

# ESS 201: Programming II

## Java

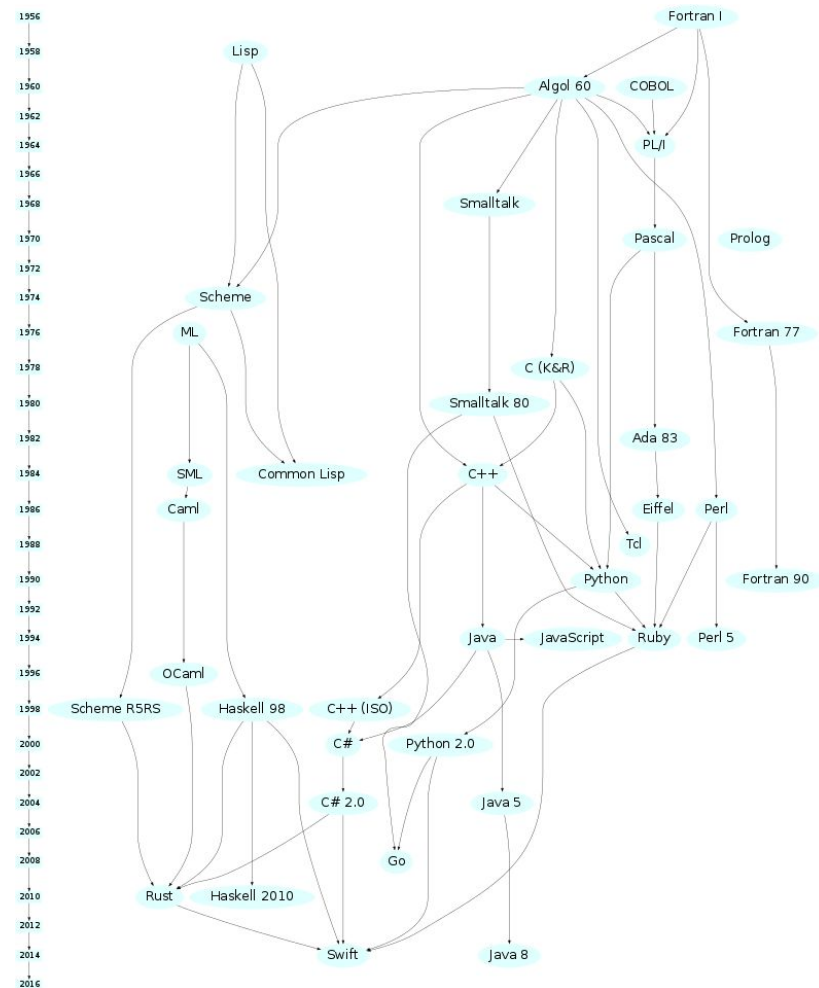
Term 1, 2019-20

Lecture: Language features

T K Srikanth  
IIIT-B

# History of languages

Indicative - not necessarily complete



# Code size trends

How big (lines of code) is a typical program?

