INTRODUCTION TO JAVA

PROGRAM

Programing Language Level

- There are four programming language levels:
  - Machine languages
    - 10010000 11110000
  - Assembly languages
    - move ax, bx
    - add ax, bx, cx
  - High-level languages
    - Java, C, C++, etc.
  - Fourth-generation languages
    - MATHLAB, or SQL

Low-Level Languages

Program Development

- The mechanics of developing a program include several activities
  - writing the program in a specific programming language (such as Java)
  - translating the program into a form that the computer can execute
  - investigating and fixing various types of errors that can occur
- Software tools can be used to help with all parts of this process

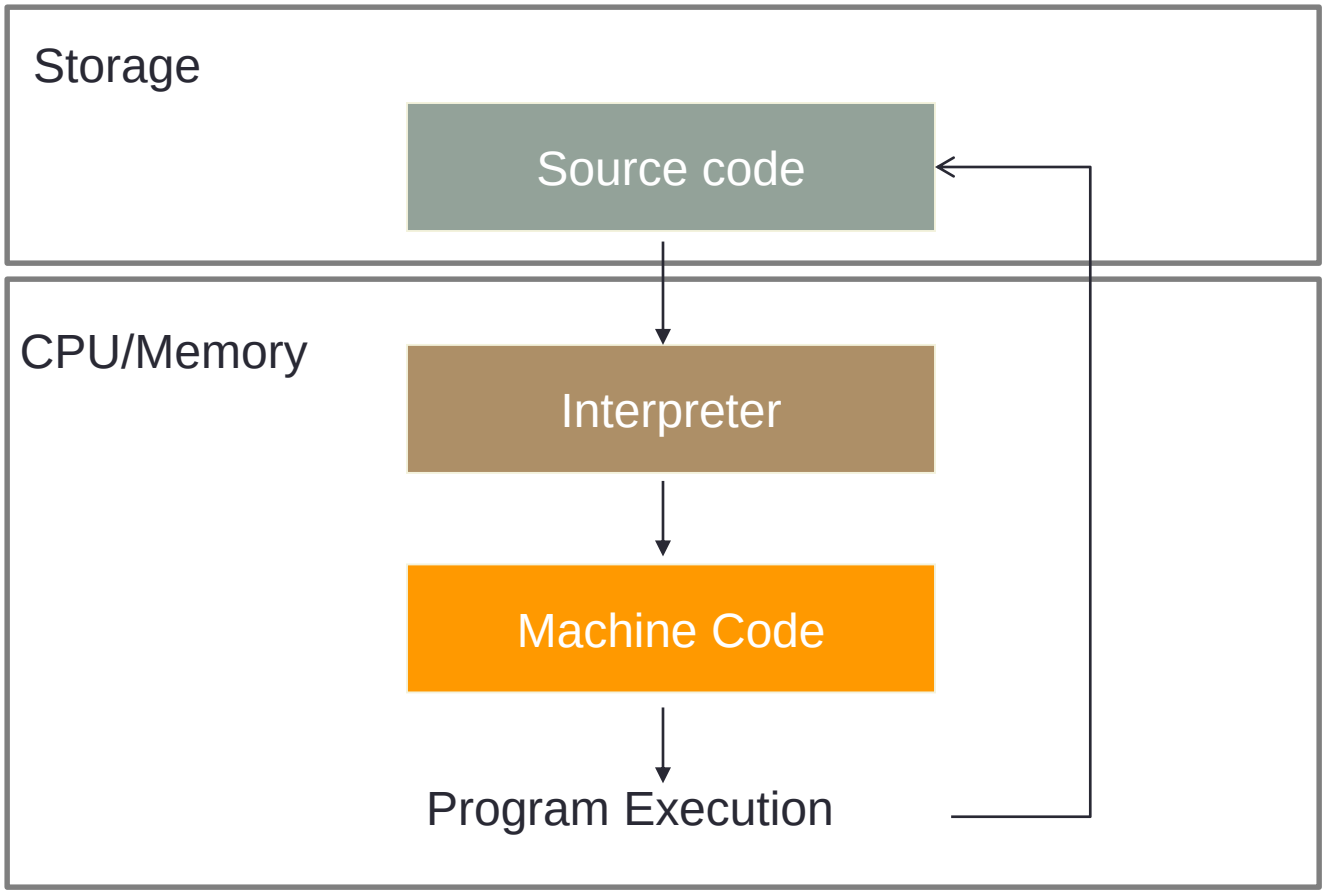
Program Translation

- A program written in a high level language is called *a **"source program"*** or ***"source code".***

