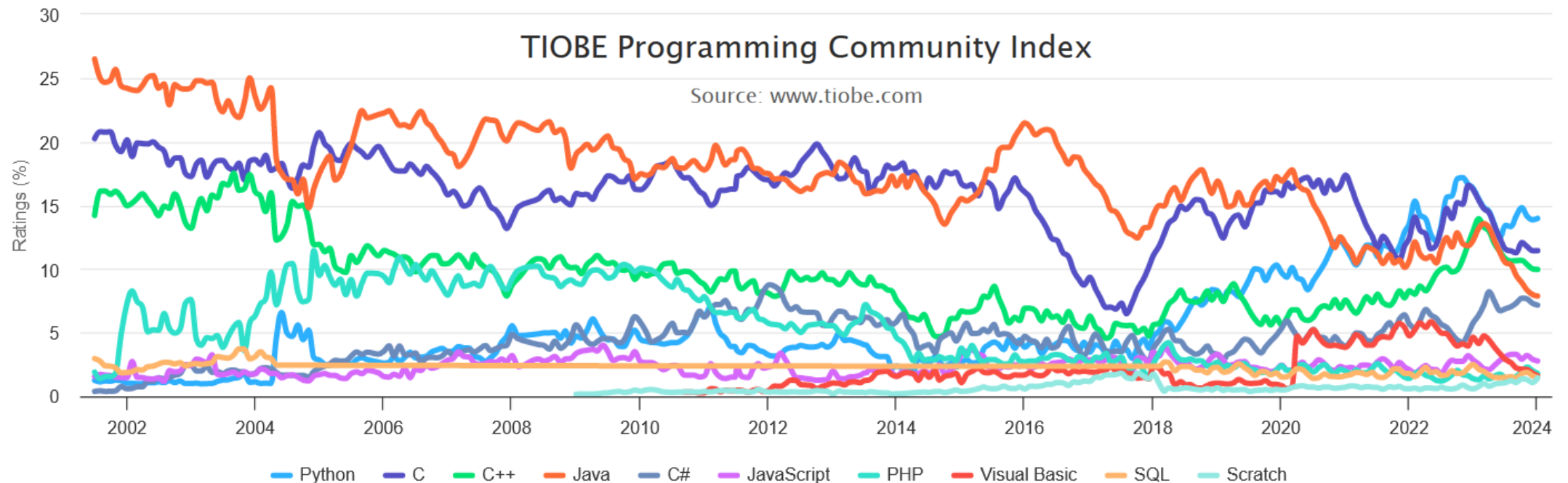


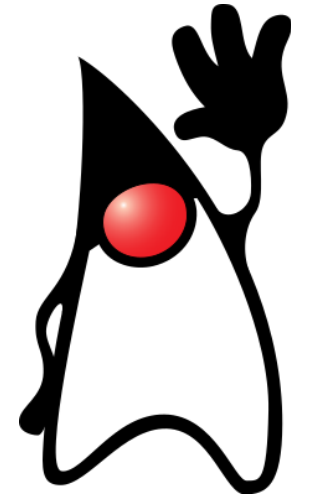
Java vs Python

Why should you know something about Java?

- Java is an example of a statically typed object oriented language (like C and C++) opposed to Python's being dynamically typed
- One of the most widespread used programming languages
- Used in other courses at the Department of Computer Science



Java history



- Java 1.0 released 1995 by Sun Microsystems (acquired by Oracle 2010)
- "Write Once, Run Anywhere"
- 1999 improved performance by the **Java HotSpot Performance Engine**
- Current version Java 22 (released March 2024)
- Java compiler generates **Java bytecode** that is executed on a **Java virtual machine (JVM)** that contains a garbage collector

PyPy is adopting the same ideas to Python (Just-in-Time compilation)

Installing Java

- To compile Java programs into **bytecode** you need a **compiler**, e.g. from Java SE Development Kit (**JDK**):

