Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

OO Fundamentals

I. Salomie, C.Pop
2017

Content

- Part 1 – UML Diagrams
- Part 2 – Inheritance
- Part 3 – Polymorphism
- Part 4 – Abstract classes and Interfaces

PART 1
UML Diagrams

Model

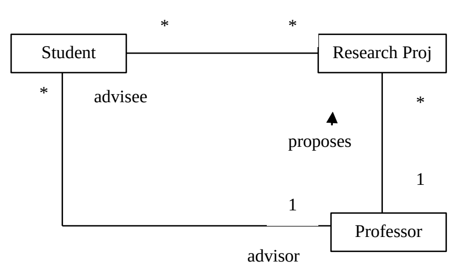
- A complete system description

UML diagrams for visualizing models

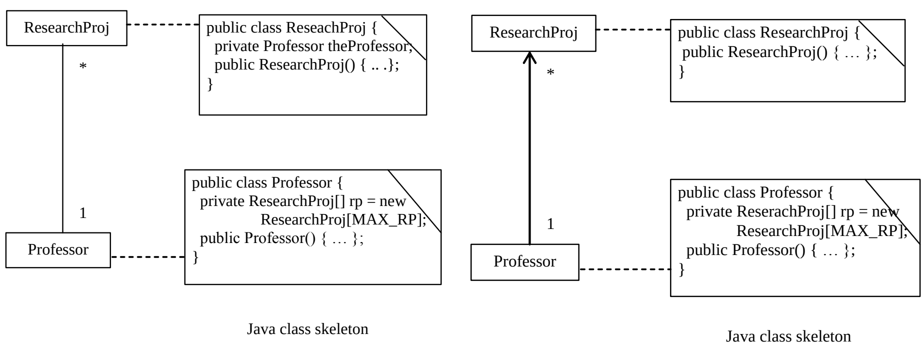
- Use case diagram - capturing requirements and illustrating user interactions with the system
- Class and object diagrams - illustrate logical structure of the system
- Sequence, Collaboration, State diagrams - illustrate system behaviour
- Activity diagrams - illustrate the flow of events in a use case
- Package diagrams - shows packages of classes and their dependencies
- Component diagrams - illustrate physical structure of software
- Deployment diagrams - shows the mapping of software to hardware configurations

Associations

- Binary relationships among classes
- Label (optional)
  - describes the associations
  - directions arrow
- Association ends (see next figure)
  - role name
  - multiplicity
- Design decisions related to navigation among class relations
  - unidirectional
  - bidirectional



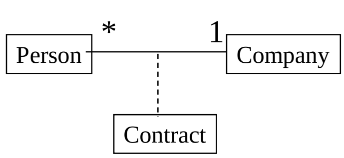
Associations



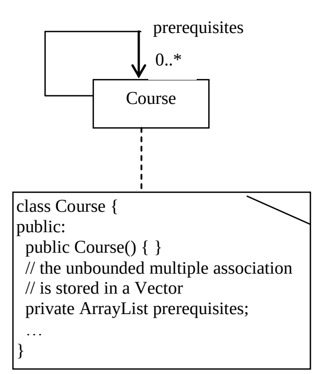
Bi-directional associations

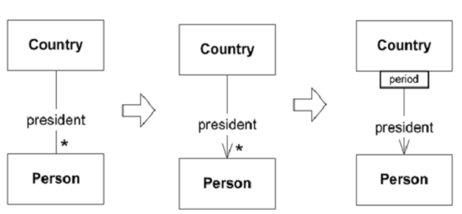
Uni-directional associations

Associations



Association as a class





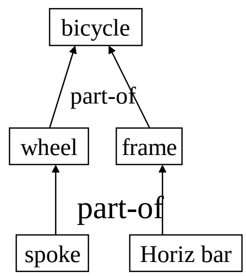
a) Association b) Association traversal c) Qualifier

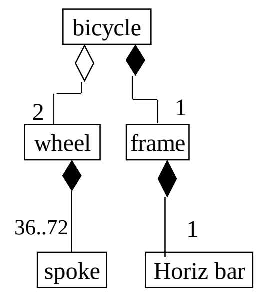
Reflexive associations

Qualifiers to reduce multiplicity

Aggregation

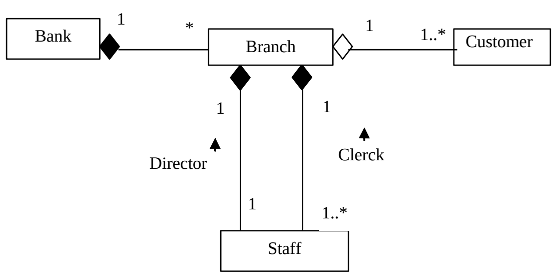
- Special form of association
- Relationship
  - "has – a" **or** "part – of"
- Weak aggregation
- Strong aggregation
- UML aggregation indicator
- Properties
  - transitive
  - not reflexive

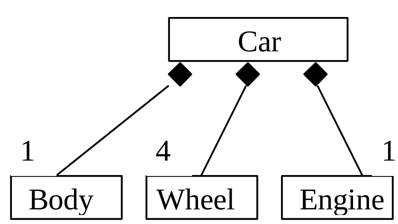




Aggregation

Strong and weak aggregation





Class and Object Diagrams

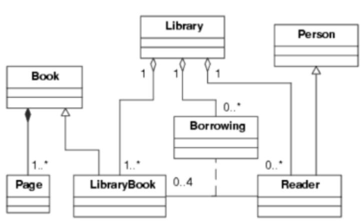
- Describes the classes and objects in the system and the relationships among them
  - static relationships
    - associations
    - aggregation
    - generalization (inheritance)
- For a class can show
  - attributes
  - operations
  - constraints

Perspectives

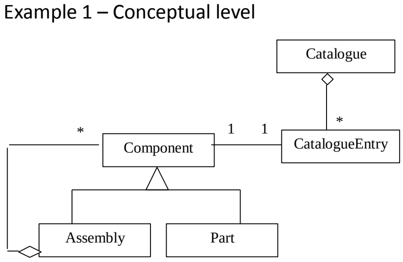
- Conceptual
  - draw a diagram representing the concepts in the problem domain
  - not related to implementation
  - focused on
    - functionality
    - relationships among concepts
- Specification
  - software perspective
  - behavioural approach (interfaces) not implementation
- Implementation
  - proper classes with attributes and operations
- Notes
  - the lines between the three approaches is not very clear
  - not big differences between ‘conceptual’ and ‘specification’
  - there are notable differences between ‘specification’ diagrams and implementation ones

