Reusing Classes

- Composition
- Inheritance
- Inheritance and Object Creation
- Final Components

OOAD 1999/2000

Claudia Niederée, Joachim W. Schmidt
Software Systems Institute

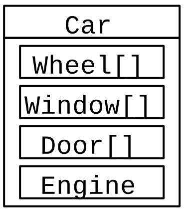
c.niederee@tu-harburg.de http://www.sts.tu-harburg.de

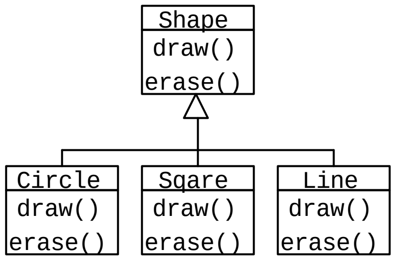
Reusing Classes

When you need a class, you can:

1) Get the perfect one off the shelf (one extreme)
2) Write it completely from scratch (the other extreme)
3) Reuse an existing class with composition
4) Reuse an existing class or class framework with inheritance

Composition vs. Inheritance





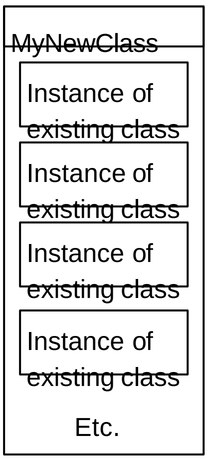
Composition:

- “Has-A” / “Part-Of” relationship
- Code reuse

Inheritance:

- “Is-A” relationship
- interface duplication for interchangeable objects
- Code Reuse

Composition Syntax




```
class Car {
  Engine eng = new Engine(100);
  Wheel[] wheels = new Wheel[4];
  Horn horn = new Horn();
  ...
}
```

- Instances of existing classes as data members
- “Has- A” relationship
- Flexibility: Can change objects at run time!
- Component Creation: In constructor or where defined

```
class Mixer {
  private String brand;
  Mixer(String brand) {
       this.brand = brand;
       System.out.println("Mixer constructed");
  }
  public String toString() { return brand + " Mixer";
  }
}
class Toaster {
  Toaster() {
       System.out.println("Toaster constructed");
  }
  public String toString() { return "Toaster"; }
}

class Knife {
  String purpose;
  Knife(String purpose) {
       this.purpose = purpose;
       System.out.println("Knife constructed");
  }
  public String toString() { return "Knife for " +
                                  purpose; }
}
```


Initializing at point of definition

Initializing in constructor

Delayed initializing

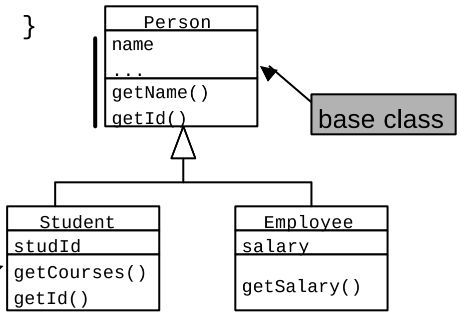
```
public class Kitchen {
  Mixer = new Mixer("Braun");
  Toaster toaster;
  Knife knife;
  int length, width;

  Kitchen() {
    this.toaster = new Toaster();
    this.length = 352;  this.width = 234;
  }
  void print() {
    if( knife == null)
      knife = new Knife("Vegetable");
    System.out.println(" knife " + knife);
    System.out.println(" mixer " + mixer);
    System.out.println(" length = " + length);}
  public static void main( String args[]) {
    Kitchen k = new Kitchen();
    k. print();
  }
}
```

Inheritance (Motivation)

