



CSC 308: Java Programming

Advanced Java

Denis L. Nkweteyim



Outline

- **Classes and objects**
- Nested Classes
- Composition
- Package access & Static import
- Inheritance
- Polymorphism
- Interfaces



Classes

- Syntax for minimal class declaration

```
class MyClass {  
    // field, constructor, and  
    // method declarations  
}
```

- Could be more elaborate, and include following components, in order
 - Modifiers such as public, private, and a number of others that you will encounter later
 - The class name, with the initial letter capitalized by convention
 - The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent
 - A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements
 - The class body, surrounded by braces, {}



Classes and Objects

- We use a case study: handling time, to illustrate key issues related to classes and objects
 - Method `setTime` sets hour, minute, second to the value passed to the method, or to 0 if the parameter passed is out of range
 - Methods `toUniversalString` and `toString` create and return two string representations of time
 - They use static String method `String.format` to format the time string
 - When time (a new `Time1`) object is created, the default constructor initializes hour, time, second to their default 0 values
 - Instance variables of class `Time1` are private
 - For objects created from `Time1`, access to these private members from other objects can only be through public methods of class `Time1`
 - Primary objective of a class's public interface (public methods) is to present the class's clients with a view of the services the class provides
 - Hence, private members of a class are directly accessible to the class's clients

Classes and Objects

```
package time1;
public class Time1Test {
    public static void main(String[] args) {
        Time1 time = new Time1(); //invoke Time1's constructor
        //output string representations of time
        System.out.print("Initial Universal Time: ");
        System.out.println(time.toUniversalString());
        System.out.print("Initial Standard Time: ");
        System.out.println(time.toString());
        System.out.println();
        //change time and output updated time
        time.setTime(15, 35, 12);
        System.out.print("Universal Time after setTime: ");
        System.out.println(time.toUniversalString());
        System.out.print("Standard Time after setTime: ");
        System.out.println(time.toString());
        System.out.println();
        //set time to invalid values and output updated time
        time.setTime(99, 99, 99);
        System.out.print("Universal Time after Invalid Setting: ");
        System.out.println(time.toUniversalString());
        System.out.print("Standard Time after Invalid Setting: ");
        System.out.println(time.toString());
        System.out.println();
    }
}
```

Classes and Objects

```
package time1;
public class Time1 {
    private int hour;    //0-23
    private int minute;  //0-59
    private int second;  //0-59
    public void setTime(int h, int m, int s) {
        hour = ((h >= 0 && h < 24) ? h : 0);    //validate hour
        minute = ((m >= 0 && m < 60) ? m : 0); //validate minute
        second = ((s >= 0 && s < 60) ? s : 0); //validate second
    }
    //convert to string in universal time format HH:MM:SS
    public String toUniversalString() {
        return String.format("%02d:%02d:%02d", hour,minute,second);
    }
    //convert to string in universal time format H:MM:SS AM or PM
    public String toString(){
        return String.format("%d:%02d:%02d %s",
            ((hour == 0 || hour == 12)? 12 : hour % 12),
            minute,second, (hour < 12 ? "AM" : "PM"));
    }
}
```

Initial Universal Time: 00:00:00
Initial Standard Time: 12:00:00 AM

Universal Time after setTime: 15:35:12
Standard Time after setTime: 3:35:12 PM

Universal Time after Invalid Setting: 00:00:00
Standard Time after Invalid Setting: 12:00:00 AM



The **this** keyword

- Within an object's method or a constructor, this is a reference to the object in question
- Comes in handy when a field is shadowed by a method or constructor parameter
- Example
- The two code fragments below are equivalent

```
public class Point {  
    public int x = 0;  
    public int y = 0;  
    // a constructor  
    public Point(int a, int b) {  
        x = a;  
        y = b;  
    }  
}
```

```
public class Point {  
    public int x = 0;  
    public int y = 0;  
    // a constructor  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```



Use of **this** within a constructor

- Can be used to call another constructor in the same class
- Example

```
public class Rectangle {  
    private int x, y;  
    private int width, height;  
  
    public Rectangle() {  
        this(0, 0, 0, 0);  
    }  
    public Rectangle(int width, int height) {  
        this(0, 0, width, height);  
    }  
    public Rectangle(int x, int y, int width, int height) {  
        this.x = x;  
        this.y = y;  
        this.width = width;  
        this.height = height;  
    }  
    ...  
}
```




Overloaded constructors

- Like overloaded methods
 - Simply provide multiple constructor declarations with different signatures
 - Recall a signature is determined by the number, type, and order of its parameters

Overloaded constructors example

```
package time2;
public class Time2 {
    private int hour;    //0-23
    private int minute;  //0-59
    private int second;  //0-59
    public Time2()        { this(0,0,0);        }
    public Time2(int h)   { this(h,0,0);        }
    public Time2(int h, int m) { this(h,m,0);    }
    public Time2(int h, int m, int s) { setTime(h, m, s); }
    public Time2(Time2 time){
        this(time.getHour(),time.getMinute(),time.getSecond());
    }
    public void setTime(int h, int m, int s) {
        setHour(h); setMinute(m); setSecond(s);
    }
    public void setHour(int h)    { hour = ((h >= 0 && h < 24) ? h : 0);    }
    public void setMinute(int m) { minute = ((m >= 0 && m < 60) ? m : 0); }
    public void setSecond(int s) { second = ((s >= 0 && s < 60) ? s : 0); }
    public int getHour()    { return hour;    }
    public int getMinute(){ return minute;    }
    public int getSecond(){ return second;    }
    public String toUniversalString() {
        return String.format("%02d:%02d:%02d", hour,minute,second);
    }
    public String toString(){
        return String.format("%d:%02d:%02d %s", ((hour == 0 || hour == 12)? 12
            : hour % 12), minute,second, (hour < 12 ? "AM" : "PM"));
    }
}
```

Overloaded constructors example contd

