Programming Languages and Techniques (CIS120)

Lecture 25

March 21, 2016

Subtyping

Chapter 23

```java
public interface Area {
  public double getArea();
}
```

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

```java
public class Circle implements Displaceable, Area {
  private int x, y, r;
  public Circle(int r, int x0, int y0,) { … }
  public int getX() { return x; }
  public int getY() { return y; }
  public void move(int dx, int dy) { … }
  public double getArea() { return Math.pi * r * r; }
}
```

What line has a type error in the program below (if any)?

```java
1. Displaceable circle = new Circle(0, 0, 3);
2. int x = circle.getX();
3. circle.move(2,3);
4. double size = circle.getArea();
```

5. none of the above

Answer: 4

Announcements

- Midterm 2, tomorrow night!

- Focus of exam: Higher-order programming in OCaml with mutable state (Lecture notes Chapters 11-20).

Types and Subtyping

Why Static Types?

- Types stop you from using values incorrectly
  - `3.m()`
  - `if (3) { return 1; } else { return 2; }`
  - `3 + true`
  - `(new Counter()).m()`
- All *expressions* have types
  - `3 + 4` has type `int`
  - `“A”.toLowerCase()` has type `String`
  - `new ResArray()` has type `ResArray`
- How do we know if `x.m()` is correct? or `x+3`?
  - depends on the type of `x`
  - variable declarations specify types of variables
- Type restrictions preserve the types of variables
  - assignment "x = v" must be to values with compatible types
  - methods "o.m(3)" must be called with compatible argument types
- HOWEVER: in Java, values can have *multiple* types....

Subtyping

Subtyping

Definition:
Type A can be a *subtype* of type B if A offers the same public methods that B does.

- Type B is called the *supertype* of A.
- Intuitively: an A object can do anything that a B object can
- Note: A may provide *more* public methods

Explicit Subtyping

- Java requires subtypes to be declared *explicitly* via keywords `implements` and `extends`
  - there is no subtyping by "coincidence" (i.e. just because the public method names happen to be the same)

- **Example**: A class that implements an interface is a subtype of the interface:

```
interface Displaceable { … }

public class ColorPoint implements Displaceable {
    …
}
```

Subtyping and Variables

- A a *variable* declared with type A can store any *object* that is a subtype of A

```
Area a = new Circle(1, 2, 3);
```

