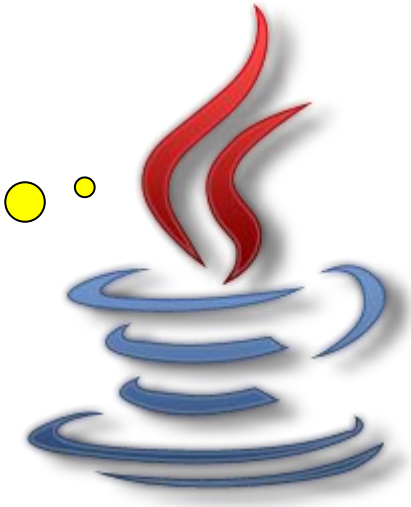
From **A** to **B** to **C** to **C++**

Interpreted language implementation.

Compiled language implementation

C/C++, Python, Java a Language Overview

Compiled language implementation, *sort of*...



Features common to all High Level Programming Languages

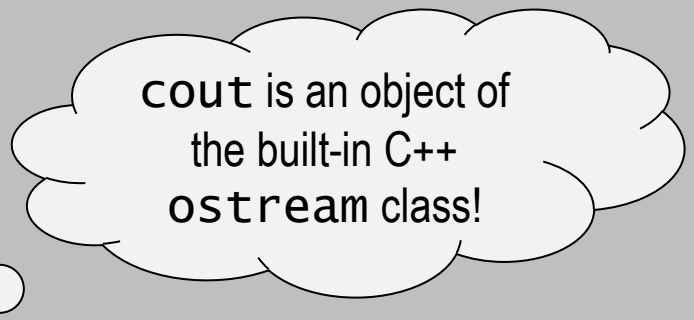
- **Provides a higher-level abstraction from the underlying hardware**
  - *Allows symbolic names to represent memory cells*
  - *Operations do not depend on instruction set*
  - *I/O is independent of the I/O device*
- Provides expressiveness
  - *Meaningful symbols convey meaning*
  - *Can use simple logical expressions for common control patterns (if-else, while, switch, etc).*
- Enhances code readability
  - *Good choice of variable and function names, along with proper indentation and line spacing, allow us to program in a self-documenting manner.*

Hello World in C

```c
/*
 * hello.c
 *
 * Hello World Example in C
 *
 * Christine Papadakis-Kanaris
 *
 * CS611
 *
 */
#include <stdio.h>

int main() {

   // statement to output the string Hello World
   printf( "Hello World" );

   return 0;
}
```

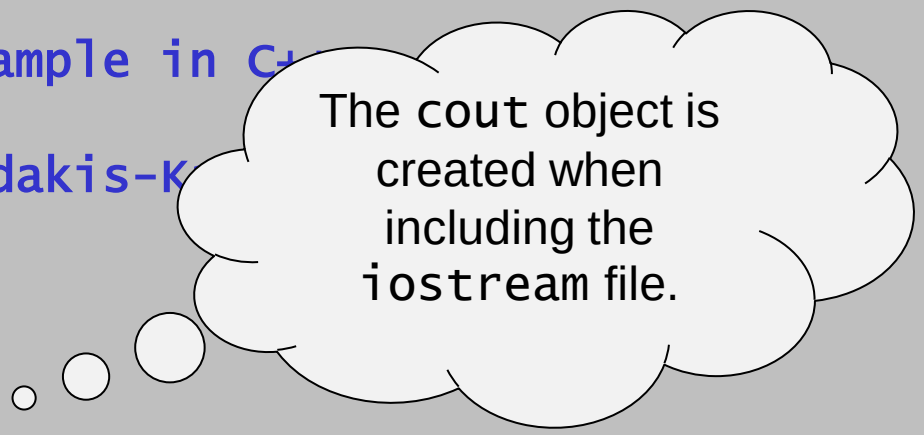
Hello World in C++

```
/*
 * hello.cc
 *
 * Hello World Example in C++
 *
 * Christine Papadakis-Kanaris
 *
 * CS611
 *
 */
#include <iostream>

int main() {

   // statement to output the string Hello World
   std::cout << "Hello World!";

   return 0;
}
```



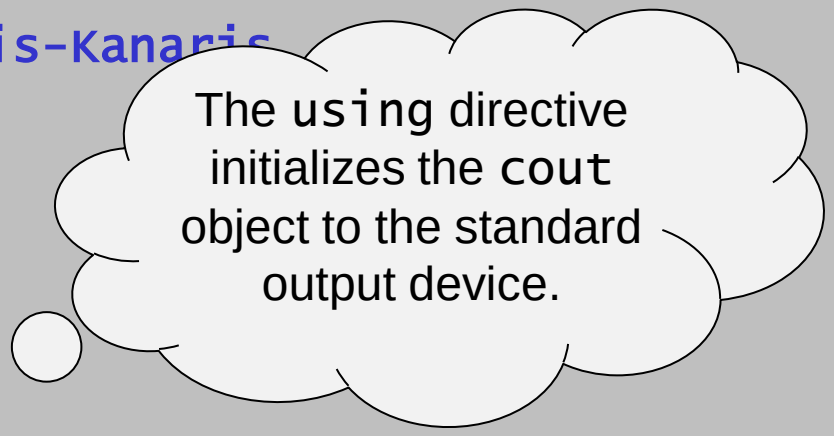
Hello World in C++

```
/*
 * hello.cc
 *
 * Hello World Example in C+
 *
 * Christine Papadakis-K
 *
 * CS611
 *
 */
#include <iostream>

int main() {

   // statement to output the string Hello World
   std::cout << "Hello World!";

   return 0;
}
```



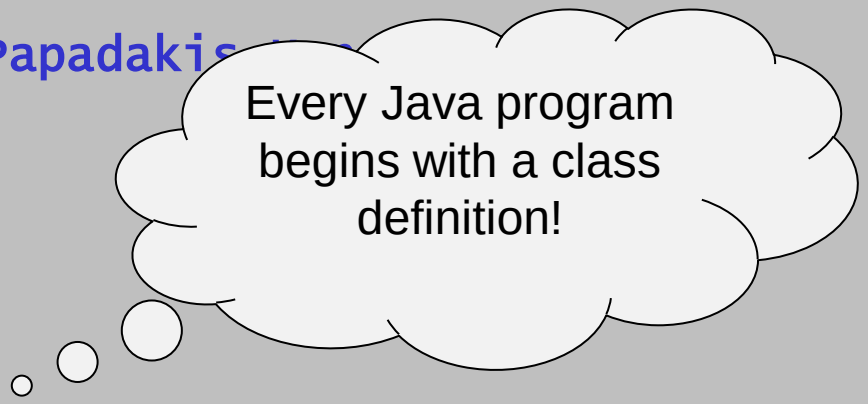
Hello World in C++

```
/*
 * hello.cc
 *
 * Hello World Example in C++
 *
 * Christine Papadakis-Kanaris
 *
 * CS611
 *
 */
#include <iostream>
using std::cout ;
int main() {

   // statement to output the string Hello World
   cout << "Hello World!";

   return 0;
}
```



Hello World in Java

```
/*
 * helloWorld.java
 *
 * Hello World Example in Java
 *
 * Christine Papadakis
 *
 * CS611
 *
 */

public class helloWorld {
   public static void main( String[] args ) {
       // statement to output Hello World
        System.out.println( "Hello World" );

   }
}
```



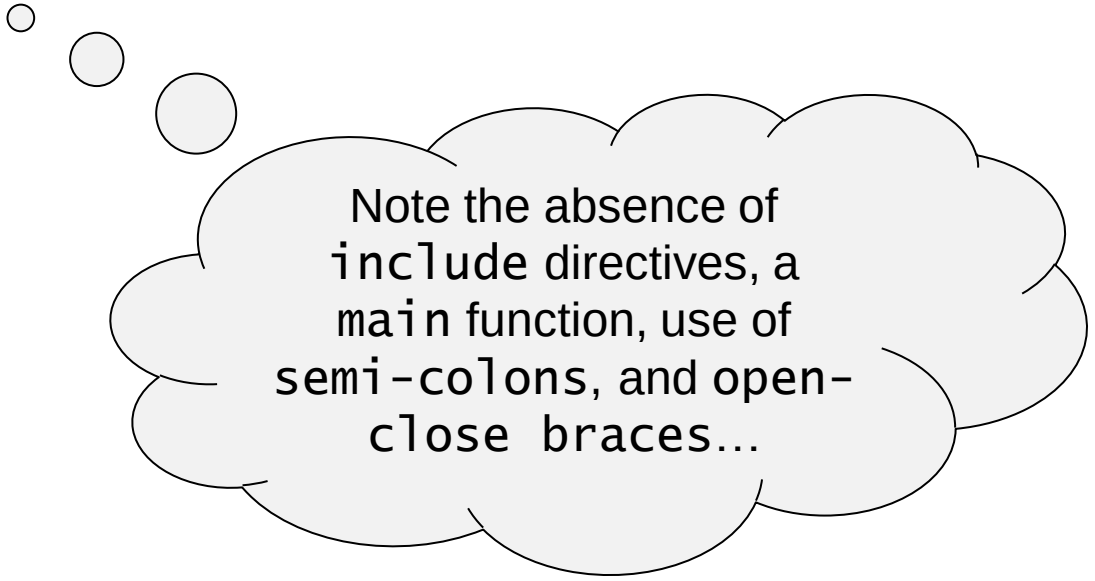
Hello World in Java

```
/*
 * helloWorld.java
 *
 * Hello World Example in Java
 *
 * Christine Papadakis
 *
 * CS611
 *
 */

public class helloWorld {
   public static void main( String[] args ) {
       // statement to output Hello World
        System.out.println( "Hello World" );

   }
}
```

Note the absence of `include` directives!

Hello World in Python

```
# hello.py
# Hello World Example in Python
# Christine Papadakis-Kanaris
# CS611

print('Hello World!')
```

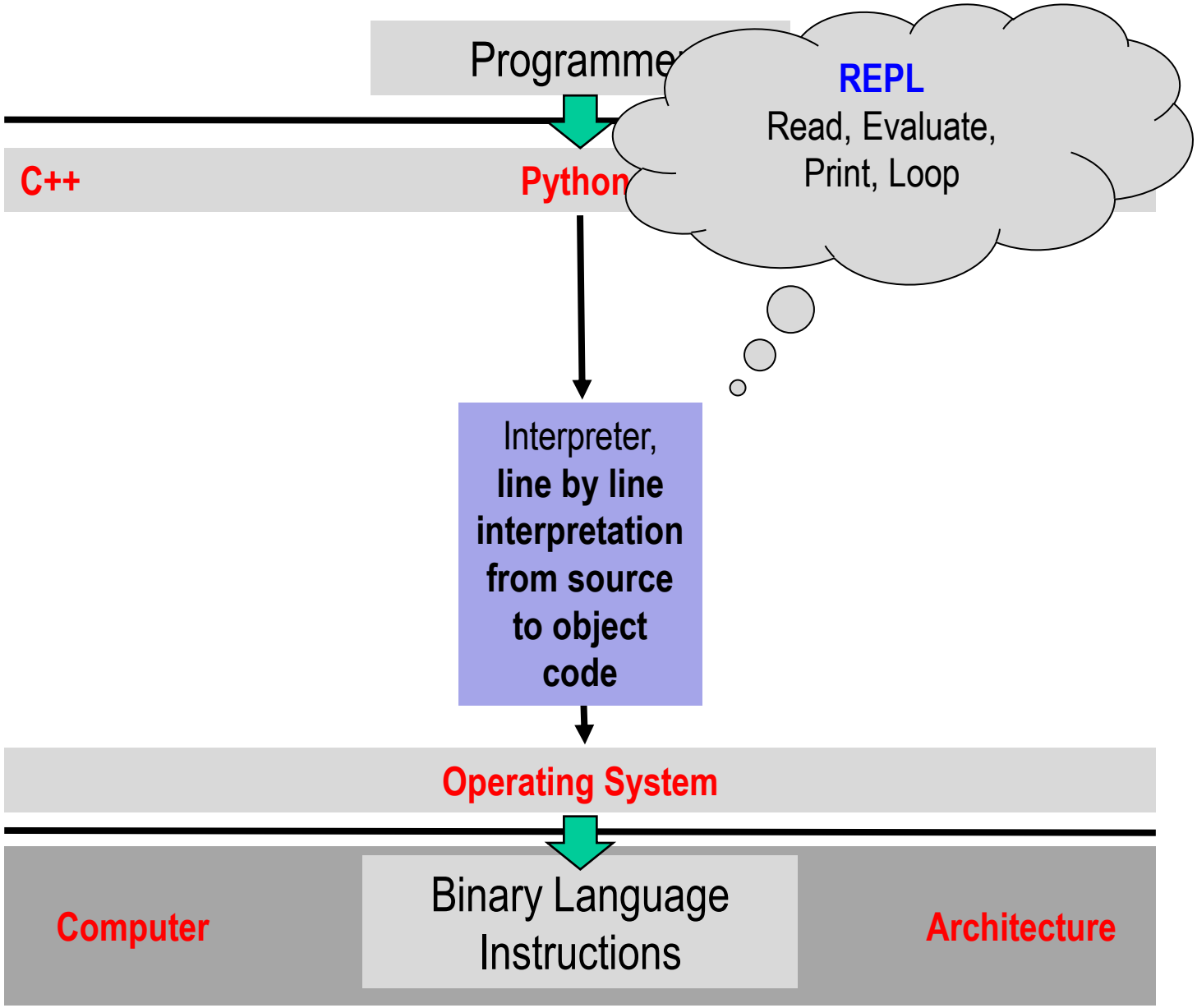


Interpretation vs. Compilation

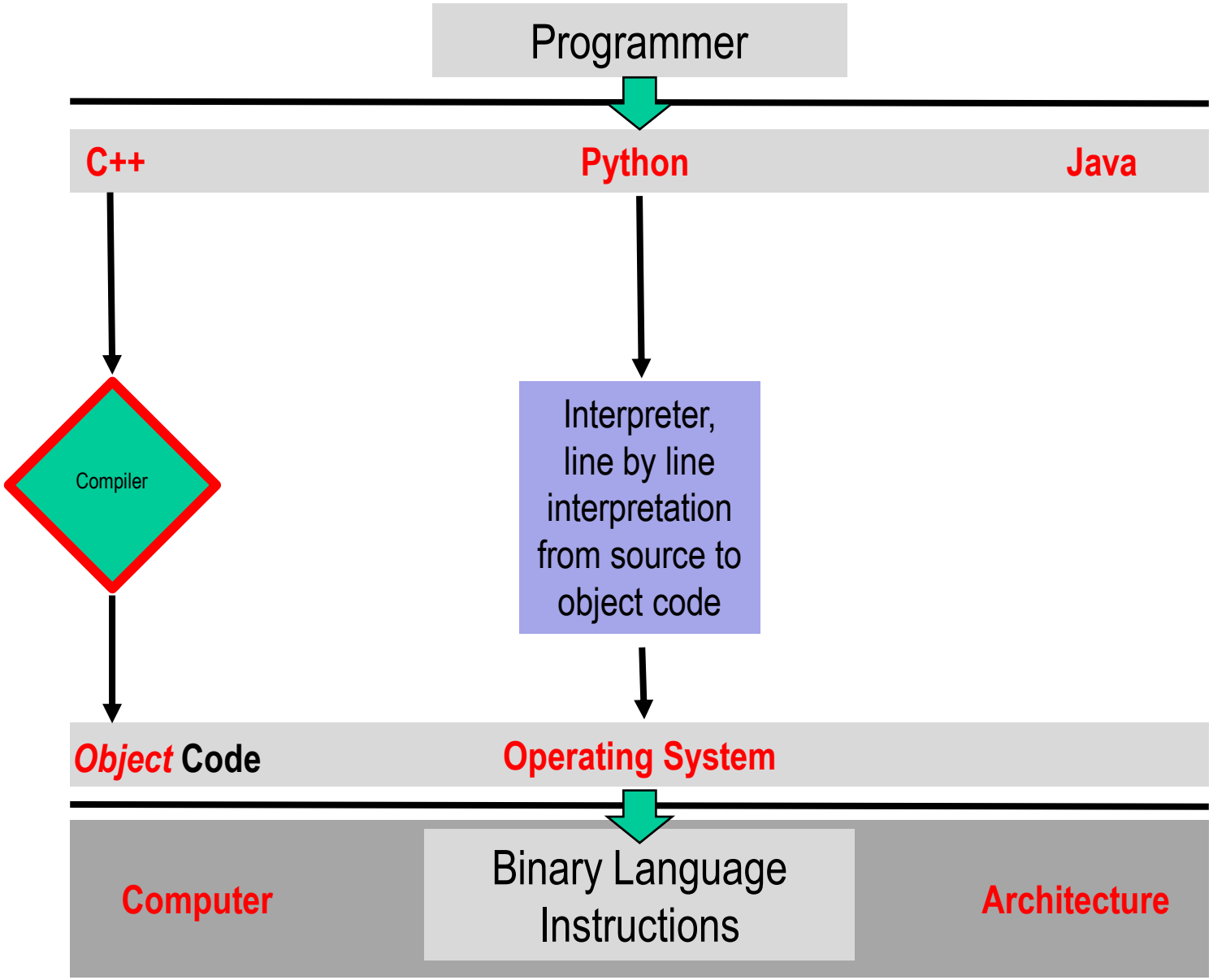
- *Languages with an Interpretation Implementation*
  - *Python, Visual Basic, C-shell, and even Java*
  - Program statements are interpreted and executed one line/statement at a time.
  - Interpreted languages may be easier to debug, make changes, and view intermediate results

- *Languages with a Compilation Implementation*
  - *C, C++, Fortran, and even Java*
  - Programs are compiled into machine language
    - the compiler does not execute program code, but creates an executable program
  - Allows the compiler to perform code enhancements and optimization
  - Can be harder to debug, must recompile after each change.

Compiled vs. Interpreted

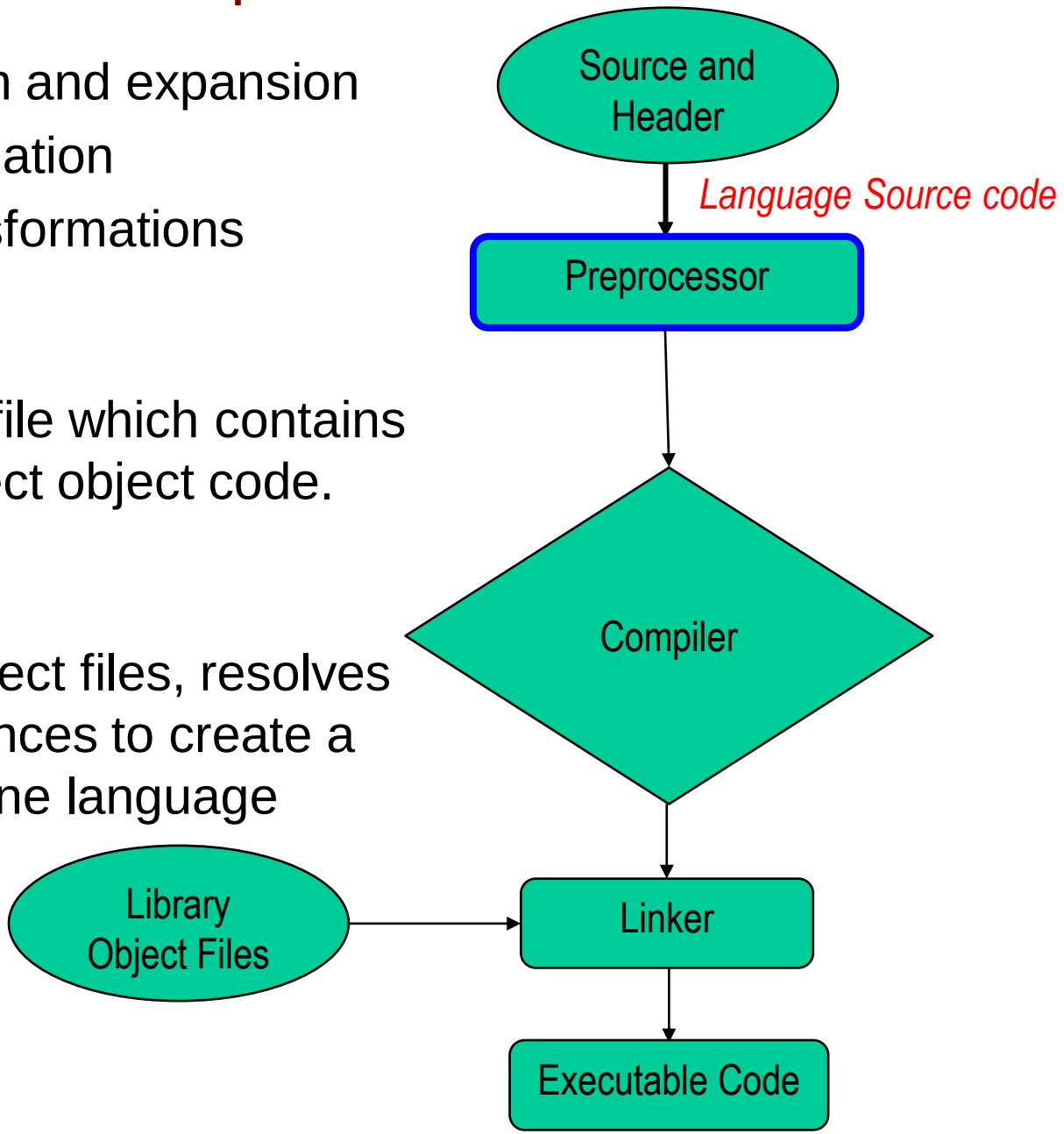


Compiled vs. Interpreted



C/C++ Compilation Process

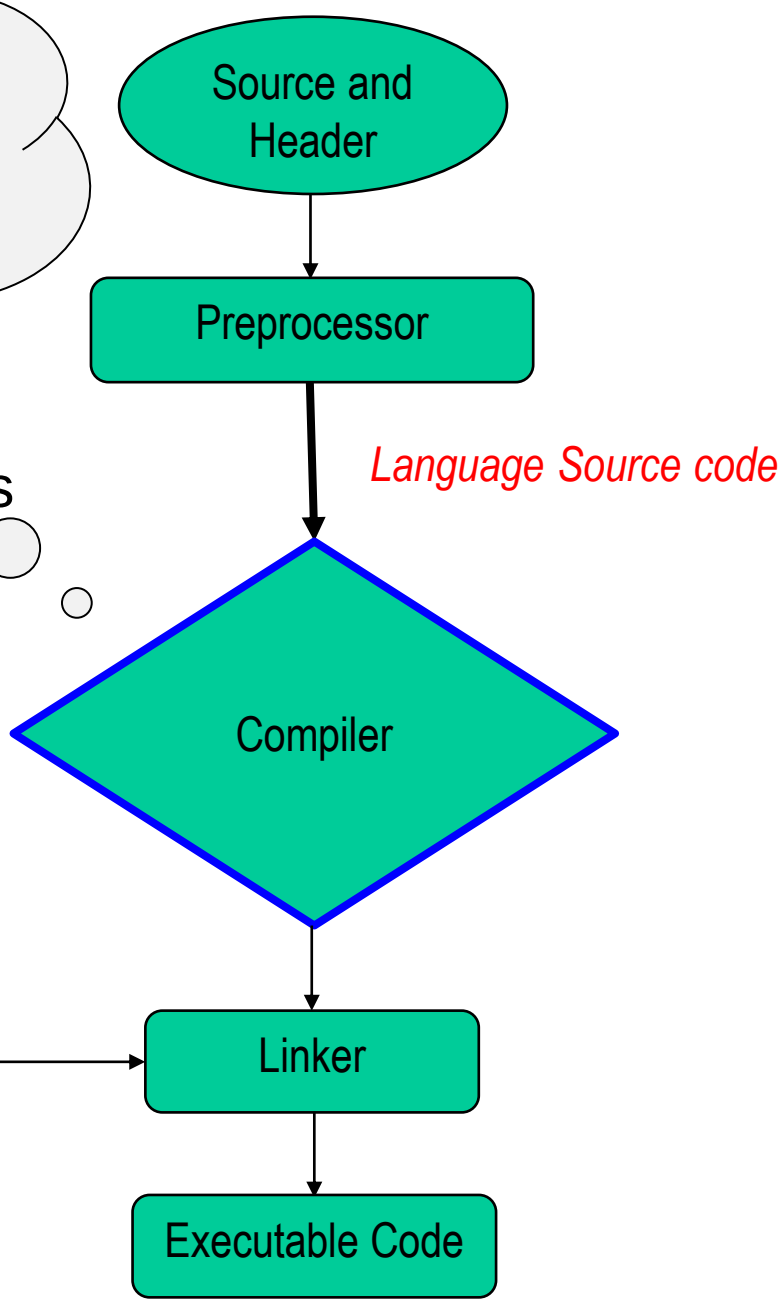
- **Preprocessor**
  - macro substitution and expansion
  - conditional compilation
  - source code transformations
- Compiler
  - generates object file which contains syntactically correct object code.
- Linker
  - Pulls in library object files, resolves all function references to create a executable machine language image.



