Basics of programming 3

Java language basics

Goldschmidt Balázs

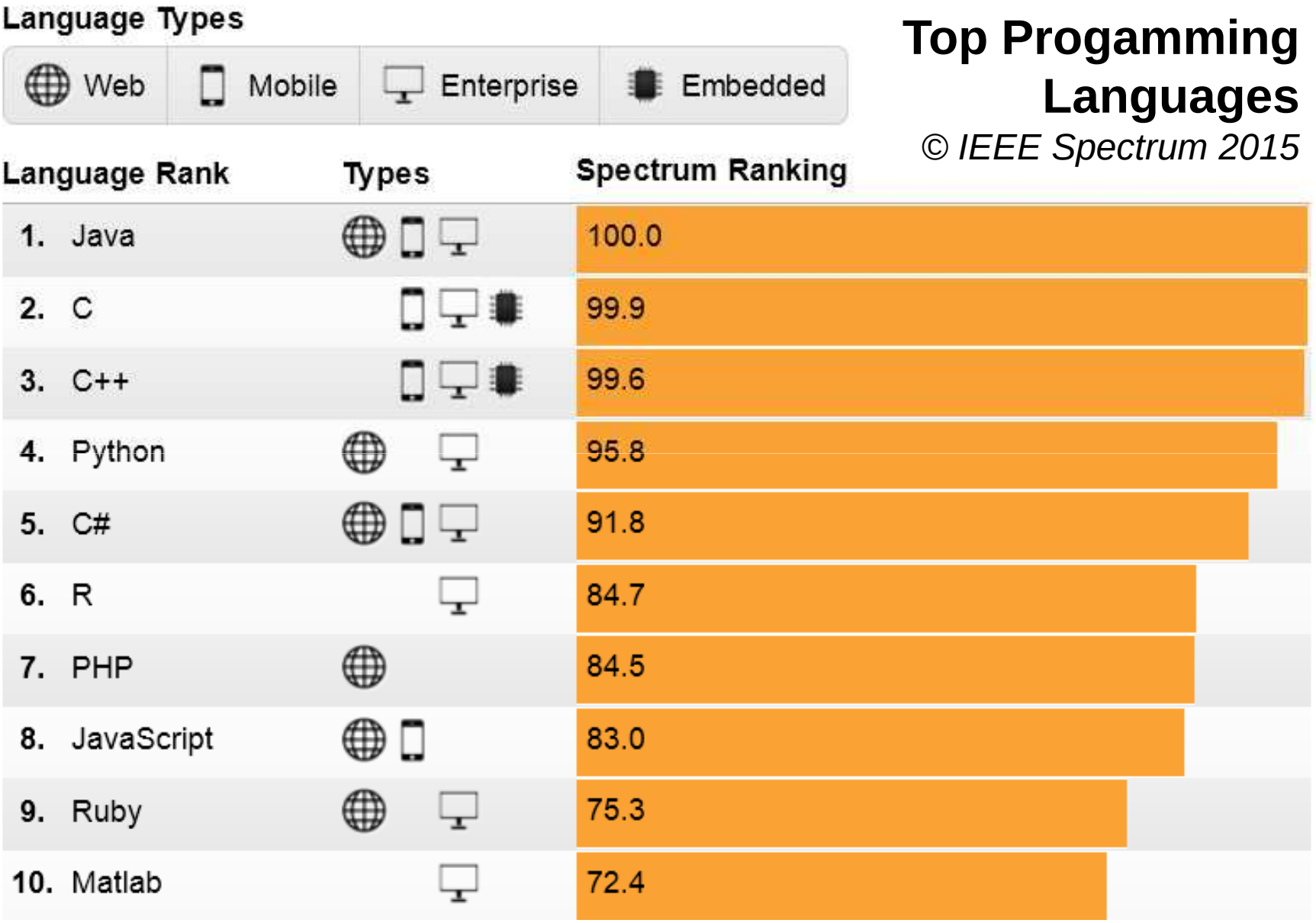
balage@iit.bme.hu

Basics of programming courses

- BoP 1: *Structural programming*
  - Variables, control, functions, data structures, etc
  - Language: *C*
- BoP 2: *OO concepts*
  - Classes, encapsulation, inheritance, polymorphism, etc
  - Language: *C++*
- BoP 3: *OO development using APIs*
  - I/O, collections, multithreading, graphics, unit tests, etc
  - Language: *Java*

