What does it mean to program with Java?

Java is an object-oriented programming language. This means that computation is achieved via **interacting objects**. Objects interact by invoking operations on one another. Operations effect local state changes on objects, hopefully leading to some useful computation, the results of which are typically presented as output to some device.

The operations associated with a particular object must be well defined via the object's **interface**. The interface effectively defines a *contract* between the user of the object (i.e., some other object in the program) and the implementer of the object (i.e., the programmer). So, there are two perspectives to every object in an object-oriented programmer: The "user view" and the "implementer view."

The interface, or contract, for a particular category of objects is defined using the **class** syntactic construct. A Java source program consists entirely of a set of classes that completely specify a set of object interfaces and their implementation at compile-time. A Java run-time program consists of a set of compiled bytecodes representing dynamic instantiations of those classes in the form of run-time objects that execute some computation when the bytecodes are interpreted by a Java virtual machine.

The task of a Java programmer is to define new classes in terms of existing classes and builtin types. This consists of writing **class definitions**, which define new types of objects, the specification of variables related to the implementation of the public methods that provide the interface for an object, and possibly other variables and methods that are internal (private) to the implementation.

So, in Java, a program consists of lots of different class definitions. Some have to be written from scratch, others are simply reused by importing their definitions from pre-defined **packages** of class definitions (i.e., libraries). Designing a computation as a collection of compile-time class definitions each representing a category of objects that interact at run-time to perform computation is called object-oriented programming.

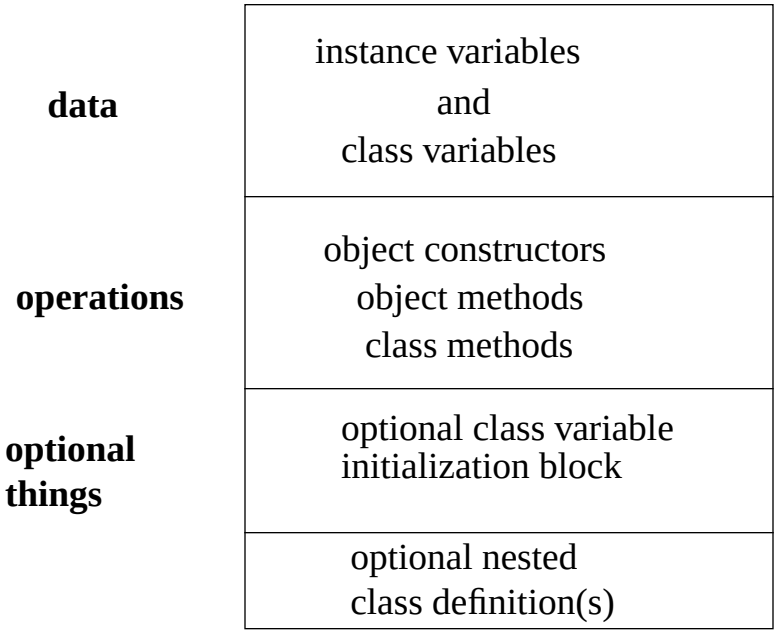
How do we program in Java?

We take the “user view” and study existing packages of class definitions to learn how to interact with pre-defined categories of objects. We also take the “implementor view” and write new packages of class definitions that define new types of objects and specifies how they interact with other objects.

A **Java program** is then constructed by instantiating objects and causing them to invoke operations on one another to effect a useful computation.

Since classes are the key to understanding pre-defined object categories and for writing new ones, we have to master the structure, syntax, and object-oriented programming conventions used by Java classes.

