ECS 140A
Programming Languages

October 12, 2023

Language #1 - Java

The language of the Internet

Java was created at Sun Microsystems by James Gosling and Bill Joy. The language is designed to be a machine-independent programming language (a very important concept) that is safe enough to be sent across networks (so that you don't have to worry that some stranger's Java code will run on your computer) and powerful enough to do what you've come to expect of machine-specific (native) executable code.

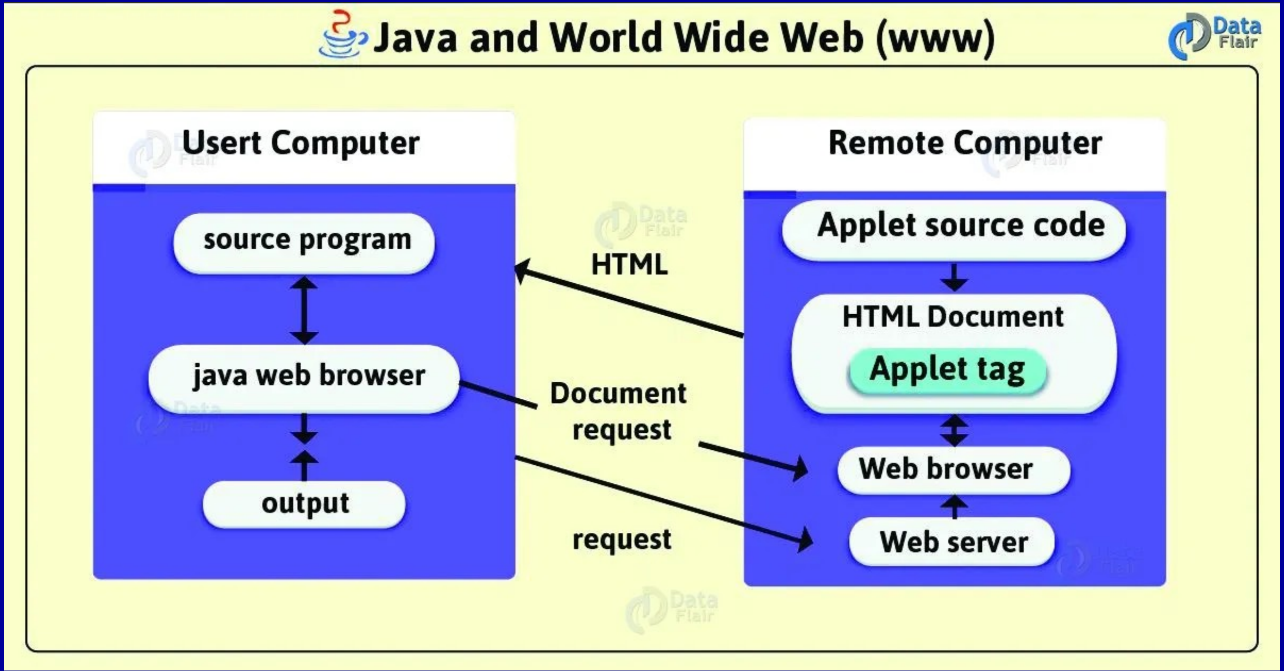
The language of the Internet

Java has its roots in a language called Oak, which the Sun folks used to program interactive TV set-top boxes. But Sun abandoned set-top boxes, and Joy and Gosling had a language with no place to go. In 1993, the World Wide Web erupted, and these guys had a small, robust, platform-independent and object-oriented language for use in cable TV networks.

But hey, the same features that made Oak good for those networks would also make it good for the Internet. So Gosling and Joy did a little more work, and Sun started distributing Java as their choice for the language of the Internet.

The language of the Internet

Back in the day, the Internet was built on small Java programs called applets. HotJava, a browser written in Java, would find these applets when a user visits a website, and compile and run the Java applets on user's local machine. These applets provide dynamic functionalities of a website (graphics, visualizations, etc.)