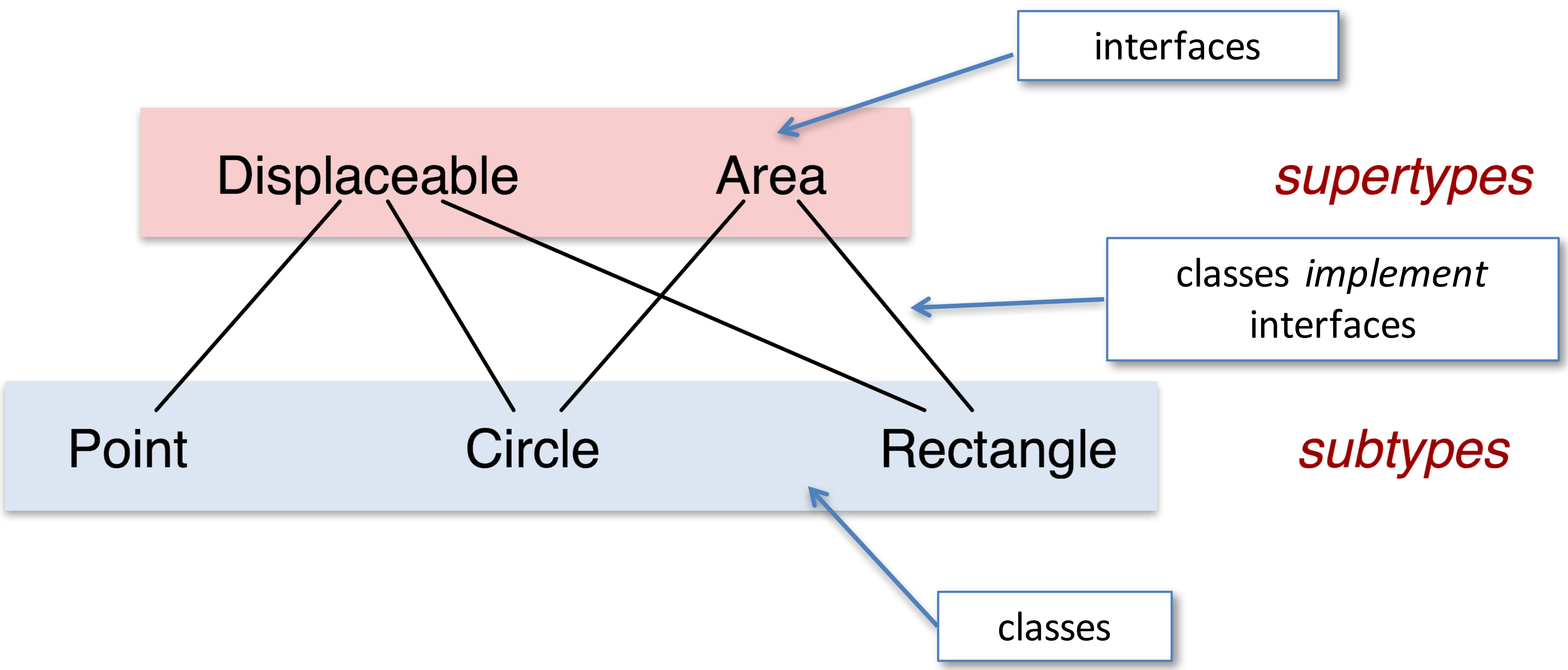
supertype of Circle   subtype of Area

- Methods with *parameters* of type A must be called with *arguments* that are subtypes of A

```
static double m (Area x) {
   return x.getArea() * 2;
}
...
C.m( new Circle(1, 2, 3) );
```

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

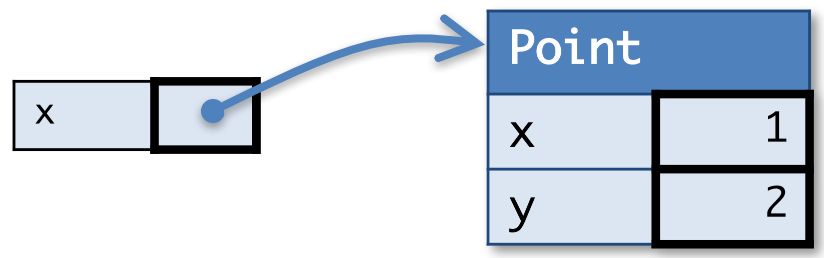
"Static" types vs. "Dynamic" classes

- The **static type** of an *expression* is a type that describes what we (and the compiler) 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 constructed from at run time

```
x = new Point(1,2)
```



- In OCaml, we only had static types
- In Java, we also have dynamic classes
  – The dynamic class will always be a *subtype* of its static type
  – The dynamic class determines what method executes at runtime

Static type vs. Dynamic class quiz

```
public Area asArea (Area s) {
   return s;
}
…
Rectangle r =
  new Rectangle (1,2,1,1);
Circle c = new Circle (1,1,3);
Area s1 = r;  // A
Area s2 = c;  // B
s2 = r;       // C

__D__ x = asArea (r);
__E__ y = asArea (s1);

s1 = c;   // F
s1 = s2;  // G
r = c;    // H
r = s1;   // I
```

What is the static type of s1 on line A?