```
package time2;
public class Time2Test {
    public static void main(String[] args) {
        Time2 t1 = new Time2();           //00:00:00
        Time2 t2 = new Time2(2);          //02:00:00
        Time2 t3 = new Time2(21,34);      //21:34:00
        Time2 t4 = new Time2(12,25,42);    //12:25:42
        Time2 t5 = new Time2(27,74,99);    //00:00:00
        Time2 t6 = new Time2(t4);         //12:25:42
        System.out.println("t1: all arguments defaulted");
        System.out.printf("    %s\n", t1.toUniversalString());
        System.out.printf("    %s\n", t1.toString());
        System.out.println("t2: hour specified; minute & second defaulted");
        System.out.printf("    %s\n", t2.toUniversalString());
        System.out.printf("    %s\n", t2.toString());
        System.out.println("t3: hour & minute specified; second defaulted");
        System.out.printf("    %s\n", t3.toUniversalString());
        System.out.printf("    %s\n", t3.toString());
        System.out.println("t4: hour, minute & second specified");
        System.out.printf("    %s\n", t4.toUniversalString());
        System.out.printf("    %s\n", t4.toString());
        System.out.println("t5: all invalid values specified");
        System.out.printf("    %s\n", t5.toUniversalString());
        System.out.printf("    %s\n", t5.toString());
        System.out.println("t6: Time2 object t4 specified");
        System.out.printf("    %s\n", t6.toUniversalString());
        System.out.printf("    %s\n", t6.toString());
    }
}
```

```
t1: all arguments defaulted
00:00:00
12:00:00 AM
t2: hour specified; minute & second defaulted
02:00:00
2:00:00 AM
t3: hour & minute specified; second defaulted
21:34:00
9:34:00 PM
t4: hour, minute & second specified
12:25:42
12:25:42 PM
t5: all invalid values specified
00:00:00
12:00:00 AM
t6: Time2 object t4 specified
12:25:42
12:25:42 PM
```



Program Notes

- The no-argument constructor `Time2()`
 - Performs the same initialization that a default constructor could have done
 - Why use it then?
 - Because the default constructor is created ONLY if there are no other constructors
 - Our program does have several other constructor
 - Hence, we must explicitly have the zero-argument constructor if our program is to be able to instantiate objects with no parameters
 - i.e., `Time2 t1 = new Time2();`



Outline

- Classes and objects
- **Nested Classes**
- Composition
- Package access & Static import
- Inheritance
- Polymorphism
- Interfaces

Nested Classes

- Nested class

- A class within another class

```
class OuterClass {  
    ...  
    class NestedClass {  
        ...  
    }  
}
```

Nested Class

Non static Class

Member Inner Class

Method-local Inner Class

Anonymous Inner Class

static Class

- Two categories of nested classes

- **Static**

- Declared static

- **Non-static** nested classes

- Also called inner classes.

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
    class InnerClass {  
        ...  
    }  
}
```

www.c4learn.com

Nested
Classes

Static
Nested Class

Non Static

Member
Class

Anonymous
Class

Local Class



Nested Classes

- A nested class
 - Is a member of its enclosing class
 - Can be declared private, public, protected, or package private
 - Recall that outer classes can only be declared public or package private



Why Use Nested Classes

- A way of logically grouping classes that are only used in one place
 - So they are kept together, and that makes their packaging better streamlined
- It increases encapsulation
 - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private (recall encapsulation, by keeping class A's members private, is good software engineering practice)
 - By hiding class B within class A, A's members can be declared private and B can access them
 - In addition, B itself can be hidden from the outside world (by making it private)
- Can improve readability and maintainability of code
 - Nesting small classes within top-level classes places the code closer to where it is used



Access to Outer Class

- Non-static nested classes (inner classes)
 - Have access to other members of the enclosing class, even if they are declared private
- Static nested classes
 - As with class methods and variables, a static nested class is associated with its outer class
 - As with static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class
 - It can use them only through an object reference



Accessing Static Nested Classes

- Accessed using the enclosing class name
 - `OuterClass.StaticNestedClass`
- For example, to create an object for the static nested class, use this syntax:
 - `OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();`



Inner Classes

- As with instance methods and variables
 - An inner class is associated with an instance (i.e., an object) of its enclosing class and has direct access to that object's methods and fields
 - Because an inner class is associated with an instance, it cannot define any static (i.e., class) members itself
- Objects that are instances of an inner class exist within an instance of the outer class (i.e., within an object created from the outer class)
- Hence, given

```
class OuterClass {  
    ...  
    class InnerClass {  
        ...  
    }  
}
```

- An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance



Instantiating an Inner Class

- Must first instantiate the outer class
- Then, create the inner object within the outer object with this syntax

```
OuterClass.InnerClass innerObject =  
    outerObject.new InnerClass();
```

- There are two special kinds of inner classes
 - Local classes and anonymous classes

Shadowing

```
public class ShadowTest {  
    public int x = 0;  
    class FirstLevel {  
        public int x = 1;  
        void methodInFirstLevel(int x) {  
            System.out.println("x = " + x);  
            System.out.println("this.x = " + this.x);  
            System.out.println("ShadowTest.this.x = " +  
ShadowTest.this.x);  
        }  
    }  
    public static void main(String... args) {  
        ShadowTest st = new ShadowTest();  
        ShadowTest.FirstLevel fl = st.new FirstLevel();  
        fl.methodInFirstLevel(23);  
    }  
}
```

Output

x = 23

this.x = 1

ShadowTest.this.x = 0

- If a declaration of a type (such as a member variable or a parameter name) in a particular scope (such as an inner class or a method definition) has the same name as another declaration in the enclosing scope, then the declaration shadows the declaration of the enclosing scope
- You cannot refer to a shadowed declaration by its name alone
- See example on the left



Shadowing

- In the program
 - 3 variables named x, used as
 - Member variable of the class ShadowTest
 - Member variable of the inner class FirstLevel
 - Parameter in the method methodInFirstLevel
 - Defined as a parameter of the method methodInFirstLevel, x shadows the variable of the inner class FirstLevel
 - Consequently, when you use the variable x in the method methodInFirstLevel, it refers to the method parameter (value 23)
 - Keyword **this** (to represent the enclosing scope) is used to refer to the member variable of the inner class FirstLevel
 - `System.out.println("this.x = " + this.x);`
 - Class name is included in the reference to member variables that enclose larger scopes
 - `System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);`



Outline

- Classes and objects
- Nested Classes
- **Composition**
- Package access & Static import
- Inheritance
- Polymorphism
- Interfaces