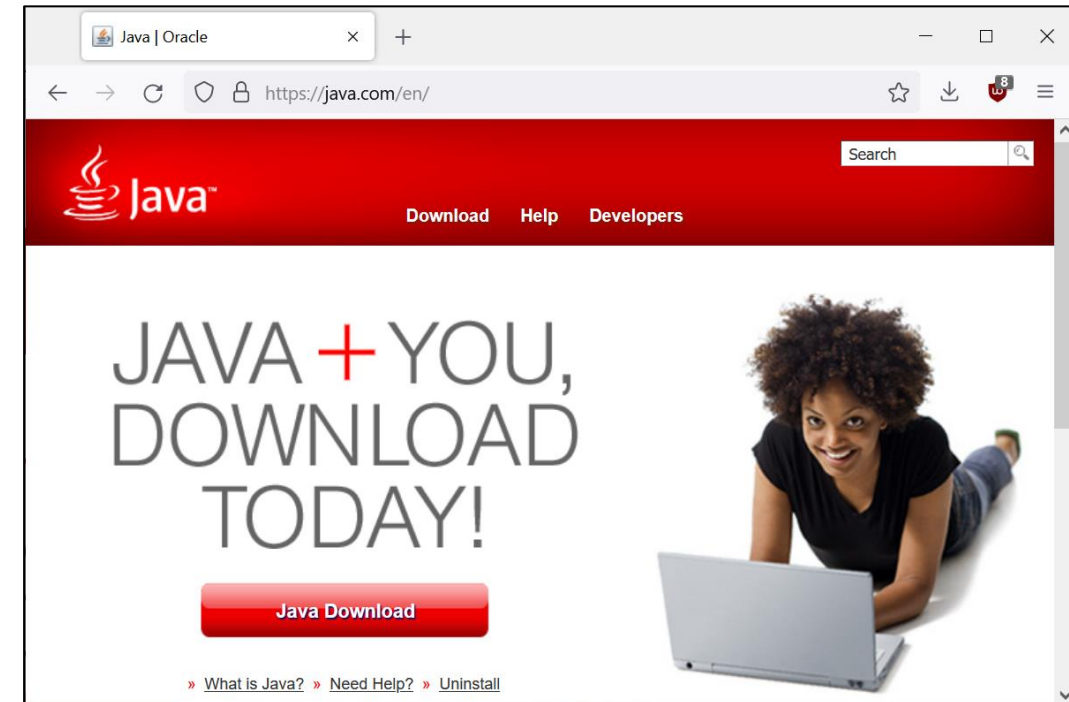
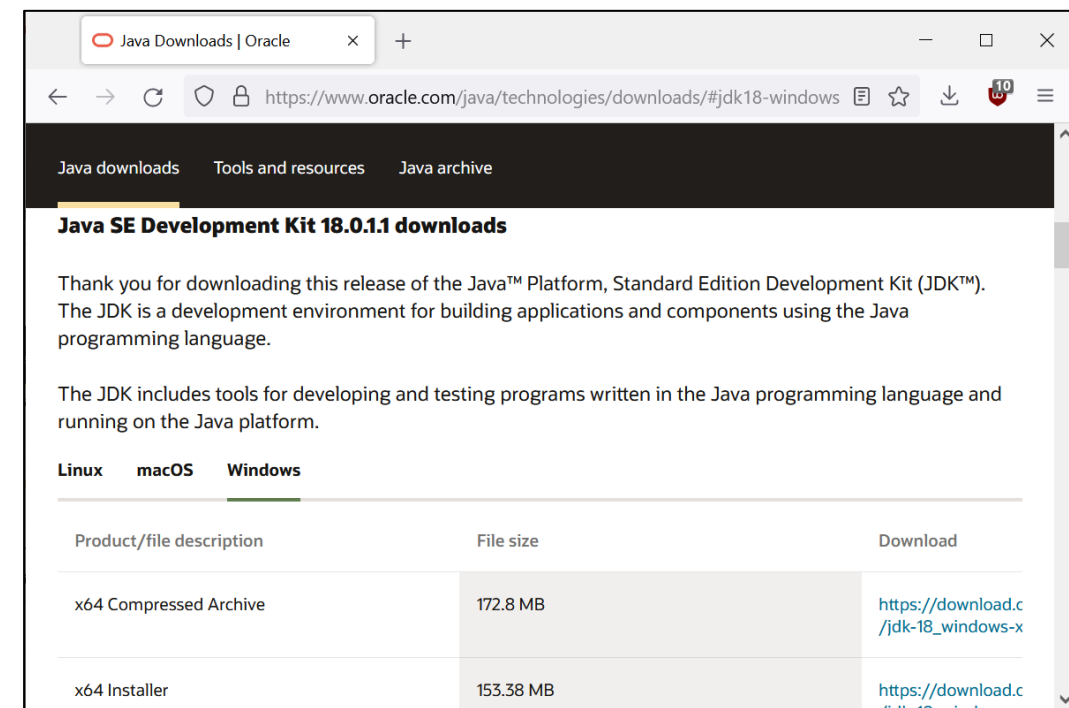
www.oracle.com/java/technologies/downloads/

