

Chapter Two

Java Language Basics

Java Language Basics

- A program in Java is a set of class declarations
- An object is an instance of a class
- A Java program can be seen as a collection of objects satisfying certain requirements and interacting through functionalities made public
- The name of the class coincides with that of the file
- The structure of Java program is:

[Documentation]

[package statement]

[import statement(s)]

[interface statement]

[class definition]

main method class definition

- In the Java language, the simplest form of a class definition is

```
class name {  
    . . .  
}
```

- **Class name** must be the same as the file name where the class lives
- A program can contain one or more class definitions but **only one public class** definition
- The program can be created in any text editor
- If a file contains **multiple classes**, the file name must be the **class name** of the class that contains the **main method**

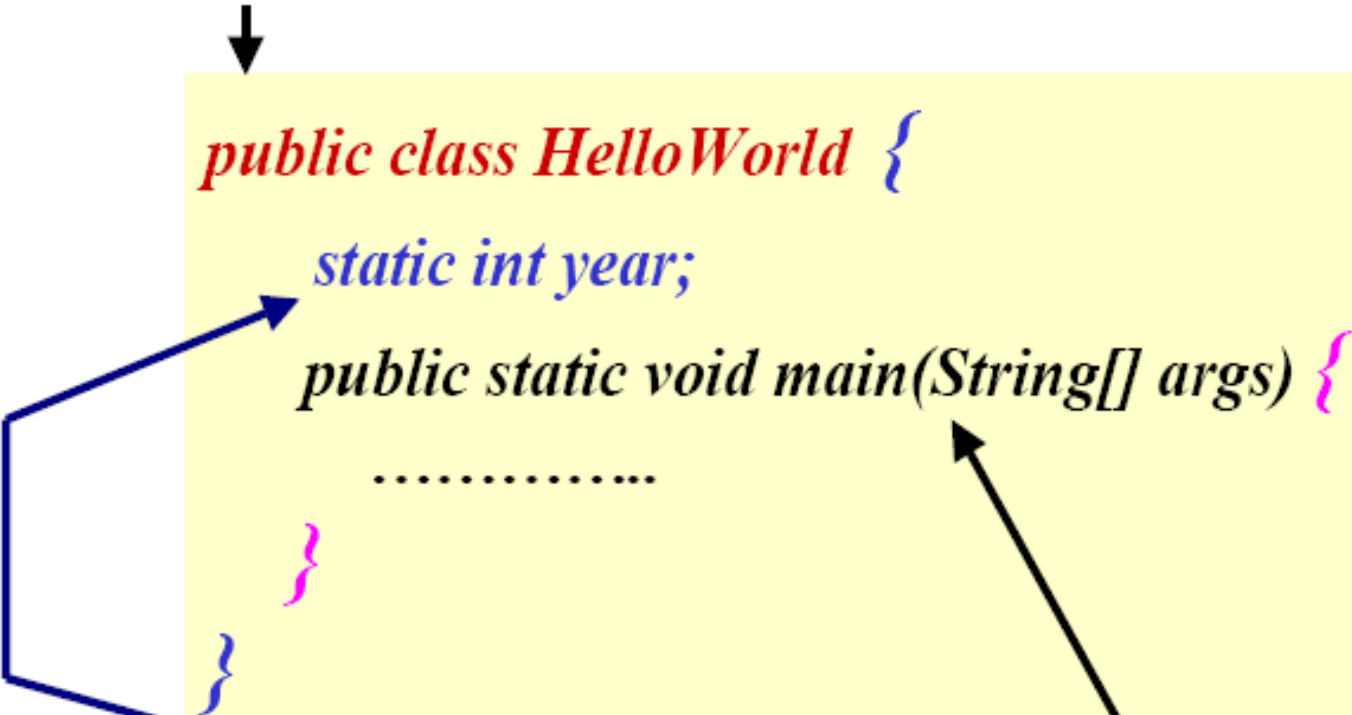
Your First Java Program

```
// your first java application
import java.lang.*;
class HelloWorld {
    public static void main(String[] args){
        System.out.println("Hello World!");
    }
}
```

- Save this file as **HelloWorld.java** (watch capitalization)

- Classes, methods and related statements are enclosed between { ... }

All Java programs must have at least one of this



```
public class HelloWorld {  
    static int year;  
    public static void main(String[] args) {  
        .....  
    }  
}
```

The diagram shows a yellow rectangular box containing the above code snippet. A black arrow points from the text 'All Java programs must have at least one of this' down to the box. A blue arrow points from the text 'Classes contain variables and methods' up to the opening curly brace of the class. Another blue arrow points from the same text up to the 'static int year;' line. A black arrow points from the same text up to the opening curly brace of the main method.

Classes contain variables and methods

main - A Special Method

- The main method is where a Java program always starts when you run a class file with the **java** command
- The **main** method has a strict signature which must be followed:

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

- **public** (access modifier) makes the item visible from outside the class
- **static** indicates that the main() method is a class method not an instance method.
 - It allows **main()** to be called without instantiating a particular instance of the class

```
class SayHi {  
    public static void main(String[] args) {  
        System.out.println("Hi, " + args[0]);  
    }  
}
```

- When java Program **arg1 arg2 ... argN** is typed on the command line,
 - anything after the name of the class file is automatically entered into the **args** array:
java SayHi Aman
- In this example **args[0]** will contain the String “**Aman**”, and the output of the program will be “**Hi, Aman**”.