Composition

- Also known as **has-a** relationship
- This is capability that enables a class to have members that are references to objects of other classes
- Example
 - An employee may have a date of birth, where date of birth is a reference to a date object



Composition Example

```
package composition;
public class Date {
    private int month;
    private int day;
    private int year;
    public Date(int month, int day, int year){
        this.month = checkMonth(month);
        this.year = year;
        this.day = checkDay(day);
        System.out.printf("Date object constructor for date %s\n", this);
    }
    private int checkMonth(int testMonth){
        if ((testMonth > 0) && testMonth <= 12)    return testMonth;
        else {
            System.out.printf("Invalid month (%d) set to 1.", testMonth);
            return 1;
        }
    }
    private int checkDay(int testDay){
        int daysPerMonth[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
        if ((testDay > 0) && testDay <= daysPerMonth[month])    return testDay;
        if (month == 2 && testDay == 29 && (year % 400 == 0 || (year % 4 == 0 && year % 100 != 0)))
            return testDay;
        System.out.printf("Invalid day (%d) set to 1.", testDay);
        return 1;
    }
    public String toString(){
        return String.format("%d/%d/%d", month, day, year);
    }
}
```





Composition Example contd.

```
package composition;
public class Employee {
    private String firstName;
    private String lastName;
    private Date birthDate;
    private Date hireDate;
    public Employee(String fName, String lName, Date dateOfBirth, Date dateOfHire){
        firstName = fName; lastName = lName; birthDate = dateOfBirth; hireDate = dateOfHire;
    }
    public String toString(){
        return String.format("%s, %s Hired: %s  Birthdate: %s",lastName,firstName,hireDate,birthDate);
    }
}
```

```
package composition;
public class EmployeeTest {
    public static void main(String[] args) {
        Date birth = new Date(12, 25, 1950);
        Date hire = new Date(2, 15, 1977);
        Employee employee = new Employee("John", "Doe", birth, hire);
        System.out.println(employee );
    }
}
```

```
Date object constructor for date 12/25/1950
Date object constructor for date 2/15/1977
Doe, John Hired: 2/15/1977  Birthdate: 12/25/1950
```



Program Notes

- Composition
 - Employee object has-a date
- Class date makes checks for validity of day and month, including for leap years; sets day or month to 1 if invalid value is received
- The `toString` method
 - Every object has a `toString` method that returns a string representation of that object, e.g., when using the `%s` formatter
 - However, the inbuilt `toString` method can be overridden if the programmer writes a `toString` method in the class
 - Hence in the Date class, `printf("%s", this);` causes the provided `toString` method to run



Outline

- Classes and objects
- Nested Classes
- Composition
- **Package access & Static import**
- Inheritance
- Polymorphism
- Interfaces



Package Access

- We have already seen the public and private access modifiers
- If no access modifier is specified for a method or variable, the method or variable will have package access
 - i.e., if the program uses multiple classes from the same package, these classes can access each other's package access members directly through references to objects of the appropriate classes
- See example program

Package Access Example

```
package packagedatatest;
public class PackageDataTest {
    public static void main(String[] args) {
        PackageData packageData = new PackageData();
        System.out.printf("After instantiation:\n%s\n", packageData);
        packageData.number = 100;
        packageData.string = "Goodbye";
        System.out.printf("After changing:\n%s\n", packageData);
    }
}
class PackageData {
    int number;
    String string;
    public PackageData(){
        number = 0;
        string = "Hello";
    }
    public String toString(){
        return String.format("number: %d; string: %s", number,string);
    }
}
```

```
After instantiation:
number: 0; string: Hello
After changing:
number: 100; string: Goodbye
```



Program Notes

- There are two classes in the same file (although they could be in separate files)
 - When compiled, two .class files will be created in the same directory
- In class `PackageData`, the instance variables have no access modifiers and so are package-access instance variables
- Notice that object `packageData` created in `main()` is able to modify the `PackageData` instance variable directly

Static import

- Previously, we demonstrated static fields with methods of class `Math`
 - Static fields and methods of class `Math` were preceded by the `Math` class name and a dot.
- A static import declaration enables us to refer to imported static members as if they were declared in the class that uses them
- Forms of static import declaration
 - Single static import
 - Imports a particular static member
 - Syntax: `import static packageName.ClassName.staticMemberName`
 - Static import on demand
 - Imports all static members of a class
 - Syntax: `import static packageName.ClassName.*`

Static import

```
package staticimporttest;
import static java.lang.Math.*;
public class StaticImportTest {
    public static void main(String[] args) {
        System.out.printf("sqrt(900.0) = %.1f\n", sqrt(900.0));
        System.out.printf("ceil(-9.8) = %.1f\n", ceil(-9.8));
        System.out.printf("ceil(9.8) = %.1f\n", ceil(9.8));
        System.out.printf("log(E) = %.1f\n", log(E));
        System.out.printf("cos(0.0) = %.1f\n", cos(0.0));
    }
}

sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
ceil(9.8) = 10.0
log(E) = 1.0
cos(0.0) = 1.0
```



Outline

- Classes and objects
- Nested Classes
- Composition
- Package access & Static import
- **Inheritance**
- Polymorphism
- Interfaces

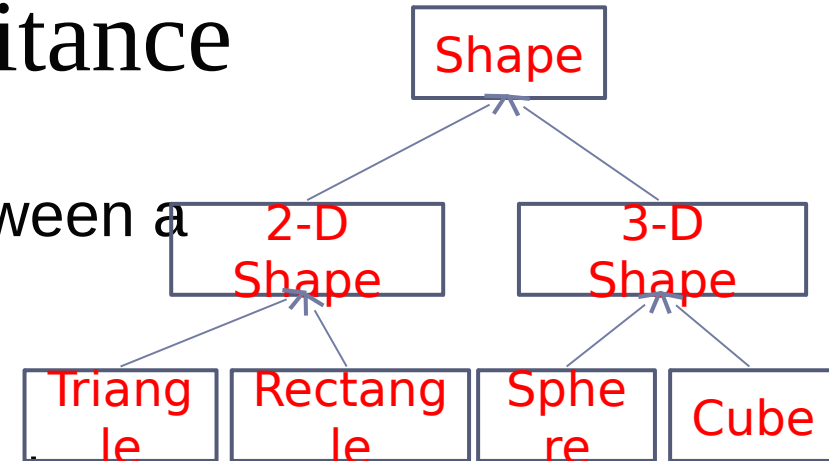


Inheritance

- We already understand the notion of inheritance
- In Java
 - A subclass inherits members (fields, methods, and nested classes) from its superclass
 - Constructors are not members, so they are not inherited by subclasses
 - But the constructor of the superclass can be invoked from the subclass