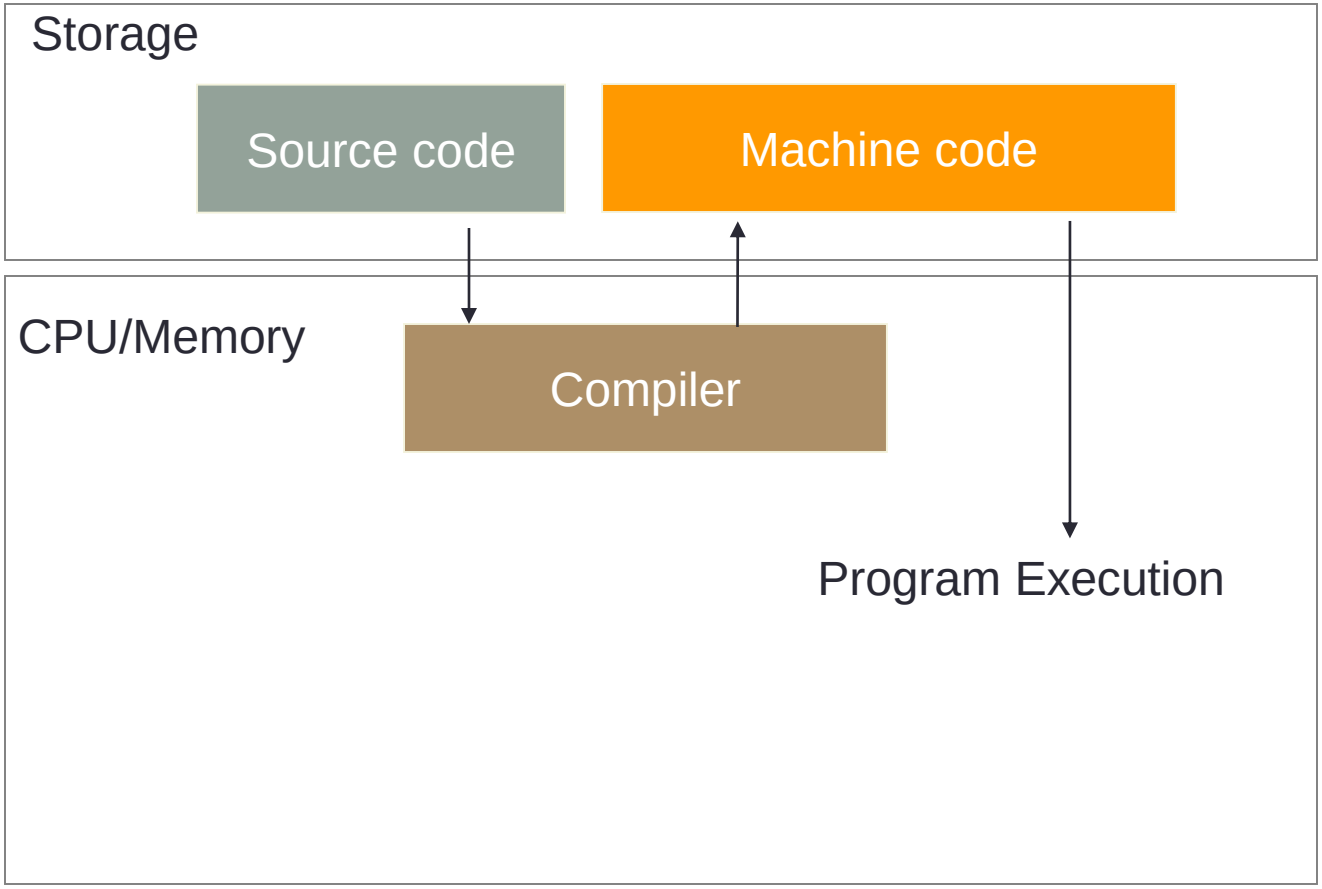
- A computer can not understand a source program

- We used the translator to translate the source program into a machine-language
  - Interpretation : Basic, Prolog, Smalltalk,…
  - Compilation: Pascal, C, Fortran,…

Interpretation



Compilation



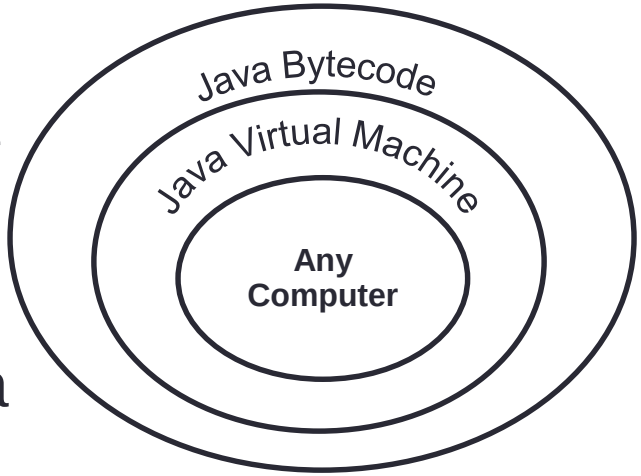
JAVA

An Introduction

Java

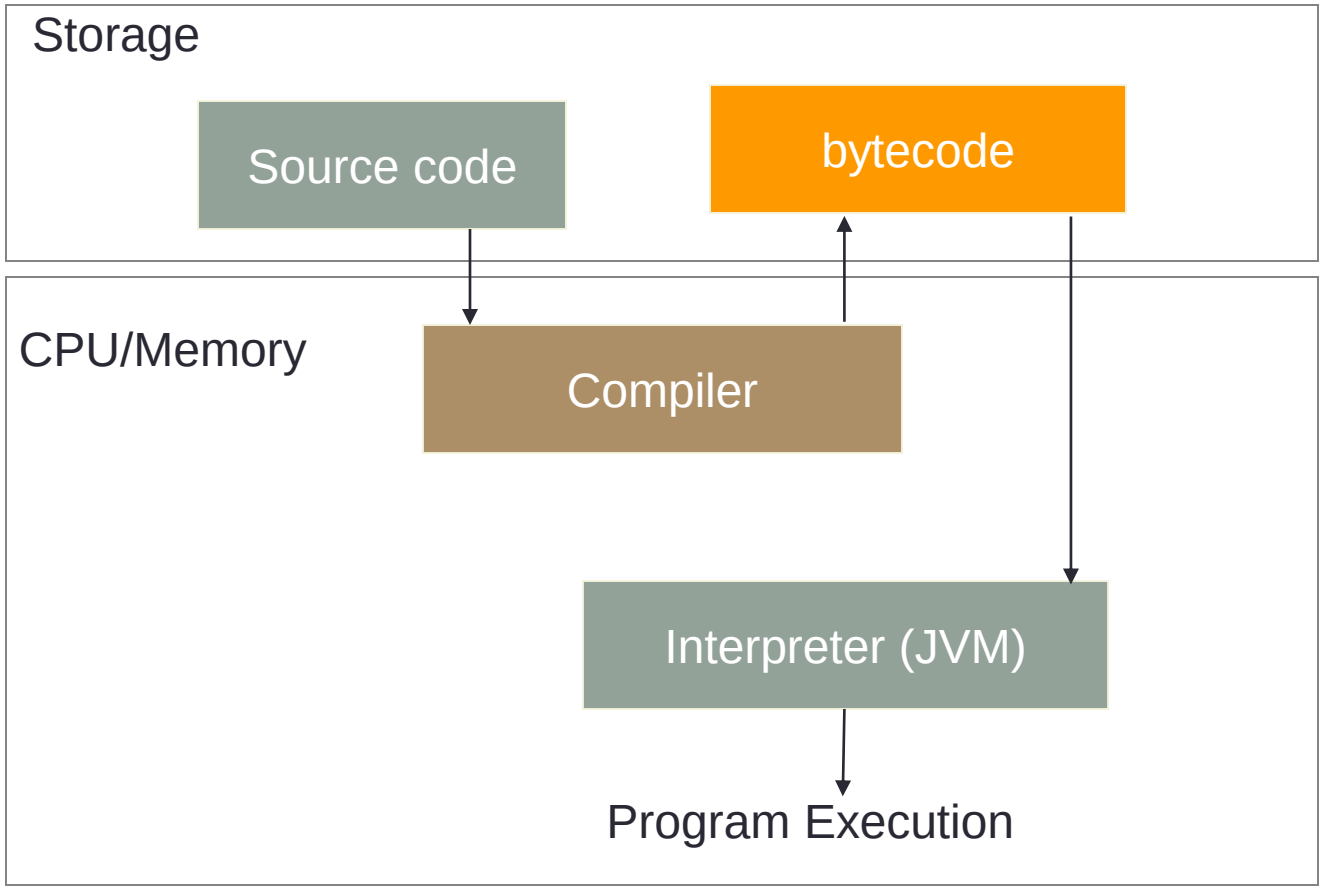
- A high level programming language created by Sun Microsystems, Inc.
- It was introduced in 1995 and it's popularity has grown quickly since
- It employs a set of rules that dictate how the words and symbols can be put together to form valid program statements
- Java was designed to run on any platform.

