COMP 520 - Compilers

Lecture 16 (April 19, 2022)

Runtime organization of object oriented languages

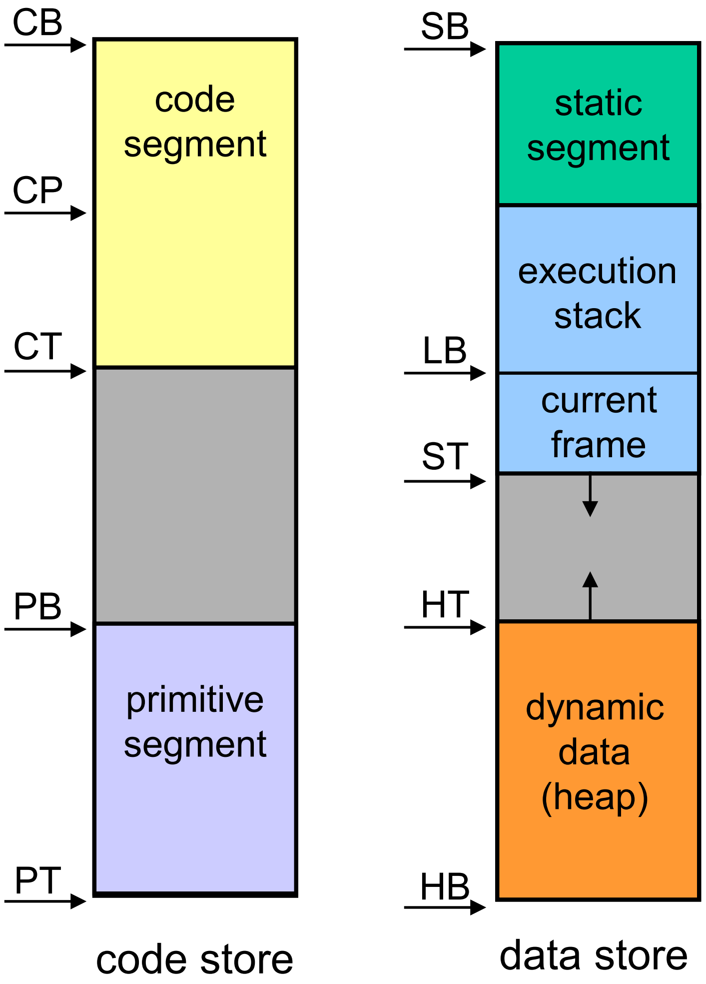
- Reading for today
  - PLPJ Chapter 6: secn 6.7
  - Also need to know: code generation, chapter 7

Today's topics

- Review of miniJava classes without inheritance
  - mJAM representation of objects
  - layout of mJAM memory
- mJAM support for classes with single inheritance
  - representation
  - mJAM support
- Related issues

mJAM memory organization

- Two separate memories
  - Code store
    - compiler-generated program is loaded into code segment
    - predefined runtime functions are located in the primitive segment
    - mJAM can not write into code store
  - Data store
    - static constants and variables are loaded into static segment
    - method invocation creates a frame
    - expression evaluation occurs at stack top
      - expands downwards
    - object instances are dynamically allocated on the heap
      - expands upwards
      - (no garbage collection)
- ABI defines fixed addresses and usage conventions
  - various locations in memories are accessed relative to machine registers (CB, SB, LB, ST, etc.)



miniJava: simple classes, no inheritance

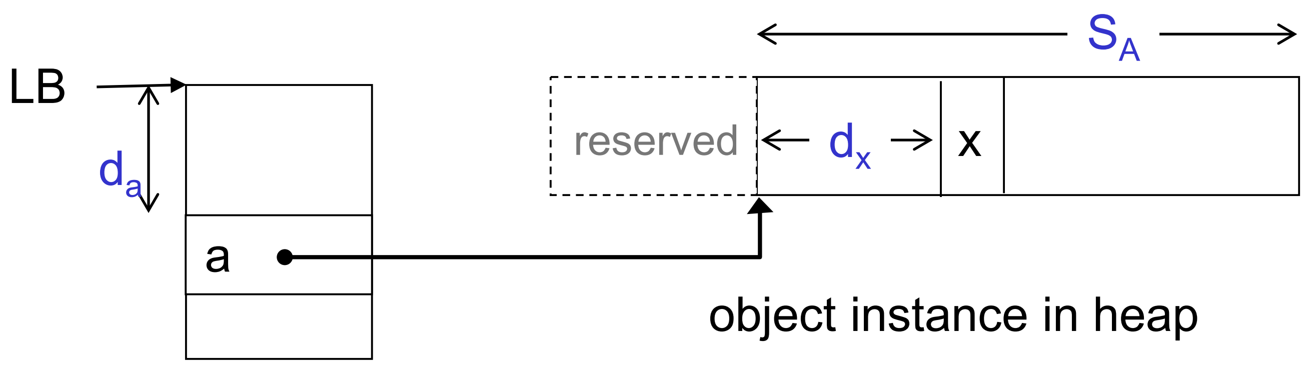
- Classes

  ```
  class A { int x; void p(){x = x + 3;} }
  ```

