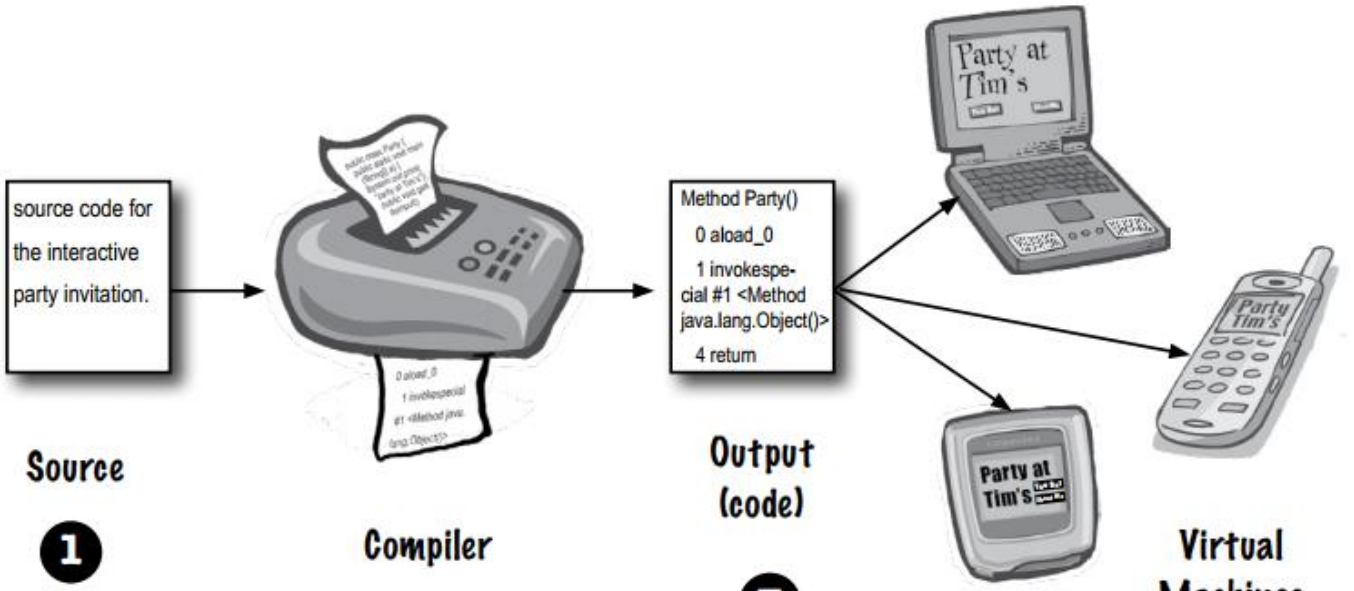
Chapter 1

Dive in a quick dip

The way Java works



1

Create a source document. Use an established protocol (in this case, the Java language).

2

Run your document through a source code compiler. The compiler checks for errors and won't let you compile until it's satisfied that everything will run correctly.

3

The compiler creates a new document, coded into Java ***bytecode***. Any device capable of running Java will be able to interpret/translate this file into something it can run. The compiled bytecode is platform-independent.

4

Your friends don't have a physical Java Machine, but they all have a ***virtual*** Java machine (implemented in software) running inside their electronic gadgets. The virtual machine reads and *runs* the bytecode.

What you'll do in Java

- Type a source code file, compile it using the javac compiler, then run the compiled bytecode on a Java Virtual Machine (JVM).

```
import java.awt.*;
import java.awt.event.*;
class Party {
  public void buildInvite() {
    Frame f = new Frame();
    Label l = new Label("Party at Tim's");
    Button b = new Button("You bet");
    Button c = new Button("Shoot me");
    Panel p = new Panel();
    p.add(l);
  } // more code here...
}
```

Source

1

Type your source code.

Save as: ***Party.java***

```
File Edit Window Help Plead
%javac Party.java
```

Compiler

2

Compile the ***Party.java*** file by running `javac` (the compiler application). If you don't have errors, you'll get a second document named ***Party.class***

The compiler-generated Party.class file is made up of *bytecodes*.

```
Method Party()
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 return
Method void buildInvite()
  0 new #2 <Class java.awt.Frame>
  3 dup
  4 invokespecial #3 <Method java.awt.Frame()>
```

Output (code)

