Ollscoil na Gaillimhe
University of Galway

CT326 Programming III

LECTURE 1

OVERLOADING CONSTRUCTORS
ABSTRACT METHODS
POLYMORPHISM

DR ADRIAN CLEAR
SCHOOL OF COMPUTER SCIENCE

Acknowledgement

These notes are adapted from material kindly provided by Dr Des Chambers.

Objectives for today

- Revise important concepts of object-orientation in Java
- Understand how to overload constructors
- Understand what abstract classes are and how to code them
- Demonstrate polymorphism through inheritance and abstract methods

Java - Textbook for this course

- Java – How to Program by Deitel & Deitel
  - Available in University Bookshop

Java - Revision

- Java uses a class to represent objects.
- An object is a thing upon which your application performs different operations.
- A class contains members - these may be:
  - Information (or data) often called class variables.
  - Functions (methods) that operate on the data.
- Each class has a unique name.
- To create an instance of a class variable, you must use the *new* operator.

Using Overloaded Constructors

- Overloaded constructors
  - Methods (in same class) may have same name
  - Must have different parameter lists

```
1  // Fig. 8.6: Time2.java
2  // Time2 class definition with overloaded constructors.
3  package com.deitel.jhtp4.ch08;
4
5  // Java core packages
6  import java.text.DecimalFormat;
7
8  public class Time2 extends Object {
9     private int hour;     // 0 - 23
10    private int minute;   // 0 - 59
11    private int second;   // 0 - 59
12
13    // Time2 constructor initializes each instance variable
14    // to zero. Ensures that Time object starts in a
15    // consistent state.
16    public Time2()
17    {
18       setTime( 0, 0, 0 );
19    }
20
21    // Time2 constructor: hour supplied, minute and second
22    // defaulted to 0
23    public Time2( int h )
24    {
25       setTime( h, 0, 0 );
26    }
27
28    // Time2 constructor: hour and minute supplied, second
29    // defaulted to 0
30    public Time2( int h, int m )
31    {
32       setTime( h, m, 0 );
33    }
34
```

Default constructor has no arguments

Overloaded constructor has one **int** argument

Second overloaded constructor has two **int** arguments

Time2.java

Lines 16-19
Default constructor has no arguments

Lines 23-26
Overloaded constructor has one **int** argument

Lines 30-33
Second overloaded constructor has two **int** arguments

```
35      // Time2 constructor: hour, minute and second supplied
36      public Time2( int h, int m, int s )
37      {
38         setTime( h, m, s );
39      }
40
41      // Time2 constructor: another Time2 object supplied
42      public Time2( Time2 time )
43      {
44         setTime( time.hour, time.minute, time.second );
45      }
46
47      // Set a new time value using universal tim
48      // validity checks on data. Set invalid val
49      public void setTime( int h, int m, int s )
50      {
51         hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
52         minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
53         second = ( ( s >= 0 && s < 60 ) ? s : 0 );
54      }
55
56      // convert to String in universal-time format
57      public String toUniversalString()
58      {
59         DecimalFormat twoDigits = new DecimalFormat( "00" );
60
61         return twoDigits.format( hour ) + ":" +
62            twoDigits.format( minute ) + ":" +
63            twoDigits.format( second );
64      }
65
66      // convert to String in standard-time format
67      public String toString()
68      {
69         DecimalFormat twoDigits = new DecimalFormat( "00" );
```

Third overloaded constructor has three **int** arguments

Fourth overloaded constructor has **Time2** argument

Time2.java

Lines 36-39
Third overloaded constructor has three **int** arguments

Lines 42-45
Fourth overloaded constructor has **Time2** argument

```
70
71          return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) +
72             ":" + twoDigits.format( minute ) +
73             ":" + twoDigits.format( second ) +
74             ( hour < 12 ? " AM" : " PM" );
75       }
76
77    }  // end class Time2
```