  – runtime entity descriptions in AST
    - class A : S_A = size of class A (# fields) = 1
    - field x: d_x = displacement of field x = 0
    - method p: d_p = displacement of code for p = ?

- Objects
  – objects are created on the heap: `A a = new A();`
  – let d_a be displacement of local var "a" in activation record

mJAM runtime layout



object instance in heap

activation record on stack

mJAM – adapted from TAM (text appx C)

- Instructions

Table C.2 Summary of TAM instructions.

Op-code	Instruction mnemonic	Effect
0	LOAD(*n*) *d*[*r*]	Fetch an *n*-word object from the data address (*d* + register *r*), and push it on to the stack.
1	LOADA *d*[*r*]	Push the data address (*d* + register *r*) on to the stack.
2	LOADI(*n*)	Pop a data address from the stack, fetch an *n*-word object from that address, and push it on to the stack.
3	LOADL *d*	Push the 1-word literal value *d* on to the stack.
4	STORE(*n*) *d*[*r*]	Pop an *n*-word object from the stack, and store it at the data address (*d* + register *r*).
5	STOREI(*n*)	Pop an address from the stack, then pop an *n*-word object from the stack and store it at that address.
6	CALL(*n*) *d*[*r*]	Call the routine at code address (*d* + register *r*), ~~using the address in register *n* as the static link.~~
7	CALLI	Pop a closure (static link and code address) from the stack, then call the routine at that code address.
8	RETURN(*n*) *d*	Return from the current routine: pop an *n*-word result from the stack, then pop the topmost frame, then pop *d* words of arguments, then push the result back on to the stack.
9	–	(unused)
10	PUSH *d*	Push *d* words (uninitialized) on to the stack.
11	POP(*n*) *d*	Pop an *n*-word result from the stack, then pop *d* more words, then push the result back on to the stack.
12	JUMP *d*[*r*]	Jump to code address (*d* + register *r*).
13	JUMPI	Pop a code address from the stack, then jump to that address.
14	JUMPIF(*n*) *d*[*r*]	Pop a 1-word value from the stack, then jump to code address (*d* + register *r*) if and only if that value equals *n*.
15	HALT	Stop execution of the program.

- *a* denotes a data address
- *c* denotes a character
- *i* denotes an integer
- *n* denotes a non-negative integer
- *t* denotes a truth value (0 for *false* or 1 for *true*)
- *v* denotes a value of any type
- *w* denotes any 1-word value

call static method at d[CB]