Class and Object Diagrams

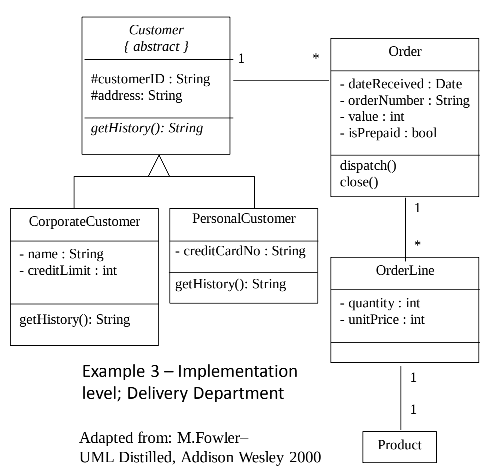
Examples



Example 1 – Conceptual level



Example 2 – Conceptual level

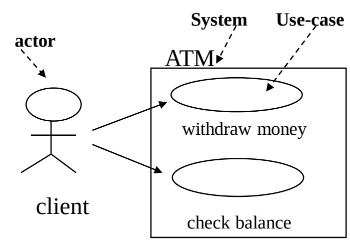


Example 3 – Implementation level; Delivery Department

Adapted from: M.Fowler– UML Distilled, Addison Wesley 2000

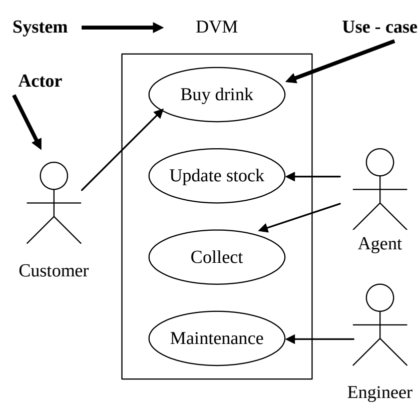
Use cases

- Use-cases
  - Introduced in SE by Ivar Jacobson (1992)
  - Represents a set of scenarios related to how the system is used
- Use cases help to discover
  - System entities,
  - System actors (roles) - a human person, machine or software play the role during system operation
  - Attributes and
  - Behavior
  - How actors are interacting with system resources
- UML introduces use case diagrams to graphically capture system actors, use-cases and their relationships
- Use-case document – describes the sequence of actor - system interaction (can be written in natural language)
- Use case "check balance"
- Use-case description
  - Client inserts the card into ATM
  - System prompts for PIN
  - System validates the PIN
  - System ask for an operation. Client selects check balance
  - System communicates with the ATM network
  - System prints balance



Use cases

- Use case - should address the following questions [jacobsen]
  - what are the main tasks performed by the actor
  - what info the actor will bring, acquire or change
  - what information does the actor desire about the system
- Use-case document
  - document about the sequence of actor - system interaction (can be written in natural language)



Example – Vending Machine

Interaction diagrams

- Describe how groups of objects collaborate
- Typically captures the behaviour of a single use-case
  - shows the objects involved and the messages passed among the objects
- Shows the collaboration among the objects of a use-case

Sequence Diagrams

Shows the objects involved in the interaction

Lifeline
- dashed vertical line below the objects
- shows the period of time the objects play a certain role in a use case

Message
- arrows pointing from a lifeline of a sending object to the lifeline of a receiving object
- messages have names and may have arguments
- each message moves the flow of control from one object to another

Scenario
- a use case may contain more scenarios
- one of the scenario is “the best case scenario”
- other scenarios can describe different events and conditions in the use case
- examples:
  - use case “buy drink” from a vending machine
  - best-case scenario
  - no change to return scenario (will return the amount and cancel transaction)

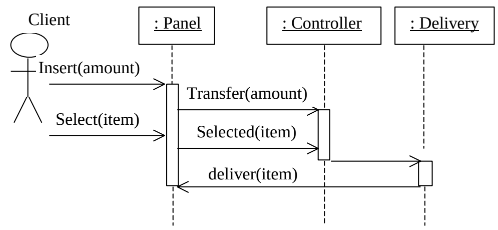
Activation
- narrow rectangle below the object along with the lifeline
- the period of time during which the object is processing the message
- when an object finishes processing a message the control returns to the sender of the message. This marks the end of the activation corresponding to that message
- it is marked by a dashed arrow going from the bottom of the activation rectangle back to the lifeline of the object that send the message
- activations and return messages are optional in a sequence diagram
- In the course of message processing one object may send messages to other objects

Sequence diagrams

"Buy drink" use-case

Vending machine components (objects): front panel, controller, delivery unit, store

1. Client inserts the money into machine front panel
2. The client makes a selection
3. The money go to the controller
4. The controller checks if the selected item is in the store unit
5. Assuming a best case scenario, the item exists, the controller updates the cash and items and ask the delivery unit to deliver the item from the store
6. The deliver unit delivers the items in the front of the machine

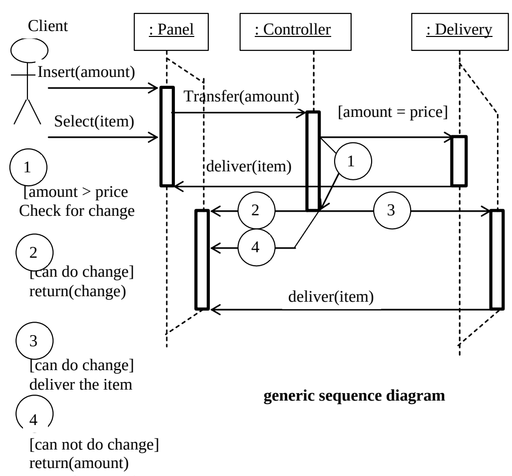


Sequence diagrams

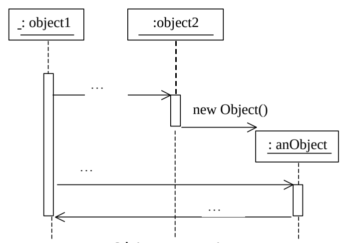
"Buy drink" use-case
Generic Sequence diagrams

- Takes into account the flow of control
- Allow more scenarios in the diagram
- The next diagram shows the sequence diagram for 'incorrect-amount-of-money' scenario
- Each condition causes a 'fork' of control in the message separating the message into paths
- Conditions are represented in square brackets