Time2.java

```
1    // Fig. 8.7: TimeTest4.java
2    // Using overloaded constructors
3
4    // Java extension packages
5    import javax.swing.*;
6
7    // Deitel packages
8    import com.deitel.jhtp4.ch08.Time2;
9
10   public class TimeTest4 {
11
12      // test constructors of class Time2
13      public static void main( String args[] )
14      {
15         Time2 t1, t2, t3, t4, t5, t6;
16
17         t1 = new Time2();                  // 00:00:00
18         t2 = new Time2( 2 );               // 02:00:00
19         t3 = new Time2( 21, 34 );          // 21:34:00
20         t4 = new Time2( 12, 25, 42 );      // 12:25:42
21         t5 = new Time2( 27, 74, 99 );      // 00:00:00
22         t6 = new Time2( t4 );              // 12:25:42
23
24         String output = "Constructed with: " +
25            "\nt1: all arguments defaulted" +
26            "\n      " + t1.toUniversalString() +
27            "\n      " + t1.toString();
28
29         output += "\nt2: hour specified; minute and " +
30            "second defaulted" +
31            "\n      " + t2.toUniversalString() +
32            "\n      " + t2.toString();
33
```

Declare six references to **Time2** objects

Instantiate each **Time2** reference using a different constructor

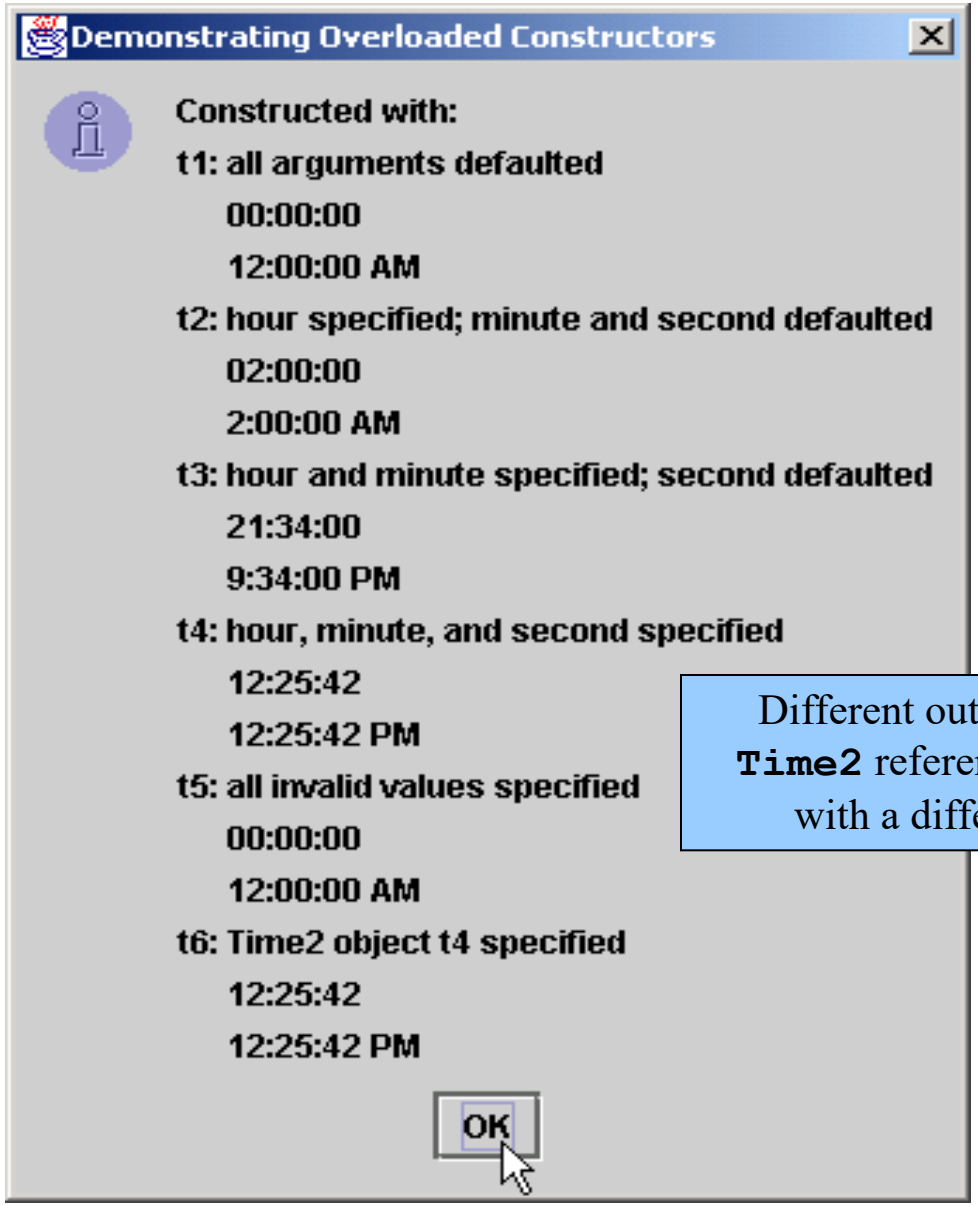
TimeTest4.java

Line 15
Declare six references to **Time2** objects

Lines 17-22
Instantiate each **Time2** reference using a different constructor

TimeTest4.java

```
34          output += "\nt3: hour and minute specified; " +
35             "second defaulted" +
36             "\n      " + t3.toUniversalString() +
37             "\n      " + t3.toString();
38
39          output += "\nt4: hour, minute, and second specified" +
40             "\n      " + t4.toUniversalString() +
41             "\n      " + t4.toString();
42
43          output += "\nt5: all invalid values specified" +
44             "\n      " + t5.toUniversalString() +
45             "\n      " + t5.toString();
46
47          output += "\nt6: Time2 object t4 specified" +
48             "\n      " + t6.toUniversalString() +
49             "\n      " + t6.toString();
50
51          JOptionPane.showMessageDialog( null, output,
52             "Demonstrating Overloaded Constructors",
53             JOptionPane.INFORMATION_MESSAGE );
54
55          System.exit( 0 );
56       }
57
58    }  // end class TimeTest4
```



`TimeTest4.java`

Different outputs, because each **`Time2`** reference was instantiated with a different constructor

Different outputs, because each **`Time2`** reference was instantiated with a different constructor

Java – Class Inheritance

- When your applications use inheritance, you use a *super* class to derive a new class:
  - The new class inherits the *super* class members.
- To initialise class members for an extended class (called a subclass), application invokes the *super* class and subclass constructors.
- Use the *this* and *super* keywords to resolve.