Inheritance

- There is an **is-a** relationship between a subclass and its superclass(es)
- Examples
 - A triangle **is a** 2-D shape and **is a** shape
 - A cube **is a** 3-D shape and **is a** shape
- Objects of classes that extend a common superclass can be treated as objects of that superclass, but the converse is not true
 - All cars are vehicles but not all vehicles are cars





Inheritance

- A subclass
 - Inherits all of the public and protected members of its parent, no matter what package the subclass is in
 - If the subclass is in the same package as its parent, it also inherits the package-private members of the parent
 - Does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass



What you can do in a subclass

- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus **hiding it** (not recommended).
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus **overriding** it.
- You can write a new static method in the subclass that has the same signature as the one in the superclass, thus **hiding** it.
- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword `super`



Inheritance Examples

- We use a number of examples centred around circles and cylinders to illustrate important issues relating to inheritance
 - Class Circle is the superclass with attribute radius, and methods to compute diameter, area, and circumference
 - Class Cylinder is a subclass of Circle, with length as additional attribute, and additional methods to compute surface area and volume of the cylinder



Example 1: The Circle Class

```
package circle;
import static java.lang.Math.*;
public class Circle {
    private double radius;
    public Circle(double r){ setRadius(r); }
    public Circle(){ radius = 0; }
    public void setRadius(double r){
        radius = (r < 0.0) ? 0.0 : r;
    }
    public double getRadius(){ return radius; }
    public double computeCircleArea(){ return PI*radius*radius; }
    public double computeCircleCircumference(){ return 2*PI*radius; }
    public double computeCircleDiameter(){ return 2*radius; }
    public String toString() {
        return String.format("%s: \n%s: %f\n%s: %f\n%s: %f\n%s: %f",
            "Circle", "Radius", getRadius(),
            "Diameter", computeCircleDiameter(),
            "Area", computeCircleArea(),
            "Circumference", computeCircleCircumference());
    }
}
```




Example 1: The Circle Class contd.

```
package circle;
public class CircleTest {
    public static void main(String[] args) {
        //instantiate circle
        Circle circle = new Circle(5.5);
        System.out.println("Circle information obtained by getter methods");
        System.out.printf("%s:\n%s: %f\n", "Circle","Radius",circle.getRadius());
        System.out.printf("%s: %f\n", "Diameter",circle.computeCircleDiameter());
        System.out.printf("%s: %f\n", "Area",circle.computeCircleArea());
        System.out.printf("%s: %f\n", "Circumference",circle.computeCircleCircumference());
        //change radius
        circle.setRadius(10);
        System.out.println("\nCircle information obtained by toString");
        System.out.printf("%s\n", circle);
    }
}
```

Circle information obtained by getter methods

Circle:

Radius: 5.500000

Diameter: 11.000000

Area: 95.033178

Circumference: 34.557519

Circle information obtained by toString

Circle:

Radius: 10.000000

Diameter: 20.000000

Area: 314.159265

Circumference: 62.831853



Example 1: The Circle Class contd.

- Notes

- The program uses both setter and getter methods and a call to the circle object's `toString` method to display the radius and computed values of the circle
- Notice that when we try to `printf` the circle, its `toString` method is automatically called
- In this example, our `toString` method ***overrides*** the default `toString` method of the object



The Cylinder class

- We first show code for the cylinder class that does not take advantage of inheritance
- Later, we demonstrate how to create subclasses and examine issues relating to private and protected access specifiers

Example 2: The Cylinder class – No inheritance

```
package circle;
import static java.lang.Math.*;
class Cylinder {
    private double radius;
    private double length;
    public Cylinder(double r,double l){
        setRadius(r);
        setLength(l);
    }
    public Cylinder(){
        radius = 0;
        length = 0;
    }
    public void setRadius(double r){
        radius = (r < 0.0) ? 0.0 : r;
    }
    public void setLength(double l){
        length = (l < 0.0) ? 0.0 : l;
    }
    public double getRadius(){
        return radius;
    }
    public double getLength(){
        return length;
    }
    public double computeCircleArea(){
        return PI*radius*radius;
    }
}
```

```
    public double computeCircleCircumference(){
        return 2*PI*radius;
    }
    public double computeCircleDiameter(){
        return 2*radius;
    }
    public double computeCylinderArea(){
        return computeCircleCircumference() * length;
    }
    public double computeCylinderCircumference(){
        return 2 * (computeCircleCircumference() + length);
    }
    public double computeCylinderVolume(){
        return computeCircleArea() * length;
    }
    public String toString() {
        return String.format("%s:\n%s: %f\n%s: %f\n%s: %f\n%s: %f",
            + "%f\n\n%s:\n%s: %f\n%s: %f\n%s: %f",
            "Circle","Radius", getRadius(),
            "Diameter", computeCircleDiameter(),
            "Area", computeCircleArea(),
            "Circumference", computeCircleCircumference(),
            "Cylinder", "Length", getLength(),
            "Surface Area",computeCylinderArea(),
            "Cylinder Circum.", computeCylinderCircumference(),
            "Volume", computeCylinderVolume());
    }
}
```