Java

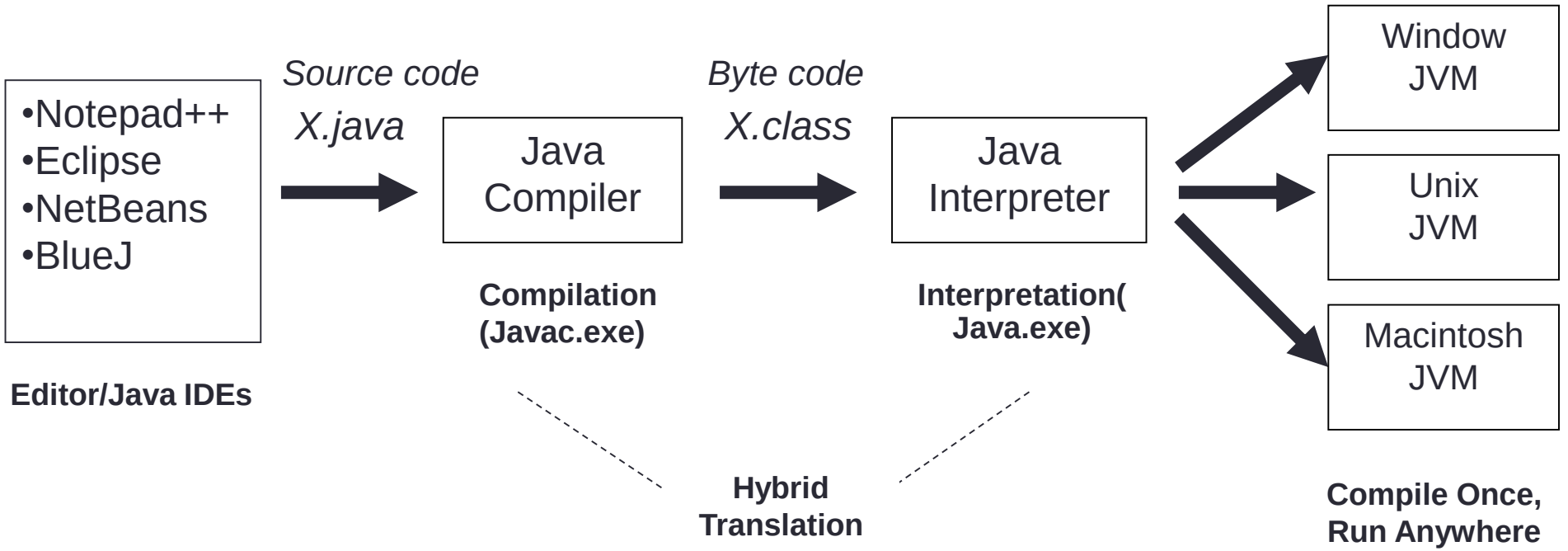


- Write the program once and compile the source program into a *byte code* with .class extension
- Java is a high-level language while Java byte code is a low-level language.
- The byte code is similar to machine instructions but is architecture neutral and can run on any platform that has a Java Virtual Machine (JVM)
- Rather than a physical machine, the virtual machine is a program that interprets Java byte code.
- Java byte code can run on a variety of hardware platforms and operating systems

Hybrid Translation



Java Environment



Compile Once, Run Anywhere

- The Java interpreter is part of the JVM
- The Java Virtual Machine (JVM) is a piece of software (not a piece of hardware) that interprets ***Java byte code*** into machine code.
- Once, you compile a program into byte code, it can be run on any machine with the JVM installed.
- The program never needs to be recompiled in order to run on different machine.

