Programming in Java

Preparatory Term, 2023
Lecture 1: Introduction and Classes

T K Srikanth
IIIT-B

Overview

Lectures organized into 4 modules:

1. Introduction and Classes
2. Modularity and Aggregation
3. Inheritance and Polymorphism
4. Generics and Collections

Books and References

1. Java: How to Program, Paul Deitel and Harvey Deitel

2. Thinking in Java, Bruce Eckel. (3rd edition available online for free download. Covers a good part of the course material)

3. The Java Tutorials: Oracle Java Documentation (useful as quick reference)

https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html

4. Java Programming: Wikibooks, ttps://en.wikibooks.org/wiki/Java_Programming

5. Effective Java, Joshua Bloch

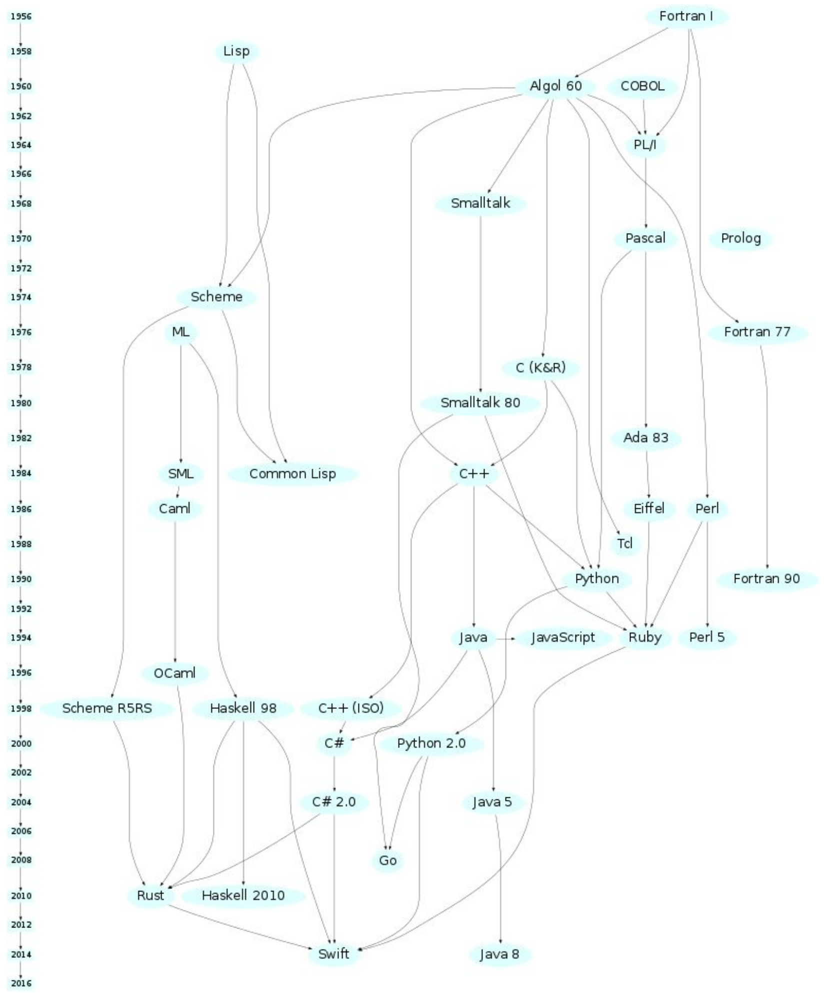
Online courses:

For those who do not have a background in Java or Object Oriented Programming, or those looking for a refresher course, below are two good online courses:

1. NPTEL: Programming in Java: Lectures 1 to 30 (https://nptel.ac.in/courses/106/105/106105191/)
2. Udacity: Object Oriented Programming in Java (Free course)

