Introduction to Computational Thinking

Lecture #7:

Boolean Expressions; Nested if/else; Switch and Conditional Operator Shortcuts; Strings Processing

Qiao Xiang, Qingyu Song

https://sngroup.org.cn/courses/ct-xmuf25/index.shtml

11/12/2025

This deck of slides are heavily based on cs112 at Yale University and cs101 at UCAS, respectively, by courtesy of Dr. Y. Richard Yang and Dr. Zhiwei Xu.

Outline

- Admin and recap
- Text I/O
  - Input: basic Scanner input
  - Output: basic printf and String.format

Recap: Input using Scanner

- Java uses an object to remember the state (e.g., source) of a scanner

```
Scanner console = new Scanner(System.in);
console.nextInt();
```

- Input can be read as tokens or a line (nextLine)

Input from File

- There are two approaches
  - Create a scanner with src as a file (more shortly)
  - Redirect: a concept in computer science, which allows the operating system to send a file's content as if it is typed in by keyboard:
    (Use command line: Terminal -> to working directory)

```
%java Plot < USA.txt
```

Plot.java USA.txt

Token and Exception

- When a token is not the type that the scanner expects, since no reasonable (non-ambiguous) return value, Scanner throws an exception (panic)

```
System.out.print("Which year will you graduate? ");
int year = console.nextInt();
```

Output:

```
Which year will you graduate? Timmy

java.util.InputMismatchException
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        ...
```

Exceptions

- **Exception**: a programming language mechanism to represent a runtime error, e.g.,
  - dividing an integer by 0
  - trying to read the wrong type of value from a `Scanner`
  - trying to read a file that does not exist
- Java has defined a set of exceptions, each with a name, e.g., ArithmeticException, InputMisMatchException, FileNotFoundException

Design Methodology: How to Handle Potential Exceptions?

- Two basic approaches
  - Test before proceed
  - Proceed and clean up (try/catch)

Robust Input Approach 1: Test Before Proceed

- Design pattern

```
if ( <ExceptionConditionFalse> ) {
  proceed;
}
else {
  System.out.println("Error message.");
}
```

return type of hasNextInt() is boolean, indicating a logical condition.

```
System.out.print("Which year will you graduate? ");
if ( console.hasNextInt() ) {
  year = console.nextInt();
}
else {
  System.out.println("Wrong type; please give int.");
}
```

Robust Input Approach 2: try and catch

- **try/catch:** if an exception happens, program execution jumps to the `catch` statement, skipping the rest in the `try` block.

```
try {

   potentially dangerous statements

} catch (ExceptionName e) {
   handle exception, such as print an error message

} // end of catch
```

Robust Input Approach 2: Example

```
import java.util.Scanner;   // for Scanner

public class ScannerInputExampleTry {
    public static void main(String[] args) {
        Scanner console = new Scanner( System.in );
        try {
          System.out.print("Which year will you graduate? ");
          int year = console.nextInt();
          System.out.println("Your give " + year);

        } catch (InputMisMatchException e) {
            // print an error message
            System.out.println("You give a wrong input type.");
        } // end of catch

    } // end of main
}
```

What Happened: the `throws` Clause

- **`throws` clause**: Keyword on a method's header to state that it may generate an exception (and will not handle it) and those using it must handle it (called a checked exception; nextInt does not declare it).

- Syntax:

```
public static <type> <name>(...) throws <type> {
```

  - Example:

    http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html#Scanner(java.io.File)

Outline

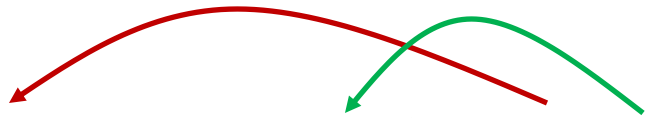
- Admin and recap
- Text I/O
  - Input: basic Scanner input
  - Output: basic printf and String.format

Printf/Format Design

```
System.out.printf("format string", parameters);
```

- Output with *placeholders* to insert parameters, e.g.,
  - `%d` integer
  - `%f` real number
  - `%s` string
    - these placeholders are used instead of + concatenation
  - Example:

```
int x = 3;
int y = -17;
System.out.printf("x is %d and y is %d!\n", x, y);

                          // x is 3 and y is -17!
```



    - `printf` does not drop to the next line unless you write `\n`

https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html

Printf/Format Design and Language Support

```
System.out.printf("format string", parameters);
```

- In the most general case, Java allows flexible (variable) method signature, e.g.,

```
public static type name( <type1> param1,
                         <type2>… param2)
```

- Number and type of parameters determined by the first parameter. We will not learn how to define such methods, but will use some: `printf()` and `format()`

printf Width

- %**W**d integer, **W** characters wide, right-aligned
- %-**W**d integer, **W** characters wide, *left*-aligned
- %**W**f real number, **W** characters wide, right-aligned
- ...

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 10; j++) {
        System.out.printf("%4d", (i * j));
    }
    System.out.println();   // to end the line
}
```

Output:

```
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
```

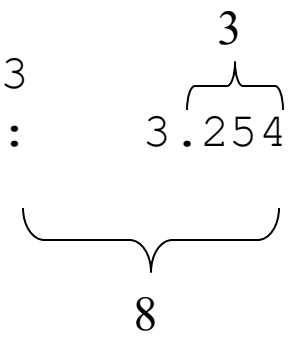
printf Precision

- %.**D**f real number, rounded to **D** digits after decimal
- %**W**.**D**f real number, **W** chars wide, **D** digits after decimal
- %-**W**.**D**f real number, **W** wide (left-align), **D** after decimal

```
double gpa = 3.253764;
System.out.printf("your GPA is %.1f\n", gpa);
System.out.printf("more precisely: %8.3f\n", gpa);
```

Output:

```
your GPA is 3.3
more precisely:    3.254
```



`printf` Formatting

- Many more formatting control options supported by `printf`, e.g., using the comma (,) to display numbers with thousands separator

```
System.out.printf("%,d\n", 58625);
System.out.printf("%,.2f\n", 12345678.9);
```

Output:

```
58,625
12,345,678.90
```

System.out.printf and String.format

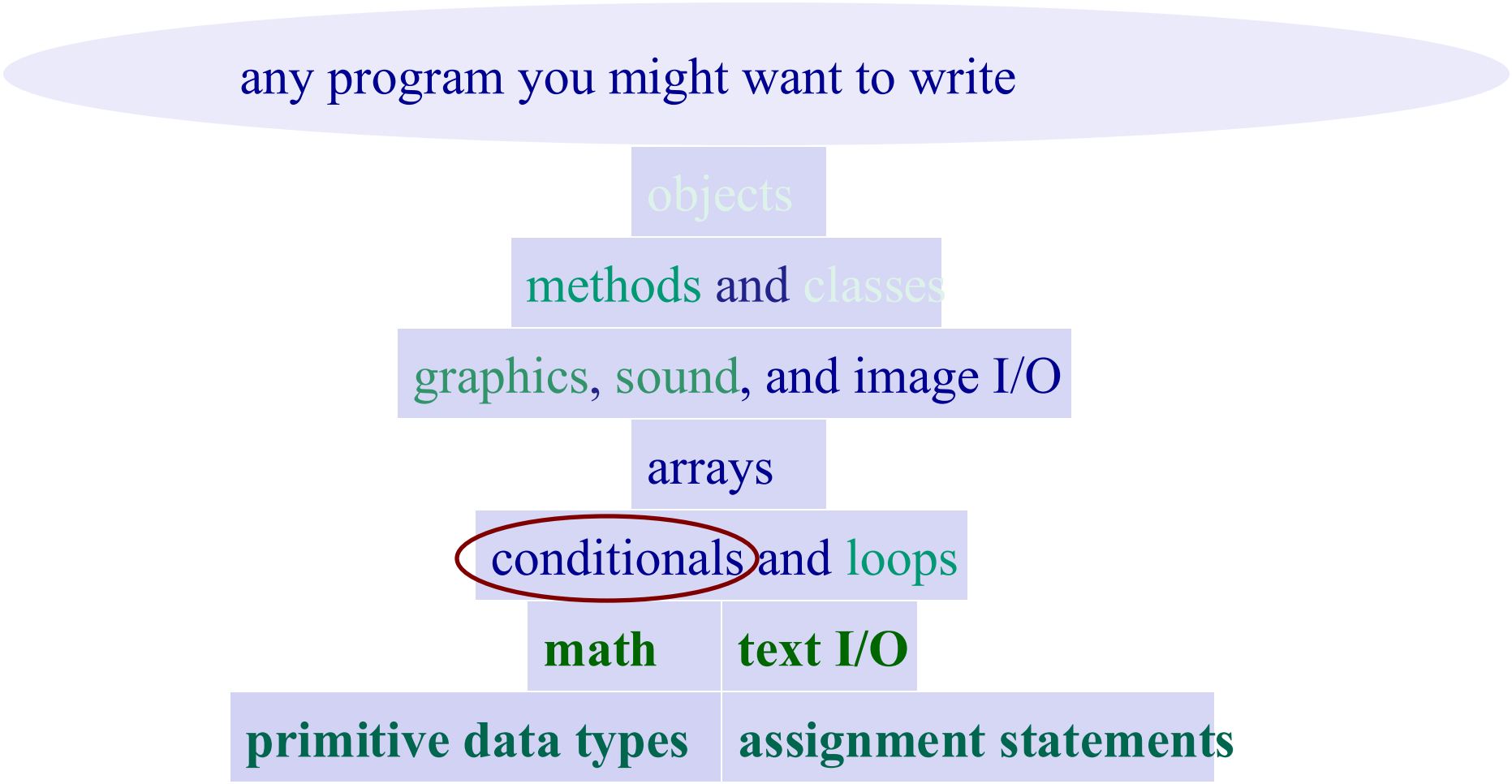
- `String.format` has the same formatting capability as `printf`, except that `printf` outputs and `String.format` returns:

