Think Java

CHAPTER 5: CONDITIONALS AND LOGIC

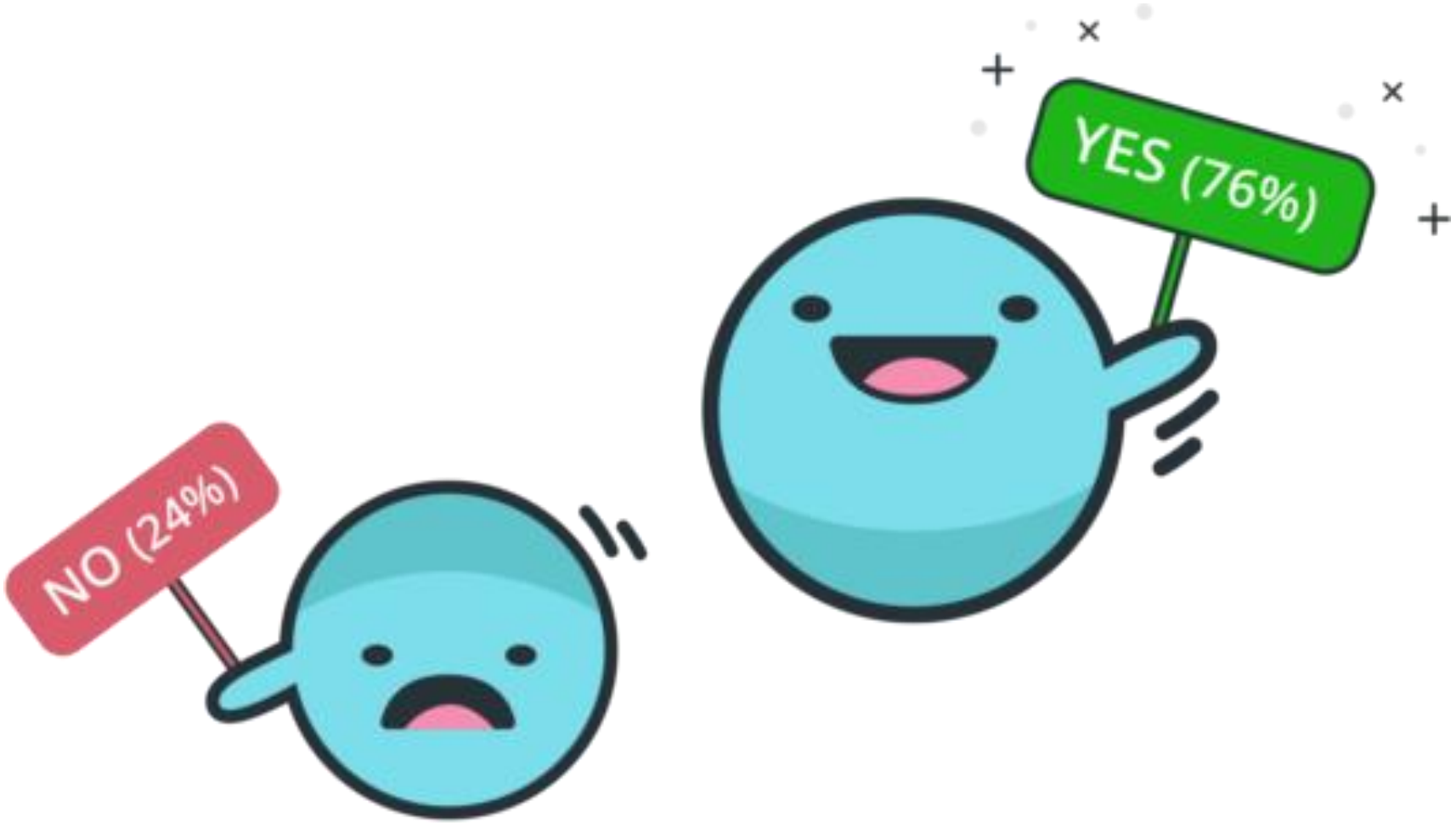
DR. ERIC CHOU IEEE SENIOR MEMBER

Objectives

- The programs we've seen in previous chapters do pretty much the same thing every time, regardless of the input. For more complex computations, programs usually react to inputs, check for certain conditions, and generate applicable results.
- This chapter introduces Java language features for expressing logic and making decisions. We'll also take a look at the Math class, which provides methods for common mathematical operations.