History of languages

Indicative - not necessarily complete

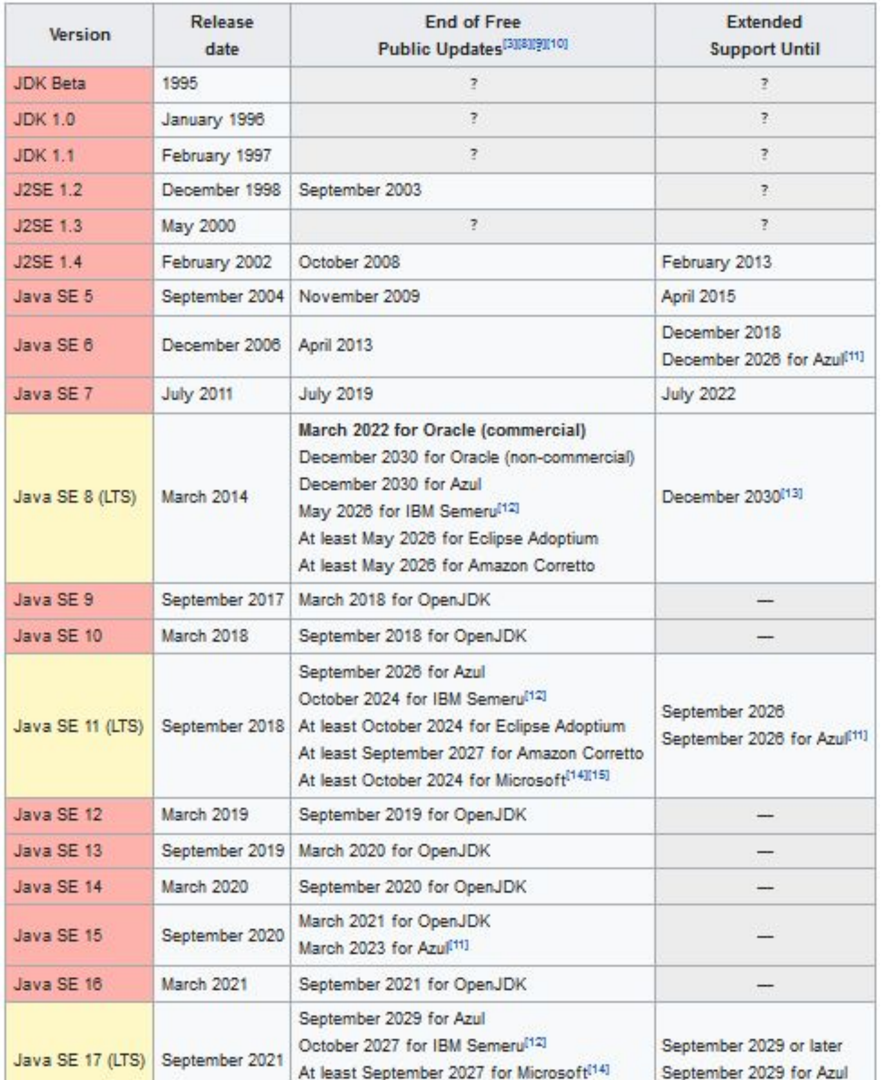


rigaux.org/language-study/diagram.html

Getting Started

Install Java Development environment (JDK) in your laptops

- “Java SE downloads” at Oracle
  - Download Java SE Development Kit - Java 11 (11.0.18) or Java 17
  - Will install the Java compiler, run-time environment and default libraries
  - or OpenJDK at https://openjdk.java.net/
- IDE such as Eclipse.