```
System.out.printf("%,.2f\n", 12345678.9);

String s = String.format("%,.2f\n", 12345678.9);
System.out.print( s );
```

Output:

```
12,345,678.90
12,345,678.90
```

Foundational Programming Concepts



Program Flow of Control

- Java provides conditional and loop statements as program flow of control statements:
  - decision statements, or conditional statements: decide whether or not to execute a particular statement
  - repetition statements, or loop statements: perform a statement over and over repetitively
- The foundation of program flow of control is the logical condition, which should be a boolean expression, e.g.,

```
if ( <boolean expression> ) {
        do something
} [else {
        so something else
}]
```

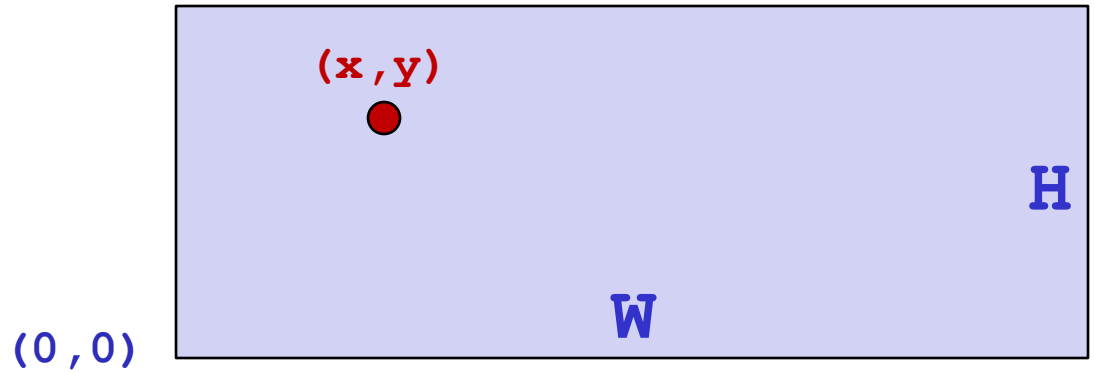
Basic Boolean Expression: Relational Tests

- A basic Boolean expression is a relational test of two values using a *relational operator* :

Operator	Meaning	Example	Value
==	equals	`1 + 1 == 2`	`true`
!=	does not equal	`3.2 != 2.5`	`true`
<	less than	`10 < 5`	`false`
>	greater than	`10 > 5`	`true`
<=	less than or equal to	`126 <= 100`	`false`
>=	greater than or equal to	`5.0 >= 5.0`	`true`

- Note the difference between the equality operator (==) and the assignment operator (=)

More Complex Relational Test: Testing Containment



Test if point (x, y) is in the rectangle?

```
(0 <= x <= W) and (0 <= y <= H)
```

SYNTAX ERROR

```
(0 <= x) && ( x <= W) && (0 <= y) && (y <= H)
```

Logical Operators

- Tests can be combined using *logical operators*:

Operator	Description	Example	Result
`&&`	and	`(2 == 3) && (-1 < 5)`	`false`
`||`	or	`(2 == 3) || (-1 < 5)`	`true`
`!`	not	`!(2 == 3)`	`true`

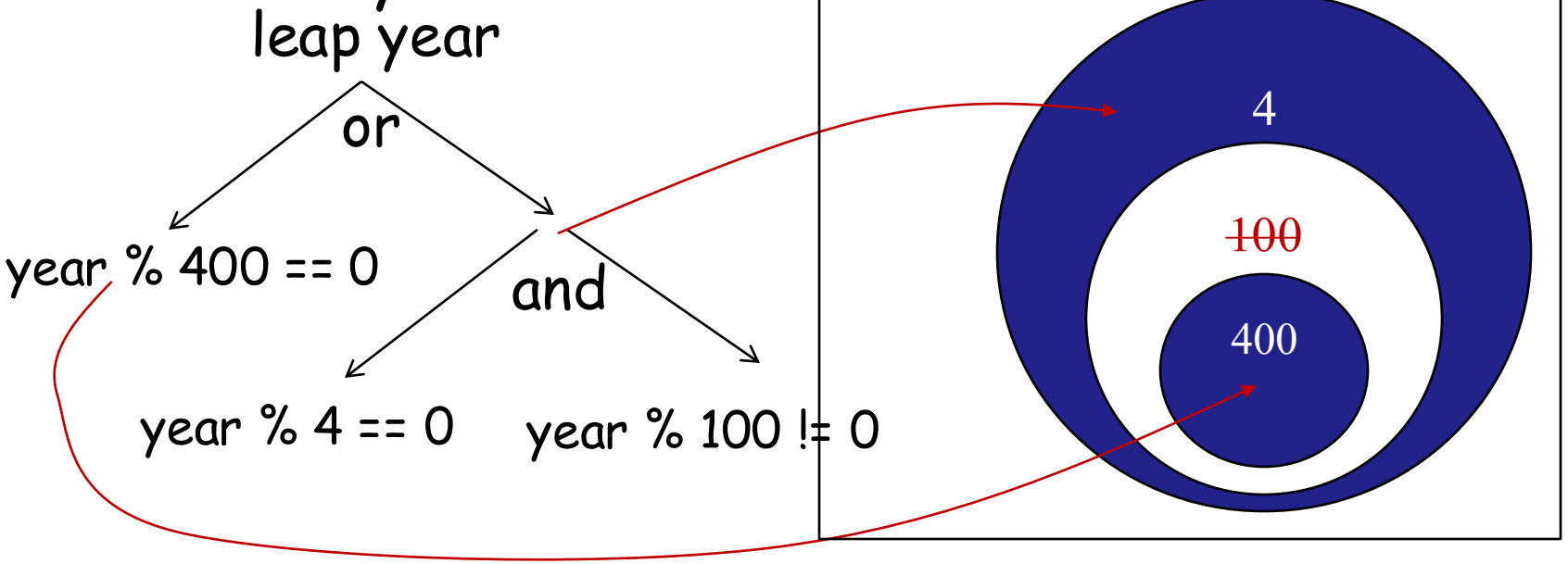
- "Truth tables" for each, used with logical values *p* and *q*:

p	q	p && q	p || q
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

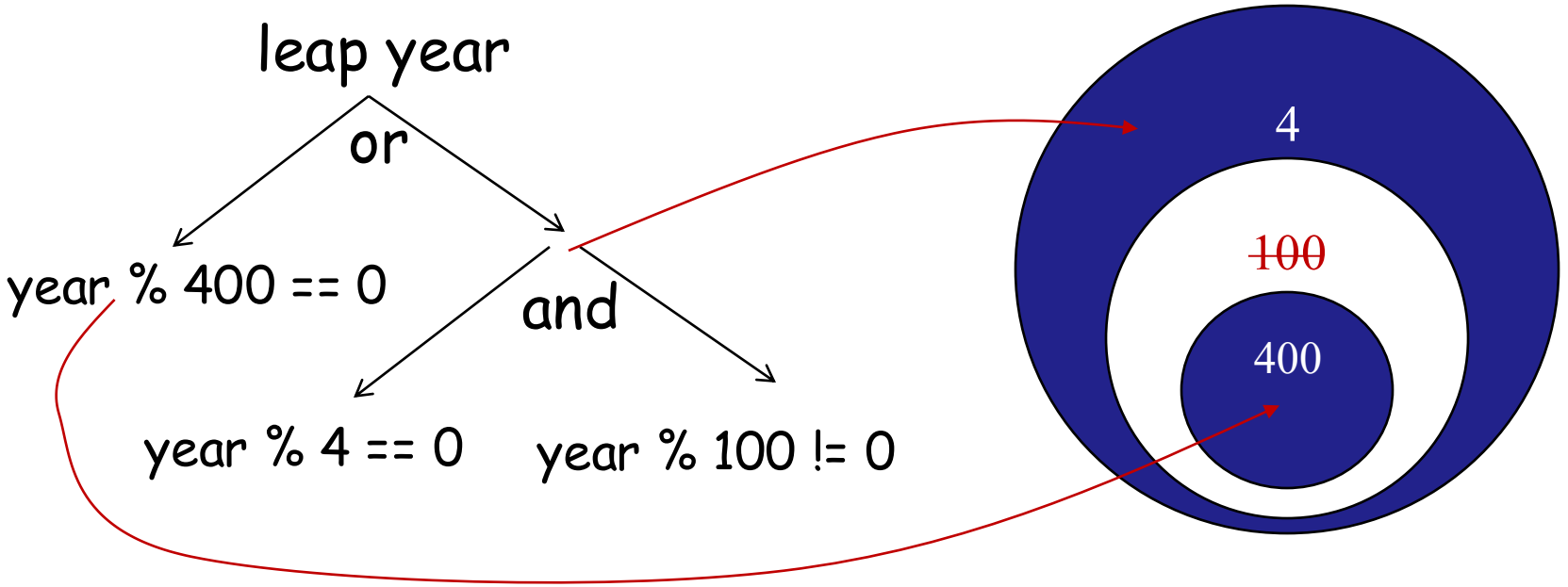
p	!p
true	false
false	true

Applying Logical Operators

- Implement leap year condition: "… most years that are evenly divisible by 4 are leap years; However, there are some exceptions to this rule: Years that are evenly divisible by 100 are *not* leap years, unless they are also evenly divisible by 400"



Testing Leap Year




