1 Part VII: Run-time

- Run-time representations
- Run-time systems
- Run-time representaion for Jex
- Interpreting Jex

2 What do we need to think about

- Run-time is, when the program actually executes.
- For a native compiler, this is when we run the native code.
- For an interpreter, this is when we interpret it.
- In Java, this is when the JVM runs the user program.
- How is the program represented at run-time? (e.g. Java Byte Code)
- How is data represented at run-time? (e.g. format for integers)
- What happens at run-time? (e.g. JVM, how do we interpret Java Bytecode)

This can be very complicated (e.g. in a JIT compiler). The characteristics of the running program (speed, memory consumption) will depend heavily on these decisions (Also the complexity of the compiler).

3 Run-time Representations

- Where are variables stored?
  - Do we need a stack?
  - How do we organize it?
  - Do we need a heap?
  - How do we organize it?
- How are they stored in memory?
  - How do we represent objects in an object oriented language?
  - Scanner Generator: How do we represent the finite automaton?
  - Parser Generator: How do we represent the stack automaton?
- How do we call other functions?
- How do we represent exceptions?
- How do we represent threads?

- How do we represent code?
  - Bytecode or native code?
  - Mixture of the two (JIT)?
  - Use Abstract Syntax Tree (Interpreter)?
- How is the program stored on the disc?
  - Java: Classfile format (**.class**)
  - C, C++: Object file format (`.o`)

4 Run-time System

- Typically a compiler comes with a run-time system.
- The run-time system consists some services.
- These are sometimes directly called by the user program, often they are called by compiler-generated code.
  - Memory allocation/deallocation
  - Garbage collection
  - Threads
  - Synchronization
  - Exceptions
  - Access to OS routines.

- When we do interpretation the run-time system is built into the interpreter.
- When we compile to machine code, the run-time system typically comes as a library that is linked to the object code.
- In JavaCup the generated class uses other classes from jaco.framework.parser. They are part of the run-time system of JavaCup.
- JLex just produces a Java class. It only needs the usual Java run-time support.
- In a Java compiler we compile to Java Bytecode. The run-time system is the Java Virtual Machine.

5 A Run-time System for Jex

- We have to represent values at run-time.
- In Jex we have different kind of values:
  classes, objects and basic values (int and boolean).
- For each of them we have a class in the interpreter, which are all subclasses of JexValue.
- We have methods on JexValue for testing what kind of value we have and for accessing the real values in JexValue (e.g. the integer).
- There is an analogy between JexValue and JexSymbol: JexSymbol is for the meaning of the different identifiers, JexValue is for the different things, identifiers can refer to at run-time.
- JexSymbol is a static concept (it is fixed during interpretation).
- JexValue is a dynamic concept (it changes during interpretation).

6 JexObjects

- They represent Java objects at run-time.
- They appear as results of **new** calls, or other calls.
- They appear as string constants "hello".
- They can be stored and loaded from variables.
- We need to remember the corresponding Java object.
- We use a field obj.
- We need methods for accessing the fields and calling the methods of the object.

```
class JexObject { ... }
    Object obj;
    JexValue callMethod(String s, JexValue[ ] args) { ... }
    JexValue getField(String s) { ... }
    void setField(String s, JexValue val) { ... }
    Object getObject() { ... }
}
```

7 JexBasic

- They represent values of basic types, **boolean** or **int**.
- They appear as results of calls or also the evaluation of operators.
- They appear as constants 3, false.
- They can be stored and loaded from variables.
- We need to remember the actual value, and its class.
- We need methods for extracting the actual integer or boolean.

```
class JexBasic {
    Object val;
    Class cls;
    int getInteger() { ... }
    boolean getBoolean() { ... }
}
```


8 JexClass

- They represent Java classes at run-time.
- They appear if we make a reference to a class name like in String s; or System.out.
- The only thing we need to remember about them is the Java class.
- Therefore we have only one field cls.
- We need methods for accessing the static fields and calling static methods and for creating new Objects of the class.

```
class JexClass {
    Class cls;
    JexValue callMethod(String s, JexValue[ ] args) { ... }
    JexValue getField(String s) { ... }
    void setField(String s, JexValue val) { ... }
    JexValue getNew(JexValue[ ] args) { ... }
}
```


9 JexValue

- We define one superclass JexValue for all of these values.
- Then we are able to pass them around in the interpreter uniformly.
- This class has test-functions, which tell us what it is (e.g. isInteger, isClass)
- It also has all of the functions mentioned above. (e.g. getInteger, getField)
- If we apply them to a JexValue, which doesn't support this method (e.g. getInteger to a JexClass), it will raise an exception).

```
class JexValue {
    boolean isInteger() { ... }
    int getInteger() { ... }
}
```


10 JexValue (2)

- We could define an abstract method **int** getInteger().
- For JexBasic we would return the integer value (if it is an integer).
- For all other cases, we would implement it as an error.
- As an alternative we can implement it for JexValue as an error message and redefine it for JexBasic.
- We save redefining it for every other JexValue.

```
class JexValue {
    boolean isInteger() { return false; }
    int getInteger() { throw new Error("internal error"); }

    class JexBasic {
        boolean isInteger() { ... }
        int getInteger() { ... }
    }
}
```