Lecture Notes

1. Comments
   a. /*   */
   b. //

2. Program Structures
   a.

```
public class ComputeArea
{
   public static void main(String[ ] args)
   {
         // input radius
         // compute area ← algorithm
         // output area
   }
}
```

Actions to be executed Order of execution of the actions

   b.

```
public class ComputeArea
{
   public static void main(String[ ] args)
   {
      // declare variables
         double radius;
         double area;

      // assign value
         radius = 20;

      // compute area
         area = radius *  radius * 3.14159;

      // output area
         System.out.println("Circle radius: " + radius + " area: " + area);
   }
}
```

Data Types

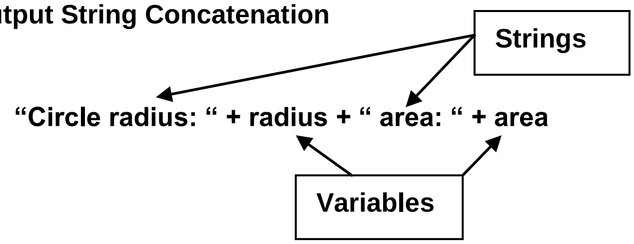
char	character	8-bit unsigned
byte	integer	8-bit signed
short	integer	16-bit signed
int	integer	32-bit signed
long	integer	64-bit signed
float	floating point	32-bit signed
double	floating point	32-bit signed

statement	radius type	radius value	area type	area value
double radius;	double	no value		
double area;			double	no value
radius = 20;		20		
area = radius * radius * 3.14159;				1256.636

3. String Concatenation
   System.out.println("Circle radius: " + radius + " area: " + area);

String ← "Circle radius: " String ← " area: " String Concatenation ← "Charles " + "Putnam"

4. **Output String Concatenation**



System.out.println(“Introduction to Java Programming, “ +
By Y. Daniel Liang”);

writing an output string on multiple lines

5. **Identifiers** **(variables, constants, methods, classes, packages)**
   a. sequence of Characters (Letters; Digits; Underscores, i.e., “_”; Dollar Sign, i.e., “$”)
   b. cannot contain spaces
   c. cannot start with a Digit
   d. normally starts with a Letter
   e. starts with an Underscore under specific situations
   f. the $ character should only used in mechanically generated source code
   g. cannot be a reserved word (see Appendix A)
   h. cannot be TRUE, FALSE, NULL
   i. can be of any length

   Legal Identifiers
   $L5, $_L5, M15, M15_a, _ks12

   Illegal identifiers
   35M, M24+6, LM 5 (no spaces allowed)

6. Java is <u>case sensitive</u>, i.e., Mag, mag, MAG, mAg, maG, etc. are all different identifiers

7. Identifiers are used for naming **<u>variables</u>**, **<u>constants</u>**, **<u>methods</u>**, **<u>classes</u>**, and **<u>packages</u>**

8. **Variable Declaration**
   - provides the <u>allocation of memory space</u> appropriate for the data type requested
   - by convention, single-word variable names are lower case
   - if a variable name consist of more than one word,
     - the words are concatenated
     - the first word is lower case
     - all subsequent words are capitalized

       e.g., double interestRate;
       double dailyCompoundInterest;
   - int x, y, z;

9. **Assignment Statements**

   int x = 1; x 🡐 1 x [1]

   The value assigned must be compatible with the data type of the variable **hence** int x = 1.0; is invalid

   double radius = 2.5; radius 🡐 2.5; radius [2.5]

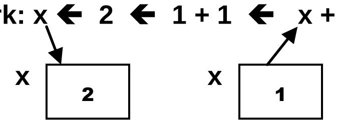
10. Assignment Expressions

x = 5 * (3/2) + 3 * 2;

area = radius * radius * 3.14159;

x = x + 1;

Remark: x 🡐 2 🡐 1 + 1 🡐 x + 1



In mathematics, the "=" symbol denotes equality, hence x = x + 1 implies that 1 = 0 which leads to a contradiction in any number system with a base > 1.
In most programming languages, the "=" symbol denotes replacement as indicated to the left of this box.