```
y % 400 == 0 || (y % 100 != 0 && y % 4 == 0)
```

Outline

- Admin and recap
- Conditionals
  - Logical conditions (Boolean expressions)
    - Basic relational tests
    - Tests using logical operators
    - Boolean type

`boolean` Type

- **`boolean`**: A primitive type whose values are `true` or `false`.
  - Like other types, it is legal to:
    - create a `boolean` variable and assign it values
    - pass a `boolean` value as a parameter
    - return `boolean` value as a method
    - call a method that returns a `boolean` and use it as a test
  - You can use Boolean variables in Boolean expressions

boolean Expressions: What

- Similar to arithmetic expressions, except that
  - the operands are Boolean values, Boolean variables, or relational tests.
  - the operators are logical operators || && !
  - Example

```
boolean lovesCS  = true;
boolean student  = age < 21;

// allow only CS-loving students over 21
if (student && lovesCS)
    System.out.println("Pass");

// an alternative
if (age < 21 && lovesCS)
    System.out.println("Pass");
```

`boolean` Expressions: Why

- Why is the type `boolean` useful?
  - Can capture a complex logical test result and use it later
  - Can write a method that does a complex test and returns it

```
boolean goodAge    = age >= 18 && age < 29;
boolean goodHeight = height >= 78 && height < 84;
boolean rich       = salary >= 100000.0 || hasStock;
boolean perfect    = school == YALE;

if ((goodAge && goodHeight) || rich || perfect) {
    System.out.println("Okay, let's go out!");
} else {
    System.out.println("It's not you, it's me...");
}
```

English vs Programming

- OR vs Exclusive OR
  - I'll either watch TV or go to gym: Exclusive OR
  - watchTV || gotoGym can be both true

- x is either 1 or 2 or 3
  - x == 1 || 2 || 3
  - x ==1 || x == 2 || x == 3

Logical AND/OR Evaluation

- Java uses shortcircuit when evaluating && and ||
  - a && b shortcircuit:
    - if a is false, b is not evaluated

```
if ((count != 0) && (total / count > AVG_THRESHOLD)){
   // …
}
```

  - a || b shortcircuit:
    - if a is true, b is not evaluated

Mixed Arithmetic, Relation, Logical, and Assignment Expression

- Example:

```
boolean mystery = 5 * 7 >= 3 + 5 * (7 - 1) && 7 <= 11;
```

- Precedence ordering of boolean expression
  - Arithmetic operators
  - Relations operators (==, !=, <, >, <=, >=)
    - Note that equality and relational operators cannot be chained (e.g., 1 < x < 3 is invalid)
  - NOT (!)
  - AND (&&)
  - OR (||)
  - Assignment operators (=)

Example

```
boolean mystery = 5 * 7 >= 3 + 5 * (7 - 1) && 7 <= 11;
                  5 * 7 >= 3 + 5 * 6 && 7 <= 11
                  35    >= 3 + 30 && 7 <= 11
                  35    >= 33 && 7 <= 11
                         true && true
                              true
```

Define and Use `boolean` Methods

- Define a `boolean` method

```
public static boolean isOdd(int n) {
    if (n % 2 == 1) {
        return true;
    } else {
        return false;
    } // see a shorter version shortly
}
```

- Calls to a `boolean` test method:

```
if ( isOdd(57) ) {
    ...
}
```

"Boolean Zen", part 1 (Use)

- Programmers new to `boolean` often test if a result is `true`:

```
if ( isOdd(57) == true ) { // reads redundant
    ...
}
```

Simplify to:

```
if ( isOdd(57) ) {              // concise
    ...
}
```

- A similar pattern can be used for a `false` test:

```
if ( isOdd(57) == false ) {    // does not read well
if ( !isOdd(57) ) {            // concise
```

"Boolean Zen", part 2 (Define)

- Programmers new to `boolean` often have an `if/else` to return `true` or `false`:

```
public static boolean isOdd(int n) {
    if (n % 2 == 1) {
        return true;
    } else {
        return false;
    }
}
```

```
public static boolean isOdd(int n) {
    return (n % 2 == 1);
}
```

"Boolean Zen" template

- Replace

```
public static boolean <name>(<parameters>) {
    if (<test>) {
        return true;
    } else {
        return false;
    }
}
```

- with

```
public static boolean <name>(<parameters>) {
    return <test>;
}
```

Outline

- Admin and recap
- Conditionals
  - Logical conditions (Boolean expressions)
    - Basic relational tests
    - Tests using logical operators
    - Boolean type
  - Nested if/else conditional statements

Motivation: Chaos Game

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

  for (int i = 0; i < 10000; i++) {
    int rand = (int) (Math.random() * 3);
    if (rand == 0) {
    //
    }

    if (rand == 1) {
      // …
    }

    if (rand == 2) {
      // …
    }
  }
}
```

Q: How many comparisons in each round (iteration)?

Revision: Nested Comparison (else if)

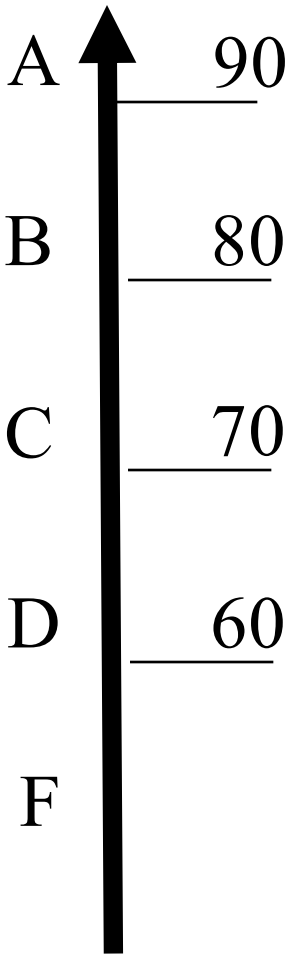
```
public static void main (String[] args) {
   …
  for (int i = 0; i < 1000; i++) {
    int rand = (int) (Math.random() * 3);
    if (rand == 0) {
    //
    }

    else if (rand == 1) {
      // …
    }

    else if (rand == 2) {
      // …
    }
  }
}
```

Add else to skip tests if already matched. This is called nested comparison (else if)

Q: Average # of comparisons per iteration?

