Subtyping; Inheritance and Dynamic Dispatch

Chapter 23 and 24
Announcements

• HW7: Chat Client
  – Available Soon
  – Due: Tuesday, November 13\textsuperscript{th} at 11:59pm

• Upcoming: Midterm 2
  – Friday, November 9\textsuperscript{th} in class

• Covers:
  – 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)
OOooooo programming

OO
Subtypes
Why Static Types?

• Types stop you from using values incorrectly
  – 3 + true
  – (new Counter()).m()

• All expressions have types
  – 3 + 4 has type int
  – “A”.toLowerCase() has type String

• How do we know if x.m() is correct? or x+3?
  – depends on the type of x

• Type restrictions preserve the types of variables
  – assignment "x = 3" must be to values with compatible types
  – methods "o.m(3)" must be called with compatible arguments

• HOWEVER: in Java, values can have multiple types....
• Give a type for an object based on what it *does*, not on how it was constructed
• Describes a contract that objects must satisfy
• Example: Interface for objects that have a position and can be moved

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

No fields, no constructors, no method bodies!
Implementing the interface

- A class that implements an interface provides appropriate definitions for the methods specified in the interface.
- That class fulfills the contract implicit in the interface.

```java
public class Point implements Displaceable {
    private int x, y;
    public Point(int x0, int y0) {
        x = x0;
        y = y0;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```
Another implementation

```java
public class Circle implements Displaceable {
    private Point center;
    private int radius;
    public Circle(int x, int y, int initRadius) {
        Point center = new Point(x, y);
        radius = initRadius;
    }
    public int getX() { return center.getX(); }
    public int getY() { return center.getY(); }
    public void move(int dx, int dy) {
        center.move(dx, dy);
    }
}
```

Objects with different local state can satisfy the same interface

Delegation: move the circle by moving the center
Implementing multiple interfaces

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

public class Circle implements Displaceable, Area {
    private Point center;
    private int radius;
    // constructor
    // implementation of Displaceable methods

    // new method
    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

Classes can implement multiple interfaces by including all of the required methods.
Assume Circle implements the Displaceable interface. The following snippet of code typechecks:

```
// in class C
public static void moveItALot (Displaceable s) {
    // omitted
}

// elsewhere
Circle c = new Circle(new Point(10,10),10);
C.moveItALot(c);
```
Assume Circle implements the Displaceable interface. The following snippet of code typechecks:

```java
// in class C
public static void moveItALot (Displaceable s) {
    // omitted
}

// elsewhere
Circle c = new Circle(new Point(10,10),10);
C.moveItALot(c);
```

1. True
2. False

Answer: True
Types and Subtyping

**Definition**: Type A can be *declared* to be a *subtype* of type B if values of type A can do anything that values of type B can do. Type B is called the *supertype* of A.

**Example**: A class that implements an interface declares a subtyping relationship.
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
Subtype Polymorphism*

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

```java
// in class C
public static void leapIt(Displaceable c) {
    c.move(1000,1000);
}
// somewhere else
C.leapIt(new Circle (p, 10));
```

• If B is a subtype of A, it provides all of A’s (public) methods
• The behavior of a nonstatic method (like move) depends on B’s implementation

*polymorphism = many shapes
Interface Extension – An interface that extends another interface declares a subtype
Class Extension – A class that extends another class declares a subtype
**Interface Extension**

- Build richer interface hierarchies by extending existing interfaces.

```java
public interface Displaceable {
    int getX();
    int getY();
    void move(int dx, int 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.
• 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.
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*, 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.
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.
public class DisplaceableImpl implements Displaceable {
    private int x; private int y;
    public DisplaceableImpl(int x, int y) { ... }
    public int getX() { return x; }
    public int getY() { return y; }
    public void move(int dx, int dy) { x += dx; y += dy; }
}

public class Circle extends DisplaceableImpl implements Shape {
    private int radius;
    public Circle(Point pt, int radius) {
        super(pt.getX(), pt.getY());
        this.radius = radius;
    }
    public double getArea() { ... }
    public Rectangle getBoundingBox() { ... }
}
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: Shapes.zip
Inheritance: Constructors

- Constructors cannot be inherited
  - They have the wrong names!
  - A subclass invokes the constructor of its super class using the keyword super
  - Super must be the first line of the subclass constructor
    - if the parent class constructor takes no arguments, it is OK to omit the call to super

```java
public Circle(Point pt, int radius) {
    super(pt.getX(), pt.getY());
    this.radius = radius;
}
```
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.
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

---

Diagram:

- **Object**
- **DisplaceableImpl**
- **Point**
- **Circle**
- **Rectangle**
- **Displaceable**
- **Area**
- **Shape**

Classes (form a tree)

Interfaces

Extend: solid line
Implement: dashed line
Subtype by fiat: dotted line
Subtype Polymorphism*

- **Main idea:**
  Anywhere an object of type A is needed, an object that is a subtype of A can be provided.

```java
// in class C
public static void leapIt(DisplaceableImpl c) {
    c.move(1000,1000);
}
// somewhere else
C.leapIt(new Circle (10, 10, 10));
```

- If B is a subtype of A, it provides all of A’s (public) methods.
- Due to dynamic dispatch, the behavior of a method depends on B’s implementation.
  - Simple inheritance means B's method is inherited from A
  - Otherwise, behavior of B should be “compatible” with A’s behavior

*polymorphism = many shapes*
Subtyping and Variables

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

```java
Displaceable a = new Circle(new Point(2,3), 1);
```

supertype of Circle    subtype of Displaceable

• Methods with parameters of type A must be called with arguments that are subtypes of A
Object

• Object is the root of the class tree
  – Classes that leave off the “extends” clause *implicitly* extend Object
  – Arrays also implement the methods of Object
  – This class provides methods useful for *all* objects to support

• Object is the highest type in the subtyping hierarchy.
"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 \texttt{a1} on line A?

```
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);
```
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);
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);
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

---

**Static type vs. Dynamic type**
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);
```

Area

Circle

None of the above

Not well typed
Static type vs. Dynamic type

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