JOOS

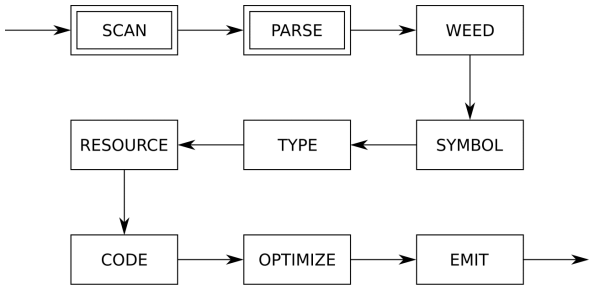
COMP 520: Compiler Design (4 credits)

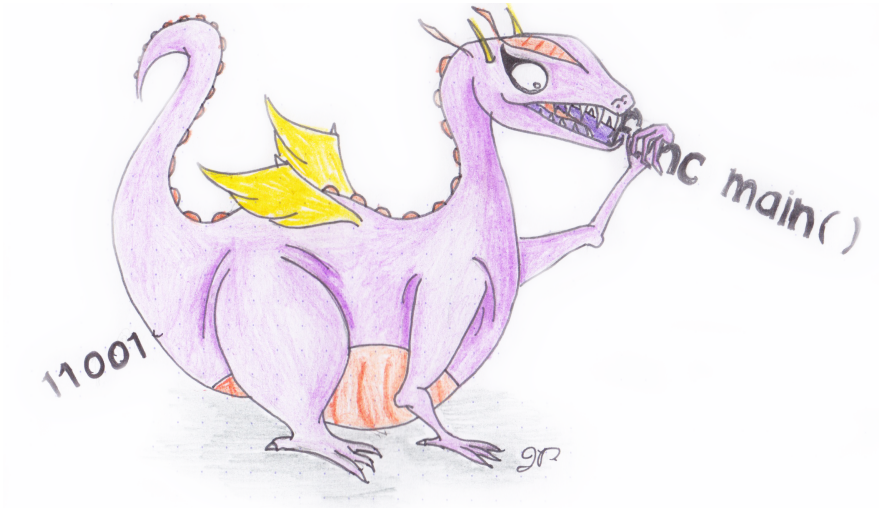
Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 8:30-9:30, TR 1080

http://www.cs.mcgill.ca/~cs520/2019/





Java Language

Overview

The Java programming language was

- Originally called Oak;
- Developed as a small, clean, OO language for programming consumer devices; and
- Used as the implementation language for (many) large applications.

Basic compilation (`.java` → `.class`)

- Java programs are developed as source code for a collection of Java classes;
- Each class is compiled into Java Virtual Machine (JVM) bytecode; and
- Bytecode is interpreted or JIT-compiled using some implementation of the JVM.

Java Language

Advantages of Java

- Object-oriented;
- A “cleaner” OO language than C++;
- Portable (except for native code);
- Distributed and multithreaded;
- “Secure”;
- Semantics are completely standardized;
- Huge standard libraries; and
- Officially open source.

Major Drawbacks of Java

- Missing many language features, e.g. genericity (until 1.5), multiple inheritance, operator overloading;
- There is no *single* standard (`JDK 1.0.2` vs. `JDK 1.1.*` vs. `...`);
- Slower than C++ for expensive numeric computations due to dynamic array-bounds checks; and
- It’s not JOOS.

Java Security

Given the number of security updates and threats, you might not think of Java as an especially secure language. However, the language itself does have some secure features.

- Programs are strongly type-checked at compile-time;
- Array bounds are checked at run-time;
- `null` pointers are checked at run-time;
- There are no explicit pointers;
- Dynamic linking is checked at run-time; and
- Class files are verified at load-time.

JOOS Language

The JOOS subset of Java was designed with the following goals in mind

- Extract the object-oriented essence of Java;
- Make the language small enough for course work, yet large enough to be interesting;
- Provide a mechanism to link to existing Java code; and
- Ensure that every JOOS program is a valid Java program, such that JOOS is a strict subset of Java.

Programming in JOOS

Like with Java, a JOOS program consists of a collection of classes. An ordinary class consists of

- Protected fields;
- Constructors; and
- Public methods.