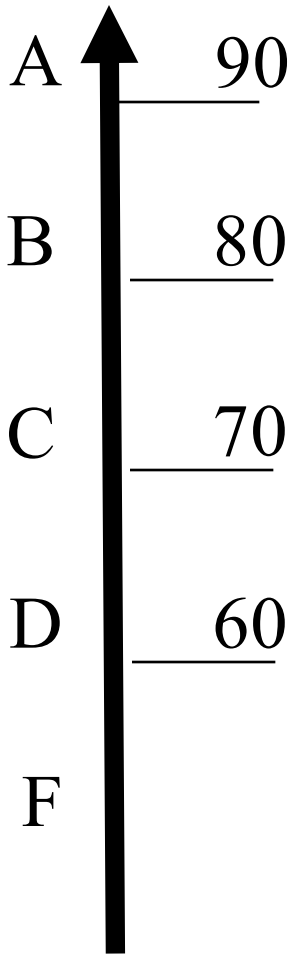
Benefit of nested comparison: reduce # comparisons

Example: Grading Curve




```
if (percent >= 90) {
    grade = A;
}

if (percent >= 80) {
    grade = B;
}

if (percent >= 70) {
    grade = C;
}

if (percent >= 60) {
    grade = D;
}

if (percent < 60) {
    grade = F;
}
```

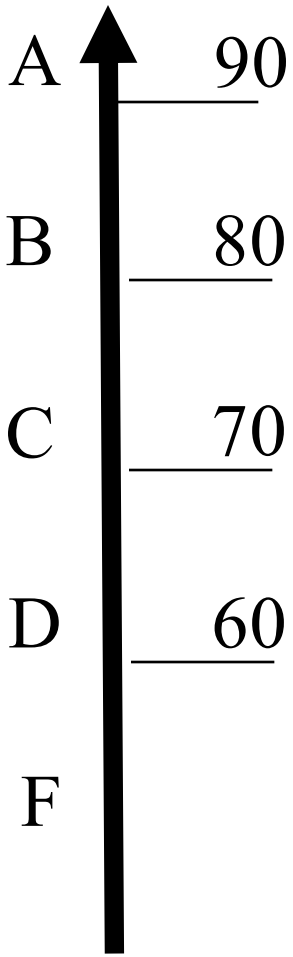
Example: Grading Curve (Design 2)




```
        if (percent >= 90) {
            grade = A;
        }
else    if (percent >= 80) {
            grade = B;
        }
else    if (percent >= 70) {
            grade = C;
        }
else    if (percent >= 60) {
            grade = D;
        }
else    if (percent < 60) {
            grade = F;
        }
```

Nested comparison to achieve mutual exclusion; high-to-low.

Example: Grading Curve (Design 3: Low to High)




```
        if (percent < 60) {
            grade = F;
        }
else    if (percent < 70) {
            grade = D;
        }
else    if (percent < 80) {
            grade = C;
        }
else    if (percent < 90) {
            grade = B;
        }
else    {
            grade = A;
        }
```

Exercise: Chaos Game2

- Draw points with these rules

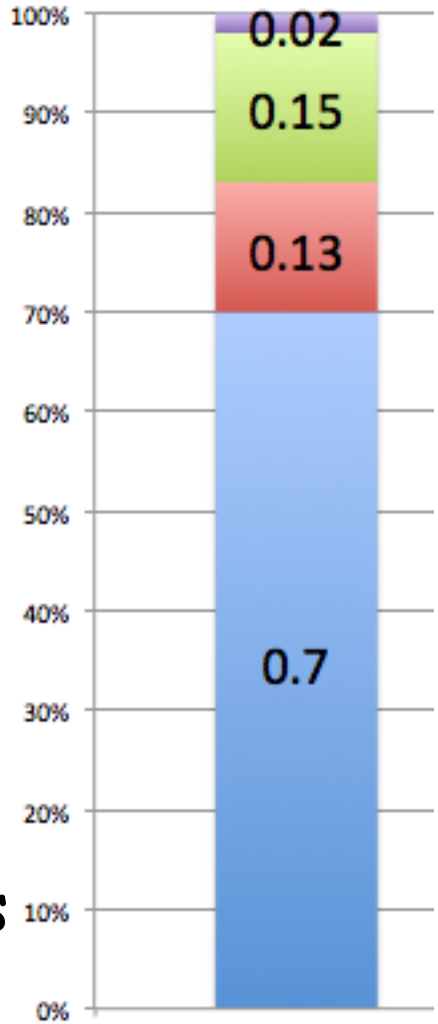
probability	new x	new y
2%	.50	.27y
15%	-.14x + .26y + .57	.25x + .22y - .04
13%	.17x - .21y + .41	.22x + .18y + .09
70%	.78x + .03y + .11	-.03x + .74y + .27

- Basic idea
  - Generate random numbers in [0, 1.0]
  - Divide 0-1.0 into 4 areas for the 4 cases
  - Check which area each number falls in

Exercise: Chaos Game2

- Draw points with these rules

probability	new x	new y
2%	.50	.27y
15%	-.14x + .26y + .57	.25x + .22y - .04
13%	.17x - .21y + .41	.22x + .18y + .09
70%	.78x + .03y + .11	-.03x + .74y + .27



- Basic idea
  - Generate random numbers in [0, 1.0]
  - Divide 0-1.0 into 4 areas for the 4 cases
  - Check which area each number falls in

Exercise: Barnsley Game

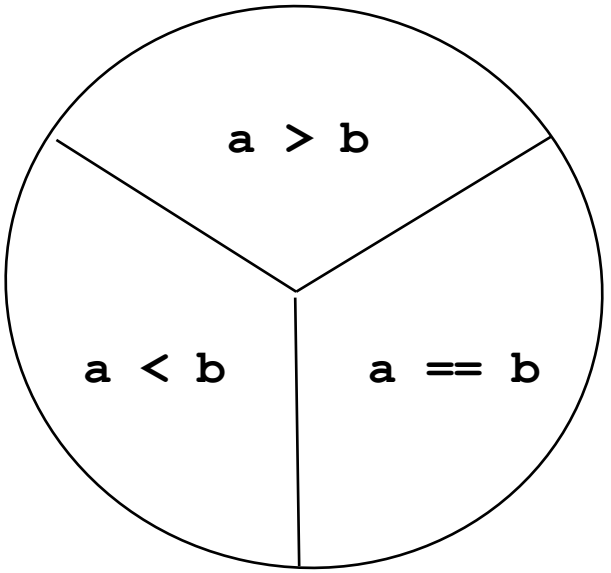
```
public static void main(String[] args) {

   StdDraw.setPenRadius(0.002);
   StdDraw.setPenColor(Color.GREEN);

   final int T = 40000;
   double x = 0.0, y = 0.0;

   for (int t = 0; t < T; t++) {
     double rand = Math.random();        // 0 - 1.0

     if (rand <= 0.7) {                  // 0 - 0.7: 70%
       x = .78 * x + .03 * y + .11;
       y = -.03 * x + .74 * y + .27;
     } else if (rand <= 0.7 + 0.15) {    // 0.7 - 0.7+0.15: 15%
       x = -.14 * x + .26 * y + .57;
       y = .25 * x + .22 * y - .04;
     } else if (rand < 0.7 + 0.15 + 0.13) { // 0.7+0.15 – 0.7+0.15+0.13: %13
       x = .17 * x - .21 * y + .41;
       y = .22 * x + .18 * y + .09;
     } else {                            // 2%
       x = 0.5;
       y = .27 * y;
     }
     StdDraw.point(x, y);

   } // end of for
} // end of main
```

Barnsley Game

- Questions to think about
  - What does computation tell us about nature?
  - What may nature tell us about computation?

