Java

Java

- Java is a successor to a number of languages, including Lisp, Simula67, CLU, and SmallTalk.
- Java is superficially similar to C and C++ because its syntax is borrowed from them. However, at a deeper level it is very different from these languages.
- Java is an object-oriented language.
  - Most of the data manipulated by programs are contained in objects
  - Objects contain both state and operations
    - State is called fields, variables
    - Operations are called methods, functions

What is a Variable?

What is a Variable?

- Value Model for Variables
  - a ← b;
    - a is an l-value (location)
    - b is an r-value
      - Data that is stored somewhere
  - Variables are “boxes” that hold a value
    - A named container for a value

b ← 2;
c ← b; // copy data stored at location b to location c
a ← b + c; // combine data at b and c and copy to a

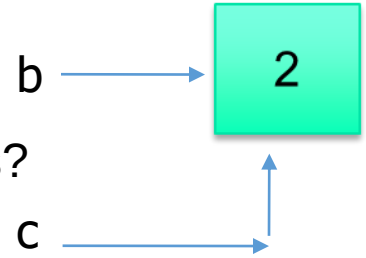
What happens if we now include b ← 3; and ask for the value at c?

Reference Model for Variables

- A variable is a reference to an object that has a value
  - Every variable is an l-value
    - A location value
  - Dereference to get r-value

b ← 2; // copy 2 to the location that b references
c ← b; // copy reference not value
a ← b + c; // what does this do?

What happens if we now execute
b ← 3 and ask for the value that c references?



Models for Variables

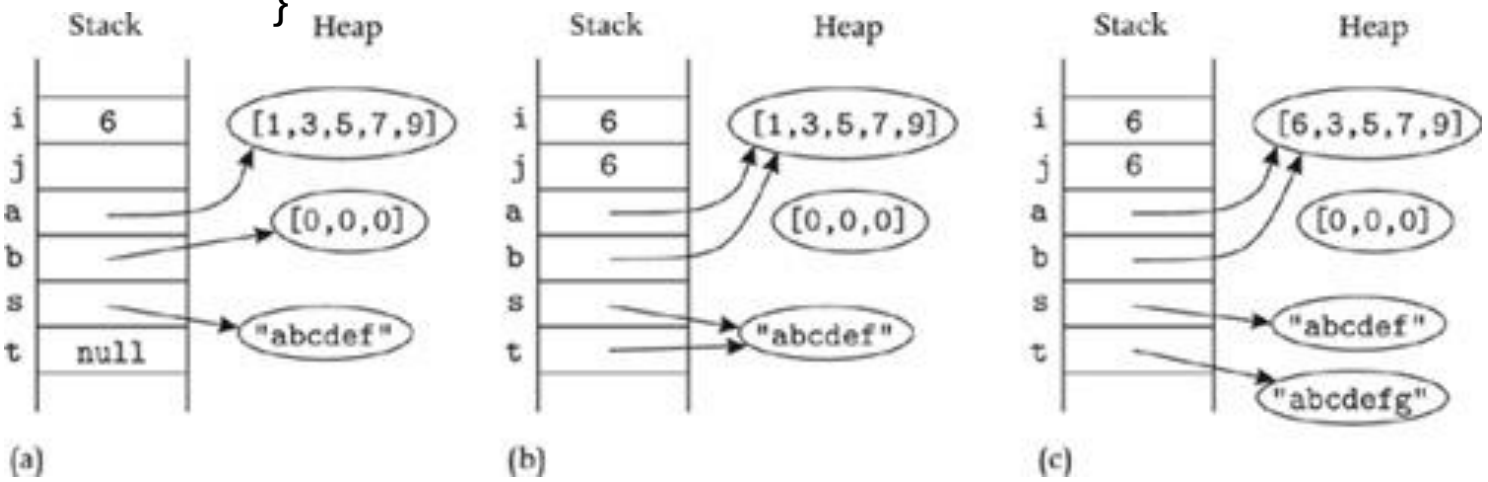
- C++ uses the value model
  - It does have *references*, but they are different from Java references
  - C++ references are aliases
  - C++ uses pointers
- Java and Python use a *mixed* model
  - Primitive types use the value model
    - byte, short, int, long, float, double, char, boolean
  - Class types use reference model
    - Objects created with new

```
int x = 7;   // value model
char c = 'A';

String s = new String("cat");  // reference model, s contains a reference
                               // to an object containing "cat"
```