Cons.java

Recursive definition of a list – think COMP 302

```
public class Cons {
   protected Object first;
   protected Cons rest;

   public Cons(Object f, Cons r) {
      super();
      first = f;
      rest = r;
   }

   public void setFirst(Object newfirst) {
      first = newfirst;
   }

   public Object getFirst() {
      return first;
   }

   public Cons getRest() {
      return rest;
   }

   public boolean member(Object item) {
      if (first.equals(item))
         return true;
      else if (rest == null)
         return false;
      else
         return rest.member(item);
   }

   public String toString() {
      if (rest == null)
         return first.toString();
      else
         return first + " " + rest;
   }
}
```

Programming in JOOS

As seen in the `Cons.java` example

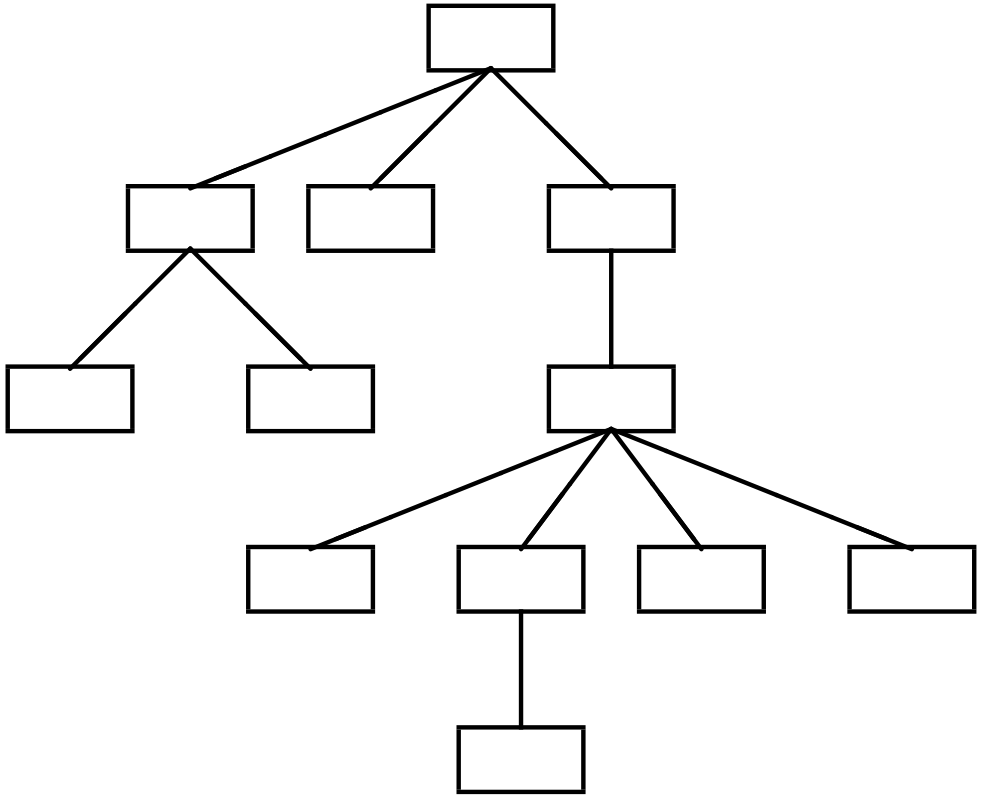
- Fields must be *protected*: they can only be accessed via objects of the class or its subclasses;
- Constructors must start by invoking a constructor of the superclass (`super(..)`);
- Methods must be *public*: they can be invoked by any object; and
- Only constructors can be overloaded, other methods cannot.

Other important notes

- Subclassing must not change the signature of a method;
- Local declarations must come at the beginning of the statement sequence in a block; and
- Every path through a non-void method must return a value (in Java such methods can also throw exceptions).

Class Hierarchies

The class hierarchies in JOOS and Java are both single inheritance, i.e. each class has exactly one superclass, except for the root class



