Polymorphism

- Literally: the ability to assume *many forms*
- OOP idea: a superclass reference can refer to many types of subclass objects
  - Each object may behave differently – if subclasses override methods
- Imagine a Shape class with a draw()method
  - Then subclasses Circle, Triangle, … all override draw()
- Depends on dynamic method binding
  - i.e., actual method is chosen at execution-time
  - A.k.a. late-binding – unlike static or final methods

Overriding `Object` methods

- All Java classes inherit methods from `Object`
  - But `Object` implementation is crude, so override
- `toString()` – "classname@hashcode" in `Object`
  - e.g., to override in `BankAccount`:

```
public String toString()
{ return "BankAccount[balance="+balance+"]"; }
```

- `equals(Object other)` – same object in `Object`
  - Usually want to change to same contents
  - And means should also override `hashCode()`
- `clone()` – `Object` makes a shallow copy
  - i.e., just copies references of instance variables

Further abstraction

- Abstract classes
  - have one or more `abstract` methods,
    - e.g.,

```
abstract class Shape { ...
    abstract void draw();
  ... }
```

    - Subclasses of `Shape` *must* implement `draw()`
  - Cannot instantiate – not concrete classes, so no such object
    - But often have constructor for subclass constructors to invoke
    - i.e., all `Shape` objects are objects of one of `Shape`'s subclasses
  - Can *refer* to objects as `Shape` – then know they can `draw()`
  - e.g., Shape Demo from old CS 5JA class
- Subclasses inherit implementation *and* interface

Interfaces (completely abstract)

- A Java `interface` has no implementation at all
  - `interface`: defines the messages a class responds to if the class implements the interface
  - e.g., "… `implements Comparable`" means the class responds to `compareTo(Object other);`
- e.g., don't extend `Shape`, implement `Drawable`:

```
interface Drawable
{ void draw(Graphics g); }
```

- A class may implement multiple interfaces
  - Not really "is a" – more aptly "can refer to as a" – e.g.:
    - `class Box implements Drawable, Comparable`
    - Now can use Drawable or Comparable reference with a Box

```
Box b = ...; Drawable d = b; d.draw(g);
```

More on interfaces

- All methods are `public abstract` – omit explicit modifiers by convention
- Constants okay too
  - All `public static final` – omitted by convention
  - Must be initialized when declared, of course
- Can extend, just like classes
  - But okay to extend more than one:

```
public interface SerializableRunnable
        extends java.io.Serializable, Runnable
```

- Tend to be much more flexible than classes
  - So they are the basis of many "design patterns" (CS 50 topic)

Abstraction/inheritance notes

- Encapsulate common traits by superclasses
  - Use polymorphism to affect uniqueness
- "Program to the interface" (not the implementation)
  - i.e., practice information hiding – *what* a class does is important, *not how* it does it
    - Best just to share the Javadocs with other programmers!
  - So it's no big deal if implementation changes
- Sometimes "is a" not best – too much coupling
  - Try "has a" instead (composition, not inheritance)
  - Or pure interface approach – Measurable.java (p. 389) Decoupling with an interface (Chapter 9, Figure 1)

Nested classes and interfaces

- Okay to define a class (or interface) inside another class (or interface)
  - Good for grouping logically related types
- Static nested class – work just like non-nested
  - Can extend, or be extended like any other class
  - e.g., private class Entry in java.util.LinkedList.java
- Inner class – non-static nested type
  - Objects are *associated with an instance* of outer type – the "enclosing object"
  - Both classes can share data – even `private`

More nested classes/interfaces

- Local inner classes – inside methods (or other blocks)
  - Not members of the class – local to the block
    - May access any fields – but just final local variables
  - See implementation of Measurer.java (p. 398-402) Even more decoupling (Chapter 9, Figure 2)
- Can even have anonymous inner classes
  - Extend a class or implement existing interface
  - Easily applicable to RectangleMeasurer example

Exception handling

- *Necessary* for reading/writing most streams
  - Also for using threads, networks, …
- And best way to treat *exceptional* situations
- Basic idea – if a method detects an exceptional situation, the method either handles it or throws it to a competent handler
  - Throwing sends it to the caller (next on stack). Then caller can throw it again, or handle it, and so on.
  - Handling an exception means catching it, and doing something about it

What is an **`Exception`**?

- Ans: instance of `Exception` (or one of its subclasses)
  - Specific feature: an object you can `throw`
  - Purpose: to signal an exceptional situation
  - Effect: terminates and writes message – *unless* you `catch` it on the way up the call chain
- Some exceptions are checked by the compiler
  - These *must* be handled or the method must declare it throws the exception in the header
    - Includes IOException and subclasses
  - Note: the most typical exceptions are unchecked

Easy to define a new exception

- First note: lots of good exceptions ready to use (API)
- Or can easily define new by extending existing one

```
public class MyException extends RuntimeException
{ // Note: a RuntimeException is unchecked – so is often a good choice
    public MyException(String message)
    {   super(message);   }
}
```

- Now okay to `throw new MyException("...");` or `catch(MyException e) { ... }`

`try`

- Denotes blocks of code that might throw exceptions
- Usually followed by one or more catch clauses
  - These identify the exceptions they will catch
  - Are checked in order – just one will execute
    - Exception hierarchy is important – always check subclasses first, or superclass will catch it first
- Also `finally` – optional clause always executes

```
try { ... // something that might throw exception }
catch(exception-type et) { ... }
catch(different-exception-type det) { ... }
finally { ... // executes no matter what }
```