Example 2: The Cylinder class – No inheritance

```
package circle;
public class CylinderTest {
    public static void main(String[] args) {
        //instantiate cylinder
        Cylinder cylinder = new Cylinder(5.5, 10.0);
        System.out.println("Cylinder information obtained by getter methods");
        System.out.printf("%s:\n%s: %f\n", "Circle","Radius",cylinder.getRadius());
        System.out.printf("%s: %f\n", "Diameter",cylinder.computeCircleDiameter());
        System.out.printf("%s: %f\n", "Area",cylinder.computeCircleArea());
        System.out.printf("%s: %f\n", "Circumference",cylinder.computeCircleCircumference());

        System.out.printf("\n\n%s\n%s: %f\n", "Cylinder","Length",cylinder.getLength());
        System.out.printf("%s: %f\n", "Surface Area",cylinder.computeCylinderArea());
        System.out.printf("%s: %f\n", "Cylinder Circum.",cylinder.computeCylinderArea());
        System.out.printf("%s: %f\n", "Volume",cylinder.computeCylinderVolume());

        //change radius
        cylinder.setRadius(10);
        System.out.println("\nCylinder information obtained by toString");
        System.out.printf("%s\n", cylinder);
    }
}
```

Example 2: The Cylinder class – No inheritance

Cylinder information obtained by getter methods

Circle:

Radius: 5.500000

Diameter: 11.000000

Area: 95.033178

Circumference: 34.557519

Cylinder

Length: 10.000000

Surface Area: 345.575192

Cylinder Circum.: 345.575192

Volume: 950.331778

Cylinder information obtained by toString

Circle:

Radius: 10.000000

Diameter: 20.000000

Area: 314.159265

Circumference: 62.831853

Cylinder:

Length: 10.000000

Surface Area: 628.318531

Cylinder Circum.: 145.663706

Volume: 3141.592654

Example 2: The Cylinder class – A First Attempt at Inheritance

```
package circle;
public class Cylinder2 extends Circle {
    double length;
    public Cylinder2(double r, double l) {
        super (r); //explicit call to superclass constructor
    }
    public void setCircleRadius(double r){
        radius = (r < 0.0) ? 0.0 : r;
    }
}
```

- Notes

- Keyword **extends** causes Cylinder2 to be created as a subclass of Circle
- The **super** keyword in the constructor for cylinder causes the constructor of the superclass (Circle) to be called
- The code does not compile because we are making reference to the radius field of Circle which has access modifier **private**
- A private access modifier causes the field to be accessible to only members of the same class

Example 3: The Cylinder class – Modifying superclass instance variable to protected

```
public class Circle2 {  
    protected double radius;  
    public Circle2(double r) {  
        setRadius(r);  
    }  
    public Circle2() {  
        radius = 0;  
    }  
}
```

- We create a new class for circle (Circle2)
 - Identical to class Circle with lines 3-10 modified as above
 - The only change is the access modifier for variable radius (changed from private to protected)
 - ... and of course references from class Circle to class Circle2
- We also create a new class for cylinder (Cylinder3)
 - Note that in the toString() method for cylinder, we can now refer to the radius of the circle as if it were declared in Class Cylinder3

Example 3: The Cylinder class – Modifying superclass instance variable to protected

```
package circle;
public class Cylinder3 extends Circle2 {
    double length;
    public Cylinder3(double r, double l) {    super (r); }
    public void setLength(double l)          {    length = (l < 0.0) ? 0.0 : l;    }
    public double getLength()                 {    return length;    }
    public double computeCylinderArea()       {    return computeCircleCircumference() * length;
}
    public double computeCylinderCircumference(){
        return 2 * (computeCircleCircumference() + length);
    }
    public double computeCylinderVolume(){ return computeCircleArea() * length;    }
    public String toString() {
        return String.format("%s: \n%s: %f\n%s: %f\n%s: %f\n%s: %f\n\n%s:\n%s: %f"
            + "\n%s: %f\n%s: %f\n%s: %f",
            "Circle", "Radius", radius,
            "Diameter", computeCircleDiameter(),
            "Area", computeCircleArea(),
            "Circumference", computeCircleCircumference(),
            "Cylinder", "Length", getLength(),
            "Surface Area", computeCylinderArea(),
            "Cylinder Circum.", computeCylinderCircumference(),
            "Volume", computeCylinderVolume());
    }
}
```

Example 3: The Cylinder class – Modifying superclass instance variable to protected

```
package circle;
public class CylinderTest3 {
    public static void main(String[] args) {
        //instantiate cylinder
        Cylinder3 cylinder = new Cylinder3(5.5, 10.0);
        System.out.println("Cylinder information obtained by getter methods");
        System.out.printf("%s:\n%s: %f\n", "Circle","Radius",cylinder.getRadius());
        System.out.printf("%s: %f\n", "Diameter",cylinder.computeCircleDiameter());
        System.out.printf("%s: %f\n", "Area",cylinder.computeCircleArea());
        System.out.printf("%s: %f\n", "Circumference",cylinder.computeCircleCircumference());

        System.out.printf("\n\n%s\n%s: %f\n", "Cylinder","Length",cylinder.getLength());
        System.out.printf("%s: %f\n", "Surface Area",cylinder.computeCylinderArea());
        System.out.printf("%s: %f\n", "Cylinder Circum.",cylinder.computeCylinderArea());
        System.out.printf("%s: %f\n", "Volume",cylinder.computeCylinderVolume());

        //change radius
        cylinder.setRadius(10);
        System.out.println("\nCylinder information obtained by toString");
        System.out.printf("%s\n", cylinder);
    }
}
```

Example 3: The Cylinder class – Modifying superclass instance variable to protected

Cylinder information obtained by getter methods

Circle:

Radius: 5.500000

Diameter: 11.000000

Area: 95.033178

Circumference: 34.557519

Cylinder

Length: 10.000000

Surface Area: 345.575192

Cylinder Circum.: 345.575192

Volume: 950.331778

Cylinder information obtained by toString

Circle:

Radius: 10.000000

Diameter: 20.000000

Area: 314.159265

Circumference: 62.831853

Cylinder:

Length: 10.000000

Surface Area: 628.318531

Cylinder Circum.: 145.663706

Volume: 3141.592654

Example 3: The Cylinder class – Modifying superclass instance variable to protected

- Program notes

- We could have declared radius of class Circle with public access
 - This is not good software engineering as it allows unrestricted access to instance variables, greatly increasing the chance of error
- With protected instance variables, subclasses get access to the variables, but not so classes which are neither subclasses nor classes in the same package
- Ultimately though, the best software engineering practice requires that instance variables be private
 - In such cases, public methods are used to control access to the private fields (as we have been doing with getter and setter methods)