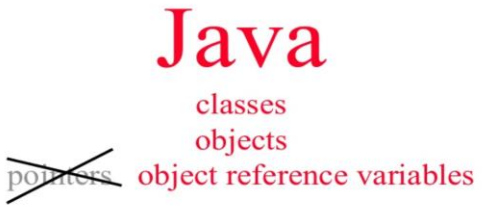
Java Uses a Mixed Models for Variables

- Local variables, such as those declared within methods, reside on the runtime stack
  - space is allocated for them when the method is called and deallocated when the method returns
- Variables of primitive types are r-values
  - Value model
  - Methods that return primitives return the value of the primitive
    - Return a copy of the value of the variable
- Variables of all other types, including strings and arrays, contain references to objects that reside on the heap
  - Reference model
  - Created with new
  - For local reference variables, a *reference* is stored on stack, object data is stored in heap

```
void some_method() {
  int i = 6;
  int j; // uninitialized
  int [ ] a = {1,3,5,7,9}; // creates a 5-element array
  int [ ] b = new int[3]; // auto initialized to 0
  String s = new String("abcdef"); // creates a new string
  String t = null;  // null is a reference

  j = i;  // copy a value
  b = a; // copy a reference; now nothing refers to [0,0,0]
  t = s;  // now s and t refer to the same object

  a[0] = 6;
  t = t +"g";  // make a new string. Why?
}
```

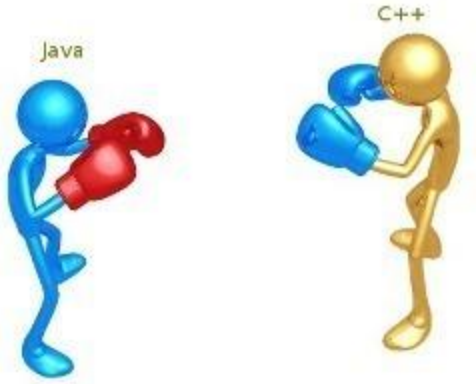


Differences between Java reference and C++ pointer variables



- There's no "reference arithmetic"
  - https://softwareengineering.stackexchange.com/questions/141834/how-is-a-java-reference-different-from-a-c-pointer
- References **might** be implemented by storing the address.
  - In Java there might be an additional level of indirection
- References are strongly typed
  - C/C++ pointers can point to any object by casting
- Java doesn't have explicit pointers; C/C++ does
  - Well, you can get addresses using the UnSafe class, but it's usually a bad idea

Java: Differences with C++



- Model for variables
  - Java uses the reference model for class types
    - No explicit pointers.
      - references instead
    - Explicitly create objects:
      - `Foo f = new Foo();`
    - Two types of equal:
      - `==` and `equals`.
      - Remember: when comparing string values, use `equals`!
- Types and type checking, type safety
- Interpretation vs. Compilation
- Interfaces, inheritance, etc. are handled differently

Java: Differences with C++

- Access control modifiers -- public, private, protected
  - Determine class member visibility
    - Java: has package visibility as default;
    - A package is a namespace
  - *protected* is slightly different than C++
- Java has built-in **garbage collection**
  - C++ does not
  - In C++, the programmer is responsible for memory management
    - The dreaded memory leak!
    - It's possible to have a memory leak in Java but the compiler warns you about it
  - In Java you don't need to explicitly free memory

Java Access Modifiers

	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
No modifier	Y	Y	N	N
private	Y	N	N	N

For example, a protected object is visible to the class containing it, the classes in the package, any subclasses of the class, but not to the outside world.

Private objects are only visible to the class containing the object.

Java: Differences with C++

- C++ allows multiple inheritance
  - The dreaded triangle inheritance dilemma
- Java allows only for single class inheritance
  - it does allow for multiple interface inheritance

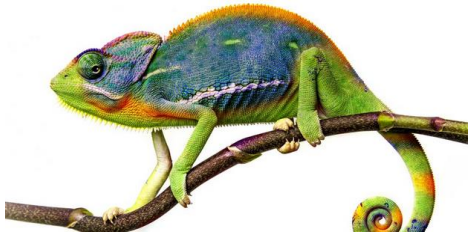
```
class B extends A implements J, K, L, .....

public interface ShapeInterface {
  public void draw();
  public double getArea();
}

public class Square implements ShapeInterface {
 ...
}
```

Java: Differences with C++

- Interface cannot contain any implementation, just method signatures
  - Contract between interface class and subclasses.
  - Can't be *instantiated*
- Abstract classes can contain implementations for some methods
  - Can't be directly instantiated but can be subclassed

```
public abstract class AbstractMapEntry<K,V> implements MapEntry<K,V> {
    public abstract K getKey();    // must be implemented by subclass
    public abstract V getValue();

    public boolean equals( Object o ) {
      if ( o == this )  // etc.
    }
    ...
}
```

Mutability



- All objects are either immutable (non-changeable) or mutable (changeable)
- Strings are immutable
  - t = t + “g”;
  - Creates a new string object
  - t now references the new object
- Arrays are mutable
  - a[i] = 27;
  - a’s ith item has changed
- If an object is shared (referenced) by two variables, it can be accessed through either of them.
  - If a mutable object is shared by two variables, changes made through one of the variables will be visible when the object is accessed through the other.

Aside: What happens under the hood with String concatenation?

```
String a = new String(“cat”);
String b = new String(“ and dog”);

String c = a + b;

// equivalent
String c = new StringBuilder(a).append(b).toString();
```

- StringBuilder creates a mutable sequence of characters from a, concatenates the string from b, and finally converts the whole thing to a String.

