Inheritance and Polymorphism, Part 2

- Abstract Classes and Methods
- Multiple Inheritance
- Interfaces
- Inner Classes
Abstract Class and Method

• An *abstract class* is a class with an abstract method.
• An *abstract method* is a method with no body, i.e., only declared but not defined.

• It is not possible to make instances of abstract classes.
• Abstract method are defined in subclasses of the abstract class.
Abstract Class and Method, Example

Abstract class C1 with abstract methods A and B

Abstract class C2. Defines method A but not method B. Adds data elements d3 and d4

Concrete class C3. Defines method B. Adds the methods D and E and the data element d5.
Abstract Classes in Java

abstract class ClassName {
    // <class body>
}

- Classes with abstract methods must be declared abstract.
- Classes without abstract methods can be declared abstract.
- A subclass to a concrete superclass can be abstract.
- Constructors can be defined on abstract classes.
- Instances of abstract classes cannot be made.

- Abstract fields not possible.
Abstract Class in Java, Example

// [Source: Kurt Nørmark]
public abstract class Stack{

    abstract public void push(Object el);
    abstract public void pop();
    abstract public Object top();
    abstract public boolean full();
    abstract public boolean empty();
    abstract public int size();

    public void toggleTop(){
        if (size() >= 2){
            Object topEl1 = top();  pop();
            Object topEl2 = top();  pop();
            push(topEl1); push(topEl2);
        }
    }

    public String toString(){
        return "Stack";
    }
}
Abstract Methods in Java

Abstract [access modifier] return type
methodName([parameters]);

- A method body does not have to be defined.
- Abstract method are overwritten in subclasses.
- Idea taken directly from C++

- You are saying: The object should have this properties I just do not know how to implement the property at this level of abstraction.
Abstract Methods in Java, Example

```java
public abstract class Number {
    public abstract int intValue();
    public abstract long longValue();
    public abstract double doubleValue();
    public abstract float floatValue();
    public byte byteValue(){
        // method body
    }
    public short shortValue(){
        // method body
    }
}
```
Multiple Inheritance, Example

- For the teaching assistant when want the properties from both Employee and Student.

```
Person
  - name()
  - cpr()

Employee
  - salary()
  - degree()

Student
  - gpa()
  - courses()

Teaching A.
```
Problems with Multiple Inheritance

- Name clash problem: Which `department` does `ta` refers to?
- Combination problem: Can `department` from Employee and Student be combined in Teaching Assistant?
- Selection problem: Can you select between `department` from Employee and `department` from Student?
- Replication problem: Should there be two `departments` in TeachingAssistant?
Multiple Classifications

- Multiple and overlapping classification for the classes X and Y, i.e.,
  - class X is Runnable and Comparable
  - class Y is Runnable, Storable, and Cloneable
Java's **interface** Concept

```
Shape
  draw()
  resize()

Circle
  draw()
  resize()

Line
  draw()
  resize()

Rectangle
  draw()
  resize()

Square
  draw()
  resize()
```

Interface

```
  extends

  implements
```
Java's `interface` Concept, cont.

```java
public interface Shape {
    double PI = 3.14;   // static and final => upper case
    void draw();        // automatic public
    void resize();      // automatic public
}

public class Rectangle implements Shape {
    public void draw() {System.out.println("Rectangle"); }
    public void resize () {} }
}

public class Square extends Rectangle {
    public void draw() {System.out.println("Square"); }
    public void resize () {} }
}
Java's `interface` Concept

- An `interface` is a collection of method declarations.
  - An interface is a class-like concept.
  - An interface has no variable declarations or method bodies.

- Describes a set of methods that a class can be forced to implement.
- An interface can be used to define a set of "constants".
- An interface can be used as a type concept.
  - Variable and parameter can be of interface types.
- Interfaces can be used to implement multiple inheritance like hierarchies.
Java's `interface` Concept, cont.