- There are three types of members:
  - **public, private and protected**

Java – Access Level Specifiers

	Class	Package	Subclass	World
private	Y	N	N	N
no specifier	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

Case Study: A Payroll System Using Polymorphism

- Abstract methods and polymorphism
  - Abstract superclass **`Employee`**
    - Method **`earnings`** applies to all employees
    - Person's earnings dependent on type of **`Employee`**
  - Concrete **`Employee`** subclasses declared **`final`**
    - **`Boss`**
    - **`CommissionWorker`**
    - **`PieceWorker`**
    - **`HourlyWorker`**
  - **`Chapter 10 of Deitels Book covers a similar example and has the code on the CD.`**

```
1    // Fig. 9.16: Employee.java
2    // Abstract base class Employee.
3
4    public abstract class Employee {
5       private String firstName;
6       private String lastName;
7
8       // constructor
9       public Employee( String first, String last )
10      {
11         firstName = first;
12         lastName = last;
13      }
14
15      // get first name
16      public String getFirstName()
17      {
18         return firstName;
19      }
20
21      // get last name
22      public String getLastName()
23      {
24         return lastName;
25      }
26
27      public String toString()
28      {
29         return firstName + ' ' + lastName;
30      }
31
```

abstract class cannot be instantiated

abstract class can have instance data and non **abstract** methods for subclasses

abstract class can have constructors for subclasses to initialize inherited data

Employee.java

abstract class cannot be instantiated

5-6 and 16-30
ract class can have instance data and non**abstract** methods for subclasses

Lines 9-13
abstract class can have constructors for subclasses to initialize inherited data

```
32      // Abstract method that must be implemented for each
33      // derived class of Employee from which objects
34      // are instantiated.
35      public abstract double earnings();
36
37   }  // end class Employee
```

Subclasses must implement **`abstract`** method

`Employee.java`

Line 35
Subclasses must implement **`abstract`** method

```
1  // Fig. 9.17: Boss.java
2  // Boss class derived from Employee.
3
4  public final class Boss extends Employee {
5     private double weeklySalary;
6
7     // constructor for class Boss
8     public Boss( String first, String last, double salary )
9     {
10       super( first, last );  // call superclass constructor
11       setWeeklySalary( salary );
12    }
13
14    // set Boss's salary
15    public void setWeeklySalary( double salary )
16    {
17       weeklySalary = ( salary > 0 ? salary : 0 );
18    }
19
20    // get Boss's pay
21    public double earnings()
22    {
23       return weeklySalary;
24    }
25
26    // get String representation of Boss's name
27    public String toString()
28    {
29       return "Boss: " + super.toString();
30    }
31
32 }  // end class Boss
```