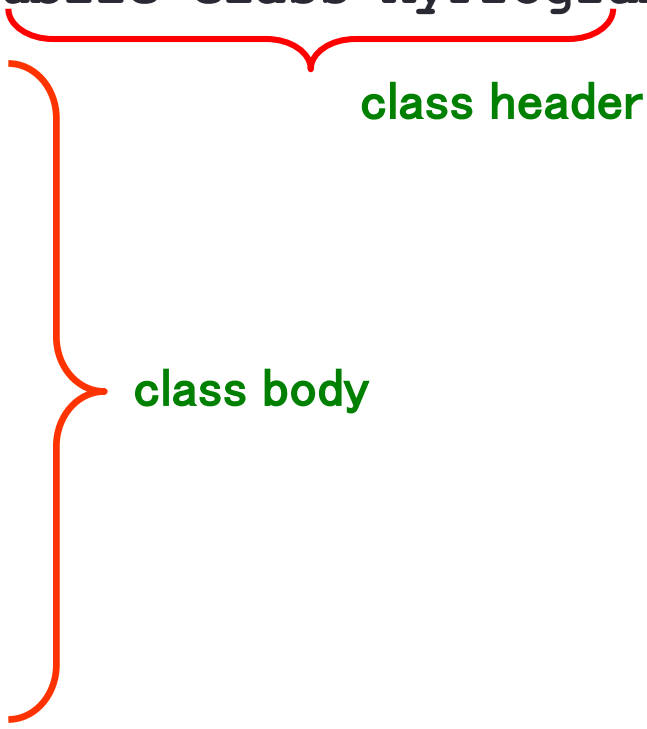
Java Program Structures

- In the Java programming language:
  - A program is made up of one or more *classes*
  - A class contains one or more *methods*
  - A method contains one or more *statements*
- In order to run a class, the class must contain a method named “main”
- The main method is the starting point of every program

Java Program Structure

```
// comments can be placed almost anywhere
// comments about the class
public class MyProgram
{


}
```



Java Program Structure

```
//  comments about the class
public class MyProgram {

    // comments about the method
    public static void main (String[] args)
    {
        System.out.println("Java Program");
    }
}
```

method header

method body

Java Program Layout

```
public class Hello{
    public static void main(String[] args){
        System.out.println("Hello World!");
    }
}
```

Good Layout

```
class
{
   public static void
   main(String[] args
      )
   {
         System.out.println
   ("Hello
    World!");
   }
     }
```

Bad Layout

Set PATH Variable

- If you want to be able to conveniently run the Java 2 SDK executables from any directory
  - javac.exe
  - java.exe
  - javadoc.exe
  - etc.
- You have to set PATH variable

Create A Java Program

- Write and save a file "Hello.java" that defines class "Hello"

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

- If your program use a "public" keyword in front of class name, you must give a filename in the same as your class name
- File and class name are case sensitive and must match exactly

Compile

- Compile Java Program

```
javac  <filename.java>
javac Hello.java
```

- This step creates a file extension "`Hello.class`"

Run

- Run Java Program

```
java  <classname>
java Hello
```

- Output

```
Hello World!
```

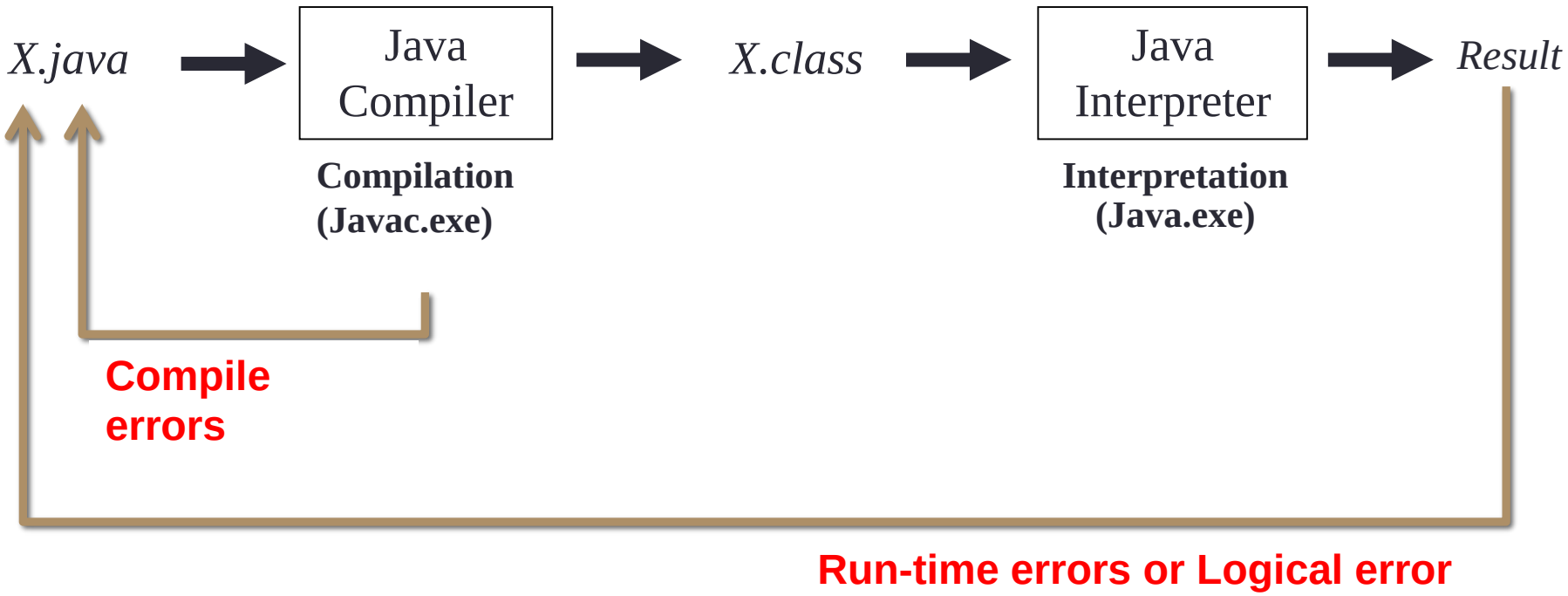
Syntax and Semantics

- The syntax rules of a language define how we can put together symbols, reserved words, and identifiers to make a valid program
- The semantics of a program statement define what that statement means (its purpose or role in a program)
- A program that is syntactically correct is not necessarily logically (semantically) correct
- A program will always do what we tell it to do, not what we meant to tell it to do

Errors

- A program can have three types of errors
  - **Compile-time errors:** The compiler will find syntax errors and other basic problems. If compile-time errors exist, a byte code version of the program is not created
  - **Run-time errors:** A problem can occur during program execution, such as trying to divide by zero, which causes a program to terminate abnormally
  - **Logical errors:** A program may run, but produce incorrect results, perhaps using an incorrect formula