```java
interface InterfaceName {
    // "constant" declarations
    // method declarations
}

class ClassName implements InterfaceName {
    ...
}

class ClassName extends SuperClass implements InterfaceName {
    ...
}

class ClassName extends SuperClass implements InterfaceName1, InterfaceName2 {
    ...
}

interface InterfaceName extends InterfaceName {
    ...
}
```
Semantic Rules for Interfaces

• Type
  - An interface can be used as a type, like classes
  - A variable or parameter declared of an interface type is polymorph
    - Any object of a class that implements the interface can be referred by the variable

• Instantiation
  - Does not make sense on an interface.

• Access modifiers
  - An interface can be public or "friendly" (the default).
  - All methods in an interface are default abstract and public.
    - Static, final, private, and protected cannot be used.
  - All variables ("constants") are public static final by default
    - Private, protected cannot be used.
The **Iterator** Interface

- The **Iterator** interface in the package `java.util` is a basic iterator that works on collections.

```java
define package java.util;
define public interface Iterator {
    // public abstract boolean hasNext()
    boolean hasNext();
    Object next();
    void remove(); // optional
}
```
The **Iterator** Interface, cont

```java
// declare a variable of Interface type
Iterator iter = myShapes.iterator();
while (iter.hasNext()) {
    Shape s = (Shape) iter.next();
    s.draw();
}
```

- Note the *cast (Shape)* since *Collection* and *Iterator* manage *Objects*.
- When a collection has a natural ordering, *Iterator* will respect it.
The **Iterator** Interface, cont

```java
// ArrayList covered in Collections lecture
ArrayList al = new ArrayList();
for (int i = 0; i < 8; i++) {
    al.add (new Integer(i));
}
Iterator iter = al.iterator();
while (iter.hasNext()) {
    Integer i = (Integer) iter.next();
    System.out.println (i);
}
```

- Put this in `main` method and execute it.
- Note the again the cast `(Integer)s`.
The **Cloneable** Interface

- A class X that implements the **Cloneable** interface tells clients that X objects can be cloned.

- The interface is empty, i.e. has no methods.

- Returns an identical copy of an object.
  - A *shallow copy*, by default.
  - A *deep copy* is often preferable.

- Prevention of cloning
  - Necessary if unique attribute, e.g., database lock or open file reference.
  - Not sufficient to omit to implement **Cloneable**.
    - Sub classes might implement it.
  - **clone** method should throw an exception:
    - **CloneNotSupportedException**
The Cloneable Interface, Example 1

// Car example revisited
public class Car implements Cloneable {
    private String make;
    private String model;
    private double price;
    // default constructor
    public Car() {
        this ("", ",", 0.0);
    }
    // give reasonable values to instance variables
    public Car(String make, String model, double price){
        this.make = make;
        this.model = model;
        this.price = price;
    }
    // the Cloneable interface
    public Object clone() {
        return new Car (this.make, this.model, this.price);
    }
}
package geometric; // [Source: java.sun.com]

/** A clonable Point */
public class Point extends java.awt.Point implements Cloneable {
    // the Cloneable interface
    public Object clone() {
        try {
            return (super.clone()); // protected in Object
        } catch (CloneNotSupportedException e) {
            return null;
        }
        // must catch exception will be covered later
        public Point(int x, int y) {
            super(x, y);
        }
    }
}
The **Serializable** Interface

- A class X that implements the **Serializable** interface tells clients that X objects can be stored on file or other persistent media.
- The interface is empty, i.e., has no methods.
# Interface vs. Abstract Class

<table>
<thead>
<tr>
<th>Interface</th>
<th>Abstract Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Methods can be declared.</td>
<td>• Methods can be declared</td>
</tr>
<tr>
<td>• No method bodies</td>
<td>• Method bodies can be defined</td>
</tr>
<tr>
<td>• “Constants” can be declared</td>
<td>• All types of variables can be declared</td>
</tr>
<tr>
<td>• Has no constructor</td>
<td>• Can have constructors</td>
</tr>
<tr>
<td>• Multiple inheritance possible.</td>
<td>• Multiple inheritance not possible.</td>
</tr>
<tr>
<td>• Has no top interface.</td>
<td>• Always inherits from <strong>Object</strong>.</td>
</tr>
<tr>
<td>• Multiple &quot;parent&quot; interfaces.</td>
<td>• Only one &quot;parent&quot; class</td>
</tr>
</tbody>
</table>
Interfaces and Classes Combined

• By using interfaces objects do not reveal which classes the belong to.
  ▪ With an interface it is possible to send a message to an object without knowing which class(es) it belongs. The client only know that certain methods are accessible
  ▪ By implementing multiple interfaces it is possible for an object to change role during its life span.

• Design guidelines
  ▪ Use classes for specialization and generalization
  ▪ Use interfaces to add properties to classes.
## Multiple Inheritance vs. Interface

<table>
<thead>
<tr>
<th>Multiple Inheritance</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Declaration and definition is inherited.</td>
<td>• Only declaration is inherited.</td>
</tr>
<tr>
<td>• Little coding to implement subclass.</td>
<td>• Must coding to implement an interface.</td>
</tr>
<tr>
<td>• Hard conflict can exist.</td>
<td>• No hard conflicts.</td>
</tr>
<tr>
<td>• Very hard to understand (C++ close to impossible).</td>
<td>• Fairly easy to understand.</td>
</tr>
<tr>
<td>• Flexible</td>
<td>• Very flexible. Interface totally separated from implementation.</td>
</tr>
</tbody>
</table>
Inner Classes

- Fundamental language feature, added in Java 1.1.
- Used a lot in JFC/Swing (GUI programming).
- Nest a class within a class.
- Class name is hidden.
- More than hiding and organization
  - Call-back mechanism.
  - Can access members of enclosing object.
Inner Classes, Example