Version	Release date	End of Free Public Updates[3][8][9][10]	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	September 2003	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
Java SE 5	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018 December 2026 for Azul[11]
Java SE 7	July 2011	July 2019	July 2022
Java SE 8 (LTS)	March 2014	**March 2022 for Oracle (commercial)** December 2030 for Oracle (non-commercial) December 2030 for Azul May 2026 for IBM Semeru[12] At least May 2026 for Eclipse Adoptium At least May 2026 for Amazon Corretto	December 2030[13]
Java SE 9	September 2017	March 2018 for OpenJDK	—
Java SE 10	March 2018	September 2018 for OpenJDK	—
Java SE 11 (LTS)	September 2018	September 2026 for Azul October 2024 for IBM Semeru[12] At least October 2024 for Eclipse Adoptium At least September 2027 for Amazon Corretto At least October 2024 for Microsoft[14][15]	September 2026 September 2026 for Azul[11]
Java SE 12	March 2019	September 2019 for OpenJDK	—
Java SE 13	September 2019	March 2020 for OpenJDK	—
Java SE 14	March 2020	September 2020 for OpenJDK	—
Java SE 15	September 2020	March 2021 for OpenJDK March 2023 for Azul[11]	—
Java SE 16	March 2021	September 2021 for OpenJDK	—
Java SE 17 (LTS)	September 2021	September 2029 for Azul October 2027 for IBM Semeru[12] At least September 2027 for Microsoft[14]	September 2029 or later September 2029 for Azul

Originally designed for appliances like set top boxes (1990), then targeted to web browsers (1994), and then to general applications across a range of devices