Compiling and Running Your First Program

- Open the command prompt in Windows.
 - Change the path to the directory where your files is stored
- To compile the program, type the following at the command prompt:
javac HelloWorld.java
- You have now created your first compiled Java program named **HelloWorld.class**
- To run your first program, type the following at the command prompt:
java HelloWorld
- Although the file name includes the **.class** extension , this part of the name must be left off when running the program with the Java interpreter.

- **Lexical Components of Java**

- Java program = Comments + Token(s) + White space

- **Comments**

- Java supports three types of comment delimiters

1. The `/*` and `*/` - enclose text that is to be treated as a comment by the compiler
2. `//` - indicate that the rest of the line is to be treated as a comment by the Java compiler
3. the `/**` and `*/` - the text is also part of the automatic class documentation that can be generated using JavaDoc.

- **Atomic elements of java Tokens**
 - The smaller individual units inside a program that the compiler recognizes when building up the program
 - Five types of Tokens in Java:
 - Reserved keywords
 - Identifiers
 - Literals
 - operators
 - separators

Reserved keywords

- words with special meaning to the compiler
- The following keywords are reserved in the Java language. They can *never* be used as identifiers:

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

- Some are not used but are reserved for further use. (Eg.: const, goto)

Identifiers

- Programmer given names
- Identify classes, methods, variables, objects, packages.
- **Identifier Rules:**
 - Must begin with
 - Letters , Underscore characters (_) or Any currency symbol
 - Remaining characters
 - Letters
 - Digits
 - As long as you like!! (until you are tired of typing)
 - The first letter can not be a digit
 - Java is case sensitive language

- Identifiers' naming conventions
 - **Class names:** starts with cap letter and should be inter-cap e.g. *StudentGrade*
 - **Variable names, method names, package names:** start with lower case and should be inter-cap e.g. *varTable*, *calTotal()*
 - **Constants:** often written in all capital and use underscore if you are using more than one word. E.g. $PI = 3.14$

Literals

- Explicit (constant) value that is used by a program
- Literals can be digits, letters or others that represent constant value to be stored in variables
 - Assigned to variables
 - Used in expressions
 - Passed to methods
- E.g.
 - `Pi = 3.14;` *// Assigned to variables*
 - `C= a * 60;` *// Used in expressions*
 - `Math.sqrt(9.0);` *// Passed to methods*

Operator

- Symbols that take one or more arguments (operands) and operates on them to produce a result
- Operators
 - Arithmetic operators
 - Assignment operator
 - Increment/Decrement operators
 - Relational operators
 - Logical operators

Arithmetic expressions and operators

- Common operators for numerical values
 - Multiplication (*)
 - Division (/)
 - Addition (+)
 - Subtraction (-)
 - Modulus (%) the remainder of dividing **op1** by **op2**
- A **unary operator** requires *one* operand.
- A **binary operator** requires *two* operands.

unary operator

↓
-273.15
└───┘
operand

binary operator

↓
9 + 7
└─┘ └─┘
↑ ↑
----- operands

binary operator

price + price * 0.23
└─┘ └─┘ └─┘
↑ ↑ ↑
----- operands
└──────────┘
operand
↑
binary operator

- **Arithmetic Operator precedence**
 - Order of Operations (**PEMDAS**)
 1. **P**arentheses
 2. **E**xponents
 3. **M**ultiplication and **D**ivision from left to right
 4. **A**ddition and **S**ubtraction from left to right
- **Assignment Operators**
 - Assigns the value of op2 to op1.
 - $op1 = op2;$
 - Short cut assignment operators
 - E.g. $op1 += op2$ equivalent $op1 = op1 + op2$

- **Increment/Decrement operators**
 - **op++**
 - Increments op by 1;
 - evaluates to the value of **op** before it was incremented
 - **++op**
 - Increments op by 1;
 - evaluates to the value of **op** after it was incremented
 - **op--**
 - Decrements op by 1;
 - evaluates to the value of **op** before it was decremented
 - **--op**
 - Decrements op by 1;
 - evaluates to the value of **op** after it was decremented

- **Relational Operators**

- Refers to the relationship that values can have with each other
- Evaluates to true or false
- All relational operators are *binary*.

- E.g. `int i = 1999;`

`boolean isEven = (i%2 == 0); // false`

`float numHours = 56.5;`

`boolean overtime = numHours > 37.5; // true`

Operator	Meaning (<i>a and b are arithmetic expressions</i>)
<code>a == b</code>	<code>a</code> is equal to <code>b</code> ?
<code>a != b</code>	<code>a</code> is not equal to <code>b</code> ?
<code>a < b</code>	<code>a</code> is less than <code>b</code> ?
<code>a <= b</code>	<code>a</code> is less than or equal to <code>b</code> ?
<code>a > b</code>	<code>a</code> is greater than <code>b</code> ?
<code>a >= b</code>	<code>a</code> is greater than or equal to <code>b</code> ?

- **Logical Operators**

- Refers to the way in which true and false values can be connected together

Operator	Use	Returns true if
&&	op1 && op2	op1 and op2 are both true, conditionally evaluates op2
	op1 op2	either op1 or op2 is true, conditionally evaluates op2
!	! op	op is false
&	op1 & op2	op1 and op2 are both true, always evaluates op1 and op2
	op1 op2	either op1 or op2 is true, always evaluates op1 and op2
^	op1 ^ op2	if op1 and op2 are different--that is if one or the other of the operands is true but not both
10/22/24		20

- **Precedence rules:**

1. Negation
2. Conditional And
3. Conditional Or

- The binary operators Conditional Or (||) and And (&&) associate from *left to right*.
- The unary operator Negation (!) associate from *right to left*.