(you might need to add the JDK directory to your PATH, e.g. C:\Program Files\Java\jdk-18.0.1.1\bin)

- To only run compiled Java programs:

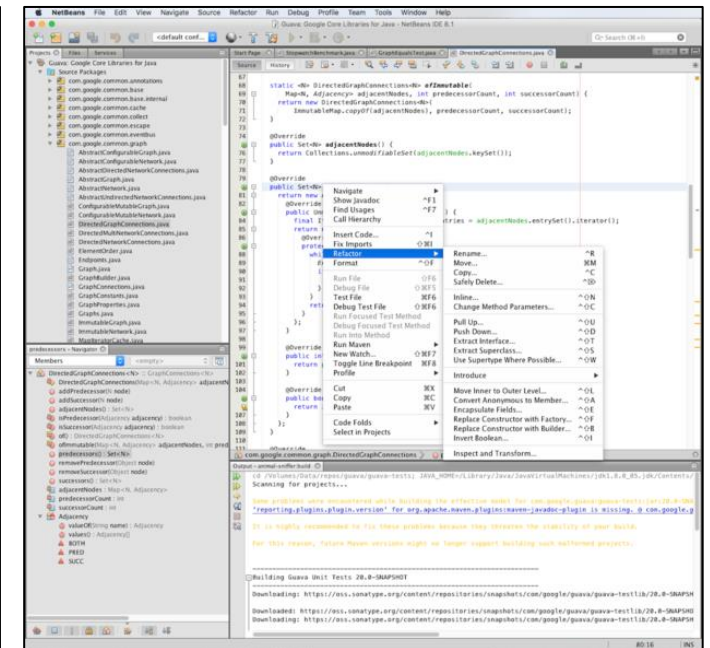
