Oggetti, tipi e sottotipo

Uso dei tipi (statico)

- Tipi come utile meccanismo per avere delle buone garanzie di uso corretto
  - 3.m()
  - if (3) { return 1; } else { return 2; }
  - 3 + true
- Le espressioni hanno un tipo
  - 3+4 ha tipo int
  - “A”.ToLowerCase ha tipo string

Tipi

- In quali casi x.m() (oppure x+3) e' corretta?
  - Dipende dal tipo della variabile x
  - Questo e' il motivo per cui esistono le dichiarazioni
- Restrizioni di tipo
  - X= 3 (assegnamento richiede tipi compatibili)
  - Obj.m(3) la chiamata del metodo richiede tipi compatibili
- Java, ML valori hanno tipi multipli

Sottotipo

- DEF: Il tipo A e' un sottotipo del tipo B se A puo' fare tutto quello che puo' fare B. B viene anche detto supertipo di A

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

public class Circle implements Area {
    private double r;
    private Point p;
    public Circle (double x0, double y0, double r0) {
       r = r0; p = new Point(x0,y0);
    }
    public double getArea ()  {
       return 3.14159 * r * r;
    }
    public double getRadius () { return r; }
}
```

Una variabile dichiarata di tipo A puo' memorizzare un qualunque oggetto che ha un sottotipo di A

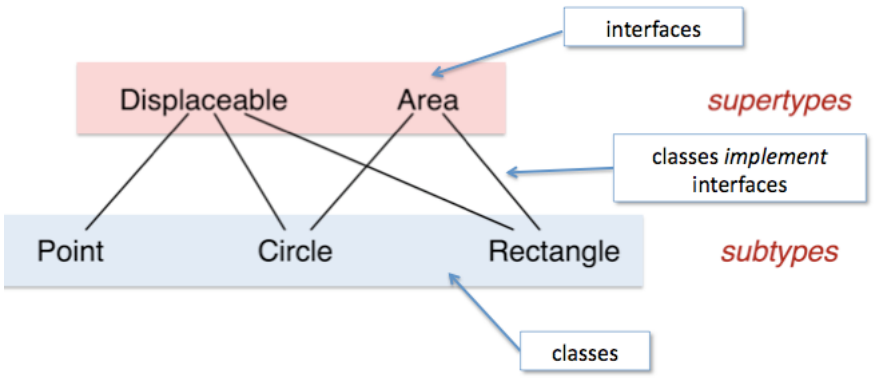
```
Area a = new Circle(1, new Point(2,3));
```

supertype of Circle

subtype of Area

Metodi con parametri di tipo A devono essere invocati con argomenti che sono sottotipi di A

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



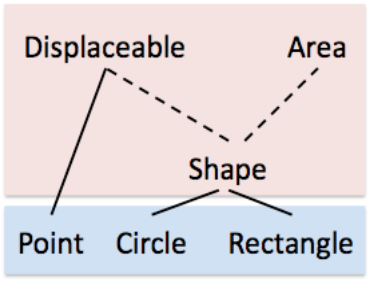
Static vs dynamic

- Il tipo statico di una espressione descrive quello che si conosce dell'espressione a tempo di compilazione
  - Area X
- Il tipo dinamico di un oggetto descrive la classe associata all'oggetto a tempo di esecuzione
  - X = new Circle(2,3)
- Ocaml: static types
- Java dynamic classes (ma il tipo dinamico e' sempre un sottotipo di quello statico)

```
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.



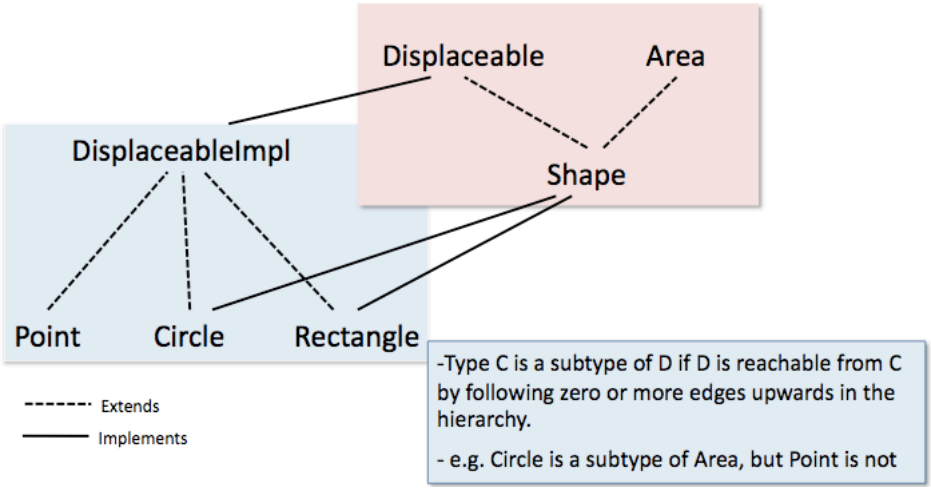

```
class Point implements Displaceable {
  … // omitted
}
class Circle implements Shape {
  … // omitted
}
class Rectangle implements Shape {
  … // omitted
}
```

Shape e' un sottotipo di Displaceable e Area

Circle e' un sottotipo di Shape (e di Area?)