Boss is an **Employee** subclass

Boss inherits **Employee**'s **public** methods (except for constuctor)

Explicit call to **Employee** constructor using **super**

Required to implement **Employee**'s method **earnings** (polymorphism)

Line 4
Boss is an **Employee** subclass

Line 4
Boss inherits **Employee**'s **public** methods (except for constuctor)

Line 10
Explicit call to **Employee** constructor using **super**

Lines 21-24
Required to implement **Employee**'s method **earnings** (polymorphism)

```
1   // Fig. 9.18: CommissionWorker.java
2   // CommissionWorker class derived from Empl
3
4   public final class CommissionWorker extends Employee {
5      private double salary;      // base salary per week
6      private double commission;  // amount per item sold
7      private int quantity;       // total items sold for week
8
9      // constructor for class CommissionWorker
10     public CommissionWorker( String first, String last,
11        double salary, double commission, int quantity )
12     {
13        super( first, last );  // call superclass constructor
14        setSalary( salary );
15        setCommission( commission );
16        setQuantity( quantity );
17     }
18
19     // set CommissionWorker's weekly base salary
20     public void setSalary( double weeklySalary )
21     {
22        salary = ( weeklySalary > 0 ? weeklySalary : 0 );
23     }
24
25     // set CommissionWorker's commission
26     public void setCommission( double itemCommission )
27     {
28        commission = ( itemCommission > 0 ? itemCommission : 0 );
29     }
30
```

`CommissionWorker` is an **`Employee`** subclass

Explicit call to **`Employee`** constructor using **`super`**

`CommissionWorker.java`

Line 4
`CommissionWorker` is an **`Employee`** subclass

Line 13
Explicit call to **`Employee`** constructor using **`super`**

```
31       // set CommissionWorker's quantity sold
32       public void setQuantity( int t
33       {
34          quantity = ( totalSold > 0
35       }
36
37       // determine CommissionWorker's earnings
38       public double earnings()
39       {
40          return salary + commission * quantity;
41       }
42
43       // get String representation of CommissionWorker's name
44       public String toString()
45       {
46          return "Commission worker: " + super.toString();
47       }
48
49    }  // end class CommissionWorker
```

Required to implement **Employee**'s method **earnings**; this implementation differs from that in **Boss**

CommissionWorker.java

Lines 38-41
Required to implement **Employee**'s method **earnings**; this implementation differs from that in **Boss**

```
1  // Fig. 9.19: PieceWorker.java
2  // PieceWorker class derived from Employee
3
4  public final class PieceWorker extends Employee {
5     private double wagePerPiece; // wage per piece output
6     private int quantity;        // output for week
7
8     // constructor for class PieceWorker
9     public PieceWorker( String first, String last,
10       double wage, int numberOfItems )
11    {
12       super( first, last );  // call superclass constructor
13       setWage( wage );
14       setQuantity( numberOfItems);
15    }
16
17    // set PieceWorker's wage
18    public void setWage( double wage )
19    {
20       wagePerPiece = ( wage > 0 ? wage : 0 );
21    }
22
23    // set number of items output
24    public void setQuantity( int numberOfItems )
25    {
26       quantity = ( numberOfItems > 0 ? numberOfItems : 0 );
27    }
28
29    // determine PieceWorker's earnings
30    public double earnings()
31    {
32       return quantity * wagePerPiece;
33    }
34
```

PieceWorker is an **Employee** subclass

Explicit call to **Employee** constructor using **super**

Implementation of **Employee**'s method **earnings**; differs from that of **Boss** and **CommissionWorker**

PieceWorker.java

Line 4
PieceWorker is an **Employee** subclass

Line 12
Explicit call to **Employee** constructor using **super**

Lines 30-33
Implementation of **Employee**'s method **earnings**; differs from that of **Boss** and **CommissionWorker**

```
35       public String toString()
36       {
37          return "Piece worker: " + super.toString();
38       }
39
40    }  // end class PieceWorker
```