Top Progamming Languages

© IEEE Spectrum 2015

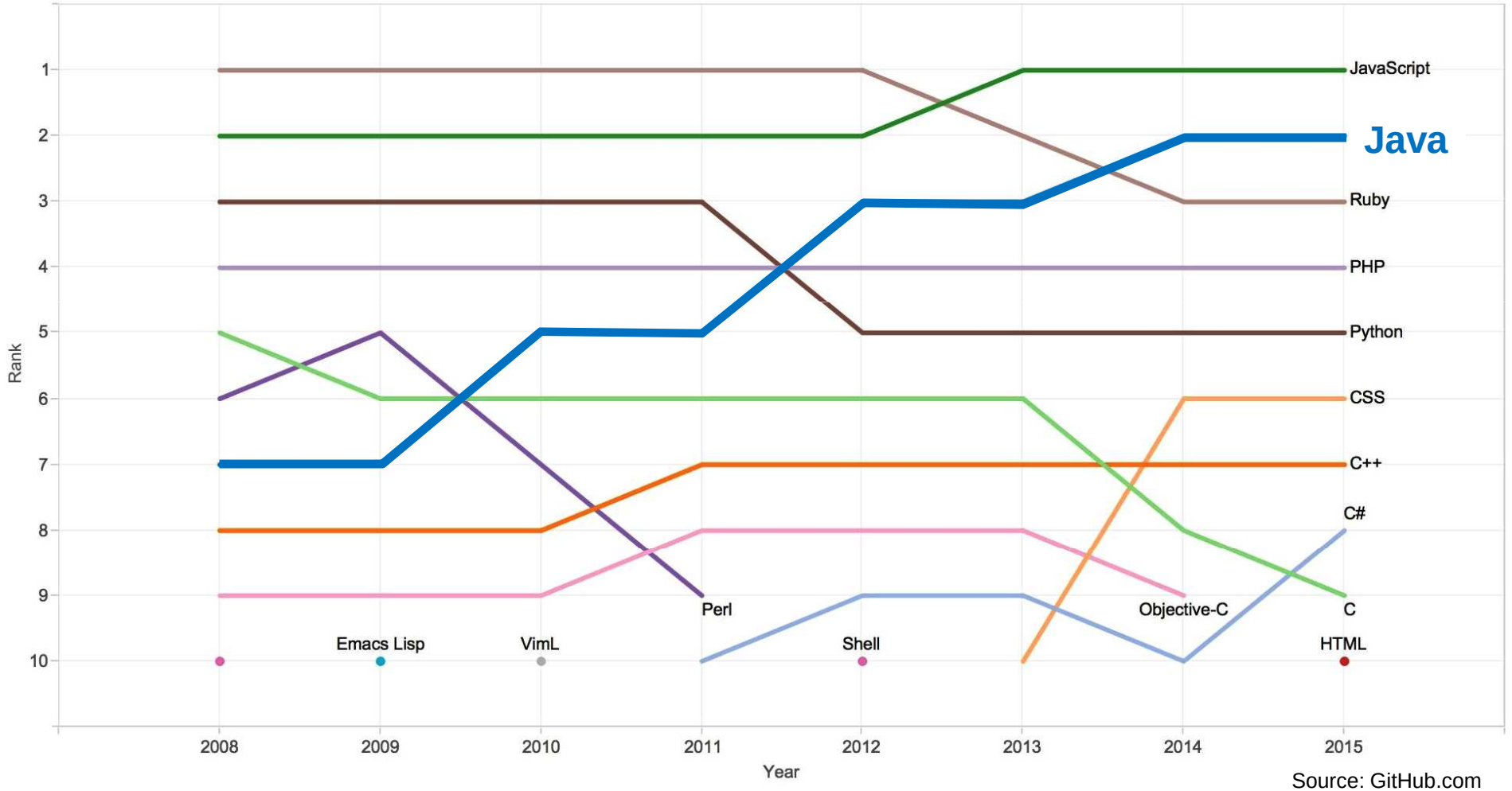


Followed by

- **11-19. (*71.4-62.4*):** Shell, SQL, Assembly, Go, Perl, Visual Basic, Arduino, Scala, Swift
- **20-26. (*58.7-43.7*):** Objective-C, HTML, Processing, Cuda, Lua, D, SAS
- **27-35. (*40.4-30.6*):** Lisp, Fortran, Delphi, Haskell, VHDL, Ada, Clojure, LabView, Erlang
- **36-40. (*29.6-21.8*):** Rust, ABAP, Verilog, Prolog, Ladder Logic
- **41-48. (*19.0-0.0*):** Cobol, Julia, Scheme, TCL, Forth, J, Actionscript, Ocaml

