Java Language Basics

Java Language Basics

- Essential Syntax
  - Primitive types
  - Reference types
    - objects
- Two Useful Classes: **`String`** and **`Array`**
- Control Flow
- Input and Output

Identifiers

- An identifier is a name given to a ***variable***, ***method*** or ***class***
  - An identifier starts with a *letter, underscore (_), or dollar sign* ($); it can be any length; and, other than the first character, it can contain any sequence of *letters, digits, underscores, or dollar signs*
    - Do not use $ in your own code; if is used for names generated by Java compiler and other tools
  - Java is case sensitive

eg:

```
x                  greeting(){…}
userName           class TestGreeting
_sys_var1          $charge
```

Keywords

- Reserved words
- Forbidden for identifiers

eg:

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

Semicolons and White Spaces

- **Semicolons**
  - a statement is one or more lines of code terminated with a semicolon (;)
  - Example

```
totals = a + b + c + d + e + f;
totals = a + b + c +
              d + e + f;
```

- **White spaces**
  - You can have white space between elements of the source code. Any amount of white space is allowed

Java Variable Types

- **Primitive Types**
  - storing whole numbers (integer types), numbers with decimal places as well (floating point types), true/false logical values, and so on

- **Reference types**
  - to hold the ***addresses of objects***. When you create objects, you store them in a separate part of memory, but you need to remember the address of where they were created

No pointer type in Java, unlike C/C++

Primitive Data Types

- Logical: **`boolean`**: 1 bit = true/false *not an integer, unlike C/C++*
- Textural: **`char`**: 2 bytes = \u0000 to \uFFFF
- Integer: *no unsigned integers, unlike C/C++*

  `byte`: 1 byte = -2^7 to 2^7-1 (-128 to 127) *use to save memory in large arrays*

  `short`: 2 bytes = -2^{15} to 2^{15} -1(-32,768 to 32,767)

  `int`: 4 bytes = -2^{31} to 2^{31}-1 (-2,147,483,648 to 2,147,483,647)

  `long`: 8 bytes = -2^{63} to 2^{63}-1 (-9,223,372,036,854,775,808L to 9,223,372,036,854,775,807L)
- Floating-point:

  `float`: 4 bytes = roughly +/-3.40282347E+38F
  (7 significant decimal digits)

  `double`: 8 bytes = roughly +/- 1.79769313486231570E+308
  (15 significant decimal digits)

3 special floating-point values: **`Infinity`**, **`-Infinity`** and **`NaN`** (not a number)

e.g. **`1.1/0.0 -> Infinity,`** **`0.0/0.0 -> NaN`** What about 1/0 ?

double literal

Enumerated Types

- A restricted set of values

```
enum Size {SMALL, MEDIUM, LARGE, EXTRA_LARGE};

Size s = Size.MEDIUM;
```

Initializing Variables

- Every Java variable has a *type*
  - Java is a strongly typed language

- Variables must be explicitly initialized

In Java, no declarations are separated from definition

```
        int counter;
        counter = 0;
    or
        int counter = 0;
```

```
        int counter;
        System.out.println(counter); //Error!
```

Literals

- Character
  - char:

```
eg. char capitalC = 'C'
```

- Integer literals
  - int:

```
eg. int i = 255;     // in decimal
    int j = 0xFEEF; // in hexadecimal
    int k = 0b11010; // in binary
```

  - long: L/l. (recommended to use L not lower case letter l)

```
eg. 255L; 0xFEEFL
```

- Floating-point literals
  - F/f/D/d (E/e)

```
eg. 123.4; 123.4F; 123.4D;
    1.234e2 // in scientific notation
```

