

# 소프트웨어 공학

Dr. Young-Woo Kwon

# Programming in Java

- In Java, a program generally has the following structure:
  - Declaring variables/ initializations
    - E.g. `int balance;`
  - Processing
    - **Read** inputs
    - **Process** inputs
    - **Show** output
  - Exit the program
    - return, exit

# Data Types – Variables & Initializations

- **String** for strings of letters  
e.g. `String LastName = "Kwon";`
- **char** for characters  
e.g. `char alphabet = 'A';`
- **byte, short, int, and long** for integer numbers  
e.g. `int number = 3;`
- **float & double** for real numbers  
e.g. `double pi = 3.14159;`
- **boolean** for logical  
e.g. `boolean check = true;`

# Expressions and Operators

- Expressions:

( ), \* , / , + , -

- Operators:

&&	for	AND
	for	OR
!	for	NOT
==	for	equal
!=	for	NOT equal

# Selection Statements

- if statement                      ex)        

```
if (A>B) {  
    System.out.println("A>B");  
}
```
- if...else statement            ex)        

```
if (A>B) {  
    System.out.println("A>B");  
} else {  
    System.out.println("A<=B");  
}
```
- switch statement            ex)        

```
switch (A) {  
    case 1: {  
        System.out.println("A=1");  
        break;  
    }  
    case 2: {  
        System.out.println("A=2");  
        break;  
    }  
    ...  
    default: {  
        System.out.println("A Undefined");  
    }  
}
```

# Repetitions (while, do while, and for)

- Using While:

```
// Variable Declaration & Initialization
int sum = 0;
int count = 0;

// Processing
while (count <= 1000) {
    sum = sum + count;
    count = count +1; //same as count++
}

// Output
System.out.println("Sum: " + sum);
```

# Repetitions(while, do while, and for)

- Using do-while:

```
// Variable Declaration & Initialization
int sum =0;
int count =0;

// Processing
do {
    sum = sum + count;
    count = count +1; //same as count++
} while (count <=1000);
```

- Using for:

```
// Variable Declaration & Initialization
int sum =0;

// Processing
for (int count =0; count <= 100; count++) {
    sum = sum +count;
}
```

# Methods

*modifier*  
*return value type*  
*method name*  
*parameter list*

```
public static double readDouble() {  
    double d = 0.0;  
  
    try {  
        String str = df.readLine();  
        st = new StringTokenizer (str);  
        d = new Double (st.nextToken()).doubleValue();  
    } catch (IOException ex) {  
        System.out.println(ex);  
    }  
  
    return d;  
}
```



# Declaring Methods

```
/**
 * This method returns the maximum of two numbers
 * @param num1 the first number
 * @param num2 the second number
 * @return either num1 or num2
 */
int max(int num1, int num2) {
    if (num1 > num2) {
        return num1;
    } else {
        return num2;
    }
}
```

# Passing Parameters

```
/**
 * This method prints a message a specified number
 * of times.
 * @param message the message to be printed
 * @param n the number of times the message is to be
 * printed
 */
void nPrintln(String message, int n) {
    for (int i=0; i<n; i++) {
        System.out.println(message);
    }
}
```

# main method

- One of these methods is required for every program.
  - If your program does not have one, then you cannot run it.
- The main method has a very specific syntax:

```
public static void main(String[] args) {  
    // Your code goes here!  
}
```

# An Example Program

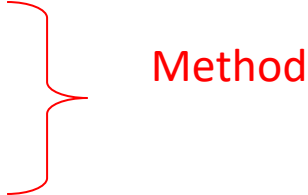

```
class MyProgram {  
    /**  
     * This method prints a message a specified number  
     * of times.  
     * @param message the message to be printed  
     * @param n the number of times the message is to be  
     *         printed  
     */  
    void nPrintln(String message, int n) {  
        for (int i=0; i<n; i++) {  
            System.out.println(message);  
        }  
    }  
  
    public static void main(String[] args) {  
        nPrintln("Hello World!", 10);  
    }  
}
```

# Your turn

- Sum from 1 to 10
    - E.g.,  $1 + 2 + 3 + \dots + 10$
1. Write `main()` method
  2. Use `for` or `while`
  3. Show an equation and its result
    - $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55$

# Class Declaration

```
class Circle {  
    public double radius = 1.0;  
  
    /**  
     * This method calculates the area of the circle.  
     * @return the area of the circle  
     */  
    public double findArea() {  
        return radius*radius*3.14159;  
    }  
}
```