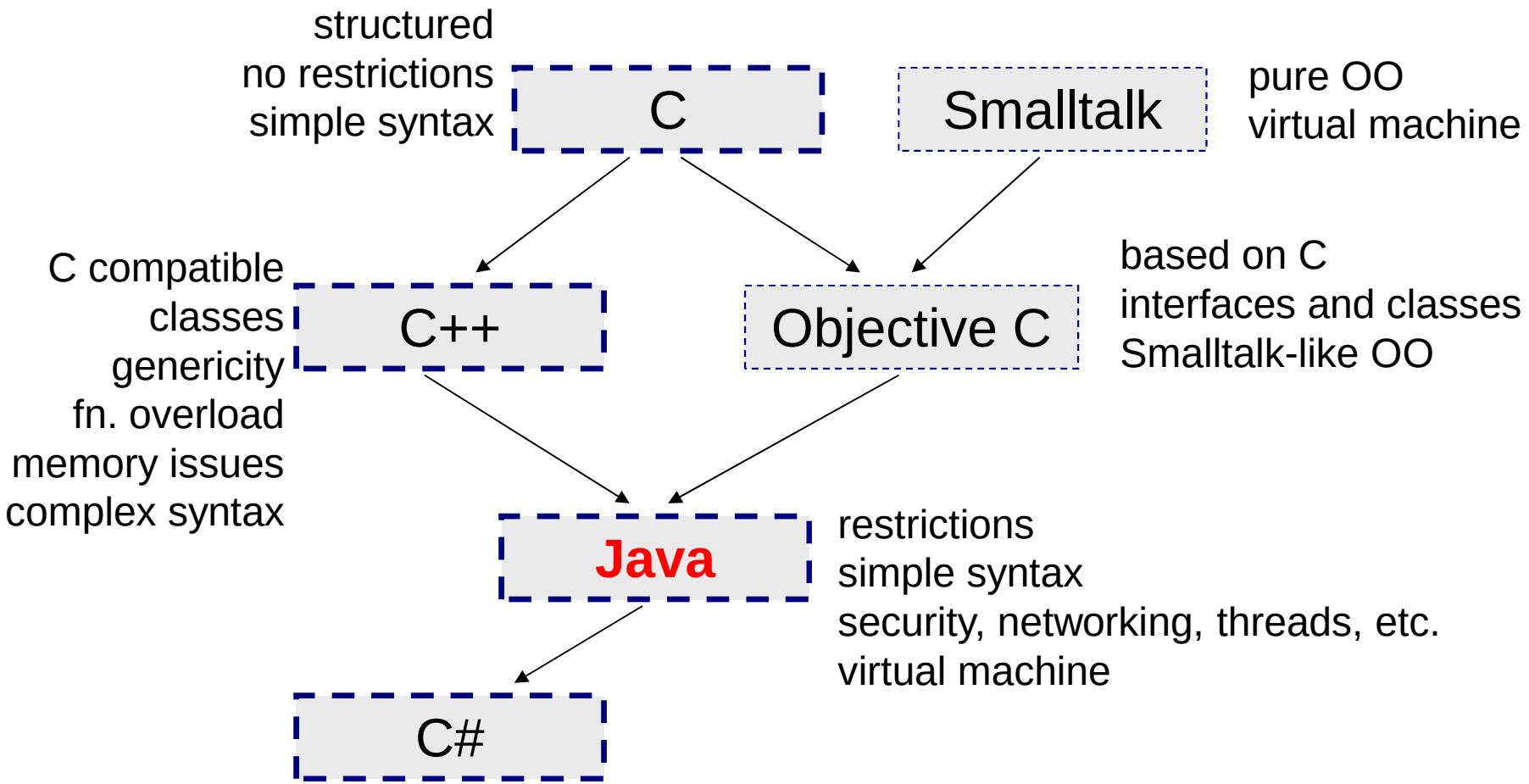
source: http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2015