Default Values for Class Fields

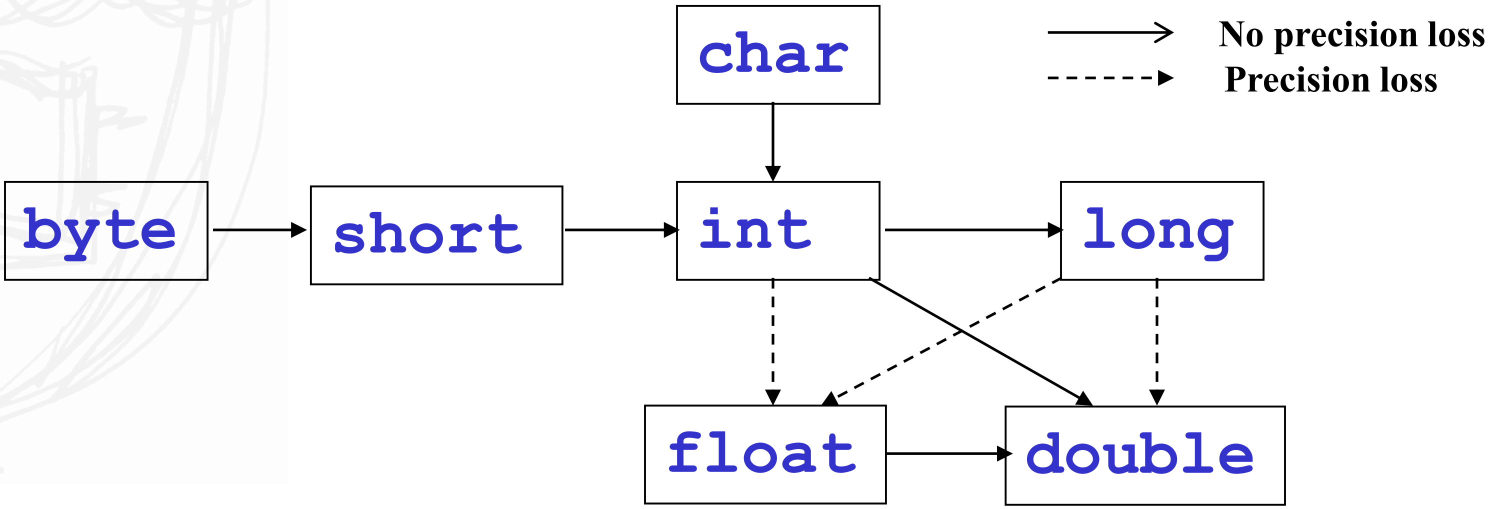
```
class Aclass {
   int counter;

   public void aMethod() {
       int i =0;
       // . . .
    }
}
```

class field

local variable

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Conversion Between Numeric Types



char

byte

short

int

long

float

double

No precision loss

Precision loss

Examples:

- Auto conversion between compatible types

```
short n = 1234;
float f = n;        //f = 1.234E3
```

A large integer may have more digits than the float type can represent

```
int n = 123456789;
float f = n;        //f is 1.23456792E8  (precision loss)
```

- ***Explicit cast*** where information would be lost

```
float pi = 3.14159F;
int i = (int)pi;  // i = 3
```

target type

Constants

- In Java, use the keyword `final` to denote a constant
  - `final` indicates that you can assign to the variable once. Its value is set once and for all
  - Customary to name constants in all uppercase

```
final double CM_PER_INCH = 2.54;

public static final double CM_PER_INCH = 2.54;
```

Class constant

Operators

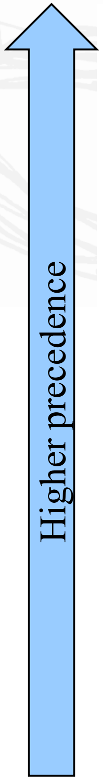
- Arithmetic: `+ - * / %`
  - integer/integer → integer, otherwise → floating-point
  - integer/0 → exception (error)
  - Floating-point/0 → **`infinite`** or **`NaN`**
  - Shortcuts for binary arithmetic operators in an assignment