For Java, C, & C++

Assignment Statements are treated as an Expression that evaluates to the value being assigned on the left-hand side of the assignment variable, e.g.,

- System.out.println(x = 1); 🡐🡒 { x = 1; & System.out.println(x); }
- i = j = k = 1; 🡐🡒 { k = 1; j = k; i = j; }

11. Initializing Variables

A variable must be declared before it Is given a value.
A variable **declared in a method**, must be assigned a value before it can be used.

int i, j, k = 2, m = k + 3; n = 5 * m;

12. Constants

- permanent data – never changes
- constants must be declared and initialized in one statement
- by convention, constant names are always UPPERCASE

final double PI = 3.14159;
area = radius * radius * PI;

PI = PI + 1; 🡒 error message invalid operation

- descriptive name for a constant
- value isolated to one location

13. Number Systems

a. Octal Numbers
b. Binary Numbers
c. Hexadecimal Numbers
d. 2's Complement Arithmetic

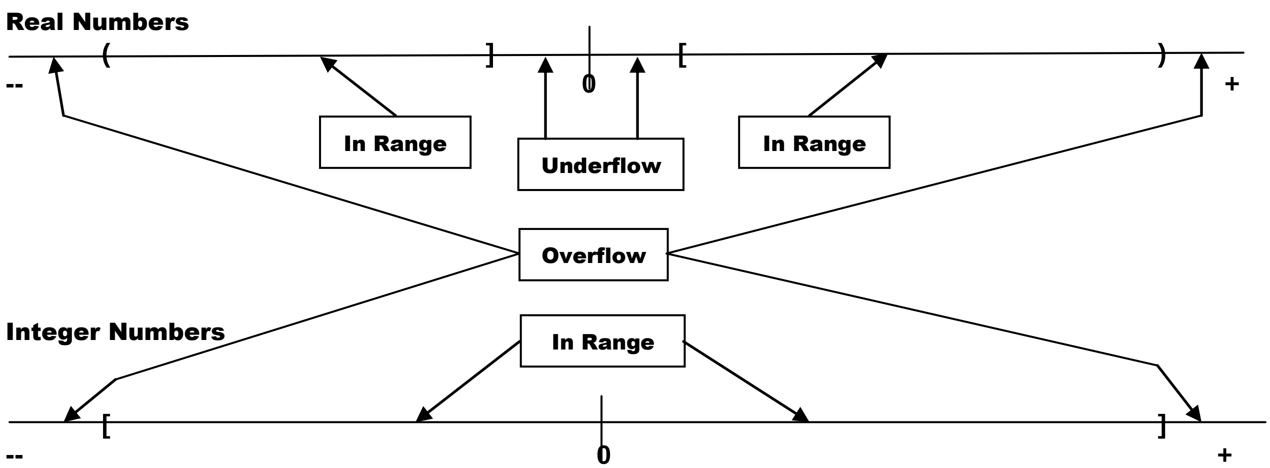
14. Numeric Data Types

Data Types

char	**character 8-bit unsigned**		
byte	**integer 8-bit signed**	**[-2^{7}, $2^{7} - 1$]**	**[-128, 127]**
short	**integer 16-bit signed**	**[-2^{15}, $2^{15} - 1$]**	**[-32768, 32767]**
int	**integer 32-bit signed**	**[-2^{31}, $2^{31} - 1$]**	**[-2,147,483,648, 2,147,483647]**
long	**integer 64-bit signed**	**[-2^{63}, $2^{63} - 1$]**	
float	**floating point 32-bit signed**		**single precision**
		negative range	**[$-3.4028235 * 10^{38}$, $-1.4 * 10^{-45}$]**
		positive range	**[$1.4 * 10^{-45}$, $3.4028235 * 10^{38}$]**
double	**floating point 32-bit signed**		**double precision**
		negative range	**[$-1.7976931348623157 * 10^{308}$, $-4.9 * 10^{-324}$]**
		positive range	**[$4.9 * 10^{-324}$, $1.7976931348623157 * 10^{308}$]**