← Data attribute

Method

# Declaring Object Variables

- Syntax:

```
ClassName objectName;
```

- Example:

```
Circle myCircle;
```

# Creating Objects

- Syntax:

```
objectName = new ClassName();
```

- Example:

```
myCircle = new Circle();
```



# Declaring/Creating Objects in a Single Step

- Syntax:

```
ClassName objectName = new ClassName();
```

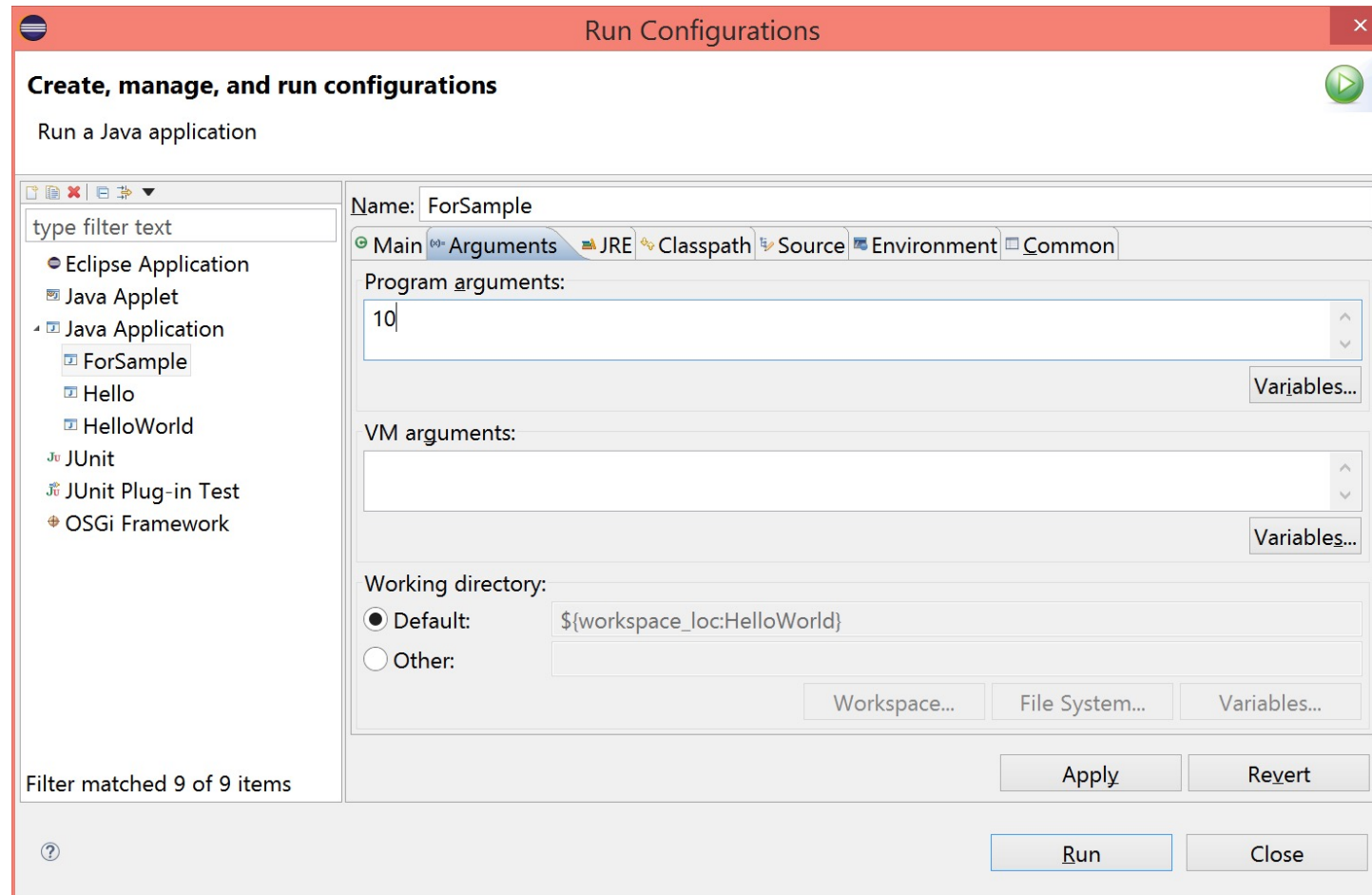
- Example:

```
Circle myCircle = new Circle();
```