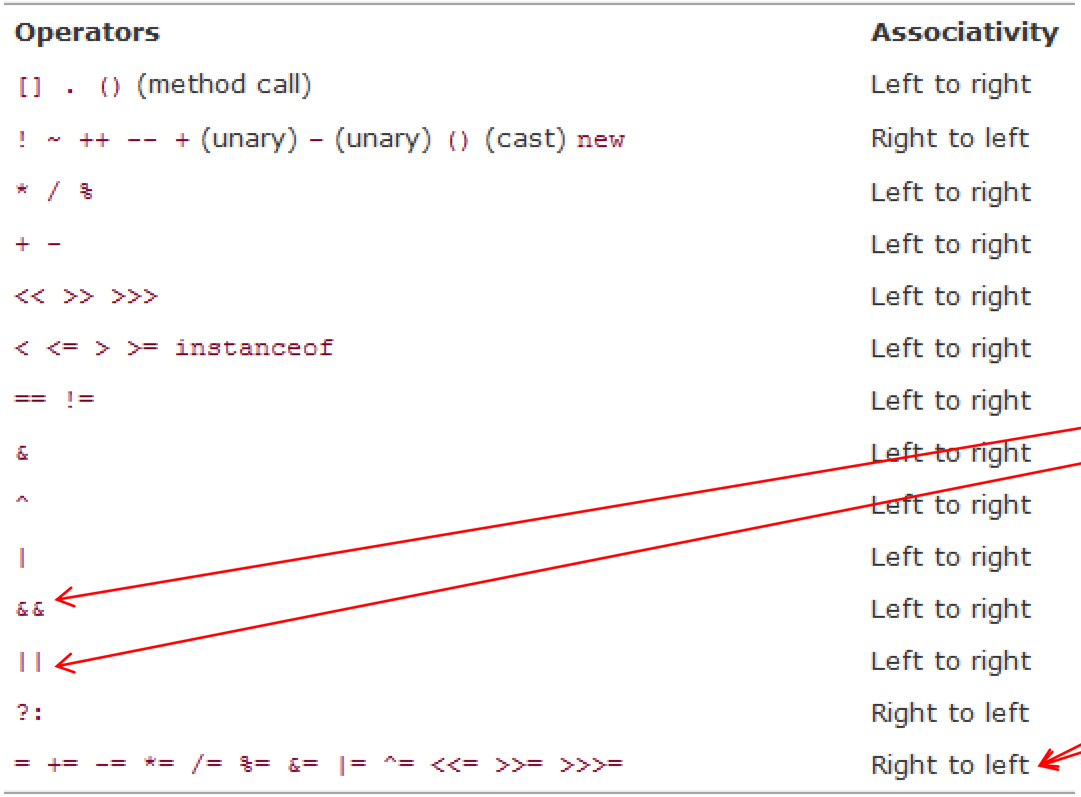
    eg:

    `x += 4;` → **`x = x + 4;`**
- Incremental and decremental: `++ --`
- Relational and conditional (**`boolean`**): `== != && ||`
  - ternary: eg: **`x < y ? x : y`**
- Bitwise: & | ^ ~ << >>

Parentheses and Operator Hierarchy

- Operator precedence





Operators	Associativity
[] . () (method call)	Left to right
! ~ ++ -- + (unary) – (unary) () (cast) new	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >= instanceof	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &= |= ^= <<= >>= >>>=	Right to left

eg:

```
   a && b || c
```

means

```
    (a && b) || c
```

```
   a += b += c
```

means

```
   a += (b += c)
```

Mathematical Functions and Constants

- **`Math`** class contains mathematical functions – static class

eg:

```
y = Math.sqrt(x)
```

or

```
import static java.lang.Math.*;
y = sqrt(x);
```

- Mathematical constants

eg:

```
Math.PI
```

What are Reference Variables

- Store references to objects of the same class type

```
class OTDate {
    int day;
    int month;
    int year;

    int getYear( ){
        //method code
    }

    int getMonth( ){
        //method code
    }

    int getDay( ) {
        //method code
    }
}
```

eg:

```
OTDate today;
   // reference variable of class type OTDate
OTDate payday;
   // another reference variable of type OTDate
```

Creating Objects

Before you can use a variable of a class type, the actual storage must be allocated. This is done by using the keyword **`new`** as shown below:

```
Date today;
today = new Date( );
```

or

```
Date today = new Date( );
```

This is the time when the life of an object starts

Constructing and Initializing Objects

- Calling

```
new ClassName()
```

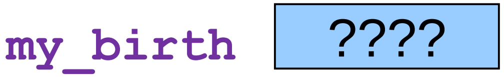
to allocate space for the new object results in:

  - **Memory allocation**: space for the new object is allocated and instance variables are initialized to their default values
  - **Explicit attribute initialization** is preformed
  - **A constructor is executed**
  - **Variable assignment is made to reference the object**

Memory Allocation and Layout

- A declaration allocates storage only for the reference

```
MyDate my_birth;
```

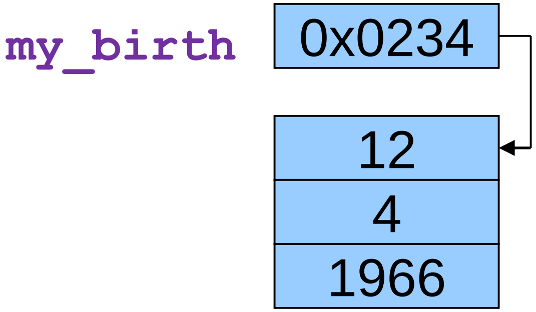


my_birth

????