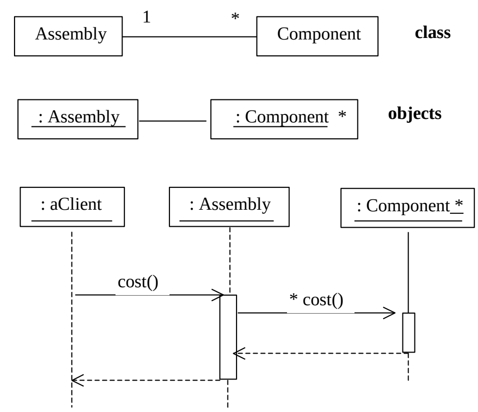
  [amount = price]
- Another scenario may involve 'out-of-drink'



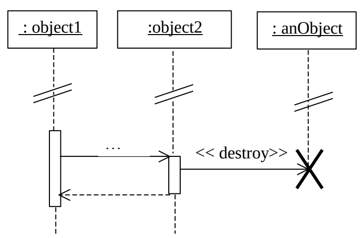
Sequence diagrams



Object creation



Iterated messages



Object destruction

Activity Diagrams

Concepts

activity

- a state of performing some tasks (a method of a class, a real world activity (e.g. digging, eating, etc.))
- activity diagram describes sequences of activities
- support for conditional and parallel behaviour

transition

branch

- a decision point at which there are two or more possible path of flow of control
- single incoming transition
- several guarded outgoing transitions
- the branch is mutual exclusive
- [else] – all others are false
- guard – boolean expression

merge

- multiple entry transitions
- single output
- marks the end of conditional behavior

fork

- one incoming transition
- several outgoing transitions
- when the incoming trans. is triggered, the outgoing transitions are taken in parallel
- can specify concurrent programming

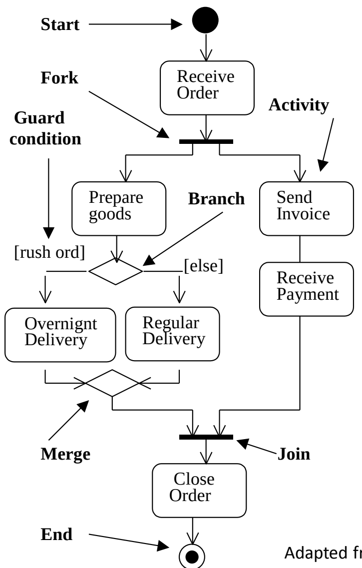
join

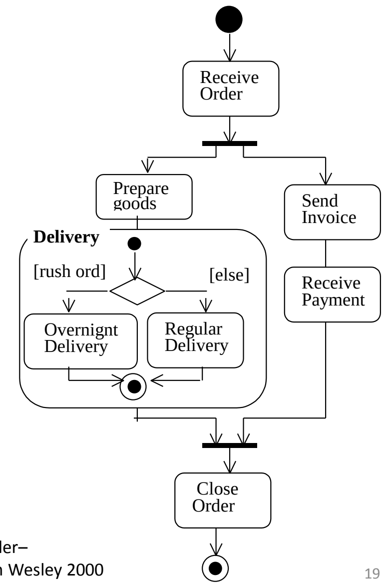
- multiple entry transitions
- one outgoing transition
- synchronization role

The representation symbols of the main concepts are shown in the activity diagram example on the next slide

Activity Diagrams

Example – Order Processing



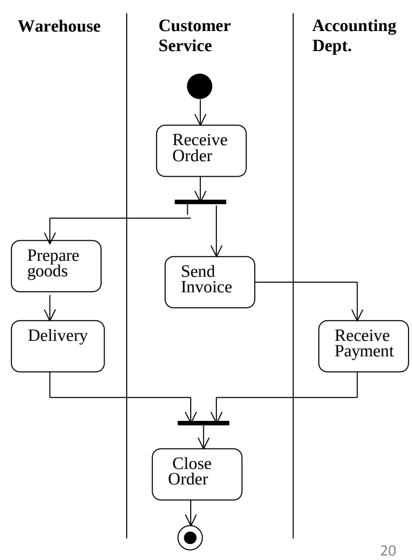


Adapted from: M.Fowler–
UML Distilled, Addison Wesley 2000

Activity Diagrams

Example – Order Processing – Swim Lanes

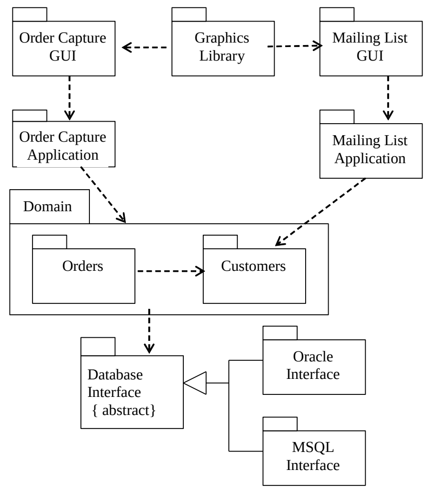
- The activities and actions of an activity diagram can be grouped into swim lanes
- The activities in each lane can be associated to a department (part) of an organization
- The lane borders are strictly conceptual delimited



Package Diagrams

- Package diagrams – help decomposition of large systems into subsystems
- In UML a *package* is a collection of modelling elements that are grouped together because they are logically related
- When defining packages, the principles of cohesion and coupling should be applied
- UML Package symbol (see next slide) may contain:
  - classes, instances, text, other packages
- A package diagram shows packages of classes and their dependencies
- Dependency
  - two elements are dependent if changes to one element may cause changes to the other

Example of a package diagram

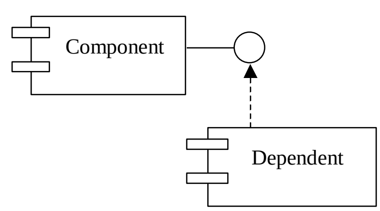


Adapted from: M.Fowler–UML Distilled, Addison Wesley 2000

Component Diagrams

- System component
  - physical elements like files, executables, etc.
- A component diagram shows how system components relate to each other
- Difference between package diagram and component diagram
  - package diagram – logical grouping of design elements
  - component diagrams – physical components
- Component diagrams can be combined with deployment diagrams to show the physical location of components of the system

