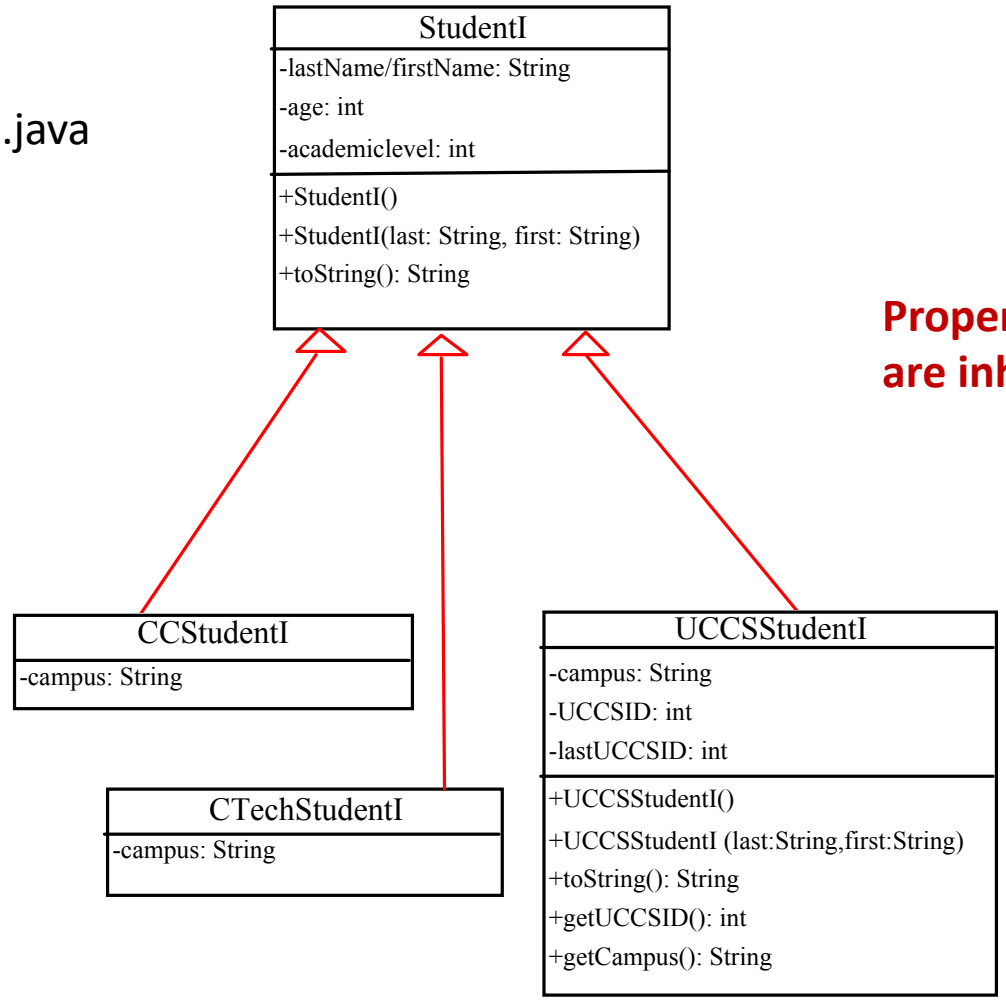
The keyword `super` refers to the superclass of the class in which `super` appears.

This keyword can be used in two ways:

- To call a superclass constructor: super()
- To call a superclass method: super.method()

Superclasses and Subclasses

Example:
StudentApp9.java



Properties and methods are inherited

Overriding vs. Overloading

- Overloading
  - We discussed overloading in methods chapter
  - Two or more methods with the **same name** but **different formal parameters**
  - The methods could be in the same class or in different classes related by inheritance
- Overriding
  - Occurs when dealing with inheritance
  - A method defined in the subclass that **matches** the *signature and return type* of the method defined in superclass

Good Luck!!