- Java overloads the “+” operator for the String class.
- Unlike C++, users can’t overload operators in Java.

Types and Type Checking, Type Safety

- What is the role of types?
  - Data abstraction
  - Safety!
- Types and type checking prevent the program from going wrong.
  - Disallows operations on objects that do not support those operations
    - `a+b` where `a` and `b` are `2DPoint`s is rejected by the type checker
    - `a.substring(0,10)` where `a` is an `Integer` is rejected too

Types and Type Checking, Type Safety

- Java is a *strongly typed* language
  - Java checks code to ensure each assignment and method call is type correct
  - Every variable declaration gives the variable type
  - The header of every method and constructor defines its type signature
    - the types of its arguments and results and also the types of any exceptions it throws
- Legal Java programs are guaranteed to be type safe
  - No type errors when the program runs
    - If you eliminate warnings
- Java provides automatic storage management for all objects.
- Java checks all array accesses to ensure they are within bounds

Type Safety

- Type safety: no operation is ever applied on object of the wrong type (i.e., object that does not support that operation)
- Java is type safe while C/C++ is type unsafe
  - In Java, the type system never allows operations on objects of the wrong type
    - no execution
  - In C++, the type system prevents most errors, but it is possible to write a program where operations on objects of the wrong type occur
- Java Goal: catch errors as early as possible!

Types and Type Checking

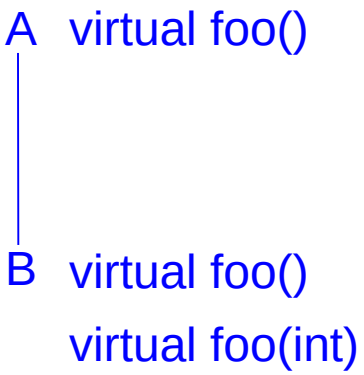
"Now! *That* should clear up a few things around here!"

- Java and C/C++ are statically typed
  - A statically typed language typically requires *type annotations*;
    - performs substantial amount of type checking before runtime
  - Expressions have static (*compile-time*) types
  - Objects have dynamic (*run-time*) types
- Alternative is dynamically typed
  - Perform substantial type checking during runtime
  - Python
    - Strongly dynamically typed

C++ can be Type Unsafe

Note: In Java, all methods are virtual; use **`final`** keyword to force non-virtual

```
// C++:
void* x = (void *) new A;
B* q = (B*) x;  //a safe downcast?
int case1 = q->foo(1);//what happens?
```



Throws ClassCastException here!
Runtime never reaches bad call.

```
// Java:
Object x = new A();
B q = (B) x;  //a safe downcast?
int case1 = q.foo(1); //what happens?
```

C++ is Type Unsafe

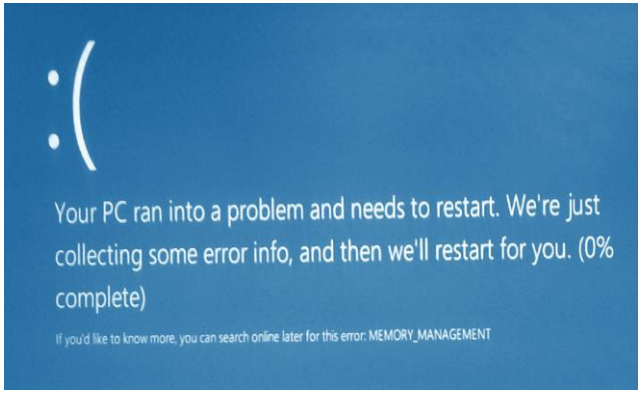
Java: **`B q; … q.foo(1);`**

- Java "honors" its promise that at runtime it will find a **`q.foo(1)`**
- if **`q`** is not null, **`q`** is a **`B`** (or subclass of **`B`**).
    - i.e. q has a foo(int)

C++: **`B* q; … q->foo(1);`**

- C++ does not "honor" its promise. At runtime, **`q`** can be a **`B`** or an **`A`** or a **`Duck`** or whatever

Memory Management



- Java manages heap memory
  - No need to free an object when it goes out of scope
  - Garbage collector frees unused objects
    - Objects that are referenced by strong reference variables are not freed
    - Objects that are referenced by weak reference variables may be freed
      - https://dzone.com/articles/java-memory-management
      - https://stackoverflow.com/questions/299659/whats-the-difference-between-softreference-and-weakreference-in-java
- C++ requires the programmer to manage heap memory
  - Can lead to memory leaks if programmer fails to free object when it goes out of scope
- Microsoft reports:
  - *The majority of vulnerabilities fixed and with a CVE (Common Vulnerability and Exposures) assigned are caused by developers inadvertently inserting memory corruption bugs into their C and C++ code.*
    - https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/

