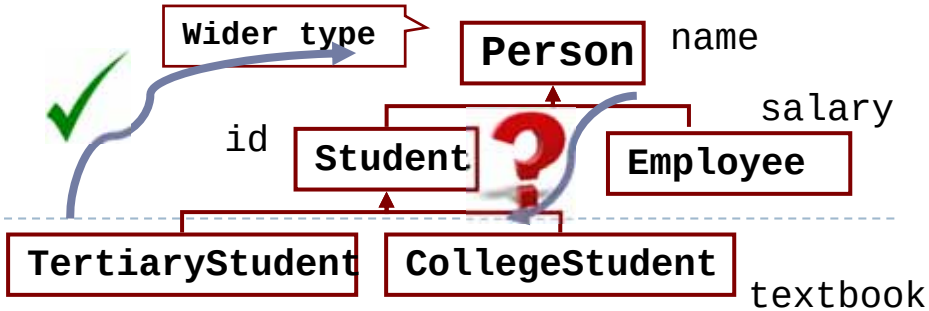
CompSci 230 S2
Object Oriented Software Development

Abstract Classes & Interfaces

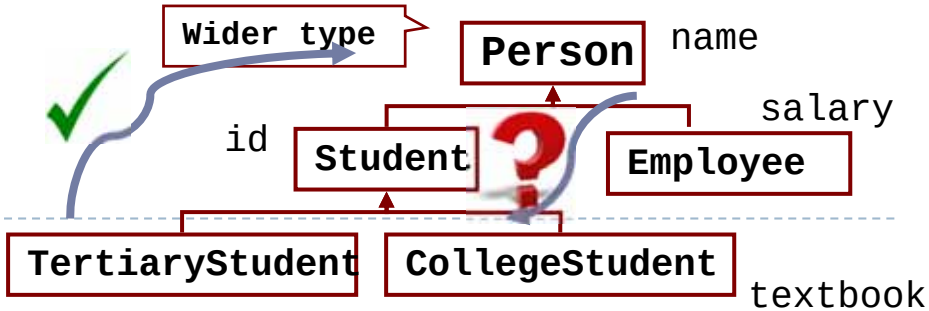
Review



- Casting from Student to Person
  - Student s = new Student(); //s1.id s1.name
  - Person p = (Person) s; //p.name p.id

- Casting from Person to Student
  - Person p1 = new Person(…);
  - Person p2 = new Student(…);
  - Person p3 = new TertiaryStudent(…);
  - Person p4 = new Employee(…);
  - Student s1 = (Student) p1; //CT: //RT: (no id at all)
  - Student s2 = (Student) p2; //CT: //RT:
  - Student s3 = (Student) p3; //CT: //RT: s3.id p3.id
  - Employee e1 = (Employee) p1; //CT: //RT:
  - Employee e2 = (Employee) s; //CT: //RT:
  - Employee e3 = (Employee) p4; //CT: //RT:

Review



- Casting from Student to Person
  - Student s = new Student(); //s1.id s1.name
  - Person p = (Person) s; //p.name p.id
- Casting from Person to Student
  - Person p1 = new Person(…);
  - Person p2 = new Student(…);
  - Person p3 = new TertiaryStudent(…);
  - Person p4 = new Employee(…);
  - Student s1 = (Student) p1; //CT: //RT: (no id at all)
  - Student s2 = (Student) p2; //CT: //RT:
  - Student s3 = (Student) p3; //CT: //RT: s3.id p3.id
  - Employee e1 = (Employee) p1; //CT: //RT:
  - Employee e2 = (Employee) s; //CT: //RT:
  - Employee e3 = (Employee) p4; //CT: //RT:

Review Quizzes

- Given the following classes and declarations, which statements are true?
- Select the three correct answers.
  - The Bar class is a legal subclass of Foo.
  - The statement b.f(); is legal.
  - The statement a.j = 5; is legal.
  - The statement a.g(); is legal.
  - The statement b.i = 3; is legal.

```
class Foo {
    private int i;
    public void f() { /* ... */ }
    public void g() { /* ... */ }
}

class Bar extends Foo {
    public int j;
    public void g() { /* ... */ }
}

// Declarations:
// ...
    Foo a = new Foo();
    Bar b = new Bar();
// ...
```