Java Program Development Process



Lexical Structures

- White spaces
- Comments
  - /* text */ /** text */
  - // text
- Tokens
  - Identifiers
  - Keywords / Reserved words
  - Literals
  - Separators
  - Operators

White Spaces

- Spaces, blank lines, and tabs are called white space
- White space is used to separate words and symbols in a program
- Extra white space is ignored
- A valid Java program can be formatted many ways
- Programs should be formatted to enhance readability, using consistent indentation

Java Program

```
public class TestNoFormat1{public static void main(String[]args){
System.out.println("Java");
System.out.println("Compile Once, Run Anywhere");}}
```

```
 public        class
      TestNoFormat2    {
                  public static void  main        (
String []                    args )
   {
   System.out.println (
"Java")
   ;          System.out.println
               (
         "Compile Once, Run Anywhere"
        )
   ;
}     }
```

Both are Bad Layout

Java Program

```
public class TestFormat{
  public static void main(String[]args){
        System.out.println("Java");
        System.out.println("Compile Once, Run Anywhere");
  }
}
```

Good Layout

Comments

- Comments in a program are called *inline documentation*
- Good Comments help programmers to explain the purpose of the program and describe processing steps
- They are not programming statements and thus are ignored by the compiler
- Java comments can take three forms:

```
// this comment runs to the end of the line

/*  this comment runs to the terminating
    symbol, even across line breaks       */

/** this is a javadoc comment   */
```

Identifiers (1)

- Identifiers are the words a programmer uses in a program
- An identifier can be made up of letters, digits, the underscore character (_), and the dollar sign
- Identifiers cannot begin with a digit
- Identifiers cannot be the same as keywords/reserved words
- Java is case sensitive - Total, total, and TOTAL are different identifiers

Identifiers (2)

- By convention, programmers use different case styles for different types of identifiers, such as
  - title case for class names – `BankAccount, Student`
  - upper case for constants – `MAXIMUM`, `TAX`
  - camelCase for variables, methods, attributes, and objects – `studentName,isFull`
- We choose identifiers ourselves when writing a program (such as HelloWorld)
- Sometimes we are using another programmer's code, so we use the identifiers that he or she chose (such as println)

Keywords/Reserved Words

- The Java reserved words:

```
abstract    else        interface   switch
assert      enum        long        synchronized
boolean     extends     native      this
break       false       new         throw
byte        final       null        throws
case        finally     package     transient
catch       float       private     true
char        for         protected   try
class       goto        public      void
const       if          return      volatile
continue    implements  short       while
default     import      static
do          instanceof  strictfp
double      int         super
```

Primitive Data Types

There are eight 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`
- One of them represents boolean values:
  - `boolean`

Literals

- *Literal* is a constant value that appears in a program

- **integer**
  - 20 ***(default int)*** 20L 20l
  - 010 (octal) 0Xf0 (hex) 0xffL

- **floating point**
  - 3.14 ***(default double)*** 3.1E12 2.0e-2
  - 2.0F 2.5f

- **character**
  - 'A' '\n' '\udddd' (hex - \udddd)
  - '\\' ' \101' (octal - \ddd)

- **boolean**
  - true
  - false

Separators

( ) Parentheses

- contain lists of parameters in method definition an invocation
- define precedence in expressions
- contain expressions in control statements
- surround cast types

{ } Braces

- define a block of code for classes, methods, and local scopes
- contain the values of automatically initialized arrays

[ ] Brackets

- declare array types
- dereference array values

Separators

; Semicolon

- terminates statements

, Comma

- separates consecutive identifiers in a variable declaration
- use to chain statements together inside a for statement

. Period

- separate package names from subpackages and classes
- use to separate an attribute or method from a reference variable

