

# COMP 202


## More on Chapter 2


### CONTENTS:


Compilation

Data and Expressions

# Programming Language Levels

- There are many programming language levels:
  - machine language 

```
011001011
110111100
```
  - assembly language 

```
Add r1,5
Mov r1,r2
```
  - high-level language 

```
Java, C, C++,
Fortran
```
- Each type of CPU (Sparc processor, Intel processor, ...) has its own specific *machine language*. These are the simple built-in instructions the CPU comes pre-designed with.
- The other levels were created to make it easier for a human to write programs

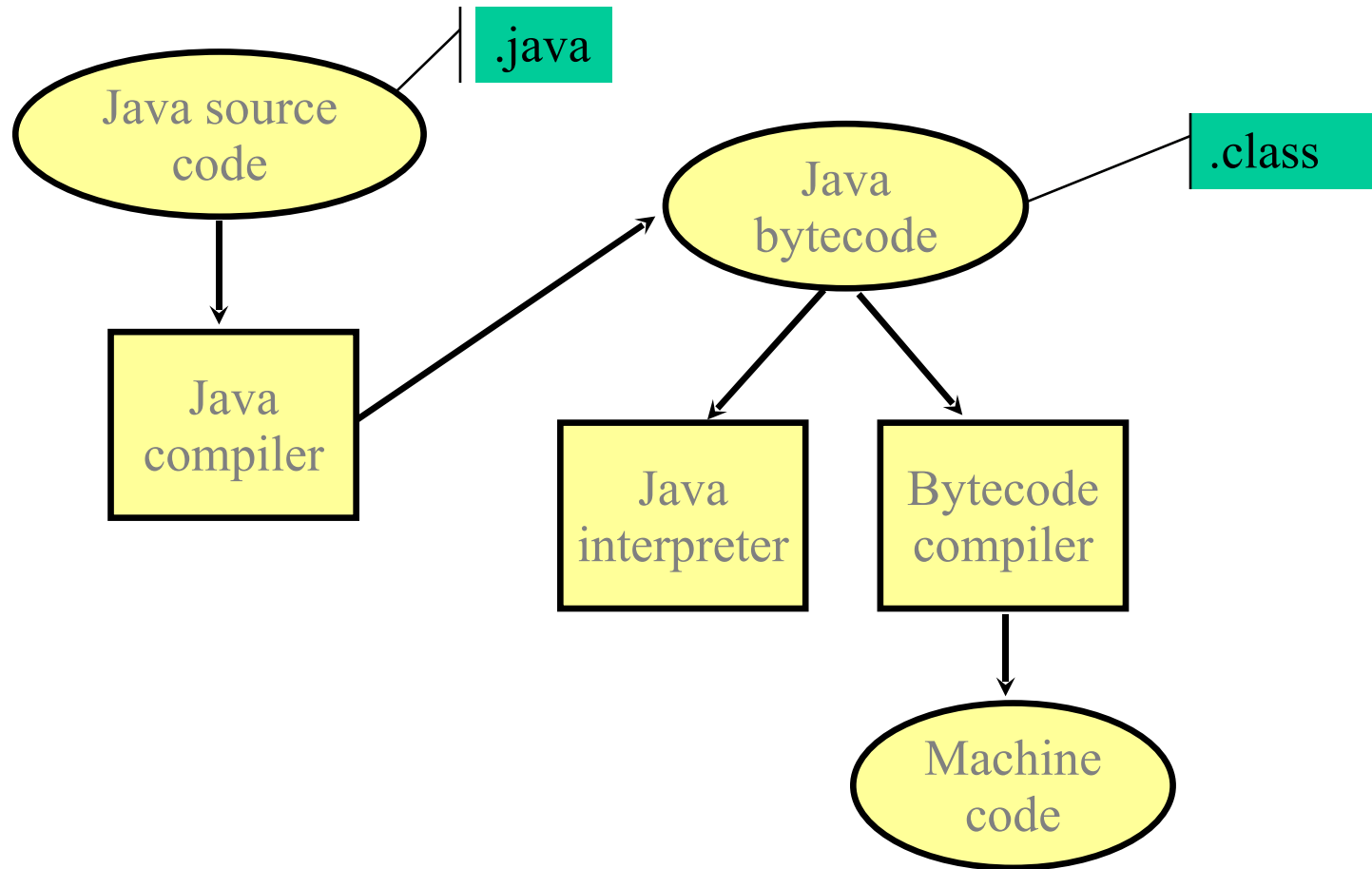
# Programming Languages

- A program must be translated into machine language before it can be executed on a particular type of CPU
- This can be accomplished in several ways
  - A *compiler* is a software tool which translates *source code* into a specific *target language*. Often, that target language is the machine language for a particular CPU type.
    - Input: files written in a high-level programming language
    - Output: executable binary file that can be processed by CPU
    - need compilers for each CPU type
  - Interpreter:
    - no output executable file
    - Instead the source code is translated and executed on-the-fly.
- The Java approach is somewhat different