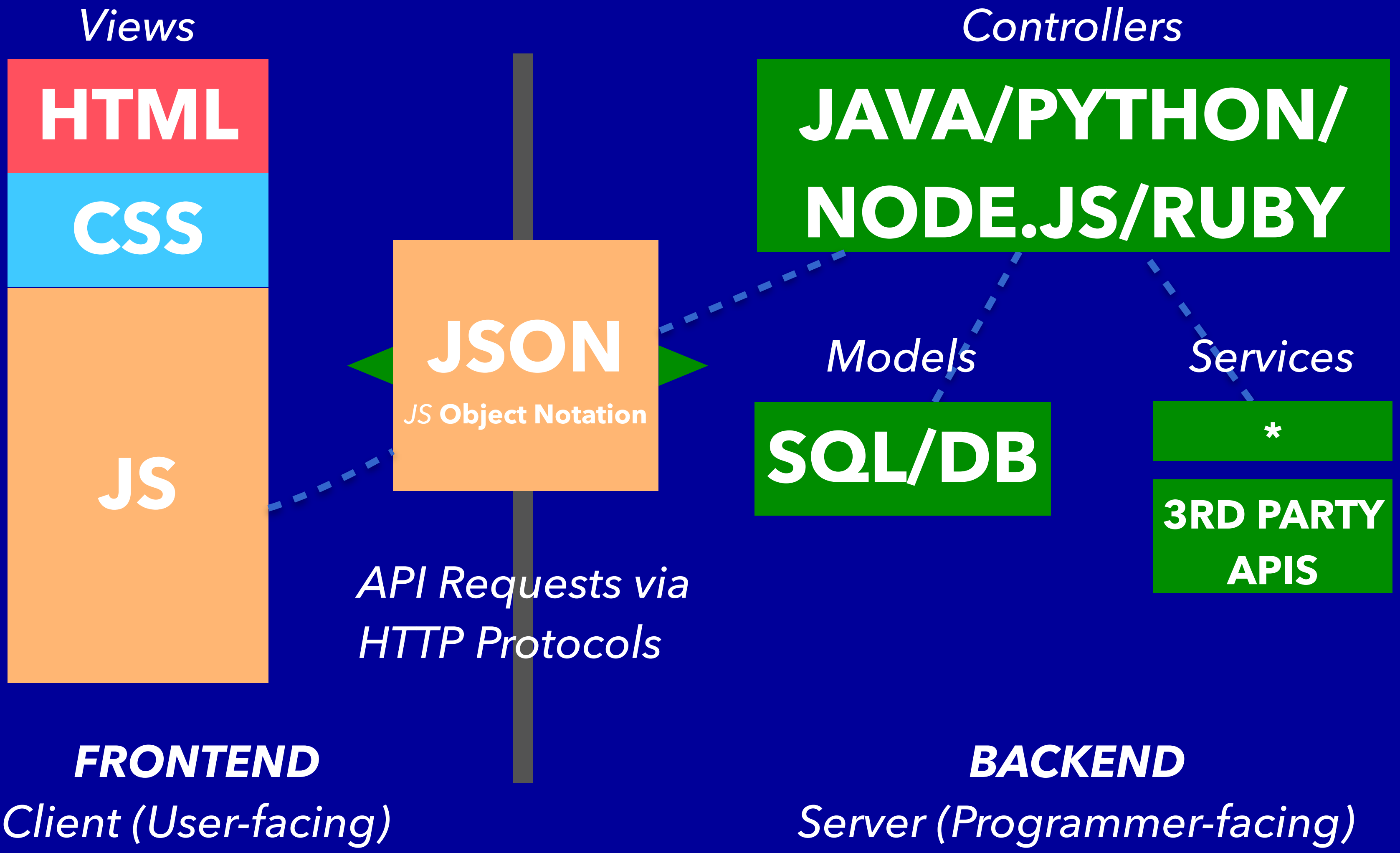


The language of the Internet

Web programming looks very different since 2010s:
Java is typically used as the backend language to write Application Programming Interfaces (APIs) and backend services, which perform business logics, interact with databases, etc.
Frontend (JavaScript) would query these APIs through various HTTP protocols (GET, POST, PUT, DELETE).
Until 2011, Java applets were many times faster than JavaScript due to access to hardware acceleration. As browsers gained support for hardware acceleration (canvas API, WebGL), most of the graphics are now rendered by the frontend in JavaScript.

Anatomy of a modern web app

How Java is used these days in web dev…



Design principles

There were five primary goals in the creation of the Java language:

1. It must be "simple, object-oriented, and familiar".
2. It must be "robust and secure".
3. It must be "architecture-neutral and portable".
4. It must execute with "high performance".
5. It must be "interpreted, threaded, and dynamic".

Platform independence

Any Java code written on any hardware (Motorola, Intel, RPi, ARM, RISCV, whatever) under any operating system (Mac, Windows, Unix, Linux, etc.) will run on any other system. If it doesn't, then by definition it's not Java (or at least it's not the current version of Java). That's a very nice feature. A language that didn't have that feature couldn't very well be the language of the Internet, could it? In other words, if you write some program on your Windows box that you want to run on my Macintosh when I'm browsing your web site, the program needs to be platform independent.

Platform independence

Java achieves that independence through what was then a fairly unusual approach. Instead of just distributing big Java compilers for every known platform, Java language processors are broken into two smaller pieces, each of which is somewhat easier to create than a larger monolithic Java compiler would be.

Platform independence

The first piece is a compiler too, but it compiles high-level Java instructions into a universally-accepted set of instructions called Java byte-code (or J-code). A C compiler, on the other hand, typically generates instructions in the machine language of the host computer. So to write a C compiler for one platform may take a whole lot of different knowledge than writing the same compiler for a different platform.

Platform independence

That byte-code is then executed on something called a Java run-time interpreter (not a compiler, so a Java system is part compiler, part interpreter...it's called a hybrid). That interpreter often goes by the name "Java virtual machine" or JVM for short. The JVM does all the work of a hardware processor executing machine instructions, but the extra layer of software allows a degree of safety we'll talk about soon. The JVM manages the activation stack and storage heap, manipulates data types, and so on, and does so within a strictly-defined open specification that anyone can follow who wants to implement a JVM on a given platform.

