Java

about the language

Object-Oriented

Object-oriented programming (OOP) concerns itself with a particular type of data structure generally referred to as an ***object***.

Definition For our purposes, an ***object*** may be regarded as: (1) a set of variables which determine the object's state and (2) a set of methods which define the object's behavior.

The conglomeration of variables and methods form the blueprint of an object, so to speak. This blueprint is the *definition of a* ***class***. We say that an object belongs to – or, is a member of – a class if it is defined by the blueprint for that class. Furthermore, an object's ***type*** is given by the name of the class to which it belongs.

```
class Carpet {
    boolean clean, magic;
    int altitude = 0;

    void clean() {
     clean = true;
    }

    void fly() {
     if(magic)
     altitude += 10;
    }

    void land() {
     if(magic)
     altitude = 0;
    }
}
```

```
// an object of type: Carpet
Carpet aladdinsCarpet = new Carpet();

// changing state directly
aladdinsCarpet.magic = true;

// changing state indirectly
aladdinsCarpet.clean();
aladdinsCarpet.fly();
```

From a single class, we can create many objects (each considered to be of the same type). Each object of a particular type has its own state, while all objects of the same type have the same behavior. In other words: objects of the same type *share behavior* and *differ in state*.

Java is considered to be a *highly* object-oriented language. This is because all program activity occurs *within a class*. That is to say: all – well, almost all – source code is written within a class definition.

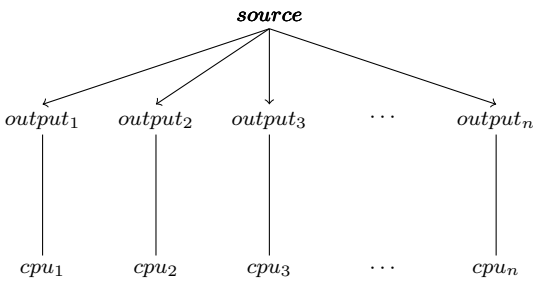
Object-oriented programming generally adopts a particular philosophy with regard to language design. Some of the defining principles of object-oriented programming are:

- ***Encapsulation*** binds code and data together and allows the programmer to regulate access to information.
- ***Polymorphism*** allows the programmer to define a general method that is implemented by various classes. Each class that implements that method then has the freedom to defines its behavior as needed.
- ***Inheritance*** is the ability for one class (the child) to inherit the variables and methods of another class (the parent). This allows for a hierarchical classification of types.

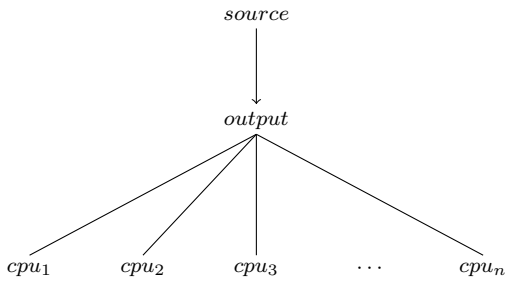
Portable

Java's portability ensures that Java code does not have to be recompiled for different computer architectures or operating systems. This is acheived through an intermediary phase between compilation and execution. Java source code is first compiled into Java ***bytecode***. The bytecode is then interpreted by the *Java Virtual Machine* (JVM). Implementations of the JVM differ according to architecture and operating system, but *all implementations of the JVM can run any bytecode*. This means that a Java program has to be compiled only once in order to run on any machine that has the JVM installed.

Traditional Compilation



Java Compilation



Aside from the obvious gains portability offers to programmers, it also promotes web development. This is because software which resides on a server may, for instance, exploit the client's processing power to execute code without concern for differing architectures or operating systems – i.e. the code doesn't have to be recompiled for each client machine. This idea finds its initial form in the ***Java applet*** – an application that runs within the context of a Java compatible web browser and is automatically downloaded and executed. Of course, allowing unknown code to execute on one's machine is a serious security concern which the designers of Java had to resolve.

Secure

Java's security is inherently tied to its portability. Since all Java programs are executed within the context of the JVM, constraints can be placed on code during runtime.

The JVM has a bytecode verifier which ensures that operations which are potentially unsafe cannot be performed. For instance, the JVM ensures that code only branches to locations within the same method. It also enforces strict bounds checking, data initialization, and type safety. Furthermore, the Java language does not provide the programmer with a means to perform pointer arithmetic as well as manual memory allocation both of which are notorious for creating security vulnerabilities.

Dynamic

Java is a *highly* dynamic language with regards to its memory management. Memory for primitive types is allocated statically, while all other memory is allocated dynamically with a call to the `new` operator. This includes the allocation of arrays of primitive types, as arrays are implemented as objects in Java. However, Java implements a *garbage collector* which takes care of memory deallocation and compaction, making memory management very simple for the user.