Topics

- Relational Operator
- Decision
- If-Then-Else
- Multiple-way if
- Nested if-statement
- Boolean Value, Variable and Method
- Validation of Input


NO (24%)
YES (76%)

LECTURE 1

Relational operators

Relational Operators

- Java has six relational operators that test the relationship between two values (e.g., whether they are equal, or whether one is greater than the other). The following expressions show how they are used:

```
x == y              // x is equal to y
x != y              // x is not equal to y
x > y               // x is greater than y
x < y               // x is less than y
x >= y              // x is greater than or equal to y
x <= y              // x is less than or equal to y
```

- The result of a relational operator is one of two special values: true or false. These values belong to the data type boolean, named after the mathematician George Boole. He developed an algebraic way of representing logic.

Relational Operators

- You are probably familiar with these operators, but notice how Java is different from mathematical symbols like =, ≠, and ≥. A common error is to use a single = instead of a double == when comparing values. Remember that = is the assignment operator, and == is a relational operator. In addition, the operators =< and => do not exist.
- The two sides of a relational operator have to be compatible. For example, the expression 5 < "6" is invalid because 5 is an int and "6" is a String. When comparing values of different numeric types, Java applies the same conversion rules we saw previously with the assignment operator. For example, when evaluating the expression 5 < 6.0, Java automatically converts the 5 to 5.0.

LECTURE 2

The if-else statement

Boolean Expression (Condition)

- To write useful programs, we almost always need to check conditions and react accordingly. Conditional statements give us this ability. The simplest conditional statement in Java is the if statement:

```
if (x > 0) {
    System.out.println("x is positive");
}
```

- The expression in parentheses is called the condition. If it is true, the statements in braces get executed. If the condition is false, execution skips over that block of code. The condition in parentheses can be any boolean expression.

Two-Way if-statements

- A second form of conditional statement has two possibilities, indicated by if and else. The possibilities are called branches, and the condition determines which one gets executed:

```
if (x % 2 == 0) {
    System.out.println("x is even");
} else {
    System.out.println("x is odd");
}
```

- If the remainder when x is divided by 2 is zero, we know that x is even, and the program displays a message to that effect. If the condition is false, the second print statement is executed instead. Since the condition must be true or false, exactly one of the branches will run.

If-Statement

The braces are optional for branches that have only one statement. So we could have written the previous example this way:

```
if (x % 2 == 0)
    System.out.println("x is even");
else
    System.out.println("x is odd");
```

However, it's better to use braces – even when they are optional – to avoid making the mistake of adding statements to an if or else block and forgetting to add the braces. This code is misleading because it's not indented correctly:

```
if (x > 0)
    System.out.println("x is positive");
    System.out.println("x is not zero");
```

Single-Statement in a Code Block

- Since there are no braces, only the first **println** is part of the if statement. Here is what the compiler actually sees:

```
if (x > 0) {
    System.out.println("x is positive");
}
    System.out.println("x is not zero");
```

- As a result, the second **println** runs no matter what. Even experienced programmers make this mistake; search the web for Apple's "goto fail" bug.

Code Block

- In all previous examples, notice how there is no semicolon at the end of the if or else lines. Instead, a new block should be defined using curly braces. Another common mistake is to put a semicolon after the condition, like this:

```
int x = 1;
if (x % 2 == 0); {  // incorrect semicolon
    System.out.println("x is even");
}
```

Erroneous Statement

- This code will compile, but the program will output "x is even" regardless what value x is. Here is the same incorrect code with better formatting:

```
int x = 1;
if (x % 2 == 0)
    ;  // empty statement
{
    System.out.println("x is even");
}
```

Erroneous Statement

- Because of the semicolon, the if statement compiles as if there are no braces, and the subsequent block runs independently. As a general rule, each line of Java code should end with a semicolon or brace – but not both.
- The compiler won't complain if you omit optional braces or write empty statements. Doing so is allowed by the Java language, but it often results in bugs that are difficult to find. Development tools like **Checkstyle** (see Appendix A.5) can warn you about these and other kinds of programming mistakes.

In-Class Demo Program

- Checking Even Numbers

EVENCHECKING.JAVA

LECTURE 3

Chaining and nesting

Multiple-Way if-statement

- Sometimes you want to check related conditions and choose one of several actions. One way to do this is by chaining a series of if and else blocks:

```
if (x > 0) {
    System.out.println("x is positive");
} else if (x < 0) {
    System.out.println("x is negative");
} else {
    System.out.println("x is zero");
}
```

- These chains can be as long as you want, although they can be difficult to read if they get out of hand. One way to make them easier to read is to use standard indentation, as demonstrated in these examples. If you keep all the statements and braces lined up, you are less likely to make syntax errors.