Example 3: The Cylinder class – Modifying superclass instance variable to protected

- One observation with class `Cylinder3`
 - The `toString()` method of this class duplicates code for displaying the radius, area, and circumference of a circle
 - Good software engineering demands that if a method performs some or all of the actions needed by another method, call that method, rather than duplicate its code
 - We next create class `Cylinder4` that
 - Calls class `Circle`'s `toString` method to provide part of its output
 - Demonstrates how to use the `super` keyword, followed by a dot, and then a method in the superclass to invoke an overridden method
 - In this case, the `toString()` method of class `Cylinder` overrides the `toString()` method of class `Circle`

Example 4: Class Cylinder4

```
package circle;
public class Cylinder4 extends Circle {
    double length;
    public Cylinder4(double r, double l) {
        super (r); //explicit call to superclass constructor
        setLength(l);
    }
    public void setLength(double l){
        length = (l < 0.0) ? 0.0 : l;
    }
    public double getLength(){
        return length;
    }
    public double computeCylinderArea(){
        return computeCircleCircumference() * length;
    }
    public double computeCylinderCircumference(){
        return 2 * (computeCircleCircumference() + length);
    }
    public double computeCylinderVolume(){
        return computeCircleArea() * length;
    }
    public String toString() {
        return String.format("%s\n\n%s:\n%s: %f\n%s: %f\n%s: %f\n%s: %f",
            super.toString(),
            "Cylinder", "Length", getLength(),
            "Surface Area", computeCylinderArea(),
            "Cylinder Circum.", computeCylinderCircumference(),
            "Volume", computeCylinderVolume());
    }
}
```



Example 4: Class Cylinder4

```
package circle;
public class CylinderTest4 {
    public static void main(String[] args) {
        Cylinder4 cylinder = new Cylinder4(5.5, 10.0);
        System.out.println("Cylinder information obtained by getter methods");
        System.out.printf("%s:\n%s: %f\n", "Circle", "Radius", cylinder.getRadius());
        System.out.printf("%s: %f\n", "Diameter", cylinder.computeCircleDiameter());
        System.out.printf("%s: %f\n", "Area", cylinder.computeCircleArea());
        System.out.printf("%s: %f\n", "Circumference", cylinder.computeCircleCircumference());

        System.out.printf("\n\n%s\n%s: %f\n", "Cylinder", "Length", cylinder.getLength());
        System.out.printf("%s: %f\n", "Surface Area", cylinder.computeCylinderArea());
        System.out.printf("%s: %f\n", "Cylinder Circum.", cylinder.computeCylinderArea());
        System.out.printf("%s: %f\n", "Volume", cylinder.computeCylinderVolume());

        //change radius
        cylinder.setRadius(10);
        System.out.println("\nCylinder information obtained by toString");
        System.out.printf("%s\n", cylinder);
    }
}
```



Outline

- Classes and objects
- Nested Classes
- Composition
- Package access & Static import
- Inheritance
- **Polymorphism**
- Interfaces




Polymorphism

- Means several forms
- Enables us to “program in the general”, rather than “program in the specific”
- Example
 - A program that simulates the movement of several types of animals: fish, frog, bird, etc.
 - Each of these animal types is a subclass of class Animals
 - Each subclass implements a method for move
 - The same call (i.e., move) is given to each subclass
 - However, the details of the move is different (takes many forms) for each
 - E.g., a fish might swim 3 m, a frog may jump 1m, a bird may fly 10 m etc.



Demonstrating polymorphic behaviour

```
package circle;
public class PolymorphismTest {
    public static void main(String[] args) {
        //assign superclass reference to superclass variable
        Circle circle = new Circle(5.5);
        //assign subclass reference to subclass variable
        Cylinder4 cylinder = new Cylinder4(5.5, 10);
        //invoke toString on superclass object using superclass variable
        System.out.printf("\nCall Circle's toString with"
            + " superclass reference to superclass object");
        System.out.printf("\n%s\n\n", circle.toString());
        //invoke toString on subclass object using subclass variable
        System.out.printf("\nCall Cylinder's toString with"
            + " subclass reference to subclass object");
        System.out.printf("\n%s\n\n", cylinder.toString());
        //invoke toString on subclass object using superclass variable
        Circle circle2 = cylinder;
        System.out.printf("\nCall Cylinder's toString with"
            + " superclass reference to subclass object");
        System.out.printf("\n%s\n", circle2.toString());
    }
}
```





Demonstrating polymorphic behaviour

Call Circle's toString with superclass reference to superclass object
Circle:

Radius: 5.500000
Diameter: 11.000000
Area: 95.033178
Circumference: 34.557519

Call Cylinder's toString with subclass reference to subclass object
Circle:

Radius: 5.500000
Diameter: 11.000000
Area: 95.033178
Circumference: 34.557519

Cylinder:
Length: 10.000000
Surface Area: 345.575192
Cylinder Circum.: 89.115038
Volume: 950.331778

Call Cylinder's toString with superclass reference to subclass object
Circle:

Radius: 5.500000
Diameter: 11.000000
Area: 95.033178
Circumference: 34.557519

Cylinder:
Length: 10.000000
Surface Area: 345.575192
Cylinder Circum.: 89.115038
Volume: 950.331778



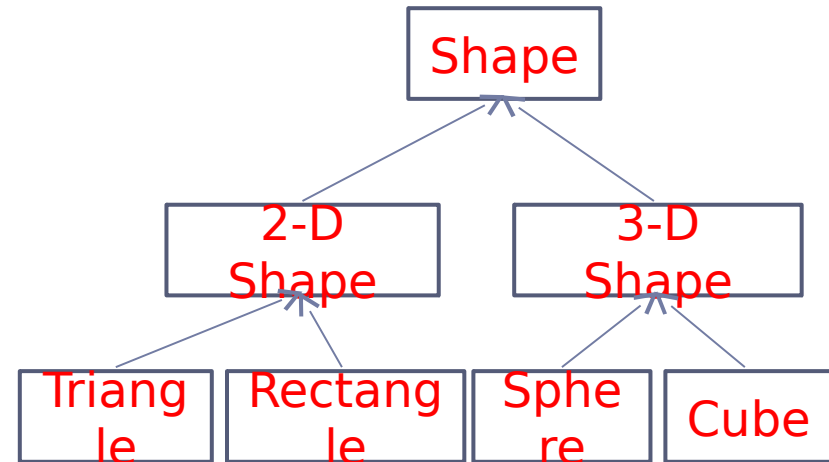
Abstract classes and methods

- Abstract class
 - A class from which objects cannot be instantiated
 - Also known as abstract superclass because they are used only as superclasses in inheritance hierarchies
 - Conversely, classes that can be used to instantiate objects are known as concrete classes