- **Using && and ||**

- Examples:

(a && (b++ > 3))

(x || y)

- Java will evaluate these expressions from left to right and so will evaluate

a before (b++ > 3)

x before y

- Java performs **short-circuit evaluation**:
it evaluates `&&` and `||` expressions from left to right and *once it finds the result, it stops*.
- **Short-Circuit Evaluations**
(`a && (b++ > 3)`)
 - What happens if **a** is false?
 - Java will not evaluate the right-hand expression (`b++ > 3`) if the left-hand operator **a** is **false**, since the result is already determined in this case to be false. This means **b** will not be incremented!(`x || y++`)
 - What happens if **x** is true?
 - Similarly, Java will not evaluate the right-hand operator **y** if the left-hand operator **x** is **true**, since the result is already determined in this case to be true.

Operator Precedence

- Arithmetic Operators
- Relational Operators
- Assignment Operators

Separators

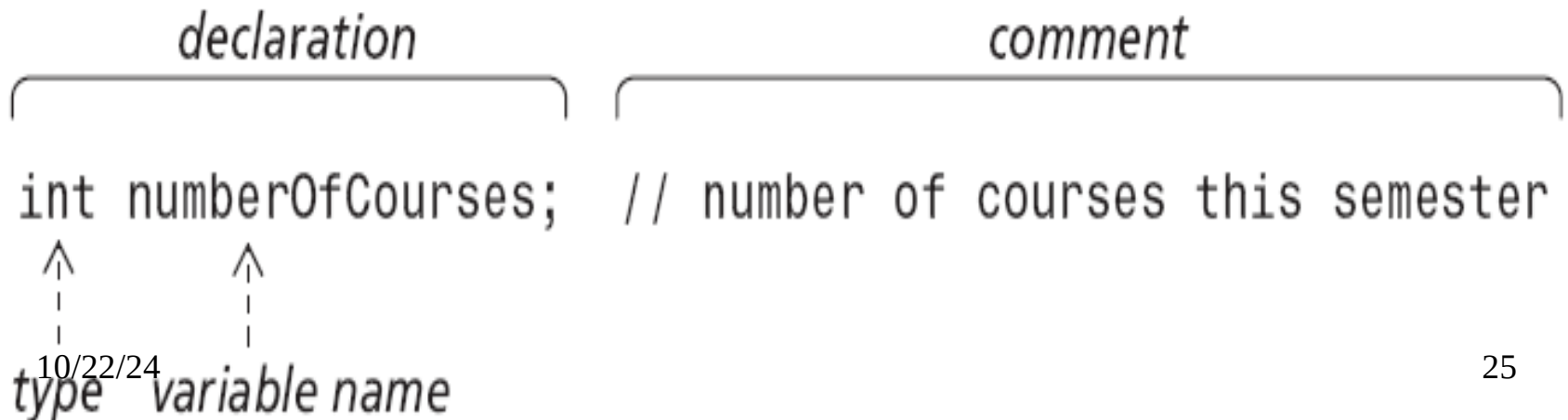
- Indicate where groups of codes are divided and arranged

Name	Symbol
Parenthesis	()
braces	{ }
brackets	[]
semicolon	;
comma ,	,
period	.

Variables and Data Types

- **Variables**

- named memory location that can be assigned a value
- an item of data named by an identifier
- An object stores its state in variables
- Declaring a variable:
 - ***Variable name*** - unique name of the memory location
 - ***Data type (or type)*** - defines what kind of values the variable can hold
 - ***Comment*** - describes its purpose, helps in understanding the program

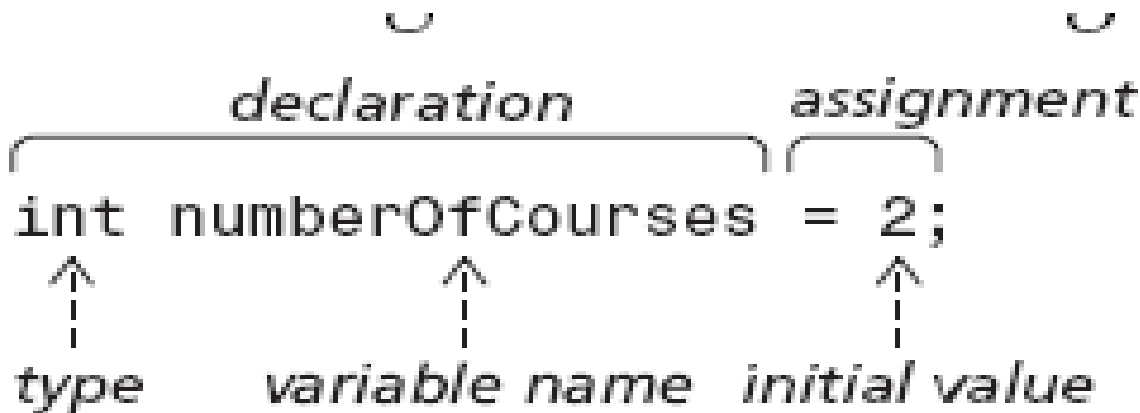


The diagram illustrates the components of a variable declaration in C++: `int numberOfCourses; // number of courses this semester`. A bracket above the code spans from the start to the semicolon, labeled *declaration*. Another bracket above the code spans from the semicolon to the end of the line, labeled *comment*. Below the code, two dashed arrows point upwards: one from the word *type* to `int`, and another from the words *variable name* to `numberOfCourses`. The date `10/22/24` is written in the bottom left corner.

```
int numberOfCourses; // number of courses this semester
```

type *variable name*

- **Best practice:**
 - Choose names that makes the purpose of the variable easy to understand.
 - Write a short comment for each variable unless the purpose is obvious from its name.
- **Declaring and initializing a variable:**
 - *Assignment* - placing a value in the memory location referred to by the variable
 - *Initialization* - the first assignment to a variable