Platform independence

So anyone who wants to write the Java compiler front end need only worry about translating Java source code to Java byte code...they don't have to worry about machine-specific issues. And the folks who have to worry about getting the JVM working only have to worry about evaluating a standard byte-code with a program running in the context of some native machine language, a task which supposedly is much more manageable than converting Java source directly to native machine instructions. Frankly, none of these jobs sounds appealing to me. In any case, the Java interpreters are relatively small by comparison to full blown compilers, and can be implemented in whatever form or language is desirable for a given platform.

Platform independence

But wait…there’s more!!

The JVM was intended to interpret byte code generated by a Java compiler, but that’s not a requirement. So you can create a compiler for some other language that generates Java byte code, and you can build that compiler without worrying about the machine-level details of the computer that you ultimately want your program to run on. Your compiler just needs to know how to produce the byte code, and then the byte code can run on any machine that has a JVM installed.

Platform independence

Are there languages that work this way? To name a few:

Java (obviously)
Scala
Clojure
Kotlin
Groovy
Processing
(plus lots of little ones)

And then implementations of existing languages:

Golang, COBOL, Haskell, Pascal, PHP, Prolog, Python, Ruby, Scheme, and many more

Safety and security

When you're browsing the Web, there are lots of programs running on your computer that didn't come with your computer, that you didn't write, and that you didn't buy. They're little Java programs that fly across the Internet from the Web site you're looking at (the server) to your computer (the client). They make the little animations happen, and they do the exciting flashing banner advertisements that you ignore, and so on. But the point here is that there are programs running on your computer that you didn't agree to have on your computer. How do you know those programs aren't installing viruses on your machine or stealing your credit card numbers or something?

Safety and security

Turns out there are some different things that the Java folks thought about to protect you from the dangers of running somebody else's programs on your machine. First of all, and this protects you from yourself too, Java prevents programs from doing stuff in memory locations that they shouldn't be messing with. With Java, you don't get to do the kind of pointer arithmetic that you can do in C. Java checks memory references at run-time, so Java programs can only access parts of memory that are set aside for that sort of thing.

Safety and security

Beyond that sort of thing, the Java Virtual Machine includes a security manager which controls access to system resources like the filesystem, network ports, and the windowing environment. The security manager can be adjusted to be more or less trusting of outside code, as the application requires. The security manager in turn relies on a class loader to manage classes brought to your computer from other sources and keep them somehow separate from your own trusted code.

Safety and security

And the whole thing relies on something called a verifier to make sure that Java byte-code to be executed is well-behaved and obeys the rules of Java. Verified code can't forge pointers or violate access permissions on objects. It can't cast values from one data type to another in illegal ways or use objects in ways that weren't intended. In short, verified code can't do lots of things that nasty people like to do to get inside your machine.

Performance

In terms of computing power, Java lets you do pretty much whatever you need to do (except maybe creep around in areas of memory that you shouldn't be creeping around). The big complaint about Java comes in the speed department. In general, an interpreted program in language X will run slower than the same compiled program, so interpreters trade away speed in return for other things like run-time detection of errors and stuff like that. So it should be no surprise that since at least part of the Java system is an interpreter, Java programs might take more time to execute than their C++ counterparts, for example.

Performance

However, that performance hit is mitigated in a couple of ways. First, since you're interpreting byte-code instead of source code, interpretation can go a lot faster (the instruction set is smaller and the interpreter itself can be smaller and faster). Second, there are ways to optimize byte-code by compiling byte-code in small chunks as needed, on the fly, using "just in time" compilation.

Java quickstart guide

A program is a collection of classes.

Below is a definition of a class. A class is a package of instructions that specify what kinds of data will be operated on and what kinds of operations there will be. All the Java programs you write will consist of one or more classes. Nothing happens without a class, because classes spawn objects. Nothing happens without objects either (usually).

```
public class Oreo
{
  public static void main (String[] args)
  {
    System.out.println ("Feed me more Oreos!");
  }
}
```

A sample Java application program

The inner part is the definition of a method. A method is a group of Java statements (instructions) that has a name and performs some task. Here the name is “main”.

```
public class Oreo
{
  public static void main (String[] args)
  {
    System.out.println ("Feed me more Oreos!");
  }
}
```

All the Java programs you create will have a main method. It’s where the execution of the program begins.

The class name must match the file name.