Review Quizzes

- What will be the result of attempting to compile and run the following program?
  - Select the one correct answer.
    - The program will fail to compile.
    - The program will compile without error and print 0 when run.
    - The program will compile without error and print 1 when run.
    - The program will compile without error and print 2 when run.
    - The program will compile without error and print 3 when run.

```
class A {
    public int f() { return 0; }
    public int g() { return 3; }
}
class B extends A {
    public int f() { return 1; }
    public int g() { return f(); }
}
class C extends B {
    public int f() { return 2; }
}
```

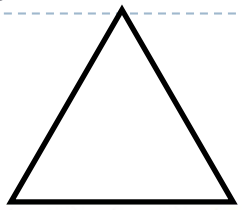
```
A ref1 = new C();
B ref2 = (B) ref1;
System.out.println(ref2.g());
```

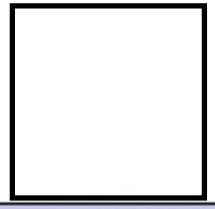
Agenda & Reading

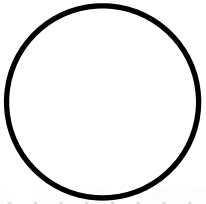
- Topics:
  - Introduction
  - Abstract Classes
  - Interfaces
  - Abstract Classes VS Interfaces
  - Extending Interfaces
  - Final variables, methods, classes
- Reading
  - Java how to program Late objects version (D & D)
    - Chapter 10
  - The Java Tutorial:
    - Abstract Methods and Classes
      - http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html
    - Writing Final Classes and Methods
      - http://docs.oracle.com/javase/tutorial/java/IandI/final.html
    - Interfaces
      - http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html

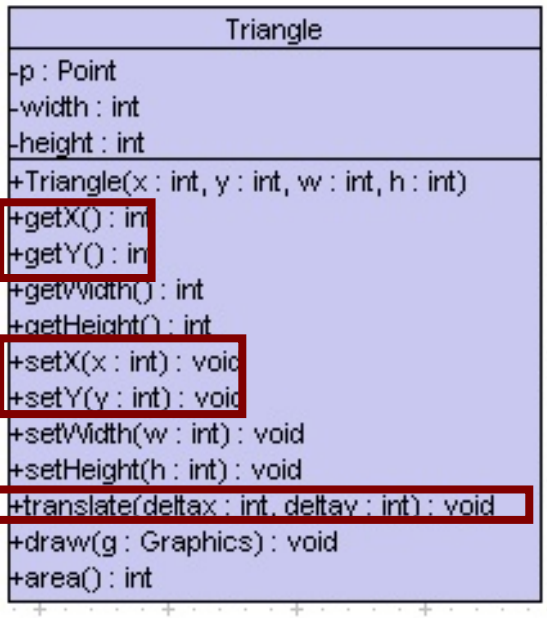
1.Introduction
Design 1: Triangle, Rectangle ...