Abstract classes and methods

- Examples

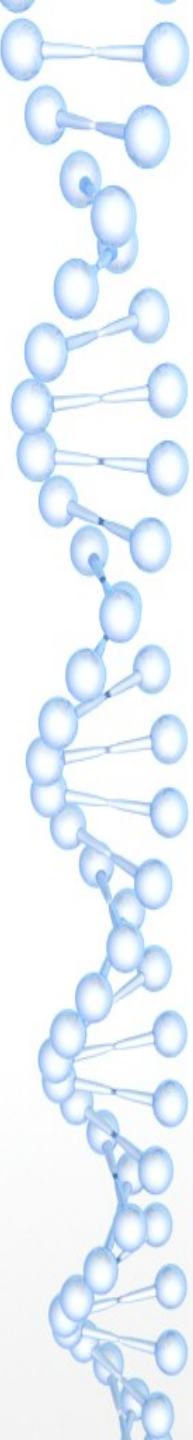
- Shape, 2-D shape and 3-D shape are abstract classes
 - They are too general to create real objects
 - They specify only what is common in subclasses
- Triangle, Rectangle, etc. on the other hand are concrete classes





Creating an abstract class

- Declare the class with the keyword ***abstract***
- An abstract class normally contains one or more abstract methods, also declared with keyword ***abstract***
 - Abstract methods can only be found in abstract classes, even if the class contain some concrete methods
 - Abstract methods do not provide implementations
 - Each concrete class of an abstract superclass must provided concrete implementations of each of the superclass's abstract methods

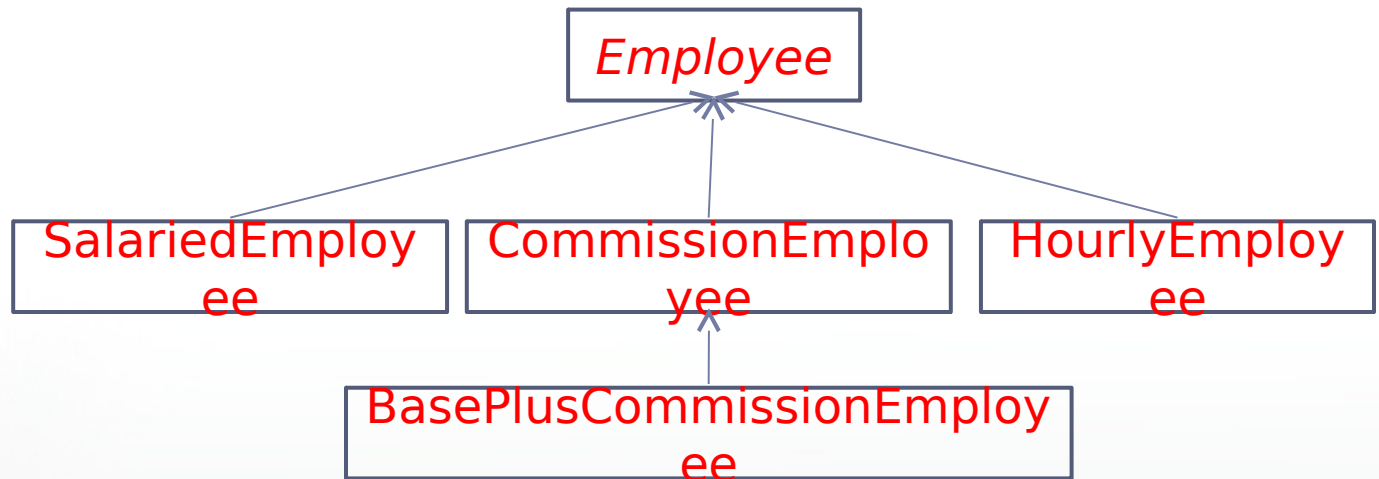


Polymorphism case study: A payroll system

- A company pays its employees on a weekly basis. The employees are of four types:
 - Salaried employees: paid a fixed weekly salary regardless of number of hours worked
 - Hourly employees: paid by the hour and receive overtime pay for all hours worked in excess of 40 hours
 - Commission employees: paid a percentage of their sales
 - Salaried-commission employees: paid a base salary plus a percentage of their sales
- For the current pay period, the company has decided to reward salaried-commission employees by adding 10% to their base salaries. The company wants to implement a Java application that performs its payroll calculations polymorphically.

Polymorphism case study: A payroll system

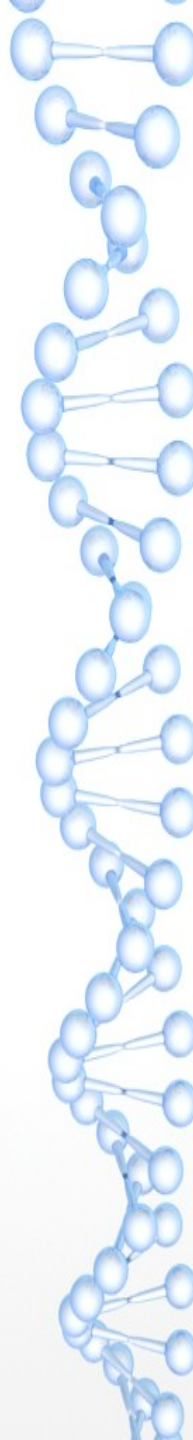
- We use the inheritance hierarchy below
 - Employee is an abstract class
 - The other classes are concrete