Reserved words

There are ~~52~~ ~~53~~ ~~51~~ ?? reserved words here…the number changes occasionally

```
abstract    do          if          private     throw
assert
boolean     double      implements  protected   throws
break       else        import      public      transient
byte        enum        instanceof  return      true
case        extends     int         short       try
catch       false       interface   static      void
char        final       long        strictfp    volatile
class       finally     native      super       while
const       float       new         switch
continue    for         null        synchronized
default     goto        package     this
```

Identifiers

An identifier must start with a letter and be followed by zero or more letters and/or digits. Digits are 0 through 9. Letters are the 26 characters in the English alphabet, both uppercase and lowercase, plus the $ and _ (also alphabetic characters from other languages). There's no limit to how many characters you can put in your identifiers

Which of the following are not valid identifiers?

```
userName      user_name      $cash      2ndName

first name    user.age       _note_     note2
```

Identifiers

An identifier must start with a letter and be followed by zero or more letters and/or digits. Digits are 0 through 9. Letters are the 26 characters in the English alphabet, both uppercase and lowercase, plus the $ and _ (also alphabetic characters from other languages). There's no limit to how many characters you can put in your identifiers

Which of the following are not valid identifiers?

`userName`	`user_name`	`$cash`	`2ndName`
`first name`	`user.age`	`_note_`	`note2`

*** We recommend camelCase**

Identifiers

Java is case sensitive.

```
        Oreo        oreo        OREO
```

are all different identifiers, so be careful. This is a common source of errors in programming.

White space

White space: the blanks between the identifiers and other symbols. Tabs and newline characters are included.

White space doesn’t affect how the program runs.

White space

```
//*******************************************************
// Oreo.java        Author:  Kurt Eiselt
//
// Demonstrating good use of white space
//*******************************************************

public class Oreo
{
  public static void main (String[] args)
  {
    System.out.println ("Feed me more Oreos!");
  }
}
```

This code can be found on tyfeng.com/ecs140a

White space

```
//*******************************************************
// Oreo3.java        Author:  Kurt Eiselt
//
// Demonstrating totally bizarre use of white space
//*******************************************************

    public
class     Oreo3
        {
  public                         static
void
         main  (String[] args)
                              {
  System.out.println
                                ("Feed me more Oreos!")
;
        }
             }
```

White space

```
//**********************************************************
// Oreo4.java       Author:  Kurt Eiselt
//
// Demonstrating deep psychological issues
//**********************************************************

public
class
Oreo4
{
public
static
void
main
(
String[]
args
)
{
System.out.println
("Feed me more Oreos!")
;
}
}
```

Got Java?

Most computers already have the Java Virtual Machine (JVM) installed. You may read or hear about the JVM using different words, like Java Runtime Environment (JRE). They're the same thing. Having the JRE installed means your computer can execute Java programs that have already been compiled. But your computer may not have the Java compiler already installed.

To install the Java compiler on your computer, you'll want to download the Java Software Development Kit (or Java SDK), which is also called just the Java Development Kit (or JDK).

Got Java?

Point your web browser to

https://www.oracle.com/technetwork/java/javase/downloads/index.html

or here

https://openjdk.java.net

Data types

For every variable, we have to declare a data type.

We can use one of eight primitive data types provided by the Java language.

For something more complicated than those eight primitive data types, we can use data types created by others and provided to us through the Java libraries, or we can invent our own.

Primitive data types (part one)

Type	Size	Min	Max
`byte`	1 byte	-128	127
`short`	2 bytes	-32,768	32,767
`int`	4 bytes	-2,147,483,648	2,147,483,647
`long`	8 bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
`float`	4 bytes	approx -3.4E38 (7 sig.digits)	approx 3.4E38 (7 sig.digits)
`double`	8 bytes	approx -1.7E308 (15 sig. digits)	approx 1.7E308 (15 sig. digits)

Six primitive data types are all about numbers...are they integer or floating point...and how big they can be.

The primitive data types all occupy fixed size in memory.

Primitive data types (part two)

Character Type:
Java has one character type named char. Java uses the Unicode character set and so each char occupies 2 bytes of memory. (This is not the same as a character string.)

Boolean Type:
Java has one Boolean type named boolean. Variables of type boolean have only two valid values: true and false. How much memory is used by a boolean type is left up to the discretion of whoever writes the Java Virtual Machine.

Variable declarations

```
public class Test3
{
    public static void main (String[] args)
    {
        int a;
        int b = 3;
        int c;
        c = 5;
        a = b + c;
        System.out.println ("The answer is " + a);
    }
}
```

This code can be found on tyfeng.com/ecs140a

The conditional statement

The conditional statement allows us to choose which statement will be executed next based on a boolean expression (a test or a condition) which evaluates to either true or false. For example:

```
if (age < 20)
   System.out.println("Really, you look like you are "
                                    + (age + 5) + ".");
```

would print the intended message if the value assigned to the variable `age` is less than 20.

Block statements

We often want to do a bunch of things based on some condition, not just one. Java allows us to replace a single statement with multiple statements by surrounding those statements with (curly) braces:

```
if (x == y)  // == is an equality test; = is assignment
{
   System.out.println("x equals y!");
   System.out.println("I'm happy");
}
else
{
   System.out.println("x is not equal to y");
   System.out.println("I'm depressed");
   System.out.println("How about you?");
}
```

Programming with classes and objects

What do we do when the data we want to work with is more complex than can be accommodated by the few primitive data types?

