Inheritance and Dynamic Dispatch
Announcements

- **Midterm 2 is Friday, November 6th in class**
  - Last names A – L  Leidy Labs 10 (here)
  - Last names M – Z  Cohen G17

- Coverage:
  - Mutable state (in OCaml and Java)
  - Objects (in OCaml and Java)
  - ASM (in OCaml and Java)
  - Reactive programming (in Ocaml)
  - Arrays (in Java)
  - Subtyping & Simple Extension (in Java)

- Review Session: TBA
Subtypes and Supertypes

- An interface represents a *point of view* about an object
- Classes can implement *multiple* interfaces

Types can have many different supertypes / subtypes
1. Interface extension
2. Class extension (Simple inheritance)
• Build richer interface hierarchies by extending existing interfaces.

```java
public interface Displaceable {
    double getX();
    double getY();
    void move(double dx, double dy);
}

public interface Area {
    double getArea();
}

public interface Shape extends Displaceable, Area {
    Rectangle getBoundingBox();
}
```

Note the use of the “extends” keyword.

The Shape type includes all the methods of Displaceable and Area, plus the new getBoundingBox method.
• Shape is a subtype of both Displaceable and Area.
• Circle and Rectangle are both subtypes of Shape, and, by transitivity, both are also subtypes of Displaceable and Area.
• Note that one interface may extend several others.
  – Interfaces do not necessarily form a tree, but the hierarchy has no cycles.
Interface Extension Demo

See: Shapes.zip
Class Extension: Inheritance

- Classes, like interfaces, can also extend one another.
  - Unlike interfaces, a class can extend only one other class.
- The extending class *inherits* all of the fields and methods of its superclass, and may include additional fields or methods.
  - This captures the “is a” relationship between objects (e.g. a Car is a Vehicle).
  - Class extension should *never* be used when “is a” does not relate the subtype to the supertype.

```java
class D {
    private int x;
    private int y;
    public int addBoth() { return x + y; }
}

class C extends D { // every C is a D
    private int z;
    public int addThree() {return (addBoth() + z); }
}
```
Simple Inheritance

• In *simple inheritance*, the subclass only *adds* new fields or methods.
• Use simple inheritance to *share common code* among related classes.
• Example: Circle, and Rectangle have *identical* code for `getX()`, `getY()`, and `move()` methods when implementing `Displaceable`.

```java
public class Displaceable {
    public int x;
    public int y;

    public void move(int dx, int dy) {
        x += dx;
        y += dy;
    }
}
```
Subtyping with Inheritance

- Type C is a subtype of D if D is reachable from C by following zero or more edges upwards in the hierarchy.
- e.g. Circle is a subtype of Area, but Point is not
Example of Simple Inheritance

See: Main2.java
Inheritance: Constructors

- Constructors *cannot* be inherited (they have the wrong names!)
  - Instead, a subclass invokes the constructor of its super class using the keyword ‘super’.
  - Super *must* be the first line of the subclass constructor, unless the parent class constructor takes no arguments, in which it is OK to omit the call to super (it is called implicitly).

```java
class D {
    private int x;
    private int y;
    public D (int initX, int initY) { x = initX; y = initY; }
    public int addBoth() { return x + y; }
}

class C extends D {
    private int z;
    public C (int initX, int initY, int initZ) {
        super(initX, initY);
        z = initZ;
    }
    public int addThree() {return (addBoth() + z); }
}
```
Other forms of inheritance

- Java has other features related to inheritance (some of which we will discuss later in the course):
  - A subclass might *override* (re-implement) a method already found in the superclass.
  - A class might be *abstract* – i.e. it does not provide implementations for all of its methods (its subclasses must provide them instead)

- These features are hard to use properly, and the need for them arises only in somewhat special cases
  - Making reusable libraries
  - Special methods: equals and toString

- We recommend avoiding *all* forms of inheritance (even “simple inheritance”) when possible – prefer interfaces and composition.

*Especially: avoid overriding.*
When do constructors execute?
How are fields accessed?
What code runs in a method call?
Revenge of the Son of the Abstract Stack Machine
How do method calls work?

• What code gets run in a method invocation?
  
  ```
  o.move(3,4);
  ```

• When that code is running, how does it access the fields of the object that invoked it?
  
  ```
  x = x + dx;
  ```

• When does the code in a constructor get executed?