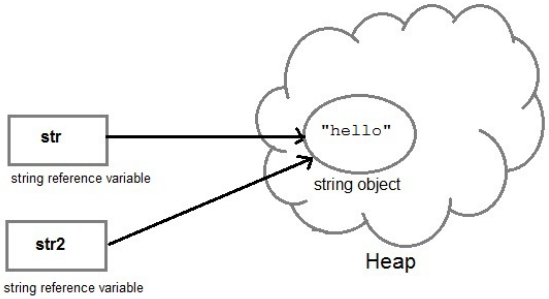
Type Safety

- Java ensures type safety with a combination of compile-time (static) and runtime (dynamic) type checking
  - Compiler rejects plenty of programs.
    - **`String s = 1`** is rejected,
    - **`String s = new Integer(1)`** is rejected
  - Some checks are left for runtime. Why?
    - **`B b = (B) x;`**
    - the Java runtime checks if **`x`** refers to a **`B`**, and throws an Exception if it doesn't
- Is Python type safe?
  - Duck typing
    - an object's suitability is determined by the presence of certain methods and properties, rather than the type of the object

Equality in the Reference Model

- Two kinds of equality
  - Reference equality
    - Do the variables refer to the same object
    - Java ==
      - For primitives (int, char, etc.), == compares values
    - Python *is*
  - Value equality
    - Do the variables reference objects with the same value
    - Java .equals()
    - Python ==

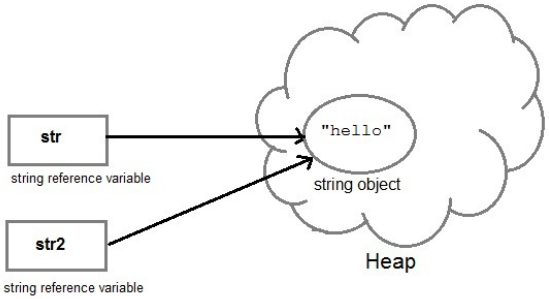
Testing for Equality



- Testing for equality: == and equals()

```
public class Point2D {
        int x,y;  // package access

        // constructor
        Point2D(int x, int y) {
                this.x = x;
                this.y = y;
        }

        public static void main(String[] args) {
                Point2D a = new Point2D(2, 5);
                Point2D b = new Point2D(2, 5);
                Point2D c = b;

               // what does this print?
                System.out.println(a == b);
                System.out.println(b == c);

                }
}
```

What About Strings?



- Testing for equality: == and equals()

```
public class StringTest {

        public static void main(String[] args) {
                String a = new String("cat");
                String b = new String("cat");

                String c = b;

                // what does this output?
                System.out.println(a == b);
                System.out.println(c == b);

                System.out.println(a.equals(b));
                System.out.println(b.equals(c));
        }
}
```

Point2D inherits Object.equals()

```
public class Point2D {
        int x,y;

        Point2D(int x, int y) {
                this.x = x;
                this.y = y;
        }

        public static void main(String[] args) {
                Point2D a = new Point2D(2, 5);
                Point2D b = new Point2D(2, 5);

                Point2D c = b;

                System.out.println(a.equals(b));
                System.out.println(b.equals(c));
        }
}
```

Point2D Inherits equals from Object Class

- java.lang.Object is the top-level class
- Every other class is a subclass of Object
  - Every class implements its methods
  - It is the superclass of all classes
  - Like *object* in Python
- It's a very simple class
  - https://docs.oracle.com/javase/10/docs/api/java/lang/Object.html
  - Object.equals uses ==
  - Tests for identity; does not test for equality of contents

Point2D inherits Object.equals()

```
public boolean equals( Object obj ) {
      return this == obj;
}
```

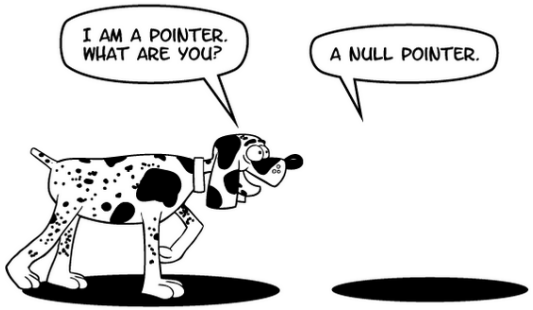
String Class Overrides the equals() Method

```
String a = new String( "ABCD" );
String b = new String( "ABCD" );

a == b                  // true or false?  false
a.equals( b )           // true or false?  true
```

- String.equals compares contents
- String class overrides Object.equals

- For Point2D to test for equal contents, it must override Object.equals.
- The way it is currently written, it does not.

Java Throws Lots of Exceptions!