We make our own data type! We do that by creating a class. A class specifies the nature of the data we want to work with as well as the operations that can be performed on that kind of data. The operations that are defined within a class are called methods.

More accurately, before we make our own we check to see if someone else hasn't already created our desired class for us. We do that by looking at the Java libraries written by other programmers.

Programming with classes and objects

For example, Java provides us with a `char` primitive data type which is a single character, but what if I want to work with a sequence of characters -- a character string?

A quick(?) look at the online Java library documentation at `https://docs.oracle.com/javase/8/docs/api/` tells us that there is a String class already written for us. (It's also described in your book.) The documentation tells us everything we need to know to use this class.

What if you want a fancier version of **`int`**? There is an Integer class written for you as well. https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html

The classes in the libraries are often referred to as the Application Programming Interfaces or just API.

Comparing Strings

To compare two Strings, use **String.equals()** instead of ==.
String.equals() is a method that compares the content of two Strings.
== is an operator that checks if both objects point to the same memory location.
You may have two Strings that point to different memory locations, but have the same value.

```
String s1 = "HELLO";
String s2 = "HELLO";
String s3 = new String("HELLO");

System.out.println(s1 == s2); // true
System.out.println(s1 == s3); // false
System.out.println(s1.equals(s2)); // true
System.out.println(s1.equals(s3)); // true
```

int - String conversion by method

int to String:

```
String five = 5; // ERROR!
String five = Integer.toString (5);
String five = “” + 5; // five = “5”
```

String to int:

```
int foo = “18”; // ERROR!
int foo = Integer.parseInt (“18”);
```

Programming with classes and objects

Recall that a primitive data type like `int` allows us to create multiple variables of type `int`, each with a different name, and each with its own assigned value -- in this case, an integer.

In somewhat the same way, the String class allows us to create multiple instances of the class String, each with a different name and each with its own associated value -- in this case, a character string.

A specific instance of a class is called an object. Classes are templates for objects. We as programmers define classes; objects are created from classes.

Creating classes and objects

Let’s say we wanted to quit this computer science career path and choose to make millions by owning and operating a bunch of vending machines. It could happen. But first, we want to simulate our collection of vending machines, using our Java programming skills. We could start off by creating a CokeMachine class, then making some CokeMachine objects from that class definition and see how they work. Or course, this is going to be a very very simplified simulation. If you want more robustness, feel free to add more code.

Creating classes and objects

A minimal simulation of a soda pop vending machine has to, at the very least, keep track of how many cans of pop were in the machine, and there has to be a way of “buying” a can of pop from the machine. So our vending machine class, which we call CokeMachine, will be written so that there’s a variable for keeping track of how many cans of pop remain. Also, there will be a method that simulates what happens when someone buys a can of pop from a machine. The next few slides highlight these two components.

Creating classes and objects

```
public class CokeMachine  // This is how we begin any class
{
```

This code can be found on tyfeng.com/ecs140a

```
}
```

Creating classes and objects

```
public class CokeMachine
{
  public int numberOfCans;  // Here's how we specify that any object
                            // created from the CokeMachine class
                            // has a variable to keep track of how
                            // many cans of pop remain.
```

This code can be found on tyfeng.com/ecs140a

```
}
```

Creating classes and objects

```
public class CokeMachine
{
  public int numberOfCans;






  public void buyCoke()                        // Here's the method for buying
  {                                            // a can of pop
    if (numberOfCans > 0)                      // If the machine is not empty
    {
      numberOfCans = numberOfCans - 1;         // then decrease the number of
      System.out.println("Have a Coke");       // cans by one and print an
      System.out.print(numberOfCans);          // appropriate message
      System.out.println(" cans remaining");
    }
    else                                       // otherwise the machine is
    {                                          // empty, so print "sold out"
      System.out.println("Sold Out");
    }
  }
}
```

This code can be found on tyfeng.com/ecs140a

Creating classes and objects

The CokeMachine class needs one more component. In Java, classes don't do anything except serve as templates for the objects generated by those classes. Nothing in Java really happens until an object is created from a class; objects do all the work (almost). So our CokeMachine class needs to specify what happens when a new CokeMachine object is created from the CokeMachine class specification.

The specification of what happens when a new objects is created from a class is given by the constructor method for that class. The constructor method is always given the same name as the class it's in. That's how Java knows where to find the constructor method. In our case, the constructor method will be called CokeMachine, but we usually distinguish class names from method names by putting () at the end of the method names (to indicate that there might be parameters...more on that later). So we write our constructor method's name as CokeMachine().

Creating classes and objects

Our simple CokeMachine() constructor method doesn’t do much. When this constructor method is invoked, the numberOfCans variable for the new CokeMachine object is initialized to some value, and a message is printed to let us know that a new CokeMachine object has been created.