call instance method at d[CB], instance code addr at stacktop

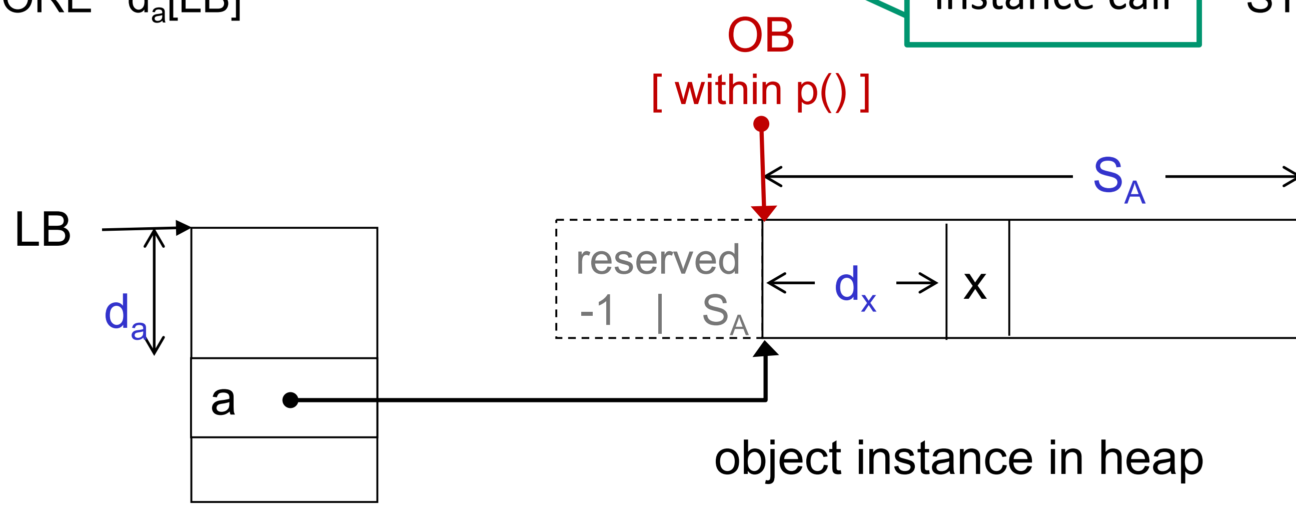
mJAM: runtime support for simple classes

- mJAM code sequences

`A a = new A();` (object creation)	`a.x;` (qualified reference)	`a.p();` (method invocation)	`x = x + 3;` (field upd within p())
LOADL -1	LOAD d_a[LB]	LOAD d_a[LB]	LOAD d_x[OB]
LOADL S_A	LOADL d_x	CALLI d_p[CB]	LOADL 3
CALL newobj	CALL fieldref		CALL ADD
STORE d_a[LB]			STORE d_x[OB]

instance address

instance call



activation record on stack

Another example

```
class T {
        public int x;

        public T get_this() {
                x = 3;
                return this;
        }
}
```

what code should be generated?

```
class Mainclass {

public static void main(String [] args) {
                T t = new T();
                T s = t.get_this();
                System.out.println(s.x);
        }
}
```

PA4 / PA5

- PA4 functionality is the goal for your miniJava compiler project
  - PA4 due Thu 4/21
  - test results will be available a few days later
  - You can make changes and resubmit a final version

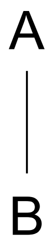
- PA5 adds *optional* extensions
  - PA5 will be distributed 4/21 and due Tue 4/26 (last class)
  - list of additional options and point values will be described
  - You can choose to implement one or more option(s) or simply go with PA4 functionality.

Classes with single inheritance (Java)

- Class hierarchy

```
class A {int x; void p(){ … } }
class B extends A {int y; void p(){ … } void q(){ … } }
```

  - inheritance hierarchy
    - "class B extends class A", or "B is a subtype of A"



  - fields
    - fields of B extend the fields of A
    - runtime layout of fields in A is a prefix of the runtime layout of fields in B

  - methods
    - methods of B extend the methods of A
    - methods of B can redefine (override) methods of A

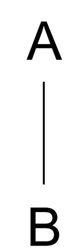
Static and dynamic type with single inheritance

- Object type
  - static type (declared type)
    - used by compiler for type checking
      - determines accessible fields and available methods on objects
      - type rules for assignments
        - » assignment: (type of RHS) must be a subtype (≤) of (type of LHS)
        - » method call: type of arg *i* must be a subtype of type of parameter *i*
  - dynamic type (run-time type)
    - generally only known at runtime
      - *part of the representation of an object*
        - » initialized at time of creation from object constructor
      - dynamic type is always a subtype of the static type (guaranteed by type system)
      - dynamic type determines which method is invoked (runtime lookup)
  - examples

```
A a = new A();
B b = new B();
A c = b;
B d = a;
a.p();
b.q();
c.p();
```

```
class A {int x; void p(){ … } }

class B extends A {
    int y;
    void p(){ … }
    void q(){ … }
}
```