Summary: nested if/else

- Nested if/else is a common structure
  - Mutually exclusive test conditions
    => Reduces # of comparisons
  - Non-mutual exclusive test conditions
    => Achieves mutual exclusion



Outline

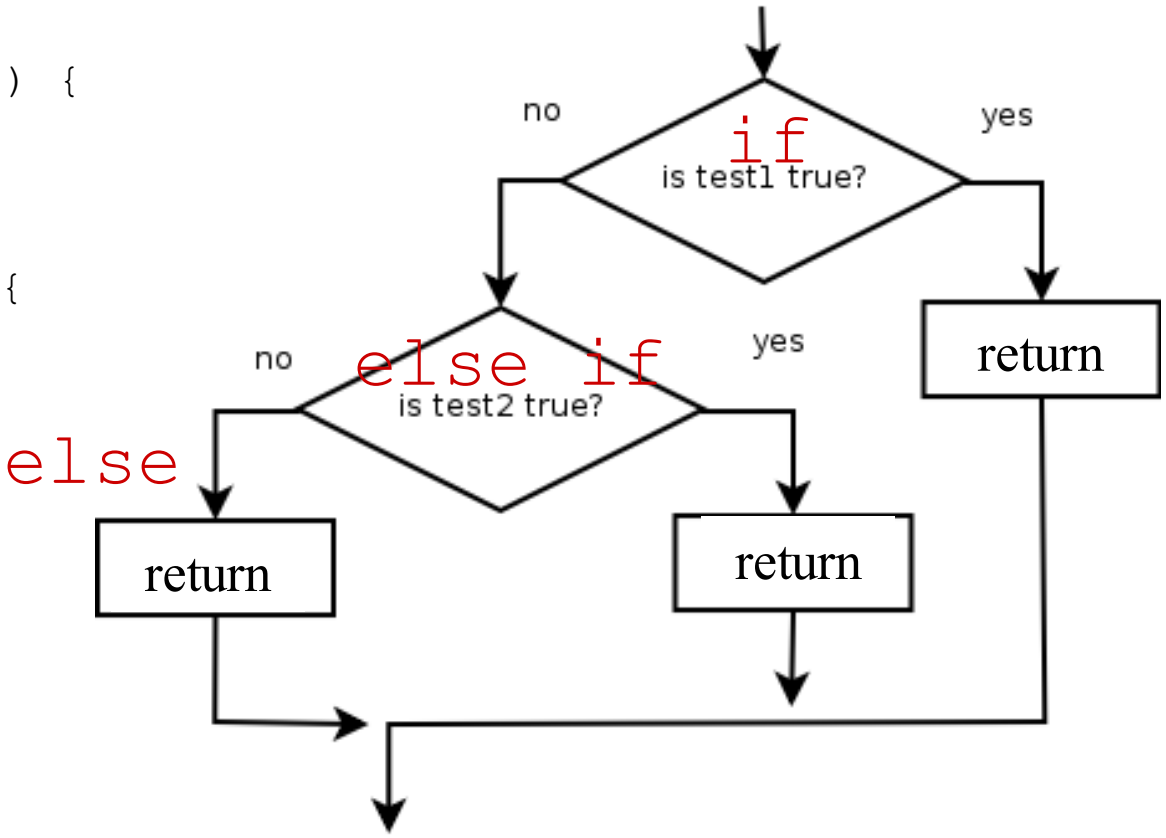
- Admin and recap
- Conditionals
  - Logical conditions (Boolean expressions)
    - Basic relational tests
    - Tests using logical operators
    - Boolean type
  - Nested if/else conditional statements
    - Benefits of nested if/else
    - Complexities of nested if/else
      - mismatched else
      - all path must return

`if/else` with `return`

- A method with a return requires that all paths through the code must reach a `return` statement.
- The compiler analyzes this by considering the syntax only!

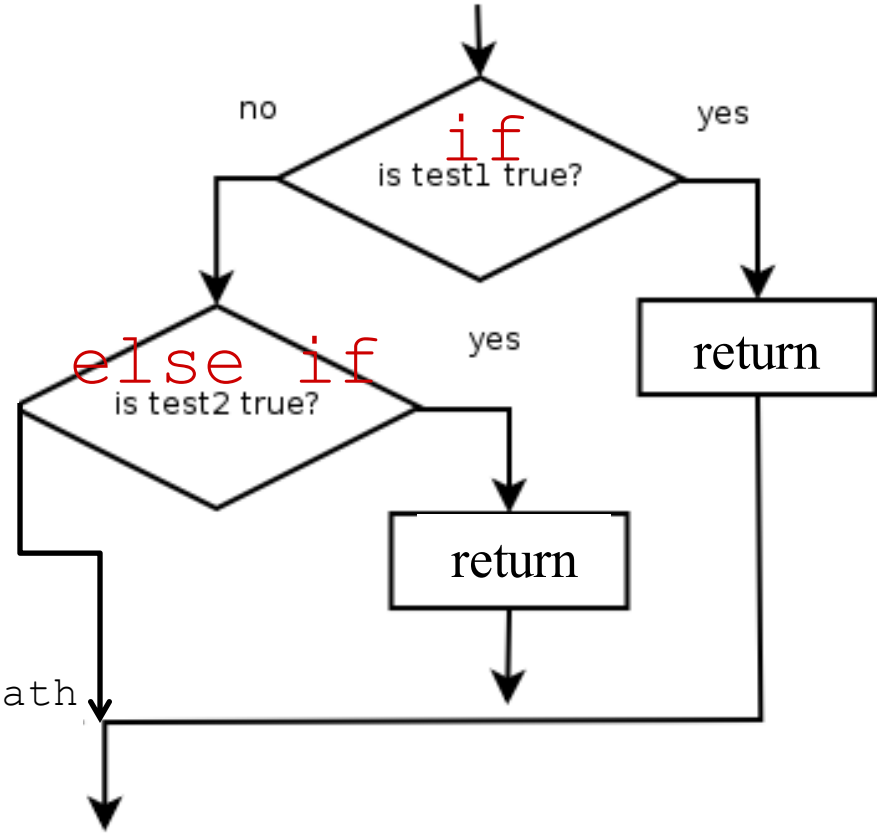
if/else with return

```
static int myMethod() {
  if (test1) {
    statement(s);
    return x;
  } else if (test2) {
    statement(s);
    return y;
  } else {
    statement(s);
    return z;
  }
} // end of method
```



if/else with return

```
static int myMethod() {
  if (test1) {
    statement(s);
    return x;
  } else if (test2) {
    statement(s);
    return y;
  } else {
    statement(s);
    return z;
  }
} // end of method
// ERROR: missing return in a path
```



if/else with return

```
public static int max(int a, int b) {
    if (a > b) {
        return a;
    }

}   // Error: not all paths return a value
```

```
public static int max(int a, int b) {
    if (a > b) {
        return a;
    } else if (a <= b) {
        return b;
    }
   // Error: syntax analysis missing a path
}
```

```
public static int max(int a, int b) {
    if (a > b) {
        return a;
    } else if (a <= b) {
        return b;
    }
   // OK
}
```

```
public static int max(int a,
                      int b)
{
    int myMax = a; // default
    if (b > a) {
        myMax = b;
    }
    return myMax;  // OK
}
```

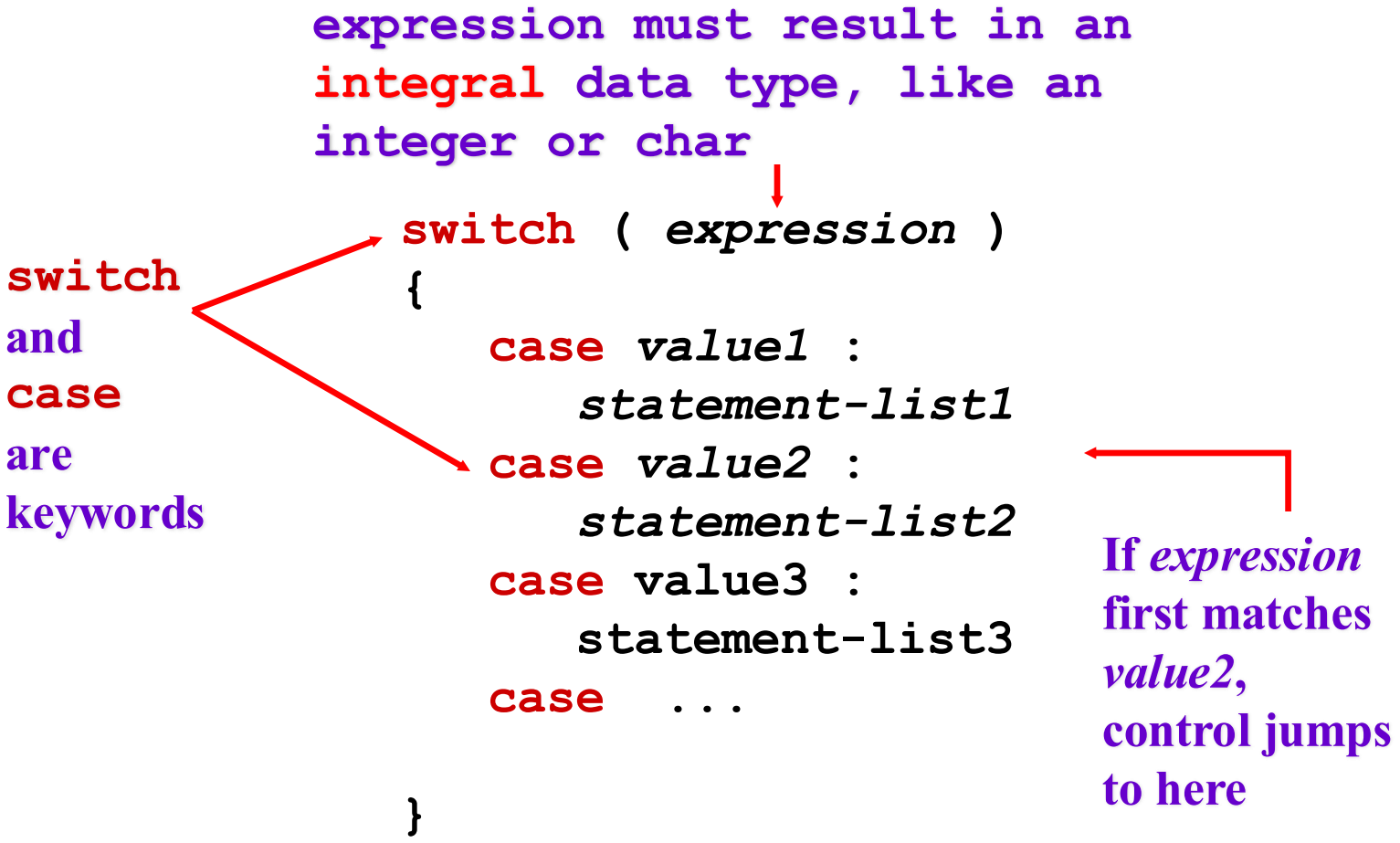
```
public static int max(int a,
                      int b)
{
    if (b > a) {
        return b;
    }
    return a;      // OK
}
```