GitHub top coding lang’s



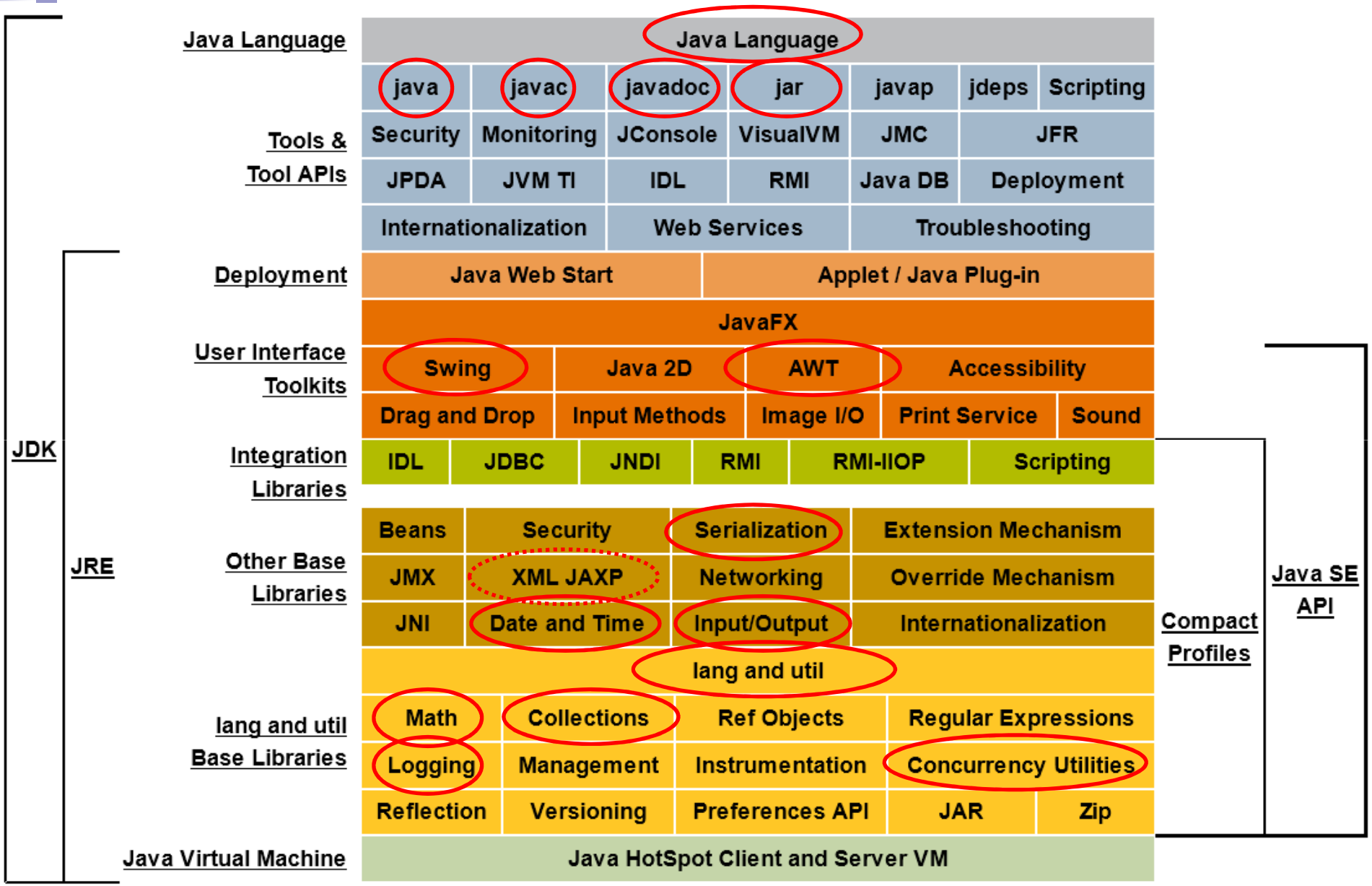
Source: GitHub.com

Geneology of Java



J2SE framework

- Java is like C
  - simple syntax
  - huge API
- Java programming is like playing lego
  - putting together already existing building blocks
  - everything is implemented
    - usually better than we could do it
  - real knowledge is that of the API
  - versions differ in API and syntax
    - latest major version: 8 (2014-03-18)


Java Language
Java Language
java
javac
javadoc
jar
javap
jdeps
Scripting
Tools &
Tool APIs
Security
Monitoring
JConsole
VisualVM
JMC
JFR
JPDA
JVM TI
IDL
RMI
Java DB
Deployment
Internationalization
Web Services
Troubleshooting
Deployment
Java Web Start
Applet / Java Plug-in
JavaFX
User Interface
Toolkits
Swing
Java 2D
AWT
Accessibility
Drag and Drop
Input Methods
Image I/O
Print Service
Sound
Integration
Libraries
IDL
JDBC
JNDI
RMI
RMI-IIOP
Scripting
JDK
JRE
Other Base
Libraries
Beans
Security
Serialization
Extension Mechanism
JMX
XML JAXP
Networking
Override Mechanism
JNI
Date and Time
Input/Output
Internationalization
lang and util
lang and util
Base Libraries
Math
Collections
Ref Objects
Regular Expressions
Logging
Management
Instrumentation
Concurrency Utilities
Reflection
Versioning
Preferences API
JAR
Zip
Java Virtual Machine
Java HotSpot Client and Server VM
Compact
Profiles
Java SE
API