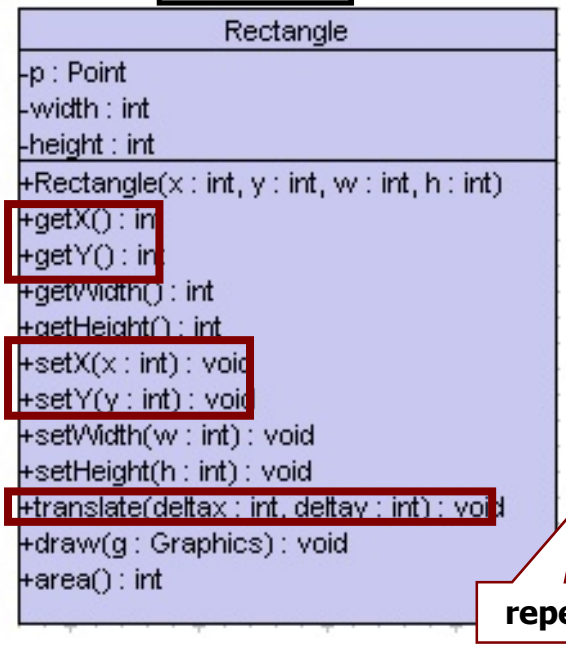
Example: ShapesDemo.java

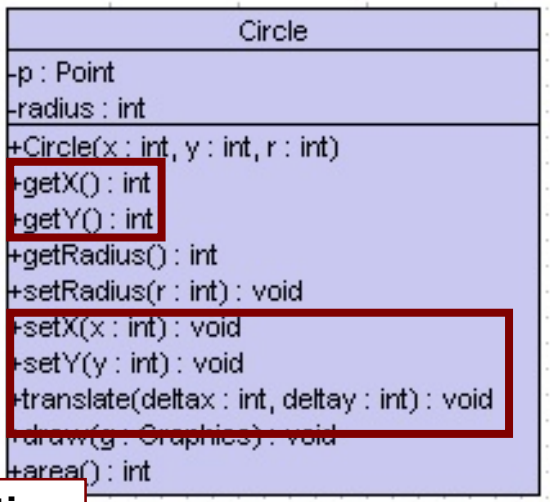










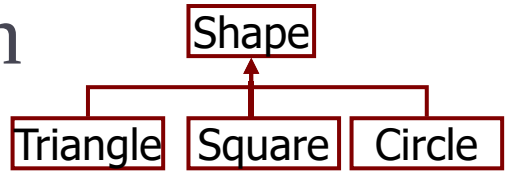


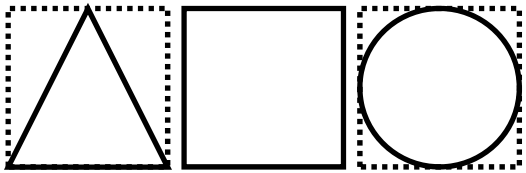
repeating

- General methods:
  - get(), getY(), setX(…), setY(…), translate(…)
- Specific methods: (need to know the type)
  - draw(…), area()

Bad design!

1.Introduction Inheritance





- Superclass: Shape
  - Common
    - Instance variables: Point p (top left corner)
    - Methods: set, get and translate (common to all shapes)
  - Specific methods : draw, area
    - Need to draw/calculate an area for all shapes
    - But this is done differently for different shapes
- Subclasses: Triangle, Square, Circle
- Use an array of Shapes
  - Create an array to store different shapes
- Invoke the draw method in the array of shapes
  - But how do you implement the draw/area method in the superclass?
    - It might not be appropriate to have any code for the area() method in the Shape class at all (you don't know how to calculate the area of an object if you don't know the type of the shape).

```
public class Shape {
 protected Point p; //top-left corner
 public int getX() { return p.x; }
 ...
```

```
Shape[] s = new Shape[3];
S[0] = new Rectangle(…);
…
for (int i =0; i< s.length; i++) {
  s[i].draw(g);
}
```

Create an array

Invoke the draw method

```
public class Shape {
 ...
 public void draw(Graphics g){??????
 public int area(){???????
 ...
```

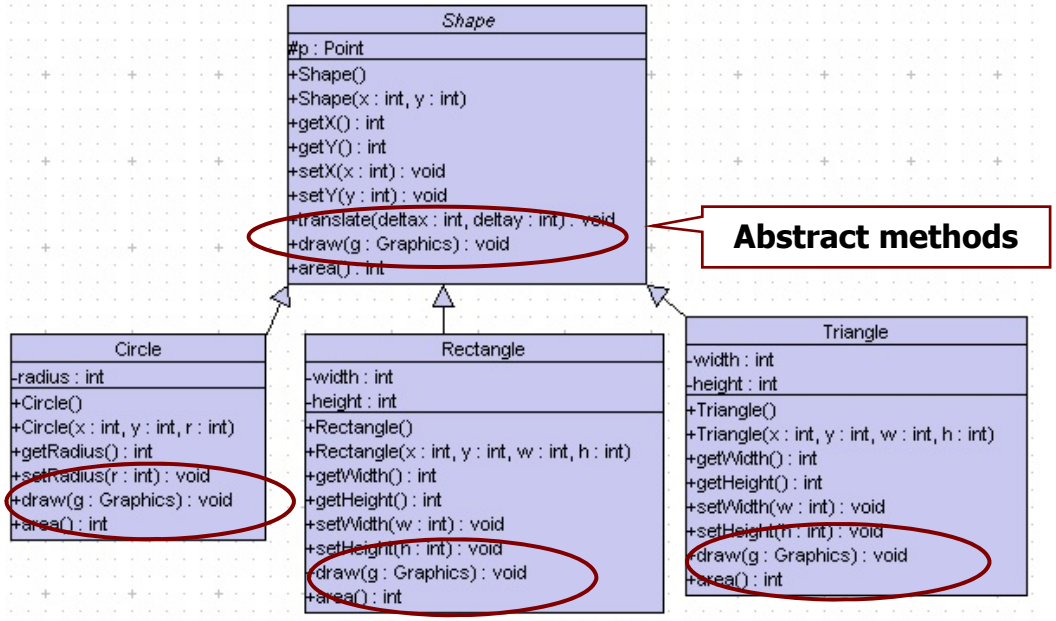
Need to Change!