# Java Translation and Execution

- The Java compiler translates Java source code into a special representation called *bytecode*
- Java bytecode is not the machine language for any traditional CPU
- Another software tool, called an *interpreter*, translates bytecode into machine language and executes it
- Therefore the Java compiler is not tied to any particular machine
- Java is considered to be *architecture-neutral*

# Java Translation and Execution



# More on System.out

- Two built-in commands to print on the screen:
  - `System.out.println(...stuff to print out...);`
    - A line-break is printed after `...stuff to print out...`
  - `System.out.print(...stuff to print out...);`
    - Only `...stuff to print out...` is printed
- Syntax:
  - `System.out.println(EXPRESSION);`
  - Where:
    - `EXPRESSION` = “anything between quotes”
    - `EXPRESSION` = variable
    - `EXPRESSION` = “anything ” + variable
    - `EXPRESSION` = “anything ” + variable + “ something more”
- Example:
  - `System.out.print(“x = ” + x);`

# Countdown.java

```
class Countdown
{
    public static void main(String args[])
    {
        System.out.print("Three... ");
        System.out.print("Two... ");
        System.out.print("One... ");
        System.out.print("Zero... ");

        System.out.println("Liftoff!");

        System.out.println("Houston, we have a problem!");
    }
}
```

What does this output?

# Countdown Result

Three... Two... One... Zero... Liftoff!  
Houston, we have a problem!

—



*Cursor ends up here*



MODIFIER TYPE IDENTIFIER = VALUE;

Where:

- MODIFIER      final, static                                (optional)
- TYPE          int, char, double, ...                     (mandatory)
- IDENTIFIER    a single word as defined previously                (mandatory)
- = VALUE       a constant matching the TYPE                        (optional)
- ;

## This is a partial definition

# Primitive Data

- There are exactly 8 primitive data types in Java
- Four of them represent integers:
  - `byte`, `short`, `int`, `long`
- Two of them represent floating point numbers:
  - `float`, `double`
- One of them represents characters:
  - `char`
- And one of them represents boolean values:
  - `boolean`

# Numeric Primitive Data

- The difference between the various numeric primitive types is their size and type, and therefore the values they can store:

<u>Type</u>	<u>Storage</u>	<u>Min Value</u>	<u>Max Value</u>
byte	8 bits	-128	127
short	16 bits	-32,768	32,767
int	32 bits	-2,147,483,648	2,147,483,647
long	64 bits	$< -9 \times 10^{18}$	$> 9 \times 10^{18}$
float	32 bits	+/- $3.4 \times 10^{38}$ with 7 significant digits	
double	64 bits	+/- $1.7 \times 10^{308}$ with 15 significant digits	

# Characters

- A `char` variable stores a single character from the *Unicode character set*
  - `char gender;`
  - `gender = 'F';`
- A *character set* is an ordered list of characters, and each character corresponds to a unique number
- The Unicode character set uses 16 bits (2 Bytes) per character, allowing for 65,536 unique characters
- It is an international character set, containing symbols and characters from many world languages
- Character literals are delimited by single quotes:  
`'a'`      `'X'`      `'7'`      `'$'`      `' , '`      `' \n '`

# Characters

- The *ASCII character set* is older and smaller than Unicode, but is still quite popular
- The ASCII characters are a subset of the Unicode character set, including:

uppercase letters

A, B, C, ...

lowercase letters

a, b, c, ...

punctuation

period, semi-colon, ...

digits

0, 1, 2, ...

special symbols

&, |, \, ...

control characters

carriage return, tab, ...

# Boolean

- A `boolean` value represents a true or false condition
- A `boolean` can also be used to represent any two states, such as a light bulb being on or off
- The reserved words `true` and `false` are the only valid values for a `boolean` type

```
boolean done = false;
```

```
...
```

```
done = true;
```

# More on Boolean Expression

- evaluates to either true or false
  - `if (denominator == 0)`  
`System.out.println("....`
- Boolean variable can be used where a boolean expression is expected
  - `if (done)`  
`System.out.println("you are done");`
  - `if (!done)`  
`System.out.println("not yet done");`
  - The `!` negates the value of a boolean expression
    - if a boolean expression `e` is true, then `!e` is false
    - if a boolean expression `e` is false, then `!e` is true

# Adding an arbitrary amount of numbers

```
import java.util.Scanner;
public class AddArbitraryAlternative
{
    public static void main (String [] args)
    {
        double input;
        double output = 0;
        boolean done = false;

        Scanner scan = new Scanner(System.in);

        // read in the values in a loop and incrementally perform calculation
        while (!done)
        {
            System.out.println("Enter number (0 indicates you want to exit):");
            input = scan.nextDouble();
            if (input == 0)
                done = true;
            else
                output = output + input;
        }
        System.out.println("The sum is: " + output);
    }
}
```