E.g.:

`ArrayIndexOutOfBounds` at **`x[i]=0;`** if **`i`** is out of bounds for array **`x`**

`ClassCastException` at **`B q = (B) x;`** if the runtime type of **`x`** is not a **`B`**

`NullPointerException` at **`x.f=0;`** if **`x`** is null

Java Throws Lots of Exceptions

- Exceptions are a good thing!
  - Tells us what went wrong
  - Prevents application of operation on wrong type
    - --- stops program from doing harm down the road

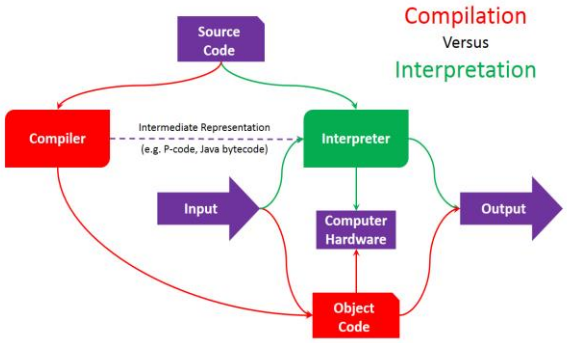
```
Object x = new A();
B q = (B) x; // ClassCastException
                    // because A is not a B
… int case1 = q.foo(1);
```

  - Exception prevents execution from reaching **`q.foo(1)`** and applying **`foo(1)`** on an object (**`A`**) that does not support **`foo(int)`**

Java Throws Lots of Exceptions

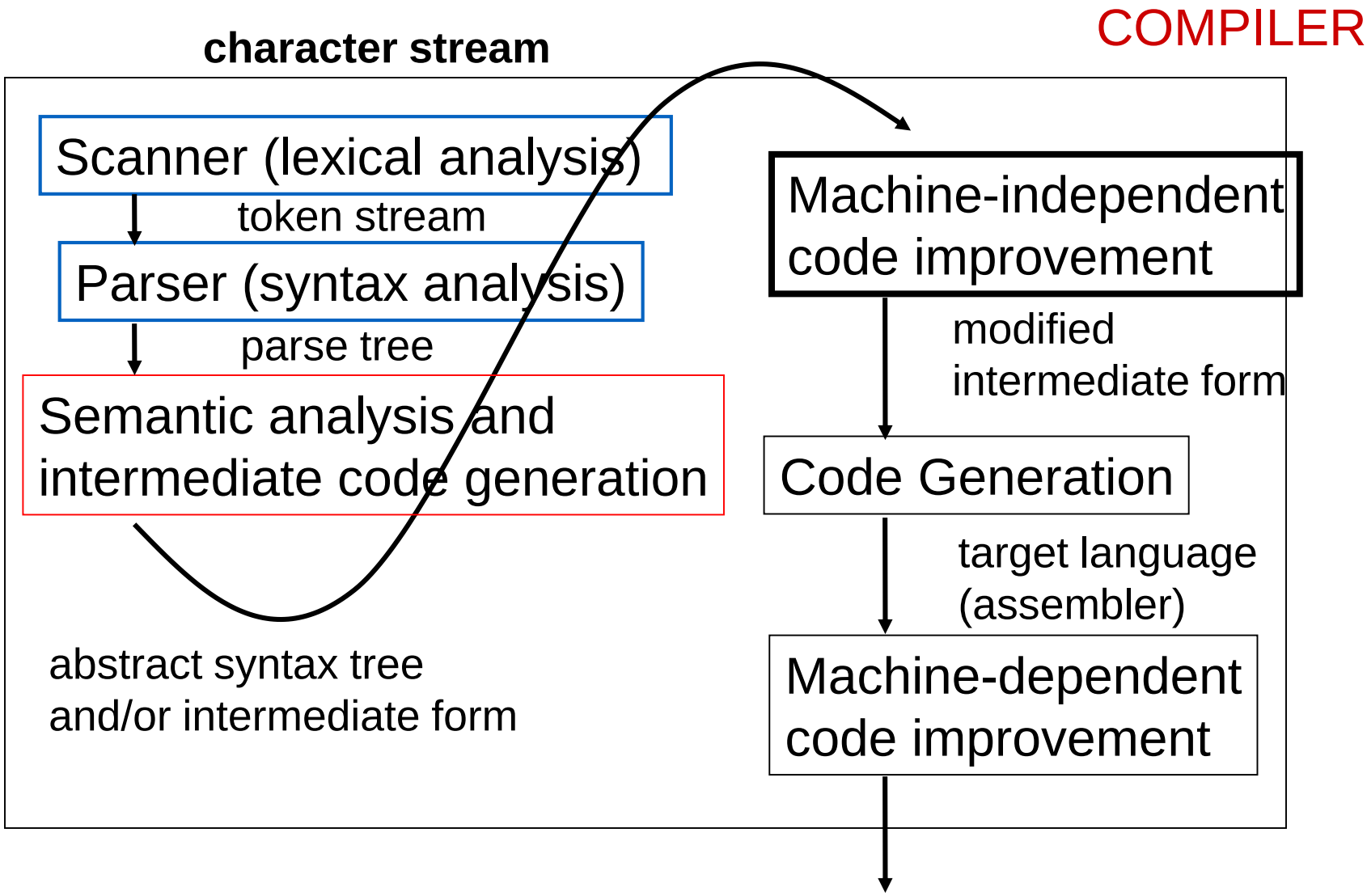

```
class A {
        A() {
                System.out.println("I'm an A");
        }

        void foo() {
                System.out.println("A foo");
        }
}

class B extends A {
        B() {
                System.out.println("I'm a B");
        }

        void foo() {
                System.out.println("B foo");
        }

        void foo(int x) {
                System.out.print("B foo int ");
                System.out.println(x);
        }

}

public class ExceptionTest {

        public static void main(String[] args) {
                Object a = new A();
                B b = (B) a; // this is not allowed
        }

}
```