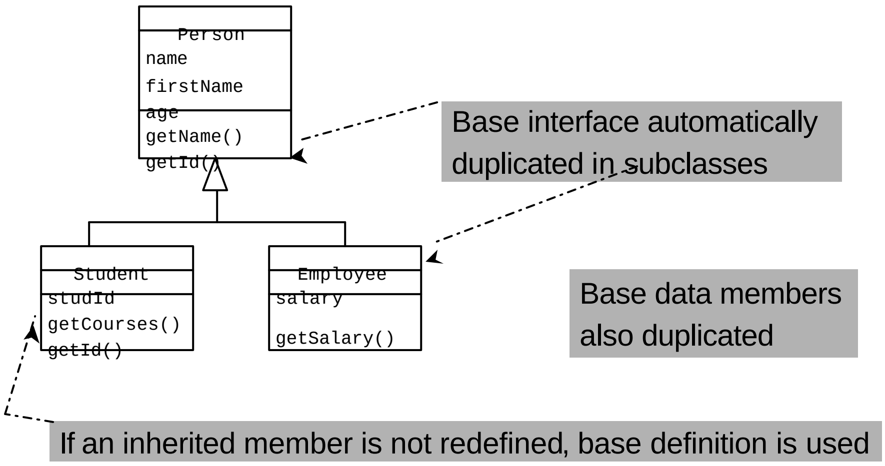
```
class Person{
  String name;
  String firstName;
  int age;
  String getName(){...}
  String getId(){...}
}
class Employee {
  String name;
  String firstName;
  int age;
  Currency salary;
  ...
  String getName(){...}
  String getId(){...}
  String getSalary(){...}
}
```

```
class Student {
  String name;
  String firstName;
  int age;
  String studId;
  String getName(){...}
  String getId(){...}
  String
  getCourses(){...}
}
```



Inheritance Semantic

- “Is-A“ Relationship



Inheritance Syntax

```
class Person{
 String name;
 String firstName;
 int age;
 String getName(){...}
 String getId(){...}
}
class Student extends Person{
 String studId;
 String getId(){...}
 String getCourses(){...}
}
class Employee extends Person{
 Currency salary;
 String getSalary(){...}
}
```


What is possible in the Subclass

- redefine inherited methods (`getId`) method overriding
  - same method name
  - same argument type list
  - same return type
- add new methods (`getCourses`)
- add new data members (`studId`)

```java
class CarWash {
  private String s = new String("CarWash");
  public void append( String a) { s += a; }
  public void wash() { append(" wash the car"); }
  public void foam() { append(" foam the car"); }
  public void dry()  { append(" dry the car"); }
  public void printProtocol() {
         System.out.println( s ); }
  public static void main( String args[]) {
    CarWash cw = new Carwash();
    cw.foam(); cw.wash(); cw.dry();
    cw.printProtocol();
  }
}
```



```java
public class ComfortCarWash extends CarWash {
  // Change a method (Method Overriding):
  public void foam() {
    append("Play some music");
    super.foam(); // Call base- class version
  }
  // Add a method to the class:
  public void wax() { append("Wax the car"); }
  // Test the new class:
  public static void main( String args[]) {
    ComfortCarWash ccw = new ComfortCarWash();
    ccw.foam();
    ccw.wash();
    ccw.dry();
    ccw.wax();
    ccw.printProtocol();
    System.out.println(" Testing base class:");
    CarWash.main(args);
  }
}
```

Creating a Subclass Object

- Java automatically calls default constructors of the base class

```
class Art {
  Art() {
     System.out.println(" Art constructor"); }}
class Drawing extends Art {
  Drawing() {
    System.out.println(" Drawing constructor"); }}
public class Cartoon extends Drawing {
  Cartoon() {
    System.out.println(" Cartoon constructor"); }
public static void main( String args[]) {
  Cartoon x = new Cartoon();
  }
}
```

Output

```
Art constructor
Drawing constructor
Cartoon constructor
```


Constructors with Arguments

- Base constructor call must happen first
- Use **super** keyword

```
class Game {
  Game(int players) {
    System. out. println("Game constructor");
    System. out. println("Players: " + players);
   }
}
class BoardGame extends Game {
  BoardGame(int players) {
    super(players);
    System. out. println("Game is a Board Game");
 }
}
```

Combining Composition & Inheritance

```
class Plate {
  Plate( int i) {
    System. out. println(" Plate constructor");}
}

class DinnerPlate extends Plate {
  DinnerPlate( int i) {
    super( i);
    System. out. println(" DinnerPlate constructor");
  }
}

class Utensil {
  Utensil(int i) {
    System. out. println(" Utensil constructor");
  }
}
```