Java class structure



A typical Java class schemata

```
package package-name;
public class  class-name {
    // instance variable declarations, with optional inline initialization
    private type-name variable-name;
    private type-name variable-name = initialization-expr;

    // class variable declaration of a constant with inline initialization
    private final static type-name variable-name = initialization-expr;

    // object constructors
    public class-name () { stmt_1; ...; stmt_n  }
    public class-name ( formal-parameters ) { stmt_1; ...; stmt_n; }

    // object method definition
    public return-type method-name ( formal-parameters ) {
      stmt_1; ...; stmt_n;
    }
    // class method definition
    public static return-type method-name ( formal-parameters ) {
      stmt_1; ...; stmt_n;
    }
}
```

Type-names and return-types are taken from the set of builtin types, and previously defined class-names. A special return-type is **void** (the “un-type”). Formal parameters are a (possibly empty) comma-separated-list of name pairs: *type-name variable-name.*

A fragment of a typical class definition

```
public class String {
    // private instance variables, each object has its own copy
    private char value[];
    private int offset = 0;
    private int count;

    // private class variable, shared by all String objects
    private static final long serialVersionUID = -6849794470754667710L;

    // constructors
    public String() { value = new char[0]; }

    public String(char value[]) {
        this.count = value.length;
        this.value = new char[count];
        System.arraycopy(value, 0, this.value, 0, count);
     }

    // public object method
    public int length() { return count; }

    // public class method
    public static String valueOf(int i) { return Integer.toString(i, 10); }
    ...
}
```

Use of ‘this’ in methods

```
public class StringBuffer {
    private char value[];
    private int count;
    private boolean shared;

    public StringBuffer() { this(16); }
    public StringBuffer(int length) {
        this.value = new char[length];
        this.shared = false;
    }
    public StringBuffer(String str) {
        this(str.length() + 16);
        append(str);
    }
    public StringBuffer append(String str) {
        if (str == null) str = String.valueOf(str);
        int len = str.length();
        ensureCapacity(this.count + len);
        copyWhenShared();
        str.getChars(0, len, this.value, this.count);
        this.count += len;
        return this;
    }
}
```

NOTE: this is not defined in a class method (i.e., a method declared static)

Reference variables and object instantions

Objects are used in Java by declaring and using reference variables that are initialized using constructors. Constructors are special class methods that do not return any value.

```
class-name variable-name;
class-name variable-name = null;
class-name variable-name = new class-name ( optional-argument-list );
class-name variable-name = initialization-expression;
```

Examples:

```
String a;              // uninitialized String reference variable
String b = null;       // explicitly setting a null object reference
String c = new String();  // the “empty” string object
String d = new String (“This is a string literal”);
String e = s;  // initialize e using String object s;
a = e;    // initialize a to refer to the same String object as e
```

Strings have a special kind of initialization expression, involving string literals.

```
String f = “this is a string literal”;
String g = “Hello” + “, “ + “world”;
```

These are equivalent to the initialization expressions:

```
String f = new String (“this is a string literal”);
String g = new String (“Hello, world”);
```

Object and class method invocations

Once an object of a class has been instantiated and a reference variable to that object is initialized, the object is typically used in some expression as the target for some kind of method invocation using the reference variable.

```
reference-var . method-name ( optional-argument-list );
```

For example:

```
String hello = new String("Hello, world");
int x = hello.length();
```

Class methods, which are globally accessible operations defined on classes not on objects, are invoked similarly, but using a class name instead of a reference variable.

```
class-name . class-method-name ( optional-arguments );
```

For example, an initialization expression can invoke a method that constructs an object of the appropriate type to perform the initialization.

```
String five = String.valueOf(5);  // String class method to "stringify" an int
```

The `String.valueOf(int)` method is a class method that constructs a new String object by converting an integer value to a String value, assuming a base 10 radix.

Some special class definition forms

```
public class  class-name {

    // publicly accessible class variables
    public static type-name variable-name = initialization-expr;
    public static type-name variable-name;

    // a class variable initialization block
    static {
      stmt_1; ...; stmt_n;
    }

    // special "main" class method used as a program entry point
    public static void main (String[] args) {
      stmt_1; ...; stmt_n;
    }
}
```

Class variables are very often declared public and are referenced using an expression of the form:

class-name . *class-variable-name*

For example, the class java.lang.Math class defines a public class variable representing a 64-bit approximation of π defined as `PI = 3.14159265358979323846` and used as follows:

```
    double area = Math.PI * radius * radius;
```