Operators

- **Arithmetic**

  + - * / %

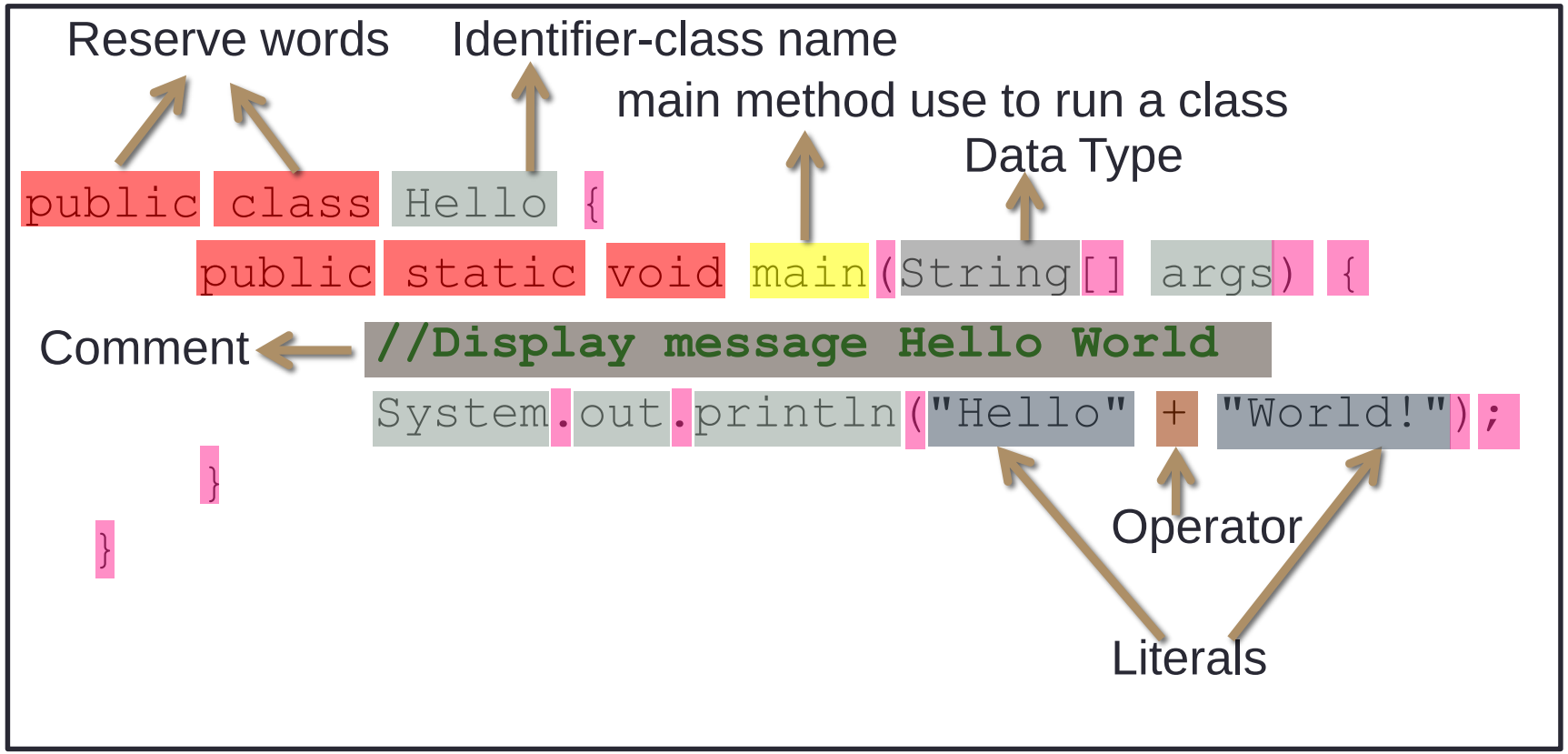
- **Relational**

  < <= > >= == !=

- **Logical**

  ! && (condition) & || (condition) |

Program Structures



reserve words
main method
literals
separators
identifiers
comment
operator
data type

Primitive Data Type

Type	Size	Default	Value Ranges	Contains
boolean	16 bits	false		true or false
byte	8 bits	0	-128 to 127	Signed integer
char	16 bits	\u0000		Unicode character
short	16 bits	0	-32,768 to 32,767	Signed integer
int	32 bits	0	-2,147,483,648 to 2,147,483,647	Signed integer
long	64 bits	0	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed integer
float	32 bits	0.0	Approximately -3.4E+38 to 3.4E+38 with 7 significant digits	IEEE754 floating point
double	64 bits	0.0	Approximately -1.7E+308 to 1.7E+308 with 15 significant digits	IEEE754 floating point

boolean

- A `boolean` value represents a true or false condition
- The reserved words `true` and `false` are the only valid values for a boolean type

```
boolean done = false;   ✓
boolean done =“true”;   ✘
boolean done  = True;   ✘
```

- A `boolean` variable can also be used to represent any two states, such as a light bulb being on or off

Integer

- Four of them represent integers:
  - `byte`, `short`, `int(default)`, `long`
- Example declarations:
  - `byte b=120;`
  - `short s=22112;`
  - `int i=13000;`
  - `long l=2124230;`

Floating Point

- Two of them represent floating point numbers:
  - `float`, `double(default)`
- Example declarations:
  - `float b= 123.23f;`
  - `double d=123.23;`

Character

- A char variable stores a single character by using single quotes:
- `'A'  'b'  '\n'  '$'  ','`

- Example declarations:
  - `char grade = 'A';`
  - `char symbol = ';';`

Character Sets

- A *character set* is an ordered list of characters, with each character corresponding to a unique number
- A `char` variable in Java can store any character from the *Unicode character set* (16-bits)
- The Unicode character set allows for 65,536 unique characters
- It is an international character set, containing symbols and characters from many world languages

ASCII Character

- The ASCII character set is older and smaller than Unicode (7-bits), 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,…
  - **Digits** - 0, 1, 2,…
  - **Symbols** - &, |, \,:, ;…
  - **Control characters** - carriage return, tab,…

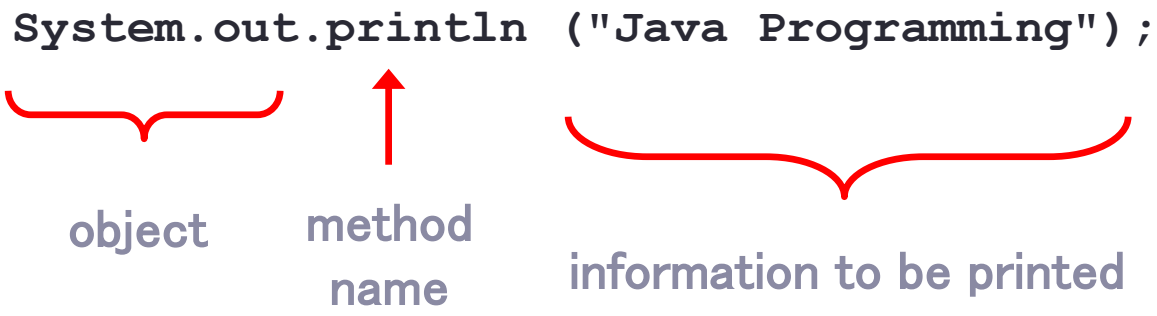
String

- A string of characters can be represented as a *string literal* by putting double quotes around the text.
  - Note the distinction between a primitive character variable, which holds only one character, and a String object, which can hold multiple characters
- Examples:

```
"This is a string example."
"SIT, KMUTT"
"A"
"Java\nProgramming"
```

System.out

- The `System.out` object send output to a destination (the monitor screen)



The print and println Method

- The `print` method is similar to the `println` method, except that it does not advance to the next line
- Therefore anything printed after a `print` statement will appear on the same line

```
System.out.print ("Three... ");
System.out.print ("Two... ");
System.out.print ("One... ");
System.out.print ("Zero... ");

System.out.println ("Liftoff!"); // appears on
first output line
```

String Concatenation (1)

- The string concatenation operator (+) is used to append one string to the end of another
- `" Java " + " Programming“`
- It can also be used to append a number to a string
- A string literal cannot be broken across two lines in a program

```
System.out.println ("learning" + 60 + " hours per course");
```

String Concatenation (2)

