# Lecture 3, 13 January 2026
# Java: Scalars, control flow, [redacted]

Madhavan Mukund, S P Suresh

Programming Language Concepts
January–April 2026

## Getting started with Java

**The C Programming Language, Brian W Kernighan, Dennis M Ritchie**

The only way to learn a new programming language is by writing programs in it. The first program is the same for all languages.

*Print the words*

`hello, world`

This is a big hurdle; to leap over it you have to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy

- In Python

```
print("hello, world")
```

- . . . C

```
#include <stdio.h>
main()
{
  printf("hello, world\n");
}
```

- . . . and Java

```
public class helloworld{
  public static void main(String[] args)
  {
    System.out.println("hello, world");
  }
}
```

## Scalar types

- Java is an object-oriented language
  - All data encapsulated as objects?

## Scalar types

- Java is an object-oriented language
  - All data encapsulated as objects?
- Not quite

| Type | Size in bytes |
|---|---|
| `int` | 4 |
| `long` | 8 |
| `short` | 2 |
| `byte` | 1 |
| `float` | 4 |
| `double` | 8 |
| `char` | 2 |
| `boolean` | 1 |

## Variables: declarations, assignment, initialization, constants

- Declare variables before use

```
int x, y;
double z;
char c;
boolean b1, b2;
```

Python

$x = 5$

⋮

$x =$ "hello"

width = 6

⋮

witdh = 7

## Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual

```
int x, y;
double z;
x = 5;
z = 7.0;
```

## Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual
- Characters — single quotes

```
char c,d;

c = 'x';
d = '\u03C0'; // Greek pi, unicode
```

## Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual
- Characters — single quotes
- Boolean constants

```
boolean b1, b2;

b1 = false;
b2 = true;
```

## Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual
- Characters — single quotes
- Boolean constants
- Declarations can come anywhere

```
int x;
x = 10;
double y;
```

## Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual
- Characters — single quotes
- Boolean constants
- Declarations can come anywhere
- Initialize with declaration

```
int x = 10;
double y = 5.7;
```

# Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual
- Characters — single quotes
- Boolean constants
- Declarations can come anywhere
- Initialize with declaration
- Constants

```
float pi = 3.1415927f;

pi = 22/7;  // Disallow
```

## Variables: declarations, assignment, initialization, constants

- Declare variables before use
- Assign values to variables as usual
- Characters — single quotes
- Boolean constants
- Declarations can come anywhere
- Initialize with declaration
- Constants

```
final float pi = 3.1415927f;

pi = 22/7;  // Flagged as error
```

## Operators, shortcuts

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`    mod

## Operators, shortcuts

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
  - No separate integer division `//`
  - When both arguments are integer, `/` is integer division

```
float f = 22/7;    // Value is 3.0
                   // Implicit conversion,
                   // int to float
```

$\boxed{\texttt{float f}} \leftarrow 3$

Python $7/3 \rightarrow 2.33\ldots$

$7//3 \rightarrow 2$

$7.0//3.0$ ? $2.0$

## Operators, shortcuts

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
  - No separate integer division `//`
  - When both arguments are integer, `/` is integer division
- Exponentiation:
  $\underline{\texttt{Math}}$`.pow(a,n)` returns $a^n$

Library

## Operators, shortcuts

- Arithmetic operators are the usual ones
  - `+, -, *, /, %`
  - No separate integer division `//`
  - When both arguments are integer, `/` is integer division
- Exponentiation: `Math.pow(a,n)` returns $a^n$
- Special operators for incrementing and decrementing integers

int i
array a

$x = a[i++]$
$x = a[++i]$

```
int a = 0, b = 10;
a++;    // Same as a = a+1
b--;    // Same as b = b-1
```

## Operators, shortcuts

- Arithmetic operators are the usual ones
  - `+`, `-`, `*`, `/`, `%`
  - No separate integer division `//`
  - When both arguments are integer, `/` is integer division
- Exponentiation:
  `Math.pow(a,n)` returns $a^n$
- Special operators for incrementing and decrementing integers
- Shortcut for updating a variable

```
int a = 0, b = 10;
a += 7;     // Same as a = a+7
b *= 12;    // Same as b = b*12
```

## Strings

- `String` is a built in class

```
String s,t;
```

## Strings

- `String` is a built in class
- String constants within double quotes `String s = "Hello", t = "world";`

## Strings

- String is a built in class
- String constants within double quotes
- + overloaded for string concatenation

```
String s = "Hello";
String t = "world";
String u = s + " " + t;
  // "Hello world"
```

## Strings

- `String` is a built in class
- String constants within double quotes
- `+` overloaded for string concatenation
- Strings are not arrays of characters