15. Overflow/Underflow

- Overflow – value too large for variable data type
- Underflow -- value too small for variable data type



- **Floating-point numbers are not stored with complete accuracy, results of calculations are approximate!**
- **Integer numbers are stored with complete accuracy, calculations with integers yield exact results!**
- **Java reports neither warnings nor errors on overflows/underflows!**

16. Numeric Operators

- Addition + binary operator ←→ two operators
  unary operator ←→ one operator
- Subtraction - binary operator ←→ two operators
  unary operator ←→ one operator
- Multiplication *
- Division /
- Remainder %

17. Integer Division

```
int   z,  r,  x = 11,  y = 3;
z = x/y;          ➔ z  [3]
r = x%y;          ➔ r  [2]
```

18. Computing Time

```
int seconds = 500;
int minutes = seconds/60; ➔ minutes [8]

seconds = seconds%60; ➔ seconds [20]
```

19. Numeric Literals

constant values that are used in statements

```
int seconds = 500;
int minutes = seconds/60;
```

literals

a. Integer Literals

- an integer literal can be assigned to an integer variable as long as it fits the data type
- integer literal with a value between $[-2^{31}, 2^{31} - 1]$ is assumed to be of type int
- to denote an integer literal of type long, append the letter “L” on the end of the number, i.e., long n = 2147483648L;
- an integer literal without a leading zero is assumed to be of base 10, i.e., a decimal number, e.g., int i = 37; (decimal)
- an integer literal with a leading zero is assumed to be of base 8, i.e., an octal number, e.g., int j = 037; (octal)
- an integer literal with a leading 0x is assumed to be of base 16, i.e., a hexadecimal number, e.g., int k = 0x37; (hexadecimal)

b. Floating-Point Literals
- decimal point required when writing a floating-point literal
- a floating-point literal with an "f" or "F" suffix is of type **float**
  float x = 3.14159f;   float x = 3.14159F;
- a floating-point literal with a "d" or "D" suffix is of type **double**
  double x = 3.14159d;   double x = 3.14159D;
- a floating-point literal without a suffix is assumed to be of type **double**

c. Scientific Notation
- 1.23456e+2 ↔ 1.23456e2 ↔ 1.23456 * 10^{2} ↔ 123.456
- 1.23456e-2 ↔ 1.23456 * 10^{-2} ↔ 0.0123456

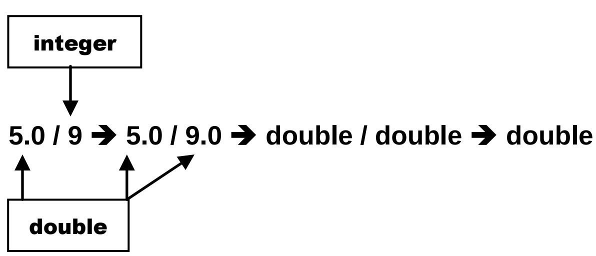
20. Evaluating Expressions

a. Evaluate operators contained inside parentheses
b. Nested parenthesis
   i. evaluate operators contained inside innermost parentheses
   j. evaluate operators contained inside outer parenthesis
c. Evaluate multiplication, division & remainder operators (evaluate operators left to right)
d. Evaluate addition & subtraction operators (evaluate operators left to right)

21. Fahrenheit → Celsius Conversion

```
double fahrenheit = 100;
double celsius = (5.0/9) *  (fahrenheit – 1);
```

5/9 → 0
5.0/9 → 0.5555555…



22. Shorthand Operators

int x = 17, a = 3, n = 7, m;

•	+=	x += a;	x = x + a;	a = 3; x = 17;	x += a; x	20
•	-=	x -= a;	x = x – a;	a = 3; x = 17;	x -= a; x	14
•	*=	x *= a;	x = x * a;	a = 3; x = 17;	x *= a; x	51
•	/=	x /= a;	x = x / a;	a = 3; x = 17;	x /= a; x	5
•	%=	x %= a;	x = x % a;	a = 3; x = 17;	x %= a; x	2