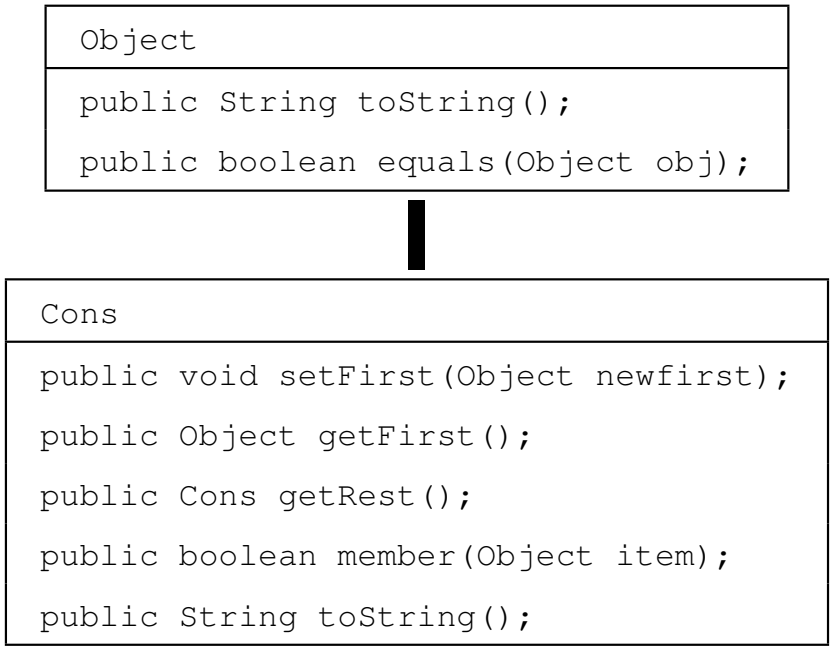
The root class is called `Object`, and any class without an explicit `extends` clause is a subclass of `Object`.

Class Hierarchies - Example

The definition of the `Cons` class is equivalent to

```
public class Cons extends Object {
      ...
}
```

which gives the class hierarchy



Types in JOOS

Primitive types

- `boolean`: `true` and `false`;
- `int`: $-\mathbf{2^{31}} \ldots \mathbf{2^{31} - 1}$;
- `char`: the ASCII characters;

User-defined class types

Externally defined class types

- `Object`;
- `Boolean`;
- `Integer`;
- `Character`;
- `String`;
- `BitSet`;
- `Vector`;
- `Date`.

Note that `boolean` and `Boolean` are different.

Types in Java and JOOS

- Java is strongly-typed;
- Java uses the name of a class as its type;
- Given a type of class `C`, any instance of class `C` or a subclass of `C` is a permitted value;
- "Down-casting" is automatically checked at run-time:

```
SubObject subobj = (SubObject)obj;
```

- There are explicit `instanceof` checks; and

```
if (subobj instanceof Object)
   return true;
else
   return false;
```

- Some type-checking must be done at run-time.

Expressions in JOOS

An expression is a computation which evaluates to a value

- Constant expressions

```
true, 13, '\n', "abc", null
```

- Variable expressions

```
i, first, rest
```

- Binary operators

```
||
&&
!= ==
< > <= >= instanceof
+ -
* / %
```

- Unary operators

```
-
!
```

- Class instance creation

```
new Cons("abc",null)
```

- Cast expressions

```
(String)getFirst(list)
(char)119
```

- Method invocation

```
l.getFirst()
super.getFirst();
l.getFirst().getFirst();
this.getFirst();
```

Statements in JOOS

A statement is an action that has no associated value (i.e. structures, controls, etc)

- Expression statements

```
x = y + z;
x = y = z;
a.toString(l);
new Cons("abc", null);
```

- Block statements

```
{
   int x;
   x = 3;
}
```

- Return statements

```
return;
return true;
```

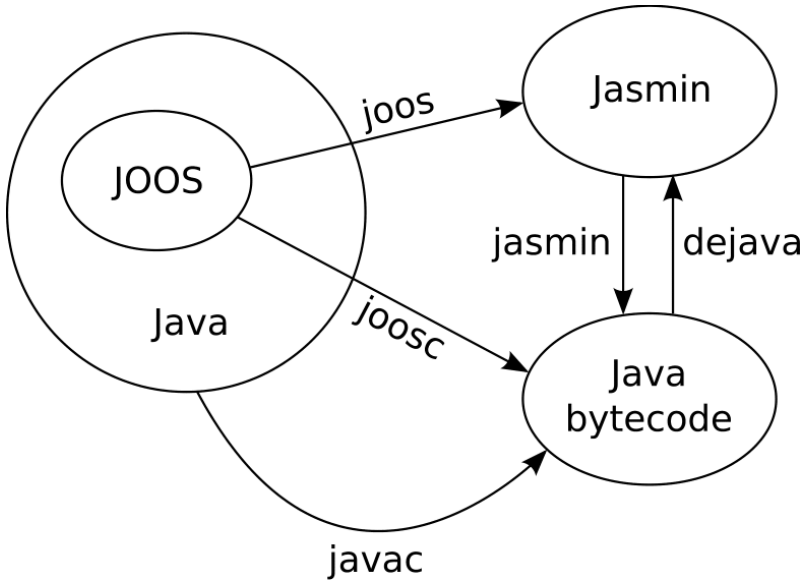
- Control structures

```
if (l.member("z")) {
   // do something
}

while (l != null) {
   l = l.getRest();
}
```