Java basics

- Everything is a class or object
  - no global functions
  - application structure:
    - packages > classes > methods and variables > statements
- Two kinds of types
  - primitive (int, double, boolean, …)
    - variable stores value
  - object (String, Vector, …)
    - variable stores reference

Java basics 2

- Syntax very similar to C/C++
  - operators (+,-, >>, …)
  - control structures (for, while, switch)
  - method call
- But
  - *no pointers*
  - *no goto*
  - *no operator overloading*
  - *separate* byte*,* char*, and* boolean *types*

Java basics 3

- Arrays are objects
  - length → run-time check

```
int a[] = new int[10];
//int[] a = new int[10]; // also OK
for (int i = 0; i < a.length; i++) {
      a[i] = i*2;
}
```

- Only pass by value
  - no pointer arithmetics
- Garbage collection
  - no delete

Hello world

```
// C/C++

int main(int argc, char** argv) {
    printf("Hello world\n");
}
```

```
// Java (Hello.java)

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

Compiling and running

- Rule of thumb:
  - for each class separate source file
    - `class Hello` → Hello.java
  - for each class separate bytecode (class) file is generated
  - `> javac Hello.java` → Hello.class
- JVM starts the *main* method of the selected class
  - `> java Hello`

Write Once, Run Anywhere

- C, C++, etc:
  - *write once, compile everywhere*
- Java:
  - source compiled into *bytecode*
  - bytecode run by virtual machine
  - no need for recompilation when migrating
- write once, debug everywhere
  - good design is important
  - it is still easy to create platform-specific application

Starting Java applications

- Simple run
  - needs command prompt or batch file
- Jar file
  - special zip file with manifest
  - *“starts when clicked”*
- Applet
  - embedded into a webpage
  - restricted functionality and permissions
  - *flash* predecessor
- Java Web Start
  - pl. NAV website

Basic types, operators, statements

Primitive types and variables

- Primitive types
  - boolean
  - char (16bit unicode)
  - byte, short, int, long (8, 16, 32, 64 bit signed integer)
  - float, double (32 and 64 bit real)
- Variable declaration and definition
  - similar to C and C++

```
int a = 13;
double d = f = 3.14;
```

Complex types

- Arrays and objects are complex
  - String, Vector, etc.
- Variable stores reference
  - resembles C++ pointer
  - no pointer arithmetic
- Assigning to variable
  - discards former reference

```
String s = “12345”;
s = “hello”; // former value discarded
```

Arrays

- Simple arrays

```
int a[] = new int[13];
double[] d = new double[20];
```

- Multidimensional arrays
  - arrays of arrays

```
int[][] a = new int[10][20];

int[][] b = new int[4][];
for (int i = 0; i < b.length; i++) {
    b[i] = new int[i*2];
}
```

Operators

- Same operators as in C/C++
  - same precedence and association rules
  - logical operators only for logical expressions
    - no logical-integer mix-up
- Removed operators (not in Java)
  - `delete`, `->`
- New or modified operators
  - >> (sign is shifted)
  - >>> (0 is inserted from left)
  - non-lazy logical operators: &, |, ^

Statements

- Similar to C/C++
  - if-else, while, do-while, for, switch-case
    - `if`, `while`, `for` (2nd expr) need logical expression
    - (Java 7: `case` for *strings* also)
  - continue, break, return
    - labels can be used for break and continue

```
int i = 1;
loop: while (i < 100) {
    for (int k = i; k < 300; k++) {
        if (k == i*2) break loop;
    }
}
```

  - no *goto*

Objects, Classes and Interfaces

Classes

- Resembles C++
  - minor and major differences
- Differences from C++
  - visibility also on class level (packages)
  - visibility separately for each attribute and method
  - attributes get default value (0, null, etc)
  - only “inline” methods
  - all methods virtual
    - private methods are hidden
  - no operator overloading

Classes 2

- Differences from C++ cont.
  - only object's reference is passed
    - no copy constructor
  - no initialization list
  - no default parameters
  - no multiple or virtual inheritance
  - *this* also for constructor call
  - destructor is *finalize()*
  - reference resembles C++ pointer, not C++ reference

Classes example

```
public class Something {
   int a; // package visibility
   private double d;
   protected long l;
   public String s;

   public Something(int a) {
       this.a = a;
   }

   public Something() {
       this(10);
       l = 14l;
   }
   // ...
```

Classes example cont.