Outline

- Admin and recap
- Conditionals
  - Logical conditions (Boolean expressions)
    - Basic relational tests
    - Tests using logical operators
    - Boolean type
  - Nested if/else conditional statements
    - Benefits of nested if/else
    - Complexities of nested if/else
      - mismatched else
      - all path must return
  - Conditional shortcuts

Alternative Testing using switch

- Java `switch` statement is a shortcut allowing clear listing of multiple choices in some settings



The `switch` Statement: default

- A switch statement can have an optional *default case* as the last case in the statement
- The default case has no associated value and simply uses the reserved word `default`
- If the default case is present, control will transfer to it if no other case value matches
- If there is no default case, and no other value matches, control falls through to the statement after the switch

The `switch` Statement Design Decision

- Flow of control jumps to the first matching case and execution, ignoring the case statements.

- Often a `break` *statement* is used at the end of each case's statement list
  - A `break` statement causes control to transfer to the end of the `switch` statement

- If a `break` statement is not used, the flow of control will continue into the next case

Exercises

- (Offline) Implement DaysInMonth.java using switch (DaysInMonthSwitch.java)
- (Offline) Implement a method to convert a number between 1 and 10 to its English word

Limitation of the `switch` Statement

- The result of the *expression* must be an integral type, e.g,
  - case x < 1: // ERROR
- The value for each case must be a constant expression

```
switch ( expression ) {
    case value1 :
        statement-list1
    case value2 :
        statement-list2
    case value3 :
        statement-list3
    default: ...

}
```

Shortcut: The Conditional Operator

- Java has a *conditional operator* that evaluates a boolean expression to determine which one of the two expressions should be evaluated:

```
condition ? expression1 : expression2
```

- If the *condition* is true, *expression1* is evaluated; if it is false, *expression2* is evaluated; the value of the evaluated expression becomes available
- For example:

```
larger = (num1 > num2) ? num1 : num2;
```

  - if `num1` is greater that `num2`, then `num1` is assigned to `larger`; otherwise, `num2` is assigned to `larger`

Examples

- Example 1:

```
Scanner scan = new Scanner( System.in );
double val = scan.nextDouble();
val = (val < 0) ? -val : val;
```

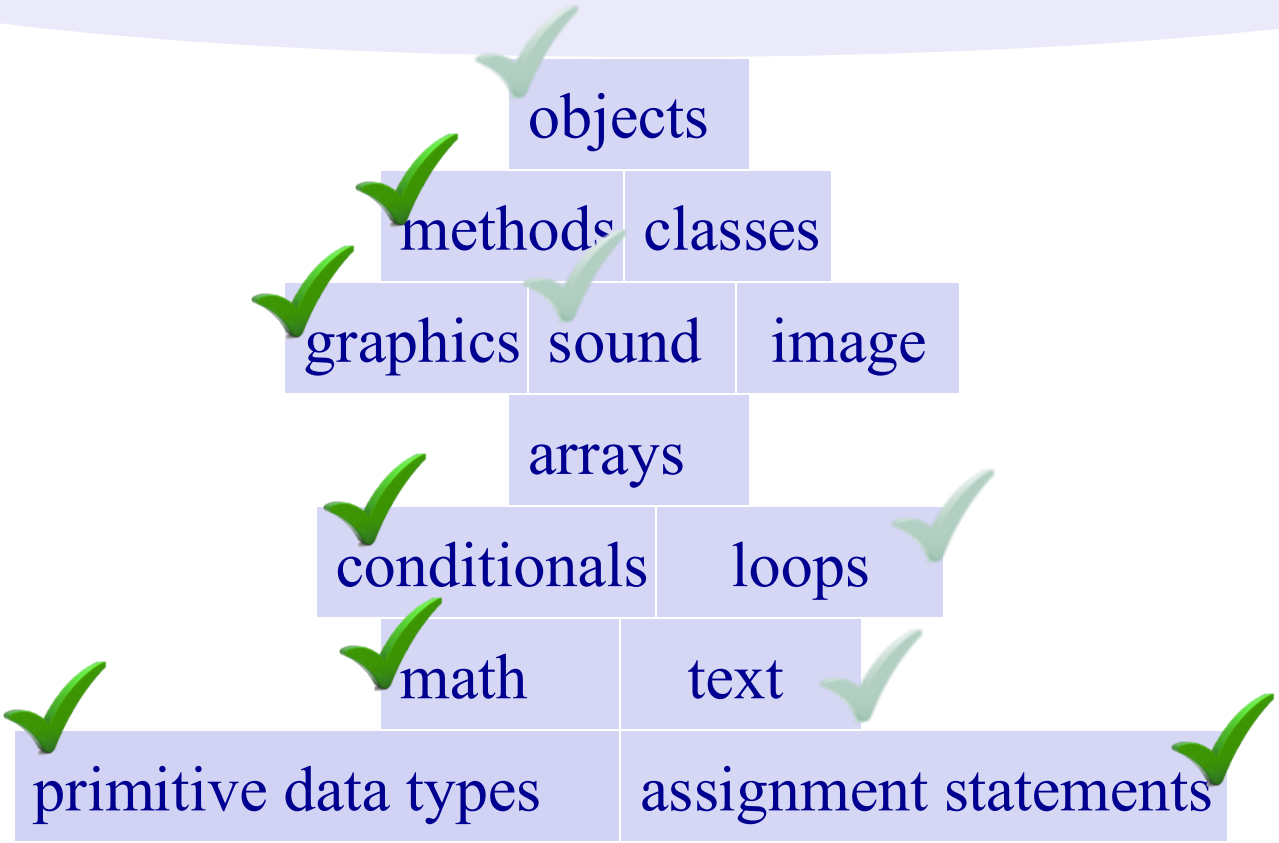
- Offline exercise: Use operator on Feb (daysInMonth)

- Exercise (use right plural):

```
Scanner scan = new Scanner( System.in );
int dollar = scan.nextInt();
System.out.println ("Your amount is "
                    + dollar +
dollar <= 1 ? " Dollar." : " Dollars.");
```

Foundational Programming Concepts

any program you might want to write



Outline

- Admin and recap
- Conditionals
- Strings

Discussion

- Functions (methods) that one may need to process text (Strings)?