1. Rectangle
2. Circle
3. Area
4. none of the above

Static type vs. Dynamic class quiz

```
public Area asArea (Area s) {
   return s;
}
…
Rectangle r =
  new Rectangle (1,2,1,1);
Circle c = new Circle (1,1,3);
Area s1 = r;  // A
Area s2 = c;  // B
s2 = r;       // C

__D__ x = asArea (r);
__E__ y = asArea (s1);

s1 = c;   // F
s1 = s2;  // G
r = c;    // H
r = s1;   // I
```

What is the dynamic class of s1 when execution reaches A?

1. Rectangle
2. Circle
3. Area
4. none of the above

Static type vs. Dynamic class quiz

```
public Area asArea (Area s) {
   return s;
}
…
Rectangle r =
  new Rectangle (1,2,1,1);
Circle c = new Circle (1,1,3);
Area s1 = r;  // A
Area s2 = c;  // B
s2 = r;       // C

__D__ x = asArea (r);
__E__ y = asArea (s1);

s1 = c;   // F
s1 = s2;  // G
r = c;    // H
r = s1;   // I
```

What is the static type of s2 on line B?

1. Rectangle
2. Circle
3. Area
4. none of the above

Static type vs. Dynamic class quiz

```
public Area asArea (Area s) {
   return s;
}
…
Rectangle r =
  new Rectangle (1,2,1,1);
Circle c = new Circle (1,1,3);
Area s1 = r;   // A
Area s2 = c;   // B
s2 = r;        // C

__D__ x = asArea (r);
__E__ y = asArea (s1);

s1 = c;    // F
s1 = s2;   // G
r = c;     // H
r = s1;    // I
```

What type should we declare for x (in blank D)?

1. Rectangle
2. Circle
3. Area
4. none of the above

Static type vs. Dynamic class quiz

```
public Area asArea (Area s) {
   return s;
}
…
Rectangle r =
  new Rectangle (1,2,1,1);
Circle c = new Circle (1,1,3);
Area s1 = r;  // A
Area s2 = c;  // B
s2 = r;       // C

__D__ x = asArea (r);
__E__ y = asArea (s1);

s1 = c;   // F
s1 = s2;  // G
r = c;    // H
r = s1;   // I
```

What is the dynamic class of x?

1. `Rectangle`
2. `Circle`
3. `Area`
4. none of the above

Static type vs. Dynamic class quiz

```
public Area asArea (Area s) {
   return s;
}
…
Rectangle r =
  new Rectangle (1,2,1,1);
Circle c = new Circle (1,1,3);
Area s1 = r;  // A
Area s2 = c;  // B
s2 = r;       // C

__D__ x = asArea (r);
__E__ y = asArea (s1);

s1 = c;   // F
s1 = s2;  // G
r = c;    // H
r = s1;   // I
```

What type should we declare for y (in blank E)?

1. `Rectangle`
2. `Circle`
3. `Area`
4. none of the above

Static type vs. Dynamic class quiz

```
public Area asArea (Area s) {
   return s;
}
…
Rectangle r =
  new Rectangle (1,2,1,1);
Circle c = new Circle (1,1,3);
Area s1 = r;  // A
Area s2 = c;  // B
s2 = r;       // C

__D__ x = asArea (r);
__E__ y = asArea (s1);

s1 = c;   // F
s1 = s2;  // G
r = c;    // H
r = s1;   // I
```

What is the dynamic class of y?

1. `Rectangle`
2. `Circle`
3. `Area`
4. none of the above

Static type vs. Dynamic class quiz

```
public Area asArea (Area s) {
    return s;
}
…
Rectangle r =
   new Rectangle (1,2,1,1);
Circle c = new Circle (1,1,3);
Area s1 = r;   // A
Area s2 = c;   // B
s2 = r;        // C

__D__ x = asArea (r);
__E__ y = asArea (s1);

s1 = c;    // F
s1 = s2;   // G
r = c;     // H
r = s1;    // I
```

Is the assignment on line F well typed?