- Use the **`new`** operator to allocate space for the **`MyDate`** object

```
MyDate my_birth = new MyDate(12, 4, 1966);
```



my_birth

0x0234

12

4

1966

Pass-by-Value

- Java only passes arguments ***by value***
- When an object instance is passed as an argument to a method, the value of the argument is ***a reference to the object***
- The contents of the object can be changed in the called method, but the ***object reference is never changed***

The `this` Reference

- To reference local attribute and method members
- To pass the current object as a parameter to another methods or constructor

```
public class MyDate {
    private int day = 1;
    private int month = 1;
    private int year = 2000;

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }
}
```

Strings

- **`String`** is a class
  - to store words or sentences

- **`String`** literal
  - double quotation marks
  - initialize without the **`new`** Keyword

eg:

```
String myString = “This is a String literal.”
```

- Substrings: using method substring

eg:

```
String greeting = “Hello”;
String s = greeting.substring(0 ,3);
```

String Concatenation with +

- The + operator performs **`String`** concatenation, which produces a new **`String`**
- One argument must be a **`String`** object and non-**`String`**s are converted to **`String`** object automatically

Examples:

```
String salutation = “Mr.”;
String name = “Peter” + “ ” + “Citizen”;
String title = salutation + “ ” + name;
```

Concatenating method:

```
        string1.concat(string2);
```

eg. `“Hello, ".concat(“world!");`

Format Strings

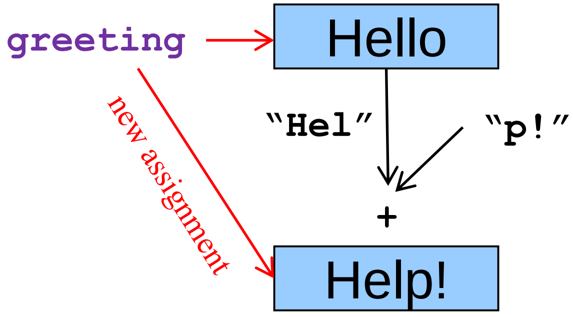
- **`format()`** method
  - Like **`printf()`**

```
String fs;
fs = String.format("The value of the float " +
                   "variable is %f, while " +
                   "the value of the " +
                   "integer variable is %d, " +
                   " and the string is %s",
                   floatVar, intVar, stringVar);

System.out.println(fs);
```

String Mutability

- Java strings are ***immutable***
  - No methods to change a character in an existing string

```
String greeting = "Hello";

greeting[3] = "p";   //error!
greeting[4] = "!";   //error!

greeting = greeting.substring(0,3) + "p!";
```



Referring to a different string

greeting now contains **Help!**

Most of the time, you do not change strings – you compare them

Building Strings

- Using **`StringBuilder`** class
  - String concatenation is inefficient
    - Constructing new **`String`** object
    - Wasting memory

eg:

```
StringBuilder builder = new StringBuilder();

builder.append(ch);   // appends a single character
bulder.append(str);   // appends a string

String completedString = bulder.toString();
```

String Equality

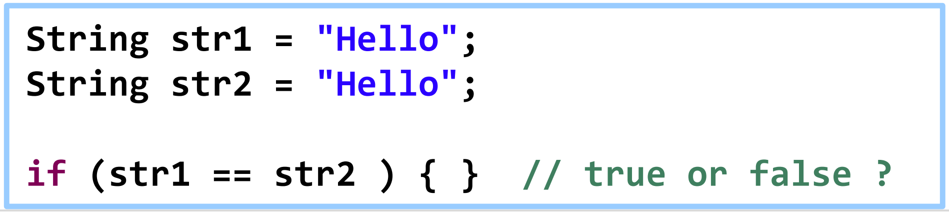
- Use **equals** method:

```
/* s and t can be string variable or string constants */
s.equals(t);
```

eg:

```
greeting.equals( “Hello” );                             ?
“Hello”.equals(greeting);    // better way
“Hello”.equalsIgnoreCase(greeting);
```