```
    // ...
    public void finalize() {
        ...
    }

    private void increment(int i) {
        a += i;
    }
    public long add(int i) {
        increment(i);
        l += i;
        return l;
    }
}
```

Classes example cont.

```
// somewhere in a class....

public static void main(String[] args) {

   // parenthesis is mandatory for ctr-s
   // s holds reference to object
   // NO '*' operator!
   Something s = new Something(5);

   // field access by .
   // NO '->' operator!
   long f = s.add(34);

}
```

Field modifiers

- private
  - same as C++: access from same class only
- *package* (no modifier, “default-access”)
  - not in C++: *access from same package only*
- protected
  - similar to C++: access from subclasses and same package
- public:
  - same as C++: access from anywhere

Field modifiers cont.

- static
  - same as C++: *class-level attribute or method*
- final
  - not in C++:
    - for methods: subclasses *must not* override
    - for variables: like C++ *const*
- abstract
  - for methods and classes only
  - same as C++ pure virtual: no implementation, subclasses must implement
  - if method is abstract, class must be abstract too

Static members

- Static members similar to C++
  - static members can only access static members directly
  - static members can be accessed by non-static methods
- Variable initialization

```
class A {
   static long l = 13; // inline
   static long k;
   static { // initialization block
        k = 15; // run when class is loaded
   }
}
```

String: a special class

- Provides usual string operations
  - `length()`, `equals()`, `startsWith()`
  - `substring()`, `trim()`, `split()`, `concat()`
  - `toUpperCase()`, `toLowerCase()`, `replace()`
  - `charAt()`, `indexOf()`, `lastIndexOf()`
  - `valueOf()`
  - ...
- Only class with + and += overloaded
  - concatenation, not efficient
- Immutable
  - object's state doesn't change

Inheritance

- Syntax different from C++
  - *extends*

```
class A {...}
class B extends A {...}
```

  - use *super()* for calling superclass' constructor
- Semantics different from C++
  - all methods virtual
  - no multiple inheritance for classes
  - topmost superclass: *Object*
  - constructors initialized differently

Inheritance example

```
class A {
   int k;
   public A() { k = 13; }
   public A(int i) { k = i; }
   public void foo() { System.out.println("A"); }
   public void bar() { foo(); }
}

class B extends A {
   public B() {}
   public B(int j) { super(j); }
   public void foo() { System.out.println("B"); }
}
```

Constructor tasks

- Creating object structure
  - attribute initialization to 0
  - initialization of virtual function tables
- Initializing superclasses
  - ...
- Initializing class
  - explicit attribute initialization
  - initialization block *(i.e. a stand-alone block)*
  - constructor as invoked

Constructor tasks

```
class A {
   int k,l;
   { k = 20; } // init. block
   public A() { l = 13; }
   public void foo() { System.out.println("A"); }
}

class B extends A {
   public B() {}
   public void foo() { System.out.println("B"); }
}
```

`Object` superclass

- Topmost superclass
- Methods
  - **`boolean equals(Object o)`**
    - for content based equality *(default impl. reference based)*

      `a == b` vs. `a.equals(b)`
  - **`int hashCode()`**
    - hash code generation for collections
  - **`void finalize()`**
    - like C++ destructor, called by garbage collector

`Object` superclass 2

- Methods cont.
  - **`String toString()`**
    - returns string representation
    - mostly for debugging
    - called where String is needed

```
"my car: "+myCar+";"
```

  - **`Object clone()`**
    - returns a copy of the object *(always of the bottommost class)*
    - *Cloneable* interface

Interfaces

- Like classes, but no implementation
  - each interface into a separate file
- Methods only declared, always implicit public
  - no implementation is specified
- May have attributes
  - automatically *public static final* (global constant)

```
interface A {
   void foo();
   int bar(String s);
   public static final int maxLength = 100;
}
```

Interfaces 2

- Multiple inheritance of interfaces is supported
  - only if no ambiguous attributes
- Class can implement multiple interfaces
  - *implements* keyword

```
class A extends B implements C, D {}
```

- Class doesn't have to implement all methods
  - must be abstract class

Interface example