1. yes
2. no

Static type vs. Dynamic class quiz

```
public Area asArea (Area s) {
    return s;
}
…
Rectangle r =
  new Rectangle (1,2,1,1);
Circle c = new Circle (1,1,3);
Area s1 = r;  // A
Area s2 = c;  // B
s2 = r;       // C

__D__ x = asArea (r);
__E__ y = asArea (s1);

s1 = c;   // F
s1 = s2;  // G
r = c;    // H
r = s1;   // I
```

Is the assignment on line G well typed?

1. yes
2. no

Static type vs. Dynamic class quiz

```
public Area asArea (Area s) {
   return s;
}
…
Rectangle r =
  new Rectangle (1,2,1,1);
Circle c = new Circle (1,1,3);
Area s1 = r;  // A
Area s2 = c;  // B
s2 = r;       // C

__D__ x = asArea (r);
__E__ y = asArea (s1);

s1 = c;   // F
s1 = s2;  // G
r = c;    // H
r = s1;   // I
```

Is the assignment on line H well typed?

1. yes
2. no

Static type vs. Dynamic class quiz

```
public Area asArea (Area s) {
   return s;
}
…
Rectangle r =
  new Rectangle (1,2,1,1);
Circle c = new Circle (1,1,3);
Area s1 = r;  // A
Area s2 = c;  // B
s2 = r;       // C

__D__ x = asArea (r);
__E__ y = asArea (s1);

s1 = c;   // F
s1 = s2;  // G
r = c;    // H
r = s1;   // I
```

Is the assignment on line I well typed?

1. yes
2. no

More Subtyping

Extension

1. Interface extension
2. Class extension (Simple inheritance)

Interface Extension

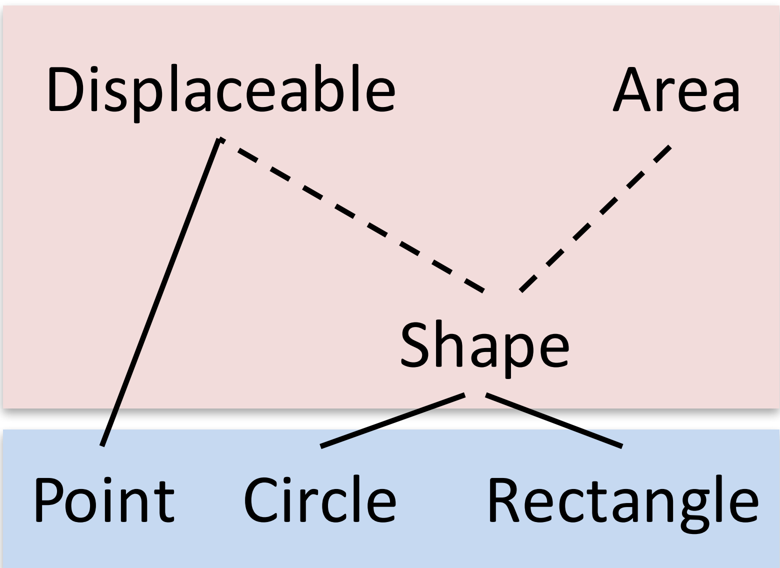
- Build richer interface hierarchies by *extending* existing interfaces.

```
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();
}
```

The Shape type includes all the methods of Displaceable and Area, plus the new getBoundingBox method.

Note the use of the “extends” keyword.

Interface Hierarchy




```
class Point implements Displaceable {
   … // omitted
}
class Circle implements Shape {
   … // omitted
}
class Rectangle implements Shape {
   … // omitted
}
```

