Varieties of Polymorphism  (B12.1)

• pure polymorphism
  – one function body, several interpretations
• overloading /ad hoc polymorphism
  – many function bodies with the same name
• overriding
  – general method in superclass
  – subclasses may provide a method with the same name and parameters
• deferred methods / abstract methods
  – defined in superclass
  – implemented in subclasses
Overloading (B12.3)

• several methods with same name, different parameters
• code executed depends on arguments given
• varieties:
  - ad hoc overloading
    • same method name in 2 or more classes unrelated by inheritance
    • e.g. `isEmpty` for `LinkedList`, `Rectangle`
  - parametric overloading
    • multiple methods with name within a single class
    • e.g. `constructors` for `Rectangle`
Overriding
(B12.4)

• general method defined in superclass
• subclass defines new body with equivalent semantics
• 2 varieties
  - replacement semantics
    • completely new body
  - refinement semantics
    • also executes superclass method
      - addCard in DiscardPile
• constructors always use refinement semantics
  - DeckPile constructor
  - default constructor
class DiscardPile extends CardPile {
    public void addCard(Card aCard) {
        if (!aCard.faceUp())
            aCard.flip();
        super.addCard(aCard);
    }
}

class DeckPile extends CardPile {
    DeckPile(int x, int y) {
        super(x, y);
        for(int i = 0; i < 4; i++)
            for(int j = 0; j <= 12; j++)
                addCard(new Card(i, j));
        Collections.shuffle(thePile);
    }
}
Abstract Methods (B12.5)

• abstract methods
  - also called deferred
  - essentially null implementation that must be overridden in subclass
• interfaces
  - all methods are abstract
• e.g. abstract class Shape
  - defines a draw method even though cannot provide any reasonable implementation
  - subclasses Circle, Triangle & Square implement draw
  - can have a polymorphic variable of type Shape that can perform draw
Pure Polymorphism (B12.6)

- multiple effects by deferring to subclass
  - e.g. `byteValue` in `Number`

```java
public abstract class Number {
    public abstract int intValue();
    
    public byte byteValue() {
        return (byte) intValue();
    }
}

public class Double extends Number {
    public int intValue() { ... }
}
```

Polymorphism & Dynamic Binding

Overview

- polymorphism is achieved via dynamic binding
  Person p1 = new Person("Fred");
  Student sp = new Student("George", 204503);
  String s1 = p1.getInfo();
  String s2 = sp.getInfo();
  Person p2 = sp;
  String s3 = p2.getInfo();

- on method call, the method body executed is determined by the class of the object not the class of the variable
• Java: default is dynamic binding
  - final: static binding
  - all objects accessed by reference
• C++: default is static binding
  - virtual: dynamic binding, method may be overridden
  - polymorphic variables require pointers
  - slicing
    Student sp("George", 204503);
    Person p2 = sp;
• Delphi:
  - virtual for methods that may be overridden, and
  - override for the overriding method
Type Checking

- full type checking at compile time still possible
- based on static type (i.e. type of variable)
- polymorphic variable can only be assigned a subclass object
- principle of substitutability says that if method exists in the superclass it exists in the subclass
  
  ```java
  p2.getInfo()
  ```
  - must be valid for any object that `p2` can reference
- cannot reference methods that exist solely in subclass since may not exist in all subclasses
- use cast to access new operations in subclass:
  ```java
  Student s = (Student) p2;
  ```
- can determine run-time type via `instanceof`
  ```java
  if (p2 instanceof Student) ...  
  ```
Heterogeneous Data Structures

• array of polymorphic variables
  - each element can contain objects from declared type or any subtype, e.g.

```java
Person[] pList = new Person [100];
for (int i = 0; i < 100; i++) {
    String st = pList[i].getInfo();
}
```
Implementation of Dynamic Binding

• method access tables
  - required calling information for each method in class
  - includes inherited (not redefined) methods
  - created at compile time
    • property of class
    - each object has link to method access table for its class
### Person object

<table>
<thead>
<tr>
<th>Fred</th>
<th></th>
</tr>
</thead>
</table>

### Student object

<table>
<thead>
<tr>
<th>George</th>
<th>204503</th>
</tr>
</thead>
</table>

#### Person method table
- Address of `getInfo` defined in `Person`  
- Address of `detailInfo` defined in `Person`  

#### Student method table
- Address of `getInfo` defined in `Student`  
- Address of `detailInfo` defined in `Person`  
- Address of `getRegNum` defined in `Student`
Procedural vs. Object-Oriented

• e.g. operations on different kinds of shapes
• procedural approach
  – one procedure per operation
  – shape specified as parameter
  – check type of shape in procedure
• object-oriented approach
  – one class per type of shape
  – common superclass with common operations
  – override operations as needed
Procedural Shapes

void draw (Shape s) {
    switch (s.kind) {
        case LINE: /* code to draw a line */ ...
        case RECTANGLE:
            /* code to draw a rectangle */ ...
        case CIRCLE: /* code to draw a circle */ ...
        default: /* error message */ ...
    }
}
Object-Oriented Shapes