Creating classes and objects

```
public class CokeMachine
{
  public int numberOfCans;

  public CokeMachine()   // This is the constructor method
  {
    numberOfCans = 2;
    System.out.println("Adding another machine to your empire");
  }

  public void buyCoke()
  {
    if (numberOfCans > 0)
    {
      numberOfCans = numberOfCans - 1;
      System.out.println("Have a Coke");
      System.out.print(numberOfCans);
      System.out.println(" cans remaining");
    }
    else
    {
      System.out.println("Sold Out");
    }
  }
}
```

Creating classes and objects

```
public class CokeMachine
{
  public int numberOfCans;

  public CokeMachine()   // This is the constructor method
  {
    numberOfCans = 2;
    System.out.println("Adding another machine to your empire");
  }

  public void buyCoke()
  {
    if (numberOfCans > 0)
    {
      numberOfCans = numberOfCans - 1;
      System.out.println("Have a Coke");
      System.out.print(numberOfCans);
      System.out.println(" cans remaining");
    }
    else
    {
      System.out.println("Sold Out");
    }
  }
}
```

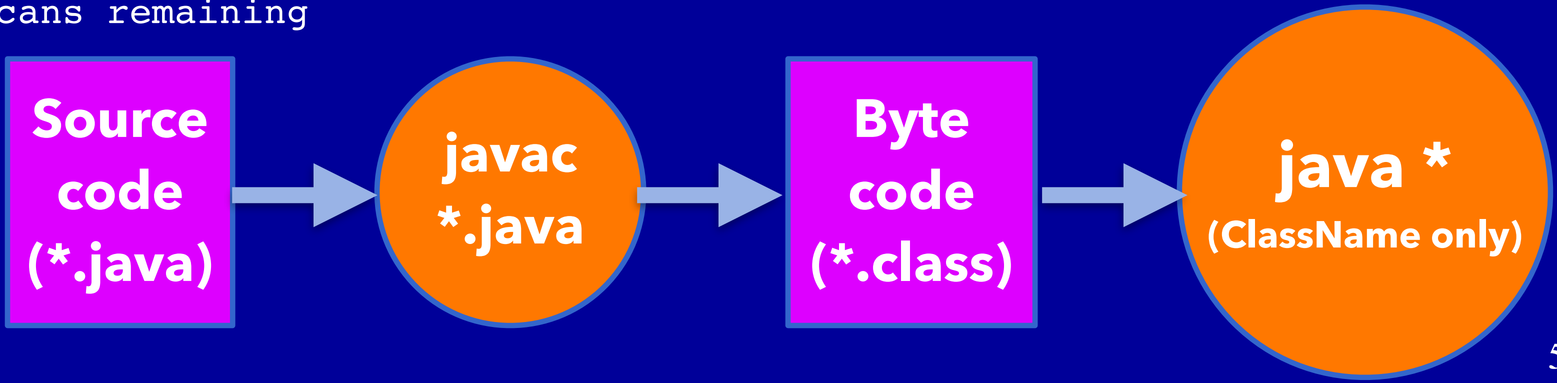
The SimCoke simulator

```
public class SimCoke
{
  public static void main (String[] args)
  {
    System.out.println("Coke machine simulator");
    CokeMachine csmachine = new CokeMachine();    // create two unique
    CokeMachine engrmachine = new CokeMachine();  // CokeMachine objects
    csmachine.buyCoke();     // buy Cokes from the CS machine until it
    csmachine.buyCoke();     // is sold out
    csmachine.buyCoke();
    engrmachine.buyCoke();   // buy Cokes from the Engineering machine
    engrmachine.buyCoke();   // and note that there are Cokes in this
  }                          // machine even though the CS machine is
}                            // empty - these two objects are unique
```

This code can be found on tyfeng.com/ecs140a

Compiling and running

```
Last login: Thu Oct 17 23:12:23 on ttys000
Kurt-Eiselts-Home-iMac:~ eiselt$ javac CokeMachine.java
Kurt-Eiselts-Home-iMac:~ eiselt$ javac SimCoke.java
Kurt-Eiselts-Home-iMac:~ eiselt$ java SimCoke
Coke machine simulator
Adding another machine to your empire
Adding another machine to your empire
Have a Coke
1 cans remaining
Have a Coke
0 cans remaining
Sold Out
Have a Coke
1 cans remaining
Have a Coke
0 cans remaining
```



Quiz time! Here's the problem

GasStation

As part of a simulation of the behavior of gas stations, you are to create a class definition called `GasStation`. Each object of the `GasStation` class keeps track of how many litres of gasoline are available at that gas station, represented as an integer. A `GasStation` object also has a method to simulate the dispensing of some number of litres of gasoline. That method, called `dispenseGas()`, reduces the amount of gasoline available at that gas station by the number of litres dispensed, assuming there are at least as many litres available as the number of litres to be dispensed. If the number of litres of gasoline to be dispensed is greater than the number of litres available, then no gasoline is dispensed. Another method, called `getAvailableGas()`, returns the number of litres of gasoline available at that gas station.

Every gas station object has 10,000 litres of gasoline when it is created. The `dispenseGas()` method always dispenses exactly 100 litres of gasoline when it is invoked.