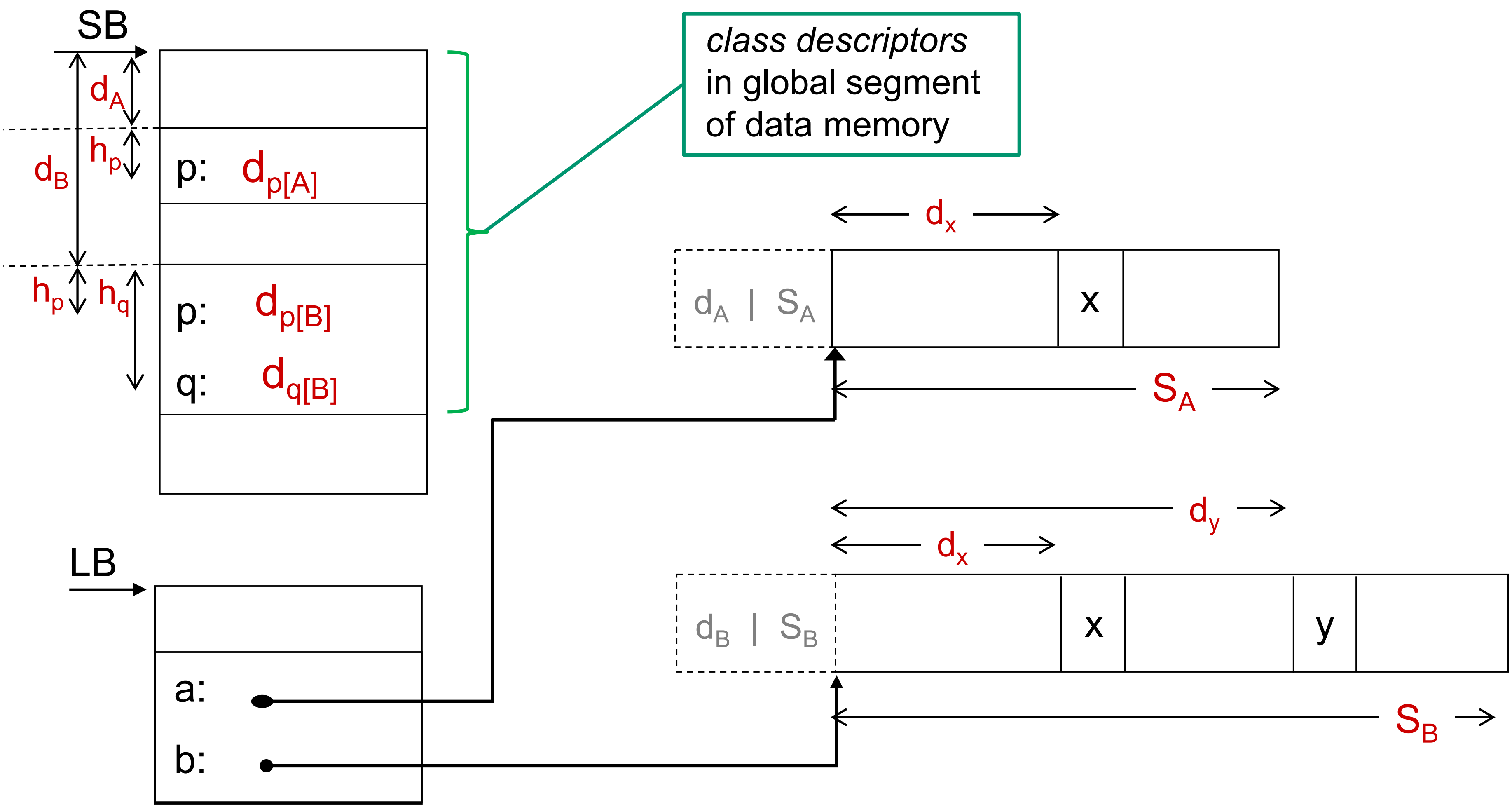
mJAM representation of single inheritance

```
class A {int x; void p(){ … } }
class B extends A
        {int y; void p(){ … } void q(){ … } }
```

- runtime entity descriptions in AST
  - class A : S_A = size of class A
  - class A: d_A = displacement of class descriptor for A
  - class B: S_B = size of class B (including size of class A)
  - class B: d_B = displacement of class descriptor for B
  - field x d_x = displacement of field x in A and B
  - field y d_y = displacement of field y in B
  - method p: h_p = index of method p in A and B
  - method q: h_q = index of method q in B
  - method p in A: $d_{p[A]}$ = displacement of code for p() in A
  - method p in B: $d_{p[B]}$ = displacement of code for p() in B
  - method q in B: $d_{q[B]}$ = displacement of code for q() in B