PieceWorker.java

```
1   // Fig. 9.20: HourlyWorker.java
2   // Definition of class HourlyWorker
3
4   public final class HourlyWorker extends Employee {
5      private double wage;   // wage per hour
6      private double hours;  // hours worked for week
7
8      // constructor for class HourlyWorker
9      public HourlyWorker( String first, String last,
10        double wagePerHour, double hoursWorked )
11     {
12        super( first, last );   // call superclass constructor
13        setWage( wagePerHour );
14        setHours( hoursWorked );
15     }
16
17     // Set the wage
18     public void setWage( double wagePerHour )
19     {
20        wage = ( wagePerHour > 0 ? wagePerHour : 0 );
21     }
22
23     // Set the hours worked
24     public void setHours( double hoursWorked )
25     {
26        hours = ( hoursWorked >= 0 && hoursWorked < 168 ?
27           hoursWorked : 0 );
28     }
29
30     // Get the HourlyWorker's pay
31     public double earnings() { return wage * hours; }
32
```

HourlyWorker is an **Employee** subclass

Explicit call to **Employee** constructor using **super**

Implementation of **Employee**'s method **earnings**; differs from that of other **Employee** subclasses

HourlyWorker.java

Line 4
PieceWorker is an **Employee** subclass

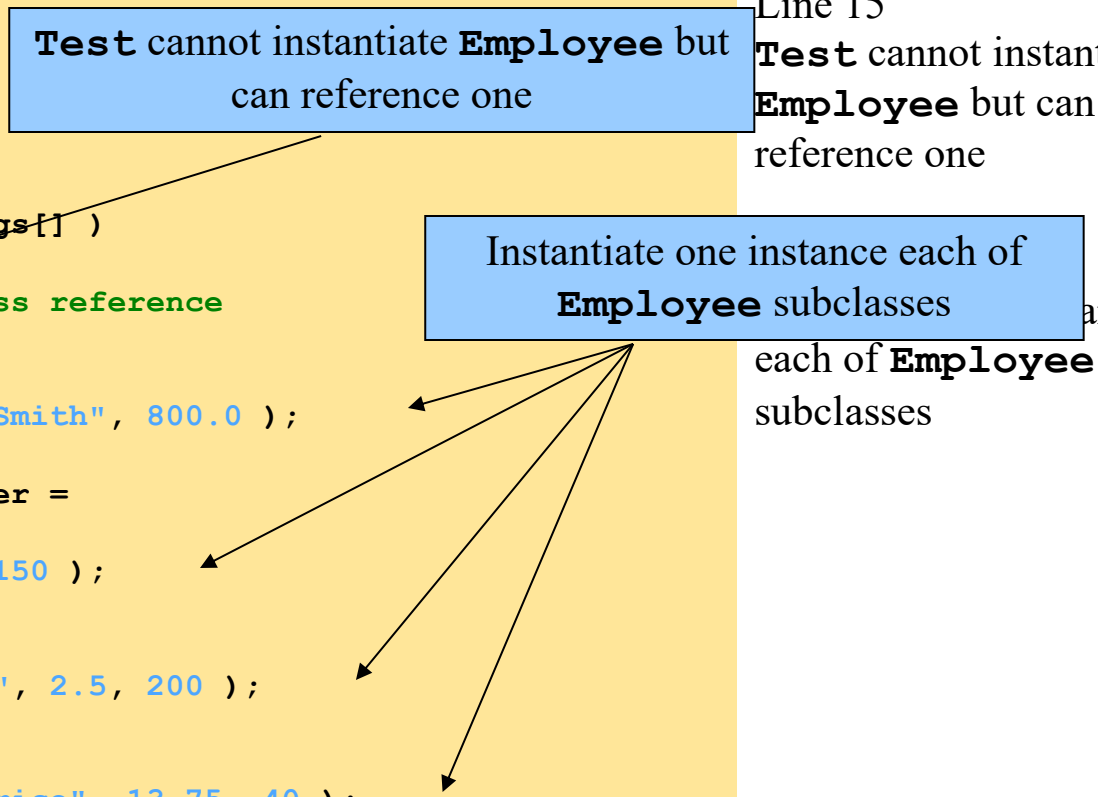
Line 12
Explicit call to **Employee** constructor using **super**

Line 31
Implementation of
od
from
yee
subclasses

```
33       public String toString()
34       {
35          return "Hourly worker: " + super.toString();
36       }
37
38    }  // end class HourlyWorker
```