•	n++;	n 8	m = n++; assignment ➔ increment	m 8	n 9	post-increment operator
•	n-- ;	n 6	m = n--; assignment ➔ decrement	m 6	n 5	post-decrement operator
•	++n;	n 8	m = ++n; Increment ➔ assignment	m 9	n 9	pre-increment operator
•	-- n;	n 6	m = --n; decrement ➔ assignment	m 5	n 5	pre-decrement operator

Valid Use:

a. For

```
int x = 17, a = 3, n = 7, m;
m = ++x - --a + n--;
```

yields m 23

b. System.out.println(x %= 4);

prints "1"

Invalid Use:

For

```
int x = 17, m;
m = ++x + x--;
```

yields either 36 or 37 since the value of m is indeterminate, i.e., it is not specified in the Java language specifications!

Remark: The shorthand operators can be used with both integer and floating point variables with the proviso that the % operator is not defined for floating point variables.

23. Numeric Conversions (in computations)

a. Example

```
byte i = 9;
long k = (i + 5)/2;
double d = (i – 3) + k*4;
```

b. Rules of Numeric Conversion

- **if one of the operands is a double**
  then convert the other operands to doubles
- **otherwise, if one of the operands is a float**
  then convert the other operands to floats
- **otherwise, if one of the operands is a long**
  then convert the other operands to longs
- **otherwise convert all operands to ints**

c. Range of Numeric Increases

byte → short → int → long → float → double

Type Casting is an operation that converts a value of a specific data type into a value of another data type, e.g., for int n = 3; float m; the assignment m = (float) n; is permissible!

- (float) n converts the value of n into a floating point number;
- m = (float) n; assigns the floating point number (float) n to the variable m.
- casting does not change the data type of the variable but only ***the data type of the value***

- It is always possible to assign a value to a numeric variable whose type supports a larger range of values, e.g., for short n = 3; long m; the assignment m = n; is permissible! Explicit type casting is not required; type casting is implicit!
- To assign a value to a numeric variable whose type supports a smaller range of values is permissible only if casting is used, e.g.,
  - for short n; long m = 3; the assignment n = m; is not permissible!
  - for short n; long m = 3; the assignment n = (short)m; is permissible but lost information may lead to inaccurate results!
  - for long n; float m = 3.7; the assignment n = (long) n; is permissible but the floating point number 3.7 is truncated to the long integer 3, i.e., information is lost!
- **Type Widening**
  - casting a variable of a type with a small range to a variable of a type with a larger range
  - performed automatically without explicit casting
- **Type Narrowing**
  - casting a variable of a type with a larger range to a variable of a type with a smaller range
  - must be explicitly performed

- **Use of Casting in Computations**

```
double purchaseAMOUNT = 197.55;
double tax = purchaseAMOUNT * 0.06;

System.out.println("Sales tax: " + (int)(tax * 100) / 100.0);
```

➔

Sales Tax: 11.85

24. Character Data Type & Operations

a. character data type variable holds only a single character, e.g.,

```
char ch = 'z';
```

b. a character literal is a single character enclosed in single quotation marks, i.e., apostrophes, e.g., 'z'

'z' requires one storage location

c. a string literal is one or more characters enclosed in quotation marks, e.g.,

"Putnam" and "A" are both strings

"Putnam" requires seven (7) storage locations

"A" requires two (2) storage locations

d. ASCII Code

8-bit ➔ 256 characters

e. Unicode Code

16-bit code ➔ 65,536 characters

ASCII subset \u0000 … \u007F

'A' ⇔ \u0041 ⇔ 41_{16} ⇔ 65_{10}

See ASCII Table Appendix B Liang

Supplementary code ➔ 1,112,064 characters

Remark: char ch = 'A'; ch++; ➔ ch | B |

f. Escape Sequences (Special Characters)

- \b backspace \u0008
- \t tab \u0009
- \n linefeed \u000A
- \f formfeed \u000C
- \r cr (return) \u000D
- \\ backslash \u005C
- \' single quote \u0027
- \" double quote \u0022

```
System.out.println("\tHello World\rGlobal Warming is fun\b\b\b=== serious");
```