Cannot write

```
String s = "Hello";
s[3] = 'p';
s[4] = '!';
```

## Strings

- String is a built in class
- String constants within double quotes
- + overloaded for string concatenation
- Strings are not arrays of characters
- Instead, use method substring in class String

```
String s = "Hello";
s = s.substring(0,3) + "p!";
```

## Strings

- String is a built in class
- String constants within double quotes
- + overloaded for string concatenation
- Strings are not arrays of characters
- Instead, use method substring in class String
- If we update a String, we get a new object
  - Java does automatic garbage collection

```
String s = "Hello";
s = s.substring(0,3) + "p!";
```

## Arrays

- Arrays are also objects

## Arrays

- Arrays are also objects
- Typical declarations

```
int[] a;
a = new int[100];
```

## Arrays

- Arrays are also objects
- Typical declarations

```
int[] a;
a = new int[100];

int a[];
a = new int[100];
```

```
int x, y;

int x, y, a[];
```

## Arrays

- Arrays are also objects
- Typical declarations

```
int[] a;
a = new int[100];

int a[];
a = new int[100];

int a[] = new int[100];
```

## Arrays

- Arrays are also objects
- Typical declarations
- Array indices run from `0` to `a.length-1`
  - `a.length` gives size of `a`
  - For `String`, it is a method `s.length()`!

## Arrays

- Arrays are also objects
- Typical declarations
- Array indices run from `0` to `a.length-1`
  - `a.length` gives size of `a`
  - For `String`, it is a method `s.length()`!
- Array constants: `{v1, v2, v3}`

## Arrays

- Arrays are also objects
- Typical declarations
- Array indices run from `0` to `a.length-1`
  - `a.length` gives size of `a`
  - For `String`, it is a method `s.length()`!
- Array constants: `{v1, v2, v3}`
- Size of an array can vary dynamically

```
int[] a;
int n;

n = 10;
a = new int[n];

n = 20;
a = new int[n];

a = {2, 3, 5, 7, 11};
```

## Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces

## Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces
- Conditional execution
  - `if (condition) { ... } else { ... }`

```
if (x<0)
   x+=1;
```

## Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces
- Conditional execution
  - `if (condition) { ... } else { ... }`
- Conditional loops
  - `while (condition) { ... }` — zero or more times
  - `do { ... } while (condition)` — one or more times

## Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces
- Conditional execution
  - `if (condition) { ... } else { ... }`
- Conditional loops
  - `while (condition) { ... }`
  - `do { ... } while (condition)`
- Iteration
  - Two kinds of `for`

## Control flow

- Program layout
  - Statements end with semi-colon
  - Blocks of statements delimited by braces
- Conditional execution
  - `if (condition) { ... } else { ... }`
- Conditional loops
  - `while (condition) { ... }`
  - `do { ... } while (condition)`
- Iteration
  - Two kinds of `for`
- Multiway branching – `switch`

## Conditional execution

- `if (c) {...} else {...}`
  - Condition must be in parentheses
  - `else` is optional
  - No braces needed if body is single statement
- No `elif`, à la Python
  - Indentation is not forced - just align `else if`
  - Nested `if` is a single statement, no separate braces required
- No surprises
- Aside: no `def` for function definition

?

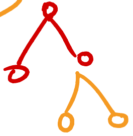
```
public class MyClass {

  ...

  public static int sign(int v) {
    if (v < 0) {
      return(-1);
    } else if (v > 0) {
      return(1);
    } else {
      return(0);
    }
  }

}
```

MyClass.sign(7)



## Conditional loops

- `while (c) {...}`
  - Condition must be in parentheses
  - No braces needed if body is single statement

```
public class MyClass {

  ...

  public static int sumupto(int n) {
    int sum = 0;

    while (n > 0){
      sum += n;
      n--;
    }

    return(sum);
  }

}
```

## Conditional loops

- `while (c) {...}`
  - Condition must be in parentheses
  - No braces needed if body is single statement
- `do {...} while (c)`
  - Condition is checked at the end of the loop
  - At least one iteration

```
public class MyClass {

  ...

  public static int sumupto(int n) {
    int sum = 0;
    int i = 0;

    do {
      sum += i;
      i++;
    } while (i <= n);

    return(sum);
  }

}
```

## Conditional loops

- `while (c) {...}`
  - Condition must be in parentheses
  - No braces needed if body is single statement
- `do {...} while (c)`
  - Condition is checked at the end of the loop
  - At least one iteration
  - Useful for interactive user input

```
do {
  read input;
} while (input-condition);
```

```
public class MyClass {

  ...

  public static int sumupto(int n) {
    int sum = 0;
    int i = 0;

    do {
      sum += i;
      i++;
    } while (i <= n);

    return(sum);
  }

}
```