# Arithmetic Expressions

- An *expression* is a combination of operators and operands
- *Arithmetic expressions* compute numeric results and make use of the arithmetic operators:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder	%

- If either or both operands to an arithmetic operator are floating point, the result is floating point

# Operator Precedence

- Operators can be combined into complex expressions

```
result = total + count / max - offset;
```

- Operators have a well-defined precedence which determines the order in which they are evaluated
- Multiplication, division, and remainder are evaluated prior to addition, subtraction, and string concatenation
- Arithmetic operators with the same precedence are evaluated from left to right
- Parentheses can always be used to force the evaluation order

# Operator Precedence

- What is the order of evaluation in the following expressions?

$a + b + c + d + e$   
1 2 3 4

$a + b * c - d / e$   
3 1 4 2

$a / (b + c) - d \% e$   
2 1 4 3

$a / (b * (c + (d - e)))$   
4 3 2 1

# Assignment Revisited

- The assignment operator has a lower precedence than the arithmetic operators

First the expression on the right hand side of the = operator is evaluated

`answer = sum / 4 + MAX * lowest;`

4

1

3

2



Then the result is stored in the variable on the left hand side

# Assignment Revisited

- The right and left hand sides of an assignment statement can contain the same variable

First, one is added to the original value of `count`

```
count = count + 1;
```



Then the result is stored back into `count` (overwriting the original value)

# TempConvert.java

```
class TempConvert
{
    public static void main (String args[])
    {
        final int BASE = 32;
        final double CONVERSION_FACTOR = 9.0 / 5.0;
        double fahrenheitTemp;
        int celsiusTemp;

        Scanner scan = new Scanner(System.in);
        celsiusTemp = scan.nextInt();

        fahrenheitTemp = celsiusTemp * CONVERSION_FACTOR + BASE;

        System.out.println(Celsius Temp    = " + celsiusTemp);
        System.out.println(Fahrenheit Temp = " + FahrenheitTemp);
    }
}
```

# Syntactic Sugar: Increment/Decrement

- increment operator
  - unary operator: adds one to its only operand
    - `counter++;`
    - `++counter;`
    - prefix and postfix forms differ when used in larger expression
  - equivalent to
    - `counter = counter + 1;`
  - `total = counter++;`
    - assign value of `counter` to `total` and then increment value of `counter`
  - `total = ++counter;`
    - increment value of `counter` and then assign the new value of `counter` to `total`
- decrement operator
  - subtracts one from operand
  - `counter--;`

# Example

```
import java.util.Scanner;
public class AddArbitrary
{
    public static void main (String [] args)
    {
        double input;
        int iterations;
        double output = 0;
        int counter;

        Scanner scan = new Scanner(System.in);

        System.out.println("Indicate the amount of number:");
        iterations = scan.nextInt();
        // read in the values in a loop and incrementally perform calculation
        counter = 1;
        while (counter++ <= iterations)
        {
            // counter = counter + 1;
            // counter++;
            System.out.println("Enter number:");
            input = scan.nextDouble();
            output = output + input;
        }
        System.out.println("The sum is: " + output);
    }
}
```

– What are the advantages/disadvantages of the different choices?



# Syntactic Sugar: Assignment Operators

- Assignment and arithmetic operations
  - Example 1:
    - `total = total + 5;`
    - `total += 5;`
  - Example 2
    - `result = result * (count1 + count2);`
    - `result *= count1 + count2`
  - Evaluate the entire expression on the right-hand side first, then use the result as the right operand of the other operation

# Assignment Operators

- There are many assignment operators, including the following:

<u>Operator</u>	<u>Example</u>	<u>Equivalent To</u>
<b>+=</b>	<b>x += y</b>	<b>x = x + y</b>
<b>-=</b>	<b>x -= y</b>	<b>x = x - y</b>
<b>*=</b>	<b>x *= y</b>	<b>x = x * y</b>
<b>/=</b>	<b>x /= y</b>	<b>x = x / y</b>
<b>%=</b>	<b>x %= y</b>	<b>x = x % y</b>

# Example

```
import java.util.Scanner;
public class AddArbitrary
{
    public static void main (String [] args)
    {
        double input;
        int iterations;
        double output = 0;
        int counter;

        Scanner scan = new Scanner(System.in);

        System.out.println("Indicate the amount of number:");
        iterations = scan.nextInt();
        // read in the values in a loop and incrementally perform calculation
        counter = 1;
        while (counter <= iterations)
        {
            System.out.println("Enter number:");
            input = scan.nextDouble();
            output += input;
            counter++;
        }
        System.out.println("The sum is: " + output);
    }
}
```