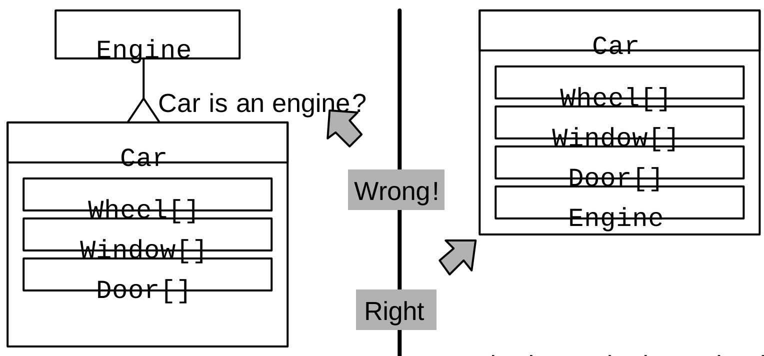
```
class Spoon extends Utensil {
  Spoon( int i) {
    super( i);
    System. out. println(" Spoon constructor");
  }
}

class Fork extends Utensil {
  Fork( int i) {
    super( i);
    System. out. println(" Fork constructor");
  }
}

class Knife extends Utensil {
  Knife( int i) {
    super( i);
    System. out. println(" Knife constructor");
  }
}
```

```java
// A cultural way of doing something:
class Custom {
  Custom( int i) {
    System. out. println(" Custom constructor");
  }
}
public class PlaceSetting extends Custom {
  Spoon sp;
  Fork frk;
  Knife kn;
  DinnerPlate pl;
  PlaceSetting( int i) {
    super( i + 1);
    sp = new Spoon( i + 2);
    frk = new Fork( i + 3);
    kn = new Knife( i + 4);
    pl = new DinnerPlate( i + 5);
    System. out. println(" PlaceSetting constructor");
  }
  public static void main( String args[]) {
    PlaceSetting x = new PlaceSetting( 9); }}
```


Choosing Composition vs. Inheritance



- You start an Engine and you start a car, so... ?

- Inheritance is determined at compile time; member binding can be delayed until run time
- In general, prefer composition to inheritance as a first choice

```java
class Engine {
  public void start() {}
  public void rev() {}
  public void stop() {}
}

class Wheel {
  public void inflate( int psi) {}
}

class Window {
  public void rollup() {}
  public void rolldown() {}
}

class Door {
  public Window window = new Window();
  public void open() {}
  public void close() {}
}
```



```java
public class Car {
  public Engine engine = new Engine();
  public Wheel wheel[] = new Wheel[ 4];
  public Door left = new Door(),
       right = new Door(); // 2- door
  Car() {
    for( int i = 0; i < 4; i++)
      wheel[ i] = new Wheel();
  }
  public static void main( String args[]) {
    Car car = new Car();
    car. left. window. rollup();
    car. wheel[ 0]. inflate( 72);
  }
}
```

The **final** Keyword

- Slightly different meanings depending on context
- “This cannot be changed”
- Two reasons for using it
  - Design
  - Efficiency
- **final** fields
- **final** parameters for methods
- **final** methods
- **final** class

final Data Members

- **final** primitives = constants

```
final int i1 = 9;
```

1) Compile- time constant
   May be “folded” into a calculation by the compiler
2) Run- time constant
   Cannot be changed from value that it’s initialized to

```
final int i4 = (int)( Math. random()* 20);
```

- **final static**: only one per class, initialized at the time the class is loaded, cannot be changed.
- **final** handles: cannot be re- bound to other objects

final Methods

- Put a “lock” on a method to prevent any inheriting class from overriding it (design).
- May limit use of the class (cannot override)
- makes method execution more efficient
- **private** methods are implicitly **final**

- **final** parameters: You are not allowed to assign a new value to the parameter inside the method

final Classes

- Cannot inherit from **final** class
- All methods are implicitly **final**
- Data members may be **final** or not, as you choose