Dependency of a component on the interface of another component

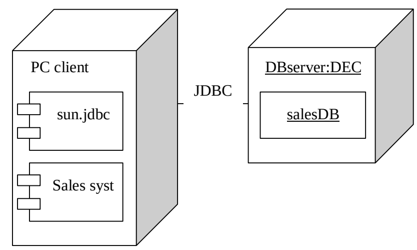


Adapted from: M.Fowler–UML Distilled, Addison Wesley 2000

Deployment Diagrams

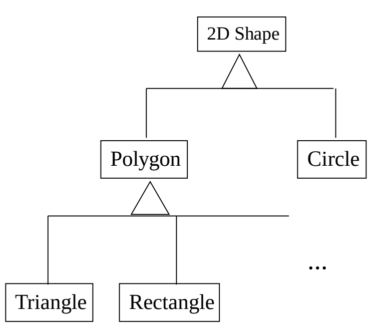
- Show the physical architecture of the system
- Show the configuration of run-time processing elements and the software components and processes located on them
- The configuration shows the nodes and communication associations
- Nodes are typically used to show computers, while communication associations show the network and protocols used to communicate between nodes
  - Nodes are shown as 3D cubes
- Deployment diagrams can show either types of machines or their names and / or the active components within the nodes

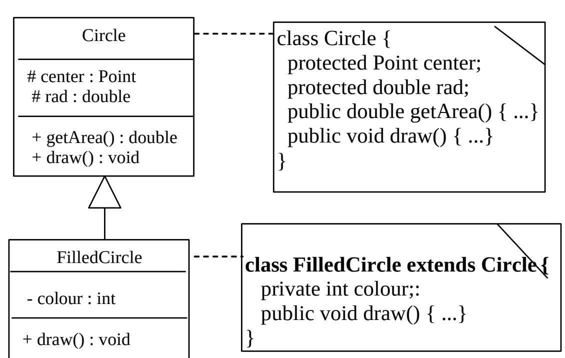
Example of a deployment diagram



Adapted from: M.Fowler–UML Distilled, Addison Wesley 2000

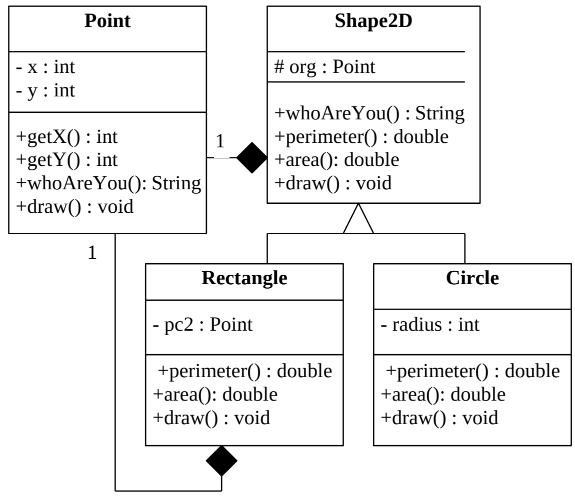
PART 2
Inheritance





Inheritance

- **Example**
- Class system (Point, Circle, Rectangle)
- Factor out the commonalities
  - structure
  - behavior
- Objects of classes Circle and Rectangle will be abstracted in the class Shape2D
- Shape2D superclass for Circle and Rectangle
- Attribute Point of Shape2D:
  - inherited by Circle (as centre)
  - inherited by Rectangle (as one corner)



Inheritance
Java example

Hierarchy of classes

Superclasses and subclasses

public class Shape2D { … }

public class Circle **extends** Shape2D { … }

public class Rectangle **extends** Shape2D { …}

A class hierarchy can be extended with other classes

public class FilledCircle **extends** Circle { … }

public class FilledRectangle **extends** Rectangle { …}

```
public class Shape2D {
    protected Point org;
    public Shape2D(Point org) { this.org =org; }
    public String whoAreYou() {return "SHAPE2D"; }
    public void draw() {
      System.out.println ("I am a generalized
      Shape2D. Don't know how to execute draw");
    }
    public double perimeter() { return 0.0; }
    public double area() { return 0.0; }
}
```

Inheritance
Java example

```
public class Circle extends Shape2D {
  protected double radius;
  public Circle(Point pc, double radius) {
    super(pc);
    this.radius = radius;
  }
  // Specific Interface
  public String whoAreYou() { return "CIRCLE"; }
  public void draw()
    {System.out.println(toString());}
  public double perimeter()
    { return 2.0 * Math.PI * radius;}
  public double area()
    {return  Math.PI * radius * radius;}
  public String toString() {
    return "CIRCLE with Center " +
    org.toString() + " and radius " +  radius;
  }
}
```

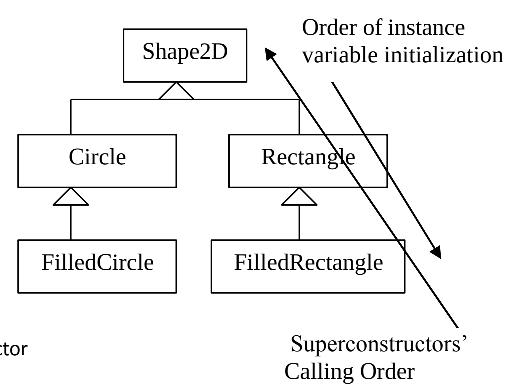
```
public class FilledCircle extends Circle {
  private int color;
  public FilledCircle(Point pc, double radius, int color) {
      super(pc, radius); this.color = color;
  }
  public String whoAreYou() { return "FILLED-CIRCLE";}
  public void draw() {  System.out.println(toString()); }
  public String toString() {
        return "FILLED-CIRCLE Center in: " + org.toString() +
      " and radius: " + radius +        "color:" + color;
  }
}
```


Constructors in the context of inheritance

Initialization steps

- Initialization of the inherited variables
- Initialization of the self defined variables
- Invoking one of the superclass constructors

```
public FilledCircle (Point pc, double radius,
                             int color) {
   super(pc, radius);
   this.color = color;
}
```



Call to the superclass constructor:

super(pc, radius);

- The first statement in the subclass' constructor
- Same policy for the class Circle constructor:

```
public Circle(Point pc, double radius) {
  super(pc);
  this.radius = radius;
}
```

Constructors in the context of inheritance this() and super()

```
import java.awt.Color;

public class ColoredPoint extends Point {
  private Color color;
  public ColoredPoint(double x,double y,Color
     color) {
     super(x,y);
     this.color = color;
  }
  public ColoredPoint(double x, double y) {
     this(x, y, Color.black); // default color value
  }
  public ColoredPoint() {
    color = Color.black;
  }
}
```

- super()
  - superclass constructor invocation
  - super() invocation - first statement in the subclass constructor
  - super(x,y) invokes super(double, double) in the superclass;
- this()
  - invokes a constructor of this class that matches the given signature
  - should be the first statement in the constructor
  - this(x, y, Color.black) invokes the constructor ColoredPoint(double, double, Color)
- In the default constructor of ColoredPoint()
  - default super constructor is invoked;