# Accessing Objects

- Referencing the object's data:
  - Syntax:  
`objectName.data`
  - Example:  
`myCircle.radius`
- Referencing the object's method (sending a message):
  - Syntax:  
`objectName.method(...)`
  - Example:  
`myCircle.findArea()`

# Passing arguments to main()



# Type Conversion

- You can write your own function, but...
- String to Integer, Float, Double, etc.
  - `Integer.parseInt(string)`
  - `Double.parseDouble(string)`
- Integer, Float, Double to String
  - `String.valueOf(number)`
  - `"" + number`
  - `Integer.toString(number)`

# Your turn

- Let's extend the first program
  - Sum from 1 to  $n$ 
    - E.g.,  $1 + 2 + 3 + \dots + n$
1. Define `int sum(int n)` method
  2. main method takes the parameter  $n$  and passes it to the `sum()` method

# Constructors

- Constructors are special methods that are called upon creation of an object.
  - Their purpose is to initialise objects as they are created.

- **Syntax:**

```
ClassName(...) {  
    // Initialisation code goes here!  
}
```

- **Example:**

```
Circle(double r) {  
    radius = r;  
}
```

# Constructors

- Example (cont):

```
Circle() {  
    radius = 1.0;  
}
```

- We can specify which constructor to use when we create a new object.
- Example:

```
myCircle = new Circle(5.0);
```

# Modifiers

By default, the class, variable, or data 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.



# Your turn

- Let's extend the second program

- Sum from 1 to n
  - E.g.,  $1 + 2 + 3 + \dots + n$

1. Define Sum class and `sum(int n)` method

- Sum is a public class
- `sum()` is a public and static method

# Packages

- **Packages** are used to organise Java classes.
  - Packages are like domain names – they are hierarchical (e.g. packages can have sub-packages, which can have sub-packages, and so on...).
  - There are five basic Java packages:
    - java.lang: Classes that are part of the basic language (e.g. Strings)
    - java.io: Classes that do input and output
    - java.net: Classes for networking
    - java.awt: Classes for creating Graphical Interfaces
    - java.util: General Utility Classes
  - So, if I want a class that does file input, I look in the input/output (io) package!

# Referencing Classes in Different Packages

- If we want to use a class that is in a package (with the exception of those in the java.lang package), we need to declare our intention at the top of the program.
- We do this with an **import** statement
  - For example:

```
import java.io.PrintStream;
```

- If we want to use multiple classes from a single package we can use the following statement:

```
import java.io.*;
```



This means "any class"

# Some Useful Java Classes

- One of the most useful sets of Java classes are the input/output (io) classes.
  - These classes are located in the java.io package and are part of the default Java API.
  - These classes allow programs to perform input/output in a variety of ways.
- There are many different types of io class.
  - You are not expected to know them all intimately, but you should know how to do a few basic things with them (e.g. User input, File input/output, etc.)

# System.out

- You have already used an io class implicitly when you write information to the screen.
- System.out refers to a data attribute of the System class.
  - This attribute is an instance of the class **PrintStream**.
  - This class is used for output only.
- The PrintStream class includes a method, **println**, that prints a string to the specified output.
  - We don't know where this output comes from!
  - We don't need to know – we just need to know where it goes to (i.e. the screen).

# Handling User Input

- To get input from the user, we use another data attribute of the System class:

```
System.in
```

- This attribute is an instance of the **InputStream** class.
  - The InputStream class is not easy to use (it works only with bytes).
  - To make things easier, we can wrap the input stream in other io classes.
- In particular, we use two classes:
  - **InputStreamReader**. This class converts the byte stream into a character stream (this is much more useful – honestly!).
  - **BufferedReader**. This class buffers data from the enwrapped stream, allowing smoother reading of the stream (it doesn't wait for the program to ask for the next character in the stream).

# Handling User Input

- To wrap an io object, we create a new io object, passing the old io object into the constructor.
  - For example, to wrap System.in within an InputStreamReader, we write:

```
new InputStreamReader(System.in)
```

- To use the BufferedReader (which, remember, is an optimisation), we wrap the InputStreamReader object we created above.
  - For example:

```
new BufferedReader(new InputStreamReader(System.in))
```

- Once we have a BufferedReader, we can use a nice method it provides called readLine() which reads a line of text!
  - This makes life much easier!