# Data Conversion

- Sometimes it is convenient to convert data from one type to another
- For example, we may want to treat an integer as a floating point value during a computation
- Conversions must be handled carefully to avoid losing information
- *Widening conversions*
  - usually go from a data type with X Bytes to a data type with X or Y>X Bytes
  - usually no information lost
- *Narrowing conversions*
  - usually go from a data type with X Bytes to a data type with Y < X Bytes
  - can lose information (e.g. when converting from `int` to a `short`)

# Assignment Conversion

- In Java, data conversions can occur in three ways:
  - *assignment conversion*
  - *arithmetic promotion*
  - *casting*
- *Assignment conversion* occurs when a value of one type is assigned to a variable of another
  - Only widening conversions can happen via assignment
  - Recall: the value of a variable of type `int` can be assigned to a variable of type `double`

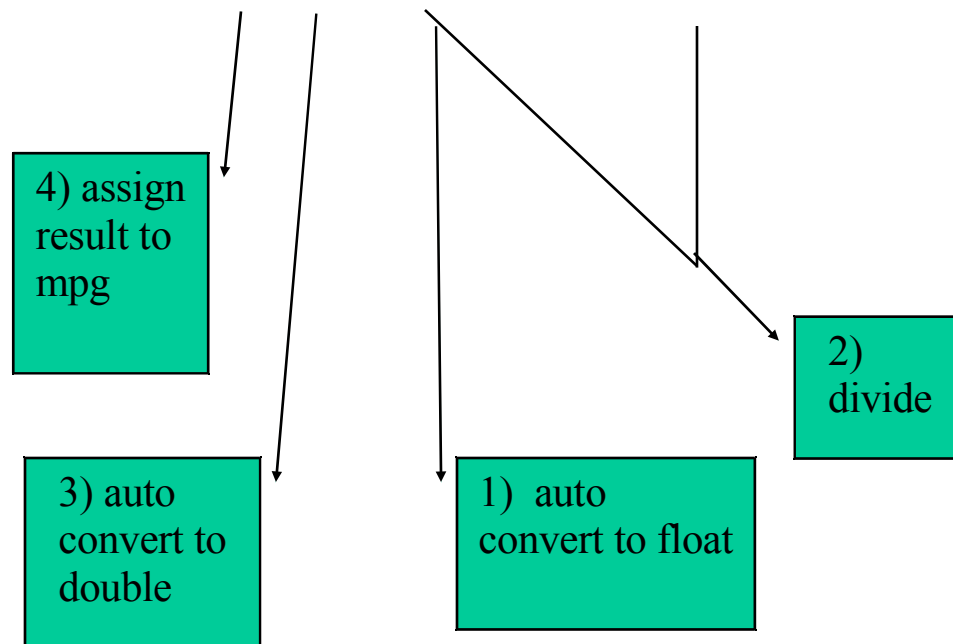
```
//money is double, dollars is int
money = dollars;
```

If `dollars` has value 25, then `money` has value 25.0 after assignment
  - If we attempt a narrowing conversion (assign the value of a variable of type `double` to a variable of type `int`), the compiler issues an error message

# Arithmetic Promotion

- Arithmetic promotion* happens automatically when operators in expressions convert their operands

```
//mpg is a double, gallons is a float, miles is an int  
mpg = miles / gallons;
```



# Casting

- Most general, but trusts that you to understand the effect
- Both widening and narrowing conversions can be accomplished by explicitly casting a value
- To cast, the type is put in parentheses in front of the value being converted
- floating point to integer cast
  - truncates fractional part

```
//money is double, dollars is int
dollars = (int) money;
```

If `money` has value 25.8, then `dollars` has value 25 after assignment
  - cast does not change the value of the casted variable
    - `money` still has 25.8 after the assignment

# Casting

- `total` and `count` are integers, result is a float
- `result = total / count;`
  - Integer division and assignment conversion
  - e.g., if `total` is 10 and `count` is 4, then `result` is assigned 2.0
- `result = (float) total / count;`
  - cast returns floating point version of value of `total`
  - arithmetic conversion now treats `count` as floating point
  - division is floating point division
  - e.g., if `total` is 10 and `count` is 4, then `result` is assigned 2.5
  - note 1: cast has higher precedence than `/`, thus cast operates on value of `total`, not on the result of division
  - note 2: cast does not change the value in `total` for the rest of the program.



# Conversion Examples

## EXPRESSION

```
double x = 5.9;  
int y = (int) x;
```

## RESULT

y has 5

```
int a = 5;  
float b = 7.3;  
double c = 10.03;  
c = b + a;
```

c has 12.3

```
int a = 2, b = 5;  
double c = 22;  
c = a / b;
```

c has 0