Compilation vs. Interpretation

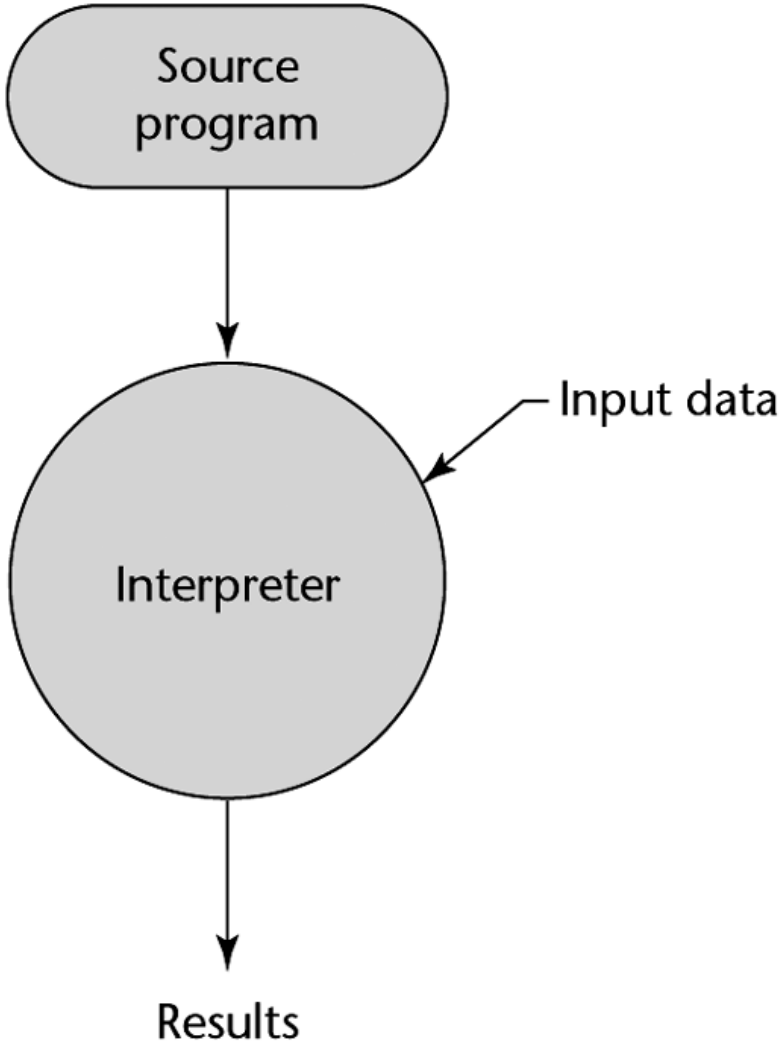


- Compilation
  - A "high-level" program is translated into executable machine code
  - C++ uses compilation
- Pure interpretation
  - A program is translated and executed one statement at a time
  - Interpreter
- Hybrid interpretation
  - A program is "compiled" into intermediate code; intermediate code is "interpreted"
  - Both a compiler and an interpreter.
    - Java