Roadmap: String Processing

- Access methods, e.g.,
  - `length, substring, charAt, indexOf`
- String generator methods (string->string), e.g.
  - `toLowerCase, toUpperCase, replace, split` (string->strings)
- Conversion from String, e.g.,
  - `Integer.parseInt, Double.parseDouble`
- Input processing, e.g.,
  - `Scanner.next, Scanner.nextLine`
- Output formatting: format string
  - `System.out.printf, String.format`
- Boolean methods, e.g.,
  - `equals, equalsIgnoreCase, startsWith, endsWith, contains, matches`

Strings

- **string**: An object storing a sequence of text characters.
  - Unlike most other objects, a `String` is so common that Java introduces a short cut and you do not need to create String with `new`—to make Strings more similar to native primitive data types

```
String name = "text";
```

```
String name = expression;
```

```
int x = 3;
int y = 5;
String point = "(" + x + ", " + y + ")";
```

String Internal: Indexes

- A string is a sequence of characters numbered with 0-based *indexes*:

```
String name = "R. Kelly";
```

index	0	1	2	3	4	5	6	7
character	R	.		K	e	l	l	y

  - First character's index : 0
  - Last character's index : 1 less than the string's length

  - The individual characters are values of type `char`

`String` Access Methods

Method name	Description
`length()`	number of characters in this string
`charAt(index)`	character at index
`indexOf(`**`str`**`)`	index where the start of the given string appears in this string (-1 if not found)

- These methods are called using the dot notation:

```
String name = "Dr. Dre";
System.out.println( name.length() );    // 7
```

String Method Examples

```
// index      01234567890123456
String s1 = “Xiamen University";
String s2 = “CT 2025, Fall";
System.out.println( s1.length() );          // 17
System.out.println( s1.indexOf("e") );      // 4
```

`String` Generation Methods

Method name	Description

`substring(`**index1**`, `**index2**`)`
or
`substring(`**index1**`)`
	A new string with characters in this string from *index1* (inclusive) to *index2* (exclusive);
	if *index2* is omitted, grabs till end of string

`toLowerCase()`	A new string with all lowercase letters
`toUpperCase()`	A new string with all uppercase letters
`replace(`**str1**`, `**str2**`)`	A new string replacing occurrences of *str1* with *str2*

- Java strings are immutable. Hence, each of the above will return a new string. With the current string not modified.

String Method Examples

```
// index      01234567890123456
String s1 = "Xiamen University";
String s2 = "CT 2025, Fall";

System.out.println( s1.length() );          // 17
System.out.println( s1.indexOf("e") );      // 4
System.out.println( s1.substring(7, 17) );// "University"

String s3 = s2.substring(9, 13);
System.out.println( s3.toLowerCase() );    // "fall"
```

Parsing a String

- Instead of substring, indexOf, you can also use scanner to parse a string (in units of tokens only).

```
String s = "Year 2025 is great!";
Scanner scan = new Scanner( s );
String word = scan.next(); // skip Year
int year = scan.nextInt();
```

Exercise

- Given the following string:

```
// index          01234567890123456789
String book = "Building Java Programs";
```

  - How would you extract the word `"Building"` from `book`?
  - More generally, how to extract the first word of a string?

One solution: substring(0, indexOf(" "))

Split a String

- Some common approaches
  - Use indexOf to find the location and then substring to extract a sub string
  - Use scanner to extract tokens

```
String s = "Years 2017 2018 will be great!";
Scanner scan = new Scanner( s );
String word = scan.next(); // skip Year
int year1 = scan.nextInt(); // 2017

word = scan.next();
int year2 = Integer.parseInt( word ); // 2018
```

  - Use split to split a string into multiple strings (more later)

Exercise

- Write a program to convert your “boring” name to a more exciting "gangsta name."
  - last initial
  - *Diddy (stage name of Sean Combs[吹牛老爹, 美国说唱歌手])*
  - first name (all caps)
  - *-izzle (Hip-Pop Ending押韵的韵脚)*

Example Output:

```
Type your name, playa: Marge Simpson
Your gangsta name is "S. Diddy MARGE-izzle"
```

Strings Answer

```
// This program prints your "gangsta" name.
import java.util.*;

public class GangstaName {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        System.out.print("Type your name, playa: ");
        String name = console.nextLine();

        // split name into first/last name and initials
        String first = name.substring(0, name.indexOf(" "));
        first = first.toUpperCase();
        String last = name.substring(name.indexOf(" ") + 1);
        String lInitial = last.substring(0, 1);

        System.out.println("Your gangsta name is \""
                           + lInitial
                           + ". Diddy "
                           + first + "-izzle\"");
    }
}
```

`String` Boolean (Pattern-Matching) Methods

Method	Description
`equals(`**str**`)`	whether two strings contain the same characters
`equalsIgnoreCase(`**str**`)`	whether two strings contain the same characters, ignoring upper vs. lower case
`startsWith(`**str**`)`	whether one contains other's characters at start
`endsWith(`**str**`)`	whether one contains other's characters at end
`contains(`**str**`)`	whether the given string is found within this one
`matches(`**regExp**`)`	whether the string matches a regular expression

```
Scanner console = new Scanner(System.in);
System.out.print("Type your name: ");
String name = console.nextLine();

if (name.startsWith(”Prof. Dr.")) {
   System.out.println(”Are you from Germany?");
} else if (name.endsWith("SquarePants")) {
   System.out.println("And I am Patrick Star!");
}
```

Exercise: Word Rhyme

- Prompt the user for two words and report whether they:
  - *"rhyme"* (end with the same last two letters尾部押韵)
  - *alliterate* (begin with the same letter头部押韵)
  - Example output: (run #1)

```
Type two words: car STAR
They rhyme!
```

(run #2)

```
Type two words: bare bear
They alliterate!
```

(run #3)

```
Type two words: sell shell
They rhyme and alliterate!
```

(run #4)

```
Type two words: extra strawberry
```

Boolean Answer

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    System.out.print("Type two words: ");
    String word1 = console.next().toLowerCase();
    String word2 = console.next().toLowerCase();

    boolean isR = rhyme(word1, word2));
    boolean isA = alliterate(word1, word2));

    // output
}

// Returns true if s1 and s2 end with the same two letters.
public static boolean rhyme(String s1, String s2) {
    return s2.length() >= 2
           && s1.endsWith(s2.substring(s2.length() - 2));
}

// Returns true if s1 and s2 start with the same letter.
public static boolean alliterate(String s1, String s2) {
    return s2.length() >= 1
           && s1.startsWith(s2.substring(0, 1));
}
```

Outline

- Strings
  - String access, modify, parse, format
  - String boolean methods
  - String vs char

String charAt method

- The chars in a String can be accessed using the charAt method.

```
String food = "cookie";
char firstLetter = food.charAt(0);    // 'c'

System.out.println(firstLetter + " is for " + food);
System.out.println("That's good enough for me!");
```

- You can use a for loop to print or examine each character.

```
String major = "CS";
for (int i = 0; i < major.length(); i++) {
    char c = major.charAt(i);
    System.out.println(c);
}
```

Output:

```
C
S
```

StringExamples.java stringChars

https://en.wikipedia.org/wiki/List_of_Unicode_characters

char vs. int

- Char internal stores the index of a char
- Using `char` in arithmetic expressions will cause char to be converted to an int
- To convert an `int` into the equivalent `char`, type-cast it.

`(char) ('a' + 2)` is `'c'`

```
for (char c = 'a'; c <= 'z'; c = (char)(c+1) ) {
    System.out.print(c);
}
```

```
for (char c = 'a'; c <= 'z'; c ++ ) {
    System.out.print(c);
}
```

StringExamples.java charInt method

char vs. String

- "h" is a String
  'h' is a char (the two behave differently)

- String is an object; it contains methods

```
String s = "h";
s = s.toUpperCase();          // 'H'
int len = s.length();         //  1
char first = s.charAt(0);     // 'H'
```

- char is primitive; you can't call methods on it

```
char c = 'h';
c = c.toUpperCase();    // ERROR
```

char vs. String (char Comparison)

- Consistent with primitive numerical types, you can compare `char` values using relational operators (ordering is alphabetical order):

```
'a' < 'b'   and   'X' == 'X'   and   'Q' != 'q'
```

```
String word = console.next();
char ch = word.charAt(word.length() - 1) ;
if ( ch == 's') {
    System.out.println(word + " is plural.");
}

