The object contract

Abstract classes
Abstract Vs. Concrete
Reference type Vs. Object type
Multiple Inheritance
Interfaces
The object contract

- The superclass defines how the objects are to be used (the contract).
  - Functionality can be overridden, but the method name/purpose defines the contract.
- Example: Say we wish to keep track of a bunch of 2D shapes to bounce around a JPanel.
- All shapes need: constructor, draw(), move()
- constructor and move() is the same for all shapes
- draw() is different code for each shape
- Polymorphism allows us to treat all objects similarly
  - object.move()
  - object.draw()
- This is the “contract” – how to use the object
  - The contract is the same for all subclasses
Polymorphism as a contract: Using the Shape Class

Roll your own.

```java
... ArrayList<Shape> shapeList;
...
public void makeShapes () {
    shapeList = new ArrayList<Shapes>();
    shapeList.add(new Circle(0,0,5,3,Color.Red));
    shapeList.add(new Square(10,10,-1,-5,Color.Green));
} // end makeShapes
...

public void paintComponent (Graphics g) {
    for (Shape shape : shapeList) {
        shape.move();
        shape.draw();
    }
} // end method paintComponent
...
How do I define Shape?

- What should the default code be for draw()?
- What should happen if I try to make a new Shape()?
  - What would it look like?

```java
... 
ArrayList<Shape> shapeList; 
...
public void makeShapes () { 
    shapeList = new ArrayList<Shapes>(); 
    shapeList.add( 
        new Circle(0,0,5,3,Color.Red)); 
    shapeList.add( 
        new Shape(10,10,1,-5,Color.Green)); 
} // end makeShapes 
... 
```
Abstract classes

- Some classes should *never* be instantiated
  - Some superclasses exist *only* to define a contract
  - You may *never* want to allow anyone to create one!
- Marking a class *abstract* tells the compiler to create an instance of the class. Its only use is to be extended!

```java
abstract class Shape {
    Shape (int x, int y, int dx, int dy, Color color) {
        ... // my code to inherit here
    } // end constructor Shape
    void move () {
        ... // my code to inherit here
    } // end method move
    void draw () {
        ... // what should go here?
    } // end method draw
} // end class Shape
```

```
Shape
Shape (x, y, dx, dy, color)
move()
draw()

Triangle
draw ()

Circle
draw ()

Square
draw ()
```
Abstract Vs. Concrete classes

- **Abstract** class: A class that cannot be instantiated
  - An abstract class has virtually no use, no value, and no purpose
  - It can have static members
  - It can define a contract (this is, it can be extended)

- **Concrete** class: A class that is not abstract
  - The objects created and doing the work at runtime are concrete
  - These objects may be *instances* of a subclass of an abstract class

**UML Note:**
Use italics to indicate abstract

```
Shape
Shape (x, y, dx, dy, color)
draw()
mov()   
```

```
<table>
<thead>
<tr>
<th>Triangle</th>
<th>Circle</th>
<th>Square</th>
</tr>
</thead>
<tbody>
<tr>
<td>draw ()</td>
<td>draw ()</td>
<td>draw ()</td>
</tr>
</tbody>
</table>

Wright State University, College of Engineering
Dr. T. Doom, Computer Science & Engineering

CS 241
Computer Programming II
Abstract methods

- Some methods in an abstract class may have no reasonable default
  - How do I draw a “Shape”? As a circle? A triangle?
- Marking a method abstract forces subclasses to implement/override it
  - An abstract methods can only exist in an abstract class
  - Defines contract, not implementation

```java
abstract class Shape {
    Shape (int x, int y, int dx, int dy, Color color) {
        ... // my code to inherit here
    } // end constructor Shape
    void move () {
        ... // my code to inherit here
    } // end method move
    abstract void draw (); // no body!
        // Subclass MUST define function
} // end class Shape
```

Diagram:

```
Shape

Triangle
draw ()
Circle
draw ()
Square
draw ()
```
Polymorphism means “Many Forms”

- How does the *compiler* know what members are/are not available?

```java
... Triangle t = new Triangle(); Shape s = (Shape) t; Object o = (Object) t; ...
```
The *compiler* cares about the type of the reference variable, not the class of the actual object at the other end of the reference (unknown till runtime!)

```java
Triangle t = new Triangle();
Shape s = (Shape) t;
Object o = (Object) t;
... 
```

- Compiler Error
  - `o.move()`
  - `s.rotate()`
- OK – resolved at runtime
  - `t.rotate()`
  - `(Triangle) o.rotate()`
  - `t.draw()`
  - `s.draw()`
- You can call a method on an object *only* if the class of the reference variable has that method!
The object contract

- For an object of type Triangle
  - Everything accessible in class Triangle defines part of your contract
  - Everything accessible in class Shape defines part of your contract
  - Everything accessible in class Object defines part of your contract

- Accessible: public or protected

Diagram:

```
Triangle
  draw()
  rotate()
  Shape()
    move()
      draw()
  Object()
    equals (object)
    toString()
    clone (object)
    getClass()
    ...

Shape
- Shape (x, y, dx, dy, color)
- move()
- draw()

Triangle
- draw()
- rotate()

Circle
- draw()

Square
- draw()
- rotate()
```
What if I want to create an object that fulfills the contracts of two separate (unrelated) existing classes?

This would allow us to use all the existing code that works for either method.

Example: GUI_Card
   - It’s a card, and can be used with existing Cardgame code
   - It also needs to be displayed on the GUI, and is essentially just a Square

We want the following to be true:
   - GUI_Card IS-A Card
   - GUI_Card IS-A Square
Multiple inheritance: Here be dragons!

- What if both superclasses have a member with the same name?
- How do we avoid ambiguity?
  - Extra syntax?
  - How do we help check for errors?
  - Basically, this is hard

- Some languages allow this (C++, for one)
  - C++ is all about handling hard
- Some languages don’t (Java, for one)
  - Java is all about the simple
- Java does provide limited multiple inheritance

Which draw() superclass method do I inherit?!?
Interfaces

- Java allows multiple inheritance to *interface* classes
- Interface classes are:
  - 100% pure abstract class
  - They contain *only* abstract, public methods
- Thus there is no ambiguity. The interface defines the *contract* but forces the subclass object to define the one and only one implementation.

```java
public interface Shape {
    public abstract void move();
    public abstract void draw();
} // end interface Shape

public class GUI_Card extends Card implements Shape {
    // Implementation must override move/draw
} // end class GUI_Card
```
When do you make a class, a subclass, an abstract class, or an interface?

- **Class**
  - Appropriate for objects that don’t extend anything
  - Fails the IS-A test for all other types (except Object)

- **Subclass (extend a class)**
  - Only when you need to make a more specific version of a class
  - Override or add new behaviors

- **Abstract class**
  - Define a template/contract for a group of subclasses
  - Has some implementation code that all subclasses can use
  - Guarantee that nobody can make an object of that type

- **Interface**
  - Define a role that other classes can play, regardless of their type