Compilation



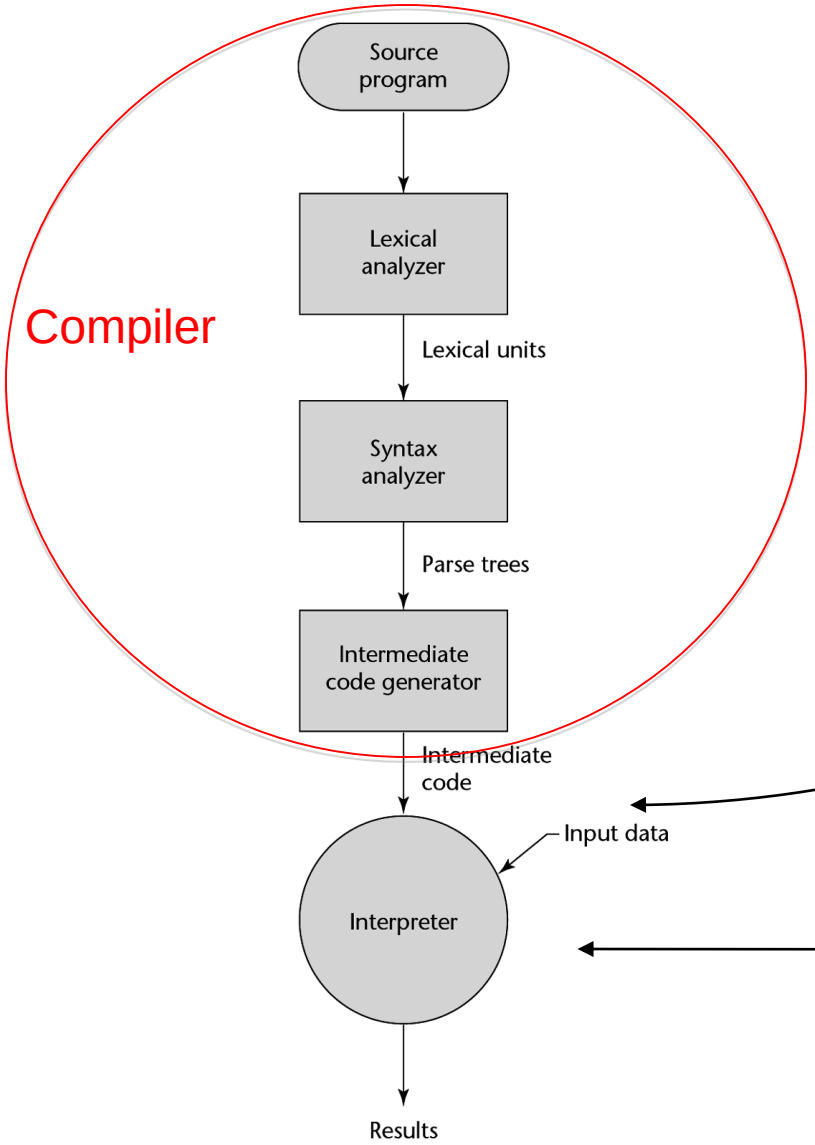
Pure Interpretation



e.g. BASIC

```
REM
COMMENT
LET X = 5
LET Y = 8
PRINT X
PRINT Y
LET Z = X
PRINT Z
...
```

Hybrid Interpretation




```
09 AB 19 29
99 73 09 AB
19 29 99 73
09 AB 19 29
99 73 09 AB
19 29 99 73
09 AB 19 29
99 73 09
```

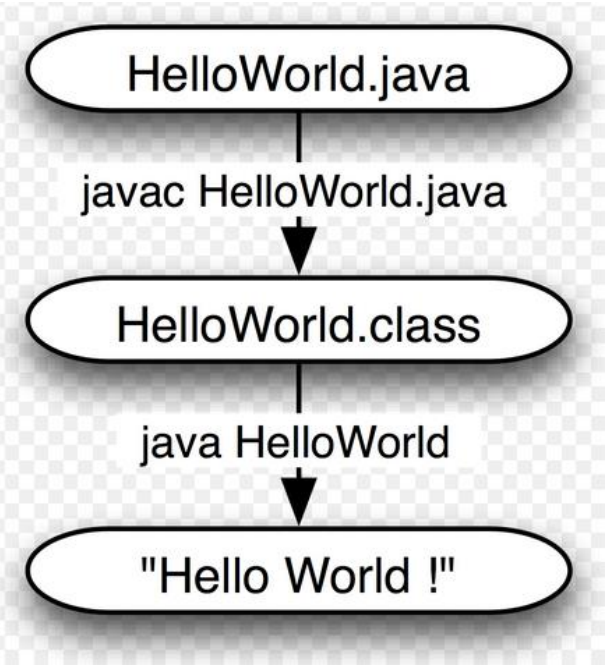
e.g. Java byte code

e.g. Java Virtual Machine (JVM)

Also Perl, Python etc.

Compiling and Running Java

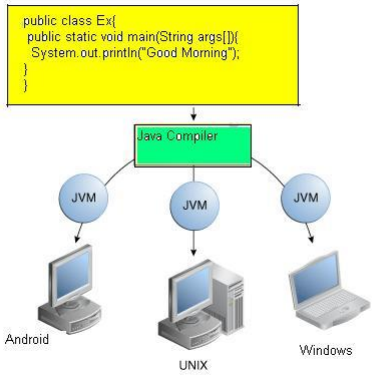
- Command line:
  - javac HelloWorld.java produces HelloWorld.class
  - java HelloWorld // runs the interpreter
- Eclipse:
  - Automatically parses for syntax errors
  - Compiles automatically when you save!
  - Run -> Run runs the interpreter



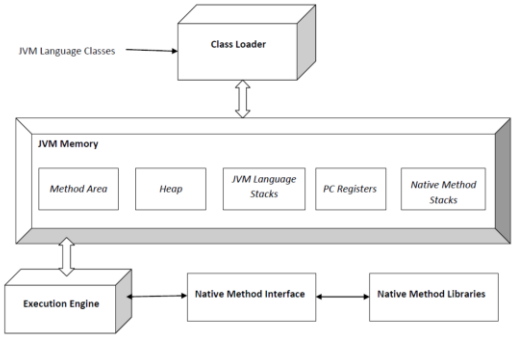
Compilation vs. Interpretation

- Advantages of compilation?
  - Faster execution
    - Usually, but it depends on many factors
- Advantages of interpretation?
  - Greater flexibility
    - Portability, sandboxing,
    - dynamic semantic (i.e., type) checks
    - other dynamic features are much easier
    - Easier to write an interpreter

