Programming Languages and Techniques (CIS120)

Lecture 26
April 3, 2019

Static Types vs. Dynamic Classes,
The Java ASM, Java Generics
Chapter 24
Announcements

• HW7: Chat Server
  – Available today
  – Due Tuesday, April 9th at 11:59pm
Static Types vs. Dynamic Classes
"Static" types vs. "Dynamic" classes

- The **static type** of an *expression* is a type that describes what we know about the expression at compile-time (without thinking about the execution of the program)
  
  ```
  Displaceable x;
  ```

- The **dynamic class** of an *object* is the class that it was created from at run time
  
  ```
  x = new Point(2,3)
  ```

- In OCaml, we only had static types

- In Java, we also have dynamic classes because of objects
  - The dynamic class will always be a *subtype* of its static type
  - The dynamic class determines what methods are run
What is the static type of `a1` on line A?

```java
public Area asArea (Area a)
    { return a; }

Point p = new Point(5,5)
Circle c = new Circle (p,3);
Area a1 = c;  // A

B y = asArea (c);
```

- Area
- Circle
- None of the above
- Not well typed
public Area asArea (Area a) 
    { return a; } 

Point p = new Point(5,5) 
Circle c = new Circle (p,3); 
Area a1 = c;  // A

__B__ y = asArea (c);

What is the static type of a1 on line A?

1. Area
2. Circle
3. None of the above
4. Not well typed
What is the dynamic class of a1 when execution reaches A?

```java
public Area asArea (Area a) {
    return a;
}
...
Point p = new Point(5,5);
Circle c = new Circle(p,3);
Area a1 = c; // A
__B__ y = asArea(c);
```

- Area
- Circle
- None of the above
- Not well typed
What is the dynamic class of a1 when execution reaches A?

1. Area
2. Circle
3. None of the above
4. Not well typed
What type could we declare for x (in blank B)?

```java
public Area asArea (Area a) {
    return a;
}
...

Point p = new Point(5,5)
Circle c = new Circle (p,3);
Area a1 = c;  // A

B y = asArea (c);
```
Static type vs. Dynamic type

What type could we declare for x (in blank B)?

1. Area
2. Circle
3. Either of the above
4. Not well typed

public Area asArea (Area a) {
    return a;
}

Point p = new Point(5,5);
Circle c = new Circle (p,3);
Area a1 = c;  // A

B y = asArea (c);

10
Inheritance and Dynamic Dispatch

When do constructors execute?
How are fields accessed?
What code runs in a method call?
What is ‘this’?
ASM refinement: The Class Table

Workspace

Stack

Heap

Class Table

...
public class Counter {
    private int x;
    public Counter () { x = 0; }
    public void incBy(int d) { x = x + d; }
    public int get() { return x; }
}

class Decr extends Counter {
    private int y;
    public Decr (int initY) { y = initY; }
    public void dec() { incBy(-y); }
}

The class table contains:
• the code for each method,
• references to each class’s parent, and
• the class’s static members.
Inside a non-static method, the variable `this` is a reference to the object on which the method was invoked.

References to local fields and methods have an implicit “this.” in front of them.

```java
class C {
    private int f;
    public void copyF(C other) {
        this.f = other.f;
    }
}
```
public class Counter {
    private int x;
    public Counter () { x = 0; }
    public void incBy(int d) { x = x + d; }
    public int get() { return x; }
}

public class Decr extends Counter {
    private int y;
    public Decr (int initY) { y = initY; }
    public void dec() { incBy(-y); }
}

// ... somewhere in main:
Decr d = new Decr(2);
d.dec();
d.dec();
int x = d.get();
public class Counter extends Object {
    private int x;
    public Counter () { super(); this.x = 0; }
    public void incBy(int d) { this.x = this.x + d; }
    public int get() { return this.x; }
}

public class Decr extends Counter {
    private int y;
    public Decr (int initY) { super(); this.y = initY; }
    public void dec() { this.incBy(-this.y); }
}

// ... somewhere in main:
Decr d = new Decr(2);
d.dec();
d.dec();
int x = d.get();
Constructing an Object

Workspace

Decr d = new Decr(2);
d.dec();
int x = d.get();

Stack

Heap

Class Table

Object
String toString(){...}
boolean equals...
...

Counter
extends

Counter()
{x = 0; }
void incBy(int d){...}
int get() {return x;}

Decr
extends

Decr(int initY) { ... }
void dec(){incBy(-y);}

Object String toString(){...}
boolean equals...
...