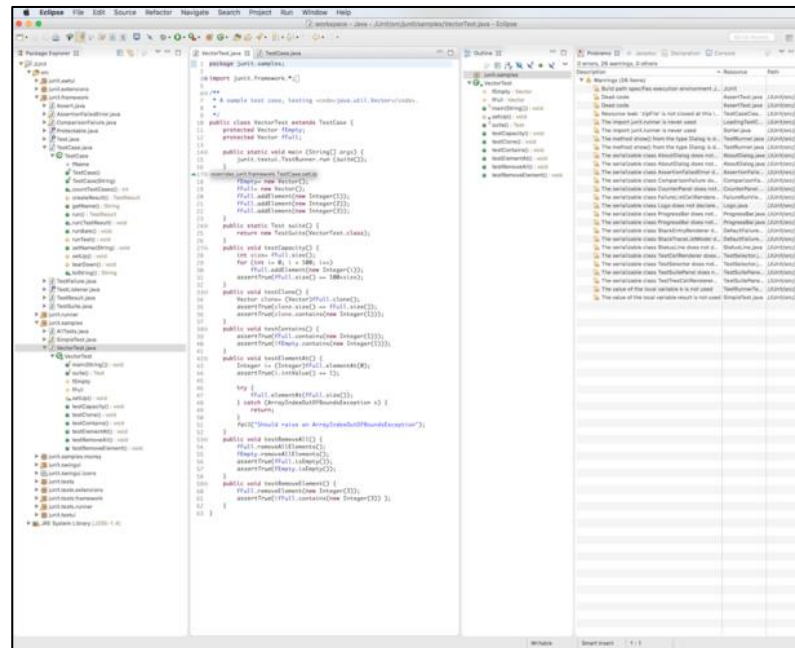
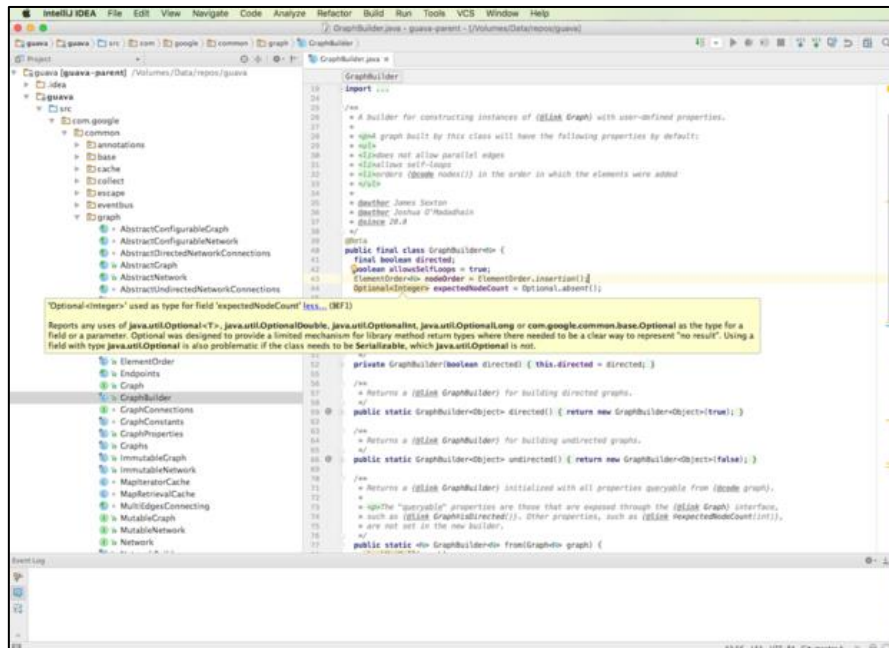
java.com/download

(If you use JDK, you should not download this)