# Handling User Input

- Now that we have a way of reading a line of text from the System input stream (i.e. the keyboard), how do we use it?

```
public String readString() {  
    BufferedReader in = new BufferedReader(  
                                new InputStreamReader(System.in));  
  
    String line = null;  
    try {  
        line = in.readLine();  
        while (line == null) {  
            line = in.readLine();  
        }  
    } catch (IOException ie) {  
        System.out.println("The following problem occurred " +  
                            "when reading input: " + ie);  
    }  
    return line;  
}
```



# Handling User Input

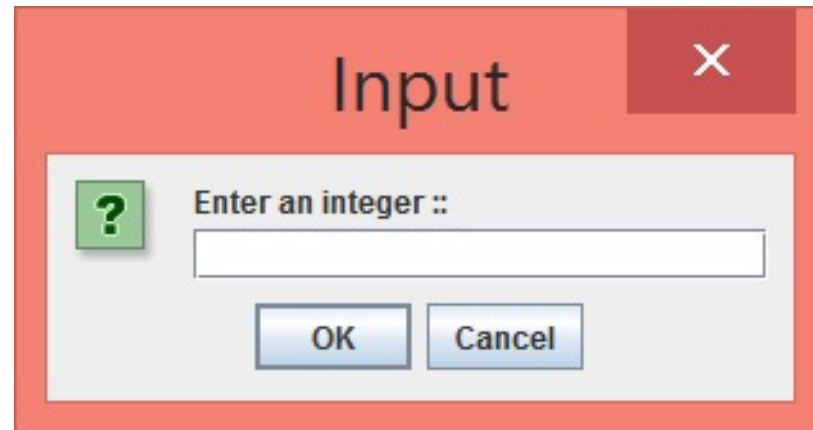
- Scanner
  - Read a line of input from its source
  - Example:

```
Scanner sc = new Scanner (System.in);  
int i = sc.nextInt();  
System.out.println("You entered" + i);
```

- This example reads a single `int` from `System.in` and outputs it to `System.out`. It does not check that the user actually entered an `int`.

# Handling User Input

- Java Swing (GUI)
  - `JOptionPane.showInputDialog()`



# Your turn

- Let's extend the third program
    - Add  $n$  integers
1. Define `sum(int[])`
  2. Using one of user input mechanisms, take  $n$  integers from the console
    1. First, you will give the number of integers
    2. Then, you will give  $n$  integers
    3. They are stored into an integer array then passed to `sum(int[])`

# I/O and File objects

- The `File` class in the `java.io` package represents files.
  - `import java.io.*;`
  - Create a `File` object to get information about a file on the disk. (Creating a `File` object doesn't create a new file on your disk.)

```
File f = new File("example.txt");
if (f.exists() && f.length() > 1000) {
    f.delete();
}
```

Method name	Description
<code>canRead()</code>	returns whether file is able to be read
<code>delete()</code>	removes file from disk
<code>exists()</code>	whether this file exists on disk
<code>getName()</code>	returns file's name
<code>length()</code>	returns number of characters in file
<code>renameTo(<i>file</i>)</code>	changes name of file

# Scanners to read files

- To read a file, create a `File` object and pass it as a parameter when constructing a `Scanner`.
- **Creating a `Scanner` for a file, general syntax:**

```
Scanner <name> = new Scanner(new File("<file name>"));
```

**Example:**

```
File f = new File("numbers.txt");
```

```
Scanner input = new Scanner(f);
```

**or:**

```
Scanner input = new Scanner(new File("numbers.txt"));
```

# File and path names

- **relative path:** does not specify any top-level folder
  - `"names.dat"`
  - `"input/kinglear.txt"`
- **absolute path:** specifies drive letter or top `" / "` folder
  - `"C:/Documents/smith/hw6/input/data.csv"`
  - Windows systems also use backslashes to separate folders. How would the above filename be written using backslashes?
- When you construct a `File` object with a relative path, Java assumes it is relative to the *current directory*.
  - `Scanner input = new Scanner(new File("data/readme.txt"));`
  - If our program is in: `H:/johnson/hw6`,  
Java will look for: `H:/johnson/hw6/data/readme.txt`.