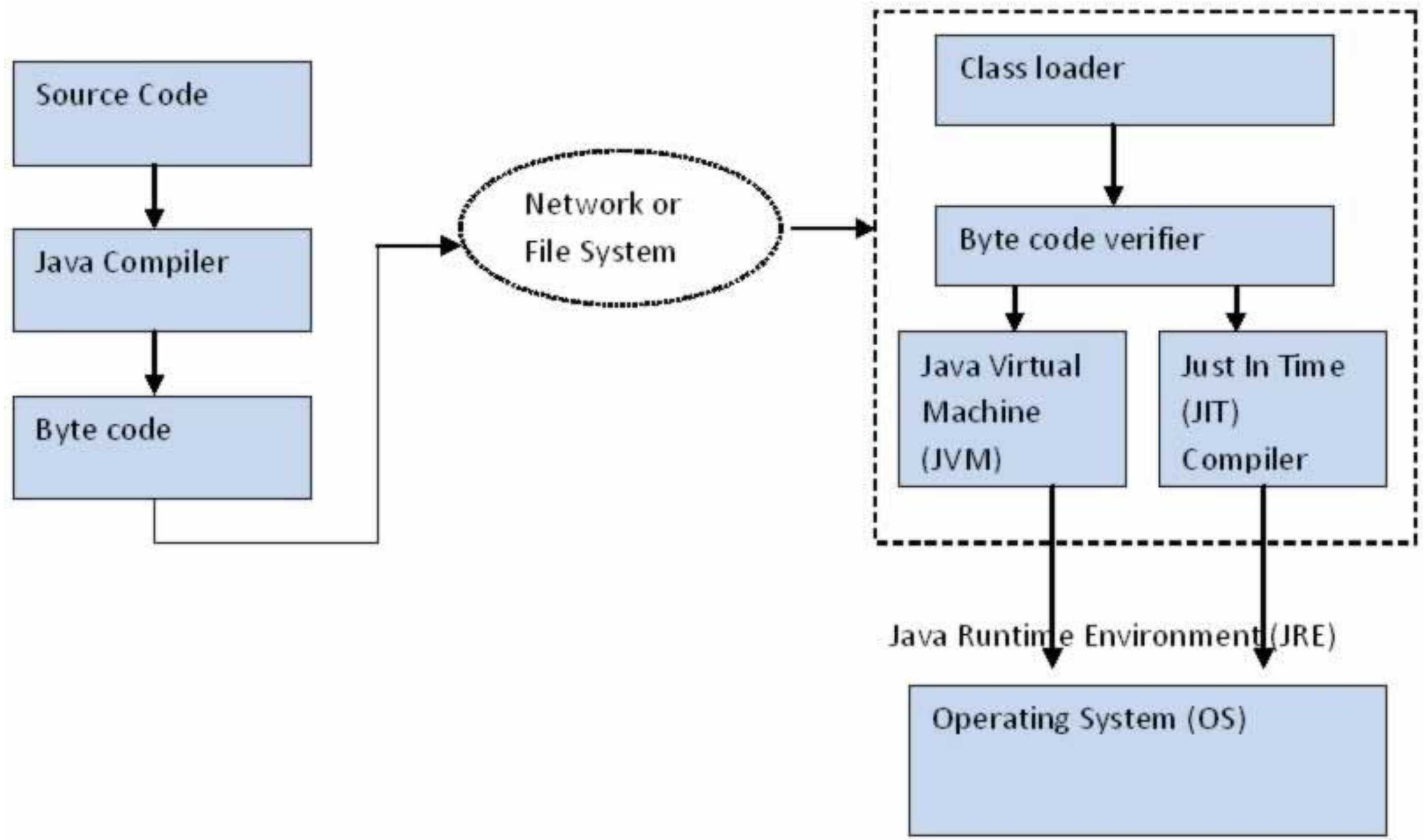
Source: Wikipedia

Java: Design Goals

“Java technology must enable the development of *secure, high performance,* and highly *robust* applications on *multiple platforms* in *heterogeneous*, *distributed networks*.” And be developer friendly

1. Simple and familiar. Leverage existing languages
2. Object-oriented. Standard set of APIs for basic and advanced capabilities
3. Robust and secure. Avoid ability to directly manipulate hardware/memory, strong type checking, make it difficult for other programs to impact a Java program (and vice versa)
4. Architecture neutral and portable: work on multiple hardware platforms, architectures and OS. “Write once, run anywhere”
5. High performance. Should not have performance challenges
6. Interpreted and dynamic. Ability to not have to go through compiling, global linking etc. Can ease the prototyping phase - quick edit/compile/link/run
7. Multi-threaded. Be able to multi-task and do multiple things at the same time.
8. Distributed computing. Run across machines, including across networks

Byte code and compilation process



Hello World!

```
public class HelloWorld{
	public static void main(String[] args) {
		System.out.println(“Greetings!”);
	}
}
```

File: HelloWorld.java

Only one public class per file

Can we have main in multiple classes/files? Why would we do that?

Compilation and running

```
> javac HelloWorld.java
```

produces .class file(s)

Automatic compilation of referenced classes

```
> java HelloWorld
```

Runs the “main” in this class

Logical collection of classes/libraries can be packed into “jar” files

Java Class Library

- Provide the programmer with a well-known set of functions to perform common tasks, such as maintaining lists of items or performing complex string parsing.
- Provide an abstract interface to tasks that would normally depend heavily on the hardware and operating system.
- Some underlying platforms may not support all of the features a Java application expects. In these cases, the class libraries can either emulate those features using whatever is available, or provide a consistent way to check for the presence of a specific feature.

Types in Java

- primitives
- reference
- Array
- (a special type) void

Primitive types

Fixed size of basic types

boolean true, false (not equivalent to 0 or 1)
char 16 bits (unicode)
byte 8 bits
short 16 bits
int 32 bits
long 64 bits
float 32 bits
double 64 bits

all numeric types are “signed”. No “unsigned”
Note: primitives <u>**are NOT**</u> objects

Incidentally, Java does not have a *sizeof* method/operator. Why?

Also, the perils of “data conversion”:

A Bug and a Crash, James Gleick, 1996 ,
https://en.wikipedia.org/wiki/Ariane_flight_V88

“The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. “*

Java operators and control statements

Operators:

- Generally follow the syntax of C/C++
- No operator overloading

Control statements:

- If-else, while, do-while, for, switch: similar to C/C++
- for: variations that allow simple iteration over arrays, lists etc

for loop - additional syntax

```
for( T elem: <<array of T>>) {
 // elem is successively bound to each element of the array
}
```

E.g.

```
int[] scores = new int[400];
… initialize scores
int total = 0;
for (int score: scores) {
   total += score;
}
```

Useful when iterating through an array, but not modifying array or its elements

switch statement

variable being tested in switch can evaluate to constants of byte, short, char, and int primitive data types

Can also be used with enum

As well as String

Objects and references

- Everything is an “object” - an instance of a “class”
- You manipulate objects through their references
- Objects (state) may change, but references do not
- Methods are passed references to objects (or primitives), so always “call by value”
- Objects are allocated on the heap, references are on the program stack or heap (depending on whether they are local variables or members of objects)
- Memory of objects are “garbage collected” when no longer referenced (How?)