1.Introduction Abstract classes

```
public abstract class Shape {
…
 public abstract void draw(Graphics g);
 public abstract int area();
}
```

Remove the code inside the area() method and make the method abstract

Example: abstract/ShapesDemo.java



Abstract methods

Provide implementation in subclasses

```
public class Rectangle extends Shape {
  private int width, height;
   public int area() {
    return (width * height);
  }
  ...
```

```
public class Circle extends Shape {
  private int radius;
  public int area() {
    return (int) (Math.PI * radius * radius);
  }
  ...
```

```
public class Triangle extends Shape {
  private int width, height;
  public int area() {
    return (width * height) / 2;
  }
 ...
```

2.Abstract classes
Concrete & Abstract classes

- Concrete classes
  - Classes that can be used to **instantiate** objects are called concrete classes.
  - Such classes provide implementations of **every** method they declare (some of the implementations can be inherited).
- Abstract classes
  - Used only as superclasses in **inheritance hierarchies**, so they are sometimes called abstract superclasses.
  - Cannot be used to instantiate objects—abstract classes are incomplete and they specify only what is common among subclasses.

```
Shape s = new Shape(); <- wrong
```

  - Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects; otherwise, these subclasses, too, will be abstract.

2.Abstract Classes
Abstract Methods

- Abstract classes permit declaration of classes that define only part of an implementation, leaving the subclasses to provide the details:
  - Implementation of **common state** and **behaviour** that will be inherited by subclasses
  - **Declaration** of **generic** behaviours that subclasses must implement
- You make a class abstract by declaring it with keyword **abstract**.

```
public abstract class Shape {
 ...
```

- An abstract class normally contains one or more abstract methods.
  - An abstract method is an instance method with keyword abstract in its declaration, as shown below:

```
public abstract void draw(Graphics g);
public abstract int area();
```

  - Abstract methods do not provide implementations

2.Abstract Classes Rules

- An abstract class is not required to have an abstract method in it.
- But a class that contains abstract methods must be an abstract class even if that class contains some concrete (non-abstract) methods.
- When a class inherits from an abstract class, it must implement every abstract member defined by the abstract class.
  - Failure to implement a superclass's abstract method in a subclass is a compilation error unless the subclass is also declared abstract.
- Constructors and static methods cannot be declared abstract.
- An abstract class cannot be instantiated, however references to an abstract class can be declared.
  - These can hold references to objects of any concrete class derived from those abstract superclasses.

Exercise 1

- Consider the Shape, Rectangle and Circle classes defined previously:
  - Add an abstract getPerimeter() method into the Shape class
  - Implement the getPerimeter() in all the subclasses.

```
public abstract class Shape {
…
 public abstract void draw(Graphics g);
 public abstract int area();
}
```

```
class Rectangle extends Shape {
  private int width, height;
   public int area() {
    return (width * height);
  }
  ...
```

```
class Circle extends Shape {
  private int radius;
  public int area() {
    return (int) (Math.PI * radius * radius);
  }
  ...
```

3. Java Interfaces
Introduction

- If an abstract class contains only abstract method declarations, it should be implemented as an interface instead.
- Interfaces offer a capability requiring that unrelated classes implement a set of common methods.
- Java Interface
  - Defines a protocol of behaviour that can be implemented
  - Defines a set of methods but does not implement them (no actual implementation)
  - is never directly instantiated
- The interface specifies **what** operations a radio must permit users to perform but does not specify **how** the operations are performed.
- A Java interface describes a set of methods that can be called on an object.