Counter
extends

Counter()
{x = 0; }
void incBy(int d){...}
int get() {return x;}

Decr
extends

Decr(int initY) { ... }
void dec(){incBy(-y);}
Allocating Space on the Heap

Invoking a constructor:
- allocates space for a new object in the heap
- includes slots for all fields of all ancestors in the class tree (here: x and y)
- creates a pointer to the class – this is the object’s dynamic type
- runs the constructor body after pushing parameters and this onto the stack

Note: fields start with a “sensible” default
- 0 for numeric values
- null for references
Call to super:
• The constructor (implicitly) calls the super constructor
• Invoking a method or constructor pushes the saved workspace, the method params (none here) and a new this pointer.

```java
super();
this.y = initY;
```

```
Decr d = _;
d.dec();
int x = d.get();
```

```
this
initY
```

```
class Decr extends Counter {
    x; // initialized to 0

    void dec() { incBy(-y); }
}
```

```
class Counter extends Object {
    int x = 0;

    void incBy(int d) { ... }

    int get() { return x; }
}
```

```
class Object {
    String toString() { ... }

    boolean equals() { ... }
}
```

```
Decr d = _;
d.dec();
int x = d.get();
```
Abstract Stack Machine

Workspace

```
super();
this.x = 0;
```

(Running Object’s default constructor omitted.)

Stack

```
Decr d = _;
d.dec();
int x = d.get();
```

Heap

<table>
<thead>
<tr>
<th>Decr</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
</tbody>
</table>

Class Table

Object

String toString(){...}

boolean equals...

...

Counter

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

Decr

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}
Assigning to a Field

Assignment into the `this.x` field goes in two steps:
- look up the value of `this` in the stack
- write to the “X” slot of that object.

```java
this.x = 0;

Class Table

Object
String toString(){...}
boolean equals...
...

Counter
extends Object
Counter() { x = 0; }
void incBy(int d){...}
int get() {return x;}

Decr
extends Counter
Decr(int initY) { ... }
void dec(){incBy(-y);}
```
Assigning to a Field

Assignment into the `this.x` field goes in two steps:
- look up the value of `this` in the stack
- write to the “x” slot of that object.
Done with the call to "super", so pop the stack to the previous workspace.
this.y = initY;

Continue in the Decr class’s constructor.
Abstract Stack Machine

Workspace

this.y = 2;

Stack

Decr d = _;
d.dec();
int x = d.get();

this

initY 2

Heap

Decr

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>0</td>
</tr>
</tbody>
</table>

Class Table

Object

String toString(){…}
boolean equals…
…

Counter

extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}

Decr

extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}
Assigning to a field

Assignment into the `this.y` field.

(This really takes two steps as we saw earlier, but we’re skipping some for the sake of brevity...)
Done with the call to the Decr constructor, so pop the stack and return to the saved workspace, returning the newly allocated object (now in the this pointer).
Returning the Newly Constructed Object

Workspace

Decr d = ___;
d.dec();
int x = d.get();

Continue executing the program.

Stack

Heap

Class Table

Object

String toString() {
...
}
boolean equals ... 
...

Counter

extends Object

Counter() {
  x = 0;
}
void incBy(int d) {
...}
int get() {
return x;
}

Decr
extends Counter

Decr(int initY) {
...
}
void dec() { incBy(-y); }

Decr

x 0
y 2
Allocating a local variable

Allocate a stack slot for the local variable d. Note that it’s mutable… (bold box in the diagram).

Aside: since, by default, fields and local variables are mutable in Java, we sometimes omit the bold boxes and just assume the contents can be modified.
Invoke the `dec` method on the object. The code is found by "pointer chasing" through the class hierarchy.

This is an example of *dynamic dispatch*: Which code is run depends on the dynamic class of the object. (In this case, `Decr`.)

Search through the methods of the `Decr`, class trying to find one called `dec`.

```
...dec();
int x = d.get();
```
Call the method, remembering the current workspace and pushing the this pointer and any arguments (none in this case).
Reading a Field’s Contents

Workspace

Stack

Heap

Class Table

Object

String toString(){...}

boolean equals...

...

Counter

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

Decr

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}

Read from the y slot of the object.
Invoke the `incBy` method on the object via dynamic dispatch.

In this case, the `incBy` method is inherited from the parent, so dynamic dispatch must search up the class tree, looking for the implementation code.

The search is guaranteed to succeed – Java’s static type system ensures this.
Running the body of `incBy`

**Workspace**
- `this.x = this.x + d;`
- `this.x = -2;`

**Stack**
- `d`
- `int x = d.get();`
- `this`
- `_;
- `d`
- `this`

**Heap**
- `Decr`
- `x 2`
- `y 2`

**Class Table**
- **Object**
  - `String toString(){...}`
  - `boolean equals...`
  - `...`
- **Counter**
  - `extends Object`
  - `Counter() { x = 0; }`
  - `void incBy(int d){...}
  - `int get() {return x;}`
- **Decr**
  - `extends Counter`
  - `Decr(int initY) { ... }
  - `void dec(){incBy(-y);}`

It takes a few steps...