# Compiler error with files

- The following program does not compile:

```
import java.io.*;    // for File
import java.util.*;  // for Scanner

public class ReadFile {
    public static void main(String[] args) {
        Scanner input = new Scanner(new File("data.txt"));
        String text = input.next();
        System.out.println(text);
    }
}
```

- The following compiler error is produced:

```
ReadFile.java:6: unreported exception java.io.FileNotFoundException; must be caught or
declared
to be thrown
    Scanner input = new Scanner(new File("data.txt"));
                                ^
```

# Files and input cursor

- Consider a file `numbers.txt` that contains this text:

```
308.2
 14.9 7.4 2.8

3.9 4.7 -15.4
 2.8
```

- A `Scanner` views all input as a stream of characters, which it processes with its *input cursor*:
  - `308.2\n 14.9 7.4 2.8\n\n3.9 4.7 -15.4\n2.8\n`  
^
- When you call the methods of the `Scanner` such as `next` or `nextDouble`, the `Scanner` breaks apart the input into *tokens*.



# Input tokens

- **token:** A unit of user input. Tokens are separated by whitespace (spaces, tabs, new lines).
- Example: If an input file contains the following:

```
23    3.14  
"John Smith"
```

- The tokens in the input are the following, and can be interpreted as the given types:

<u>Token</u>	<u>Type(s)</u>
1. 23	int, double, String
2. 3.14	double, String
3. "John	String
4. Smith"	String

# Consuming tokens

- Each call to `next`, `nextInt`, `nextDouble`, etc. advances the cursor to the end of the current token, skipping over any whitespace.
  - We call this *consuming* input.

```
input.nextDouble()
```

- **308.2**\n    14.9 7.4    2.8\n\n\n3.9 4.7 -15.4\n2.8\n  
          ^

```
input.nextDouble()
```

- 308.2\n    **14.9** 7.4    2.8\n\n\n3.9 4.7 -15.4\n2.8\n  
          ^

# Your turn

- Consider an input file named `numbers.txt` that contains the following text:

```
308.2  14.9 7.4  2.8 3.9 4.7 -15.4 2.8
```

- Write a program that reads the first 5 values from this file and prints them along with their sum. Its output:

```
number = 308.2  
number = 14.9  
number = 7.4  
number = 2.8  
number = 3.9  
Sum = 337.19999999999993
```

# File input answer

```
// Displays the first 5 numbers in the given file,  
// and displays their sum at the end.
```

```
import java.io.*;    // for File, FileNotFoundException  
import java.util.*;
```

```
public class Echo {  
    public static void main(String[] args)  
        throws FileNotFoundException {  
        Scanner input = new Scanner(new File("numbers.txt"));  
        double sum = 0.0;  
        for (int i = 1; i <= 5; i++) {  
            double next = input.nextDouble();  
            System.out.println("number = " + next);  
            sum += next;  
        }  
        System.out.println("Sum = " + sum);  
    }  
}
```

# Test before read question

- Rewrite the previous program so that it reads the entire file. Its output:

```
number = 308.2
number = 14.9
number = 7.4
number = 2.8
number = 3.9
number = 4.7
number = -15.4
number = 2.8
Sum = 329.29999999999995
```

# Test before read answer

```
// Displays each number in the given file,  
// and displays their sum at the end.  
  
import java.io.*;  
import java.util.*;  
  
public class Echo2 {  
    public static void main(String[] args)  
        throws FileNotFoundException {  
        Scanner input = new Scanner(new File("numbers.dat"));  
        double sum = 0.0;  
        while (input.hasNextDouble()) {  
            double next = input.nextDouble();  
            System.out.println("number = " + next);  
            sum += next;  
        }  
        System.out.println("Sum = " + sum);  
    }  
}
```

# File processing question

- Modify the preceding program again so that it will handle files that contain non-numeric tokens.
  - The program should skip any such tokens.
- For example, the program should produce the same output as before when given this input file:

```
308.2  hello
      14.9 7.4  bad stuff 2.8

3.9 4.7  oops  -15.4
:-)      2.8  @#* ($&
```

# File processing answer

```
// Displays each number in the given file,  
// and displays their sum at the end.  
  
import java.io.*;  
import java.util.*;  
  
public class Echo3 {  
    public static void main(String[] args)  
        throws FileNotFoundException {  
        Scanner input = new Scanner(new File("numbers.dat"));  
        double sum = 0.0;  
        while (input.hasNext()) {  
            if (input.hasNextDouble()) {  
                double next = input.nextDouble();  
                System.out.println("number = " + next);  
                sum += next;  
            } else {  
                input.next();    // consume / throw away bad token  
            }  
        }  
        System.out.println("Sum = " + sum);  
    }  
}
```

48



# File processing question

- Write a program that accepts an input file containing integers representing daily high temperatures.