3.Java Interfaces
Create an Interface

- An interface declaration begins with the keyword **interface** and contains only constants and abstract methods.
  - All interface members must be public.
  - Interfaces may not specify any implementation details, such as concrete method declarations and instance variables.
  - All methods declared in an interface are implicitly public abstract methods.
  - All fields are implicitly public, static and final.
  - Example:
    - Interface in everyday life:VCR

```
public interface VCR {
    public boolean loadTape(String tapeName);
    public boolean eject();
    public boolean play();
    public boolean pause();
    public boolean stop();
    ...
    public static final int PLAYING=0, PAUSED=1, STOPPED=2,
      FORWARD=3, BACKWARD=4, NOTAPE=5;
 }
```

3. Java Interfaces
Use an interface

```
public interface Bounceable {
  double GRAVITY = 9.8;
  void hitWall(String wallName);
}
```

Methods are implicitly abstract

- To use an interface, a concrete class must specify that it **implements** the interface and must declare each method in the interface with specified signature.
  - Add the **implements** keyword and the name of the interface to the end of your class declaration's first line.
- A class that does not implement all the methods of the interface is an abstract class and must be declared **abstract**.
- Implementing an interface is like signing a contract with the compiler that states, "I will declare all the methods specified by the interface or I will declare my class abstract."

```
public class Basketball implements Bounceable {
  public void hitWall(String wallName) {
    System.out.println("I hit " + wallName);
 }
  public static void main(String[] args) {
    Basketball b = new Basketball();
    b.hitWall("Wall A");
  }
}
```

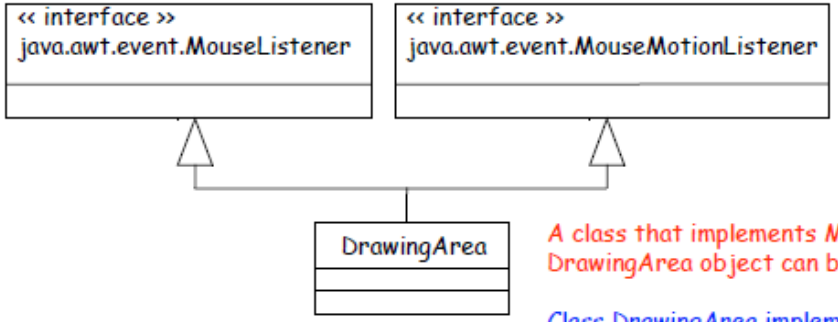
3. Java Interfaces
More Examples - MouseListener

- MouseListener Interface
  - The listener interface for receiving "interesting" mouse events (press, release, click, enter, and exit) on a component

```
public interface MouseListener ... {
 mouseClicked(MouseEvent e)
 mouseEntered(MouseEvent e)
 mouseExited(MouseEvent e)
 mousePressed(MouseEvent e)
 mouseReleased(MouseEvent e)
}
```

```
public interface MouseMotionListener ... {
 mouseDragged(MouseEvent e)
 mouseMoved(MouseEvent e)
}
```

- MouseMotionListener Interface
  - The listener interface for receiving mouse **motion** events on a component



Define methods for describing mouse events. An instance of a class that implements these interfaces can be registered such that its MouseListener and MouseMotionListener methods will be called in response to mouse events.

A class that implements MouseListener and MouseMotionListener. A DrawingArea object can be notified of a user's interaction with the mouse.