Initialization and defaults

Any variable **must** be initialized before its value is used

- Compiler error if not

All data members of classes are initialized to their “default” value at construct time

- equivalent of “zero”/null/false as appropriate

String and Array

Both are **classes**

Specific methods, constructors, operations

String: immutable in size and content - can only be queried

Operators such as concatenation (+) defined.

Array:

Size defined at construct time

Can be queried for length (arr.length)

Class definition

```
class IceCreamBar {

  String getName() {   }

  String getFlavour() {   }

  float getTemp() {   }

  float getPrice() {   }

  ….

}
```

Methods

Class definition

```
class IceCreamBar {

  String getName() {   }

  String getFlavour() {   }

  float getTemp() {   }

  float getPrice() {   }

  String name, flavour;

  float temp, price;

}
```

Methods

Data members

Class definition

```
class IceCreamBar {

  String getName() {   }

  String getFlavour() {   }

  float getTemp() {   }

  float getPrice() {   }


  private String name, flavour;

  private float temp, price;

}
```

Methods

Data members

Class definition

```
class IceCreamBar {
```

Constructor

```
  IceCreamBar(String iname, String iflavour,

              float itemp, float iprice) { }

  String getName() {  }

  String getFlavour() {  }

  float getTemp() {  }

  float getPrice() {  }
```

Methods

```
  private String name, flavour;

  private float temp, price;
```

Data members

```
}
```

Class definition

```
class IceCreamBar {

  IceCreamBar(String iname, String iflavour, float itemp, float iprice) { }

  String getName() { return name; }

  String getFlavour() { return flavour; }

  float getTemp() { return temp; }

  float getPrice() { return price; }

  …

}
```

Constructors

Every class should have at least one constructor

Constructor that takes no arguments is called the *default constructor*

Compiler defines a default constructor <u>if no</u> constructors have been defined for the class. This default version sets all data members to their default initial value.

All instance variables are initialized to default values, if not explicitly initialized in a constructor

If <u>any constructor</u> is defined for the class, then the compiler <u>does not define</u> the default constructor

Note: no notion of "*destructors*". Facility for cleanup

Class definition

```
class IceCreamBar {
  IceCreamBar(String iname, String iflavour, float itemp, float iprice) {
       name = iname; flavour = iflavour;
       temp = itemp;
       price = iprice;
  }
  String getName() { return name; }
  String getFlavour() { return flavour; }
  float getTemp() { return temp; }
  float getPrice() { return price; }
…
}
```

Class definition

```
class IceCreamBar {
  IceCreamBar(String iname, String iflavour, float itemp, float iprice) {
       name = iname; flavour = iflavour;
       temp = itemp;
       price = iprice;
  }
  String getName() { return name; }
  String getFlavour() { return flavour; }
  float getTemp() { return temp; }
  float getPrice() { return price; }
  void setTemp(float t) { temp = t;}
  void setPrice(float p) { price = p;}
  ...
}
```

Invoking class methods

```
IceCreamBar ic1 = new IceCreamBar(“CH”, “Chocolate”, 8.0, 90.0);

….

System.out.println(“Flavour “ + ic1.getFlavour() +
                  “ from “ + ic1.getName() +
                  “costs “ + ic1.getPrice() +
                  “ and is stored at “ + ic1.getTemp() + “ degrees.”);
```

Should print:

```
Flavour Chocolate from CH costs 90.0 and is stored at 8.0 degrees.
```

Method signatures

The name and parameter list (arguments and their types) defines the *signature* of a method

Is return type part of the signature of a method?