3

Compiled code: ***Party.class***

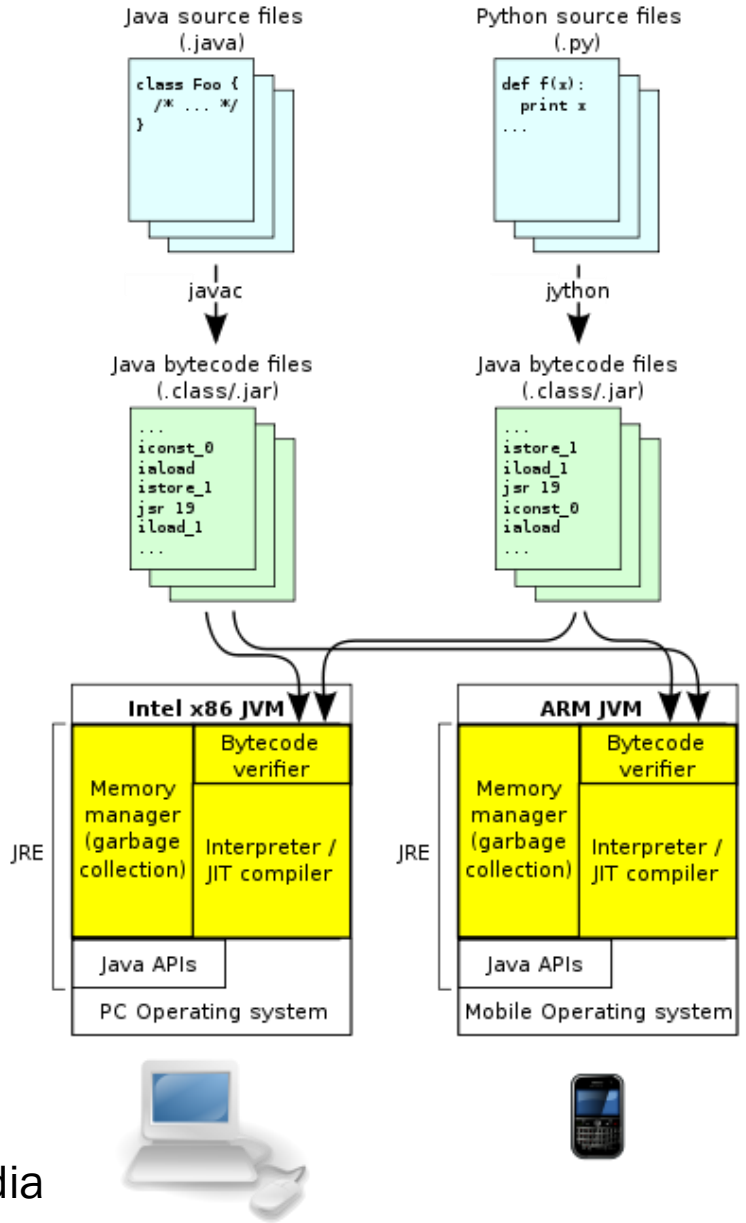
```
File Edit Window Help Swear
%java Party
```

Virtual Machines

4

Run the program by starting the Java Virtual Machine (JVM) with the ***Party.class*** file. The JVM translates the *bytecode* into something the underlying platform understands, and runs your program.

- A **Java virtual machine** (**JVM**) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode.

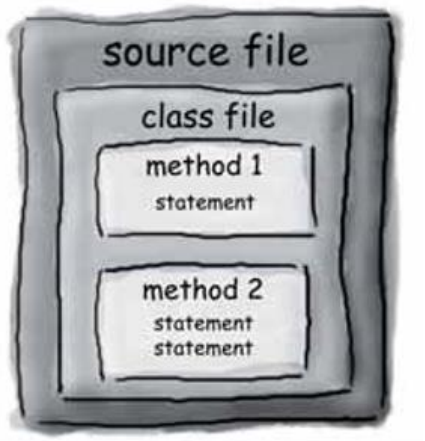


Image/article source: Wikipedia

Quiz:
What is the difference between the **java compiler** and **JVM**?

- Some important Java versions
- 2014 – Java 8 (LTS). LTS stands for long term support
- 2018 – Java 11 (LTS)
- 2019 – Java 13
- 2021 – Java 17 (LTS)
- 2022 – Java 18 (to be released by March 2022)
- Java SE support roadmap
  - https://www.oracle.com/java/technologies/java-se-support-roadmap.html

- Code structure in Java
- Put a **class** in a source file
- A source file holds one class definition
- Put **methods** in a class
- A class has one or more methods
- Put **statements** in a method