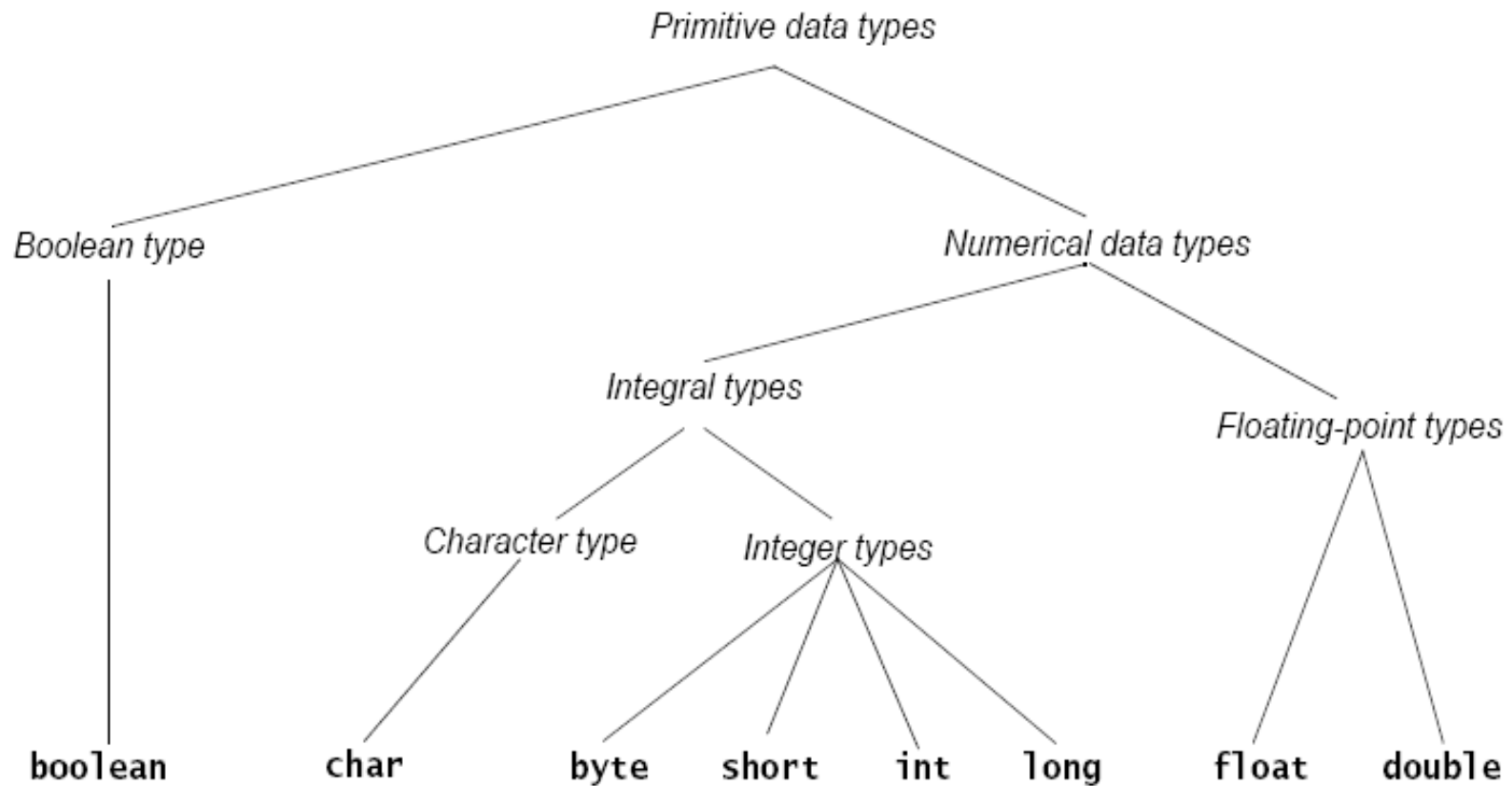


The diagram illustrates the components of a variable declaration and assignment in C++ using the example code: `int numberOfCourses = 2;`. A horizontal curly brace above the code spans from `int` to `numberOfCourses` and is labeled *declaration*. Another horizontal curly brace above the code spans from `=` to `2` and is labeled *assignment*. Below the code, three dashed arrows point upwards to specific parts: the first arrow points to `int` and is labeled *type*; the second arrow points to `numberOfCourses` and is labeled *variable name*; the third arrow points to `2` and is labeled *initial value*.

- There are three kinds of variables in Java.
 1. **Instance variables:** variables that hold data for an instance of a class.
 2. **Class variables:** variables that hold data that will be shared among all instances of a class.
 3. **Local variables:** variables that pertain only to a block of code.
- **Choosing variable names:**
 - Remember what Java accepts as valid names
 - Names can contain letters and digits, but cannot start with a digit.
 - Underscore (__) is allowed, also as the first character of a name.
 - Java distinguishes between upper and lower case letters in names.
 - Keywords like int cannot be used as variable names.
 - Determine which of the following names are valid and follow conventions.
 - int teamId, Team_Leader, 24x7, bestGuess!, MAX_COUNT