Nested If-Statement

- Notice that the last branch is simply else, not else if (x == 0). At this point in the chain, we know that x is not positive and x is not negative. There is no need to test whether x is zero, because there is no other possibility.
- In addition to chaining, you can also make complex decisions by nesting one conditional statement inside another. We could have written the previous example as:

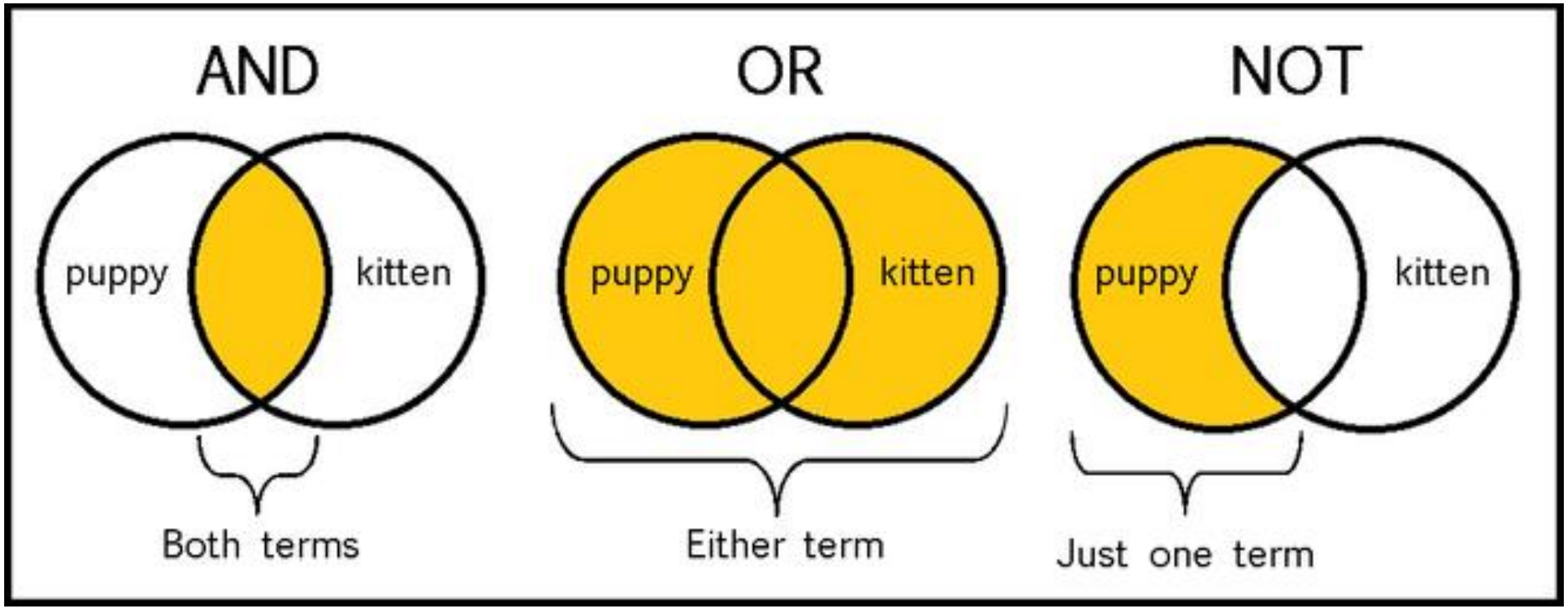
```
if (x > 0) {
    System.out.println("x is positive");
} else {
    if (x < 0) {
        System.out.println("x is negative");
    } else {
        System.out.println("x is zero");
    }
}
```

Dangling if-else

- The outer conditional has two branches. The first branch contains a print statement, and the second branch contains another conditional statement, which has two branches of its own. These two branches are also print statements, but they could have been conditional statements as well.
- These kinds of nested structures are common, but they can become difficult to read very quickly. Good indentation is essential to make the structure (or intended structure) apparent to the reader.

LECTURE 4

Logical operators


AND
puppy
kitten
Both terms
OR
puppy
kitten
Either term
NOT
puppy
kitten
Just one term

Boolean values and calculations

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

- A boolean value must evaluate to true or false
- Two boolean values can be compared with **and** or **or**
- Use parentheses if you want to combine and and or, i.e. (x and y) or z or x and (y or z), to disambiguate

Logic Condition

- In addition to the **relational operators**, Java also has three logical operators: &&, ||, and !, which respectively stand for and, or, and not. The results of these operators are similar to their meanings in English.
- For example, x > 0 && x < 10 is true when x is both greater than zero and less than 10. The expression **evenFlag || n \% 3 == 0** is true if either condition is true, that is, if **evenFlag** is true or the number n is divisible by 3. Finally, the ! operator inverts a boolean expression. So !evenFlag is true if evenFlag is false.
- In order for an expression with && to be true, both sides of the && operator must be true. And in order for an expression with || to be false, both sides of the || operator must be false.