- 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: Main1.java

Class Extension: Inheritance

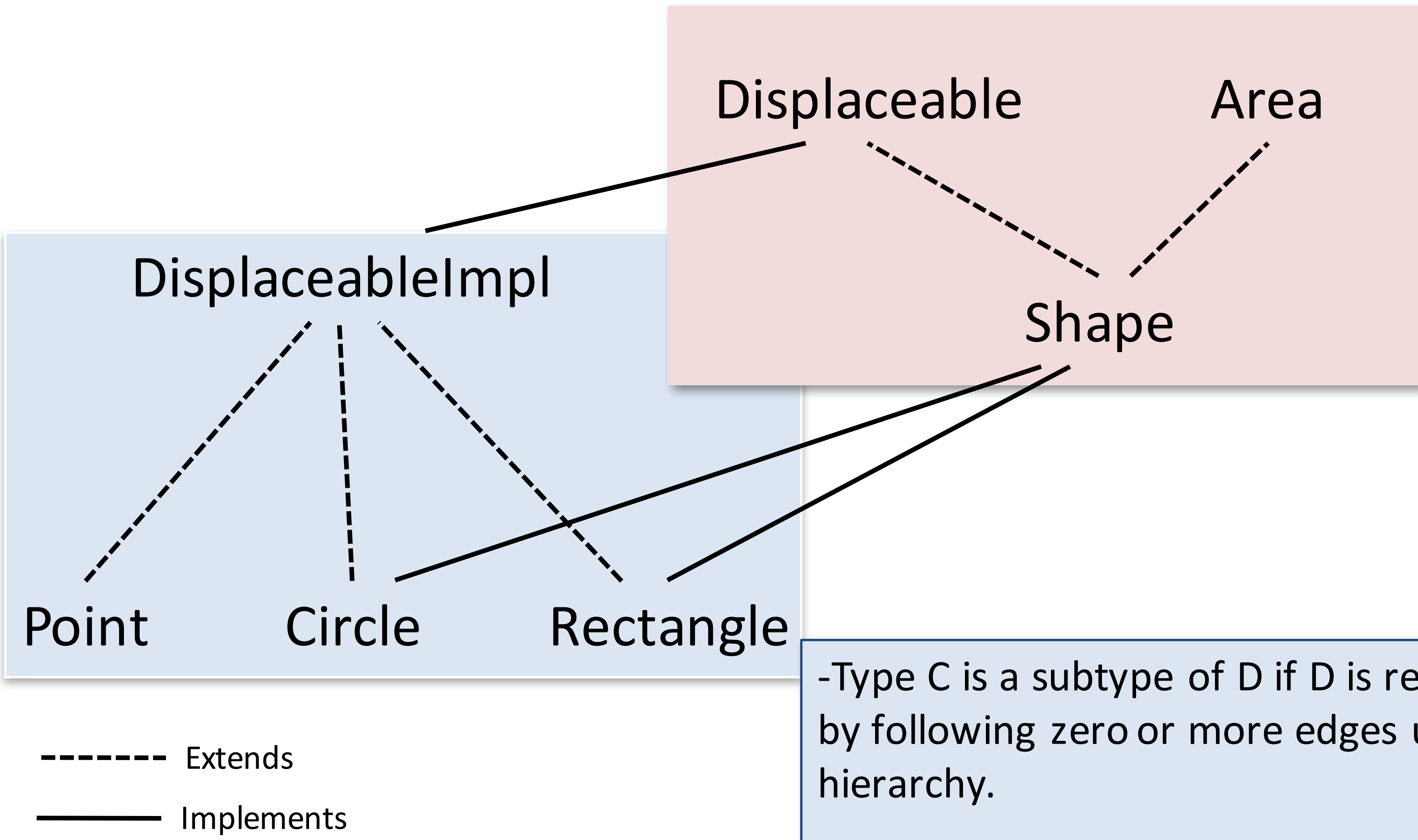
- Classes, like interfaces, can also extend one another.
  - Unlike interfaces, a class can extend only *one* other class.
- The extending class *inherits* all of the fields and methods of its *superclass*, and may include additional fields or methods.
  - This captures the “is a” relationship between 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.

```
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: Point, Circle, and Rectangle have *identical* code for getX(), getY(), and move() methods when implementing Displaceable.

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

- Contructors *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).

```
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 (see Main3.java).

Especially: avoid overriding.