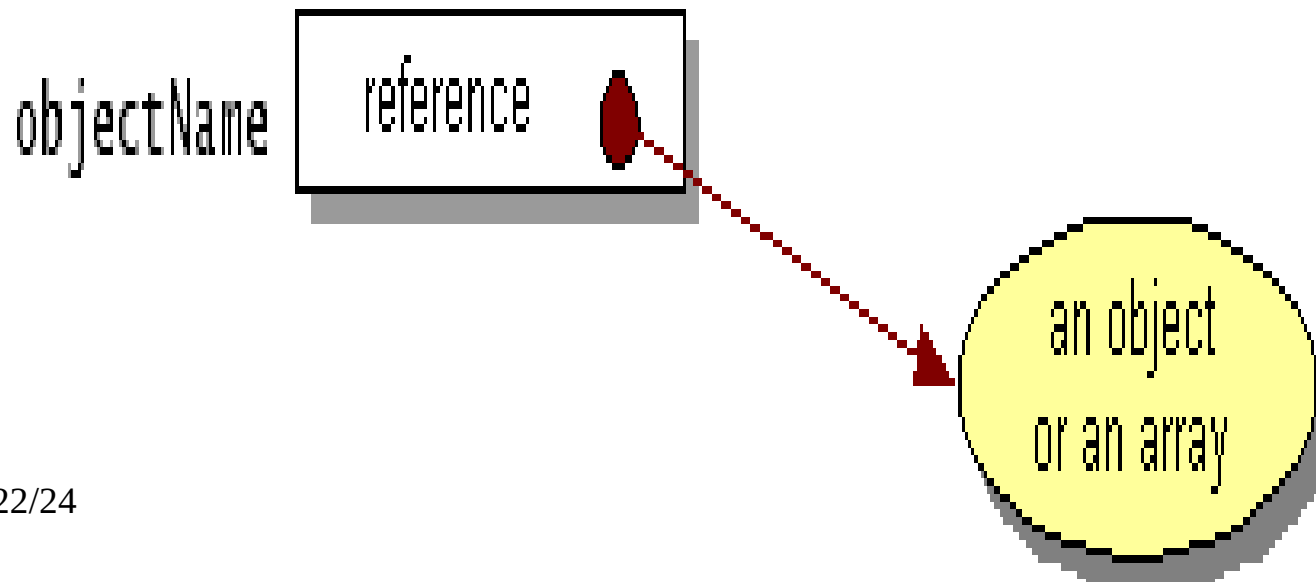
Data types

- Java has two basic data types
 - Primitive types
 - Reference types



Keyword	Description	Size/Format
<i>(integers)</i>		
byte	Byte-length integer	8-bit two's complement
short	Short integer	16-bit two's complement
int	Integer	32-bit two's complement
long	Long integer	64-bit two's complement
<i>(real numbers)</i>		
float	Single-precision floating point	32-bit IEEE 754
double	Double-precision floating point	64-bit IEEE 754
<i>(other types)</i>		
char	A single character	16-bit Unicode character
boolean ^{10/22/24}	A boolean value (true or false)	true or false ²⁹

- Boolean type has a value true or false (there is no conversion b/n Boolean and other types).
- All integers in Java are signed, Java doesn't support unsigned integers.
- Reference Type
 - Arrays, classes, and interfaces are *reference* types
 - The value of a reference type variable, in contrast to that of a primitive type, is a reference to (an address of) the value or set of values represented by the variable



Default values

- We can not use variables with out initializing them in Java.

<i>Primitive</i>	<i>Default</i>
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	null
boolean	false
all references	null

Scope and Lifetime of variables

- scope is the region of a program within which the variable can be referred to by its simple name
- scope also determines when the system creates and destroys memory for the variable
 - **A block defines a scope.** Each time you create a block of code, you are creating a new, nested scope.
 - Objects declared in the outer block will be visible to code within the inner block down from the declaration. (The reverse is not true)
 - **Variables are created** when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope.
 - Variable declared within a block will lose its value when the block is left. Thus, the **lifetime** of a variable is confined to its scope.

- You can't declare a variable in an inner block to have the same name as the one up in an outer block.
- But it is possible to declare a variable down outside the inner block using the same name.

- **Blocks**

- a group of zero or more statements between balanced braces
- E.g

```
if (Character.isUpperCase(aChar)) {  
    System.out.println(aChar + " is upper case.");  
}  
else {  
    System.out.println(aChar + " is lower case.");  
}
```

Expression

- Combine literals, variables, and operators to form expressions
- Expression is segments of code that perform computations and return values .
- Expressions can contain: **Example:**
 - a number literal, 3.14
 - a variable, count
 - a method call, Math.sin(90)
 - an operator between two expressions (binary operator),
phase + Math.sin(90)
 - an operator applied to one expression (unary operator),
-discount
 - expressions in parentheses.

(3.14-amplitude)

10/22/24

Statements

- Roughly equivalent to sentences in natural languages
- Forms a complete unit of execution.
- terminating the expression with a semicolon (;)
- Three kinds of statements in java
 - expression statements
 - declaration statements
 - control flow statements
- expression statements
 - Null statements
 - Assignment expressions
 - Any use of ++ or --
 - Method calls
 - Object creation expressions

- *Example*
 - `aValue = 8933.234;` //assignment statement
 - `aValue++;` //increment statement
 - `System.out.println(aValue);` //method call statement
 - `Integer integerObject = new Integer(4);` //object creation
- A declaration statement declares a variable
 - `double aValue = 8933.234;` // declaration statement
- A *control flow statement* regulates the order in which statements get executed
 - E.g - if, for

Conversion between primitive data types

- Evaluation of an arithmetic expression can results in *implicit conversion* of values in the expression.
 - **Integer arithmetic** yields results of type **int** unless at least one of the operands is of type long.
 - **Floating-point arithmetic** yields results of type **double** if one of the operands is of type double.
- Conversion of a "*narrower*" data type to a "*broader*" data type is (usually) done without loss of information, while conversion the other way round may cause loss of information.
- The compiler will issue a error message if such a conversion is illegal.