Simplification of Logic Condition

- The && operator can be used to simplify nested if statements. For example, following code can be rewritten with a single condition.

```
if (x == 0) {
    if (y == 0) {
        System.out.println("Both x and y are zero");
    }
}
// combined
if (x == 0 && y == 0) {
    System.out.println("Both x and y are zero");
}
```

Simplification of Logic Condition

- Likewise, the || operator can simplify chained if statements. Since the branches are the same, there is no need to duplicate that code.

```
if (x == 0) {
    System.out.println("Either x or y is zero");
} else if (y == 0) {
    System.out.println("Either x or y is zero");
}
// combined
if (x == 0 || y == 0) {
    System.out.println("Either x or y is zero");
}
```

Simplification of Logic Condition

- Of course if the statements in the branches were different, we could not combine them into one block. But it's useful to explore different ways of representing the same logic, especially when it's complex.
- Logical operators evaluate the second expression only when necessary. For example, true || anything is always true, so Java does not need to evaluate the expression anything. Likewise, false && anything is always false.
- Ignoring the second **operand**, when possible, is called short circuit evaluation, by analogy with an electrical circuit. Short circuit evaluation can save time, especially if anything takes a long time to evaluate. It can also avoid unnecessary errors, if anything might fail.

LECTURE 5

De Morgan’s laws

Basic Rules of Boolean Algebra

1. $A + 0 = A$	7. $A \cdot A = A$
2. $A + 1 = 1$	8. $A \cdot \overline{A} = 0$
3. $A \cdot 0 = 0$	9. $\overline{\overline{A}} = A$
4. $A \cdot 1 = A$	10. $A + AB = A$
5. $A + A = A$	11. $A + \overline{A}B = A + B$
6. $A + \overline{A} = 1$	12. $(A + B)(A + C) = A + BC$

DeMorgan's Theorem

$$\overline{(AB)} = (\overline{A} + \overline{B}) \qquad \overline{(A + B)} = (\overline{A}\ \overline{B})$$

De Morgan's Laws

- Sometimes you need to negate an expression containing a mix of relational and logical operators. For example, to test if x and y are both nonzero, you could write:

```
if (!(x == 0 || y == 0)) {
    System.out.println("Neither x nor y
is zero");
}
```

De Morgan's Laws

- This condition is difficult to read because of the ! and parentheses. A better way to negate logic expressions is to apply De Morgan's laws:

```
!(A && B)  is the same as  !A || !B
!(A || B)  is the same as  !A && !B
```

- Negating a logical expression is the same as negating each term and changing the operator. The ! operator takes precedence over && and ||, so you don't have to put parentheses around the individual terms !A and !B.

De Morgan's Laws

- De Morgan's laws also apply to the relational operators. In this case, negating each term means using the "opposite" relational operator.

```
!(x < 5 && y == 3)  is the same as  x >= 5 || y != 3
!(x >= 1 || y != 7)  is the same as  x < 1 && y == 7
```

- It may help to read these examples out loud in English. For instance, "If I don't want the case where x is less than 5 and y is 3, then I need x to be greater than or equal to 5, or I need y to be anything but 3."

De Morgan's Laws

- Returning to the previous example, here is the revised condition. In English, it reads "if x is not zero and y is not zero." The logic is the same, and the source code is easier to read.

```
if (x != 0 && y != 0) {
    System.out.println("Neither x nor y
is zero");
}
```

LECTURE 6

Boolean variables

Boolean Value and Boolean Variable

To store a true or false value, you need a boolean variable. You can declare and assign them like other variables. In this example, the first line is a variable declaration, the second is an assignment, and the third is both:

```
boolean flag;
flag = true;
boolean testResult = false;
```

Since relational and logical operators evaluate to a boolean value, you can store the result of a comparison in a variable:

```
boolean evenFlag = (n % 2 == 0); // true if n is even
boolean positiveFlag = (x > 0);  // true if x is positive
```

Boolean Condition

- The parentheses are unnecessary, but they make the code easier to understand. A variable defined in this way is called a flag, because it signals or “flags” the presence or absence of a condition.
- You can use flag variables as part of a conditional statement:

```
if (evenFlag) {
  System.out.println("n was even when I checked it");
}
```