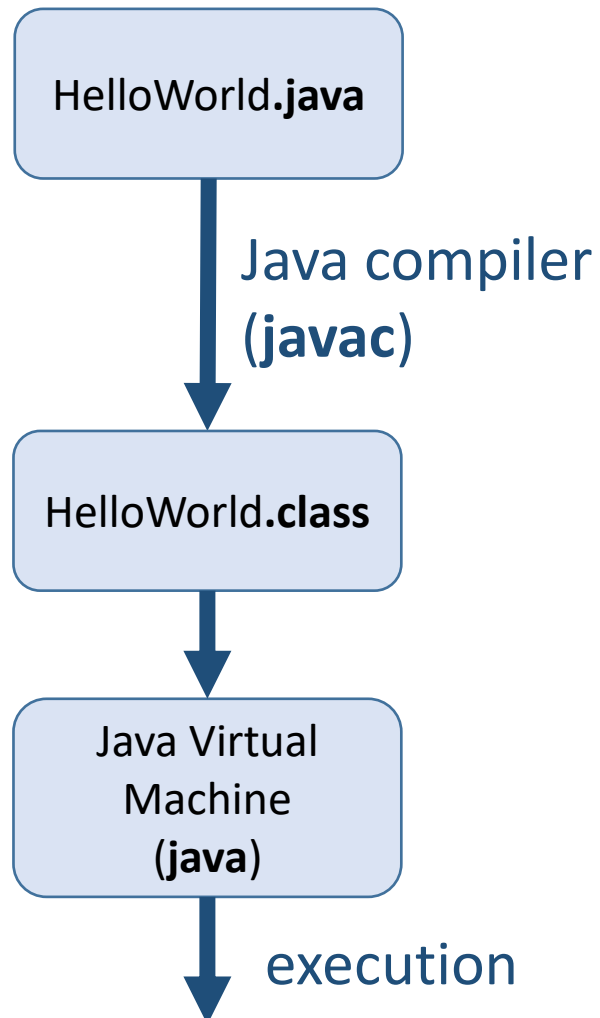


Java IDE

- Many available, some popular: Visual Studio Code, IntelliJ IDEA, Eclipse, and NetBeans
- An IDE for beginners: BlueJ



Compiling and running a Java program



HelloWorld.java

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

Command Prompt

C:\Users\au121\Desktop>javac HelloWorld.java

C:\Users\au121\Desktop>dir HelloWorld*

Volume in drive C is OSDisk

Volume Serial Number is 3CDB-90D8

Directory of C:\Users\au121\Desktop

04-05-2020	17:40	426	HelloWorld.class
09-05-2019	21:18	132	HelloWorld.java

2 File(s) 558 bytes

0 Dir(s) 372.191.944.704 bytes free

C:\Users\au121\Desktop>java HelloWorld

Hello World!

C:\Users\au121\Desktop>

Java : `main`

- `name.java` must be equal to the public class `name`
- A class can only be executed using `java name` (without `.class`) if the class has a class method `main` with signature

```
public static void main(String[] args)
```
- (`main` is inherited from C and C++ sharing a lot of syntax with Java)
- Java convention is that class names should use CamelCase

`PrintArguments.java`

```
public class PrintArguments {  
    public static void main( String[] args ) {  
        for (int i=0; i<args.length; i++)  
            System.out.println( args[i] );  
    }  
}
```

`shell`

```
> java PrintArguments x y z  
| x  
| y  
| z
```


a static method in
a class is a class method
(exists without creating objects)

method name

type of return value
(void = no return value)

class name containing main
must have same name as file

type of argument,
array of String values

name of argument

For-loop equivalent to

```
int i=0
while (i<args.length) {
    code
    i++;
}
```

i += 1

java arrays are indexed from
0 to args.length - 1
and the length of an array
object is fixed once created

the print statement is found
in the System class

declare new int variable
locally inside for-loop

the main method must
be public to be visible
outside class

there can be several classes
in a file – but only one class
should be public and have
same name as file

PrintArguments.java

```
public class PrintArguments {
    public static void main( String[] args ) {
        for (int i=0; i<args.length; i++)
            System.out.println( args[i] );
    }
}
```


Argument list also exists in Python...

```
PrintArguments.py
```

```
import sys  
print(sys.argv)
```

```
shell
```

```
> python PrintArguments.py a b 42  
| ['PrintArguments.py', 'a', 'b', '42']
```

Primitive.java

```
/**
 * A Java docstring to be processed using 'javadoc'
 */
// comment until end-of-line
public class Primitive {
    public static void main( String[] args ) {
        int x; // type of variable must be declared before used
        x = 1; // remember ';' after each statement
        int y=2; // indentation does not matter
        int a=3, b=4; // multiple declarations and initialization
        System.out.println(x + y + a + b);
        int[] v={1, 2, 42, 3}; // array of four int
        System.out.println(v[2]); // prints 42, arrays 0-indexed
        /* multi-line comment
           that continues until here */
        v = new int[3]; // new array of size three, containing zeros
        System.out.println(v[2]); // prints 0
        if (x == y) { // if-syntax '(' and ')' mandatory
            a = 1;
            b = 2;
        } else { // use '{' and '}' to create block of statements
            a = 4; b = 3; // two statements on one line
        }
    }
}
```

Why state types – Python works without...