HourlyWorker.java

```
1    // Fig. 9.21: Test.java
2    // Driver for Employee hierarchy
3
4    // Java core packages
5    import java.text.DecimalFormat;
6
7    // Java extension packages
8    import javax.swing.JOptionPane;
9
10   public class Test {
11
12      // test Employee hierarchy
13      public static void main( String args[] )
14      {
15         Employee employee;  // superclass reference
16         String output = "";
17
18         Boss boss = new Boss( "John", "Smith", 800.0 );
19
20          CommissionWorker commisionWorker =
21            new CommissionWorker(
22            "Sue", "Jones", 400.0, 3.0, 150 );
23
24        PieceWorker pieceWorker =
25          new PieceWorker( "Bob", "Lewis", 2.5, 200 );
26
27        HourlyWorker hourlyWorker =
28           new HourlyWorker( "Karen", "Price", 13.75, 40 );
29
30       DecimalFormat precision2 = new DecimalFormat( "0.00" );
31
```



Test.java

Line 15
Test cannot instantiate **Employee** but can reference one

ance each of **Employee** subclasses

```
32       // Employee reference to a Boss
33       employee = boss;
34
35       output += employee.toString() + " earned $" +
36          precision2.format( employee.earnings() ) + "\n" +
37          boss.toString() + " earned $" +
38          precision2.format( boss.earnings() ) + "\n";
39
40       // Employee reference to a CommissionWorker
41       employee = commissionWorker;
42
43       output += employee.toString() + " earned $" +
44          precision2.format( employee.earnings() ) + "\n" +
45          commissionWorker.toString() + " earned $" +
46          precision2.format(
47             commissionWorker.earnings() ) + "\n";
48
49       // Employee reference to a PieceWorker
50       employee = pieceWorker;
51
52       output += employee.toString() + " earned $" +
53          precision2.format( employee.earnings() ) + "\n" +
54          pieceWorker.toString() + " earned $" +
55          precision2.format( pieceWorker.earnings() ) + "\n";
56
```

Use **Employee** to reference **Boss**

Method **employee.earnings** dynamically binds to method **boss.earnings**

Do same for **CommissionWorker** and **PieceWorker**

Test.java

to

Line 36
Method **employee.earnings**
to
boss.earnings

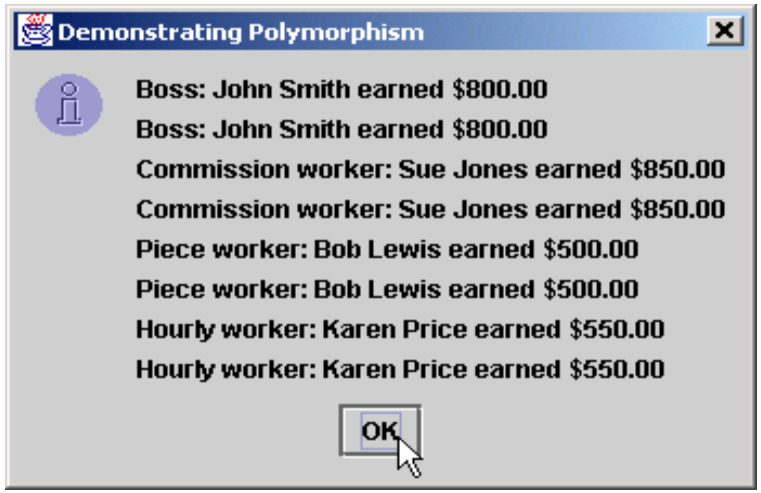
Lines 41-55
Do same for **CommissionWorker** and **PieceWorker**

```
57        // Employee reference to an HourlyWorker
58        employee = hourlyWorker;
59
60        output += employee.toString() + " earned $" +
61        precision2.format( employee.earnings() ) + "\n" +
62             hourlyWorker.toString() + " earned $" +
63             precision2.format( hourlyWorker.earnings() ) + "\n";
64
65          JOptionPane.showMessageDialog( null, output,
66             "Demonstrating Polymorphism",
67             JOptionPane.INFORMATION_MESSAGE );
68
69          System.exit( 0 );
70       }
71
72    }  // end class Test
```

Repeat for **HourlyWorker**

Test.java

Lines 58-63
Repeat for
HourlyWorker



Next time…

- A practical example of using using the command line and a text editor to develop Java programs
- Common programming errors and how to address them