• **Shape** is abstract class
  - abstract method **draw**
    • defined in **Shape**
    • implementation in **Rectangle, Circle**
• abstract class:
  - contains 1 or more abstract methods
  - cannot create objects of that class
  - subclasses are:
    • concrete if they implement all abstract methods,
    • else they too are abstract
**OO Hierarchy for Shapes**

```
<table>
<thead>
<tr>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>#height</td>
</tr>
<tr>
<td>#width</td>
</tr>
<tr>
<td>#position</td>
</tr>
</tbody>
</table>

...  
+move()  
+draw()  
...

---

<table>
<thead>
<tr>
<th>Polygon</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>+draw()</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>+draw()</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
```
OO Shapes

```java
public abstract class Shape {
   // declaration of protected attributes and
   // public operations common to all shapes
   ...
   public abstract void draw();
   // definition of other abstract methods
   ...
}

class Line extends Shape {
   ...
   public void draw() {
      // code to draw a line ...
   } // draw
} // Line

...
Shape ashape;
ashape = new Line(...);
ashape.draw();
ashape.length(); // not valid
```
Multiple Inheritance

- class with more than one parent
  - inherits variables and methods from each parent
- C++, Eiffel: allowed
- Java, Ada: not allowed
- operation with same name, parameters inherited from >1 superclass:
  - Eiffel: one of methods must be renamed in subclass
  - C++: use scope resolution operator `::`
- Java interfaces
  - class may inherit specification from more than one interface
Multiple Specification Inheritance

interface Moveable {
    public void horiz(int val);
    public void vert(int val);
} // Moveable

public class Circle extends Shape implements Moveable, Serializable {
    ...
} // Circle
Information Hiding and Inheritance

• problems
  - subclass may provide methods to access or change protected attribute in superclass
  - set of related protected attributes may be changed independently in subclass

```java
class Date {
    protected int day, month, year;
    public Date(int d, int m, int y) {
        // check that d, m and y make a sensible
        // date before creating the object
        ...
    } // constructor
    public void tomorrow() {... //new date is valid }
    ...
}

class NewDate extends Date {
    public void tomorrow ( ) { d = d+1 ;}
}
```
• solutions
  - private attributes with protected methods for access/update
  - methods that may not be overridden (final)
  - package visibility
• Eiffel
  - separation of inheritance and information hiding
  - subclass inherits all attributes and methods of superclasses
  - each class has export list
    • usable but not modifiable by client
Behavioural Inheritance

• weak form of principle of substitutability
  – $S$ is a subtype of $T$ if substituting an object of type $S$ wherever an object of type $T$ is expected does not introduce the possibility of type errors at run time.

• syntactic conditions
  – all services in superclass present in subclass
  – additional services, attributes may be present in subclass
  – if service redefined in subclass then it must be compatible with original service
Compatible Services

- redefined service is compatible with original service if:
  - it has same number of parameters
  - *contravariance rule*: type of each input parameter is supertype of (or same type as) corresponding parameter in original
  - type of each output parameter is subtype of (or same type as) corresponding parameter in original
  - Java example: car, vehicle

- **Covariance**:
  - type of parameter in redefined service may be subtype of corresponding parameter in original
  - used in Eiffel
  - second level of checking to detect possible run-time errors
public class Vehicle {
    protected int licNum;
    ...
    public int getNumV () {
        return licNum;
    }
    public boolean eq(Vehicle v) {
        return v.getNumV == licNum;
    } // eq
} // Vehicle

public class Car extends Vehicle {
    private int numSeats;
    ...
    public int getNumS() {
        return numSeats;
    }
    public boolean eq(Car v) { // overloads eq
        return (v.getNumS() == numSeats) && (v.getNumV == licNum);
    } // eq
} // Car

Car c1 = new Car(27, 4); Car c2 = new Car(27, 5);
Vehicle v1 = c1; Vehicle v2 = new Vehicle(27);
// v1.eq(c2) is the same as v1.eq(v2)
Implementation

Inheritance

• primary concern: re-use of code
• example: stack, queue
• inclusion polymorphism does not guarantee behavioural inheritance
• strong form of principle of substitutability
  - \( S \) is a subtype of \( T \) if, for each object \( s \) of type \( S \), there is an object \( t \) of type \( T \) such that, for all programs \( P \) defined in terms of \( T \), the behaviour of \( P \) is unchanged when \( s \) is substituted for \( t \)
  - almost impossible to enforce
Implementation Inheritance - Example

```java
public class Queue {
    protected int[] vals;
    protected int first, last, numElements;

    public Queue() { ... }
    public void add(int item) {
        ... // add element to back of Queue ... }
    public int remove() {
        ... // return front element of Queue ... }
} // Queue

public class Stack extends Queue {
    public Stack() { ... }
    public int remove() {
        ... // return top element of Stack ... }
} // Stack
```