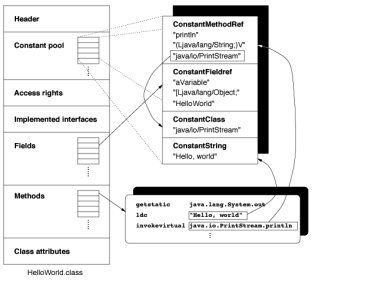
# Project 4

# Java class File Format

- Header (magic number)
- Constant pool
  - information needed to dynamically resolve symbolic references to classes, fields, and methods at run-time (coded with string constants), strings, integers, floats, longs, and doubles
- Access rights
- Implemented interfaces
- Fields
  - Contains pointers to constant pool (index)
- Methods
  - actual byte code
- Class attributes

# Java class File Format




# Java Byte Code

- Stack operations
- Arithmetic operations
- Control flow
- Load and store operations for local variables
- Field access
  - getfield, putfield
- Method invocation
  - invokestatic and invokevirtual
- Object allocation
  - New, newarray
- Conversion and type checking

# Runtime Data Areas

- The pc register
- Stack
  - holds local variables and partial results
  - Frames pushed and popped
  - Memory does not need to be contiguous
- Heap
  - Memory for all class instances and arrays
  - Garbage collected
- Method area
  - Runtime constant pool (serves a function similar to a symbol table)

# Byte Code Engineering Library (BCEL)

- ClassGen
  - Types
  - Generic fields and methods
  - Instructions
  - Instruction lists
    - Appending
    - Inserting
    - Deleting

## Example

```
import java.io.*;

public class HelloWorld {
  public static void main(String[] argv) {   BufferedReader in = new
    BufferedReader(new InputStreamReader(System.in));
   String name = null;
    try {
      System.out.print("Please enter your name> ");
      name = in.readLine();
   } catch(IOException e) { return; }

System.out.println("Hello, " + name); } }
```

## Initialization

- Create an empty class and an instruction list

```
ClassGen cg = new ClassGen("HelloWorld", "java.lang.Object", "<generated>",
    ACC_PUBLIC | ACC_SUPER, null);
ConstantPoolGen cp = cg.getConstantPool();
                        // cg creates constant pool
InstructionList il = new InstructionList();

MethodGen mg = new MethodGen(ACC_STATIC | ACC_PUBLIC, // access
    flags
                              Type.VOID, // return type
                               new Type[] { // argument types
                                 new ArrayType(Type.STRING, 1) },
                               new String[] { "argv" }, // arg names
                               "main", "HelloWorld", // method, class
                                il, cp);
```

## Your modifications

- GenProcCode – generate code for a procedure
- genInstCode – generate code for an astNode representing TREE_BLOCK, TREE_IF, or TREE_WHILE
- genExpCode – generate code for a single TREE_INSTRUCTION
- genOpCode – generate code for expNode representing an operator

## Adding New Instructions

InstructionList x;

genExpCode(x, op1); // generate instructions for 1st operand, append to x

genExpCode(x, op2);  // generate instructions for 2nd operand, append to x

x.append(new IADD()); // generate IADD instruction, append to x

## Using Labels

```
InstructionList x, y;
InstructionHandle label;

label = x.append(new NOP()); // returns
        // label for conditional branch targets
y.append(new IFEQ(label)); // create new
        // conditional branch instruction
y.append(x);
```