```java
public class Parcel1 { // [Source: bruceeckel.com]
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
        Parcel1 p = new Parcel1();
        p.ship("Tanzania");
    }
}
```
Interfaces and Inner Classes

- An outer class will often have a method that returns a reference to an inner class.

    // [Source: bruceeckel.com]

    public interface Contents {
        int value();
    }

    public interface Destination {
        String readLabel();
    }
Interfaces and Inner Classes, cont

```java
public class Parcel3 { // [Source: bruceeckel.com]
    private class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
    public Destination dest(String s) {
        return new PDestination(s);
    }
    public Contents cont() {
        return new PContents();
    }
}
```
class Test { // [Source: bruceeckel.com]
    public static void main(String[] args) {
        Parcel3 p = new Parcel3();
        Contents c = p.cont();
        Destination d = p.dest("Tanzania");
        // Illegal -- can't access private class:
        //! Parcel3.PContents pc = p.new PContents();
    }
}
Anonymous Inner Classes, Example

- When a class in only needed in one place.
- Convenient shorthand.
- Works for both interfaces and classes.

```java
// [Source: bruceeckel.com]

class Parcel6 {
    public Contents cont() {
        return new Contents() {
            private int i = 11;
            public int value() { return i; }
        };
    }

    public static void main(String[] args) {
        Parcel6 p = new Parcel6();
        Contents c = p.cont();
    }
}
```

Note the semicolon
Why Inner Classes?

• Each inner class can independently inherit from other classes, i.e., the inner class is not limited by whether the outer class is already inheriting from a class.

• With concrete or abstract classes, inner classes are the only way to produce the effect of "multiple implementation inheritance"
Summary

• Purpose: Interfaces and abstract classes can be used for program design, not just program implementation [Meyer pp 239 ff].

• Abstract classes
  ▪ Complete abstract class no methods are abstract but instatiation does not make sense.
  ▪ Incomplete abstract class, some method are abstract.

• Java only supports single inheritance.

• Java "fakes" multiple inheritance via interfaces.
  ▪ Very flexible because the object interface is totally separated from the objects implementation.

• Classes can be nested in Java
  ▪ Name inner classes
  ▪ Anonymous inner classes