➔

<tab>Hello World

Global Warming is ~~fun~~ serious");

g. **Character Data Conversion**

char ch = (char)0xAB0041; ➔ lower 16 bits is assigned to ch ➔ ch → [41_{16} ➔ 65_{10} ➔ 'A']

char ch = (char) 65.25; ➔ 65.25 is converted to an integer 65_{10} which is assigned to ch ↑

int i = '2' + '3'; ➔ i = { (int) '2' ➔ 50_{10} & (int) '3' ➔ 51_{10} } hence i contains [50_{10} + 51_{10} ➔ 101_{10}]

int j = 2 + 'a'; ➔ (int)'a' ➔ 97_{10} & (char) j ➔ 'c'

int d = 'a' – 'A'; ➔ d [32_{10}]

conversion of lowercase ch to uppercase ch1

```
char ch1 = (char)('A' + (ch – 'a'));
```

conversion of uppercase ch to lowercase ch1

```
char ch1 = (char)('a' + (ch – 'A'));
```

25. String Type

String msg = "Hello World"; ➔ msg [Hello World]

- **String is a predefined class in the Java library**
- **String is not a primitive type; it is a *reference type***
- **Byte, short, int, long, float, double, & char are *primitive types***

String first, last, complete, filename;

first = "Charles"; last = "Putnam"; complete = first + " " + last; ➔ complete [Charles Putnam]

fileName = "Grades" + 2010; ➔ fileName [Grades2010]

for int i = 1, j = 2;

System.out.println("i + j is " + i + j); ➔ i + j is 12
first concatenation "i + j is " + i ➔ "i + j is 1"
second concatenation "i + j is 1" + j ➔ "i + j is 12"

System.out.println("i + j is " + (i + j)); ➔ i + j is 3

26. Scanner Class (Input Operations)

a. **System.out** refers to the ***Standard Output Device*** → <u>console</u> (default)
println method displays primitive values &/or strings to the console

b. **System.in** refers to the ***Standard Input Device*** → <u>keyboard</u> (default)
input is not directly supported by java, i.e., there does not exist a "<u>readln</u>" method that allows direct input such as <u>println</u> supports output

Input requires the <u>use of the Scanner class</u> to build an ***object*** to read input from System.in, i.e., the **Standard Input Device**, e.g.,

```
Scanner input = new Scanner(System.in);
```

assigns the object reference to the variable input

creates an object of the Scanner class

creates a variable of the Scanner type

c. Methods contained in Scanner Objects

- nextByte( ) — reads an integer of the **byte** type
- nextShort( ) — reads an integer of the **short** type
- nextInt( ) — reads an integer of the **int** type
- nextLong( ) — reads an integer of the **long** type
- nextFloat( ) — reads an integer of the **float** type
- nextDouble( ) — reads an integer of the **double** type
- next( ) — reads a string that ends before a WHITESPACE character
  e.g., **' ', '\t', '\f', '\r', '\n'**
- nextline( ) — reads a line of characters, i.e.,
  a string ending with a LINE SEPARATOR

d. Input Statement

```
System.out.print("Enter double value: ");   ← prompt for input statement
Scanner input = new Scanner(System.in);     ← create Scanner object
double d = input.nextDouble( );             ←
```

use the nextDouble( ) method of the Scanner object input **to read a value into the double variable** d

```
int i = input.nextInt( );
long L = input.nextLong( );
short s = input.nextShort( );
byte b = input.nextByte( );
float f = input.nextFloat( );
String s = input.next( );
String s1 = input.nextLine( );
```

e. Print Statements

- **println(...);** ➔ prints the information & moves the cursor to the next line
- **print(...);** ➔ prints the information & keeps the cursor on the same line