Using class variables and methods

Class variables and methods can be used before any objects are ever instantiated in a Java program. The keyword **static** is used to denote class variables and methods because they are considered to be “statically” instantiated attributes of the class, in contrast to objects, which are all dynamically instantiated.

```
class Hello {
    public static String username = null;

    static {
      try {
        username = System.getProperty(“user.name”);
      } catch (Exception e) { System.err.println(e); }
    }

    public static void main (String[] args) {
      System.out.println(“Username is: “ + username);
    }
}
```

A static initialization block us typically used to initialize class variables that require more than a simple inline initialization expression, or simply to reduce “declaration clutter” and collect together all the static initializations in one place. The special class method **main** is used as the starting point for a program. It is possible for multiple classes in the same Java program to have distinct main methods. The main method called to start a Java program is determined by which class the Java VM is asked to load first.

More special class definition forms

```
public final class  class-name {

    // public, globally accessible constant declaration
    public final static type-name variable-name = initialization-expr;

    // a "final" object method, which cannot be overridden by a subclass
    public final return-type method-name ( formal-parameters ) {
      stmt_1; ...; stmt_n;
    }

    // a "native" object method, which has an external implementation
    public native return-type method-name ( formal-parameters );
    public final native return-type method-name ( formal-parameters );
}
```

A final class cannot be extended. That is to say, a new class cannot be defined as an extension of a final class.

A final static variable is the same thing as a constant. A final method is one that cannot be overridden by a subclass. In a final class, all methods are implicitly final. In a non-final class, methods can be selectively made final. Only object methods can be final. Static methods are implicitly final.

A native method consists of just a method declaration with no method body. The method implementation is typically written in C/C++ and is linked into the Java run-time, usually for efficiency reasons.

Example usage of “final” and “native” qualifiers

```
public final class Math {

    public static final double E = 2.71828182845904523536;
    public static final double PI = 3.14159265358979323846;

    public static native double sin (double a);
    public static native double cos (double a);
    public static native double tan (double a);
    ...
    public static native double exp(double a);
    public static native double log(double a);
    public static native double sqrt(double a);
    public static native double pow(double a, double b);
      ...
    public static int abs(int a) { return (a < 0) ? -a : a; }
    public static long abs(long a) { return (a < 0) ? -a : a; }
    public static float abs(float a) { return (a < 0) ? -a : a; }
    public static double abs(double a) { return (a < 0) ? -a : a; }
    ...
}
```

Note that it is necessary to **overload** the absolute value method for each the builtin numeric types. In Java, you can define many methods with the same name, as long as the type signatures are unique.

Declaration of methods that throw exceptions

```
public class  class-name {
    ...
    // a “final” object method, which cannot be overridden by a subclass
    public return-type method-name ( formal-parameters )
    throws exception-type-list
    {
      stmt_1; ...; stmt_n;
    }
};
```

An exception-type-list is a comma-separated list of previously defined exception types (i.e., class names). In general, you should invoke any method that will throw an exception inside the context of a **try block**, but not necessarily in the same scope level. A catch block is defined for each specific exception type that you want to catch. An optional finally block can be defined to contain a set of statements that will be excuted whether or not the exception occurs.

```
try {
    ...; variable-name.method-name (arguments); ...;
}
catch (exception-type1 e1) { stmt_1; ...; stmt_n; }
catch (exception-type2 e2) { stmt_1; ...; stmt_n; }
...
finally {
    stmt_1; ...; stmt_n;
}
```

Example: a method that throws an exception

```
public final class Byte {
    public static final byte   MIN_VALUE = -128;
    public static final byte   MAX_VALUE = 127;

    public static byte parseByte(String s) throws NumberFormatException {
        return parseByte(s, 10);
    }
    public static byte parseByte(String s, int radix)
    throws NumberFormatException {
        int i = Integer.parseInt(s, radix);
        if (i < MIN_VALUE || i > MAX_VALUE)
            throw new NumberFormatException();
        return (byte)i;
    }
}

public static void main (String[] args) {
    int err_code = 0;
    try {
      byte b = Byte.parseByte(args[0]);
    }
    catch (ArrayIndexOutOfBoundsException e) { err_code = -1; }
    catch (NumberFormatException e) { err_code = -2; }
    finally { System.exit(err_code); }
}
```