**Body of incBy:**
- Reads `this.x`
- Looks up `d`
- Computes result `this.x + d`
- Stores the answer `-2` in `this.x`
After a few more steps...

Workspace

```java
int x = d.get();
```

Stack

```java
d

Decr

```

Heap

```java
<table>
<thead>
<tr>
<th>x</th>
<th>-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>2</td>
</tr>
</tbody>
</table>

Class Table

```

Object

```java
String toString(){...}
boolean equals...
```

Counter

```java
extends Object
Counter() { x = 0; }
void incBy(int d){...}
int get() {return x;}
```

Decr

```java
extends Counter
Decr(int initY) { ... }
void dec(){incBy(-y);}
```
After yet a few more steps…

Workspace

Stack

<table>
<thead>
<tr>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
</tbody>
</table>

Heap

<table>
<thead>
<tr>
<th>Decr</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
</tbody>
</table>

Class Table

Object

String toString(){...}

boolean equals...

...

Counter

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

Decr

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}

Done! (Phew!)
Summary: this and dynamic dispatch

• When object’s method is invoked, as in `o.m()`, the code that runs is determined by `o`’s *dynamic* class.
  – The dynamic class, represented as a pointer into the class table, is included in the object structure in the heap
  – If the method is inherited from a superclass, determining the code for `m` might require searching up the class hierarchy via pointers in the class table
  – This process of *dynamic dispatch* is the heart of OOP!

• Once the code for `m` has been determined, a binding for `this` is pushed onto the stack.
  – The `this` pointer is used to resolve field accesses and method invocations inside the code.
What is the value of $x$ at the end of this computation?

```java
public class Counter {
    private int x;
    public Counter () { x = 0; }
    public void incBy(int d) { x = x + d; }
    public int get() { return x; }
}

class Decr extends Counter {
    private int y;
    public Decr (int initY) { y = initY; }
    public void dec() { incBy(-y); }
}

// somewhere in main:
Decr d = new Decr(2);
d.dec();
int x = d.get();
```
Inheritance Example

```java
public class Counter {
    private int x;
    public Counter () { x = 0; }
    public void incBy(int d) { x = x + d; }
    public int get() { return x; }
}

class Decr extends Counter {
    private int y;
    public Decr (int initY) { y = initY; }
    public void dec() { incBy(-y); }
}

// ... somewhere in main:
Decr d = new Decr(2);
d.dec();
int x = d.get();
```

What is the value of x at the end of this computation?

1. -2
2. -1
3. 0
4. 1
5. 2
6. NPE
7. Doesn't type check

Answer: -2
Static members and the Java ASM
Static Members

- Classes in Java can also act as *containers* for code and data.
- The modifier `static` means that the field or method is associated with the class and *not* instances of the class.

```java
class C {
    public static int x = 23;
    public static int someMethod(int y) {
        return C.x + y;
    }
    public static void main(String args[]) {
        ...
    }
}

// Elsewhere:
C.x = C.x + 1;
C.someMethod(17);
```

You can do a static assignment to initialize a static field.

Access to the static member uses the class name `C.x` or `C.foo()`.
Based on your understanding of *this*, is it possible to refer to *this* in a static method?

No
Based on your understanding of ‘this’, is it possible to refer to ‘this’ in a static method?

1. No
2. Yes
3. I’m not sure
The class table entry for `C` has a field slot for `x`.

Updates to `C.x` modify the contents of this slot: `C.x = 17`;

A static field is a *global* variable

- There is only one heap location for it (in the class table)
- Modifications to such a field are visible everywhere the field is
  - if the field is public, this means *everywhere*
- Use with care!
• Static methods do *not* have access to a `this` pointer
  – Why? There isn’t an instance to dispatch through!
  – Therefore, static methods may only directly call other static methods.
  – Similarly, static methods can only directly read/write static fields.
  – Of course a static method can create instance of objects (via `new`) and then invoke methods on those objects.

• Gotcha: It is possible (but confusing) to invoke a static method as though it belongs to an object instance.
  – e.g. `o.someMethod(17)` where `someMethod` is static
Java Generics

Subtype Polymorphism

vs.

Parametric Polymorphism
Review: Subtype Polymorphism*

- Main idea:

  Anywhere an object of type A is needed, an object that is a subtype of A can be provided.

- If B is a subtype of A, it provides all of A’s (public) methods.