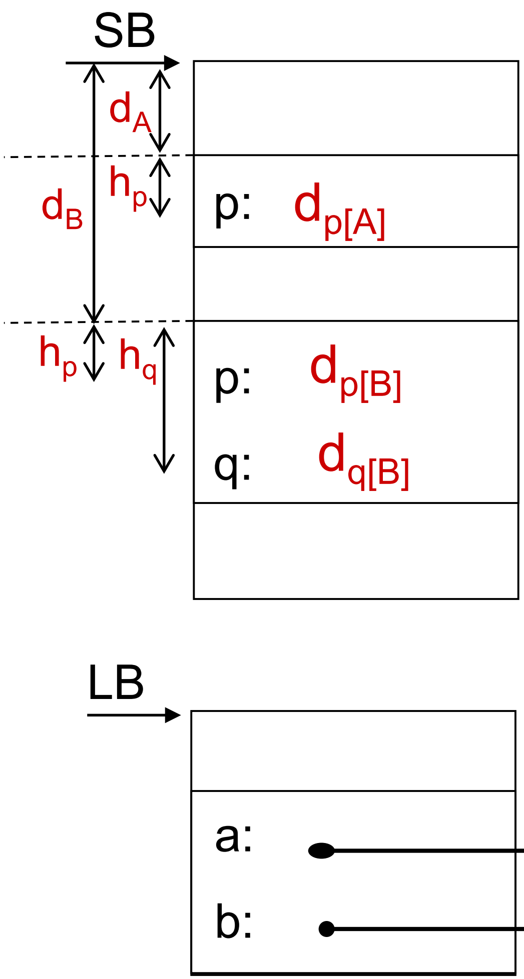
Classes with single inheritance

- mJAM runtime layout



Classes with single inheritance

- mJAM code sequences (only changed sequences are shown)



`A a = new A();`

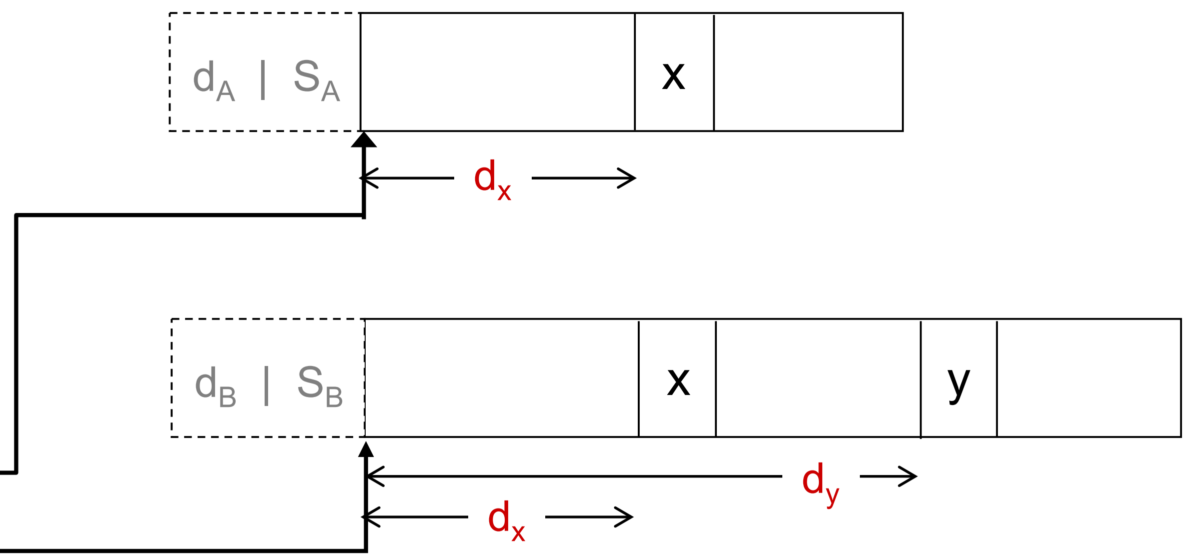
(object creation)

```
LOADL   dA
LOADL   SA
CALL    newobj
STORE   da[LB]
```