- Just enforcing a different programming style (also C and C++)
- Helps users to avoid mixing up values of different types
- (Some) type errors can be caught at compile time
- More efficient code execution

type_error.py

```
x = 3
y = 'abc'
print('program running...')
print(x / y)
```

Python shell

```
| program running...
...
| ----> 4 print(x / y)
| TypeError: unsupported operand type(s) for
| /: 'int' and 'str'
```

TypeError.java

```
public class TypeError {
    public static void main( String[] args ) {
        int x = 3;
        String y = "abc";
        System.out.println(x / y);
    }
}
```

shell

```
> javac TypeError.java
| javac TypeError.java
| TypeError.java:5: error: bad operand types for
| binary operator '/'
|         System.out.println(x / y);
|                             ^
|
| first type:  int
| second type: String
| 1 error
```

Basic Java types

Type	Values
boolean	true or false
byte	8 bit integer
char	character (16-bit UTF)
short	16 bit integer
int	32 bit integer
long	64 bit integer
float	32 bit floating bout
double	64 bit floating point
class BigInteger	arbitrary precision integers
class String	strings

BigIntegerTest.java

```
import java.math.*; // import everything
// import java.math.BigInteger; // alternativ

public class BigIntegerTest {
    public static void main( String[] args ) {
        BigInteger x = new BigInteger("2");
        while (true) {
            // BigIntegers are immutable
            x = x.multiply(x);
            // java.math.BigInteger.toString()
            System.out.println(x);
        }
    }
}
```

shell

```
4
16
256
65536
4294967296
18446744073709551616
340282366920938463463374607431768211456
...
```

Java arrays

- The size of a builtin Java array can not be modified when first created. If you need a bigger array you have to instantiate a new array.
- Or better use a standard collection class like **ArrayList**
- **ArrayList** is a generic class (type of content is given by *<element type>*; generics available since Java 5, 2004)
- The for-each loop was introduced in Java 5

ConcatenateArrayLists.java

```
import java.util.*; // java.util contains ArrayList

public class ConcatenateArrayList {
    public static void main( String[] args ) {
        // ArrayList is a generic container
        ArrayList<String> a = new ArrayList<String>();
        ArrayList<String> b = new ArrayList<String>();
        ArrayList<String> c = new ArrayList<String>();

        a.add("A1"); // in Python .append
        a.add("A2");
        b.add("B1");

        c.addAll(a); // in Python .extend
        c.addAll(b);

        for (String e : c) { // foreach over iterator
            System.out.println(e);
        }
    }
}
```

shell


```
| A1
| A2
| B1
```

Tired of writing all these types...

- In Java 7 (2011) the “diamond operator” `<>` was introduced for type inference for generic instance creation to reduce verbosity
- In Java 10 (2018) the `var` keyword was introduced to type infer variables

ArrayListTest.java

```
import java.util.*; // java.util contains ArrayList

public class ArrayListTest {
    public static void main( String[] args ) {
        // ArrayList is a generic container
        ArrayList<String> a = new ArrayList<String>(); // Full types
        List<String> b = new ArrayList<String>();      // ArrayList is subclass of class List
        ArrayList<String> c = new ArrayList<>();       // <> uses type inference
        List<String> d = new ArrayList<>();            // <> and ArrayList subclass of List
        var e = new ArrayList<String>();              // use var to infer type of variable
         var v = Math.floor(1.5); // not obvious what type v is (double)
    }
}
```