- Do not use == to test string equality
  - It tests if two strings are stored in the ***same location***



```
String str1 = "Hello";
String str2 = "Hello";

if (str1 == str2 ) { }  // true or false ?
```

Empty and `null` Strings

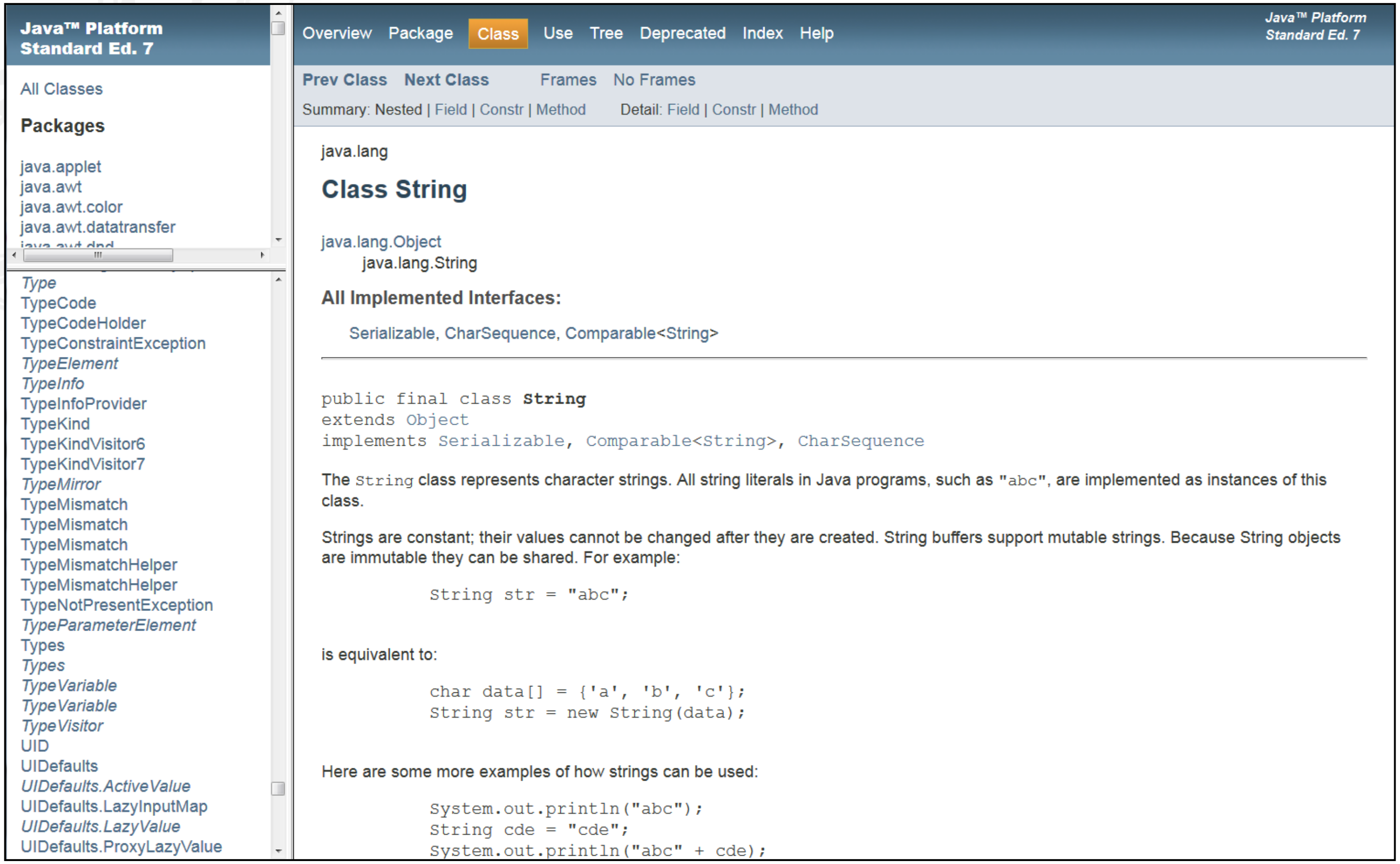
- ***Empty string*** `""` is a string of length 0
- ***`null` string*** is a String variable which holds a special value null
  - It is an error to invoke a method on a **`null`** value