```
interface A {
   void foo();
   int bar(String s);
}
abstract class B implements A {
   ...
   public void foo() { System.out.println("B"); }
   abstract public int bar(String s);
}
class C implements A {
   ...
   public void foo() { System.out.println("B"); }
   public int bar(String s) { return s.length();}
}
```

Packages and wrapper classes

Packages

- Provide hierarchical namespace
  - like *namespaces* in C++
- Package hierarchy with corresponding directories (folders)
  - same name, same hierarchy
- Classes and interfaces
  - source code must specify the packages
    - `package foo.bar.baz;`
  - source file must be put into the folder of the package

Packages and class names

- Full name
  - `foo.bar.baz.MyClass`
- Importing names
  - only classes and interfaces
  - similar to *using namespace X*
  - specifies packages to be searched for identifiers
  - if colliding, full names must be used
    - e.g. *List* is part of *java.util* and *java.awt*
  - static import for fields

```
import foo.bar.baz.*;
import mypack.MyClass;
```

Wrapper classes

- For each primitive type a class is defined
  - Integer, Byte, Boolean, etc.
  - sometimes an object is needed
- Constructors
  - `Integer(int i)`, `Integer(String s)`
- Constants
  - MAX_VALUE, MIN_VALUE, SIZE
  - NaN, NEGATIVE_INFINITY, etc.

Wrapper classes and boxing

- Transformations
  - rotateLeft, reverse, signum, etc.
- Conversion methods
  - **intValue()**, **longValue()**, **parseInt(String s)**, **valueOf(...)**
- Wrapper ↔ primitive conversion is automatic

```
int a = 2;
Integer b = 3;
a = a + b;
Integer d = 3+3;  // int -> Integer
System.out.println(d*3); // Integer -> int
```

Wrapper classes and boxing 2

- Conversion is not efficient

```
public static void main(String args[]) {
	Integer i1 = Integer.parseInt(args[0]);// boxing
	Integer i2 = Integer.parseInt(args[0]);// boxing

	i1.equals(i2);		// true, no boxing
	i1 == i2;		// only if -128<i1≤127
	i1 <= i2;		// true, boxing
	i1++;			// incr, boxing
	i1 > 120;		// boxing
}
```

Memory handling

- **C**: memory problems
  - pointers + arithemtics
  - void*
  - malloc/calloc/realloc/free

```
a[3] ≡ *(a+3) ≡ *(3+a) ≡ 3[a]
```

- **C++** tries to overcome problems, but fails
  - copy constructor
  - vitrual destructor
  - assignment
  - new/delete

```
class C : A, virtual B {
  int l; Complex c;
public:
  C(Complex k, int i)
    : A(i), c(k), l(i)
      { l++; }
};
```

Memory handling 2

- Java has a built in Garbage Collector (GC)
  - **`new`** : allocates on heap
  - **`delete`**: not explicitly, GC frees
- GC deletes objects with no reference
  - **`void finalize()`** is called
- Starting GC explicitly:
  - **`System.gc()`** or **`Runtime.gc()`**

Exception handling

Exceptions in **C++**

- Classic exception handling