Negation

- Flags may not seem that useful at this point, but they will help simplify complex conditions later on. Each part of a condition can be stored in a separate flag, and these flags can be combined with logical operators.
- Notice that you don't have to write if (evenFlag == true). Since evenFlag is a boolean, it's already a condition. Furthermore, to check if a flag is false:

```
if (!evenFlag) {
    System.out.println("n was odd when I checked it");
}
```

LECTURE 7

Boolean methods

Boolean Method

- Methods can return boolean values, just like any other type, which is often convenient for hiding tests inside methods. For example:

```
public static boolean isSingleDigit(int x) {
    if (x > -10 && x < 10) {
        return true;
    } else {
        return false;
    }
}
```

- The name of this method is isSingleDigit. It is common to give boolean methods names that sound like yes/no questions. Since the return type is boolean, the return statement has to provide a boolean expression.

Boolean Method

- The code itself is straightforward, although it is longer than it needs to be. Remember that the expression x > -10 && x < 10 has type boolean, so there is nothing wrong with returning it directly (without the if statement):

```
public static boolean isSingleDigit(int x) {
    return x > -10 && x < 10;
}
```

Boolean Method

- In main, you can invoke the method in the usual ways:

```
System.out.println(isSingleDigit(2));
boolean bigFlag = !isSingleDigit(17);
```

- The first line displays true because 2 is a single-digit number. The second line sets bigFlag to true, because 17 is not a single-digit number.

Boolean Method

- Conditional statements often invoke boolean methods and use the result as the condition:

```
if (isSingleDigit(z)) {
    System.out.println("z is small");
} else {
    System.out.println("z is big");
}
```

- Examples like this one almost read like English: “If is single digit z, print ... else print ...”

In-Class Demo Program

- Check if a year is leap year or not

LEAPYEAR.JAVA

LECTURE 8

Validating input

Validating input

- One of the most important tasks in any computer program is to validate input from the user. People often make mistakes while typing, especially on smartphones, and incorrect inputs may cause your program to fail.
- Even worse, someone (i.e., a hacker) may intentionally try to break into your system by entering unexpected inputs. You should never assume that users will input the right kind of data.

Validating input

- Consider this simple program that prompts the user for a number and computes its logarithm:

```
Scanner in = new Scanner(System.in);
System.out.print("Enter a number: ");
double x = in.nextDouble();
double y = Math.log(x);
System.out.println("The log is " + y);
```

- In mathematics, the natural logarithm (base e) is undefined when x < 0. If you ask for Math.log(-1), it will return NaN which stands for “not a number”. Many users are confused when they see NaN; it often looks like a bug. We can use an if statement to make the output more user friendly.

Validating input

```
if (x >= 0) {
    double y = Math.log(x);
    System.out.println("The log is " + y);
} else {
    System.out.println("The log is undefined");
}
```

- The output is better now, but there is another problem. What if the user doesn't enter a number at all? What would happen if they typed the word "hello", either on accident or on purpose?

```
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextDouble(Scanner.java:2413)
    at Logarithm.main(Logarithm.java:8)
```

Validating input

- If the user inputs a String when we expect a double, Java reports an “input mismatch” exception. We can prevent this run-time error from happening by testing the input first.
- The Scanner class provides hasNextDouble, which checks whether the next input can be interpreted as a double. If not, we can display an error message.

```
if (!in.hasNextDouble()) {
    String word = in.next();
    System.err.println(word + "is not a number");
    return;
}
```

Validating input

- In contrast to **in.nextLine**, which returns an entire line of input, the in.next method returns only the next token of input. We can use in.next to show the user exactly which word they typed was not a number.
- This example also uses **System.err**, which is an OutputStream for error messages and warnings. Some development environments display output to System.err with a different color or in a separate window.
- The return statement allows you to exit a method before you reach the end of it. Returning from main terminates the program.
- Notice the use of the ! operator before in.hasNextDouble(), instead of testing the condition in.hasNextDouble() == false. Since hasNextDouble returns a boolean, it is already a condition.

LECTURE 9

Example program

In-Class Demo Program

- Validation of Inputs

LOGARITHM.JAVA

Validated Input

- What started as five lines of code at the beginning of Section 5.8 is now a 30-line program. Making programs robust (and secure) often requires a lot of additional checking, as shown in this example.
- It's important to write comments every few lines to make your code easier to understand. Comments not only help other people read your code, they also help you document what you're trying to do. If there's a mistake the code, it's a lot easier to find when there are good comments.

Homework

Homework

- Textbook Exercise
- Project 5