`a.p();`

(dynamic invocation)

```
LOAD    da[LB]
CALLD   hp
```



Related issues

- single inheritance
  - type operations
    - instanceof
    - casting
  - `super()` superclass constructor invocation

- multiple inheritance
  - we lose the prefix property of runtime layout!
  - not supported as such in Java, instead provides “interfaces”

- optimization
  - dynamic method dispatch has high cost
  - converting dynamic to static calls

- dynamically loaded classes
  - Java loads classes on demand, hence cannot use simple representations such as those used by mJAM

Interfaces and classes

- interface
  - specifies methods (name, signature) required of an implementation

```
interface List {
    …
    add(Object x);
    …
}
```

  - is a type (can be used in type declarations)

```
List a = new ArrayList();
```

- class
  - *implements* one or more interfaces
  - provides method bodies

```
class ArrayList implements List
{
    …
    add(Object x) { … }
    …
}
```

  - is a type

```
Arraylist a = b;
```

  - has a constructor

```
Arraylist a = new ArrayList();
```

interface vs inheritance

- inheritance
  - extends a single super-class
    - fields and methods are extended or overridden
  - requires compile time and run-time support

- interface
  - an interface can extend one or more interfaces
    - it just adds additional requirements, there is no implementation
  - requires only compile-time support

- a class
  - can **`implement`** many interfaces
  - can only **`extend`** (inherit) one other class
    - when a class extends a superclass, it inherits an implementation
    - inherited methods can be overridden

static vs. dynamic types

- Variables and expressions have a static (compile-time) type
  - derived from declarations
  - applicability defined by scope rules
  - known at compile time, without running the program
  - does not change

- Every *object* has a dynamic (run-time) type
  - obtained when the object is created using `new`
  - dynamic type can be any subtype of the static type
  - dynamic type can depend on inputs and is undecidable, in general

run-time dispatching of overridden methods

- required for objects
  - when dynamic type specifies an overridden method

- not needed for interfaces
  - interfaces cannot be instantiated (with new)
  - so static type is always equal to dynamic type
  - and compiler can work out correct method to invoke at compile time

The PA4 checkpoint (4/21)

- your pa4 directory should have
  - miniJava package
    - Compiler.java
    - SyntacticAnalyzer
    - AbstractSyntaxTrees
    - ContextualAnalyzer
    - CodeGenerator (new subpackage)
  - mJAM package (supplied on our web page)
    - Interpreter.java
    - Disassembler.java
    - Instruction.java
    - Machine.java
    - ObjectFile.java
- mJAM is needed only to check everything is working
  - pa4 testing will not copy your mJAM, it uses the mJAM as distributed
- pa4 readiness check will be available: `/check/pa4.pl`

Compiling and running miniJava programs (Linux)

- Compiling `test.java`
  - `java miniJava/Compiler test.java`
    - use mJAM.ObjectFile to write `test.mJAM` (note spelling!), be sure that it is written in the same directory as `test.java`
    - do not run the generated program as part of compilation!

- Disassembling `test.mJAM`
  - `java mJAM/Disassembler test.mJAM`
    - should write `test.asm` in same directory as `test.mJAM`

- Executing `test.mJAM`
  - `java mJAM/Interpreter test.mJAM`
    - `System.out.println` results from test.java will appear on stdout prefixed by ">>> "

- Debugging `test.mJAM`
  - `java mJAM/Interpreter test.mJAM test.asm`
    - Show machine data store and state, show code, set/remove breakpoints, single instruction execution
    - Type "?" for help

Check results

- To compare miniJava and java semantics of program `foo.java`

1. Run as miniJava program

```
java miniJava/Compiler foo.java
java mJAM/Interpreter foo.mJAM
```

2. Run as java program

```
javac foo.java
java  foo.class
```

- Note that mJAM println prefixes output with "`>>> `"

PA4codegenExample

- The PA4-example (lec 15) is available on our web page
  - generates code for the Counter.java example (lec 16)
    - illustrates the `Machine` interface to generate mJAM instructions
  - .. then executes the generated code using mJAM
    - the Interpreter is started in debug mode so you can trace execution of the example code
    - to simplify the testing of your code generator you can install a similar shortcut to automatically execute generated code (e.g. in your compiler driver)
      - <u>If you do so, be sure to restore standard functionality before submitting PA4</u>