Constructors in the context of inheritance

- No constructor in the subclass
  - Default constructor is provided by default
  - It invokes the default constructor of the superclass

```
public class SubC extends SupC {
  public SubC() { super(); }
  // ... fields and methods
}
```

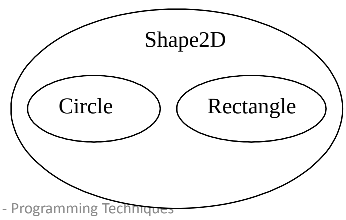
- No default constructor in the superclass
  - Compile error
- A parameter constructor is defined in the superclass
- Default constructor is not implicitly provided

Other remarks

- Inheritance - strong coupling between constructors
  - any modification in the superclass constructor is reflected in the subclass constructor
- Limit the number of hierarchy levels to 3 or 4
- **For code reuse, prefer delegation over inheritance**

PART 3
Polymorphism
Inheritance and Subtypes

- A subclass extends features of its superclass
  - Adds new features
  - Specialization and generalization
- Every instance of the subclass is an instance of the superclass and not vice versa
  - Every circle is a Shape2D but not every Shape2D is a Circle



- Inheritance, types and subtypes
  - Each class defines a type
  - All instances of a class - the set of valid type values
  - Every instance of a subclass is also an instance of its superclass
  - The type defined by subclass is valid subset of the type defined by its superclass
- Subtype relation applies to class types, interface types and primitive types
  - Class inheritance relation - is a subtype relation
  - Each interface also defines a type
  - Interface extension and implementation relations are also subtype relations

Substitutability of subtypes

- A value of a subtype can appear wherever a super-type is expected
- In the context of classes and objects
  - A subclass object can appear wherever a superclass object is expected
  - An instance of a subclass can always substitute for an instance of its superclass

Conversion of reference types

- Governed by subtype relation
- Widening
  - The conversion of a subtype to one of its supertypes
  - Widening is always allowed
  - Is carried out implicitly whenever necessary
- Narrowing (downcasting)
  - The conversion of a supertype to one of its subtypes
  - Requires explicit casts
  - Allowed at compile time
  - Not always safe – may generate run time exceptions
- Differences between conversions of primitive types and reference types
  - In case of primitive types
    - Change of representation
  - In case of reference types
    - No effect on representation

Polymorphism

Polymorphism

- Taking many forms
- Polymorphic behavior of a method
- Polymorphism [Cardelli & Wegner]:
  - Universal (Parametric and Inclusion)
  - Ad-hoc (Coercion and Overloading)

Binding (Linking)

- Static binding – during compile time
- Dynamic binding – during run time

Polymorphism and type checking

- Polymorphism => flexibility
- Type checking => rigidity
- Compatible by linking
- Trade - off

Type checking and linking

- Static linking
  - Static type checking
    - Guarantees the correctness
    - Rigid interpretation
  - Dynamic type checking
    - Invalid combination
- Dynamic linking
  - Static type checking
    - Guarantees the correctness
    - Flexible interpretation
  - Dynamic type checking
    - doesn't guarantees the correctness
    - Flexible interpretation

Polymorphic Assignments

- In static languages such as C
  - Left and right side of an assignment must be compatible types
- In OO languages
  - Polymorphic assignment
  - Powerful form of assignment
  - Rule of assignment

Rule of Assignment in Java

- Downcasting a reference variable to a subtype of its declared type
- Explicit casting is allowed at compile time
- The validity of explicit casting - checked at run time
- In case of invalid cast ClassCastException
- Example: c1 = (Circle) s2;
  - compile time
  - run time

Example

```
class Shape2D { … }
class Circle extends Shape2D { …}
class Rectangle extends Shape2D { … }
Shape2D s1, s2;

// polymorphic assignments here
// no explicit casting is necessary
s1 = new Circle();
s2 = new Rectangle();

Circle c1;

// the following line: compile error even though actually
// s1 holds a reference to a Circle instance
// the declared type of s1 is Shape2D and this is not subtype
// of left hand side (i.e. Circle)
// type checking takes place at compile time
c1 = s1;

// Explicit cast is necessary here
c1 = (Circle) s1; // ok, explicit cast
```

UTCN - Programming Techniques 35

Polymorphic Assignments
Downcasting techniques

Pessimistic approach

```
if(s2 instanceof Circle) {
  Circle cref = (Circle) s2;
}
else {
  // … do something else
}
```

Optimistic approach

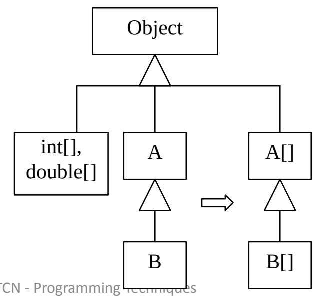
```
try {
  // …
  Circle cref = (Circle) s2;
  // …
} catch (ClassCastException e) {
  // … do something
}
```

UTCN - Programming Techniques 36

Subtypes and arrays

Rule 1
- All array types (i.e. int[], double[]) are subtypes of Object

Rule 2
- If class or interface B is subtype of class or interface A then B[] is also subtype of A[]



- The following sequence is OK

```
Shape2D sa[];
Circle ca[] = new Circle[10];
// …
sa = ca; // polymorphic assignment
Shape2D s1 = sa[2];
Circle c1 = ca[3];
```

- Compile error

```
Circle c2 = sa[0];
```

- Explicit downcasting is necessary

```
Circle c2 = (Circle)sa[0]; // ok
```


Overriding

- Overriding
  - Class, subclass
  - Method specialization
  - Subclass method has the same elements with the method in the superclass
    - Name
    - Parameters
    - Return type
- Overloading
  - Methods of the same class
    - Same name
    - Different parameters

```
class A {
  public void m() { … }
}
class B extends A {
  public void m() { … }
}
```

- Method m of class A is overridden by implementation of method m in class B
- For a given object, only one m is available to be invoked

```
A a = new A();
B b = new B();
a.m(); // call m of class A
b.m(); // call m of class B
```