Which is a better test?

```
if (str != null && str.length() !=0) {}

if (str.length() !=0 && str != null) {}
```

String API

The **`String`** class in Java contains more than 50 methods.



java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
     String str = "abc";
```

is equivalent to:

```
     char data[] = {'a', 'b', 'c'};
     String str = new String(data);
```

Here are some more examples of how strings can be used:

```
     System.out.println("abc");
     String cde = "cde";
     System.out.println("abc" + cde);
```

Arrays

- Declaring an array

  eg:

```
char s[];
Point p[];
char[] s;
Point[] p;
```

- Creating Arrays

  eg:

```
s = new char[20];
P = new Point[100];
```

- Multidimensional Arrays

  eg:

```
Int twoDimInt[ ][ ] = new int[4][6];
```

Default Initial Values of Arrays

- Numbers: zero

  eg:

  ```
  short:  0
  float:  0.0F
  ```

- Char: '\U0000'
- Boolean: false
- Reference type = null

Initialization and Anonymous Array

- Shorthand to create an array object and supply initial values at the same time

```
int[] smallPrimes = {2, 3, 5, 7, 11, 13};
```

- Anonymous array

```
smallPrimes = new int[] {2, 3, 5, 7, 11, 13};
```

means

```
int[] anomymous = {2, 3, 5, 7, 11, 13};
smallPrimes = anomymous;
```

Array Copying

- Copying one array variable into another results in both variables referring to the same array

```
int[] luckyNumbers = smallPrimes;
luckyNumber[5]  = 12;  // now smallPrimes[5] is also 12
```

- Using **`copyOf()`** method in the **`Array`** class to copy all values of an array into another

```
int[] copiedNumbers = Array.copyOf(origNumbers, origNumbers.length);
```

- Using **`arraycopy()`** method in the **`System`** class

```
System.arraycopy(from, fromIndex, to, toIndex, count);
```

The to array must have sufficient space

Example: Using `arraycopy`

- A special method in the **`System`** class, **`arraycopy`**

eg:

```
int myArray[] = {1,2,3,4,5};
int newArray[] = {11,10,9,8,7,6,5,4,3,2,1};
System.arraycopy(myArray, 0, newArray, 0,
                 myArray.length);
```

Resulted content of `newArray`: **`1,2,3,4,5,6,5,4,3,2,1`**

Control Flow

Similar to C/C++

- Branching Statements
  - `if/else`
  - `switch/case`

  case label:
    - A constant of type `char, byte, short, int` (Character, Byte, Short, Integer)
    - Enumerated constant
    - A string literal

- Looping Statements
  - `for` and `“for each”`
  - `while`
  - `do/while`

- Special control flow: `break`/`continue`/`label`
  - Good practice tip
    - do not use; they are confusing
    - You can always express the same logic without them

Example: for each Loop

Syntax:

```
for (variable : collection) statement
```

Traversing elements of an array

Don’t have to worry about index values

eg:

```
int[] a = {1, 3, 5, 7, 9};
for (int element : a)
   System.out.println(element);
```

means

```
for (int i = 0; i < a.length; i++)
   System.out.println(a[i]);
```

Looping on the index

Input and Output

- Reading input

```
import java.util.*;

Scanner in = new Scanner(System.in);
System.out.print("What is your name? ");
String name = in.nextLine();

System.out.print("How old are you? ");
int age = in.nextInt();
```

- Formatting output

```
double x = 10000.0 / 3.0;
System.out.print(x);                 // 3333.3333333333335
System.out.printf("%8.2f", x);       // 3333.33
```

File Input and Output

- Read from a file

```
Scanner in = new Scanner(Path.get("myfile.txt"));
```

- Write to a file

```
PrintWriter out = new PrintWriter("myfile.txt");
```

What is the output?

```
Scanner in = new Scanner("myfile.txt");
String name = in.nextLine();
System.out.println(name);
```

?

myfile.txt

```
This is my file.
```

Lookup API documentation

`Scanner(Path source)`
Constructs a new `Scanner` that produces values scanned from the specified file.

`Scanner(String source)`
Constructs a new `Scanner` that produces values scanned from the specified string.