JOOS Representations

Converting between JOOS & Java source code (`*.java`, `*.joos`), Jasmin assembler (`*.j`) and Java bytecode (`*.class`)



`joosc` simply calls `joos` and then `jasmin`.

JOOS AST Nodes

JOOS follows the same idea of 1 AST node per programming language construct

PROGRAM	CLASSFILE	CLASS
FIELD	TYPE	LOCAL
CONSTRUCTOR	METHOD	FORMAL
STATEMENT	EXP	RECEIVER
ARGUMENT	LABEL	CODE

Each node consists of the child nodes, resources, and code.

```
typedef struct METHOD {
   int lineno;
   char *name;
   ModifierKind modifier;
   int localslimit; /* resource */
   int labelcount; /* resource */
   struct TYPE *returntype;
   struct FORMAL *formals;
   struct STATEMENT *statements;
   char *signature; /* code */
   struct LABEL *labels; /* code */
   struct CODE *opcodes; /* code */
   struct METHOD *next;
} METHOD;
```

JOOS Constructors

And each AST node kind has an associated constructor function for ease of use.

```
METHOD *makeMETHOD(char *name, ModifierKind modifier, TYPE *returntype,
                   FORMAL *formals, STATEMENT *statements, METHOD *next)
{
   METHOD *m = malloc(sizeof(METHOD));
   m->lineno = lineno;
   m->name = name;
   m->modifier = modifier;
   m->returntype = returntype;
   m->formals = formals;
   m->statements = statements;
   m->next = next;
   return m;
}

STATEMENT *makeSTATEMENTwhile(EXP *condition, STATEMENT *body)
{
   STATEMENT *s = malloc(sizeof(STATEMENT));
   s->lineno = lineno;
   s->kind = whileK;
   s->val.whileS.condition = condition;
   s->val.whileS.body = body;
   return s;
}
```

JOOS Scanner

```
[ \t]+          /* ignore */;
\n              lineno++;
\/\/[^\n]*      /* ignore */;
abstract        return tABSTRACT;
boolean         return tBOOLEAN;
break           return tBREAK;
byte            return tBYTE;

[...]

"!="            return tNEQ;
"&&"            return tAND;
"||"            return tOR;
"+"             return '+';
"-"             return '-';

[...]
```

JOOS Scanner

```
0|([1-9][0-9]*) {
    yylval.intconst = atoi(yytext);
    return tINTCONST;
}

true {
    yylval.boolconst = 1;
    return tBOOLCONST;
}

false {
    yylval.boolconst = 0;
    return tBOOLCONST;
}

\"([^\"])*\" {
    yylval.stringconst = (char*)malloc(strlen(yytext)-1);
    yytext[strlen(yytext)-1] = '\0';
    sprintf(yylval.stringconst,"%s",yytext+1);
    return tSTRINGCONST;
}
```

JOOS Parser

```
method : tPUBLIC methodmods returntype tIDENTIFIER '(' formals ')' '{' statements '}'
            {$$ = makeMETHOD($4,$2,$3,$6,$9,NULL);}

       | tPUBLIC returntype tIDENTIFIER '(' formals ')' '{' statements '}'
            {$$ = makeMETHOD($3,modNONE,$3,$5,$8,NULL);}

       | tPUBLIC tABSTRACT returntype tIDENTIFIER '(' formals ')' ';'
            {$$ = makeMETHOD($4,modABSTRACT,$3,$6,NULL,NULL);}

       | tPUBLIC tSTATIC tVOID tMAIN '(' mainargv ')' '{' statements '}'
            {$$ = makeMETHOD("main",modSTATIC,makeTYPEvoid(),NULL,$9,NULL);}
;

whilestatement : tWHILE '(' expression ')' statement
            {$$ = makeSTATEMENTwhile($3,$5);}
;
```

Notice the conversion from concrete syntax to abstract syntax that involves dropping unnecessary tokens.