Java Virtual Machine



- Java virtual machine (JVM) I
  - a virtual machine that enables a computer to run Java programs
  - Runs programs written in other languages that are compiled to Java bytecode
    - Clojure, a functional Lisp dialect
    - Groovy, a dynamic programming and scripting language
    - JRuby, an implementation of Ruby
    - Jython, an implementation of Python
    - Kotlin, a statically-typed language from JetBrains, the developers of IntelliJ IDEA
    - Scala, a statically-typed object-oriented and functional programming language
    - Many others
    - https://en.wikipedia.org/wiki/List_of_JVM_languages



Some Terminology

- C++: Base class and derived class
- Java: Superclass and subclass
- C++: Member variable, member function
- Java: field (instance or static), method (again instance or static)
- Java has interfaces (collections of method signatures)
  - Single class inheritance (**`class B extends A {`**…),
  - Multiple interface inheritance (**`class A implements I,J,K {`**…)
  - **`class B extends A implements I,J,K {`**…

Subtyping and Dynamic Method Binding

- Subtyping and subtype polymorphism
  - The ability to use a subclass where a superclass is expected
  - Open for extension but closed for modification

```
abstract class Shape {
   abstract void draw();
}

class Circle extends Shape { ... }  // implements draw()
class Square extends Shape { ... } // implements draw()

void DrawAll( Shape[] list ) {
   for ( int i = 0 ; i < list.length ; i++ ){
      Shape s = list[i];
      s.draw();
   }
}
```

Subtyping and Dynamic Method Binding

- In C++, static binding is default
  - dynamic binding is specified with keyword *virtual*
- In Java, dynamic binding is default
  - *private*, *final* or *static* methods in a class use static binding.
- Static binding happens at compile time
  - Static methods cannot be overridden
  - Compiler knows class of object
- Dynamic binding happens at runtime
  - Methods can be overridden
  - Compiler doesn't know what type object will be at runtime

Difference Between Static and Dynamic Binding in Java

- Static binding occurs at **compile time**
- Dynamic binding occurs at **runtime**
- **Overloaded** methods are bound using static binding
- **Overridden** methods are bound using dynamic binding
  - Overloading is the ability to use same interface name but with different arguments
    - void foo() and void foo(int)
  - Overriding is used in the context of inheritance
    - When there is a need to change the behavior of a particular inherited method, it will be overridden in the sub-class.
    - Argument types must be the same, but return type may be different

Benefits of Subtype Polymorphism

- Enables extensibility and reuse
  - we can extend the Shape hierarchy further modifying Circle, Square, DrawAll() method
  - We can reuse Shape and DrawAll
- Subtype polymorphism enables the open/closed principle
  - *open to extension, but closed to modification*

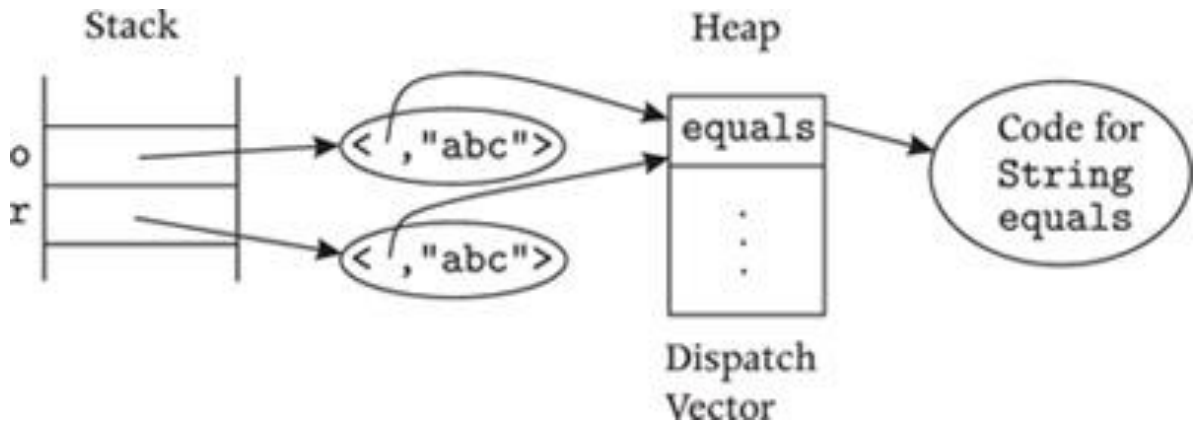
Dispatching

```
String t = new String("ab‟);
Object o = t + "c"; // concatenation
String r = new String("abc‟);
boolean b = o.equals(r);
```

- Are o and r “equal”?
  - What does “equal” mean here?
  - Object *equals* compares references
  - String *equals* compares values
- The compiler doesn’t necessarily know what code to call at compile time
  - Compiler knows the apparent type
    - o - Object
  - At runtime, the object has an actual type
    - o - String

Dispatching

- Dispatching calls code in the actual type



Returns true