Process

- Preproc
  - ma
  -
  -
- **Compiler**
  - generates object file which con ns syntactically correct object code.
- Linker
  - Pulls in library object files, resolves all function references to create a executable machine language image.

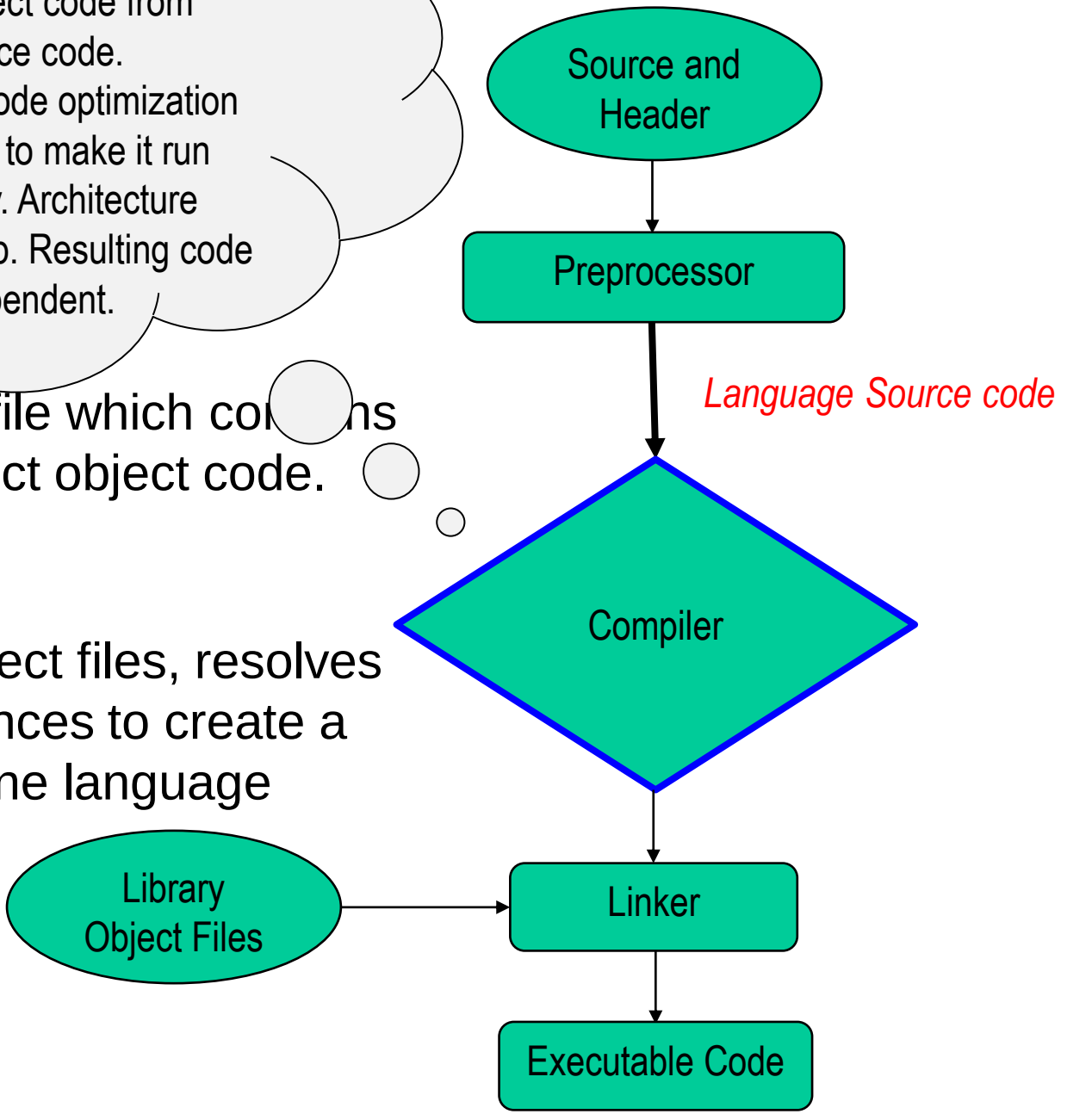
Parses and tokenizes source code to identify its pieces, variables, expressions, statements, functions, etc. depends on language. Architecture independent.



Process

- Prep
  -
  -
  -
- **Compiler**
  - generates object file which co ns syntactically correct object code.
- Linker
  - Pulls in library object files, resolves all function references to create a executable machine language image.

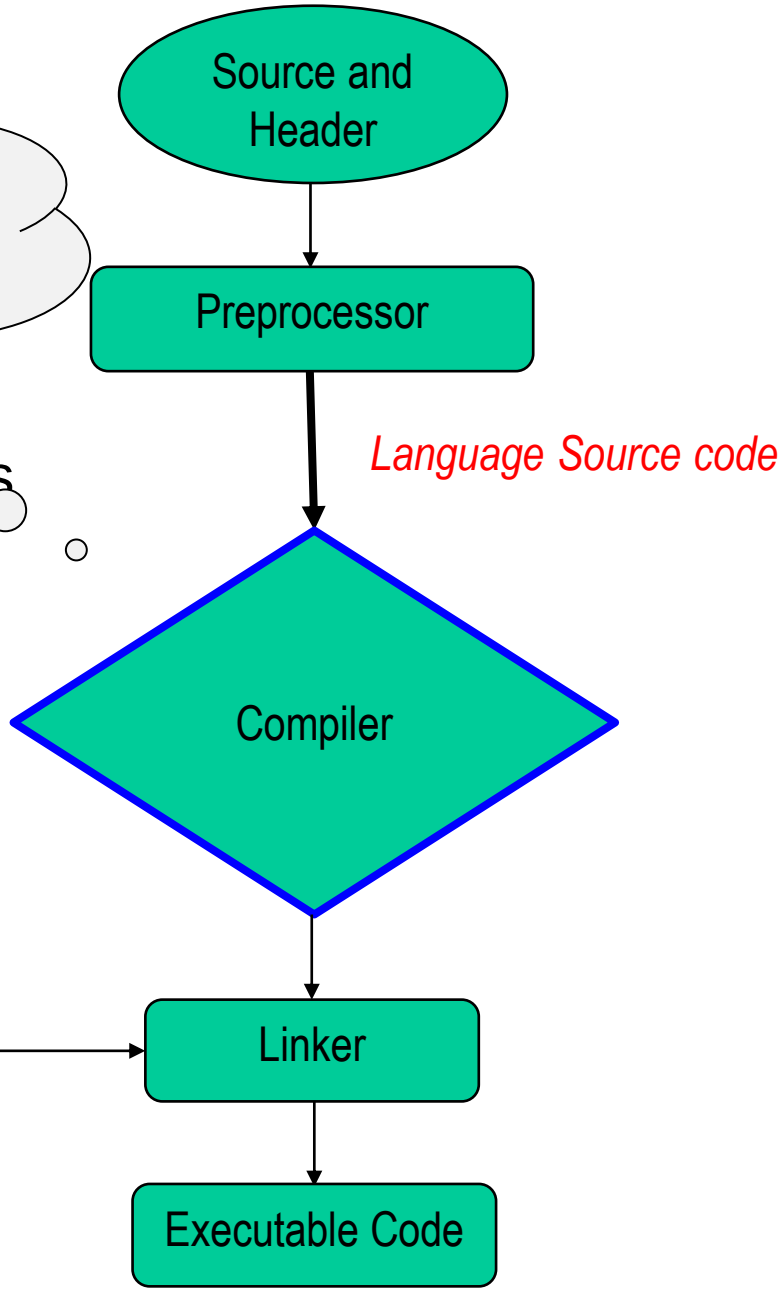
Generates object code from tokenized source code. May perform code optimization on object code to make it run more efficiently. Architecture dependent step. Resulting code is machine dependent.



C/C++ Compilation Process

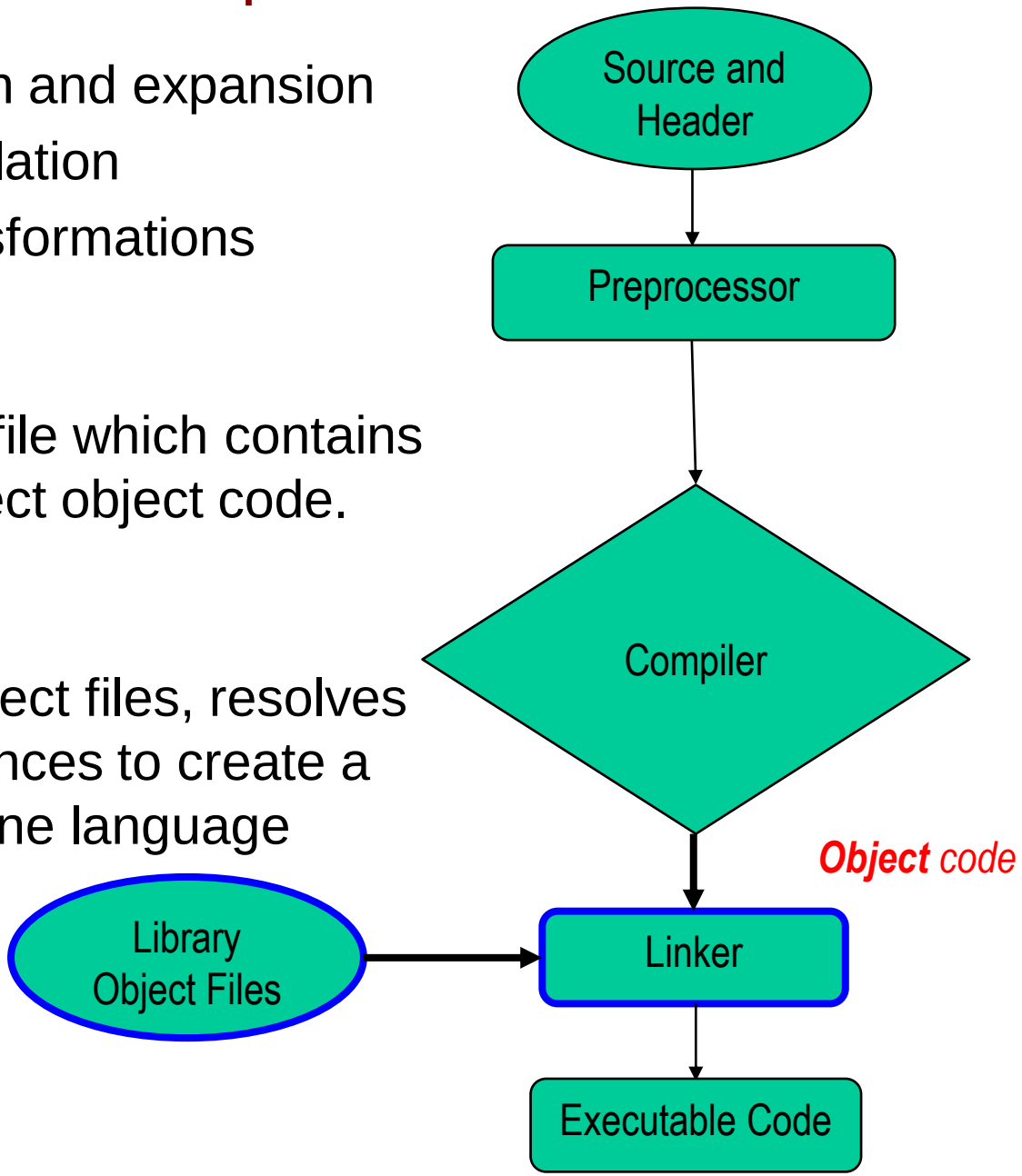
- Preprocessor
  - macro substitution and expansion
  - co
- Compi
  - generates object file which contains syntactically correct object code.
- Linker
  - Pulls in library object files, resolves all function references to create a executable machine language image.

Creates a symbol table which maps variables to memory cells.



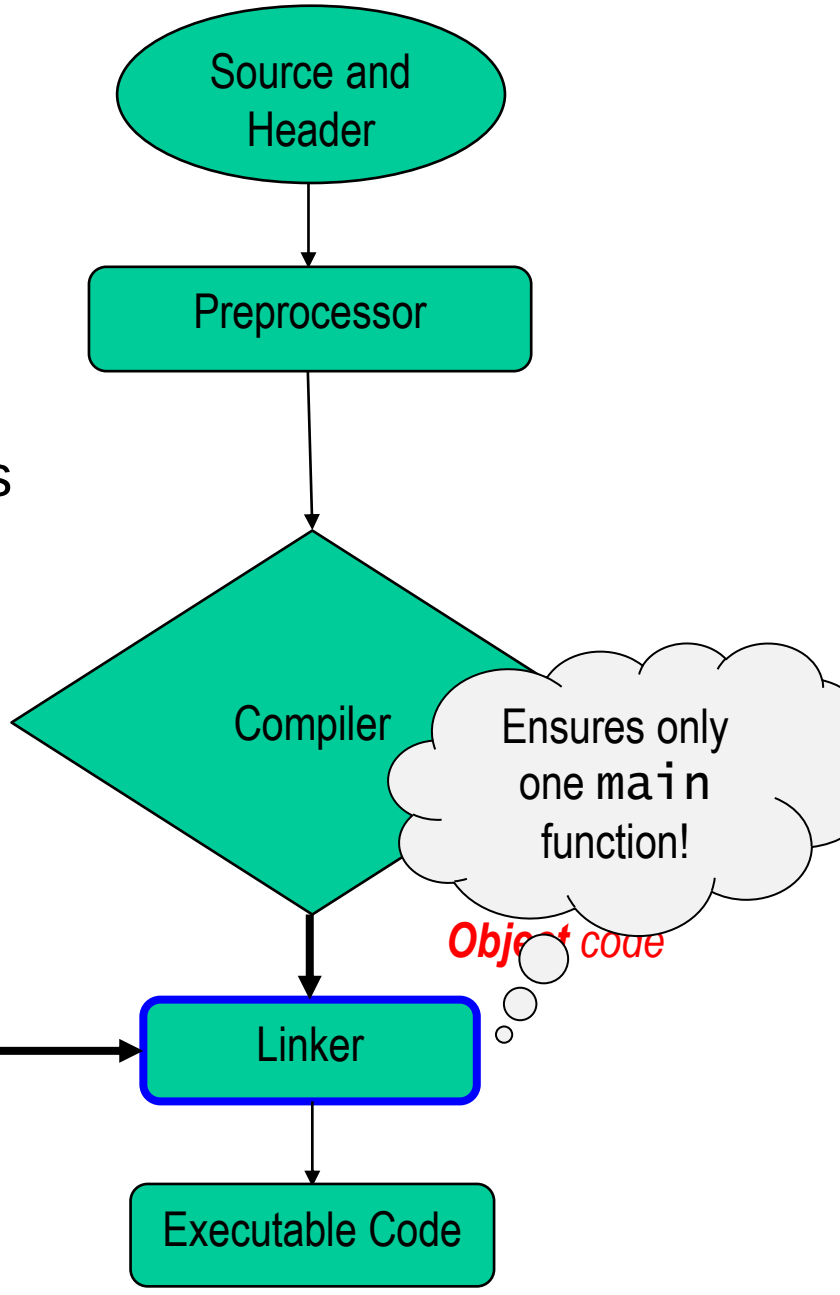
C/C++ Compilation Process

- Preprocessor
  - macro substitution and expansion
  - conditional compilation
  - source code transformations
- Compiler
  - generates object file which contains syntactically correct object code.
- **Linker**
  - Pulls in library object files, resolves all function references to create a executable machine language image.



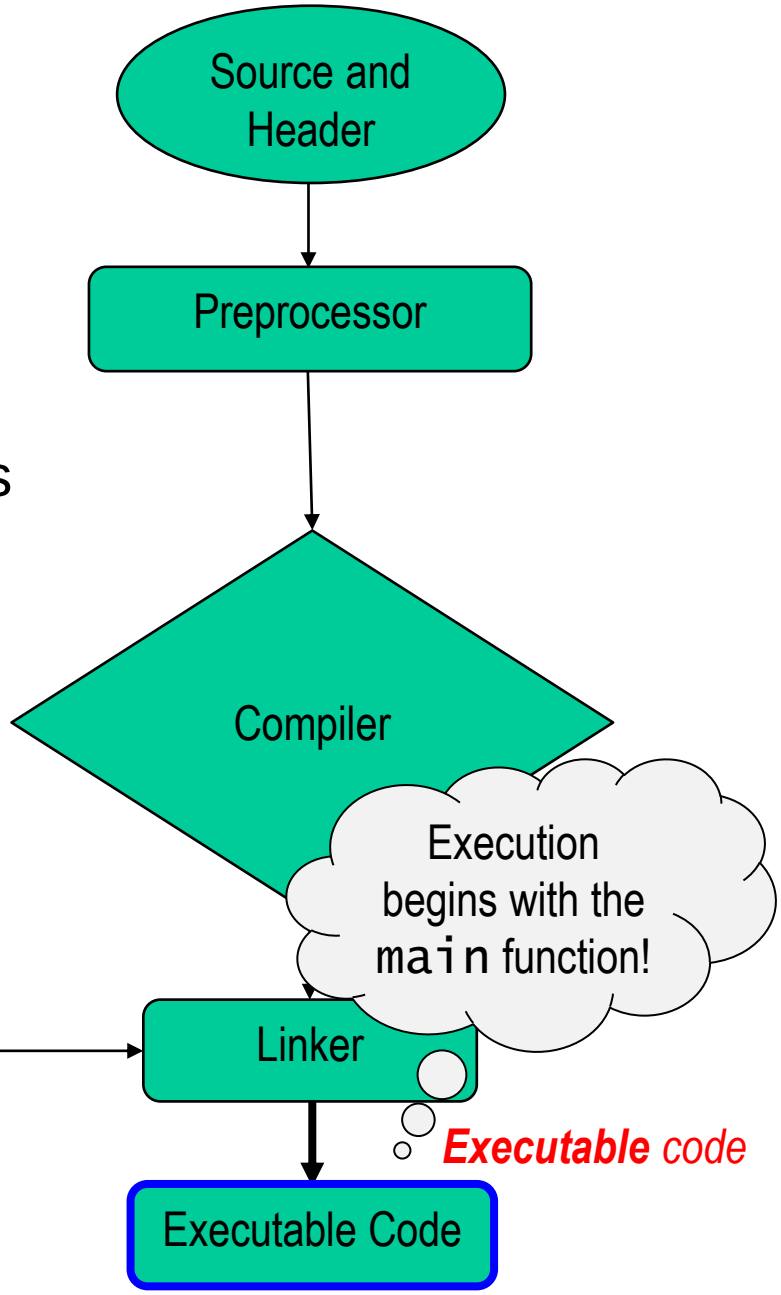
C/C++ Compilation Process

- Preprocessor
  - macro substitution and expansion
  - conditional compilation
  - source code transformations
- Compiler
  - generates object file which contains syntactically correct object code.
- **Linker**
  - Pulls in library object files, resolves all function references to create a executable machine language image.