```
try {
  ...
} catch (E e) { // E type exception
} catch (...) { // everzthing else
  ...
  throw;
}
```

- No declared exception type
  - STL introduced class *exception*

Exceptions in **`Java`**

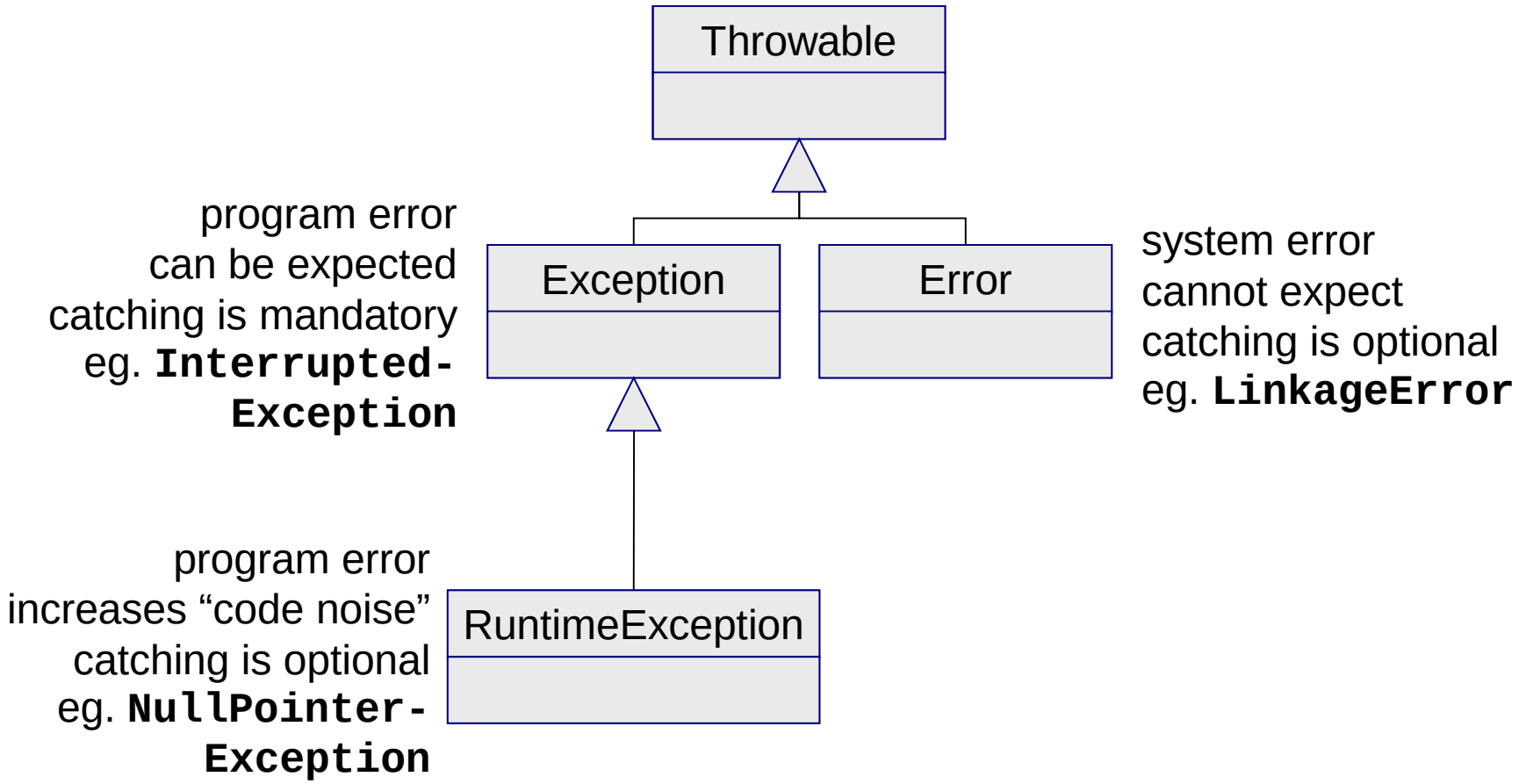
- based on C++
- common exception superclass
  - **`Throwable`**
- all possible exceptions must be declared
  - **`throws`**
- superclasses for different error types
  - **`Exception`**, **`RuntimeException`**, **`Error`**
- closing block: **`finally`**
- see also: *try with resources* (Java7)

Exceptions in **Java**

```
void foo(String s) throws IOException {
        …
}
```

```
FileInputStream fis = new FileInputStream("a");
try {
        ...
        foo(s);
        ...
} catch (IOException e) {
        ...
        throw e;
} finally {
        fis.close();
}
```

Exception hierarchy



Cascade exceptions

```
public AST parseFile(File f) throws ParseException {
        AST ast = new AST();
        try {
                …
        } catch (IOException e) {
                ParseException pe = new ParseException();
                pe.initCause(e);
                throw pe;
        }
        …
        return ast;
}
```

```
try { ast = parseFile(f); }
catch (ParseException pe) { pe.printStackTrace(); }
```

Cascade exceptions 3

```
ParseException
        at Test2.parseFile(Test2.java:19)
        ...

Caused by: IOException
        at Test2.nextString(Test2.java:12)
        ...
        at Test2.parseFile(Test2.java:17)
```

- in **`Throwable`** *cause* can be set since Java 1.4
- shown in the *stack-trace*

Coding and style

Identifier style

- Variables, attributes and methods
  - `camelCase`, initial lower case
    - `getSecondBiggestNumber()`
    - `int importantVariable;`
- Class names
  - `CamelCase`, initial upper case
    - `StringBuffer`
- Package names
  - lower case
    - `java.util`

Parenthesis style

- Parenthesis
  - opening at end of line

```
while (true) {
```

  - continuation after closing

```
if (a<b) {
  ...
} else {
  ...
}
```