- The + operator is also used for arithmetic addition
- The function that it performs depends on the type of the information on which it operates
- If both operands are strings, or if one is a string and one is a number, it performs string concatenation
- If both operands are numeric, it adds them
- The + operator is evaluated left to right, but parentheses can be used to force the order

```
System.out.println ("24 and 45 concatenated: " + 24 + 45);
System.out.println ("24 and 45 added: " + (24 + 45));
```

Escape Sequences

- What If we wanted to print the quote character?
- The following line would confuse the compiler because it would interpret the second quote as the end of the string

- `System.out.println ("I said "Hello" to you.");`

- An escape sequence is a series of characters that represents a special character
- An escape sequence begins with a backslash character (\)

- `System.out.println ("I said \"Hello\" \nto you.");`

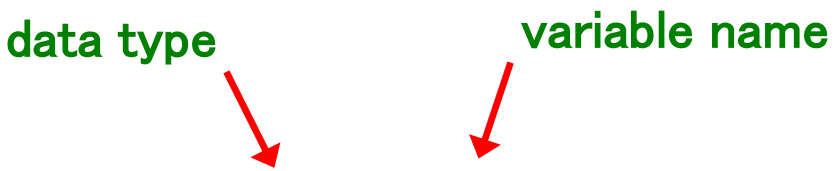
Escape Sequences

- Some Java escape sequences:

Escape Sequence	Name
`\b`	backspace
`\t`	tab
`\n`	newline
`\r`	carriage return
`\"`	double quote
`\'`	single quote
`\\`	backslash

Variables

- A variable is a name for a location in memory
- A variable must be declared by specifying the variable's name and the type of information that it will hold
- You must declare variable before using




```
int total;

int count, temp, result;
```

Multiple variables can be created in one declaration

Variable Initializations

- A variable can be given an initial value in the declaration

```
int sum = 0;
int base = 32, max = 149;
```

```
int point = 40;
System.out.println ("your score is " + point  + " points.");
```

Assignments

- An assignment statement changes the value of a variable
- The assignment operator is the = sign

```
total = 55;
```

- The expression on the right is evaluated and the result is stored in the variable on the left
- The value that was in total is overwritten
- You can only assign a value to a variable that is consistent with the variable's declared type

Named Constants (1)

- A constant is an identifier that is similar to a variable except that it holds the same value during its entire existence
- As the name implies, it is constant, not variable
- The compiler will issue an error if you try to change the value of a constant
- In Java, we use the `final` modifier to declare a constant

```
final int MIN_HEIGHT = 69;
```

Named Constants (2)

- Constants are useful for three important reasons
- First, they give meaning to otherwise unclear literal values
  - For example, `MAX_LOAD` means more than the literal 250
- Second, they facilitate program maintenance
  - If a constant is used in multiple places, its value need only be updated in one place
- Third, they formally establish that a value should not change, avoiding inadvertent errors by other programmers

Expressions

- An *expression* is a combination of one or more 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 used by an arithmetic operator are floating point, then the result is a floating point

Division and Remainder

- If both operands to the division operator (/) are integers, the result is an integer (the fractional part is discarded)

```
14 / 3      equals      4

8 / 12      equals      0
```

- The remainder operator (%) returns the remainder after dividing the second operand into the first

```
14 % 3      equals      2

8 % 12      equals      8
```

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, but parentheses can be used to force the evaluation order

Precedence Level	Operator	Operation	Associativity
1	Method()	Method Invocation	L to R
	.	Object Member Reference	
	[]	Array Indexing	
	++, --	**Post-Increment, Post-Decrement**	
2	**++, --**	**Pre-Increment, Pre-Decrement**	R to L
	+, -	**Unary Plus , Unary Minus**	
	!	Logical Not	
3	new	Object Instantiation	R to L
	(<type>)	**Cast (Type Conversion)**	
4	***, /, %**	**Arithmetic Operators**	L to R
5	**+, -**	**Arithmetic Operators**	L to R
	+	**String Concatenation**	
6	<, <=, >, >=	Relational Operators	L to R
7	==, !=	Equality Operators	L to R
8	&	Logical AND	L to R
9	|	Logical OR	L to R
10	&&	Short-Circuit AND	L to R
11	||	Short-Circuit OR	L to R
12	?:	Conditional Operator	R to L
13	**=, *=, /=, %=, +=, -=**	**Assignment with operation**	R to L

Order of Evaluation

- When the Java interpreter evaluates an expression, it performs the various operations in an order specified by
  - the parentheses in the expression
  - the precedence of the operators
  - the associativity of the operators.
- Before any operation is performed, however, the interpreter first evaluates the operands of the operator.
- The interpreter always evaluates operands in order from left to right.
- This matters if any of the operands are expressions that contain side effects

Order of Evaluation Example

- For example,
- int a = 2;
- int result = ++a + ++a * ++a;
- Although the multiplication is performed before the addition, the operands of the + operator are evaluated first.
- Thus, the expression evaluates to 3+4*5, or 23.