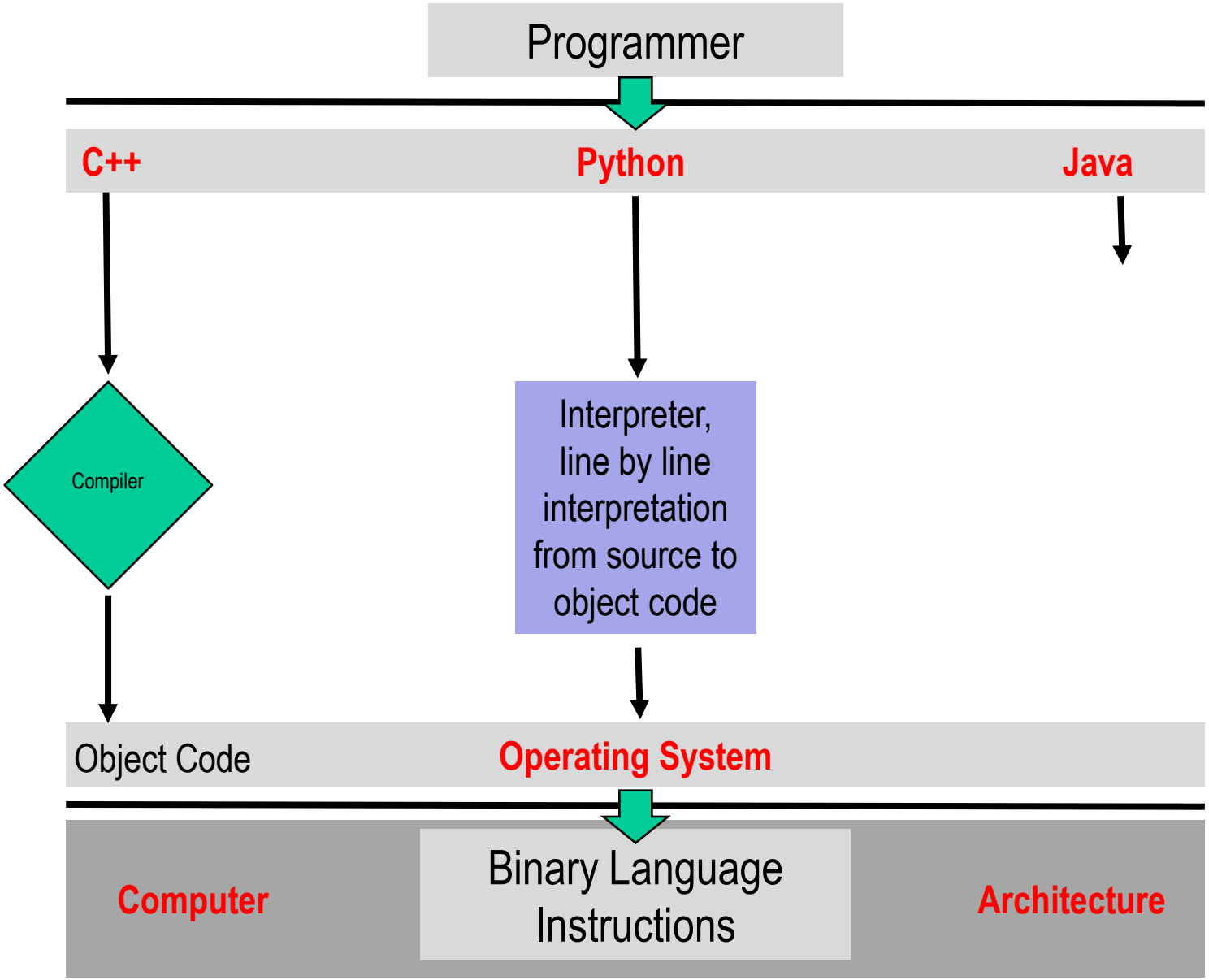


C/C++ Compilation Process

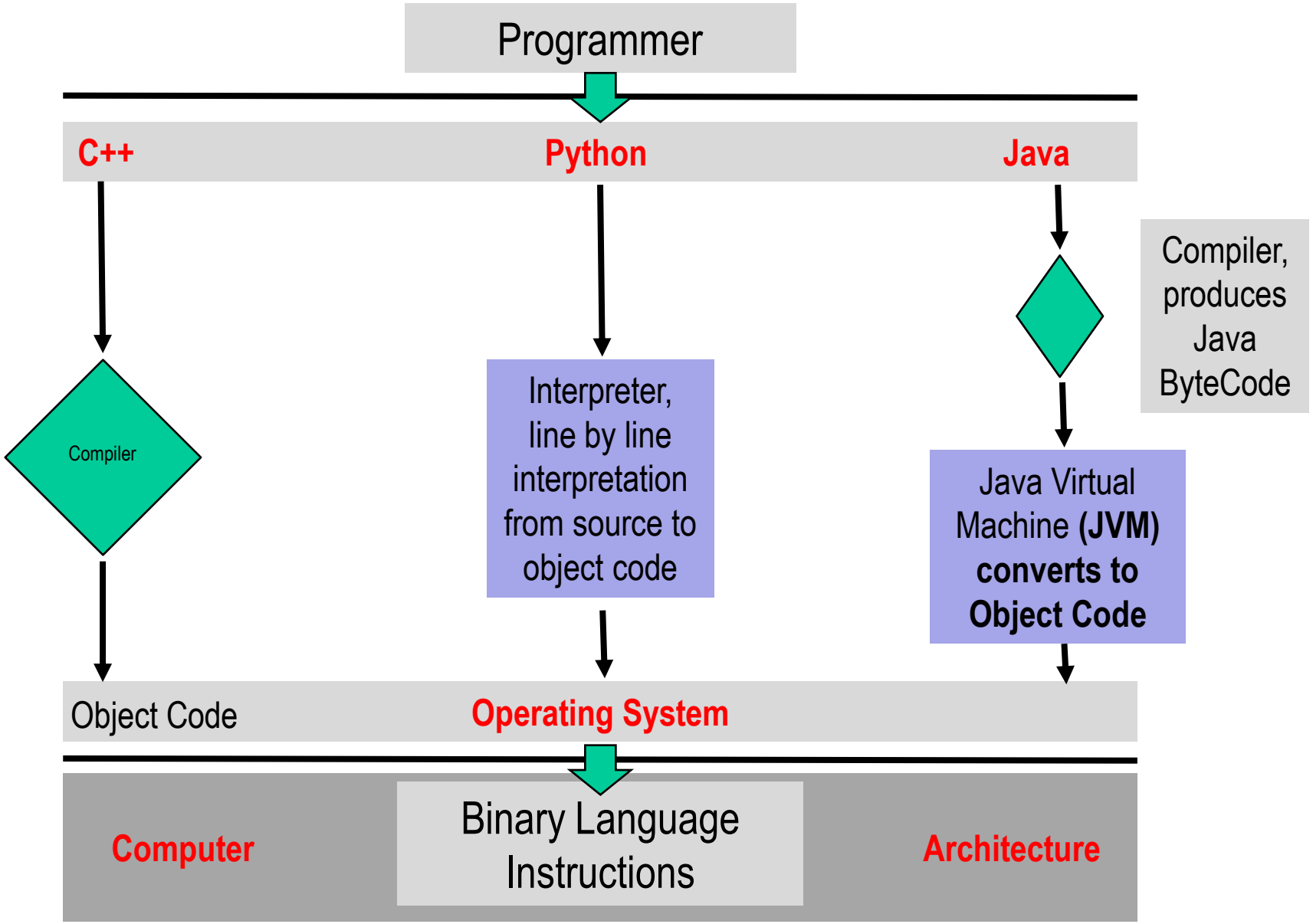
- Preprocessor
  - macro substitution and expansion
  - conditional compilation
  - source code transformations
- Compiler
  - generates object file which contains syntactically correct object code.
- **Linker**
  - Pulls in library object files, resolves all function references to create a executable machine language image.



Compiled vs. Interpreted



Compiled vs. Interpreted

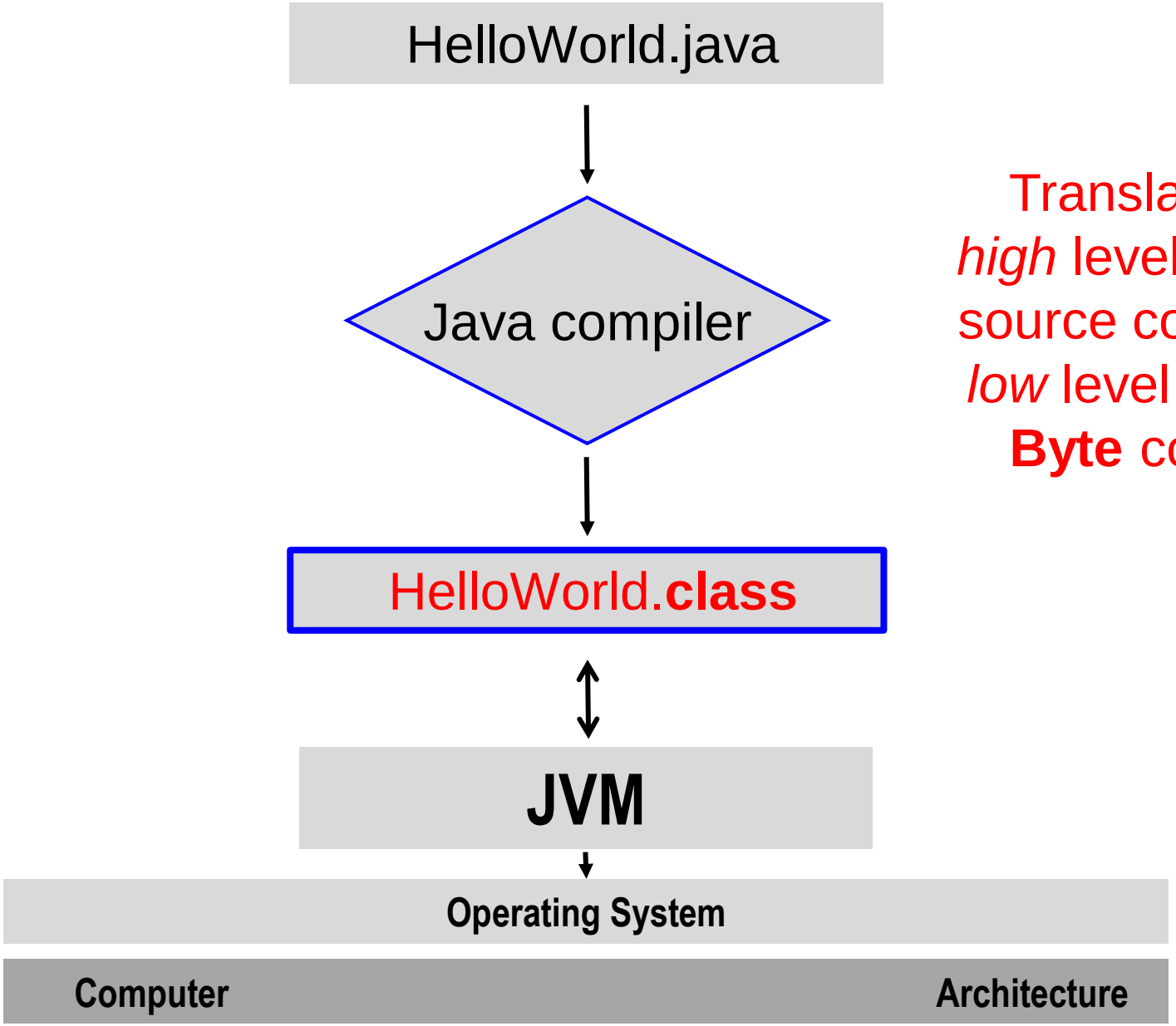


Hello World in Java:

*but the JVM will implicitly invoke the **main** method of the class*

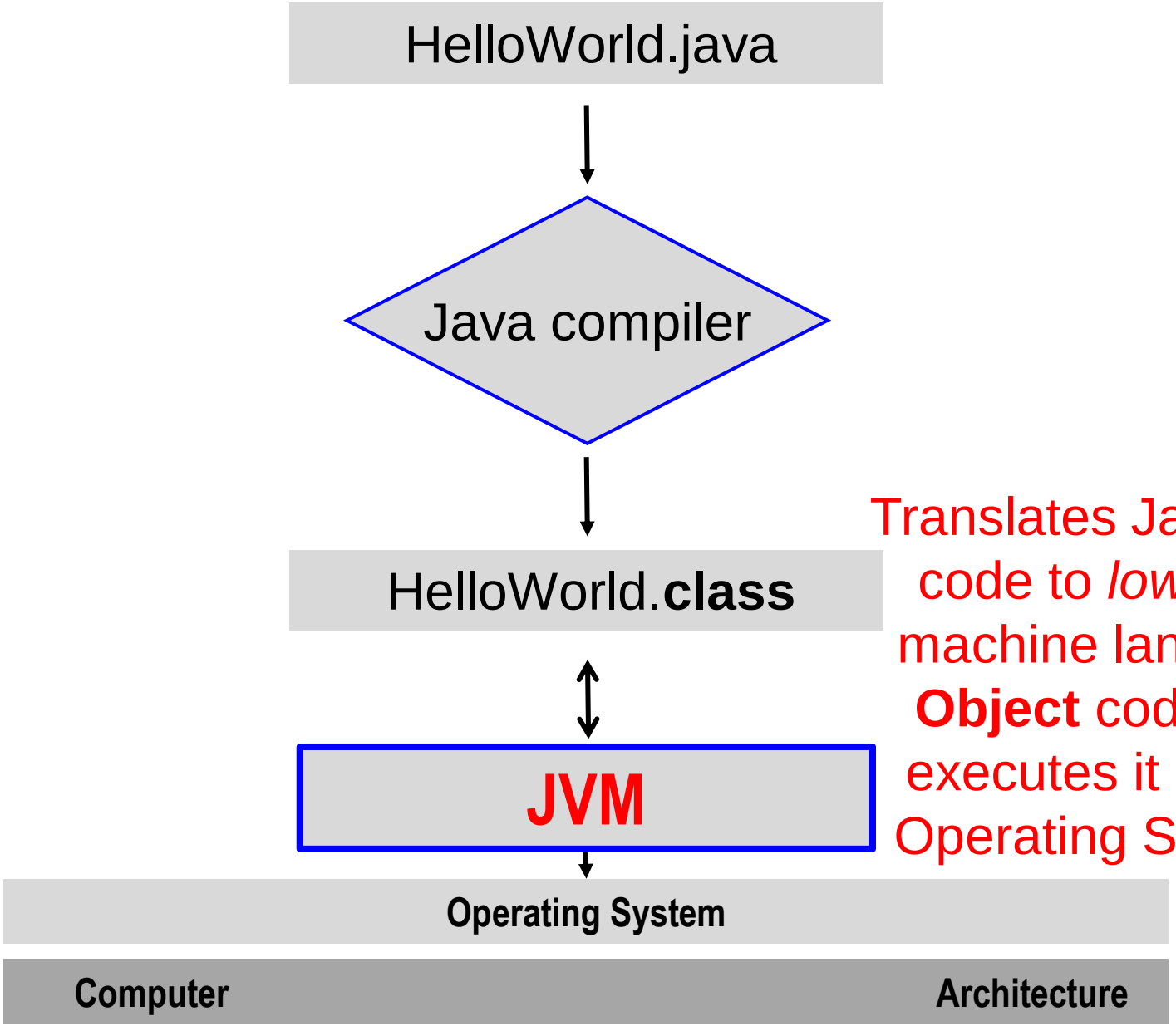
```
/*
 * CS611
 * ... Information as needed...
 */
public class helloWorld {
   public static void method2( ) {
       // body of method2
   }

   public static void main( String[] args ) {
       // statement to output Hello World
       System.out.println( "Hello World" );
       method1(); // call method1
   }

   public static void method1( ) {
       // body of method1
       method2(); // call method2
   }
}
```

Executing a Java Program



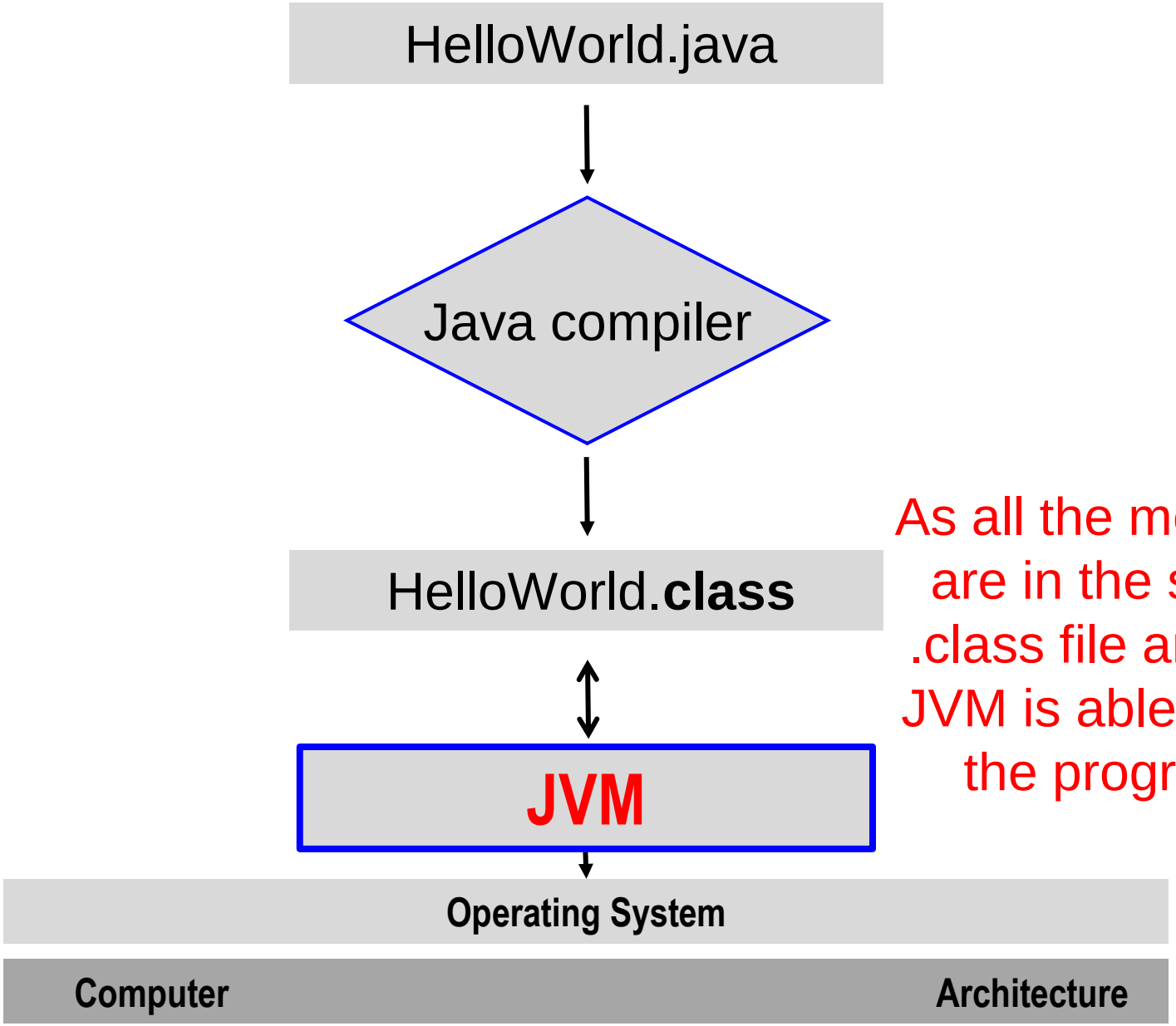
Translates *high* level Java source code to *low* level Java **Byte** code

Executing a Java Program



Translates Java byte code to *low* level machine language **Object** code and executes it on the Operating System.

Executing a Java Program



As all the methods are in the same .class file and the JVM is able to run the program.

Hello World in Java:

a program class can contain multiple methods

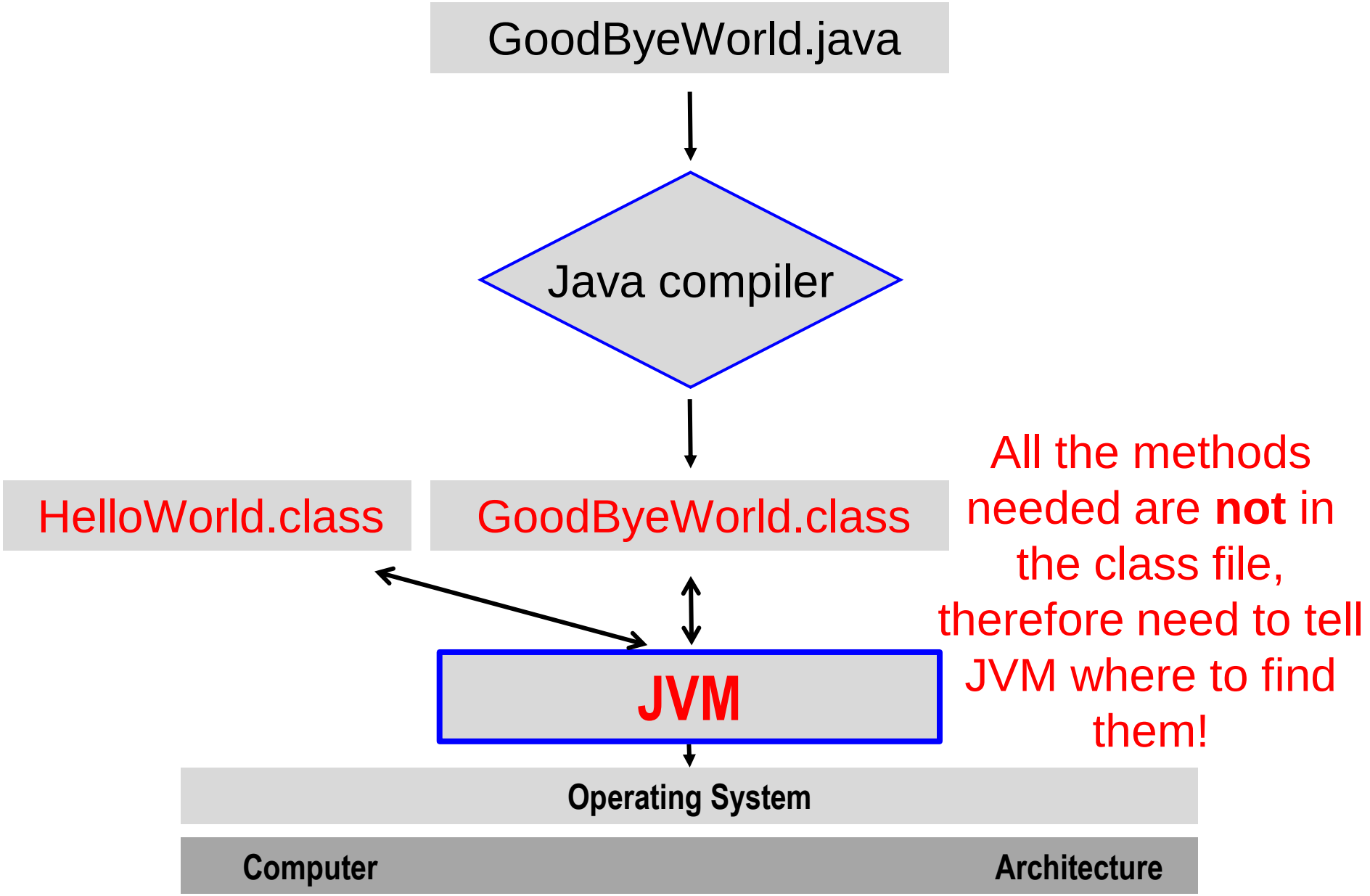
```
/*
 * CS611
 * ... Information as needed...
 */
public class goodByeWorld {
   public static void main( String[] args ) {
       // statement to output Hello World
       System.out.println( "Good Bye World" );
       method1(); // call method1 in goodByeWorld
       HelloWorld.method1();
   }

   public static void method1( ) {
       // body of method1
   }

}
```