Navigate to tyfeng.com/ecs140a/lecture5_practice.html

Quiz time! Here's the problem

Here's a simple program that we might use to test your `GasStation` class:

```
public class GasStationTester
{
  public static void main (String[] args)
  {
    GasStation chevron = new GasStation();
    GasStation shell = new GasStation();
    System.out.println("Chevron has " + chevron.getAvailableGas());
    chevron.dispenseGas(); // 100 litres dispensed
    chevron.dispenseGas(); // 100 more litres dispensed
    System.out.println("Chevron has " + chevron.getAvailableGas());
    System.out.println("Shell has " + shell.getAvailableGas());
    shell.dispenseGas();   // 100 litres dispensed
    System.out.println("Shell has " + shell.getAvailableGas());
  }
}
```

And here's the output we would expect after running this program (assuming of course that everything compiles correctly):

```
> javac GasStation.java
> javac GasStationTester.java
> java GasStationTester
Chevron has 10000
Chevron has 9800
Shell has 10000
Shell has 9900
```

Quiz time! Here's *a* solution

```
public class GasStation {
  // instance variables go here
  public int availableGas;
  final public int START_AMOUNT = 10000;
  final public int DISPENSE_AMOUNT = 100;

  // constructor methods go here
  public GasStation() {
    availableGas = START_AMOUNT;
  }

  // instance methods go here
  public void dispenseGas() {
    if (availableGas >= DISPENSE_AMOUNT) {
      availableGas = availableGas - DISPENSE_AMOUNT;
    }
  }

  public int getAvailableGas() {
    return availableGas;
  }
}
```

Quiz time! Here's another problem

Create a class called FooCorporation. Write a method to print pay based on base pay and hours worked for an employee.
Overtime: More than 40 hours, paid 1.5 times base pay.
Minimum Wage: $15.00/hour. If base pay rate is not provided (null), is not valid int, or is less than 15, then use minimum wage.
Maximum Work: 60 hours a week. If an employee has worked more than 60 hours, only the first 60 hours will be paid.

The method should take 3 arguments (`String employeeName, double hoursWorked, double basePayRate`). It should print "<EmployeeName> has a total pay of $<TotalPay>." when it is called with those 3 arguments.

Navigate to tyfeng.com/ecs140a/lecture5_practice.html

Quiz time! Here's *a* solution

```
public class FooCorporation {
    private static final double OVERTIME_PAY_FACTOR = 1.5;
    private static final double MINIMUM_WAGE = 15.00;
    private static final double MAXIMUM_HOURS = 60.00;

    public void calculatePay(String employeeName, double hoursWorked, Double basePayRate) {
      // if basePayRate is null or smaller than min wage, assign MINIMUM_WAGE to validBasePayRate.
Else, use basePayRate.
        double validBasePayRate = (basePayRate == null || basePayRate < MINIMUM_WAGE)
                ? MINIMUM_WAGE : basePayRate;
        /* This is the equivalent code implemented using if and else as the line above
        double validBasePayRate;
        if (basePayRate == null || basePayRate < MINIMUM_WAGE) {
           validBasePayRate = MINIMUM_WAGE;
        } else {
           validBasePayRate = basePayRate;
        } */
        double validHours = Math.min(hoursWorked, MAXIMUM_HOURS);
        double regularPay = validHours <= 40.00 ? validHours * validBasePayRate :
                            40.00 * validBasePayRate;
        double overTimePay = validHours > 40.00 ? (validHours - 40.00) * validBasePayRate *
OVERTIME_PAY_FACTOR : 0.00;
        double totalPay = regularPay + overTimePay;
        System.out.println(employeeName + " has a total pay of $" + totalPay + ".");
    }

    public static void main(String[] args) {
        FooCorporation fooCorp = new FooCorporation();
        fooCorp.calculatePay("John Doe", 45.00, 20.00);
        fooCorp.calculatePay("Jane Doe", 65.00, null);
        fooCorp.calculatePay("Doe John", 35.00, 10.00);
    }
}
```

Questions?

Using the `while` statement

```
public class WhileDemo
{
  public static void main (String[] args)
  {
    int limit = 3;
    int counter = 1;

    while (counter <= limit)
    {
      System.out.println("The square of " + counter +
                         " is " + (counter * counter));
      counter = counter + 1;
    }
    System.out.println("End of demonstration");
  }
}
```

Other loop statements

```
public class ForDemo
{
  public static void main (String[] args)
  {

    for (int counter = 1; counter <= 3; counter++)
    {
      System.out.println("The square of " + counter +
                         " is " + (counter * counter));
    }
    System.out.println("End of demonstration");
  }
}
```

This for loop is the equivalent of the while loop on the previous slide.

Visibility modifiers

```
public class SimCoke
{
  public static void main (String[] args)
  {
    System.out.println("Coke machine simulator");
    CokeMachine csmachine = new CokeMachine();
    CokeMachine engrmachine = new CokeMachine();
    csmachine.buyCoke();
    csmachine.buyCoke();
    csmachine.buyCoke();
    engrmachine.buyCoke();
    engrmachine.buyCoke();
  }
}
```

How can we refill an empty Coke Machine?