• What if the method was inherited from a superclass?
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); }
}

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.

class C {
    private int f;
    public void copyF(C other) {
        this.f = other.f;
    }
}

Stack
  this

Heap
  C
  f 0

...
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();
int x = d.get();
...with Explicit this and super

```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

<table>
<thead>
<tr>
<th>Workspace</th>
<th>Stack</th>
<th>Heap</th>
<th>Class Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\text{Decr}\ d = \text{new Decr}(2);]</td>
<td>[\text{Counter}]</td>
<td>[\text{Heap}]</td>
<td>[\text{Object}]</td>
</tr>
<tr>
<td>[d.\text{dec}();]</td>
<td>[\text{Function}]</td>
<td>[\text{Memory}]</td>
<td>[\text{String\ toString()}]</td>
</tr>
<tr>
<td>[\text{int}\ x = d.\text{get}();]</td>
<td>[\text{Counter}]</td>
<td>[\text{Class\ Table}]</td>
<td>[\text{boolean\ equals...}]</td>
</tr>
</tbody>
</table>

### Class Table

- **Object**
  - `String\ toString(){}...`
  - `boolean\ equals...`
  - `...`

- **Counter**
  - `extends`
  - `Counter() \{ x = 0; \}`
  - `void\ incBy(int\ d){...}
  - `int\ get() \{\text{return\ }x;\}`

- **Decr**
  - `extends`
  - `Decr(int\ initY) \{ ... \}
  - `void\ dec()\{incBy(-y);\}`
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.
Abstract Stack Machine

Workspace

super();
this.x = 0;

(Calling Object’s default constructor omitted.)

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);}
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.
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.
Continue in the `Decr` class's constructor.

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

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

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

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

```
<table>
<thead>
<tr>
<th></th>
<th>Workspace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>this.y = initY;</td>
</tr>
</tbody>
</table>
```

```java
Object
String toString() {

boolean equals...

...}
```

```java
Counter
extends Object

Counter() {
    x = 0;
}

void incBy(int d) {
}

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

```java
Decr
extends Counter

Decr(int initY) {
    ...
}

void dec() {
    incBy(-y);
}
```
Decr\(d = _;\)
\(\text{d.incBy}(-y);\)

\(\text{Int} x = \text{d.get}();\)

\(\text{this.y} = 2;\)

\textbf{Abstract Stack Machine}

\textbf{Workspace}

\textbf{Stack}

\textbf{Heap}

\textbf{Object}

\textbf{Counter}

\textbf{Decr}

\textbf{Class Table}

\begin{itemize}
  \item \textbf{Object} \[
    \begin{align*}
    \text{String} & \quad \text{toString}() \{ \ldots \} \\
    \text{boolean} & \quad \text{equals}() \ldots
  \end{align*}
  \end{itemize}

\begin{itemize}
  \item \textbf{Counter} \[
    \begin{align*}
    &\text{extends} \quad \text{Object} \\
    &\text{Counter()} \{ x = 0; \} \\
    &\text{void} \quad \text{incBy}() \{ \ldots \} \\
    &\text{int} \quad \text{get()} \{ \text{return} \ x; \}
  \end{align*}
  \end{itemize}

\begin{itemize}
  \item \textbf{Decr} \[
    \begin{align*}
    &\text{extends} \quad \text{Counter} \\
    &\text{Decr()} \{ \text{initY} \} \{ \ldots \} \\
    &\text{void} \quad \text{dec()} \{ \text{incBy}(-y); \}
  \end{align*}
  \end{itemize}
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...)

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

this.y = 2;
```
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).
Continuing the program:

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

**Heap**:
- `Decr`:
  - `x`: 0
  - `y`: 2

**Class Table**:
- **Object**
  - `toString()`: ...
  - `equals()`: ...
- **Counter**
  - Extends `Object`
  - `Counter()`: `x = 0;`
  - `incBy(int d)` ...
  - `get()`: `return x;`
- **Decr**
  - Extends `Counter`
  - `Decr(int initY)`: ...
  - `dec()`: `incBy(-y);`
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, we often omit the bold boxes and just assume the contents can be modified.
Dynamic Dispatch: Finding the Code

Invoke the `dec` method on the object. The code is found by “pointer chasing” through the class hierarchy.

This process is called *dynamic dispatch*: Which code is run depends on the dynamic class of the object. (In this case, `Decr`.)
Call the method, remembering the current workspace and pushing the `this` pointer and any arguments (none in this case).
Decr extends Counter
Decr(int initY) { ... }
void dec() { incBy(-y); }

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

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

Heap
Decr
0
2

Stack
d
x
this

Workspace
this.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

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

```java
class Counter {
  int x = 0;
  void incBy(int d) {
    this.x = this.x + d;
  }
  int get() { return this.x; }
}
class Decr extends Counter {
  int initY;
  void dec() {
    incBy(-initY);
  }
}
```
Now use dynamic dispatch to invoke the get method for d. This involves searching up the class hierarchy again...
After yet a few more steps...

Workspace

Stack

Heap

Class Table

```
Workspace

Stack

d

x

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);}
```

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

• When object’s method is invoked, as in \texttt{o.m()}, the code that runs is determined by \texttt{o}’s \textit{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 \texttt{m} might require searching up the class hierarchy via pointers in the class table
  – This process of \textit{dynamic dispatch} is the heart of OOP!

• Once the code for \texttt{m} has been determined, a binding for \texttt{this} is pushed onto the stack.
  – The \texttt{this} pointer is used to resolve field accesses and method invocations inside the code.