## Iteration

- `for (init; cond; upd) {...}`
  - `init` — initialization
  - `cond` — terminating condition
  - `upd` — update
  - Inherited from C

Iteration

Step through values in a fixed sequence

for i = 1 to n

i

A[i] - -

## Iteration

- `for (init; cond; upd) {...}`
  - `init` — initialization
  - `cond` — terminating condition
  - `upd` — update
  - Inherited from C
- Intended use is
  `for(i = 0; i < n; i++){...}`

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    int i;

    for (i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```

## Iteration

- `for (init; cond; upd) {...}`
  - `init` — initialization
  - `cond` — terminating condition
  - `upd` — update
  - Inherited from C
- Intended use is
  `for(i = 0; i < n; i++){...}`
- Completely equivalent to

```
i = 0;
while (i < n) {
  i++;
}
```

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    int i;

    for (i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```

## Iteration

- `for (init; cond; upd) {...}`
  - `init` — initialization
  - `cond` — terminating condition
  - `upd` — update
  - Inherited from C
- Intended use is
  `for(i = 0; i < n; i++){...}`
- Completely equivalent to
  ```
  i = 0;
  while (i < n) {
    i++;
  }
  ```
- Not good style to write `for` instead of `while`

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    int i;

    for (i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```

## Iteration

- Can define loop variable within loop
  - The scope of `i` is local to the loop
  - An instance of more general local scoping allowed in Java

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;
    int i;

    for (i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```

## Iteration

- Can define loop variable within loop
  - The scope of `i` is local to the loop
  - An instance of more general local scoping allowed in Java

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;

    for (int i = 0; i < n; i++){
      sum += a[i];
    }

    return(sum);
  }

}
```

local to loop

print i ? Error

```
:
{
  int x;
  float a;
  :
}
```

not visible outside

## Iterating over elements directly

- Java later introduced a `for` in the style of Python

```
for x in l:
  do something with x
```

## Iterating over elements directly

- Java later introduced a `for` in the style of Python

```
for x in l:
  do something with x
```

Python

- Again `for`, different syntax

```
for (type x : a)
  do something with x;
}
```

a is an array of "type"

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    ~~int n = a.length;~~

    for (int v : a){
      sum += v;
    }

    return(sum);
  }


}
```

## Iterating over elements directly

- Java later introduced a `for` in the style of Python

```
for x in l:
  do something with x
```

- Again `for`, different syntax

```
for (type x : a)
  do something with x;
}
```

- In this version of `for`, the loop variable must be declared in local scope

```
public class MyClass {

  ...

  public static int sumarray(int[] a) {
    int sum = 0;
    int n = a.length;

    for (int v : a){
      sum += v;
    }

    return(sum);
  }


}
```

## Multiway branching

- `switch` selects between different options

```
public static void printsign(int v) {
  switch (v) {
    case -1: {
      System.out.println("Negative");
      break;
    }
    case 1: {
      System.out.println("Positive");
      break;
    }
    case 0: {
      System.out.println("Zero");
      break;
    }
  }
}
```

## Multiway branching

- `switch` selects between different options
- Be careful, default is to "fall through" from one case to the next
  - Need to explicitly `break` out of switch
  - `break` available for loops as well
  - Check the Java documentation

```
public static void printsign(int v) {
  switch (v) {
    case -1: {
      System.out.println("Negative");
      break;
    }
    case 1: {
      System.out.println("Positive");
      break;
    }
    case 0: {
      System.out.println("Zero");
      break;
    }
  }
}
```

## Multiway branching

- `switch` selects between different options
- Be careful, default is to “fall through” from one case to the next
  - Need to explicitly `break` out of switch
  - `break` available for loops as well
  - Check the Java documentation
- Options have to be constants
  - Cannot use conditional expressions

```
public static void printsign(int v) {
  switch (v) {
    case -1: {
      System.out.println("Negative");
      break;
    }
    case 1: {
      System.out.println("Positive");
      break;
    }
    case 0: {
      System.out.println("Zero");
      break;
    }
  }
}
```

## Multiway branching

- `switch` selects between different options
- Be careful, default is to "fall through" from one case to the next
  - Need to explicitly `break` out of switch
  - `break` available for loops as well
  - Check the Java documentation
- Options have to be constants
  - Cannot use conditional expressions
- Aside: here return type is `void`
  - Non-`void` return type requires an appropriate `return` value

```
public static void printsign(int v) {
  switch (v) {
    case -1: {
      System.out.println("Negative");
      break;
    }
    case 1: {
      System.out.println("Positive");
      break;
    }
    case 0: {
      System.out.println("Zero");
      break;
    }
  }
}
```