Hello World in Java:

a program class can contain multiple methods

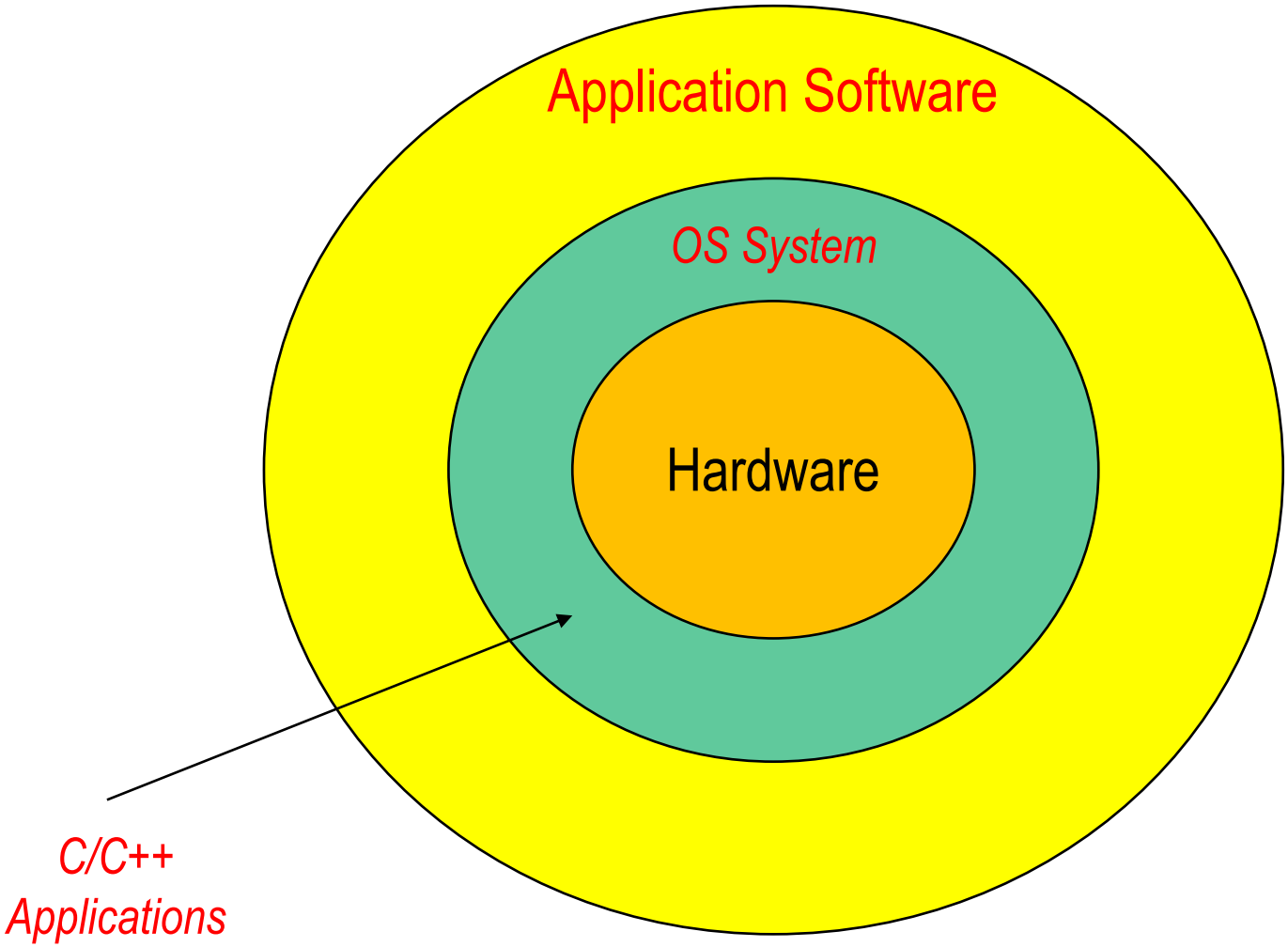
```
/*
 * CS611
 * ... Information as needed...
 */
public class goodByeWorld {
   public static void main( String[] args ) {
       // statement to output Hello World
        System.out.println( "Good Bye World" );
        method1(); // call method1 in goodByeWorld
  ——→  HelloWorld.method1();
    }

   public static void method1( ) {
        // body of method1
   }

}
```

Executing a Java Program

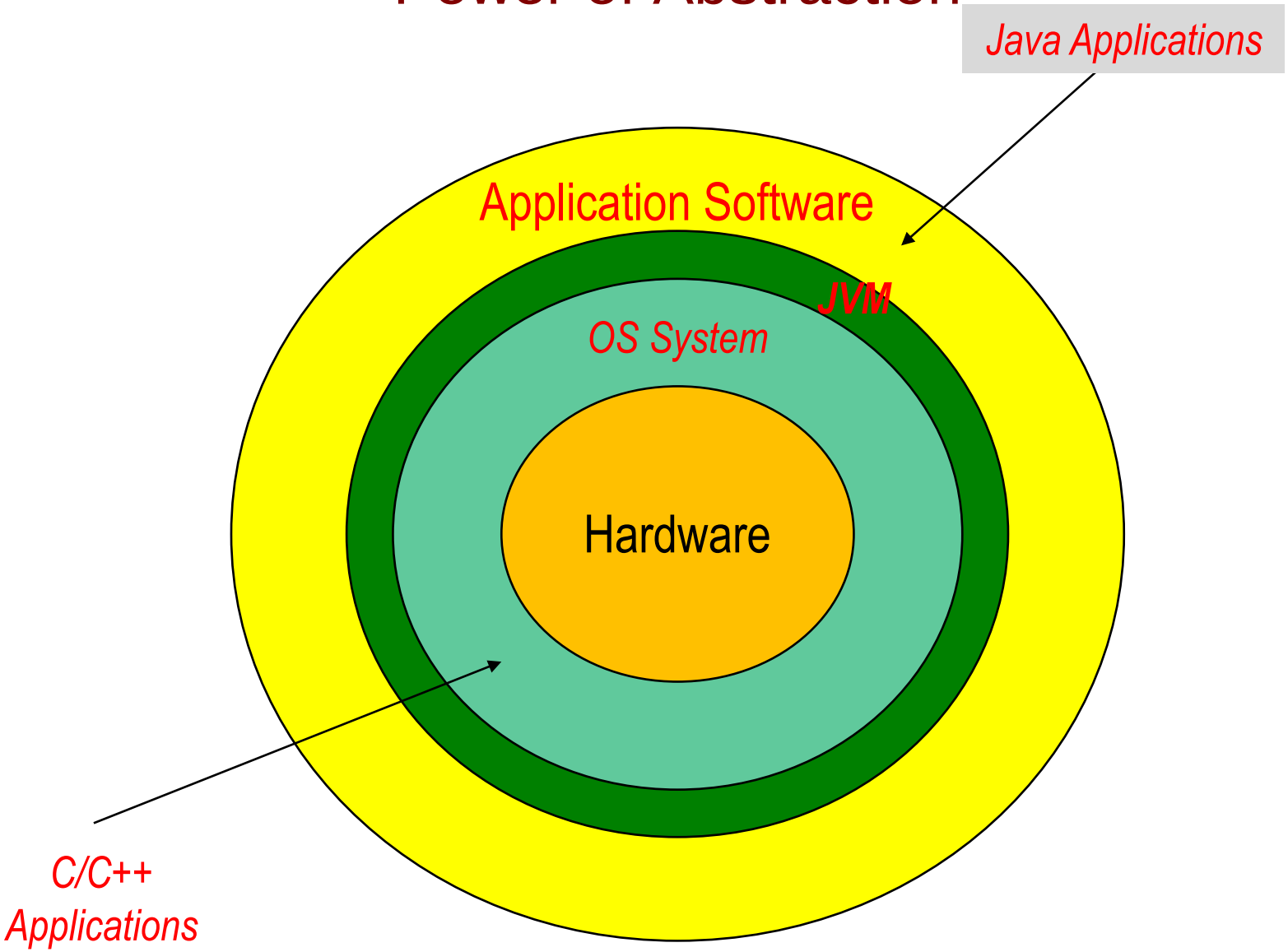


All the methods needed are **not** in the class file, therefore need to tell JVM where to find them!

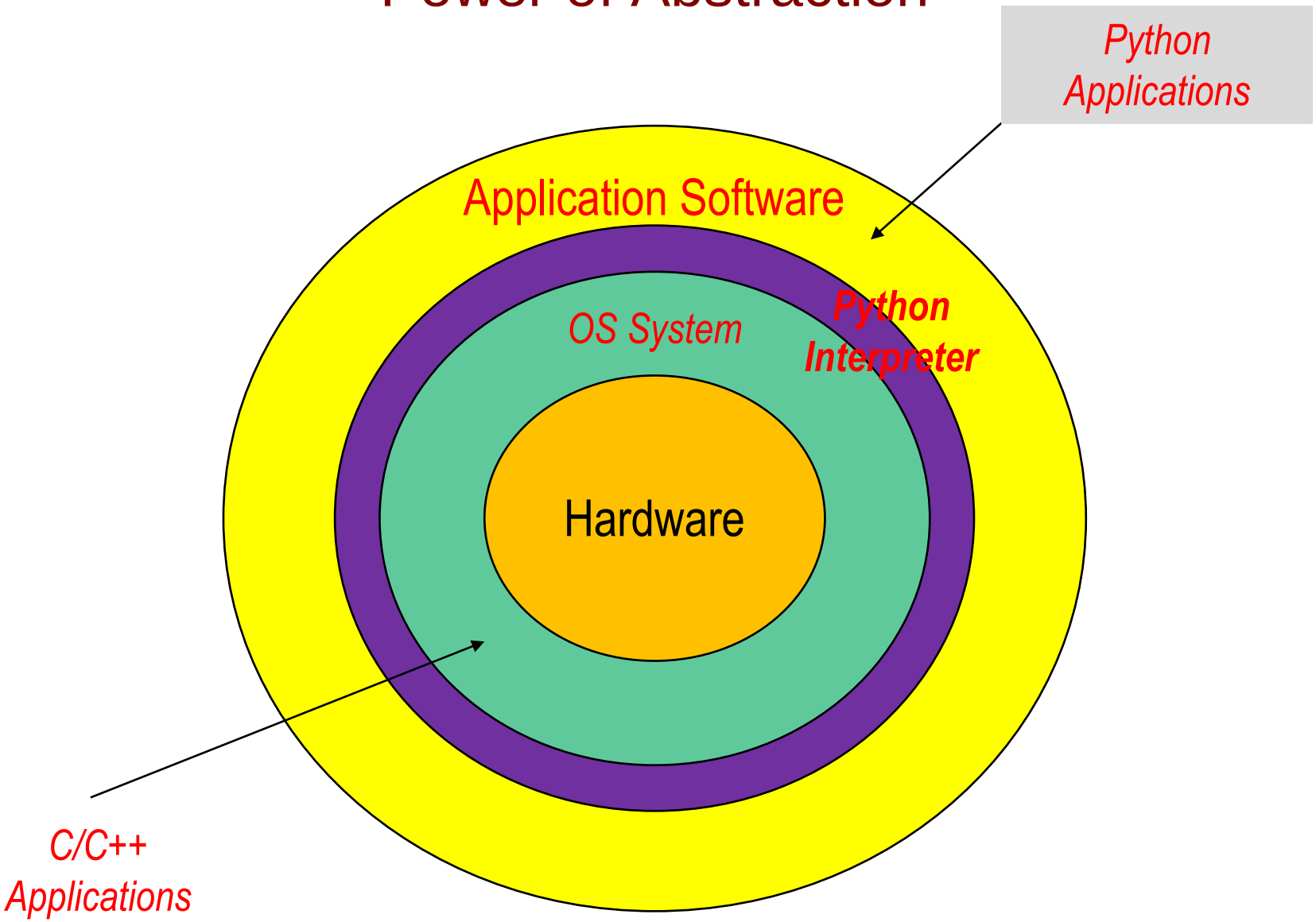
Power of Abstraction



Power of Abstraction



Power of Abstraction



To Type or Not to Type

Simple Program

JAVA

```java
public class Play {

   public static void main( String[] args ) {

      int num1 = 5;
      int num2 = 7;
      int num3 = 12;

      int sum = num1 + num2 + num3;
                  .
   }
}
```

Python

```python
num1 = 5
num2 = 7
num3 = 12

sum = num1 + num2 + num3
```

Declaring a Variable

- Strongly typed languages require that we specify the type of data that a variable will store before we attempt to use it.
- This is called *declaring* the variable.
  - syntax:

    type variable;
  - examples:

```
int count;
double area;
```

says that `count` will store an integer

says that `area` will store a floating-point number (one with a decimal)

type of the variable

name of the variable

- Optional: you can assign an initial value at the same time:

```
int count = 0, sum = 0, total;
final double PI = 3.14159; // constant variable in Java
```

Declaring a Variable

- Strongly typed languages require that we specify the type of data that a variable will store before we attempt to use it.
- This is called *declaring* the variable.
  - syntax:

    type variable;
  - examples:

```
int count;
double area;
```

says that `count` will store an integer

says that `area` will store a floating-point number (one with a decimal)

type of the variable

name of the variable

- Optional: you can assign an initial value at the same time:

```
int count = 0, sum = 0, total;
const double PI = 3.14159; // constant variable in C/C++
```

Commonly Used Data Types

- `int` - an integer

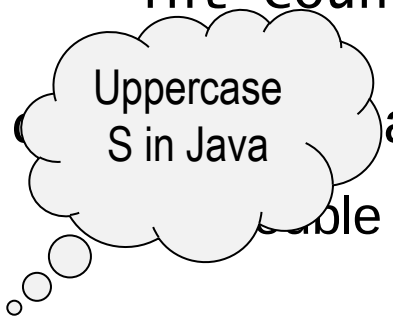
```
int count = 0;
```

- `double` - a floating-point number (one with a decimal)
  - double area = 125.5;

Commonly Used Data Types

- `int` - an integer

```
int count = 0;
```



- `d`ating-point number (one with a decimal)

```
ble area = 125.5;
```

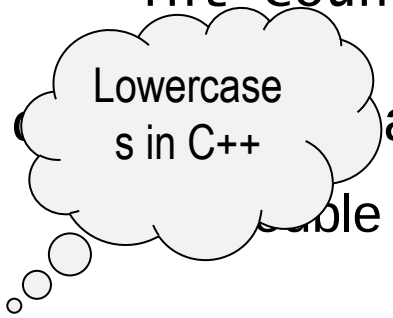
- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 112!";
```

Commonly Used Data Types

- `int` - an integer

```
int count = 0;
```



- C ating-point number (one with a decimal)

```
ble area = 125.5;
```

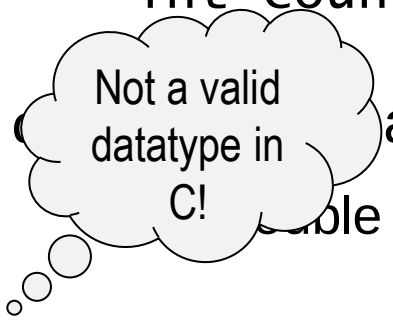
- `string` - a sequence of 0 or more characters

```
String message = "Welcome to CS 112!";
```

Commonly Used Data Types

- `int` - an integer

```
int count = 0;
```



- ating-point number (one with a decimal)

```
ble area = 125.5;
```

- `string` - a sequence of 0 or more characters

```
String message = "Welcome to CS 112!";
```

Commonly Used Data Types

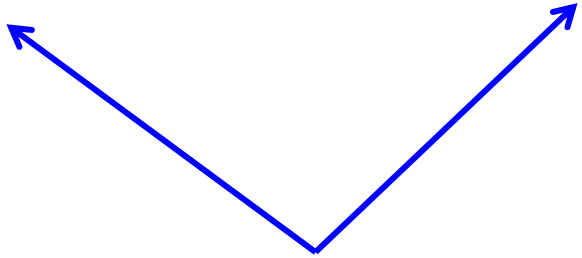
- `int` - an integer

```
int count = 0;
```

- `double` - a floating-point number (one with a decimal)
  - double area = 125.5;

- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 112!";
```



Commonly Used Data Types

- `int` - an integer

```
int count = 0;
```

- `double` - a floating-point number (one with a decimal)
  - double area = 125.5;

- `String` - a sequence of 0 or more characters

```
String message = "n";   // a string of one char
```

Commonly Used Data Types

- `int` - an integer

```
int count = 0;
```

- `double` - a floating-point number (one with a decimal)
  - double area = 125.5;

- `String` - a sequence of 0 or more characters

```
String message = "\n";  // a string of ? char
```

Commonly Used Data Types

- `int` - an integer

```
int count = 0;
```

- `double` - a floating-point number (one with a decimal)
  - double area = 125.5;

- `String` - a sequence of 0 or more characters

```
String message = "\n";  // newline character
```

Commonly Used Data Types

- `int` - an integer

```
int count = 0;
```

- `double` - a floating-point number (one with a decimal)
  - double area = 125.5;
- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 611!";
```

- `char` - a single character
  - char abc = 'Z', nline = '\n';
- `boolean` - either `true` or `false`
  - boolean isPrime = false;

Commonly Used Data Types

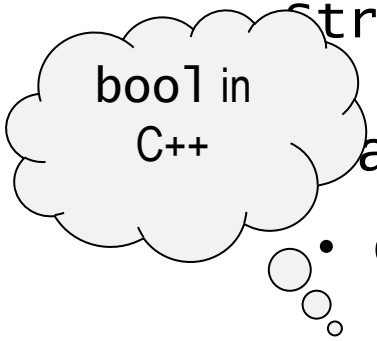
- `int` - an integer

```
int count = 0;
```

- `double` - a floating-point number (one with a decimal)
  - double area = 125.5;

- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 611!";
```



- a single character
  - char abc = 'Z', nline = '\n';

- `boolean` - either `true` or `false`
  - boolean isPrime = false;

Commonly Used Data Types

- `int` - an integer

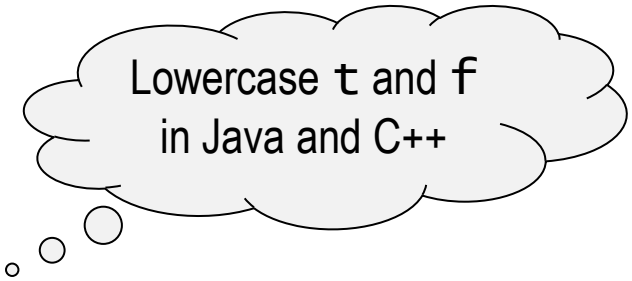
```
int count = 0;
```

- `double` - a floating-point number (one with a decimal)
  - double area = 125.5;

- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 611!";
```

- `char` - a single character
  - char abc = 'Z', nline = '\n';



- `boolean` - either `true` or `false`
  - boolean isPrime = false;

Commonly Used Data Types

- `int` - an integer

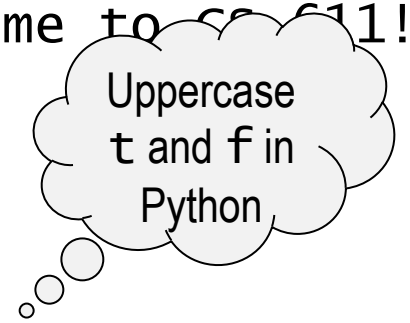
```
int count = 0;
```

- `double` - a floating-point number (one with a decimal)
  - double area = 125.5;

- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 611!";
```

- `char` - a single character
  - char abc = 'Z', nline = '\n';



- `boolean` - either `True` or `False`
  - boolean isPrime = false;

Commonly Used Data Types

- `int` - an integer

```
int count = 0;
```

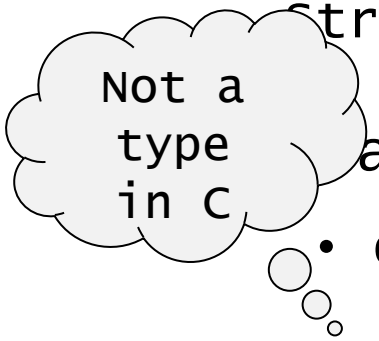
- `double` - a floating-point number (one with a decimal)
  - double area = 125.5;

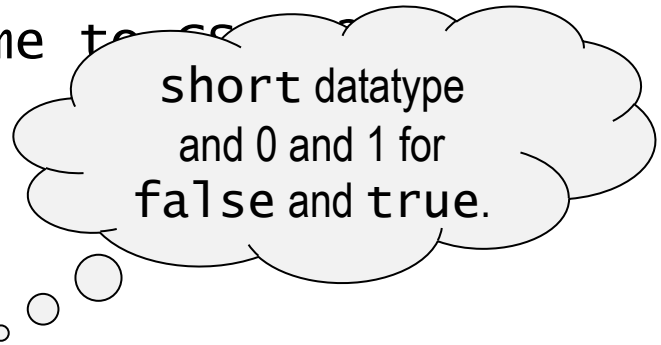
- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS
```

- a single character
  - char abc = 'Z', nline = '\n';

- `boolean` - either `True` or `False`
  - boolean isPrime = false;





Categorizing Data Types **in Java**

- `int` - an integer stored using 4 bytes

```
int count = 0;
```

- `long` - an integer stored using 8 bytes

```
long result = 1;
```

- `double` - a floating-point number (one with a decimal)
  - double area = 125.5;
- `boolean` - either `true` or `false`
  - boolean isPrime = false;

Primitive types

- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 611!";
```

- `Scanner` – an object for getting input from the user

```
Scanner scan = new Scanner(System.in);
```

Categorizing Data Types **in Java**

- `int` - an integer stored using 4 bytes

```
int count = 0;
```

- `long` - an integer stored using 8 bytes

```
long result = 1;
```

- `double` - a floating-point number (one with a decimal)
  - double area = 125.5;
- `boolean` - either `true` or `false`
  - boolean isPrime = false;
- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 611!";
```

- `Scanner` – an object for getting input from the user

```
Scanner scan = new Scanner(System.in);
```

Reference types

Weak vs. Strong Typed Languages

Python

```
x = 5

x = 1.967

x = "Christine"
```

Java

```
int x = 5;

x = 1.967;

x = "Christine";
```

Why?

Variable Declarations and Data Types

- Bytes of memory allocated for different types is architecture dependent but in general:

primitive type	**size**
`int`	`4 bytes`
`double`	`8 bytes`
`long`	`8 bytes`
`boolean`	`1 byte`

- Declaring a variable tells the compiler how much memory *(i.e. how many bytes*) to allocate **and** the *type* of the data!

- The (*binary representation of the data*) is stored in that memory cell.

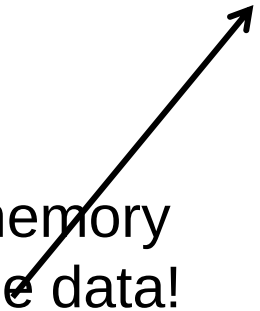
A note about `double` and `float`

- Bytes of memory allocated for different types is architecture dependent but in general:

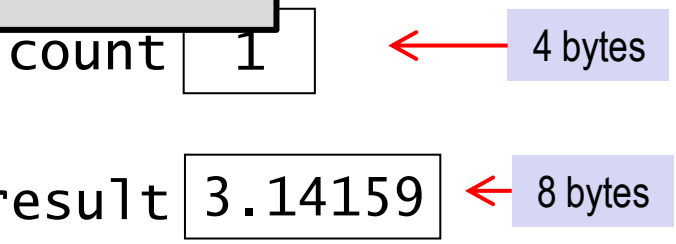
primitive type	size
`int`	4 bytes
`double`	8 bytes
`float`	4 bytes

The distinction is in the precision. Floats usually allow up to 7 decimal digits of precision, whereas doubles allow up to 15 decimal digits of precision.