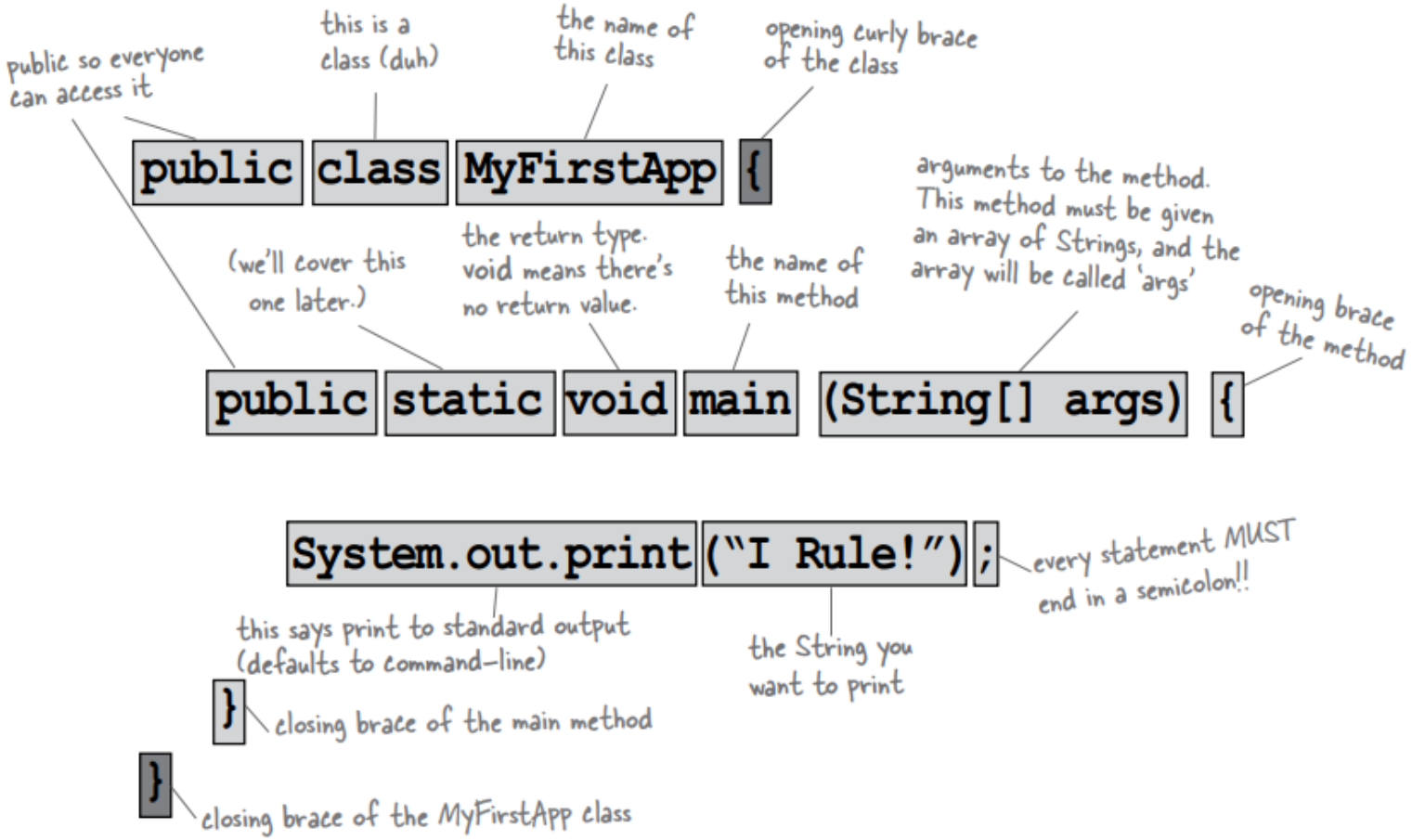



```
public class Dog {

}
```
class

```
public class Dog {
   void bark() {

   }
}
```
method

```
public class Dog {
   void bark() {
     statement1;
     statement2;
   }
}
```
statements

- Your program may look like below:



- When the JVM starts running, it looks for the class you give it at the command-line. Then it starts looking for a specially-written method called “main”

```
public static void main (String[] args) {
   // your code goes here
}
```

- Every Java application must have at least one class and at least one main method (just one main per *application*)
- In java, everything goes in class. Your type your source file (with *.java* extension), then compile it into a new class file (with a *.class* extension)
- Running your program means to tell JVM to “**load a class then start executing main() method**”

- The logical operators in Java is almost the same as C.
- Example: You wrote a program and expect the output of a program is listed below

```
3 bottles of beer on the wall, 3 bottles of beer.
Take one down and pass it around, 2 bottles of beer on the wall.

2 bottles of beer on the wall, 2 bottles of beer.
Take one down and pass it around, 1 bottle of beer on the wall.

1 bottle of beer on the wall, 1 bottle of beer.
Take one down and pass it around, no more bottles of beer on the wall.

No more bottles of beer on the wall, no more bottles of beer.
```

```
public class BeerSong {
    public static void main(String[] args) {
        int beerNum = 3;
        String word = "bottles";
        while (beerNum > 0)
        {
            if (beerNum == 1)
            {
                word = "bottle";
            }
            System.out.println(beerNum + " " + word + " of beer on the wall");
            System.out.println(beerNum + " " + word + " of beer");
            System.out.println("Take one down and pass it around,");
            beerNum = beerNum - 1;
            if (beerNum > 0)
            {
                System.out.println(beerNum +  " " + word + " of beer on the wall");
            }
            else
            {
                System.out.println("No more bottles of beer on the wall");
            }
        }
    }
}
```

There's still one little flaw in this code. Spot it and try to fix it!