- Example
 - `int i = 10;`
 - `double d = 3.14;`
 - `d = i; // OK!`
 - `i = d; // Not accepted without explicit conversion (cast)`
 - `i = (int) d; // Accepted by the compiler`
- Explicit type casting
 - Syntax:
 - ***(<dest-type>)<var_Idf or value>.***
- *Example:*
 - `float f = 3.5f;` `int x = (int)2.7;`
 - `f = (float)2.9;`
 - `float x = 98;`
 - `byte b = 2;`
 - `float y = x + b;`
 - `b = (byte)y;`

Java Arrays

- Obtaining a Java array is a two step process:
 1. Declare a variable that can reference an array object
 2. Create the array object by using operator "**new**"

- The two steps may be done separately ...

int numbers[]; *or* *int[] numbers;*
numbers=new int[10]; *numbers=new int[10];*

OR

int numbers[] = new int[10]; or int[] numbers = new int[10];

- Array creation defines the size of the array and allocates space for the array object
- By placing the brackets in front of the variable name, you can more easily declare multiple arrays
 - *For example: int [] firstArray, secondArray;*

- **Programmer-Initialization of Arrays:**
 - `int[] nums={ 30,50,-23,16 };`
- **Automatic Initialization of Arrays:**
- If not programmer-initialized, array elements will be automatically initialized as follows:

type	default initialization
integer	0
boolean	false
char	null
String	null
object	null

- **Array Initializing**

- `int intArray[] = {1,2,3,4,5};`

- `char [] charArray = {'a', 'b', 'c'};`

- `String [] stringArray = {"A", "Four", "Element", "Array"};`

- **Array Access**

- Items in a Java array are known as the **components of the array**

- **For example:**

- `int a = intArray[0];` // a will be equal to 1

- Java arrays are numbered from 0 to one less than the number of components in the array

- Attempting to access an array beyond the bounds of the array will result in a runtime exception,

- ***ArrayIndexOutOfBoundsException.***

- **Limitations of Arrays in C++**

- No bound checking (overflow) or no array index checking.
- Array copying is not possible using the assignment statement.
- *Example: Creating and coping arrays*

```
Public class ArrayCopy{  
    public static void main(String args[]){  
        int copyArray[];  
        int originalArray[] = {10, 20, 30, 40, 50};  
        copyArray = originalArray;  
        copyArray[0] = 0;  
        for(int i = 0; i<originalArray.length; i++)  
            System.out.print(copyArray[i]+" ");  
    }  
}
```

- **Multidimensional Arrays**

- In Java, multidimensional arrays are actually arrays of arrays.
- When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension.
- You can allocate the remaining dimensions separately.

- **Two dimensional arrays**

- can be created with an array creation expression.

int b[][];

b = new int[3][4];

- A multidimensional array in which each row has a different number of columns can be created as follows.

int b[][];

b = new int[2][];

b[0] = new int[5];

b[1] = new int[3];

// Manually allocate differing size second dimensions.

```
class TwoDAgain {  
    public static void main(String args[]) {  
        int twoD[][] = new int[3][];  
        twoD[0] = new int[1];  
        twoD[1] = new int[2];  
        twoD[2] = new int[3];  
        int i, j, k = 0;  
        for(i=0; i<3; i++)  
            for(j=0; j<i+1; j++) {  
                twoD[i][j] = k;  
                k++;    }  
        for(i=0; i<3; i++) {  
            for(j=0; j<i+1; j++)  
                System.out.print(twoD[i][j]  
+"");  
            System.out.println(); } } }
```

- This program generates the following output:

0

1 2

3 4 5

- To initialize two dimensional array

```
int arr1[][] = {{1,2,3}, {4,5,6}};
```

```
int srr2[][] = {{1}, {1,2}, {1,2,3}, {1,2}};
```

- The use of uneven (or, irregular) multidimensional arrays is not recommended for most applications,
 - because it runs contrary to what people expect to find when a multidimensional array is encountered.
 - However, it can be used effectively in some situations.
 - **For example**, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used)

Decision Statements

- A **decision statement** allows the code to execute a statement or block of statements conditionally.

Example:

```
public class Equivalence {  
    public static void main(String[] args) {  
        Integer n1 = new Integer(47);  
        Integer n2 = new Integer(47);  
        System.out.println(n1 == n2);  
        System.out.println(n1 != n2);  
    }  
}
```

- If the two operands are object, == and != operators compares object references. Hence, the Output: **False** and then **True**
- the special method **equals()** is used to compare the contents of the objects.

Example:

```
public class EqualsMethod {  
    public static void main(String[] args) {  
        Integer n1 = new Integer(47);  
        Integer n2 = new Integer(47);  
        System.out.println(n1.equals(n2));  
    }  
}
```

The output is **True**.

- The method **compareTo()** is used to compare two strings.
 - It returns 0 if two strings are equal,
 - returns negative if the first string is less than the second
 - otherwise, it returns positive.

- There are two types of decisions statements in Java:
 - **if** statements
 - **switch** statements
- **if Statement**
 - Syntax:
if (expression) {
statement;
}
//rest_of_program;
- expression must evaluate to a **boolean** value, either **true** or **false**
- If expression is **true**, **statement** is executed and then **rest_of_program**
- If expression is **false**, **statement** is not executed and the program continues at **rest_of_program**

- **if - else Statement**

```
    if (expression) {  
        statement1;  
    }  
    else{  
        statement2;  
    }  
    next_statement;
```

- Again, expression must produce a **boolean** value
If expression is **true**, **statement1** is executed and then **next_statement** is executed.
- If expression is **false**, **statement2** is executed and then **next_statement** is executed.