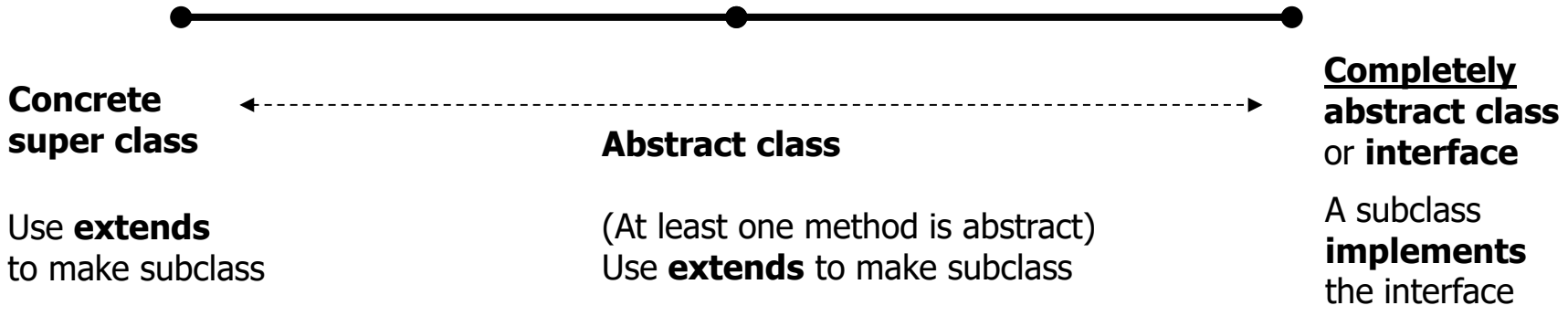
Class DrawingArea implements MouseListener, MouseMotionListener { … }

3. Java Interfaces
is-a relationship

- Once a class **implements** an interface, must provide an implementation of **every method** defined within the interface.
  - A class may implement some additional methods (but these extra methods aren't accessible through this interface)
- All objects of that class have an **is-a** relationship with the interface type, and all objects of the class are guaranteed to provide the functionality described by the interface.
  - This is true of all subclasses of that class as well.
- A class can implement many interfaces.
- In Java, classes are less extendible than interfaces, because a Class can extend at most one other Class ("single inheritance").
- Beware: adding another method to an existing Interface will "break" every current implementation of this Interface!
  - Extension is the preferred way to add new methods to an Interface

3. Java Interfaces
Abstract classes Vs Interfaces

- Abstract classes Vs Interfaces
  - An interface cannot implement any methods, whereas an abstract class can. (i.e. An interface is where all the methods are abstract)
  - A class that implements an interface must either provide definitions for all methods or declare itself abstract
  - Classes can extend only one class, but can implement one or more interfaces
  - Unrelated classes can implement the same interface
- Interface Continuum



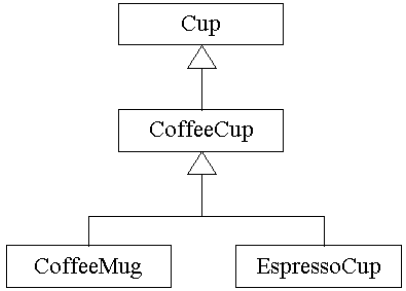
4.Interface in unrelated classes

- An interface is often used when disparate classes (i.e., unrelated classes) need to share common methods and constants.
  - Allows objects of unrelated classes to be processed polymorphically by responding to the same method calls.
  - You can create an interface that describes the desired functionality, then implement this interface in any classes that require that functionality.

4.Interface in unrelated classes Design 1

- Given a family of classes, polymorphism allows you to treat a subclass object as if it were a superclass object
  - create a single method that could wash any kind of cup in your virtual café.

```
public static void prepareACup(Cup cup) {
  ...
  cup.wash();
  ...
```



  - The prepareACup() method can invoke wash() on many different objects

```
class Cup {
  public void wash() {
    ...
```

```
class EspressoCup extends CoffeeCup {
    public void wash() {
    ...
```

```
class CoffeeCup extends Cup {
    public void wash() {
    ...
```

```
class CoffeeMug extends CoffeeCup {
    public void wash() {
    ...
```

4.Interface in unrelated classes Design 2: Washable?

- What if you wanted to have a method that can wash any kind of object ("washable" object)?
  - For example: wash a window, wash your car, or wash a dog
  - Objects don't seem to fit into the same family
  - You may want to…

```
class Dog{
    public void wash() {
    ...
```

```
class Window {
  public void wash() {
    ...
```

```
class Car {
    public void wash() {
    ...
```

```
class Cleaner {
  public static void cleanAnObject(Object obj) {
    if (obj instanceof Cup) {
      ((Cup) obj).wash();

    } else if (obj instanceof Dog) {
      ((Dog) obj).wash();

    } else if ...
```