27. Case Studies – read Liang pages 46-51 (important to ask questions)

28. Programming Style & Documentation

a. Comments

i.	Single line comments	// …	use within methods
ii.	Block comments	/* … */	use for header information, i.e., name, etc.
iii.	javadoc comments	/** … */	can be extracted into a HTML file see www.java.sun.com/j2se/javadoc extraction will not be used in Comp 110 **use for comments on entire class or method; must be placed before class or method**

b. Naming Conventions

- choose **descriptive names** with meanings related to the intended purpose
- in general, do not choose abbreviations, use complete words
- names are **case sensitive**
- names for variables & methods
  - single word names should be lower case
  - multiple word names
    - first word should be lower case
    - capitalize the first letter of each subsequent word
    - concatenate the words, e.g., **accountDue**
    - do not leave blank spaces in a name, e.g., **account Due** is not a proper name
    - the underline character may be used to separate words within a name, e.g., **account_Due**
- names for classes
  - Capitalize the first word of each word in a class name
  - Do not choose class names that are in the Java Library

    Hint: If the program encounters problems when compiling, one area to consider is that you have chosen a name that is in the Java Library

- names for constants
  - Capitalize all letters in each word constant name
  - Use the underline character to separate each word of the name, e.g., PI, MIN_MAX, etc.

c. Spacing

- i = j + k / 2; proper style
- i=j+k/2; improper style – difficult to read

d. Indentation

Proper Indentation for Comp 110 (next line block style)

```
public class ComputeArea
{
   public static void main(String[ ] args)
   {
      // declare variables
                double radius;
                double area;

      // assign value
                radius = 20;

      // compute area
                          area = radius *  radius * 3.14159;

      // output area
      System.out.println("Circle radius: " + radius + " area: " + area);
   }
}
```

Easy to find alignment errors, i.e., easy to determine unbalanced brackets, e.g., follow vertical lines

2 – 3 spaces indentation; more than 4 spaces may make the program difficult to read; for a large program, such large indentations may make it difficult to see the entire program on a monitor or to print a readable hard copy!

Improper Indentation for Comp 110 (end of line block style) (used by Liang)

```
public class ComputeArea {

   public static void main(String[ ] args) {
      // declare variables
                double radius;
                double area;

      // assign value
                radius = 20;

      // compute area
                          area = radius *  radius * 3.14159;

      // output area
      System.out.println("Circle radius: " + radius + " area: " + area);
   }
}
```

Difficult to find alignment errors, i.e., unbalanced brackets are not easy to spot.

29. Programming Errors

a. Syntax Errors

- **detected during compilation**
- **errors in code construction**
  - **mistyping keywords**
  - **omitting punctuation**
  - **mismatched braces – missing "{" or "}"**
- **compilation error messages**
  - **line number**
  - **"^" indicator, e.g., Systm.out.println("Age: " + i);**
    ^
- **removing errors**
  - **start at top of the document**
  - **remove first error**
  - **working down through the document, remove all understandable errors**
  - **recompile**
  - **repeat as required**

b. Runtime Errors

- **detected by abnormal termination of the program runtime**
- **environment detects an operation that is impossible to carry out**
- **typically caused by input or computational errors**
  - **input a floating point number into a variable designed for the long data type**
  - **divide a number by zero**
- **runtime termination error messages**

c. Logic Errors (bugs)

- **program contains neither syntax nor runtime errors**
- **program does not perform as it was intended**
  - **does not produce the correct output**
  - **does not terminate correctly**
  - **etc.**

30. Debugging

a. Trace program -- check variable values during runtime

- **Hand-trace**
- **Insert print statements**
- **Debugging software – JDK command line debugger – jdb**
  > jdb Hello.java
  - **Execute a single statement at a time**
  - **Step over a method**
  - **Execute each statement in a method (trace a method)**
  - **Set breakpoints for specific statements – program stop at each breakpoint**
  - **Display the content of selected variables**
  - **Modify the content of selected variables**
  - **Display call stacks –**
    - **trace method calls**
    - **view lists of all pending calls**

b. Surgery

- **Selectively comment sections of code**
- **Recompile & execute new program, looking for areas which produce the errors**

c. **Review the Design**
   - **Check areas of the design documents that could produce the errors**
   - **Be sure to leave your ego behind**

d. **Combined Approach**
   - **Use all of the above techniques discussed above**

31. Graphical user Interface (GUI)

- **Liang pages 55 – 57**
- **Liang Powerpoint Slides**