ch = word.charAt(0) ;
if ( 'a' <= ch  && ch <= 'z') {
    System.out.println(word
            + "starts w/ a letter");
}
```

StringExamples.java charCompare method

`char` vs. `String` (Comparison)

- <, <=, >, >= are not defined on objects and hence you cannot write str1 <= str2
- != and == are defined on all objects, but they compare *references* (seen later), not values. So == often gives `false` even when two `String`s have the same letters.

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

String name = console.next(); // user input Barney

if (name == "Barney") {
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

char vs. String (Comparison)

- String comparisons based on content use the method `equals`.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name.equals("Barney")) {
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

- String defines `compareTo(anotherString)` to compare the lexicographic order of two strings.

StringExamples.java stringCompare method

char vs. String

	String	char
Store	An object w/ a sequence of chars	A primitive value storing a unicode index
Methods	.length, .toUpperCase, …	No methods
==, !=	Compare reference, not value	Compare value
<, <=, >, >=	No defined. Use compareTo(anotherString) method	Lexicographic order
.equals method	Compare content	Not defined

https://symbl.cc/en/unicode-table/

Caesar Cipher

If he had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the fourth letter of the alphabet, namely D, for A, and so with the others.

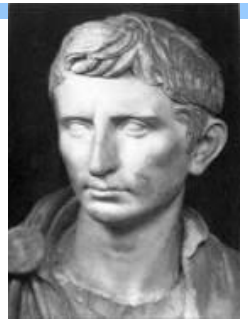
—Suetonius, Life of Julius Caesar 56

Caesar Cipher

Plaintext	Ciphertext
attack said zeus	dwwdfn vdlg chxv
προσβάλλω	στςφείξξό

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c

Additional Caesar-Style Cipher

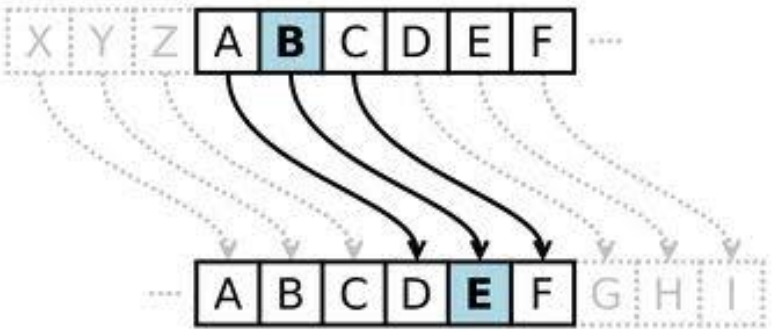


- Augustus
  - Right shift 1 without rotate, with AA for Z
- Mezuzah
  - Shift of one to encrypt the names of God

- Vigenere (a variant) was used by the Confederacy during Civil War

- Mafia boss Provenzano (2006): A (4), …

Why Programming?



Dear Cleopatra,

I love you as big as the Aegean sea, as high as Mount Olympus, as many as the stars of Hyades, Orion, Sirius, and Ursa Major combined... My soul, heart, body and kingdom are yours for the taking. When I think of you, Cleo, my heart beats like that of a gladiator facing death at the claws of a rabid tiger ...

Caesar Cipher Encoding (Design 1)

Letter	a	b	c	d	…	…	w	x	y	z
Orig	97	98	99	100	…	…	119	120	121	122
Shift (+3)	100	101	102	103			122	123	124	125
Cipher	100	101	102	103			122	97	98	99

```
int cipher = ch + key;
if (cipher > 'z') cipher -= 26;
ch  = (char)cipher;
```

Q: What if key is 100?

a b c d e f g h i j k l m n o p q r s t u v w x y z

d e f g h i j k l m n o p q r s t u v w x y z a b c

Caesar Cipher: Encoding (Design 2)

Letter	a	b	c	d	...	...	w	x	y	z
Orig	0	1	2	3	...	...	22	23	24	25
Shift (+3)	3	4	5	6			25	26	27	28
Cipher (% 26)	3	4	5	6			25	0	1	2

int cipherInt = (origInt + key) % 26

Assumption: a-z mapped to 0 to 25.

How to map a char in a-z to 0 to 25? char – 'a'

a b c d e f g h i j k l m n o p q r s t u v w x y z

d e f g h i j k l m n o p q r s t u v w x y z a b c

Caesar Cipher Encoding

Letter	a	b	c	d	…	…	w	x	y	z
Orig	0	1	2	3	…	…	22	23	24	25
Shift (+3)	3	4	5	6			25	26	27	28
Cipher (% 26)	3	4	5	6			25	0	1	2

```
int chRelativeIndex = ch – ‘a’;
int cipherRelInd = (chRelativeIndex + key) % 26;
ch = (char)(cipherRelInd + ‘a’);
```

a b c d e f g h i j k l m n o p q r s t u v w x y z

d e f g h i j k l m n o p q r s t u v w x y z a b c

Offline Exercise/Think: How to Decode? (See Backup Slides)

Extension: Encrypt a Text File

- Goal: Instead of reading text from terminal, user specifies a file name, which gives the content to be encrypted.

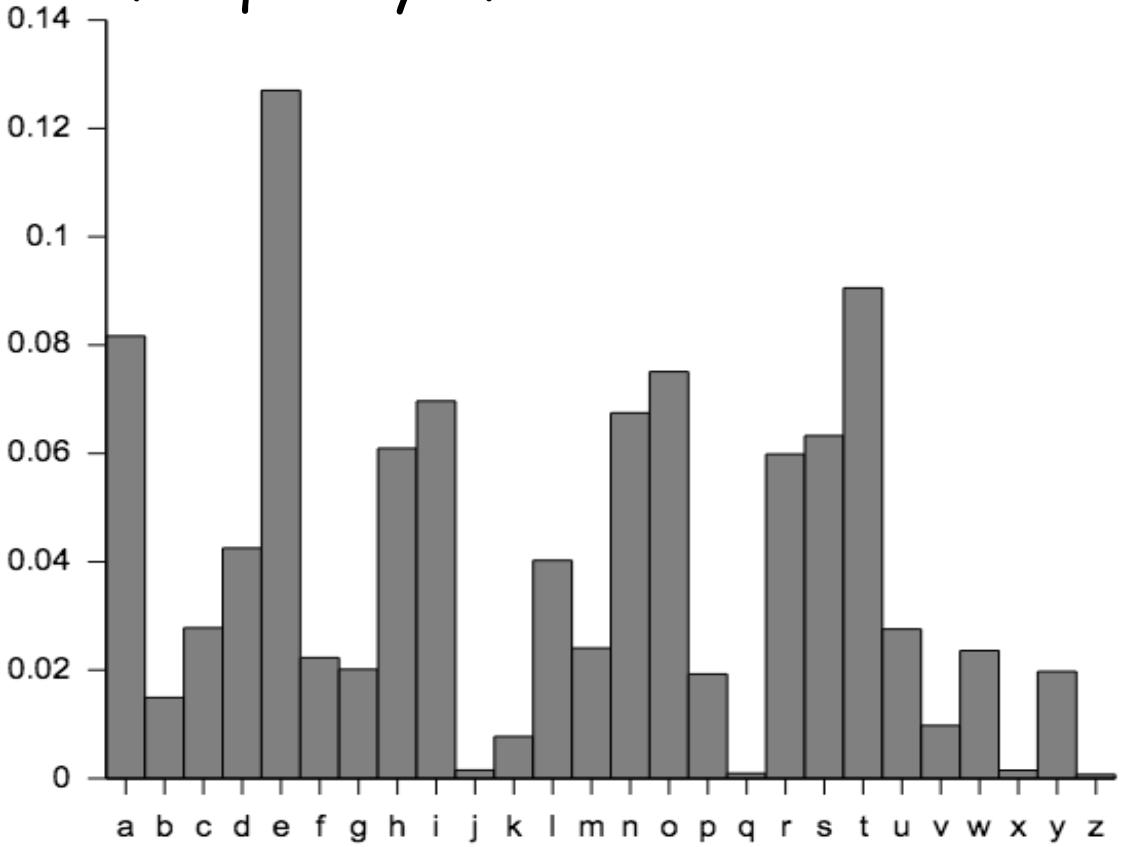
CaesarFile using a for Loop

```
public class ReadFile {
    public static void main(String[] args) {
        try {
            Scanner input = new Scanner(new File("data.txt"));
            encode(input, 3);
        } catch (FileNotFoundException e) {
            // print an error message
            System.out.println("File not found exception");
        } // end of catch
    } // end of main
}
```

```
public static void encode(Scanner scan, int key) {

  for (int line = 0; line < NLINES; line ++ ) {
       String text = scan.nextLine();
       String result = Caesar.encode(text, key);
       System.out.println( result );
  }
}
```

Exercise: Breaking Caesar Encryption

- Caesar encryption (or permutation ciphers in general) is easy to break because it does not change the frequency of letters.



Caesar Cipher: Decoding

Letter	*a*	*b*	*c*	*d*	*...*	*...*	*w*	*x*	*y*	*z*
Orig	*0*	*1*	*2*	*3*	*...*	*...*	*22*	*23*	*24*	*25*
Shift (+3)	3	4	5	6			25	26	27	28
Cipher (% 26)	3	4	5	6			25	0	1	2
Back (-3)	0	1	2	3			22	-3	-2	-1
Range (+26% 26)	0	1	2	3			22	23	24	25

Unifying Design

$$\text{int cipherInt} = (\text{origInt} + \text{key}) \% 26$$

$$\text{int origInt} = (\text{cipherInt} - \text{key} + 26) \% 26$$

- A single encode method with parameter key
  - To encode
    - encode(key)
  - To decode
    - encode(26 - key)

Caesar.java