doesn't use polymorphism

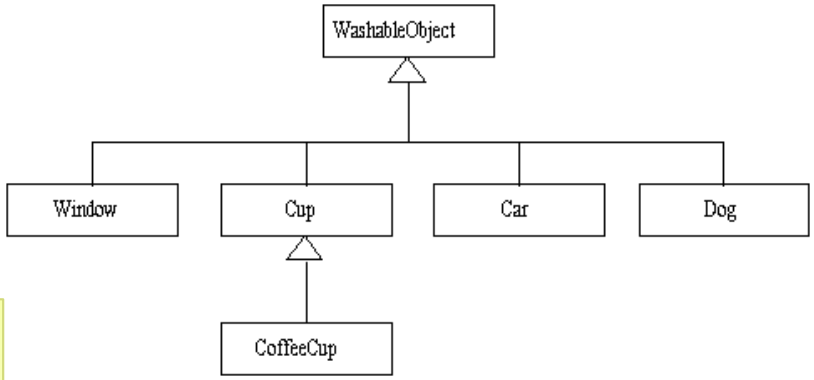
4.Interface in unrelated classes Design 3: WashableObject family

- Four classes -- cups, cars, windows, and dogs--are combined into the WashableObject family
  - Full benefit of polymorphism in the cleanAnObject() method

```
abstract class WashableObject {
  public abstract void wash();
}
```

```
class Window extends WashableObject {
  public void wash() {
    ...
```

```
class Dog extends WashableObject {
    public void wash() {
    ...
```

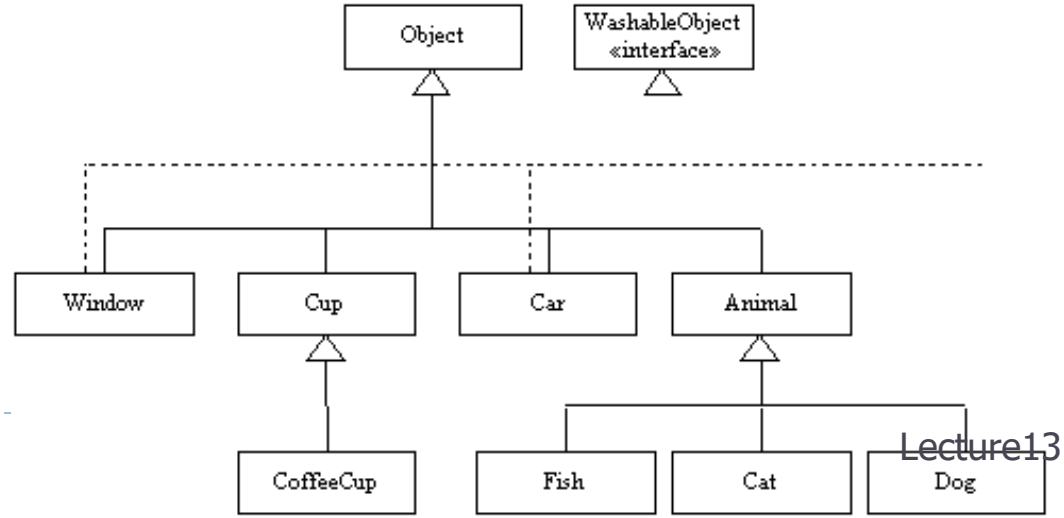



```
class Cleaner {
  public static void cleanAnObject(WashableObject wo) {
    //...
    wo.wash();
```

- However ..
  - What exactly is a washable object? What does one look like? Washable is an adjective, not a noun. It describes a behaviour exhibited by objects, not an object itself!

4.Interface in unrelated classes
Design 4: Washable Interface

- Interfaces in Java allow you to get the benefits of polymorphism without requiring you to build a singly-inherited family of classes.
- Although a class can extend only one other class, it can "implement" multiple interfaces.
- Solution
  - Family of classes for cups, another for animals, one for vehicles (including cars), one for parts of a building etc
  - Then each washable class can implement the Washable interface.



5.Extending Interfaces

- Interfaces can extend other interfaces, which brings rise to **sub-interfaces** and **super-interfaces**
  - Interface extends interface, but a class **cannot** extends an interface.
- Unlike classes, however, an interface can extend more than one class at a time

```
public interface Displayable extends Drawable, Printable {
  //additional constants and abstract methods
  ...
```

- Another good reason to extend interfaces is that INTERFACES CANNOT GROW.
  - Think about when you add methods to a class that's already being used by other classes. It doesn't effect them really, they can still use your class just as before.
  - But, if you add methods to an interface, you break the classes that use this interface, since they now must implement another method.
  - So, when something else is required in an interface, it is best to extend that interface, add the new method and then use the new sub-interface.