Example input file:

```
42 45 37 49 38 50 46 48 48 30 45 42 45 40 48
```

- Your program should print the difference between each adjacent pair of temperatures, such as the following:

```
Temperature changed by 3 deg F
Temperature changed by -8 deg F
Temperature changed by 12 deg F
Temperature changed by -11 deg F
Temperature changed by 12 deg F
Temperature changed by -4 deg F
Temperature changed by 2 deg F
Temperature changed by 0 deg F
Temperature changed by -18 deg F
Temperature changed by 15 deg F
Temperature changed by -3 deg F
Temperature changed by 3 deg F
Temperature changed by -5 deg F
Temperature changed by 8 deg F
```

# File processing answer

```
import java.io.*;
import java.util.*;

public class Temperatures {
    public static void main(String[] args)
        throws FileNotFoundException {
        Scanner input = new Scanner(new File("weather.dat"));
        int temp1 = input.nextInt();

        while (input.hasNextInt()) {
            int temp2 = input.nextInt();
            System.out.println("Temperature changed by " +
                               (temp2 - temp1) + " deg F");
            temp1 = temp2;
        }
    }
}
```

# Conditionals and Loops

- Now we will examine programming statements that allow us to:
  - make decisions
  - repeat processing steps in a loop

# Conditional Statements

- A *conditional statement* lets us choose which statement will be executed next
- Therefore they are sometimes called *selection statements*
- Conditional statements give us the power to make basic decisions
- The Java conditional statements are the:
  - *if statement*
  - *if-else statement*
  - *switch statement*

# The if Statement

- The *if statement* has the following syntax:

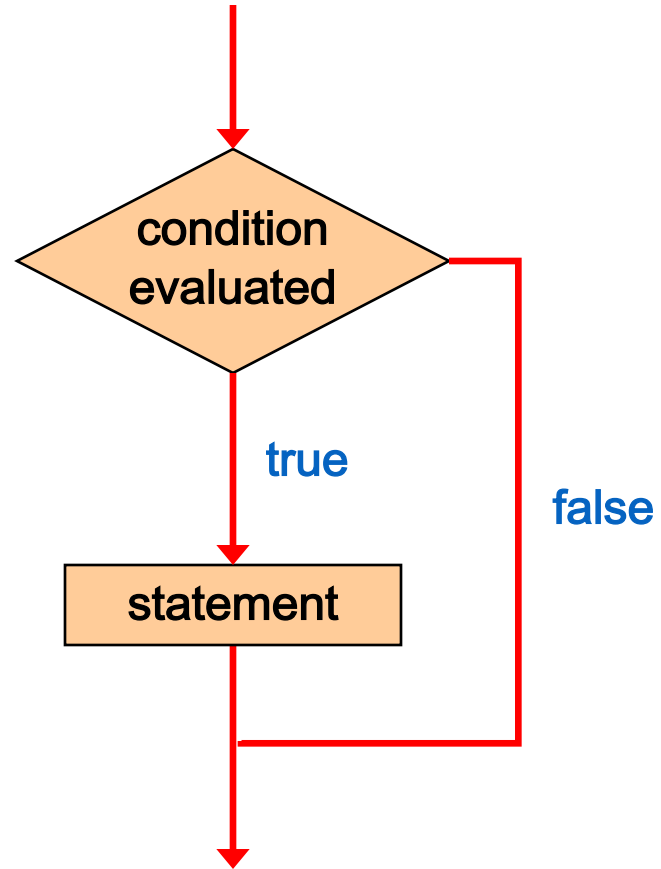
`if` is a Java  
reserved word

The *condition* must be a  
boolean expression. It must  
evaluate to either true or false.

`if ( condition )  
    statement;`

If the *condition* is true, the *statement* is executed.  
If it is false, the *statement* is skipped.