The if - else if – else Ladder

```
if (grade == 'A')  
    System.out.println("You got an A.");  
else if (grade == 'B')  
    System.out.println("You got a B.");  
else if (grade == 'C')  
    System.out.println("You got a C.");  
else  
    System.out.println("You got an F.");
```

```
public class StringComparison {  
    public static void main(String args[]){  
        String str = "Hello";  
        String str1 = "hello";  
        if(str.equals(str1))  
            System.out.println("They are equal");  
        else  
            System.out.println("They are not equal");  
    }  
}
```

- Exercise: Write a program that accepts students' mark and calculate the grade based on the following rule:
 - If Mark < 40, Grade = F
 - If Mark < 45, Grade = D-
 - If Mark < 50, Grade = D
 - If Mark < 55, Grade = C-
 - If Mark < 65, Grade = C
 - If Mark < 70, Grade = C+
 - If Mark < 75, Grade = B-
 - If Mark < 80, Grade = B
 - If Mark < 85, Grade = B+
 - If Mark < 90, Grade = A-
 - Otherwise, Grade = A

- **Switch Statements**

- The `switch` statement enables you to test several cases generated by a given expression.

- **For example:**

```
switch (expression) {  
    case value1:  
        statement1;  
    case value2:  
        statement2;  
    default:  
        default_statement;  
}
```

- Every statement after the true case is executed
- The expression must evaluate to a `char`, `byte`, `short` or `int`, but not `long`, `float`, or `double`.

- **Break Statements in Switch Statements**

- The **break** statement tells the computer to exit the switch statement

- **For example:**

```
switch (expression) {  
    case value1:  
        statement1;  
        break;  
    case value2:  
        statement2;  
        break;  
    default:  
        default_statement;  
        break;  
}
```

- if-else chains can be sometimes be rewritten as a “switch” statement.

- switches are usually simpler and faster

Loops

- A **loop** allows you to execute a statement or block of statements repeatedly.
- Three types of loops in Java:
 1. while loops
 2. for loops
 3. do-while loops

- **The while Loop**

```
while (expression){  
    statement  
}
```

- while loop executes as long as the given logical expression between parentheses is true

- For example:

```
int sum = 0;
int i = 1;
while (i <= 10){
    sum += i;
    i++;
}
```

- What is the value of sum?

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55$$

- **do-while**

- Its general form is

```
do {  
    // body of loop  
} while (condition);
```

- Each iteration of the **do-while loop first executes the body of the loop and then evaluates** the conditional expression.
- If this expression is true, the loop will repeat. Otherwise, the loop terminates.

For example

```
int sum = 0;  
int i = 1;  
do {  
    sum += i;  
    i++;  
} while (i <= 10);
```

- **The for Loop**

```
for (init_expr; loop_condition; increment_expr) {  
    statement;  
}
```

- The control of the for loop appear in parentheses and is made up of three parts:
 1. The first part, the **init_expr**, sets the initial conditions for the loop and is executed before the loop starts.
 2. Loop executes so long as the **loop_condition** is true and exits otherwise.
 3. The third part of the control information, the **increment_expr**, is usually used to increment the loop counter. This is executed at the end of each loop iteration.

- **For example:**

```
int sum = 0;  
for (int i = 1; i <= 10; i++) {  
    sum += i;  
}
```

- What is the value of sum?

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55$$

- **Example :**

```
for(int div = 0; div < 1000; div++){  
    if(div % 2 == 0) {  
        System.out.println("even: " + div);  
    }  
    else {  
        System.out.println("odd: " + div);  
    }  
}
```

- What will this for loop do?
 - prints out each integer from 0 to 999,
 - correctly labeling them even or odd
- If there is more than one variable to set up or increment they are separated by a comma.

```
for(i=0, j=0; i*j < 100; i++, j+=2) {  
    System.out.println(i * j);  
}
```

- **for Loop Variations**

- One of the most common variations involves the conditional expression.
- This expression does not need to test the loop control variable against some target value.

```
boolean done = false;  
for(int i=1; !done; i++) {  
    // ...  
    if(interrupted()) done = true;  
}
```

- Either the initialization or the iteration expression or both may be absent,

```
int n = 0;  
for(; n <= 100;) {  
    System.out.println(++n);  
}
```

- You can intentionally create an infinite loop (a loop that never terminates) if you leave all three parts of the **for empty**.

- **For example:**

```
for( ; ; ) {  
    // ...  
}
```

- **The for-each Loop**

- The general syntax for the for-each loop is

```
for ( <type> <variable> : <array> )  
    <loop body>
```

- The for-each loop is a very convenient way to iterate over a collection of items.
- By using a for-each loop, we can compute the sum as follows:

```
int sum = 0;  
for (int value : number) {  
    sum = sum + value;  
}
```

- The loop iterates over every element in the **number** array, and the loop body is executed for each iteration.
- The variable **value** refers to each element in the array during the iteration.
- We cannot use the for-each loop directly to a **String** to access individual characters in it.
- However, the **String** class includes a method named **toCharArray** that returns a string data as an array of char data, and we can use the for-each loop on this array of char data.
- Here's an example:

```
String sample = "This is a test";  
char[] charArray = sample.toCharArray();  
for (char ch: charArray) {  
    System.out.print(ch + " ");
```

```
}
```

- If we do not need to access the char array later, then we can write the code in a slightly less verbose way as follows:

```
String sample = "This is a test";  
for (char ch: sample.toCharArray()) {  
    System.out.print(ch + " ");  
}
```