Function arguments

- Must declare the number of arguments and their types, and the return type
- The argument types are part of the *signature* of the function
- Several functions can have the same name, but different type signatures
- Python keyword arguments, * and ** do not exist in Java 😞

Functions.java

```
public class Functions {  
    private static int f(int x) {  
        return x * x;  
    }  
    private static int f(int x, int y) {  
        return x * y;  
    }  
    private static String f(String a, String b) {  
        return a + b; // string concatenation  
    }  
  
    public static void main( String[] args ) {  
        System.out.println(f(7));  
        System.out.println(f(3, 4));  
        System.out.println(f("abc", "def"));  
    }  
}
```

shell

```
| 49  
| 12  
| abcdef
```

functions.py

```
def f(x, y=None):  
    if y == None:  
        y = x  
    if type(x) is int:  
        return x * y  
    else:  
        return x + y  
print(f(7), f(3, 4), f('abc', 'def'))
```


Class

- Constructor = method with name equal to class name (no return type)
- `this` = refers to current object (Python “self”)
- Use `private` / `public` on attributes / methods to give access outside class
- Use `new` *name*(arguments) to create new objects
- There can be multiple constructors, but with distinct type signatures

AClass.java

```
class Rectangle {
    private int width, height; // declare attributes
    // constructor, class name, no return type
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public Rectangle(int side) {
        width = side; // same as this.width = side
        height = side;
    }

    public int area() {
        return width * height;
    }
}

public class AClass {
    public static void main( String[] args ) {
        Rectangle r = new Rectangle(6, 7);
        System.out.println(r.area());
    }
}
```

shell

Inheritance

- Java supports single inheritance using `extends`
- Attributes and methods that should be accessible in a subclass must be declared `protected` (or `public`)
- Constructors are not inherited but can be called using `super`

Inheritance.java

```
class BasicRectangle {
    // protected allows subclass to access attributes
    protected int width, height;
    public BasicRectangle(int width, int height) {
        this.width = width; this.height = height;
    }
}

class Rectangle extends BasicRectangle {
    public Rectangle(int width, int height) {
        // call constructor of super class
        super(width, height);
    }
    public int area() {
        return width * height;
    }
}

public class Inheritance {
    public static void main( String[] args ) {
        Rectangle r = new Rectangle(6, 7);
        System.out.println(r.area());
    }
}
```

shell

Generic class

- Class that is parameterized by one or more types (comma separated)
- Primitive types cannot be type parameters
- Instead use *wrappers*, like `Integer` for `int`

GenericPair.java

```
class Pair<element> {  
    private element x, y;  
    public Pair(element x, element y) {  
        this.x = x; this.y = y;  
    }  
    element first() { return x; }  
    element second() { return y; }  
}  
  
public class GenericPair {  
    public static void main( String[] args ) {  
        var p = new Pair<Integer>(6, 7);  
        System.out.println(p.first() * p.second());  
    }  
}
```

shell

Interface

- Java does not support multiple inheritance like Python
- But a class can implement an arbitrary number of **interfaces**
- An interface specifies a set of attributes and methods a class must have
- The type of a variable can be an interface, and the variable can hold any object where the class is stated to **implement** the interface

RectangleInterface.java

```
interface Shape {
    public int area(); // method declaration
}

class Rectangle implements Shape {
    private int width, height;

    // constructor, class name, no return type
    public Rectangle(int width, int height) {
        this.width = width; this.height = height;
    }

    public int area() {
        return width * height;
    }
}

public class RectangleInterface {
    public static void main( String[] args ) {
        Shape r = new Rectangle(6, 7);
        System.out.println(r.area());
    }
}
```

Abstract classes

- **Abstract class** = class that cannot be instantiated, labeled `abstract`
- **Abstract method** = method declared without definition, labeled `abstract`, must be in abstract class
- An abstract class can be **extended** to a non-abstract class by providing the missing method definitions