Visibility modifiers

```
public class SimCoke
{
  public static void main (String[] args)
  {
    System.out.println("Coke machine simulator");
    CokeMachine csmachine = new CokeMachine();
    CokeMachine engrmachine = new CokeMachine();
    csmachine.buyCoke();
    csmachine.buyCoke();
    csmachine.buyCoke();
    csmachine.numberOfCans = 12465;
    csmachine.buyCoke();
    engrmachine.buyCoke();
    engrmachine.buyCoke();
  }
}
```

How can we refill an empty Coke Machine? Does this work? Is it a good idea?

Visibility modifiers

```
public class CokeMachine
{
  public int numberOfCans;

  public CokeMachine()
  {
    numberOfCans = 2;
    System.out.println("Adding another machine to your empire");
  }
```

Our problem is right up there. We're using the public visibility modifier to describe our variable numberOfCans. Variables (and methods) declared to have public visibility can be directly accessed from anywhere outside of the object created from the class. That violates the principle of encapsulation.

Encapsulation is the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components. Encapsulation is used to hide the values or state of a structured data object inside a class, preventing direct access to them by clients in a way that could expose hidden implementation details or violate state invariance maintained by the methods.
Thanks, Wikipedia.

Visibility modifiers

```
public class CokeMachine
{
  public int numberOfCans;

  public CokeMachine()
  {
    numberOfCans = 2;
    System.out.println("Adding another machine to your empire");
  }
```

To prevent direct external access to the variable while still allowing the methods defined within the class to access the same variable, we change the visibility modifier to private. This will force everyone to use only the methods we define to get at that variable. From now on, we should make all our instance variables (or fields) private.

The revised CokeMachine class

```
public class CokeMachine
{
  private int numberOfCans;

  public CokeMachine()
  {
    numberOfCans = 2;
    System.out.println("Adding another machine to your empire");
  }

  public void buyCoke()
  {
    if (numberOfCans > 0)
    {
      numberOfCans = numberOfCans - 1;
      System.out.println("Have a Coke");
      System.out.print(numberOfCans);
      System.out.println(" cans remaining");
    }
    else
    {
      System.out.println("Sold Out");
    }
  }
}
```

Parameters

type

parameter name

```
public int getNumberOfCans()
{
  return numberOfCans;
}

public void reloadMachine(int loadedCans)
{
  numberOfCans = loadedCans;
}
```

To pass a value to a method, we have to put the data type of the value and the name by which the method will refer to the value in the parameter list.

Parameters

```
public int getNumberOfCans()
{
  return numberOfCans;
}

public void reloadMachine(int cansOfCoke, int cansOfDietCoke)
{
  bin1 = cansOfCoke;
  bin2 = cansOfDietCoke;
}
```

Multiple parameters are separated by commas in the parameter list.

(And this example assumes that we've had the foresight to declare variables called bin1 and bin2 somewhere. Of course.)

Scope of a variable

The scope of a variable (or constant) is that part of a program in which the value of that variable can be accessed. As noted previously, final variables should be defined as public, and their scope would then extend through the entire program. That’s ok, because the lazy sloppy programmers can’t change the value of a final variable.

Scope of a variable

```
public class CokeMachine4
{
  private int numberOfCans;

  public CokeMachine4()
  {
    numberOfCans = 2;
    System.out.println("Adding another machine to your empire");
  }

  public int getNumberOfCans()
  {
    return numberOfCans;
  }

  public void reloadMachine(int loadedCans)
  {
    numberOfCans = loadedCans;
  }
```

The numberOfCans variable is an instance variable declared inside the class but not inside a particular method, so its scope is the entire class. That is, it can be accessed by any method in the class.

Scope of a variable

```
public class CokeMachine4
{
  private int numberOfCans;

  public CokeMachine4()
  {
    numberOfCans = 2;
    System.out.println("Adding another machine to your empire");
  }

  public double getVolumeOfCoke()
  {
    double totalLitres = numberOfCans * 0.355;
    return totalLitres;
  }

  public void reloadMachine(int loadedCans)
  {
    numberOfCans = loadedCans;
  }
```

On the other hand, if we declare a variable within a method, it can only be accessed from within the method. Its scope is just the method. This is a local variable; it has local scope. (Note also that variables defined in methods don't have visibility modifiers.)

Scope of a variable

```
public class CokeMachine4
{
  private int numberOfCans;

  public CokeMachine4()
  {
    numberOfCans = 2;
    System.out.println("Adding another machine to your empire");
  }

  public int getNumberOfCans()
  {
    return numberOfCans;
  }

  public void reloadMachine(int loadedCans)
  {
    numberOfCans = loadedCans;
  }
```

The scope of a parameter named in the parameter list of a method is just like that of a variable declared within that method -- the parameter is also local data. It can be accessed only within that method.