- Suppose we have an array of 100 Person objects called person:

```
Person[] person = new Person[100];
```

- we can list the name of every person in the array by using the following for-each loop:

```
for (Person p : person) {  
    System.out.println(p.getName());  
}
```

- Contrast this to the standard for loop:

```
for (int i = 0; i < person.length; i++) {  
    System.out.println(person[i].getName());  
}
```

- The for-each loop is, in general, cleaner and easier to read.
- There are several restrictions on using the for-each loop.
- **First**, you cannot change an element in the array during the iteration.
 - The following code does not reset the array elements to 0:

```
int [ ] number = {10, 20, 30, 40, 50};  
for (int value : number) {  
    value = 0;  
}
```
 - For an array of objects, an element is actually a reference to an object, so the following for-each loop is ineffective.
 - Specifically, it does not reset the elements to null.

```
Person[] person = new Person[100];  
for (int i = 0; i < person.length; i++) {  
    person[i] = ...; //code to create a new  
    Person object  
}  
for (Person p : person) {  
    p = null;  
}
```

- Although we cannot change the elements of an array, we can change the content of an object if the element is a reference to an object.
- For example, the following for-each loop will reset the names of all objects to **Java**:

```
for (Person p : person) {  
    p.setName("Java");  
}
```

– **Second**, we cannot access more than one array using a single for-each loop.

- Suppose we have two integer arrays **num1** and **num2** of length 200 and want to create a third array **num3** whose elements are sum of the corresponding elements in num1 and num2. Here's the standard for loop:

```
int[] num1 = new int[200];
int[] num2 = new int[200];
int[] num3 = new int[200];
//code to assign values to the elements of
  num1 and num2
//compute the sums
for (int i = 0; i < num3.length; i++) {
    num3[i] = num1[i] + num2[i];
}
```

- Such a loop cannot be written with a for-each loop.

- **Third**, we must access all elements in an array from the first to the last element.
 - We cannot, for example, access only the first half or the last half of the array.
 - We cannot access elements in reverse order either.
- This restriction complicates the matter when we try to access elements in an array of objects.
- Consider the following code:

```
Person[] person = new Person[100];
for (int i = 0; i < 50; i++) {
    person[i] = ...; //code to create a new Person object
}
for (Person p : person) {
    System.out.println( p.getName()); //This loop will result in a
//NullPointerException error.
}
```

- This code will crash when the variable `p` is set to the 51st element (i.e., an element at index position 50), because the element is `null`.
- Notice that only the first 50 elements actually point to `Person` objects. The elements in the second half of the array are all **`null`**.

- **The `continue` Statement**

- The `continue` statement causes the program to jump to the next iteration of the loop.

```
// prints out "5689"
for(int m = 5; m < 10; m++) {
    if(m == 7) {
        continue;
    }
    System.out.print(m) ;
}
```

- Another continue example:

```
int sum = 0;
for(int i = 1; i <= 10; i++){
    if(i % 3 == 0) {
        continue;
    }
    sum += i;
}
```

- What is the value of sum?

$1 + 2 + 4 + 5 + 7 + 8 + 10 = 37$

- **Using Continue as a Form of Goto**

- continue may specify a label to describe which enclosing loop to continue.

// Using continue with a label.

```
class ContinueLabel {
    public static void main(String args[]) {
        outer: for (int i=0; i<10; i++)
        {
            for(int j=0; j<10; j++) {
                if(j > i) {
                    System.out.println();
                    continue outer;
                }
                System.out.print(" "+ (i * j));
            }
        }
        System.out.println();
    }
}
```

- **The break Statement**

- We have seen the use of the break statement in the switch statement.
- You can also use the break statement to exit the loop entirely.

```
// prints out numbers unless
// num is ever exactly 400
while (num > 6) {
    if (num == 400) {
        break;
    }
    System.out.println(num);
    num -= 8;
}
```

- **Using break as a Form of Goto**

- Java defines an expanded form of the **break statement**.
- Using this **break** you can break out of one or more blocks of code.
- These blocks need not be part of a loop or a **switch**. They can be any block.
- you can specify precisely where execution will resume, because this form of **break works with a label**.
- The general form

`break label;`

- A **label** is any valid Java identifier followed by a colon.
- Once you have labeled a block, you can then use this label as the target of a **break statement**.
- Doing so causes execution to resume at the end of the ***labeled block***.

// Using break as a civilized form of goto.

```
class Break {  
    public static void main(String args[]) {  
        boolean t = true;  
        first: {  
            second: {  
                third: {  
                    System.out.println("Before the break.");  
                    if(t) break second;  
                    System.out.println("This won't execute");  
                }  
                System.out.println("This won't execute");  
            }  
            System.out.println("This is after second block.");  
        } } }
```

- Keep in mind that you cannot break to any label which is not defined for an enclosing block.

Nested Loops

- You can nest loops of any kind one inside another to any depth. What does this print?

```
for(int i = 10; i > 0; i--) {  
    if (i > 7) {  
        continue;  
    }  
    while (i > 4) {  
        if(i == 6) {  
            break;  
        }  
        System.out.println(--i);  
    }  
    System.out.println(i);  
}
```

6
6
4
4
3
2
1

For each loop for Two Dimensional Array

```
public static void main(String[] args)      {
    int x[][] = new int[3][2];
    for(int i=0;i<3;i++)
        for(int j=0;j<2;j++)
            x[i][j]=j;
    for(int a[]:x){
        for(int b:a){
            System.out.println(b);
        }
    }
}
```