5.Extending Interfaces Example 4

```
public interface Bounceable {
  double GRAVITY = 9.8;
  void hitWall(String wallName);
  void AddMethod();
}
```

The new method

- All classes that implement the old Bounceable interface will break because they don't implement all methods of the interface anymore.

```
public interface Bounceable {
  void hitWall(String wallName);
  ...
```

```
public class Basketball implements Bounceable {
  public void hitWall(String wallName) {
    System.out.println("I hit " + wallName);
  }
  ...
```

No AddMethod is defined in the Basketball. Compile-time error

- Solution: write a new sub-Interface
  - Now users of your code can choose to continue to use the old interface or to upgrade to the new interface

```
public interface Bounceable {
  double GRAVITY = 9.8;
  void hitWall(String wallName);
}
```

```
public interface BounceablePlus extends Bounceable {
  void AddMethod();
}
```

```
public class BasketballNew implements BounceablePlus {
  public void hitWall(String wallName) {
    ...
  public void AddMethod() {
    ...
```

Exercise 2

- Implement the Comparable interface to sort Shape objects by the area() method in the Shape class.

```
abstract class Shape implements Comparable<Shape> {
  protected Point p; //top-left corner
  public Shape() {
   p = new Point();
  }
  public Shape(int x, int y) {
    p = new Point(x,y);
  }
  public int compareTo(Shape others) {
    // complete this

  }
 ...
}
```

```
Shape[] s = new Shape[3];
s[0] = new Rectangle(20, 30, 40, 50);
s[1] = new Circle(50, 60, 10);
s[2] = new Rectangle(10, 10, 60, 10);
System.out.println(Arrays.toString(s));
Arrays.sort(s);
System.out.println(Arrays.toString(s));
```

```
[class Rectangle:2000, class Circle:314, class Rectangle:600]
[class Circle:314, class Rectangle:600, class Rectangle:2000]
```

6.Typing Rules

- The typing rules for interfaces are similar to those for classes.
  - A reference variable of interface type T can refer to an instance of any class that implements interface T or a sub-interface of T.
  - Through a reference variable of interface type T, methods defined by T and its super interfaces can be called.

```
interface I1 {
  public void m1();
  public void m2();
}
```

```
class c1 implements I1 {
  public void m1() {}
  public void m2() {}
}
```

```
inteface I2 extends I1 {
  public void m3();
}
```

```
C1 a = new C1();
C1 b = new C1();
I1 p = a;
p.m1();
p.m2();

C2 c = new C2();
I2 q = c;
q.m1();
q.m2()
q.m3();
```

a is a reference variable of type C1
p is a reference variable of type I1

q is a reference variable of type I2

```
class c2 implements I2 {
  public void m1() {}
  public void m2() {}
  public void m3() {}
}
```

6.Typing Rules
The instanceof operator

- You can use the instanceof operator to test an object to see if it implements an interface, before you invoke a method in this interface.
  - This might improve readability and correctness.
  - This might be a hack.
    - Where possible, you should extend classes and interfaces to obtain polymorphic behaviour, rather than making a runtime check.

```
if( b instanceof Bounceable ) {
  b.hitWall( "Wall A" );
}
```

7.Final variables, methods, classes

- The final keyword can be applied to prevent some of the inheritance effects
  - Final field: constant
  - Final argument: cannot change the data within the called method
  - Final method: cannot override method in subclasses
  - Final class: cannot be subclassed (all of its methods are implicitly final as well)

```
class ChessAlgorithm {
. . .
  final void nextMove(ChessPiece pieceMoved, BoardLocation newLocation) {
    ...
  }
}
```

Exercise 3

- Given the following two interfaces:

```
interface Cowboy {
  void draw();  // draw the gun - get ready to pump some iron
  void drink(); // enjoy some beverage
  void burp();  // communicate with other people in the bar (or where ever)
}
```

```
interface Drawable {
  void draw(int x, int y);  // draw the drawable item on the screen
}
```

- Write a person class which implement the two interfaces (the implementation should be very simple, e.g. just a println statement in each method)