- Overriding and final methods

Overrdiding
Invoking methods

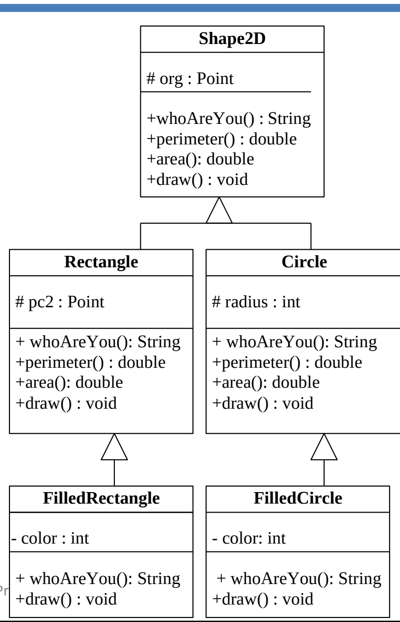
```
public class Point { …}
public class ColoredPoint extends Point
  { … }

public class Point {
  public boolean equals (Object other) {
    if(other != null) && other instanceof
     Point) {
      Point p = (Point) other;
      return (x == p.x) && (y == p.y);
    }
    else { return false; }
  }
  // …
}
```

```
public class ColoredPoint extends Point {
  public boolean equals (Object other) {
    if(other != null) && other instanceof
                             ColoredPoint) {
      ColoredPoint p = (ColoredPoint) other;
      return (super.equals(p) &&
                   color.equals(p.color));
    }
    else { return false; }
  }
  // …
}
```

- **equals** of ColoredPoint overrides equals of Point
- equals method of Point is invoked through reference super

Inheritance and overriding




```
…
FilledCircle cp1 = new FilledCircle(…);
FilledRectangle dp1 = new
     FilledRectangle(…);
System.out.println(cp1.area());
System.out.println(cp1.perimeter());
System.out.println(dp1.area());
System.out.println(dp1.perimeter());
…
```

Inheritance and overriding

Polymorphic method call

- Class system (Point, Circle, Rectangle)

```
Shape2D s;
if(cond) s = new Circle(…);
else s = new Rectangle(…);
s.draw();
```

- The invoked draw() method depends on the actual class of the object referenced by s at runtime
  - It doesn't depends on the declared type of s
  - This is polymorphic method invocation
  - **Implementation of a method is bound to an invocation dynamically at run time**

Dynamic binding algorithm

- Consider the polymorphic method invocation:

var.m(…);

Step 1:

crtClass = the class of the object referenced by *var*

Step 2:

if *m()* is implemented by *crtClass*

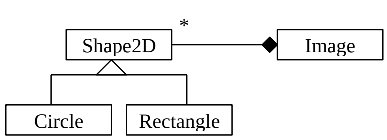
then invoke *m()*

else *crtClass* = the superclass of *crtClass*

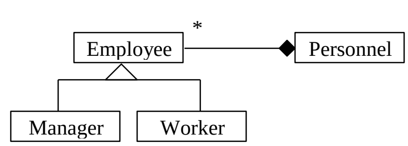
repeat Step 2

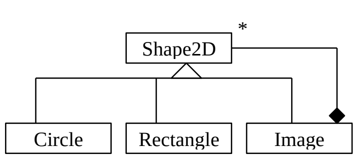
Polymorphism and collection processing

- Heterogeneous collections
- Definition
- (Some) Methods of the superclass are overridden in the subclasses
- Polymorphic behaviour when processing heterogeneous collections
  - ex. Calling the most appropriate method when iterating collections

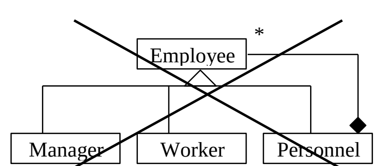


No multilevel images



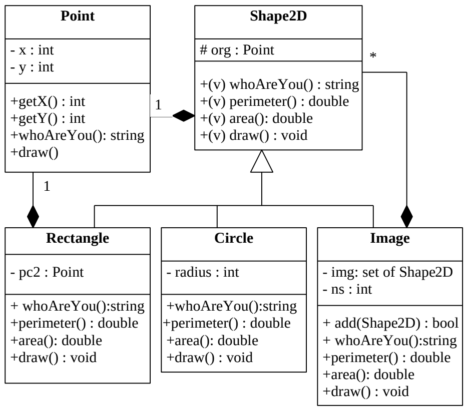


Allows for multilevel images



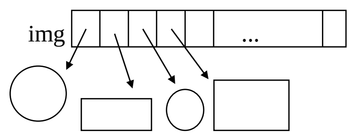
Multilevel Personnel not appropriate

Polymorphism and collection processing




```
public class Image extends Shape2D {
  private Shape2D[] img = new Shape2D[MAXSIZE];
  private int ns; // effective number of shapes
  public void draw() {
    for(int i = 0; i < ns; i++) {
      // polymorphism - dynamic linking
      img[i].draw(); } }

  public double area() {
    double totalArea = 0.0;
    for (int i =0; i<ns; i++) {
      // polymorphism - dynamic linking
      totalArea += img[i].area(); }
    return totalArea;
  }}
}
```



Generic methods through dynamic linking and polymorphism

- Example - external method to compare the area of two generic Shape2D objects
  - *Alternative to compareTo method defined by the Comparable interface*

  public Class Shape2D implements Comparable { …}

```
public double deltaArea(Shape2D s1, Shape2D s2){
    double d = s1.area() - s2.area();
    return d;
}

// usage
Circle c1 = new Circle(…);
Rectangle r1 = new Rectangle(…);
if(deltaArea (c1, r1) <= EPS) {
  // ... nearly equal area shapes
}
else  {
  // ... different areas
}
```

PART 4
Abstract classes and interfaces
Abstract classes

- A Java abstract class has at least one abstract method
- An abstract method has no definition body
- Abstract methods must be defined in the subclasses
- Abstract classes could not be instantiated into objects
- References to abstract classes could be passed as parameters in methods
- Abstract class content
  - instance and class (static) variables,
  - abstract and implemented methods
  - constructors
- Usefulness - in the generalization process
- All abstract methods are inherited
- How should we approach the abstract methods?
  - the subclass defines the abstract method
  - the subclass redefines it as abstract an let its subclasses to approach it

- Consider Shape2D defined as an abstract class
- It's an error to instantiate objects of class Shape2D:

```
Shape2D f1 = new Shape2D (…); // ERROR
```

- It's OK to define a method of a class taking a parameter of type Shape2D:

```
public class C {
    …
    public void aMethod(…, Shape2D f, …) { … }
}
```

- When calling aMethod
  - Instead of 'f' we have to supply a reference to an object instance of Shape2D subclass (i.e. Circle or Rectangle)
  - aMethod features a polymorphic behaviour

