Polymorphism

References & Inheritance

- A reference can refer to any object of that type.
- eg. `Shape s;`
  - Now, `s` can refer to any shape
- …but classes that inherit `Shape` are also shapes.
  - Remember the "is-a" restriction on the design.
  - So, `s` could also refer to a `Circle` or `Rectangle`.
  - A big part of why we insist on the "is-a" relationship

References & Inheritance

- Perfectly legal:

```
Shape s1;
Rectangle r1 = new Rectangle(…);
s1 = r1;  // all rectangles are shapes
Shape s2 = new Rectangle(…);
Object o = new Circle(…);
```

- Not okay:

```
Shape s = new Circle(…);
Rectangle r = s; // not all shapes are
                 // rectangles.
```

Compile-Time Checking

- Validity of assignments is checked when the program **compiles**, not while it's running.
  - Lets programs run faster; strong typing lets the compiler catch problems immediately.
- Causes non-obvious restriction:

```
Shape s = new Rectangle(…);
…
Rectangle r = s; // not all shapes are
                 // rectangles.
```

Is `s` still a Rectangle?

Casting

- If you really need to assert a more specific type, the reference can be cast to the proper type:

```
Shape s = new Rectangle(…);
Rectangle r = (Rectangle) s;
```

- Casts should be avoided if possible.
  - unavoidable before Java 5.0 since all collections contained only `Object` references
  - `myArrayList.get(0)` would **always** return an Object and need to be cast to its real type.

Which Method?

- Consider…

```
Shape s = new Rectangle(…);
String myOutput = s.toString();
```

- If `toString()` was defined in `Shape` and overridden in `Rectangle`, which definition is used?
  - `s` is a `Shape` reference: use the one from `Shape`.
  - `s` actually refers to a `Rectangle` instance: use the one from `Rectangle`.

Which Method?

- Look at it this way:

- The reference is followed to the actual instance.
  - The reference is just pointing the to thing we use.
- It's the instance's method that gets used.
  - So in the example, we use the `toString` from `Rectangle`.

Why?

- Lets you used a more generic reference when needed, but still get the right method.

```
public void draw(Collection<Shape> shapes) {
    for(Shape s: shapes) {
        s.draw();
    }
}
```

- Different shapes will have different draw methods (they look different after all)
  - … but this will use the right one in each case.

Interfaces, again

- Implementing an interface is very similar to inheriting a class.

```
class MyClass implements AnInterface { … }
```

  - … this takes everything from `AnInterface` and puts it into `MyClass`, just like inheriting.
  - Except, **all** of the methods must be implemented here.
  - No previous implementations to fall back on.

Interfaces vs. Abstract Classes

- Similarities:
  - Neither can be instantiated.
  - Both can be used as a starting-point for a class.
- Differences:
  - A class can contain implementations of methods.
  - A class can implement many interfaces, but can only inherit one class.
    - … in Java. Other languages allow multiple inheritance and have no interfaces.

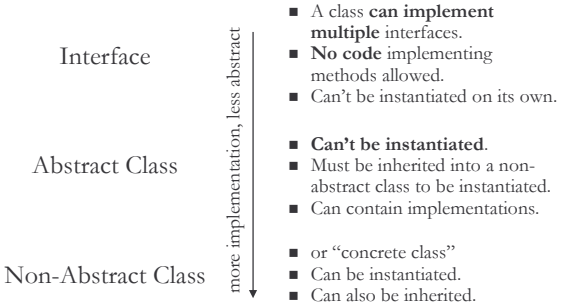
Example with Interfaces

- Also works with interfaces:

```
public boolean startsWith(List<int> list,
        int val) {
    return list.get(0)==val;
}
```

- Whether the argument is an `ArrayList`, `Vector`, or `LinkedList`, this function works.
  - uses the `.get()` method appropriate to the underlying implementation.

Comparison



Similarities

Interface

Abstract Class

Non-Abstract Class

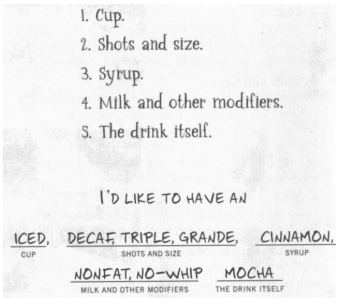
- You can **declare a reference** to any of these.
- Any **object that implements/ inherits** the reference type can be used for that reference.
- Any method that is defined by the **reference type** can be used.
- The **implementation in the actual instance** will be used when its called.

Example: Coffee

The Problem

- Ordering at a coffee bar.
  - Espresso, mochas, low-fat milk, …
  - Lots of structure, many specific subtypes to make use of inheritance.
- Suppose we are creating a point-of-sale system and have to represent the orders taken by the cashier.
  - They will be passed on to those making the drinks.

How hard is this?



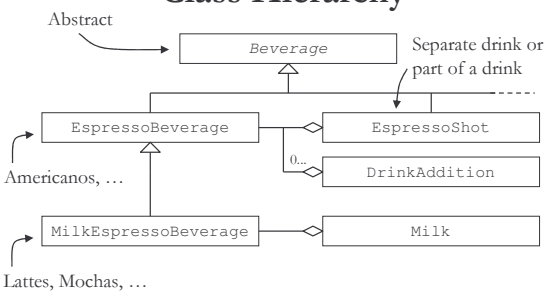
Stuff to Represent

- Cup (to-go, for-here, iced, personal cup)
- Size (small, medium, large)
- Shots (1 or more espresso shots, caf/decaf)
  - default determined by size, but can be changed
- Syrups (0 or more flavour shots)
- Milk, if applicable (whole, 2%, skim, soy)
- Toppings (whipped cream, caramel, …)
- Drink (espresso shot, Americano, mocha, …)

Other Stuff

- There's also a lot of other stuff that doesn't fall into these categories.
  - blended drinks, juice, teas, …
- We should make it possible to represent these too.
  - … but won't implement.

Class Hierarchy



Implementation…