```
double dval = 99.5;  // d is the default
float fval = 99.7f;  // f explicitly used
```



memory
he data!



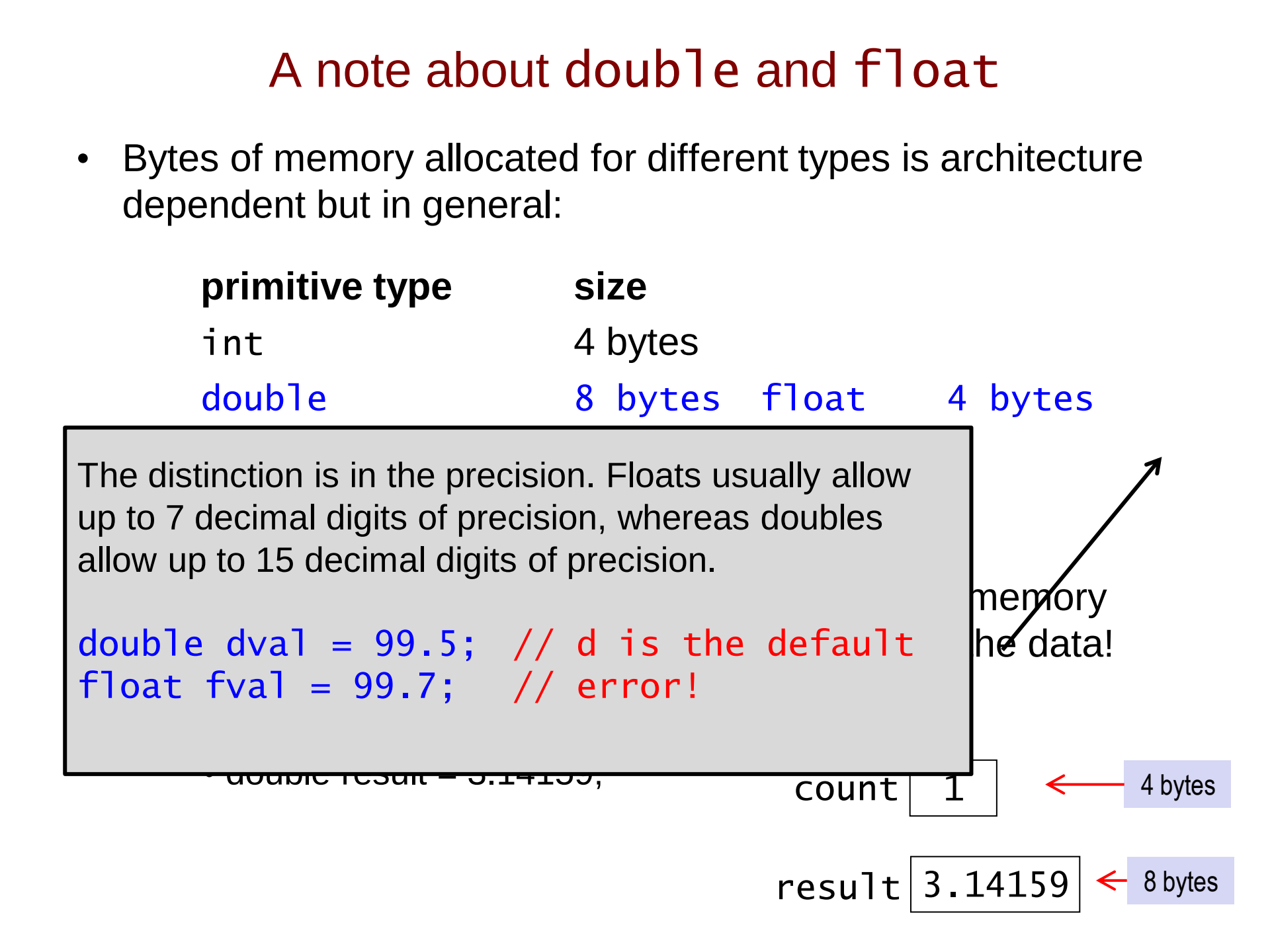
A note about `double` and `float`

- Bytes of memory allocated for different types is architecture dependent but in general:

primitive type	size
`int`	4 bytes
`double`	8 bytes
`float`	4 bytes

The distinction is in the precision. Floats usually allow up to 7 decimal digits of precision, whereas doubles allow up to 15 decimal digits of precision.

```
double dval = 99.5;  // d is the default
float fval = 99.7;   // error!
```



Java Data Types

- `int` - an integer stored using 4 bytes

```
int count = 0;
```

- `long` - an integer stored using 8 bytes

```
long result = 1;
```

- `double` - a floating-point number (one with a decimal)

```
double area = 125.5;
```

- `boolean` - either `true` or `false`

```
boolean isPrime = false;
```

- `String` - a sequence of 0 or more characters

```
String message = "Welcome to CS 112!";
```

- `Scanner` – an object for getting input from the user

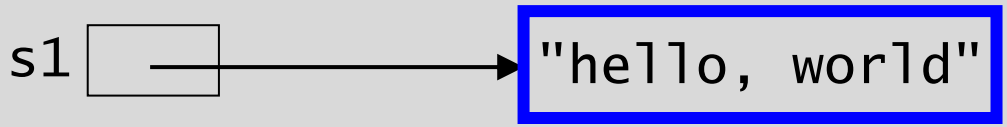
```
Scanner scan = new Scanner(System.in);
```

Reference types

Reference Types

- Variables of reference types *reference* objects!

```
String s1 = "hello, world";
```



  - the object is located elsewhere in memory
  - the variable stores a reference to the object

- Data types that work this way are known as *reference types.*
  - variables of those types are *reference variables*

- Example of two Java *reference types*:
  - `String`
  - `Scanner`

Weak vs. Strong Typed Languages

Python

```
x = 5

x = 1.967

x = "Christine"
```

Java

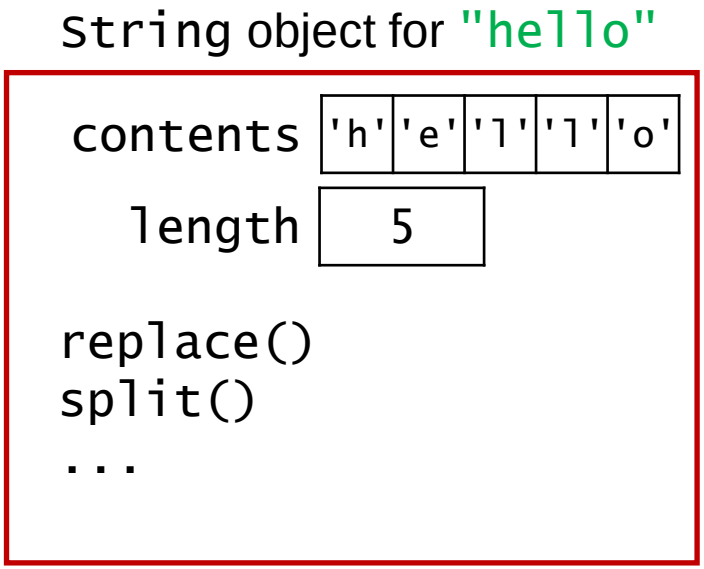
```
int x = 5;

x = 1.967;

x = "Christine";
```

Everything is an object!

Object vs. Primitive:
summary

- An object is a construct that groups together:
  - one or more data values
    (the object's *attributes* or *fields*)
  - one or more functions
    (known as the object's *methods*)

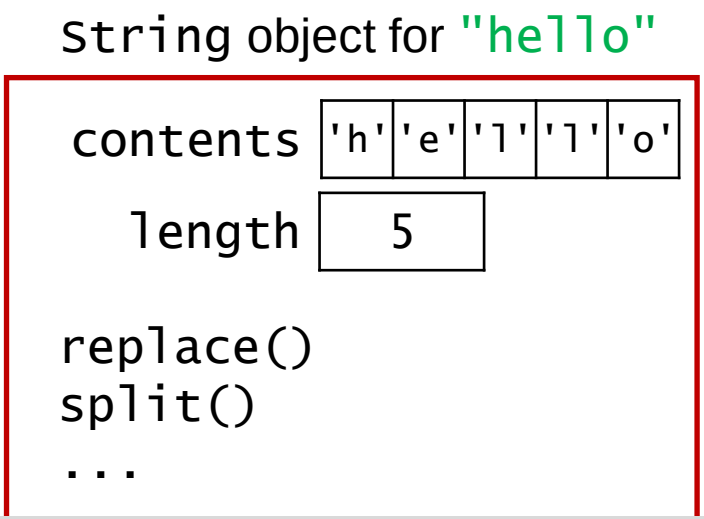
- Every object is an *instance*
  of a class.



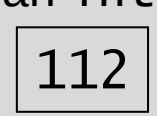
Object vs. Primitive:
summary

- An object is a *physical* construct that groups together:
  - one or more data values
    (the object's *attributes* or *fields*)
  - one or more functions
    (known as the object's *methods*)

- Every object is an *instance*
  of a class.



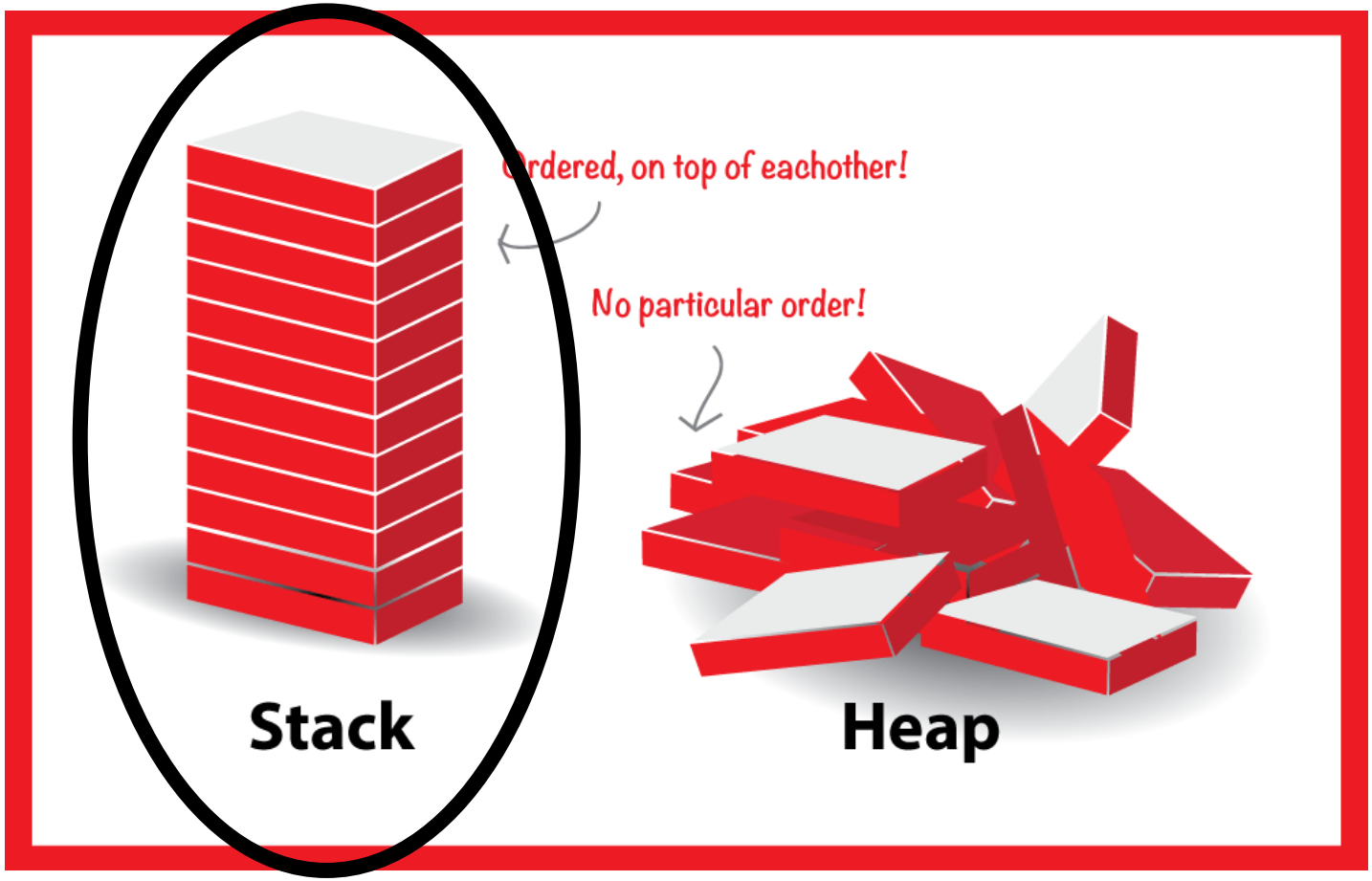
- Primitive values are *not* objects.
  - they are just "single" values
  - there is nothing else grouped with the value
  - they are not instances of a class
  - they require a fixed number of bytes based on their type
    - their value is stored in the allocated memory cell!



Memory Management: Looking Under the Hood

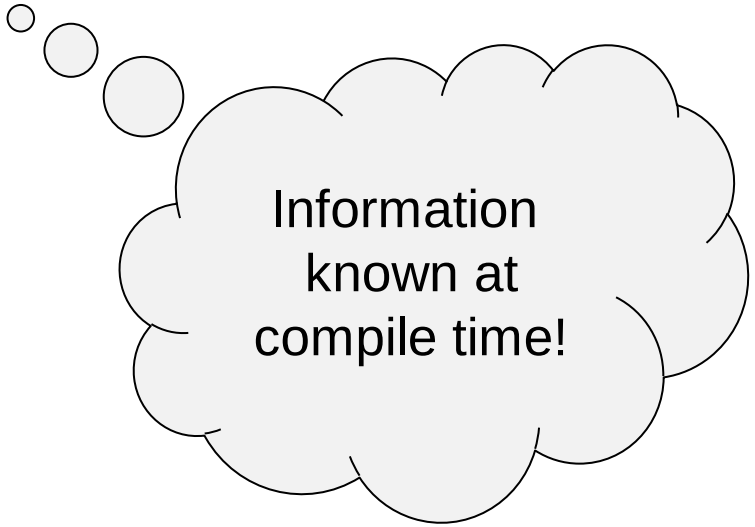
- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory:
  - Static
  - Stack
  - Heap

The Memory Stack



Memory Management: Looking Under the Hood

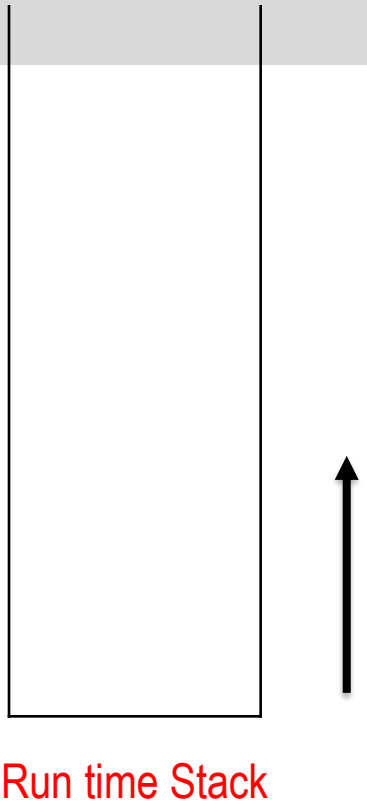
- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory:
  - Static — class variables
  - **Stack** — **local variables, parameters**
  - Heap — objects



Memory Management: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.
- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```

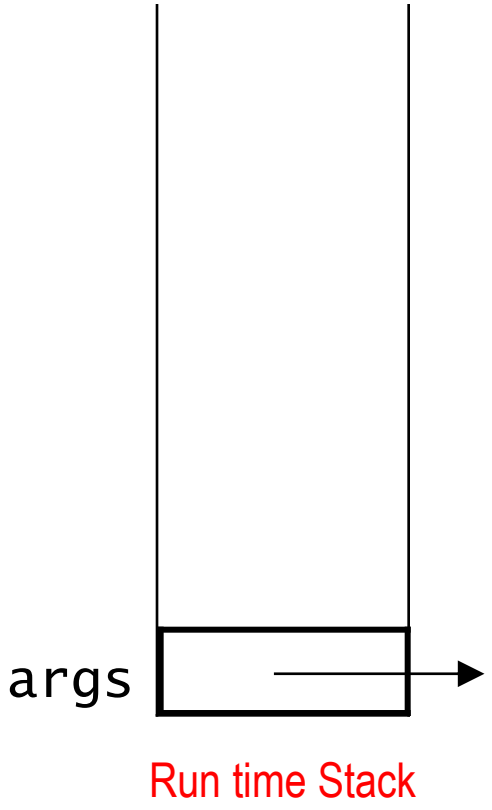


Memory Management: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```

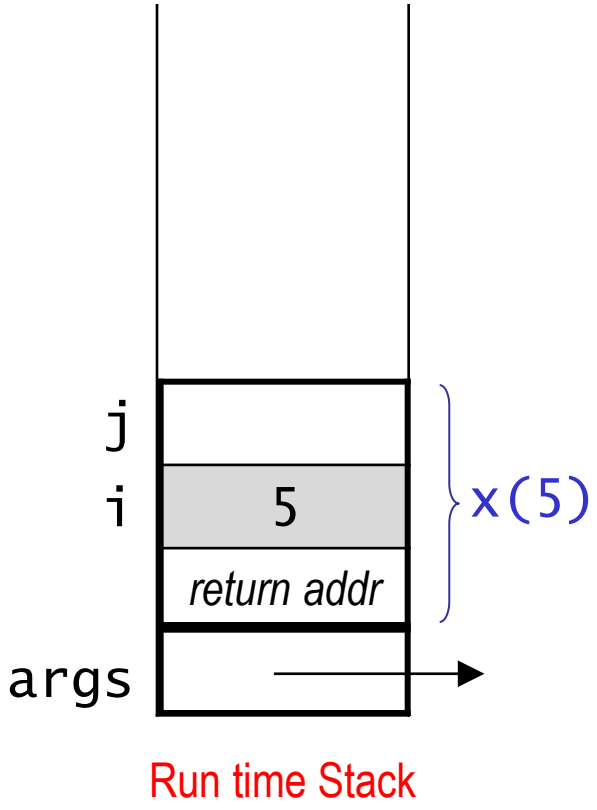


Memory Management: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```

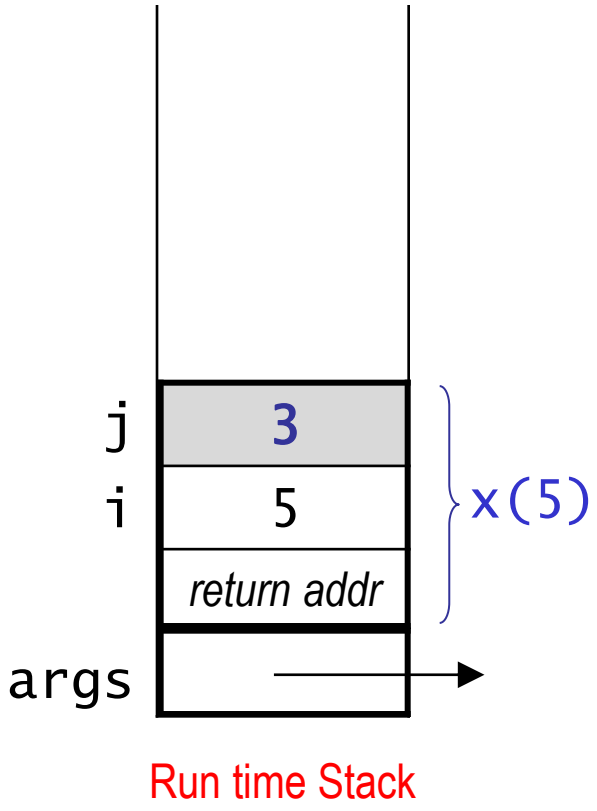


Memory Management: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```

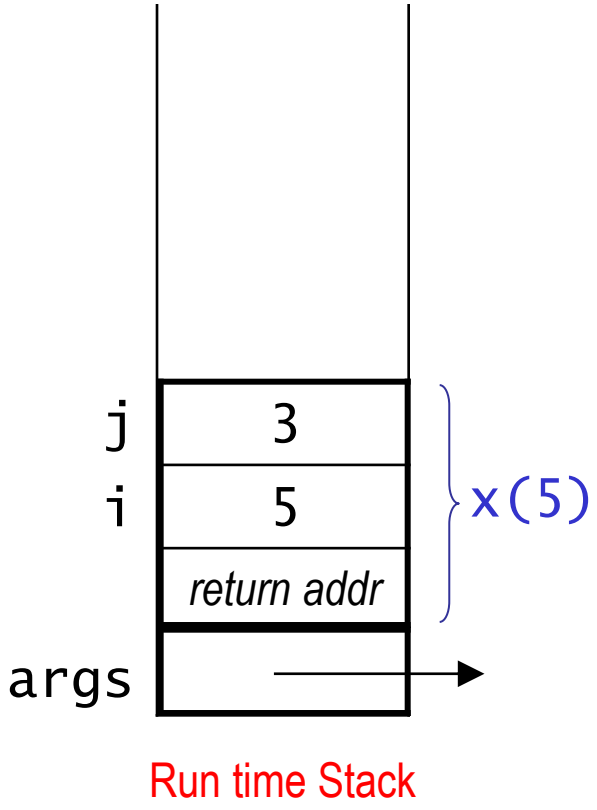


Memory Management: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```

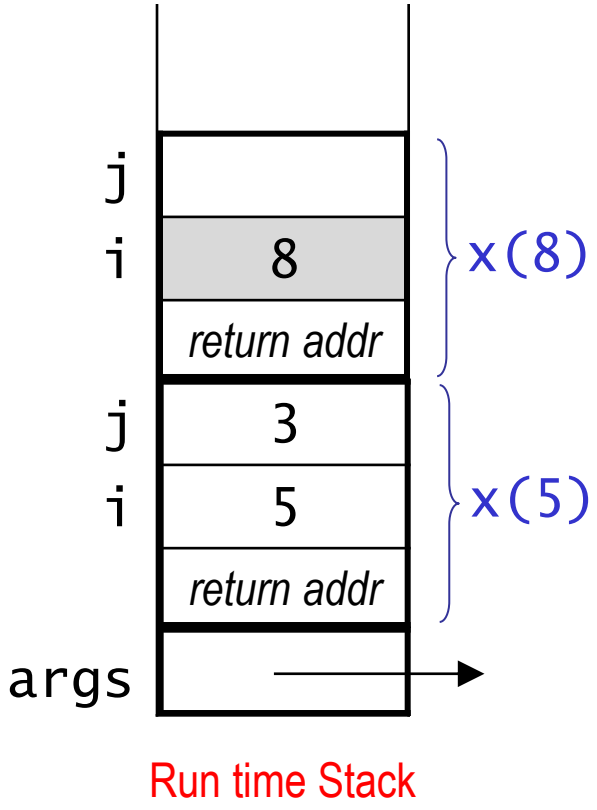


Memory Management: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```

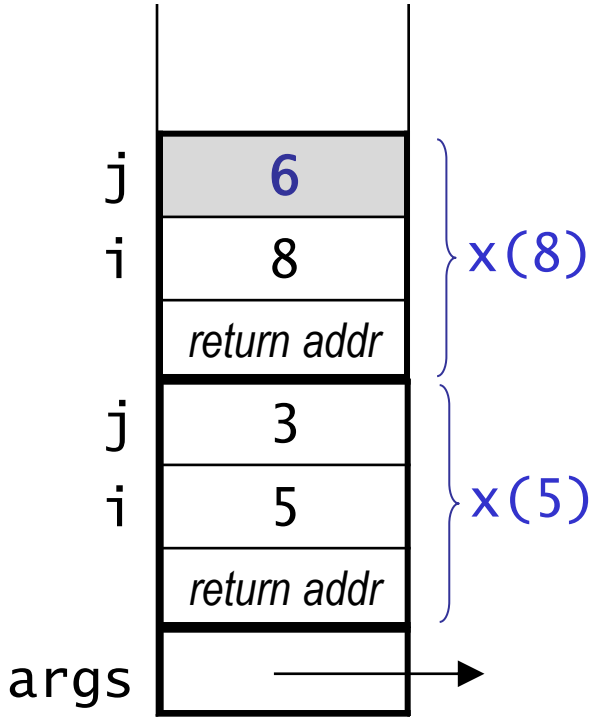


Memory Management: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```

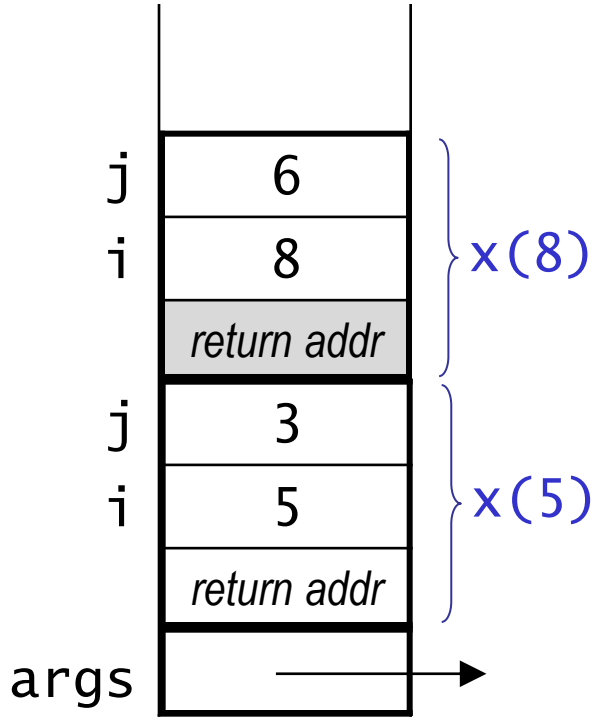


Memory Management: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



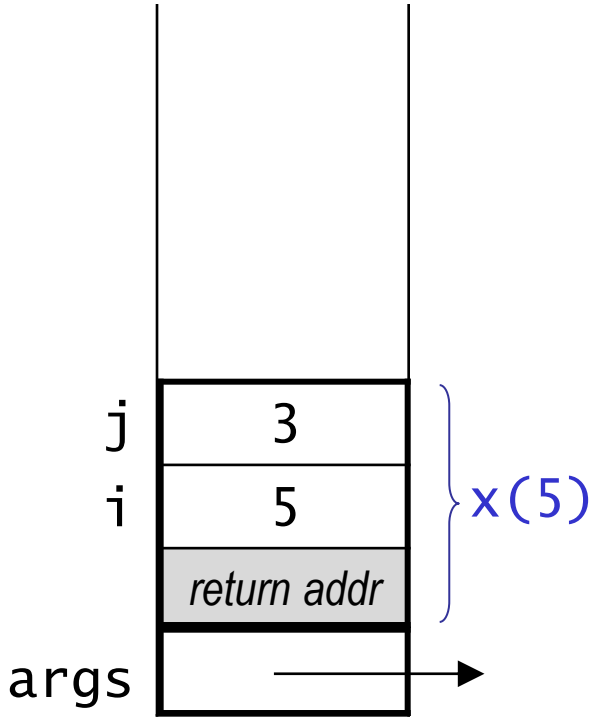
- When a method completes, its stack frame is removed.

Memory Management: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



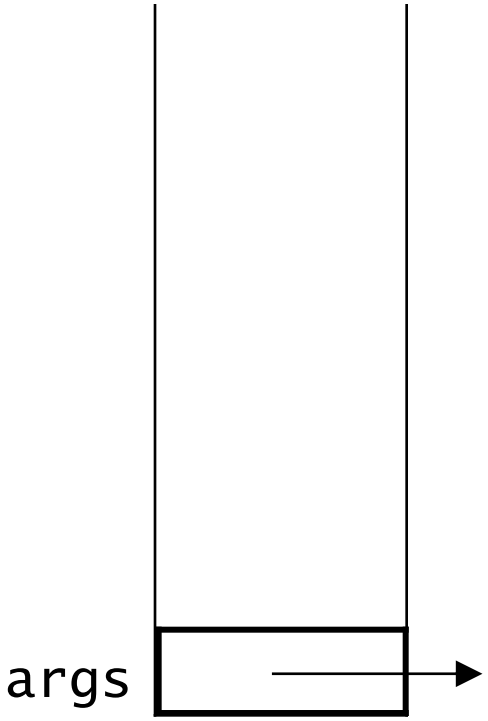
- When a method completes, its stack frame is removed.

Memory Management: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



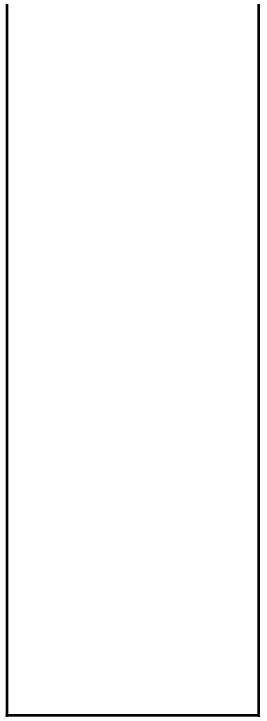
- When a method completes, its stack frame is removed.

Memory Management: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack.

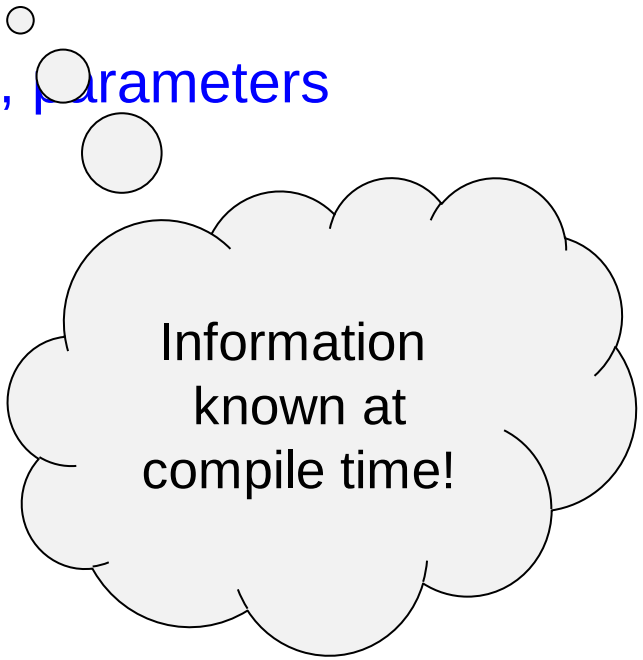
```
public class Foo {
  public static void x(int i) {
      int j = i - 2;

