Dynamic Dispatch and Generics
Announcements

• HW08 (Text Adventure) is due Monday at 11:59:59pm

• *Midterm 2 is Friday, March 29th in class*
  – Mutable state (in OCaml and Java)
  – Objects (in OCaml and Java)
  – ASM (in OCaml and Java)
  – Reactive programming (in OCaml)
  – Arrays in (Java)
  – Subtyping & Inheritance (in Java)

• Practice exams available on course website
The Java Abstract Stack Machine and the Class Table

1. When do constructors execute?
2. How are fields accessed?
3. What code runs in a method call?
```java
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();
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);}
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/constructor pushes the saved workspace, the method params (none here) and a new this pointer.

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);}
Abstract Stack Machine

Workspace

super();
this.x = 0;

Stack

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

this

initY 2

_; this.y = initY;

this

Heap

Decr

x 0

y 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);}

(Running Object’s default constructor omitted.)
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.
Assigning 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.
Continue in the `Decr` class’s constructor.
Abstract Stack Machine

Workspace

Stack

Heap

Class Table

this.y = 2;

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

this

initY 2

Decr

x 0

y 0

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);}

CIS120 / Spring 2013
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...)

```
this.y = 2;
Decr d = _;
d.dec();
int x = d.get();
this
initY = 2
Decr
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 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

Stack

Heap

Class Table

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

Decr

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>2</td>
</tr>
</tbody>
</table>

Object

String toString(){...

Counter

extends Object

Counter() { x = 0; }

void incBy(int d){...}

int get() {return x;}

Decr

extends Counter

Decr(int initY) { ... }

void dec(){incBy(-y);}
AllocaSng a local variable

Workspace

Stack

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

Heap

Derc

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);}
Dynamic Dispatch: Finding the Code

Invoke the dec method on the object. The code can be found by “pointer chasing”.

This process is called *dynamic dispatch* – which code is run depends on the dynamic class of the object. (In this case, `Decr`.)
Dynamic Dispatch: Finding the Code

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);}

Call the method, remembering the current workspace and pushing the this pointer and any arguments (none in this case).

d

\_

int x = d.get();

this

this.incBy(-this.y);
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);}
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

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

CIS120 / Spring 2013
After a few more steps...

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

Now use dynamic dispatch to invoke the `get` method for `d`. This involves searching up the class tree again...

```
Decr
  x  -2
  Y  2

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

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

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

Workspace

Stack

Heap

Class Table

Object
After a few more steps...

Now use dynamic dispatch to invoke the `get` method for `d`. This involves searching up the class tree again...
After yet a few more steps...

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);
}

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, which is just a pointer to a class, 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 is called dynamic dispatch (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.
Java Generics

Queues in Java
Mutable Queue ML Interface

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

  (* Remove the first occurrence of the value. *)
  val remove : 'a -> 'a queue -> unit
end
public interface Queue<E> {

    /** Determine if the queue is empty */
    public boolean is_empty ();

    /** Add a value to the end of the queue */
    public void enq (E elt);

    /** Remove the front value and return it (if any) */
    public E deq ();

    /** Remove the first occurrence of the value */
    public void remove (E elt);

}
module type QUEUE =
  sig
    type 'a queue
    val create : unit -> 'a queue
    val is_empty :
      'a queue -> bool
    val enq :
      'a -> 'a queue -> unit
    val deq : 'a queue -> 'a
    val remove :
      'a -> 'a queue -> unit
  end

public interface Queue<E> {

  public boolean is_empty ();

  public void enq (E elt);

  public E deq ();
  public void remove (E elt);
}
Why Generics?
public interface ObjQueue {
    public void enq(Object o);
    public Object deq();
    public boolean isEmpty();
    public boolean contains(Object o);
    ...
}

ObjQueue q = ...;

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

// What type for A? Object
// Is this valid? No!
// What type for B? Object
Generics (Parametric Polymorphism)

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

Queue<String> q = ...;

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

// What type for A?  String
// Is this valid?  Yes!
// What type for B?  Point
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(0);
```

// Ok? Sure!
// Ok?

Nonononono!
The Java Collections Library

A case study in subtyping and generics.

(Also very useful!)
Interfaces* of the Collections Library

Reminder: Collection<E> is a generic collection type, in OCaml we’d write: ‘e collection

*not all of them
We’ve already seen this interface in the OCaml part of the course.

Most collections are designed to be *mutable* (like queues)

* Why not E? Internally, collections use the equals method to check for equality – membership is determined by o.equals, which does not have to be false for objects of different types. Most applications only store and remove one type of element in a collection, in which case this subtlety never becomes an issue.
Sequences

Iterable\(<E>\)

Collection\(<E>\)

List\(<E>\)    Deque\(<E>\)

LinkedLIst\(<E>\)

ArrayList\(<E>\)    ArrayDeque\(<E>\)

CIS120 / Spring 2013
Sets and Maps

- Collection<E>
  - Set<E>
    - HashSet<E>
    - TreeSet<E>
    - SortedSet<E>
  - SortedMap<K,V>
    - TreeMap<E>
    - HashMap<E>

- Map<K,V>
  - SortedMap<K,V>

CIS120 / Spring 2013
1. Collection<E>
2. List<E> and Set<E>
3. Iterable<E> and Iterator<E>
Iterating over collections

Iterators, while, for, for-each loops
interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void delete(); // optional
}

interface Iterable<E> {
    public Iterator<E> iterator();
}
While Loops

Syntax:

```java
// repeat body until condition becomes false
while (condition) {
    body
}
```

Example:

```java
List<Book> shelf = ... // create a list of Books

// iterate through the elements on the shelf
Iterator<Book> iter = shelf.iterator();
while (iter.hasNext()) {
    Book book = iter.next();
    catalogue.addInfo(book);
    numBooks = numBooks + 1;
}
```
For Loops

Syntax:

```java
for (init-stmt; condition; next-stmt) {
    body
}
```

Equivalent While Loop:

```java
init-stmt;
while (condition) {
    body
    next-stmt;
}
```

Example:

```java
List<Book> shelf = ...  // create a list of Books

// iterate through the elements on the shelf
for (Iterator<Book> iter = shelf.iterator();
    iter.hasNext();
    book = iter.next()) {
    catalogue.addInfo(book);
    numBooks = numbooks+1;
}
```
For-each Loops

syntax:

```java
// repeat body for each element in collection
for (type var : coll) {
  body
}
```

example:

```java
List<Book> shelf = ... // create a list of books

// iterate through the elements on a shelf
for (Book book : shelf) {
  catalogue.addInfo(book);
  numBooks = numBooks+1;
}
```
For-each can be used to iterate over arrays or any class that implements the `Iterable<E>` interface (notably `Collection<E>` and its subinterfaces).

```java
int[] arr = ... // create an array of ints

// count the non-null elements of an array
for (int elt : arr) {
    if (elt != 0) cnt = cnt+1;
}
```
Java Packages

• Java code can be organized into *packages* that provide namespace management.
  – Somewhat like OCaml’s modules
  – Packages contain groups of related classes and interfaces.
  – Packages are organized hierarchically in a way that mimics the file system’s directory structure.

• A .java file can *import* (parts of) packages that it needs access to:

```java
import org.junit.Test;       // just the JUnit Test class
import java.util.*;         // everything in java.util
```

• Important packages:
  – java.lang, java.io, java.util, java.math, org.junit

• See documentation at:
  http://download.oracle.com/javase/6/docs/api/index.html