Can they compile?

A

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      if ( x > 3) {
        System.out.println("big x");
      }
    }
  }
}
```

B

```
public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
}
```

C

```
class Exercise1b {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
      }
    }
}
```

- **Java programming style guide (conventions)**
- Make sure that all lines can be read without the need for horizontal scrolling
  - A maximum of 79 characters per line is recommended.
- Indentation - all indenting is done with **spaces**, not **tabs**. All indents are **four spaces**.
  - If your tabbing is set to 4 and you share a file with someone that has tabbing set to 8(or 2), everything explodes
- Putting an opening curly brace "{" at the end of a line
- A keyword followed by a parenthesis should be separated by a space

```
void foo() {
    …
}
```

- All if, while and for statements must use braces even if they control just one statement
- ***Consistency is easier to read***. Plus, less editing is involved if lines of code are added or removed.

```
if (a < b){
    c++;
}
```

- The keywords if, while, for, switch, and catch must be followed by a space.

```
if(handsome)    // BAD!
if (handsome)   // YES!
```

- All method names should be immediately followed by a left parenthesis (no blank space between them)

```
void foo ()   // BAD!
```

- The if part should be smaller than the else part.
- Below is a **bad** code:

```
if (condition) {
    //30 lines code.
}
else {
    //one-line statement.
}
```

- Commas and semicolons are always followed by whitespace.

```
for (int i = 0;i < 10;i++)    // BAD!
for (int i = 0; i < 10; i++)    // YES!

myFunction(a,b);  // BAD!
myFunction(a, b);  //YES!
```

- All array dereferences should be immediately followed by a left square bracket

```
a [5];  // BAD!
```

- Binary operators should have a space on either side.

```
a=b+c  // BAD!
a = b + c // YES!
```

- Unary operators should be immediately preceded or followed by their operand.

```
count ++;  // BAD!
count++;   // YES!
```

- All **class** and **interface** identifiers, use ***upper*** camel case

```
class MyClass
{
    ….
}
```

- For **variables**, **methods** and **parameters**, use ***lower*** camel case

```
int numOfStudents
```

- Hungarian notation (or say, System Hungarian) violates OO abstraction

```
String sFirstName;   // BAD!
int iNumber;         // BAD!
double dbPi;         // BAD!
```

- vUsing adjHungarian nnotation vmakes nreading ncode adjdifficult.

- Test code is permitted to use underscores in identifiers for methods and fields
  - Code to test a method `double foo(int x)` may use variables such as
    `foo_x`
    `foo_return`

- Never use **do..while** – it's confusing. (When encountering a loop, the first thing the programmer wants to know is ***what terminates the loop***)
- Don't use **return** in the middle of a method (except its inside a conditional branch)
- Use a separate line for an increment or decrement

```
foo(x++);   // BAD!

foo(x);
x++;            // YES!
```

- Declare **local variables** as close as possible to where they are used

- Documentation should be written ***during*** the coding/debugging process
- Get in the habit of documenting a variable when you declare it and specifying a method ***before*** you write its body.
- If you decide to change what a method is supposed to do, change its specification first!
- You document a program well so that others have an easier time understanding it
- A professional programmer is well regarded if their code is **easy to maintain** by others.

- Omit needless words. Use the active voice.
- Remember, you're not writing an article. You're trying to make others understand your code ASAP.

```
/**
 This function searches value y in array x and
 returns true if array x contains y.
 */
boolean isIn(int y, int[] x)
```

```
/**
* Check if y in x
* @param y   value you want to search
* @param x   target array
* @return  true if y in x
*/
```

- There are 3 kinds of comments in Java
- // : one-line comments
- /* … */ : multiple lines comments
- /** … */ : Javadoc comments.
- The Javadoc comments are placed right above a method.
- When writing a Java program, if you hover the mouse over a call of this method, a pop-up window opens with that Javadoc in it.
  - Many popular IDEs such as Eclipse support this feature.

- **Don't over-comment!**

```
i= i+1; // add one to i
```

- Self-Documenting Code
  - The naming of variables should close to what they represents conceptually

```
boolean isRunning;
int numOfStudents;
String dogName;
```

- (May not be necessary) You can write some specifications in Javadoc

```
/** note: width in 0 to 1920, height in 0 to 1080*/
int area(int width, int height)
```