      if (i < 6)
          x(i + j);
  }

  public static void
    main(String[] args) {
      x(5);
  }
}
```



- When a method completes, its stack frame is removed.

Memory Management: Looking Under the Hood

- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory:
  - **Static** **class variables**
  - Stack local variables, parameters
  - Heap objects



Memory Management: Static Storage

- Static storage is used in Java for *class variables*, which are declared using the keyword `static`:

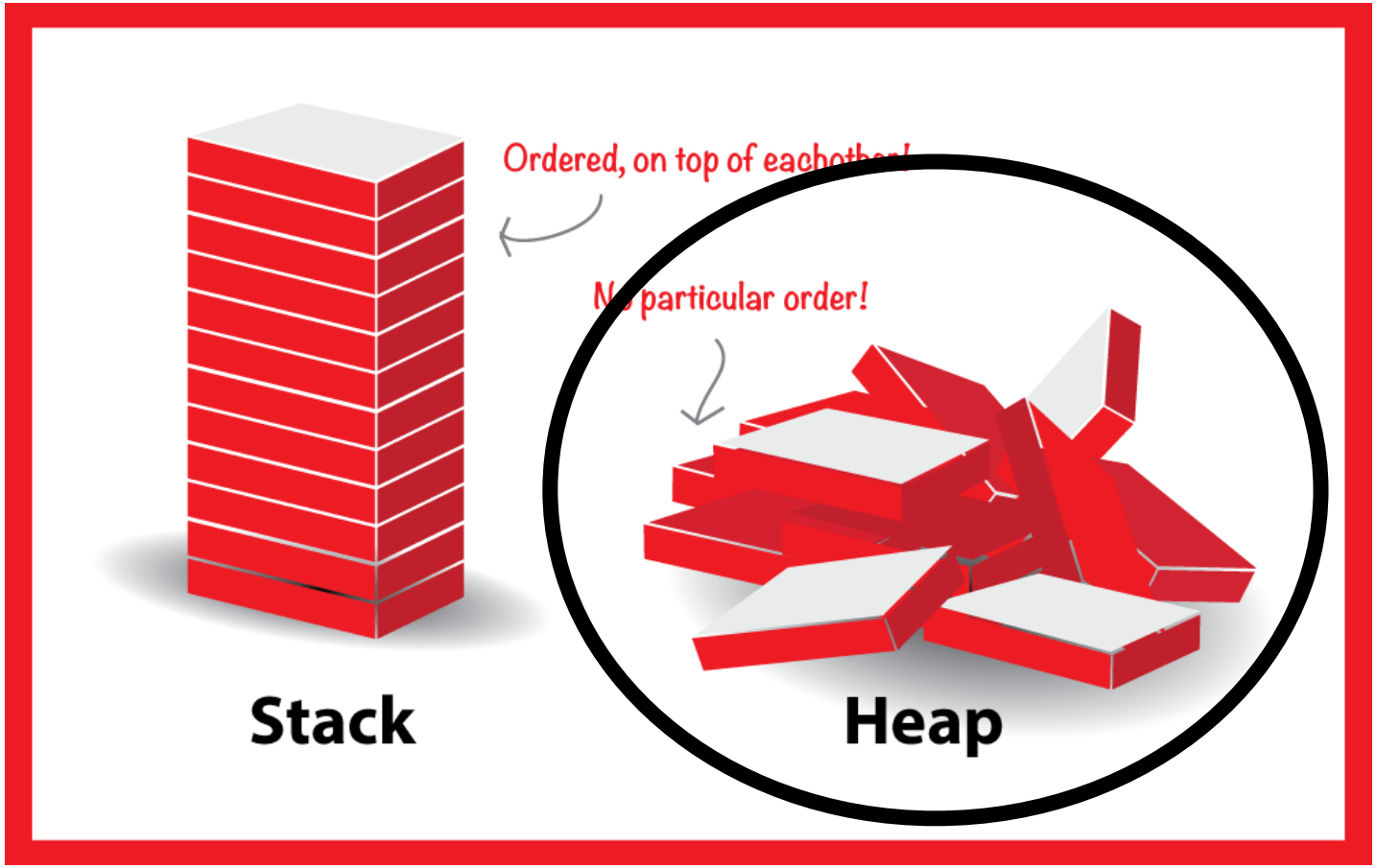
```
public static final double PI = 3.1495;
public static int numCompares;
```

- There is only one copy of each class variable; it is shared by all instances (i.e., all objects) and all methods of the class.

- The Java runtime system allocates memory for *class variables* when the class is first encountered.
  - this memory stays fixed for the duration of the program

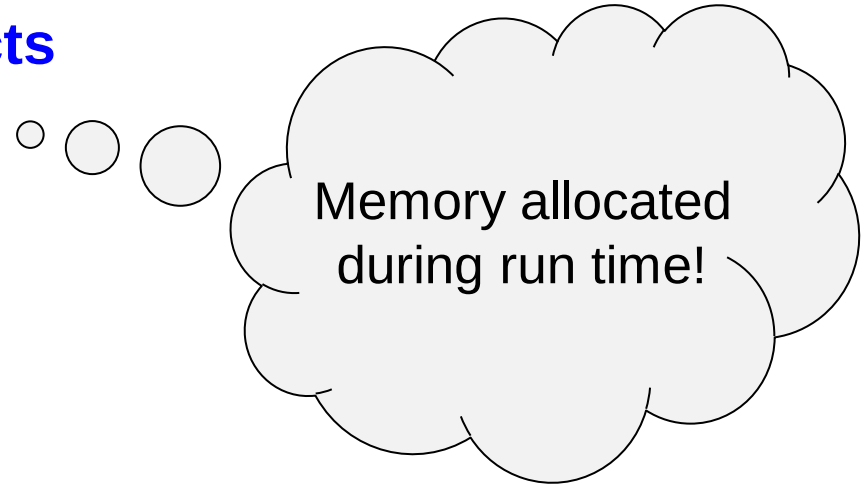
- Keyword *final* makes the variable *read-only*. Once a variable declared as final is assigned a value, it cannot be re-assigned.

The Memory Heap



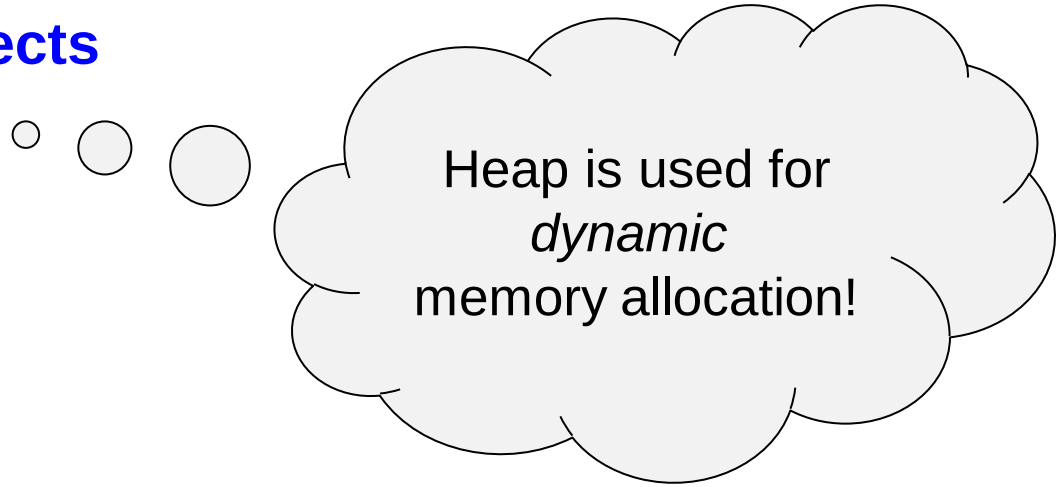
Memory Management: Looking Under the Hood

- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory:
  - Static — class variables
  - Stack — local variables, parameters
  - Heap — **objects**



Memory Management: Looking Under the Hood

- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory:
  - Static — class variables
  - Stack — local variables, parameters
  - Heap — **objects**



The Heap

- Objects are stored in a memory region known as *the heap*.
- Memory on the heap is allocated using the `new` operator:

```
String str = new String( “Hello World” );
Scanner inp = new Scanner();
```

- **`new`** returns the memory address of the start of the object on the heap.
  - a reference!
- An object persists until there are no remaining references to it.
- Unused objects are automatically reclaimed by a process known as *garbage collection*.
  - makes their memory available for other objects

Memory Management: Heap Storage

- Example:

Example: creating a Scanner object

```
Scanner scan = new Scanner(System.in);
```

Stack

Heap

Memory Management: Heap Storage

- Example:

Example: creating a Scanner object

```
Scanner scan = new Scanner(System.in);
```

Memory Management: Heap Storage

- Example:

Example: creating a Scanner object

```
Scanner scan = new Scanner(System.in);
```

Memory Management: Heap Storage

- Example:

Example: creating a String object

```
String str = new String("CS611");
```

Memory Management: Heap Storage

- Example:

Example: creating a String object

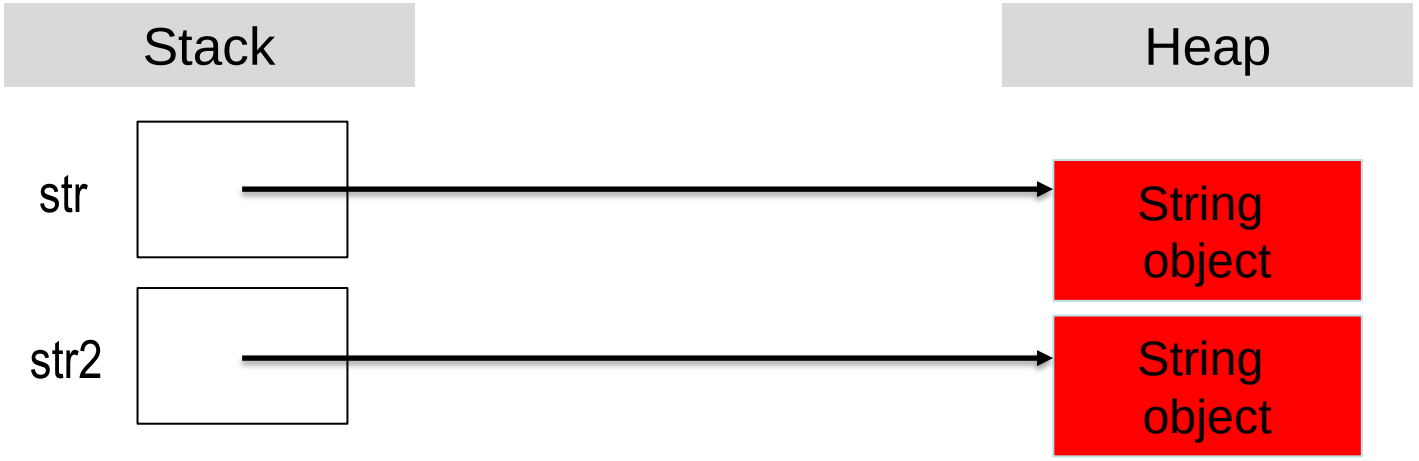
```
String str = new String("CS611");
```

Memory Management: Heap Storage

- Example:

Example: creating a String object

```
String str = new String("CS611");
String str = new String("CS611");
```



Memory Management: Heap Storage

- Example:

Example: creating a "**literal**" String object

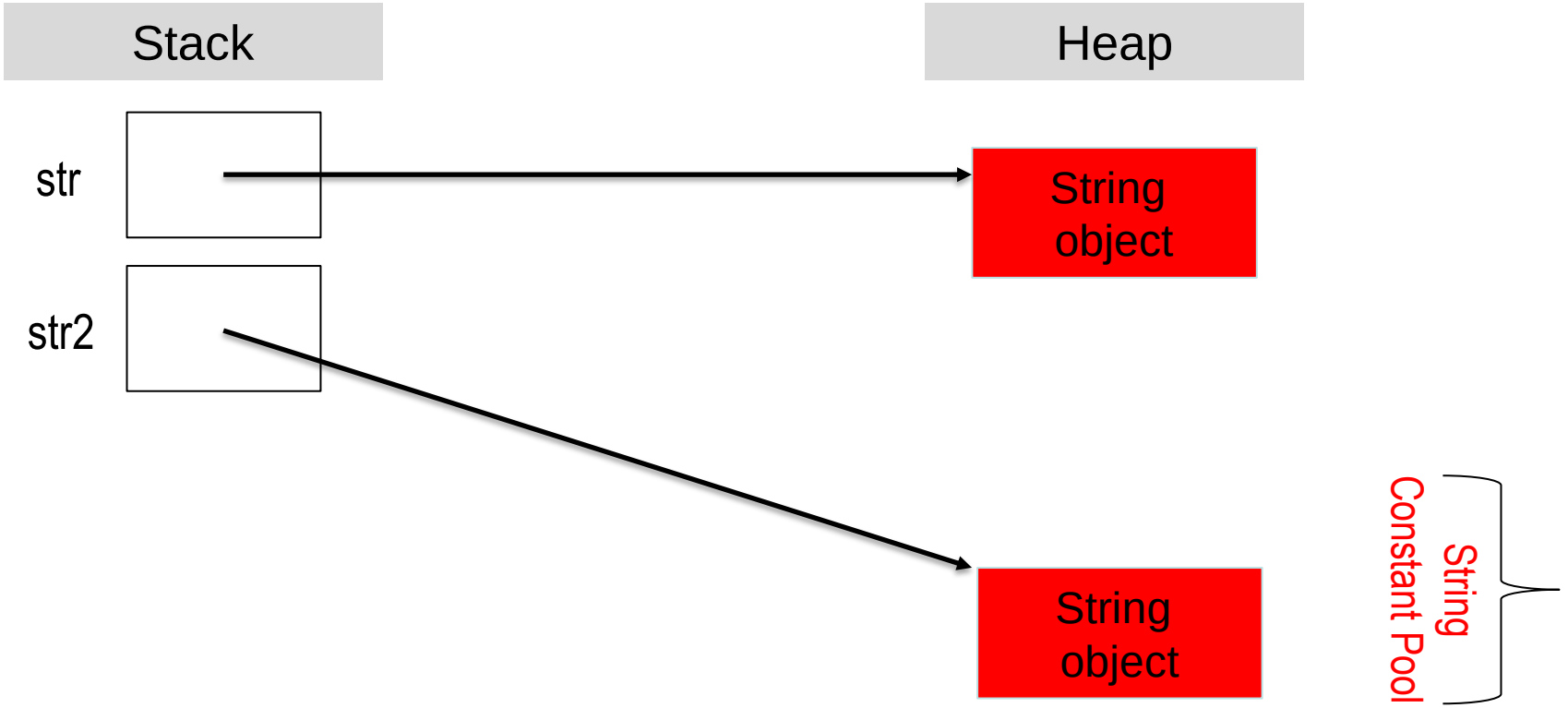
```
String str = new String("CS611");
String str2 = "CS611";
```

Memory Management: Heap Storage

- Example:

Example: creating a String object