AbstractRectangle.java

```
abstract class Shape {
    abstract public int circumference();
    abstract public int area();
    public double fatness() {
        // convert int from area() to double before /
        return (double)area() / circumference();
    }
}

class Rectangle extends Shape {
    private int width, height;
    // constructor, class name, no return type
    public Rectangle(int width, int height) {
        this.width = width; this.height = height;
    }
    public int area() {
        return width * height;
    }
    public int circumference() {
        return 2 * (width + height);
    }
}

public class AbstractRectangle {
    public static void main( String[] args ) {
        Shape r = new Rectangle(6, 7);
        System.out.println(r.fatness());
    }
}
```

Default methods in interfaces

- Before Java 8 all methods in an interface were abstract (no definition)
- Since Java 8 interfaces can have **default** methods with definition
- The distinction between abstract classes and interfaces gets blurred
 - a class can only extend one abstract class
 - a class can implement more interfaces⇒ multiple “inheritance” is possible in Java

DefaultInterface.java

```
interface Shape {
    public int circumference();
    public int area();
    default public double fatness() {
        // convert int from area() to double before /
        return (double)area() / circumference();
    }
}

class Rectangle implements Shape {
    private int width, height;
    // constructor, class name, no return type
    public Rectangle(int width, int height) {
        this.width = width; this.height = height;
    }
    public int area() {
        return width * height;
    }
    public int circumference() {
        return 2 * (width + height);
    }
}

public class DefaultRectangle {
    public static void main( String[] args ) {
        Shape r = new Rectangle(6, 7);
        System.out.println(r.fatness());
    }
}
```

Multiple Inheritance

- Class C implements both interfaces A and B
- Inherits default methods sayA and sayB
- Cannot inherit sayHi, since in both A and B. Must be overridden in C
- Can use @override to enforce compiler to check if method exists in super class
- new A() {} creates an instance of an *anonymous class* (extending or implementing A)

MultipleInheritance.java

```
interface A {
    default public void sayA() { System.out.println("say A"); }
    default public void sayHi() { System.out.println("A say's Hi"); }
}

interface B {
    default public void sayB() { System.out.println("say B"); }
    default public void sayHi() { System.out.println("B say's Hi"); }
}

class C implements A, B {
    @Override // (optional) requests compiler to
               // check if sayHi exists in supertype
    public void sayHi() {
        System.out.println("C say's Hi");
        (new A(){}).sayHi(); // instantiate an anonymous class
    }

    public void test() {
        sayA();
        sayB();
        sayHi();
    }
}

public class MultipleInheritance {
    public static void main( String[] args ) {
        new C().test();
    }
}
```

Shell output

```
say A
say B
C say's Hi
A say's Hi
```


Lambda expression

- Lambda expressions are possible since Java 8
- Syntax : *argument -> expression*

LambdaPrinting.java

```
import java.util.*; // ArrayList
public class LambdaPrinting {
    public static void main(String[] args) {
        var elements = new ArrayList<Integer>();
        for (int i = 1; i <= 3; i++)
            elements.add(i);
        elements.forEach(e -> System.out.println(e));
    }
}
```

LambdaPrinting.java

1
2
3



Welcome to Java for Python Programmers

Contents:

- 1. Java for Python Programmers
 - 1.1. Preface
 - 1.2. Introduction
 - 1.3. Why Learn another programming Language?
 - 1.3.1. Why Learn Java? Why not C or C++?
 - 1.4. Lets look at a Java Program
 - 1.5. Java Data Types
 - 1.5.1. Numeric
 - 1.5.1.1. Import
 - 1.5.1.2. Declaring Variables
 - 1.5.1.3. Input / Output / Scanner
 - 1.5.2. String
 - 1.5.3. List
 - 1.5.4. Arrays