Method overloading

- we can have multiple methods in a class with the same name, but with different arguments. E.g. constructors with different sets of arguments

When an overloaded method is invoked, the compiler maps this to that method whose signature most closely matches the invocation

- Number and order of arguments must match, and the types of arguments should be “compatible” (to be discussed later)

Primitive wrapper classes

primitive variables are not objects - i.e. not instances of some class

Java provides “wrapper” classes corresponding to each primitive type: Byte, Short, Integer,, Float, Double, …

Compiler supports automatic conversion between primitives and their corresponding wrapper classes: auto boxing and unboxing (like an implicit cast).

- Passed as a parameter to a method that expects an object of the corresponding wrapper class.
- Assigned to a variable of the corresponding wrapper class.

Auto boxing and unboxing

```
    int i=3;
    Integer j = i;
    int k = j;
------------------

void m1(Integer i, Character c) {}
```

Can be invoked as:

```
    int i=3;
    char c = ‘a’;
    m1(i, c);
```

Scope and lifetime (of primitives, references, objects)

The scope of primitives and references defined by the block where they are declared. And so is their lifetime (memory no longer available once out of scope)

```
{
	int x = 7;
	Box b1 = new Box();
	...
}
```

```
Box b;
{
	int x = 7;
	Box b1 = new Box();
	b = b1;
}
```

Objects live on independent of the block where they were constructed. Accessible as long as some reference to them is still alive.

Memory Management

- Objects are allocated on the heap, references are on the program stack or heap (depending on whether they are local variables or members of objects)
- Memory of objects are “garbage collected” when no longer referenced (How?)

static methods and data members

A method specified as **static** is considered as a method defined on the class, and not tied to any instance.

Can be invoked even without creating instances of that class

static methods cannot access data (or methods) that are non-static - even of the same class

Similarly, static data members are shared by all instances of that class - there is only one piece of storage associated with that data member.

Static methods/members are accessed with

classname.member

static methods

```
class Account {
  …
  static Account max(Account[] acs) {}

   private String name;
   private float balance;
}
```

Usage:

```
Account[] accounts = new
Account[10];

// …. Iniitalize array

Account largest =

  Account.max(accounts);
```

static data fields

```
class Account {
  …
   Account() {
        accountNumber = nextId++;
  }
  static Account max(Account[] acs) {}
  ...
   private static int nextId = 1;
   private String name;
   private float balance;
   private int accountNumber;
}
```

final variables

A variable labelled as **final** implies value cannot be changed once it is initialized

final variables must be initialized

- when declared
- or, in every constructor (also called “blank final”)
- Compiler error if attempt to re-assign a value

Note: if a reference to an object is final, the reference itself cannot change, however, the object it refers to can change.

For example, in the IceCreamBar example, the name and flavour should be defined final, if the intent is that these cannot change once the object is initialized.

Writing to Standard output

System.out - a pre-defined instance of PrintStream, defined in class System

```
String s1 = "Hello ";

int year = 2022;

System.out.println("myString is: " + s1);

System.out.println("year is: " + year);
```

Reading from Standard Input

System.in - a pre-defined instance of InputStream, defined in class System

```
Scanner scanner = new Scanner(System.in); // or new Scanner(“filename”);

String myString = scanner.next();

int myInt = scanner.nextInt();

while (sc.hasNextLong()) {

          long aLong = sc.nextLong();

      }
scanner.close();
```

I/O mechanisms

Stream type		Unbuffered	Buffered	Formatted
Byte		InputStream/FileInputStream OutputStream/FileOutputStream	BufferedInputStream BufferedOutputStream	PrintStream/ PrintWriter
	Character	Reader/FileReader Writer/FileWriter	BufferedReader BufferedWriter	Scanner Formatter
Data			DataInputStream DataOutputStream	
Object			ObjectInputStream ObjectOutputStream	