*polymorphism = many shapes
Recap: Subtyping

- Interfaces extend (possibly many) interfaces
- Classes implement (possibly many) interfaces
- Classes (except Object) extend exactly one other class (Object by default)
- Interface types (and arrays) are subtypes “by fiat” of Object
Is subtype polymorphism enough?
Mutable Queue Interface in OCaml

```ocaml
module type QUEUE =
 sig
 (* type of the data structure *)
type 'a queue

 (* Make a new, empty queue *)
 val create : unit -> 'a queue

 (* Add a value to the end of the queue *)
 val enq : 'a -> 'a queue -> unit

 (* Remove the front value and return it (if any) *)
 val deq : 'a queue -> 'a

 (* Determine if the queue is empty *)
 val is_empty : 'a queue -> bool
 end
```

How can we translate this interface to Java?
Java Interface using Subtyping

module type QUEUE =

sig

  type 'a queue

  val create : unit -> 'a queue

  val enq : 'a -> 'a queue -> unit

  val deq : 'a queue -> 'a

  val is_empty : 'a queue -> bool

end

OCaml

interface ObjQueue {

  // no constructors
  // in an interface

  public void enq(Object elt);

  public Object deq();

  public boolean isEmpty();

}
interface ObjQueue {
    public void enq(Object elt);
    public Object deq();
    public boolean isEmpty();
}

ObjQueue q = ...;
q.enq("CIS 120");
A x = q.deq();

What type should we write for A?
1. String
2. Object
3. ObjQueue
4. None of the above

ANSWER: Object
interface ObjQueue {
    public void enq(Object elt);
    public Object deq();
    public boolean isEmpty();
}

ObjQueue q = ...;
q.enq(" CIS 120 ");
Object x = q.deq();
System.out.println(x.trim());

ANSWER: No

trim is a method of the String class (removes extra spaces)
Subtype Polymorphism

```java
interface ObjQueue {
    public void enq(Object elt);
    public Object deq();
    public boolean isEmpty();
}
```

ObjQueue q = ...;
q.enq("CIS 120");
Object x = q.deq();
//System.out.println(x.trim());
q.enq(new Point(0.0,0.0));
___B___ y = q.deq();

What type for B?

1. Point
2. Object
3. ObjQueue
4. None of the above

ANSWER: Object
Parametric Polymorphism (a.k.a. Generics)

- Main idea:
  
  Parameterize a type (i.e. interface or class) by another type.

```
public interface Queue<E> {
    public void enq(E o);
    public E deq();
    public boolean isEmpty();
}
```

- The implementation of a parametric polymorphic interface cannot depend on the implementation details of the parameter.
  - the implementation of `enq` cannot invoke any methods on `o` (except those inherited from Object)
  - i.e. the only thing we know about E is that it is a subtype of Object
Generics (Parametric Polymorphism)

```java
public interface Queue<E> {
    public void enq(E o);
    public E deq();
    public boolean isEmpty();
    ...
}
```

```java
Queue<String> q = ...;
q.enq("CIS 120");
String x = q.deq(); // What type of x? String
System.out.println(x.trim()); // Is this valid? Yes!
q.enq(new Point(0.0,0.0)); // Is this valid? No!
```
Subtyping and Generics
Subtyping and Generics*

Java generics are *invariant*:

- Subtyping of *arguments* to generic types does not imply subtyping between the instantiations:

```
Queue<String> qs = new QueueImpl<String>();
Queue<Object> qo = qs;
qo.enq(new Object());
String s = qs.deq();
```

Hardest part to learn about generics and subtyping...

* Subtyping and generics interact in other ways too. Java supports *bounded polymorphism* and *wildcard types*, but those are beyond the scope of CIS 120.
Subtyping with Generics

Which of these are true, assuming that class QueueImpl<E> implements interface Queue<E>?

1. QueueImpl<Queue<String>> is a subtype of Queue<Queue<String>>
2. Queue<QueueImpl<String>> is a subtype of Queue<Queue<String>>
3. Both
4. Neither
Which of these are true, assuming that class QueueImpl<E> implements interface Queue<E>?

1. QueueImpl<Queue<String>> is a subtype of Queue<Queue<String>>
2. Queue<QueueImpl<String>> is a subtype of Queue<Queue<String>>
3. Both
4. Neither

Answer: 1
Other subtleties with Generics

• Unlike OCaml, Java classes and methods can be generic only with respect to reference types.
  – Not possible to do: Queue<int>
  – Must instead do: Queue<Integer>

• Java Arrays cannot be generic
  – Not possible to do:

```java
class C<E> {
    E[] genericArray;
    public C() {
        genericArray = new E[];
    }
}
```