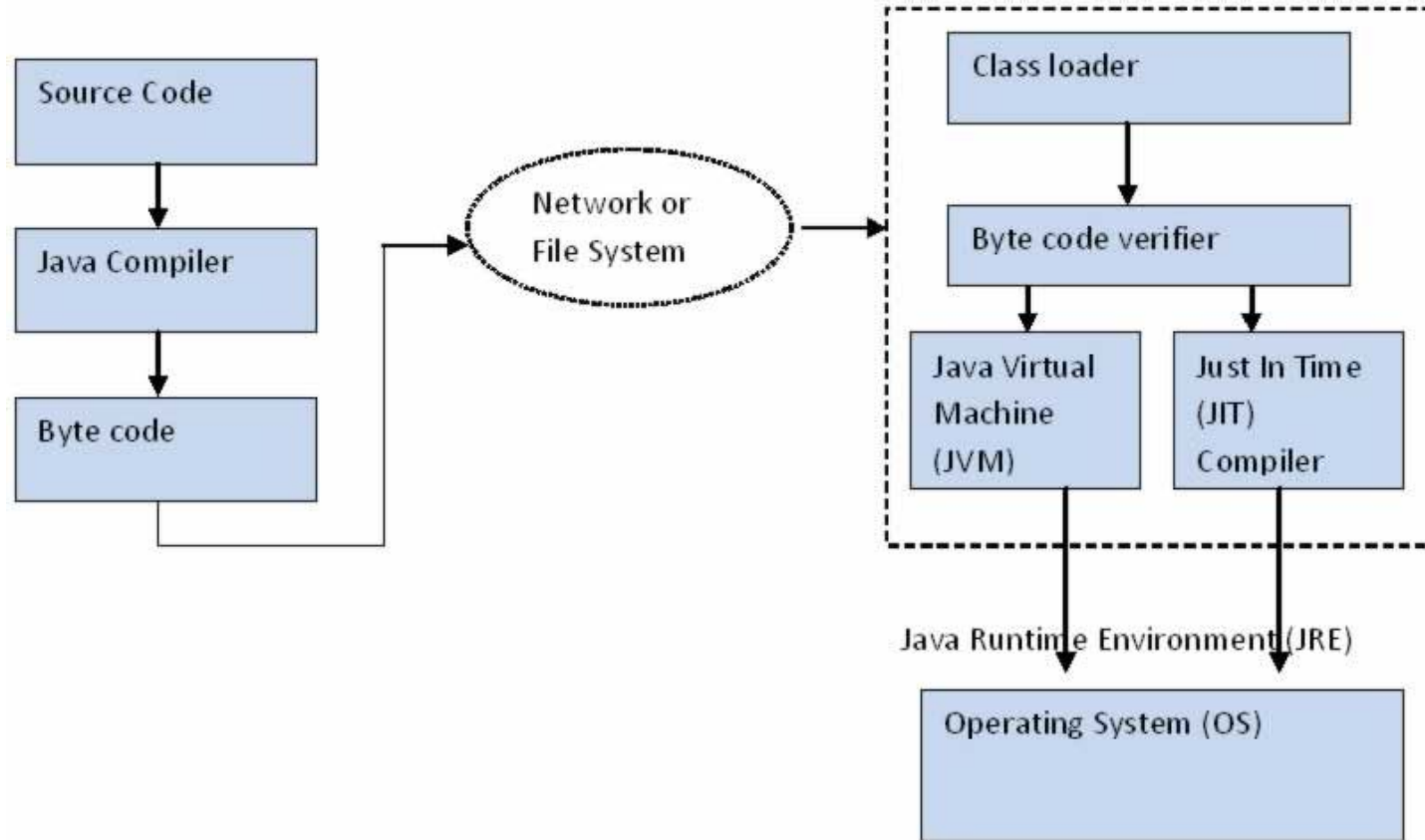
<https://informationisbeautiful.net/visualizations/million-lines-of-code/>

# 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



Version	Release date	End of Public Updates <sup>[4]</sup>
JDK Beta	1995	?
JDK 1.0	1996	?
JDK 1.1	1997	?
J2SE 1.2	1998	?
Java SE 8 (LTS)	2014	January 2019 (commercial) December 2020 (non-commercial)
Java SE 9	2017	March 2018
Java SE 10 (18.3)	2018	September 2018
Java SE 11 (18.9 LTS)	2018	N/A

Source: Wikipedia

# 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?)

# 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 **are NOT** objects



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

An example of why it is important to understand the size of data types

Or, the perils of “data conversion”:

A Bug and a Crash, James Gleick, 1996

<https://around.com/ariane.html>

# 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

# 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();  
    ...  
}
```

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

# **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 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];  
  
// .... Initialize 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.

# Method signatures

The name and parameter list (arguments and their types) defines 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)
- Is the return type part of the signature of a method?

# Hiding the implementation

*“separating the things that change from the things that stay the same.”* - Bruce Eckel, Thinking in Java

The public methods that are used to create/access/modify objects should largely remain the same, although the implementation (including any other classes internal to this implementation) will likely change over time.

Access specifiers are one way to enable this separation.

# Access specifiers

classes, methods and data fields can have specifiers added to them that control who all (which other classes) can access them

**public**, **private** (and later, **protected**, as well as a *default* (i.e. no specifier)) access

**public**: visible to all other classes

**private**: only visible from within (methods or class definition) of the same class

Does a *private class* make sense?

# Public/private members

Visibility of members (methods or data) of a class A from within class B:

Specifier (in class A)	For class A	For class B
public	Y	Y
private	Y	N

# Some rules (best practices)

1. All data members are private
2. No casts\*
3. No compiler warnings

\* except is very specific situations - to be discussed later