```
String str = new String("CS611");
String str2 = "CS611";
```

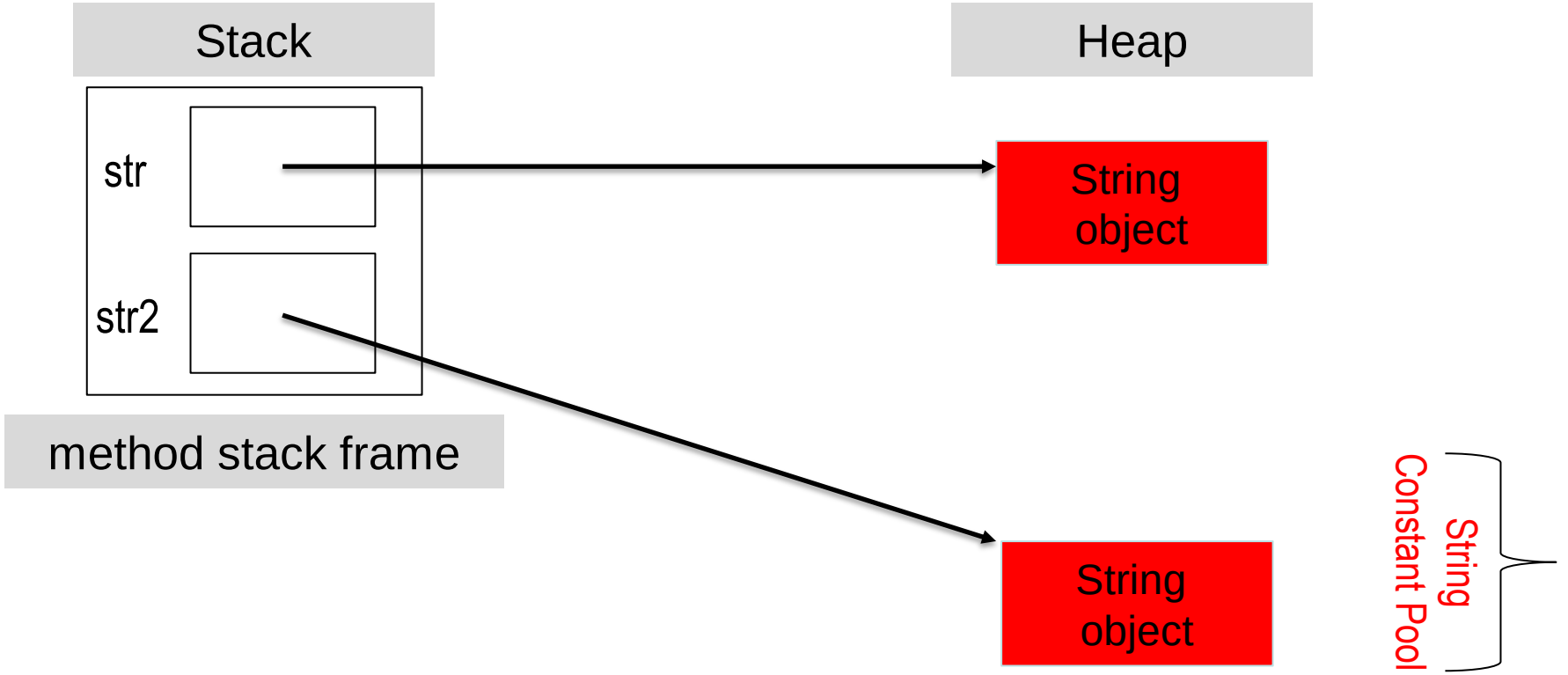


Memory Management: Heap Storage

- Example:

Example: creating a String object

```
String str = new String("CS611");
String str2 = "CS611";
```



Primitive vs. Reference types:

another look

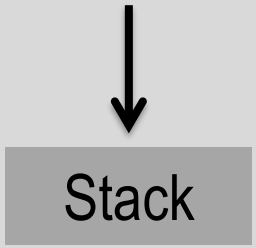
- Static variables are stored in *Static memory…*
- Objects are stored on *the Heap*…
- Primitive variables are stored on *the Stack*…

Primitive vs. Reference types:

another look

- Static variables are stored in *Static memory…*
- Objects are stored on *the Heap*…
- Primitive variables are stored on *the Stack*…

```
int p_var = 5;          // primitive variable
```



Primitive vs. Reference types:

another look

- Static variables are stored in *Static memory…*
- Objects are stored on *the Heap*…
- Primitive variables are stored on *the Stack*…

```
int p_var = 5;       // primitive variable

// Java Wrapper Classes for primitive types
```

Primitive vs. Reference types:

another look

- Static variables are stored in *Static memory…*
- Objects are stored on *the Heap*…
- Primitive variables are stored on *the Stack*…

```
int p_var = 5;          // primitive variable

Integer i_ref = new Integer(5);
```

Primitive vs. Reference types:

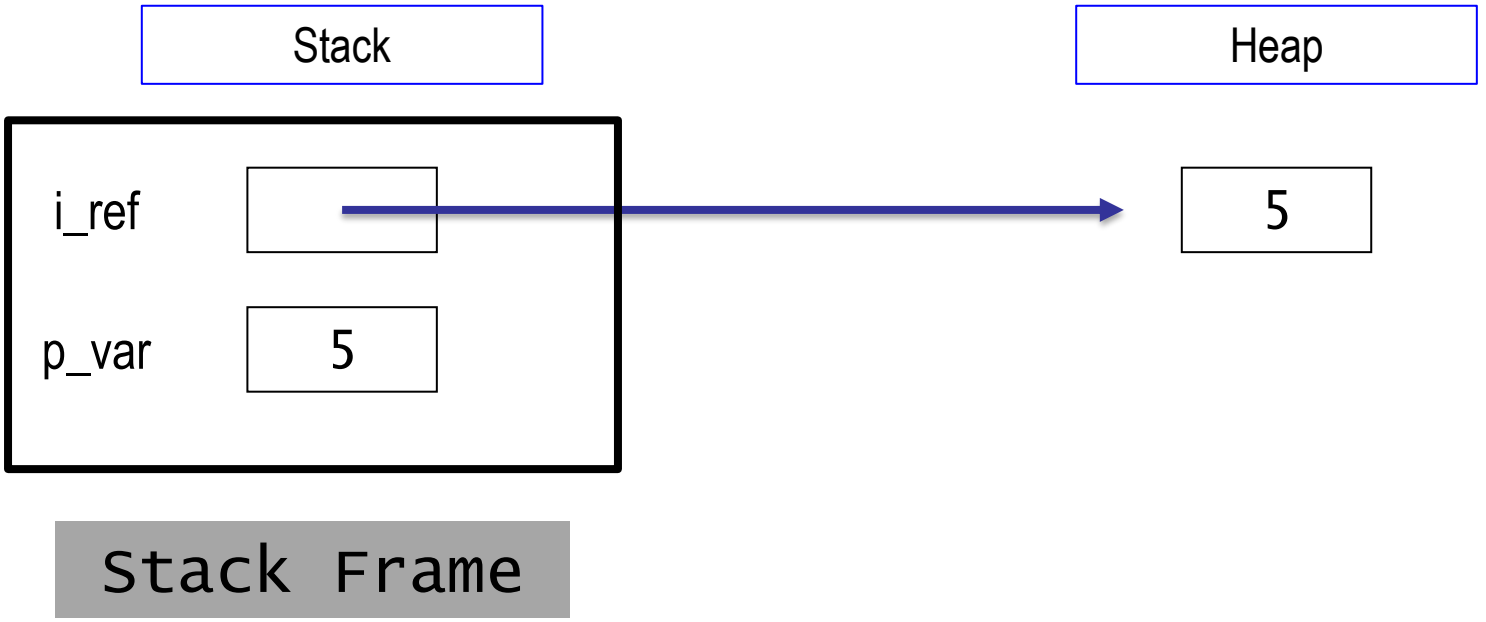
another look

- Static variables are stored in *Static memory…*
- Objects are stored on *the Heap*…
- Primitive variables are stored on *the Stack*…

```
int p_var = 5;       // primitive variable

Integer i_ref = new Integer(5);

Character c_ref = new Character( 'c' );
Double d_ref = new Double(5.555555);
Float f_ref = new Float(5.5);
Boolean b_ref = new Boolean( true );
```

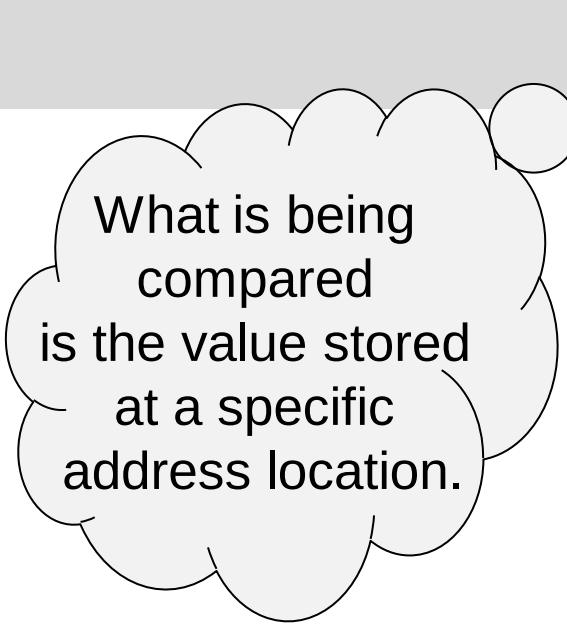
int Primitive vs. Integer Object

```
Integer i_ref = new Integer(5);   // an integer object
int p_var = 5;                    // primitive variable
```



Testing for Equivalent *Primitive* Values

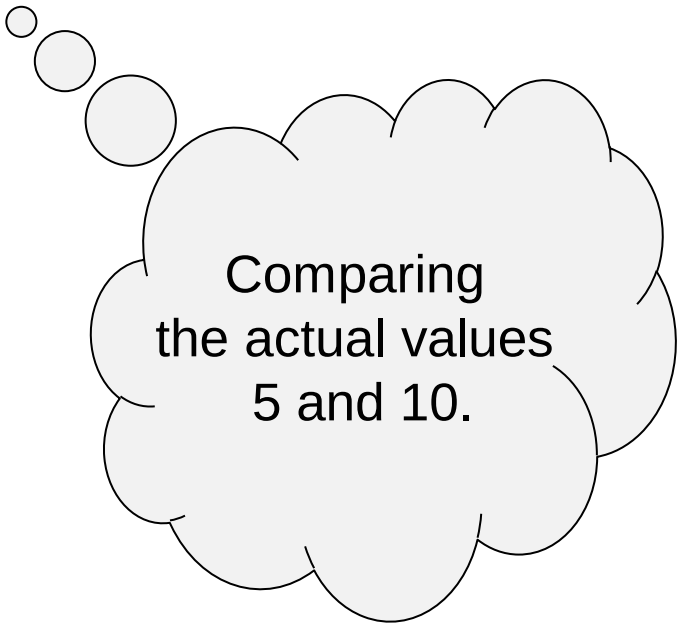
- The `==` and `!=` operators are used to compare primitives.
  - `int`, `double`, `char`, etc.

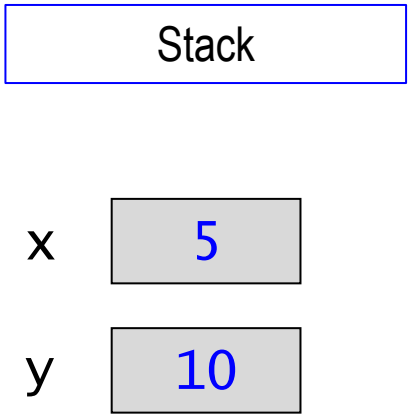


Testing for Equivalent *Primitive* Values

- The `==` and `!=` operators are used to compare primitives.
  - `int`, `double`, `char`, etc.

```
int x = 5;
int y = 10;
if ( x == y ) {


}
```

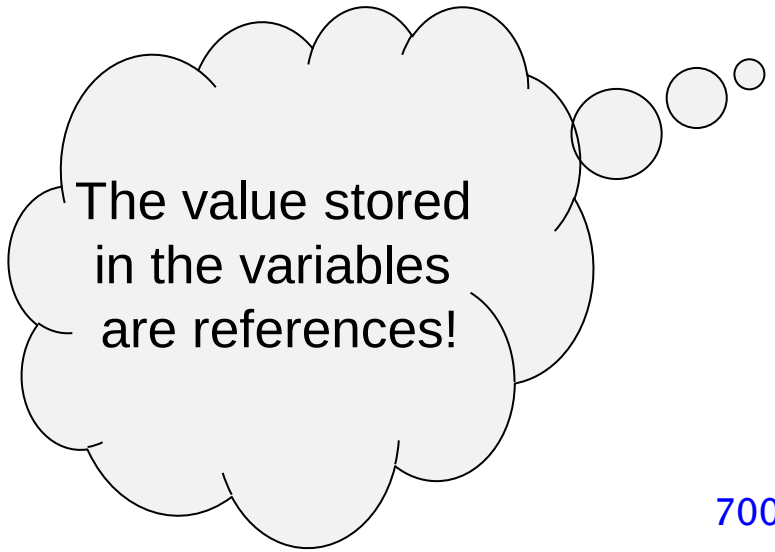


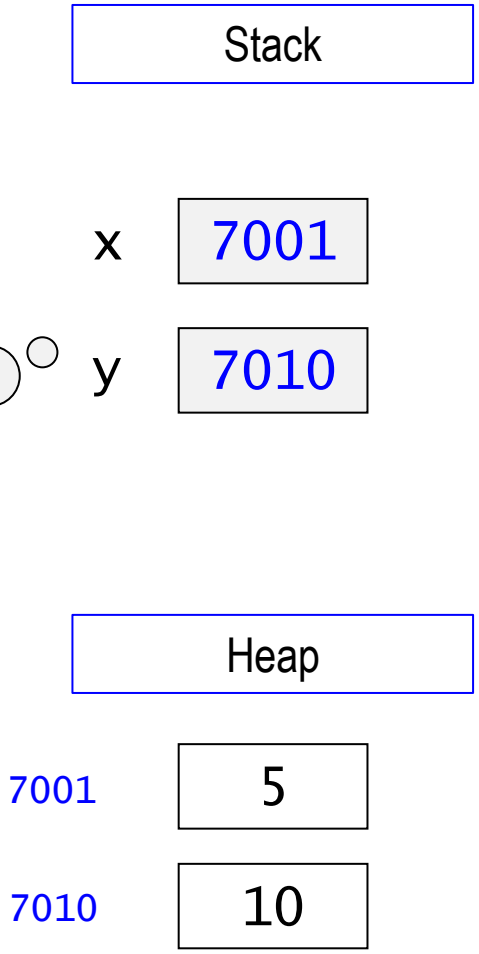


Testing for Equivalent *Objects:*
Numeric Wrapper Classes

- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
Integer x = new Integer(5);
Integer y = new Integer(10);
if ( x == y ) {


}
```



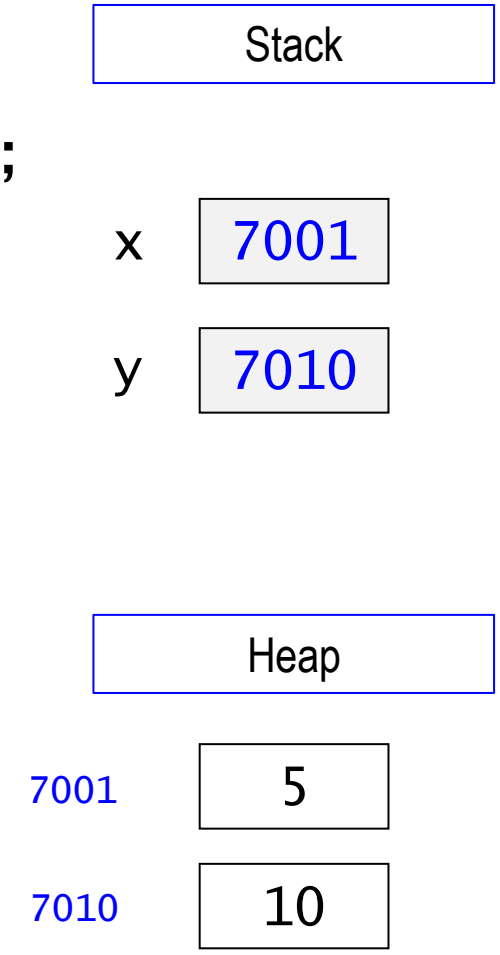


Testing for Equivalent *Objects:*
Numeric Wrapper Classes

- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
Integer x = new Integer(5);
Integer y = new Integer(10);
if ( x == y ) {


}
```

Comparing the address locations of the Integer objects!

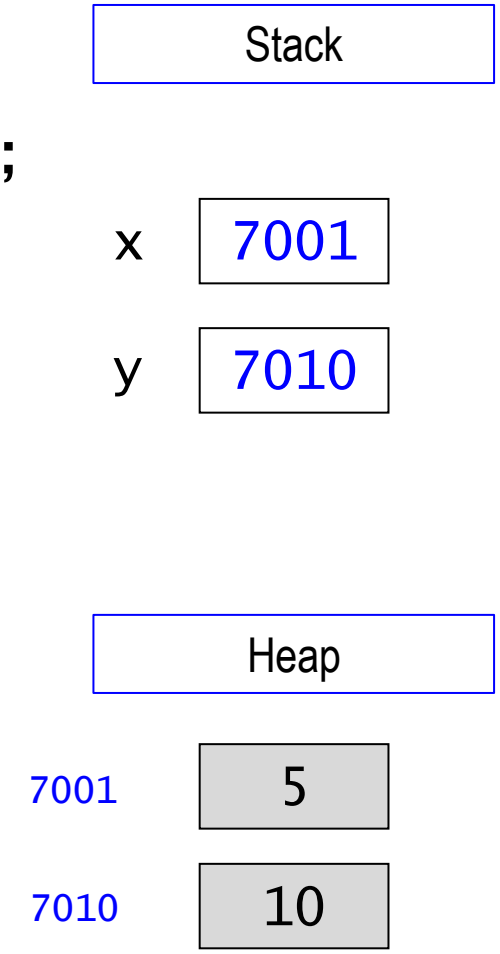


Testing for Equivalent *Objects:*
Numeric Wrapper Classes

- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
Integer x = new Integer(5);
Integer y = new Integer(10);
if ( ? ) {


}
```

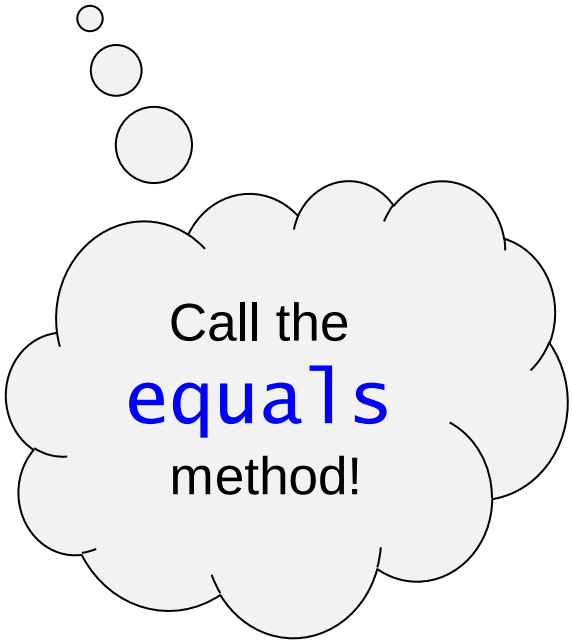
How would we compare the objects being referenced?