- 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); }
}
```



Metodo Costruttore

Cosa succede al metodo costruttore quando avviene l'ereditarieta'?

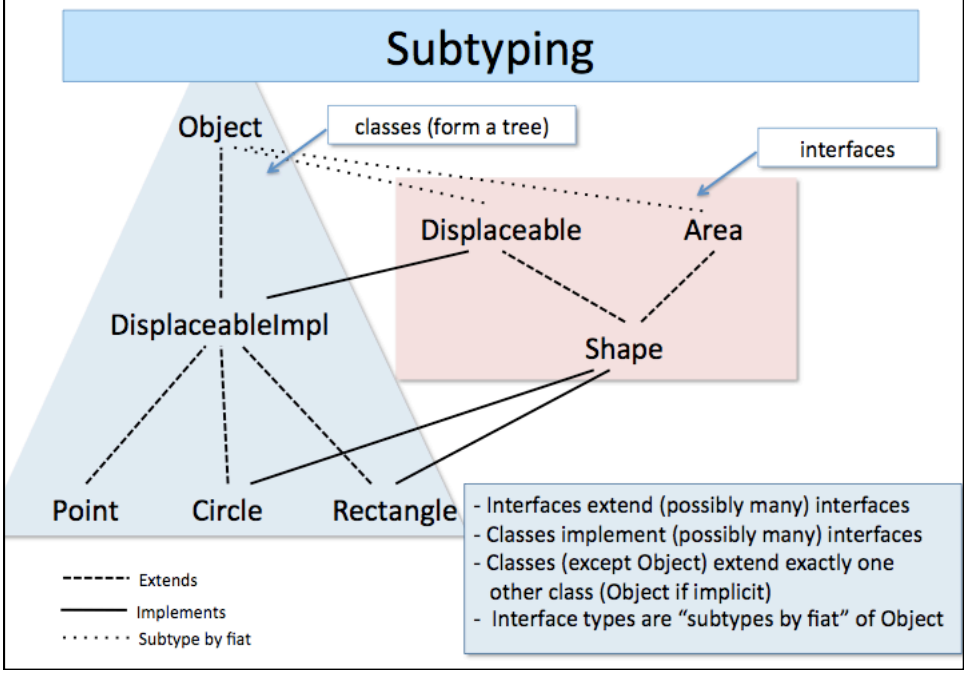
```
class D {
  private int x;
  private int y;
  public D (int initX, int initY) { x = initX; y = initY; }
  public int addBoth() { return x + y; }
}

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); }
}
```

Object

```
public class Object {
    boolean equals(Object o) {
        … // test for equality
    }
    String toString() {
        … // return a string representation
    }
    …  // other methods omitted
}
```

- 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 (override these!)
- Object is also the top type of the subtyping hierarchy.

Subtyping



Subtype Polymorphism

```
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();                 // What type for A?
System.out.println(x.toLowerCase()); // Does this work?
q.enq(new Point(0.0,0.0));
___B___  y = q.deq();
```

Generics (Parametric Polymorphism)

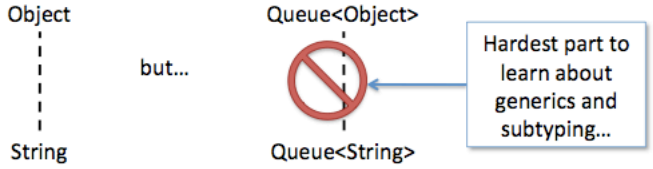
```
public interface Queue<E> {
  public void enq(E o);
  public E deq();
  public boolean isEmpty();
  public boolean contains(Object o);
  …
}
```

```
Queue<String> q = …;

q.enq("CIS 120");
___A___  x = q.deq();                 // What type for A?
System.out.println(x.toLowerCase()); // Does this work?
q.enq(new Point(0.0,0.0));           // What about this?
___B___  y = q.deq();
```

Subtyping and Generics

```
Queue<String> qs = new QueueImpl<String>();     // OK
Queue<Object> qo = qs;                          // OK?

qo.enq(new Object());
String s = qs.deq(0);                    // oops!
```

- Java generics are *invariant*:
  - Subtyping of *arguments* to generic types does not imply subtyping between the instantiations:



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:

```
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