UTCN - Programming Techniques 45

Abstract classes

- A Java abstract class has at least one abstract method
- An abstract method has no definition body
- Abstract methods must be defined in the subclasses
- Abstract classes could not be instantiated into objects
- References to abstract classes could be passed as parameters in methods
- Abstract class content
  - instance and class (static) variables,
  - abstract and implemented methods
  - constructors
- Usefulness - in the generalization process
- All abstract methods are inherited
- How should we approach the abstract methods?
  - the subclass defines the abstract method
  - the subclass redefines it as abstract an let its subclasses to approach it

- Consider Shape2D defined as an abstract class
- It's an error to instantiate objects of class Shape2D:

```
Shape2D f1 = new Shape2D (…); // ERROR
```

- It's OK to define a method of a class taking a parameter of type Shape2D:

```
public class C {
    …
    public void aMethod(…, Shape2D f, …) { … }
}
```

- When calling aMethod
  - Instead of 'f' we have to supply a reference to an object instance of Shape2D subclass (i.e. Circle or Rectangle)
  - aMethod features a polymorphic behaviour

UTCN - Programming Techniques 46

Interfaces

- Interface declares a behaviour
  - Set of abstract methods
- Classes implementing interfaces
  - Multiplicity
  - Method implementation
    - All methods should be implemented
- Interfaces extending interfaces
  - Multiplicity

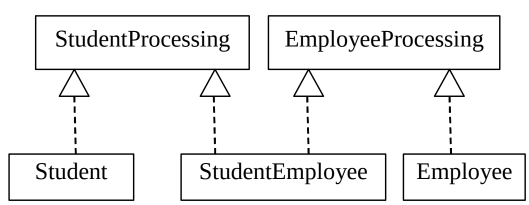
Subtypes (revisited)

- Interface – defines a type
- Interface extension and implementation
  - subtype relations
- Reference types in Java
  - class type,
  - array type or
  - interface type
- The complete subtype relations

- Assume:
  - C, C1, C2 are classes
  - I, I1, I2 are interfaces
  - T, T1, T2 are types (references or primitives)
- If C1 extends C2 => C1 is a subtype of C2
- If I1 extends I2 => I1 is a subtype of I2
- If C implements I => C is subtype of I
- For every I => I is a subtype of Object
- For every T => T[] is a subtype of Object
- If T1 is subtype of T2 => T1[] is a subtype of T2[]
- For any C that is not Object => C is subtype of Object

Interfaces and polymorphic classes

- Interfaces can be used to define polymorphic classes
- A class can implement multiple interfaces
- A class shows polymorphic behavior (assumes different roles in different contexts)
- StudentEmployee is subtype of both StudentProcessing and EmployeeProcessing
- Instances of StudentEmployee can be viewed as students or employees




```
public interface StudentProcessing {
  public double calculateAvgMarks();
  public double getAvgMarks();
  // ... other methods
}

public interface EmployeeProcessing {
  public double calculateSalary();
  public double getSalary();
  // ... other methods
}
```

Interfaces and polymorphic classes

Instances of StudentEmployee seen as student

```
StudentProcessing[] students = new
   StudentProcessing[SIZE];
students[0] = new Student(…);
students[1] = new StudentEmployee(…);
// …
for(int i = 0; i <students.length; i++) {
 … students[i].calculateAvgMarks();
}
```

In other context, instances of StudentEmployee can be viewed as employees

```
EmployeeProcessing[] employees = new
  EmployeeProcessing[SIZE];
employees[0] = new Employee(…);
employees[1] = new StudentEmployee(…);
// …
for(int i = 0; i <employees.length; i++) {
 … employees[i].calculateSalary();
}
```

Conclusion

A **student employee** can play two different roles in two different contexts

Interfaces and polymorphic classes

A source of inconsistency (see the highlighted methods)

```
public class Student implements
                StudentProcessing {
   protected double avgMarks;
   public double calculateAvgMarks() {
     // … method implementation
   }
   public double getAvgMarks() {
     // … method implementation
   }
}

public class Employee implements
             EmployeeProcessing {
  protected double salary;
   public double calculateSalary() {
     // … method implementation
   }
  public double getSalary() {
     // … method implementation
  }
}
```

```
public class StudentEmployee implements
    StudentProcessing, EmployeeProcessing {
   protected double avgMarks;
   protected double salary;
   // … other variables
   public double calculateAvgMarks() {
     // … method implementation
   }
   public double getAvgMarks() {
     // … method implementation
   }
   public double calculateSalary() {
     // … method implementation
   }
   public double getSalary() {
     // … method implementation
   }
}
```

-The problem is generated because there is no common implementation of the two methods to be inherited (and eventually specialized)
-Java is not defining multiple inheritance

Interfaces and polymorphic classes

A source of inconsistency

```
public class StudentProcessing {
  public double getAvgMarks() { … }
  public double calculateAvgMarks() { … }
  protected double avgMarks;
  // … other resources
}

public class EmployeeProcessing {
  public double getSalary() { … }
  public double calculateSalary() { … }
  protected double salary;
  // … other resources
}

public class Student extends
                StudentProcessing {
  // implementations of getAvgMarks()
  // and calculateAvgMarks are inherited
  // … other resources
}
```

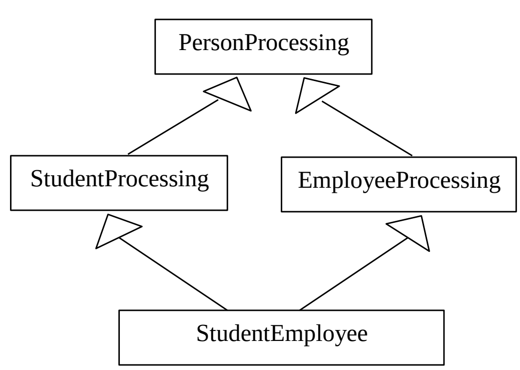
```
public class Employee extends EmployeeProcessing {
  // implementations of getSalary() and
  // calculateSalary() are inherited
  // … other resources
}

// the following would solve the problem
// but its is illegal Java code
public class StudentEmployee extends
        StudentProcessing, EmployeeProcessing {
    // implementation of getAvgMarks()
    // and getSalary()
    // and calculateAvgMarks() and
    // calculateSalary() are inherited
    // … other resources
}
```


Interfaces and polymorphic classes

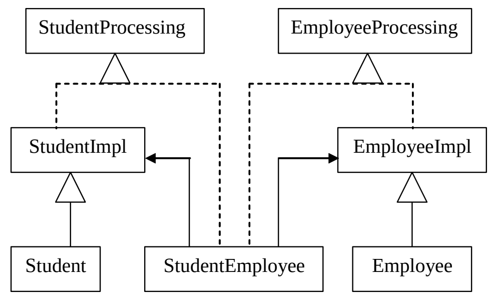
Single and Multiple Inheritance

- Debated topic in OO language design
- C++, C# - multiple inheritance
- Java – simple inheritance
- Main problem of multiple inheritance
- Diamond shape inheritance