Operator Precedence

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

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

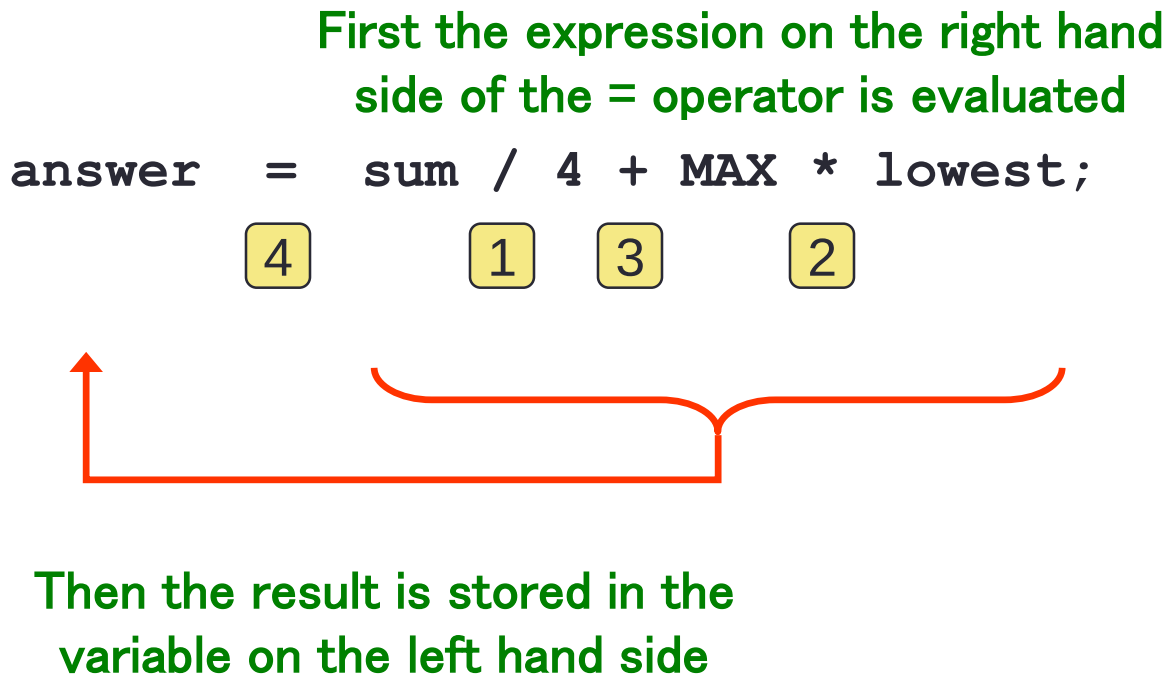
```
a + b * c - d / e
  3   1   4   2
```

```
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



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)

Increment and Decrement

- The increment and decrement operators use only one operand
- The *increment operator* (++) adds one to its operand
- The *decrement operator* (−−) subtracts one from its operand
- The statement

```
count++;
```

  is functionally equivalent to

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

Increment and Decrement

- The increment and decrement operators can be applied in *postfix form*:

```
count++
```

- or *prefix form*:

```
++count
```

- When used as part of a larger expression, the two forms can have different effects

- Because of their subtleties, the increment and decrement operators should be used with care

Increment and Decrement Example

```
int num1=1;
int num2=1;
num1++;
++num2;
System.out.println(num1); //num1 = 2
System.out.println(num2); //num2 = 2
```

Increment and Decrement Example

```
int num1=1;
int num2=1;

int num3=++num1 + 2;
System.out.println(num3); //num3 = 4
System.out.println(num1); //num1 = 2

int num4=num2++ + 2;
System.out.println(num4); //num4 = 3
System.out.println(num2);//num2 = 2
```

Assignment Operators

- Often we perform an operation on a variable, and then store the result back into that variable
- Java provides *assignment operators* to simplify that process
- For example, the statement

```
num += count;
```

is equivalent to

```
num = num + count;
```

Assignment Operators

- There are many assignment operators in Java, including the following:

Operator	Example	Equivalent To
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`%=`	`x %= y`	`x = x % y`

Assignment Operators

- The right hand side of an assignment operator can be a complex expression
- The entire right-hand expression is evaluated first, then the result is combined with the original variable
- Therefore

```
result /= (total-MIN) % num;
```

is equivalent to

```
result = result / ((total-MIN) % num);
```

Assignment Operator Example

```
int result=50, num=4, total=11;
final int MIN=4;
result /= (total-MIN) % num;
System.out.println(result); //result = 16
```

Assignment Operators

- The behavior of some assignment operators depends on the types of the operands
- If the operands to the += operator are strings, the assignment operator performs string concatenation
- The behavior of an assignment operator (+=) is always consistent with the behavior of the corresponding operator (+)

Type Conversions

- Java is strongly typed checking, that means each data value is associated with a particular type.
- It is sometimes helpful or necessary to convert a data value of one type to another type, but we must be careful that we don't lose important information in the process
- There is strict enforcement of type rules
  - **Widening Conversions**
  - **Narrowing Conversions**
- Note that Widening a type can be performed automatically without explicit casting. Narrowing a type must be performed explicitly

Type Conversions (2)

- **Widening Conversions** – are the safest because they usually do not lose information. They convert from one data type to another type that uses greater amount of space to store the value (such as a `short` to an `int`)
- **Narrowing Conversions** – are more likely to lose information than widening conversions are. They often convert from one type to a type that use less space to store a value, and therefore some of the information may be lost (such as an `int` to a `short`)
- These conversions do not change the type of a variable or the value that's stored in it – they only convert a value as part of a computation

Java Widening Conversions

Type	Convert to
byte	short, int, long, float, double
short	int, long, float, or double
char	int, long, float, or double
int	long, float, or double
long	float or double
float	double

Note that When converting int or long to float or from long to double, some of the significant digits may be lost precision

Java Narrowing Conversions

Type	Convert to
byte	char
short	byte or char
char	byte or short
int	byte, short, or char
long	byte, short, char, or int
float	byte, short, char, int, or long
double	byte, short, char, int, long, or float

Note that boolean values cannot be converted to any other primitive type and vice versa

Data Conversions

- In Java, data conversions can occur in three ways:
  - Assignment conversion
  - Numeric promotion
  - Casting conversion

Assignment Conversions

- *Assignment conversion* occurs when a value of one type is assigned to a variable of another
- If `money` is a `float` variable and `dollars` is an `int` variable, the following assignment converts the value in `dollars` to a `float`

```
money = dollars
```

- Only widening conversions can happen via assignment
- Note that the value or type of `dollars` did not change

Numeric Promotions

- *Numeric Promotion* happens automatically when operators in expressions convert their operands
- For example, if `sum` is a `float` and `count` is an `int`, the value of `count` is converted to a floating point value to perform the following calculation:

```
result = sum / count;
```

Casting Conversions

- *Casting conversion* is the most powerful, and dangerous, technique for conversion
- 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
- For example, if `total` and `count` are integers, but we want a floating point result when dividing them, we can cast `total`:

```
result = (float) total / count;
```