# Logic of an if statement



# Boolean Expressions

- A condition often uses one of Java's *equality operators* or *relational operators*, which all return boolean results:

==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

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

# The if Statement

- An example of an `if` statement:

```
if (sum > MAX)
    delta = sum - MAX;
System.out.println ("The sum is " + sum);
```

- **First the condition is evaluated -- the value of `sum` is either greater than the value of `MAX`, or it is not**
- **If the condition is true, the assignment statement is executed -- if it isn't, it is skipped.**
- **Either way, the call to `println` is executed next**



# The if Statement

- What do the following statements do?

```
if (top >= MAXIMUM)
    top = 0;
```

Sets `top` to zero if the current value of `top` is greater than or equal to the value of `MAXIMUM`

```
if (total != stock + warehouse)
    inventoryError = true;
```

Sets a flag to true if the value of `total` is not equal to the sum of `stock` and `warehouse`

- The precedence of the arithmetic operators is higher than the precedence of the equality and relational operators

# Logical Operators

- Boolean expressions can also use the following *logical operators*:

!	Logical NOT
& &	Logical AND
	Logical OR

- They all take boolean operands and produce boolean results
- Logical NOT is a unary operator (it operates on one operand)
- Logical AND and logical OR are binary operators (each operates on two operands)

# Logical NOT

- The *logical NOT* operation is also called *logical negation* or *logical complement*
- If some boolean condition  $a$  is true, then  $!a$  is false; if  $a$  is false, then  $!a$  is true
- Logical expressions can be shown using a *truth table*

$a$	$!a$
true	false
false	true

# Logical AND and Logical OR

- The *logical AND* expression

`a && b`

is true if both `a` and `b` are true, and false otherwise

- The *logical OR* expression

`a || b`

is true if `a` or `b` or both are true, and false otherwise

# Logical Operators

- Expressions that use logical operators can form complex conditions

```
if (total < MAX+5 && !found)
    System.out.println ("Processing..") ;
```

- **All logical operators have lower precedence than the relational operators**
- **Logical NOT has higher precedence than logical AND and logical OR**

# Logical Operators

- A truth table shows all possible true-false combinations of the terms
- Since `&&` and `||` each have two operands, there are four possible combinations of conditions `a` and `b`

<code>a</code>	<code>b</code>	<code>a &amp;&amp; b</code>	<code>a    b</code>
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

# Boolean Expressions

- Specific expressions can be evaluated using truth tables

<code>total &lt; MAX</code>	<code>found</code>	<code>!found</code>	<code>total &lt; MAX &amp;&amp; !found</code>
false	false	true	false
false	true	false	false
true	false	true	true
true	true	false	false

# The if-else Statement

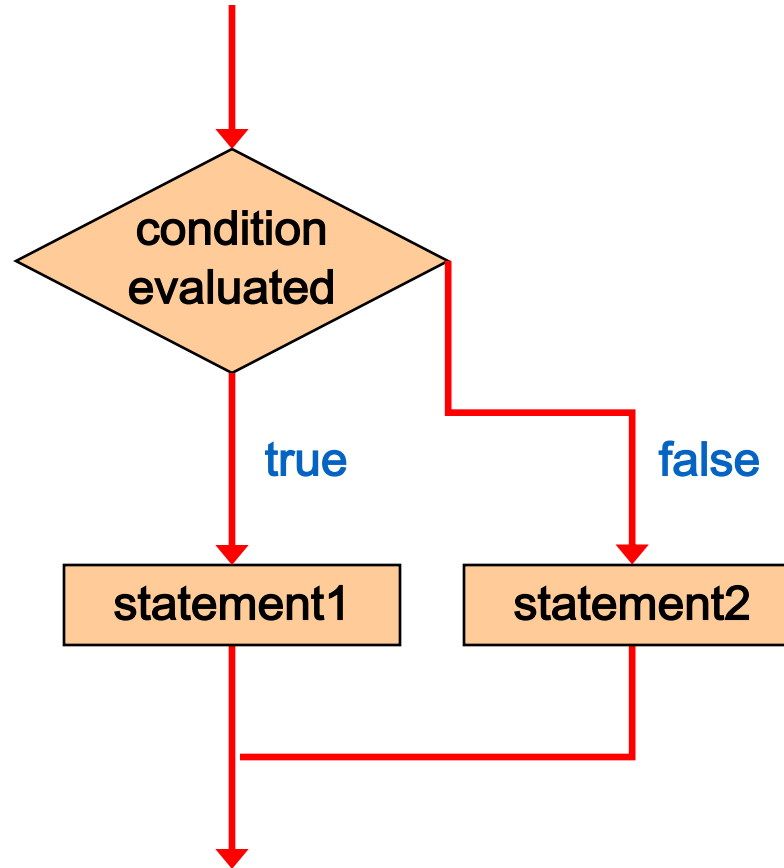
- An *else clause* can be added to an `if` statement to make an *if-else statement*

```
if ( condition )  
    statement1;  
else  
    statement2;
```