Interfaces and polymorphic classes
A Java solution to the inconsistency problem

- Implementation reuse in Java
  - Simple inheritance plus
  - Delegation technique



Interfaces and polymorphic classes
A Java solution to the inconsistency problem

```
public interface StudentProcessing {
  public double getAvgMarks();
  public double calculateAvgMarks();
}
public interface EmployeeProcessing {
  public double getSalary();
  public double calculateSalary();
}
public class StudentImpl implements
               StudentProcessing {
  protected double avgMarks;
  public double calculateAvgMarks() {… }
  public double getAvgMarks() {… }

}
public class EmployeeImpl implements
               EmployeeProcessing {
  protected double salary;
  public double calculateSalary() { … }
  public double getSalary() { … }
}
public class Student extends StudentImpl {
  // … getAvgMarks() and calculateAvgMarks()
  // are inherited … avgMarks field is inherited
}
public class Employee extends EmployeeImpl {
  // … getSalary() and calculateSalary()  are
  // inherited … salary field is inherited
}
```

```
// Below – reuse through delegation
// (see highlighted methods)
// consistency is preserved
public class StudentEmployee implements
       StudentProcessing, EmployeeProcessing {
  protected StudentImpl studentImpl;
  protected EmployeeImpl employeeImpl;
  public StudentEmployee() {
    studentImpl = new StudentImpl();
    employeeImpl = new EmployeeImpl();
    // …other constructor statements
  }
  public double getAvgMarks() {
    // delegation is used
    return studentImpl.getAvgMarks();
  }
  public double getSalary() {
    // delegtion is used
    return employeeImpl.getSalary();
  }
  // … similar for calculateAvgMarks() and
  //   calculateSalary()
}
```

Interfaces vs. Abstract classes

```
public abstract class AC {
  public abstract void m();
}

public interface IF {
  public void m();
}
```

Common features

- Both define a contract that must be implemented by a class
- A concrete class that extends AC must define m()
- A class that implements IF must define m()

Main differences

Content

Abstract classes
- Methods
- Instances
- Constructors
- A partial implementation

Interfaces
- Public methods
- Constants
- No implementation allowed

Features

Interfaces
- Interface inheritance
- Allow for multiple interface extensions
- A class can implement multiple interfaces

Class
- Can only extend ("implementation inheritance") one other class.

Interfaces vs. Abstract classes

Comparison criteria

- system evolution
- *able or can do* vs. *is-a*
- plug-in
- homogeneity
- third party functionality

Interfaces vs. Abstract classes
System evolution

- Abstract classes are easier to evolve over time with less implications over the existing programs
- Consider two operational systems
  - one uses the AC
  - one uses IF
- Scenario - adding a new method m1()
  - Adding the method to AC
  - Adding the method to IF

Interfaces vs. Abstract classes
System evolution

Abstract class approach
-m1() can be implemented
-abstract class is partial implemented

```
public abstract class AC {
  public abstract void m();
  public void m1() { // … implementation }
}
```

AC subclasses
- use m1 as it is defined in the AC
- override it

AC - useful when you need to provide a partial implementation

Note. Also consider an AC implementing an interface

Interface approach

```
public interface IF {
  public void m();
  public void m1();
}
```

All classes that implemented IF should implement m1()
- The existing system is broken (invalidated)
- Changing the interfaces will break a lot of code

Conclusions
-IF are better choices than AC providing you will not modify it
=> when designing interfaces - make sure they won't change very often and very soon

Interfaces vs. Abstract classes
able or *can do* vs. *is-a*

- Interface
  - Is not describing class main role
  - Describes the peripheral class abilities
  - Example
    - Bicycle may implement Recyclable
    - Many other (unrelated) classes may implement Recyclable
- Abstract class
  - defines the core identity of its descendants
  - Example – class Dog
- Implemented interfaces
  - Specifies what a class can do
  - Doesn't specify what a class is.

Interfaces vs. Abstract classes

Plug-in

Interface
- New implementations – no common code with previous implementations
- Start from scratch
- Freedom to implement a totally new internal design

Abstract class
- AC should be used as it is (good or bad)
- Imposes a certain structure to the new implementer

Third party functionality

Interface
- Interface implementation may be added to any existing third party class

Abstract class
- Third party class must be rewritten to extend from the abstract class

Homogenity

Interfaces
- All the various implementations share is the method signatures

Abstract class
- All various implementations are all of a kind and share a common status and behavior

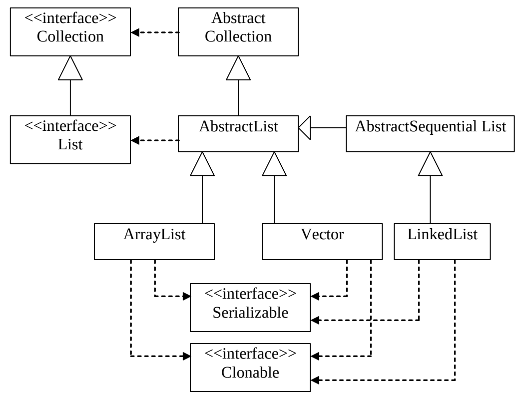
In terms of subclasses
- Abstract class' subclasses are homogeneous
- Interface subclasses are heterogeneous, use interface

Interfaces **and** Abstract classes
JCF Example

Level 0 - Top hierarchy
- Interfaces such as Collection, List
- Describe contracts (behavior specification)

Level 1
- Abstract classes such as AbstractList
- Provide partial implementations

Level 2
- Concrete classes such as ArrayList or Vector
- Define all abstract methods that are not already defined



Benefits of programming
- Much of the implementation is already done in superclasses
- Easy switch between implementations
- Develop parallel implementations

Interfaces vs. Abstract classes
Conclusions

- Pro abstract class - you plan on using inheritance; AC provides a common base class implementation to subclasses
- Pro abstract class - you want to be able to declare non-public members. In an interface, all methods must be public
- Pro abstract class – you plan to add methods in the future; if you add new method headings to an interface, all of the classes that already implement that interface will have to be changed to implement the new methods.
- Pro interface – if you think that the API will not change for a long time
  Pro interface - when you need something similar to multiple inheritance (because a class may implement multiple interfaces)