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
What is the static type of a1 on line A?

1. Area
2. Circle
3. None of the above
4. Not well typed
Static type vs. Dynamic class quiz

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 dynamic class of a1 when execution reaches A?

1. Area
2. Circle
3. None of the above
4. Not well typed

Circle
Static type vs. Dynamic class quiz

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

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

• Homework 7 out yesterday (a little early)
  – Due Tuesday, April 3, 11:59 p.m.

• If you are doing a 2nd major in CIS or considering it:
  You are invited to an advising session
  **Wednesday, March 28 from 6 - 7pm**
  Wu and Chen Auditorium
  Dr. Val Tannen answer questions that you may have
Inheritance and Dynamic Dispatch

When do constructors execute?
How are fields accessed?
What code runs in a method call?
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;
    }
}
**An 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; }
}

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

**Class Table**

**Object**
- `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);}`
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;

(Running Object’s default constructor omitted.)

Stack

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

this

initY

2

this

initY

-;
this.y = initY;

this

Heap

Decr

x

y

0

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);}

Decr

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

this

initY

2

this

initY

-;
this.y = initY;

this

0

0

0
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
Decr d = _;
d.dec();
int x = d.get();
this.y = initY;
```

```
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);}
```
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

Workspace

;  

Stack

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

this

initY 2

this

-; this.y = initY;

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);}

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();
this.y = initY;
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);}
```
\textbf{Abstract Stack Machine}

\textbf{Workspace}
\begin{itemize}
  \item this.y = 2;
\end{itemize}

\textbf{Stack}
\begin{itemize}
  \item Decr $d = \_;$
  \item $d$.dec();
  \item int $x = d$.get();
\end{itemize}

\textbf{Heap}
\begin{itemize}
  \item \textbf{Decr}
    \begin{tabular}{c|c}
      \hline
      x & 0 \\
      \hline
      y & 0 \\
    \end{tabular}
\end{itemize}

\textbf{Class Table}
\begin{itemize}
  \item \textbf{Object}
    \begin{itemize}
      \item \text{String toString()}\{…\}
      \item \text{boolean equals…}
    \end{itemize}
  \item \textbf{Counter}
    \begin{itemize}
      \item \text{extends Object}
      \item \text{Counter()} \{ x = 0; \}
      \item \text{void incBy(int d)}\{…\}
      \item \text{int get()} \{\text{return x;}\}
    \end{itemize}
  \item \textbf{Decr}
    \begin{itemize}
      \item \text{extends Counter}
      \item \text{Decr(int initY) \{ … \}}
      \item \text{void dec()}\{\text{incBy(-y);}\}
    \end{itemize}
\end{itemize}
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).
Decr d = new Decr();
d.dec();
int x = d.get();

Continue executing the program.
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.
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`.)

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

---

**Workspace**

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

---

**Stack**

```
d
```

---

**Heap**

```
Decr
| x | 0 |
| y | 2 |
```

---

**Class Table**

**Object**

- `toString()`
- `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.incBy(-this.y);
```
Reading A Field’s Contents

Workspace

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

Stack

```
d

int x = d.get();

this
```

Heap

```
Decr

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

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.

Search through the methods of the `Decr` class trying to find one called `incBy`. If the search fails, recursively search the parent classes.
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**
- `toString()`
- `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`
int x = d.get();

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

d 

x -2

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 `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.
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?

1. No
2. Yes
3. I’m not sure