- If the *condition* is true, *statement1* is executed;  
if the condition is false, *statement2* is executed
- One or the other will be executed, but not both



# Logic of an if-else statement



# The switch Statement

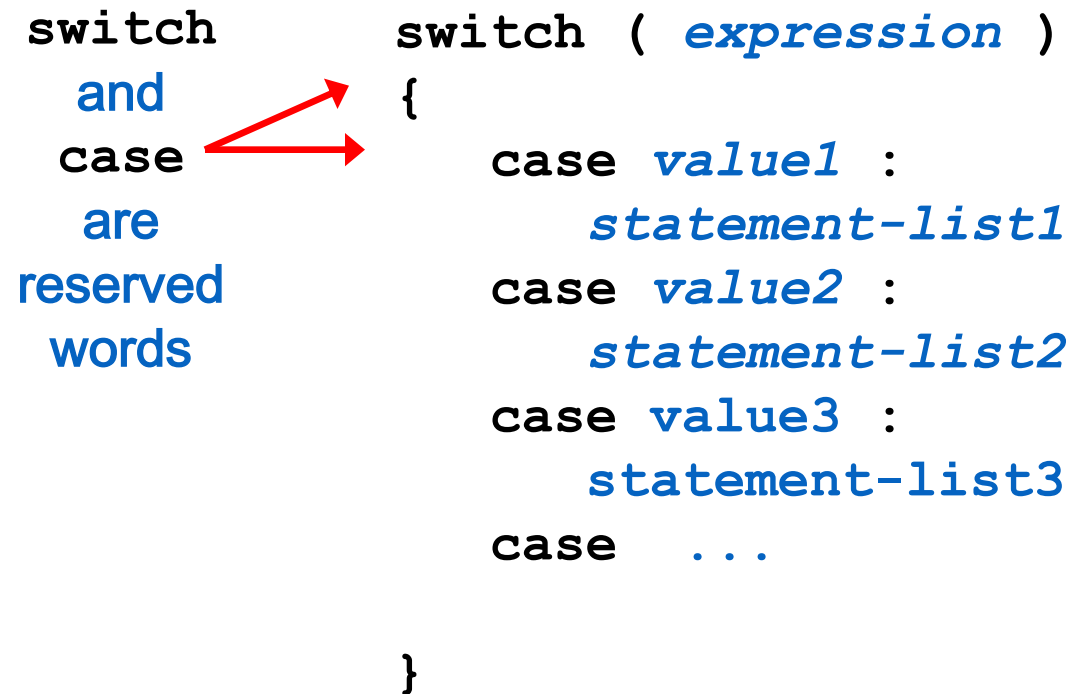
- The *switch statement* provides another way to decide which statement to execute next
- The `switch` statement evaluates an expression, then attempts to match the result to one of several possible *cases*
- Each case contains a value and a list of statements
- The flow of control transfers to statement associated with the first case value that matches

# The switch Statement

- The general syntax of a `switch` statement is:

`switch`  
`and`  
`case`  
`are`  
`reserved`  
`words`

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



If *expression*  
matches *value2*,  
control jumps  
to here

# The switch Statement

- Often a *break statement* is used as the last statement in 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
- Sometimes this may be appropriate, but often we want to execute only the statements associated with one case

# The switch Statement

- An example of a switch statement:

```
switch (option)
{
    case 'A':
        aCount++;
        break;
    case 'B':
        bCount++;
        break;
    case 'C':
        cCount++;
        break;
}
```

# The switch Statement

- A `switch` statement can have an optional *default case*
- 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

- The expression of a `switch` statement must result in an *integral type*, meaning an `int` or a `char`
- It cannot be a `boolean` value, a floating point value (`float` or `double`), or another integer type
- The implicit boolean condition in a `switch` statement is equality
- You cannot perform relational checks with a `switch` statement

# Your turn

- Grade Report
  - Reads a score from the user and prints a grade
  - E.g., 89 = B+, 91 = A-, ...



# Home Work

- Grade Report
  - Read students' scores from a file
  - And prints their statistics
    - Total student number
    - Average, variance, standard deviation, etc
    - Write the results into a file