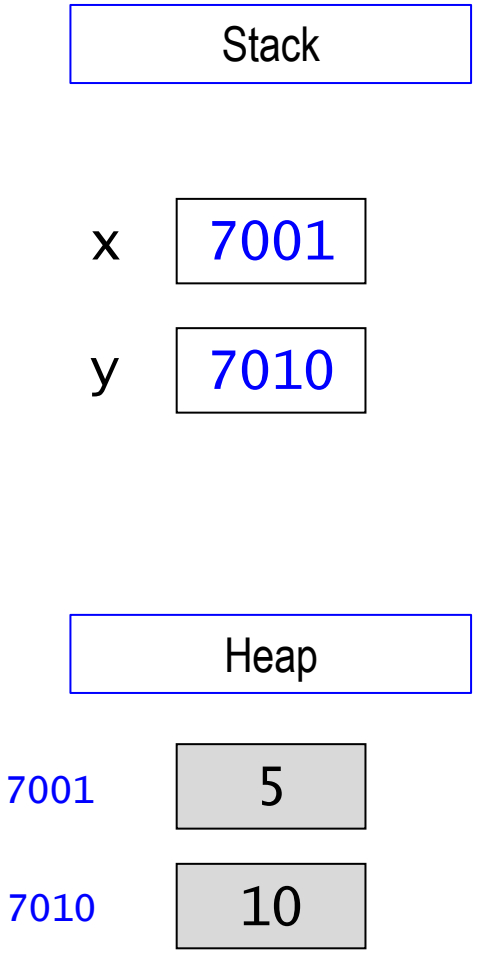


Testing for Equivalent *Objects:*
Numeric Wrapper Classes

- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
Integer x = new Integer(5);
Integer y = new Integer(10);
if ( x.equals(y) ) {


}
```

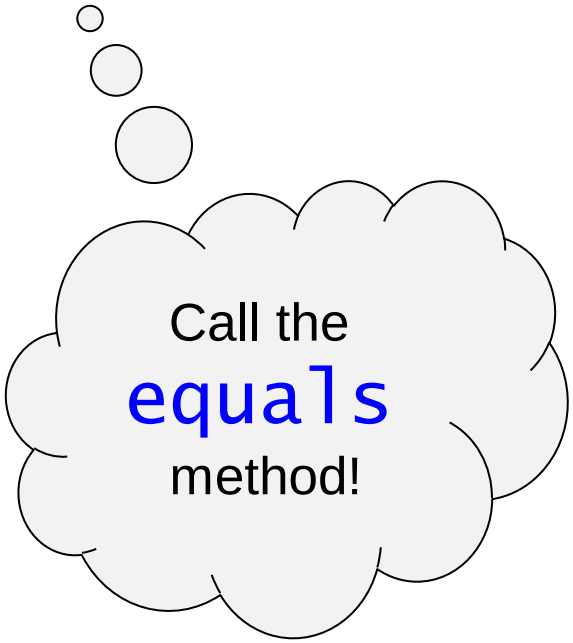


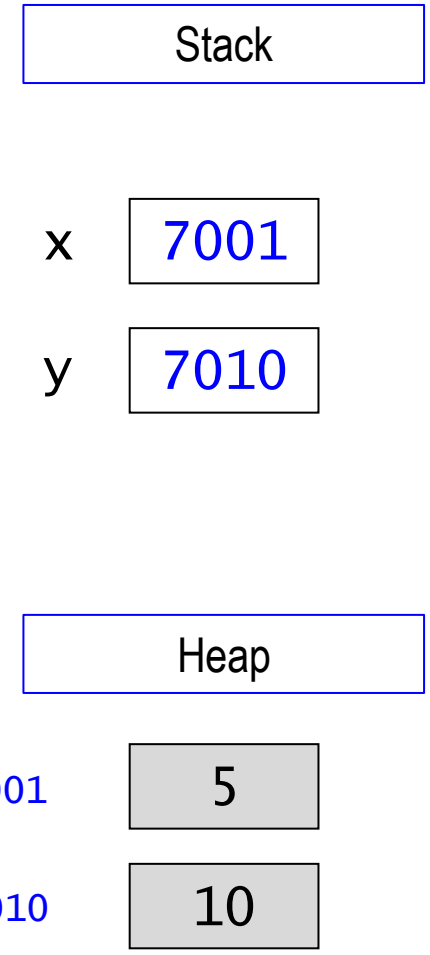


Testing for Equivalent *Objects:*
Numeric Wrapper Classes

- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
Integer x = new Integer(5);
Integer y = new Integer(10);
if ( y.equals(x) ) {


}
```

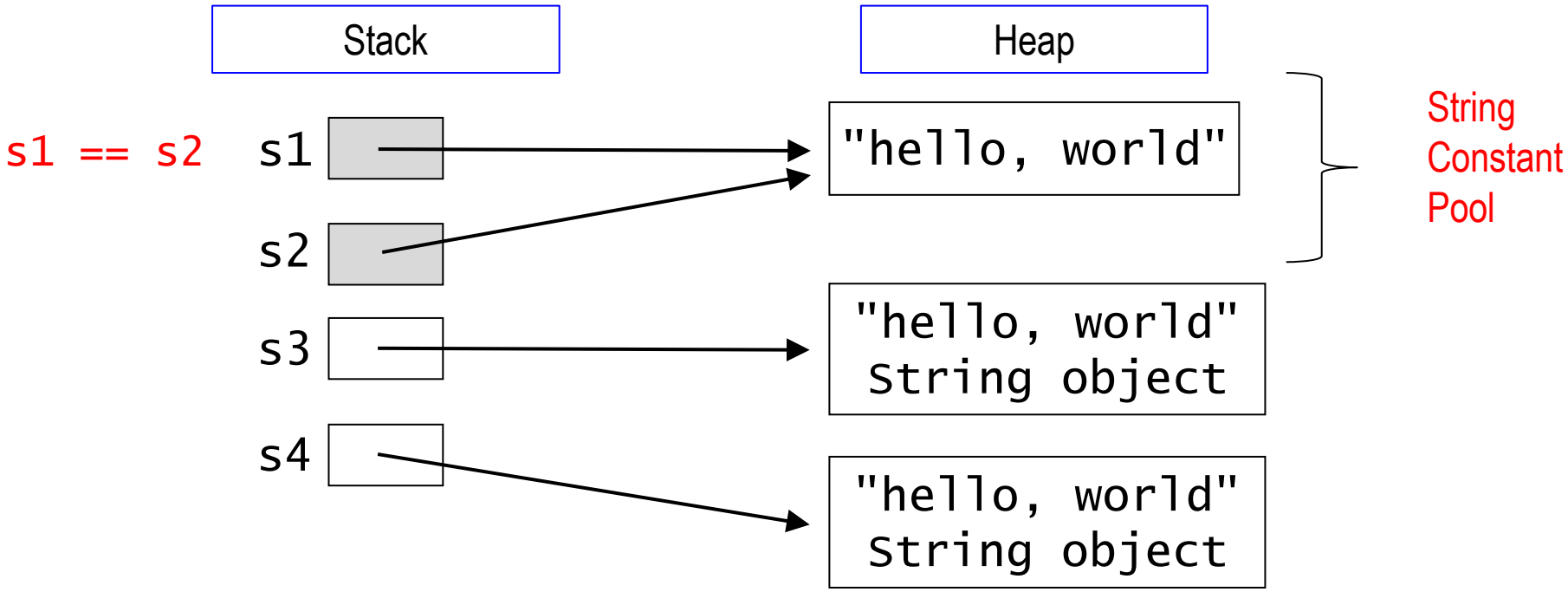




Testing for Equivalent *Strings*

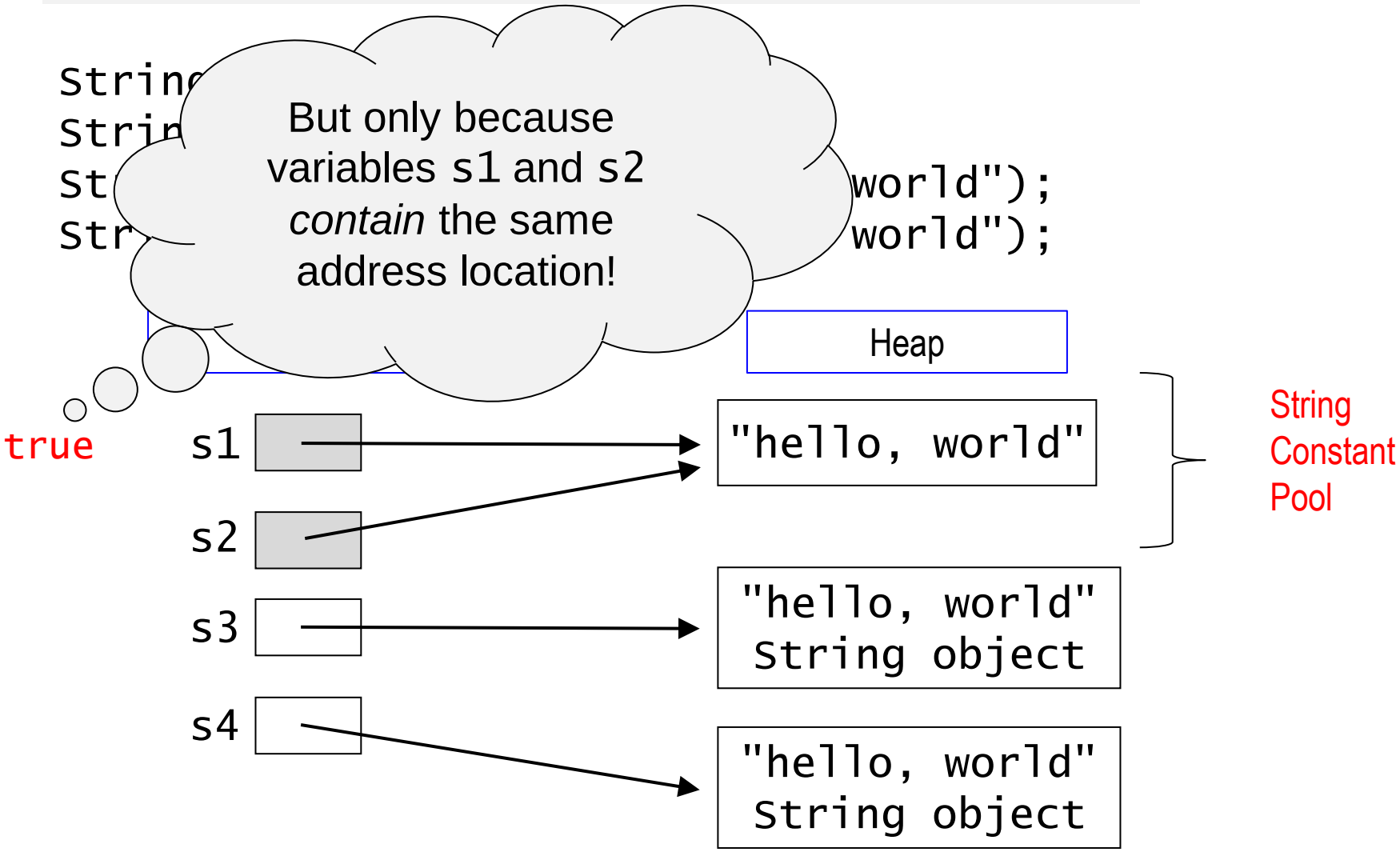
- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



Testing for Equivalent *Strings*

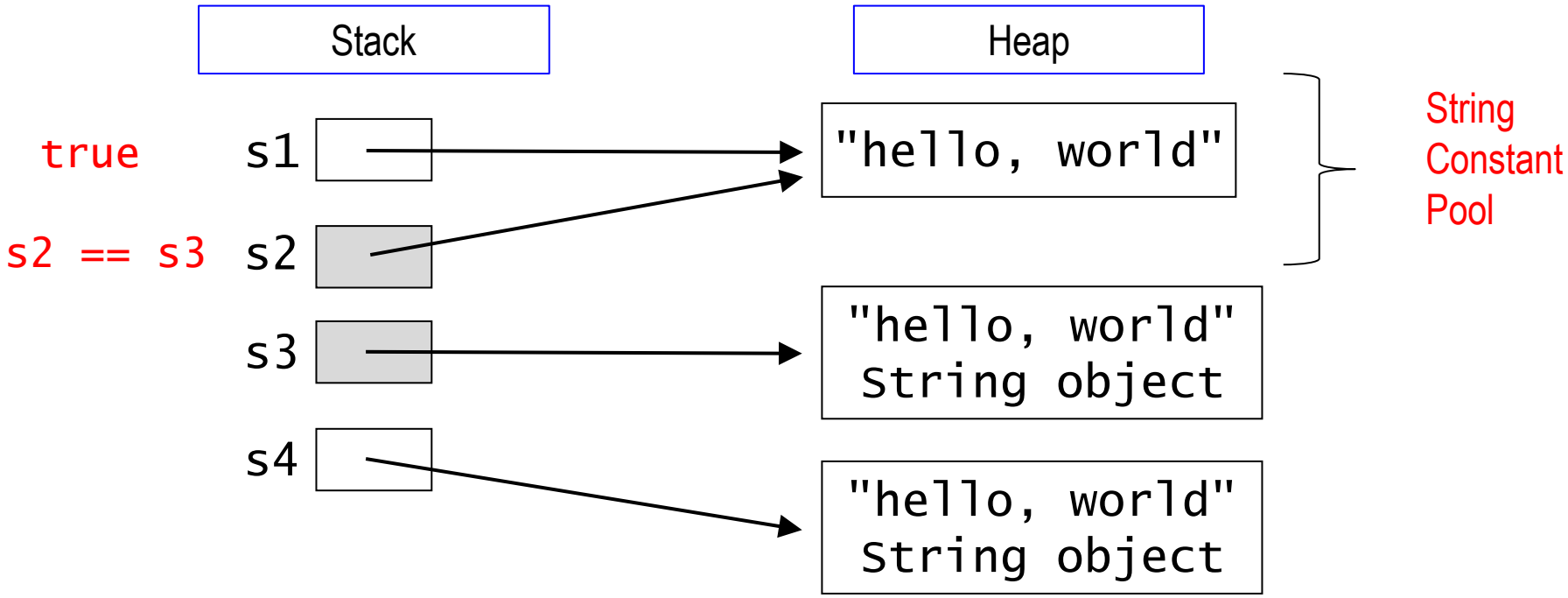
- The `==` and `!=` operators do *not* typically work when comparing *objects*



Testing for Equivalent *Strings*

- The `==` and `!=` operators do *not* typically work when comparing *objects*

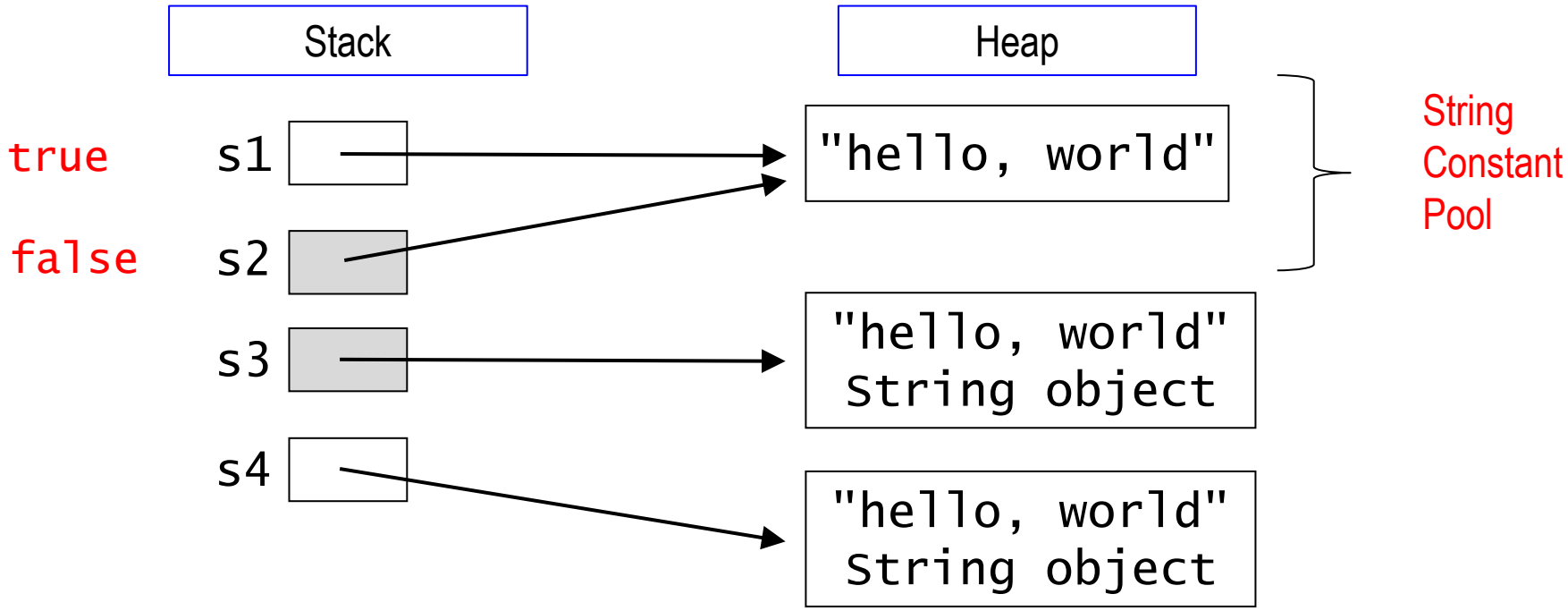
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



Testing for Equivalent *Strings*

- The `==` and `!=` operators do *not* typically work when comparing *objects*

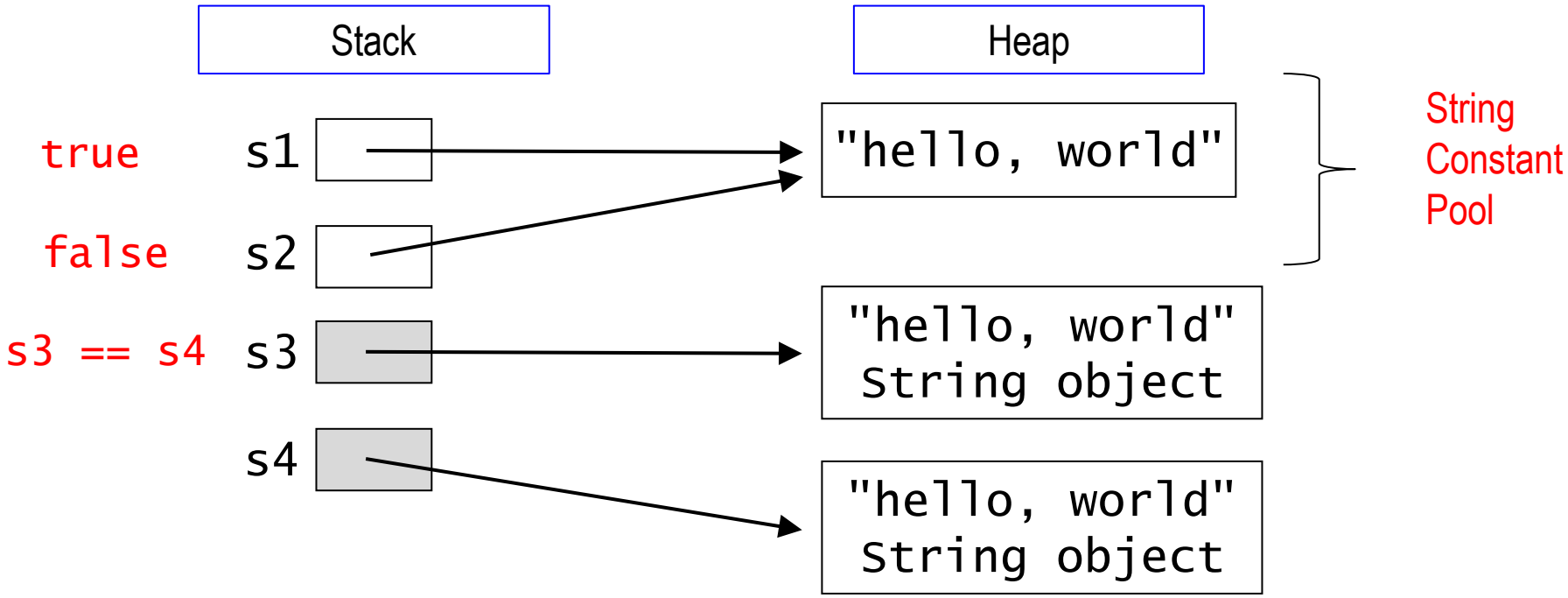
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



Testing for Equivalent *Strings*

- The `==` and `!=` operators do *not* typically work when comparing *objects*

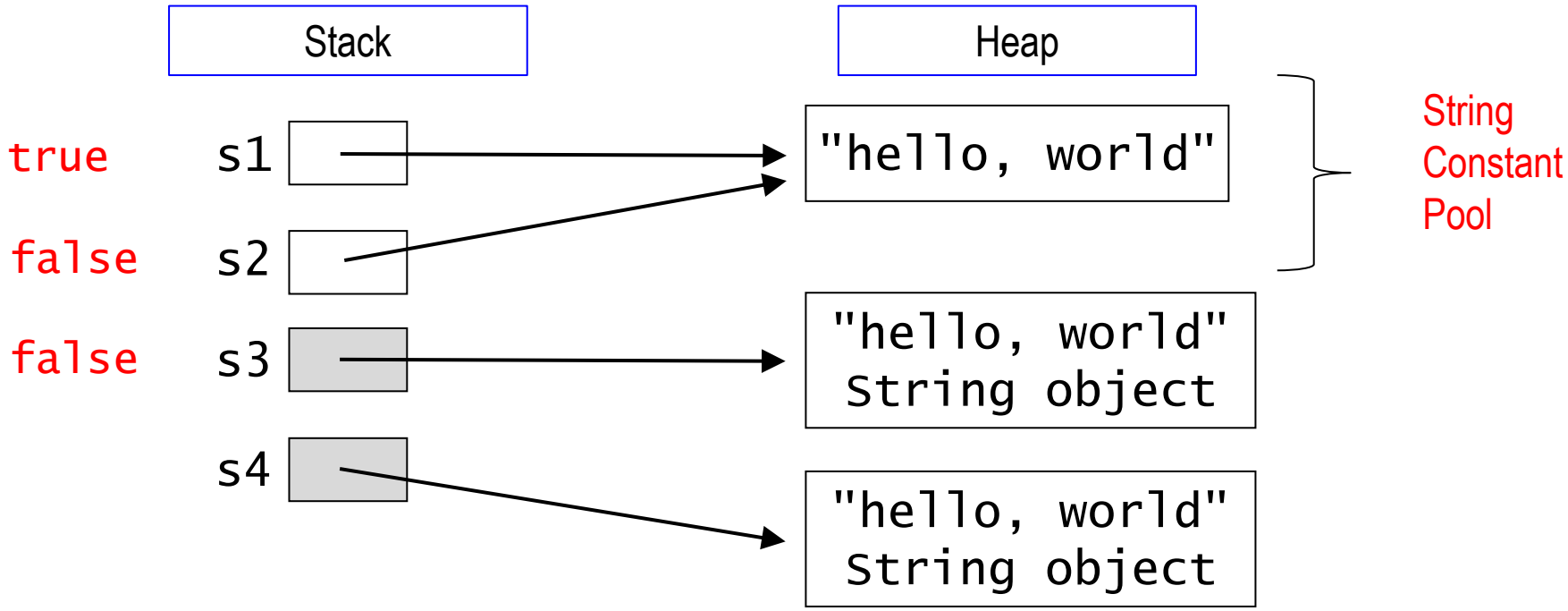
```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



Testing for Equivalent *Strings*

- The `==` and `!=` operators do *not* typically work when comparing *objects*

```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");
```



Testing for Equivalent *Strings:*
another look

- The == and != operators do *not* typically work when comparing *objects*

```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");

if ( s1.equals(s2) )
   // the strings referenced by s1 and s2 are identical

if ( s2.equals( s4 )
   // the strings referenced by s2 and s4 are identical

String s5 = new String("Hello, world");

if ( s4.equals( s5 )
  // the strings referenced by s4 and s5 are not identical
```

Testing for Equivalent *Strings:*
another look

- The == and != operators do *not* typically work when comparing *objects*

```
String s1 = "hello, world";
String s2 = "hello, world";
String s3 = new String("hello, world");
String s4 = new String("hello, world");

if ( s1.equals(s2) )
   // the strings referenced by s1 and s2 are identical

if ( s2.equals( s4 )
   // the strings referenced by s2 and s4 are identical

String s5 = new String("Hello, world");

if ( s4.equalsIgnoreCase( s5 )
   // the strings referenced by s4 and s5 are identical
   // except for case
```

The API of a Class

- The methods defined within a class are known as the *API* of that class.
  - API = application programming interface
- We can consult the API of an existing class to determine which operations are supported.
- The API of all classes that come with Java is available here:

  `https://docs.oracle.com/javase/8/docs/api/`

  - there's a link on the resources page of the course website

Understanding Java Strings

Assume the following code segment:

```
String s1 = "hello";
String s2 = s1;
String s3 = new String( "hello" );
String s4 = "hello";
String s5 = new String( "hello" );
s1 = "hello world";
```

Which of the following statement is true?

A. s1 == s2
B. s3 == s4
D. s3 == s5
E. s2 == s4
F. None are true
G. All are true
H. Some combination, but not sure which?

Understanding Java Strings

Assume the following code segment:

```
String s1 = "hello";
String s2 = s1;
String s3 = new String( "hello" );
String s4 = "hello";
String s5 = new String( "hello" );
s1 = "hello world";
```

Which of the following statement is true?

A. s1 == s2
B. s3 == s4
D. s3 == s5
E. s2 == s4
F. None are true
G. All are true
H. Some combination, but not sure which?