Polymorphism case study: A payroll system

- The five classes, and their corresponding earnings and toString methods

	Earnings	toString
Employee	abstract	<i>firstName lastName</i> id num: <i>id</i>
Salaried-Employee	weeklySalary	salaried employee: <i>firstName lastName</i> id num: <i>id</i> weekly salary: <i>weeklySalary</i>
Hourly-Employee	If hours <= 40 <i>wage</i> * hours else 40 * <i>wage</i> + (hours-40) * <i>wage</i> * 1.5	hourly employee: <i>firstName lastName</i> id num: <i>id</i> Hourly wage: <i>wage</i> ; hours worked: <i>hours</i>
CommissionEmployee	<i>commissionRate</i> * grossSales	commission employee: <i>firstName lastName</i> id num: <i>id</i> gross sales: <i>grossSales</i> ; commission rate: <i>commissionRate</i>
BasePlus-CommissionEmployee	(<i>commissionRate</i> *grossSales) + baseSalary	base salaried commission employee: <i>firstName lastName</i> id num: <i>id</i> gross sales: <i>grossSales</i> ; commission rate: <i>commissionRate</i> base salary: <i>baseSalary</i>



Polymorphism case study: A payroll system

- Note
 - Class Employee's earnings method has to be abstract
 - Why?
 - It does not make sense to provide an implementation of the earnings() method in class Employee
 - Earnings cannot be computed for a general employee
 - We must first know the employee type before we can compute his/her earnings

Polymorphism case study: A payroll system

- Class Employee

```
package employee;

public abstract class Employee {
    private String firstName;
    private String lastName;
    private int id;
    public Employee (String first,
                     String last, int id){
        setFirstName(first);
        setLastName(last);
        setId(id);
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
}
```

```
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String toString() {
        return String.format("%s %s\nID: %s",
                              getFirstName(), getLastName(), getId());
    }
    //abstr method: overridden by subclasses
    public abstract double earnings();
}
```

Polymorphism case study: A payroll system

- Subclass

SalariedEmployee

- Extends class Employee
- Overrides abstract method Earnings() to compute weekly salary
- Notes on method toString()
 - It overrides Employees's toString() method
 - It calls superclass's toString() method to display employee names and ID, and then calls method getWeeklySalary() to display employee's salary

```
package employee;
public class SalariedEmployee extends Employee {
    private double weeklySalary;
    public SalariedEmployee(String first,
        String last, int id, double salary) {
        super(first, last, id);
        setWeeklySalary(salary);
    }
    public double getWeeklySalary() {
        return weeklySalary;
    }
    public void setWeeklySalary(double salary) {
        weeklySalary = salary < 0.0 ? 0.0 : salary;
    }
    public double earnings() {
        return getWeeklySalary();
    }
    public String toString() {
        return String.format("Salaried Employee: "
            + "%s\n%s: %, .2f", super.toString(),
            "Weekly Salary", getWeeklySalary());
    }
}
```

Polymorphism case study: A payroll system

- Subclass HourlyEmployee
 - Performs similar actions to SalariedEmployee, but this time for HourlyEmployee

```
package employee;
public class HourlyEmployee extends Employee{
    private double wage;
    private double hours;
    public HourlyEmployee(String first,
        String last, int id, double hourlyWage,
        double hoursWorked) {
        super(first, last, id);
        setWage(hourlyWage);
        setHours(hoursWorked);
    }
    public double getHours() {
        return hours;
    }
    public void setHours(double hours) {
        this.hours = hours;
    }
}
```

```
public double getWage() {
    return wage;
}
public void setWage(double wage){
    this.wage = wage;
}
public double earnings() {
    if (getHours() <= 40)
        return getWage()*getHours();
    else
        return 40*getWage() +
            (getHours()-40) *
            getWage()*1.5;
}
public String toString() {
    return String.format("Hourly "
        + "Employee: %s\n%s: %, .2f",
        super.toString(), "Hourly Wage",
        getWage());
}
}
```

Polymorphism case study: A payroll system

- Subclass
CommissionEmployee

```
package employee;
public class CommissionEmployee extends
Employee {
    private double grossSales;
    private double commissionRate;
    public CommissionEmployee(String first,
        String last, int id, double sales,
        double rate) {
        super(first, last, id);
        setGrossSales(sales);
        setCommissionRate(rate);
    }
    public double getCommissionRate() {
        return commissionRate;
    }
}
```

```
    public void setCommissionRate(double
commissionRate) {
        this.commissionRate=commissionRate;
    }
    public double getGrossSales() {
        return grossSales;
    }
    public void setGrossSales(double grossSales{
        this.grossSales = grossSales;
    }
    public double earnings() {
        return getCommissionRate()*getGrossSales();
    }
    public String toString() {
        return String.format("Commission "
            +"Employee: %s\n%s: %,.2f\n%s: %,.2f",
            super.toString(), "Gross Sales",
            getGrossSales(), "Commission Rate",
            getCommissionRate());
    }
}
```

Polymorphism case study: A payroll system

- Indirect Subclass BasePlusCommissionEmployee

```
package employee;

public class BasePlusCommissionEmployee extends CommissionEmployee {
    private double baseSalary;

    public BasePlusCommissionEmployee(String first, String last,
        int id, double sales, double rate, double salary) {
        super(first, last, id, sales, rate);
        setBaseSalary(salary);
    }

    public double getBaseSalary() {
        return baseSalary;
    }

    public void setBaseSalary(double baseSalary) {
        this.baseSalary = baseSalary;
    }

    public double earnings() {
        return getBaseSalary() + super.earnings();
    }

    public String toString() {
        return String.format("Base "
            + "Salaried: %s\n%s: %, .2f",
            super.toString(), "Base Salary",
            getBaseSalary());
    }
}
```



Polymorphism case study: A payroll system

• Test Program

```
package employee;

public class PayrollSystemTest {
    public static void main(String[] args) {
        //subclass objects
        SalariedEmployee salariedEmployee = new SalariedEmployee("John","Smith",121,200000);
        HourlyEmployee hourlyEmployee = new HourlyEmployee("Karen","Price",225,4000,55);
        CommissionEmployee commissionEmployee = new CommissionEmployee("Sue","Jones",85,
500000,.06);
        BasePlusCommissionEmployee basePlusCommissionEmployee = new
BasePlusCommissionEmployee("Bob","Lewis",200,50000,.04,150000);
        System.out.println("Employees processed individually:\n");
        System.out.printf("%s\n%s: %, .2f\n\n",salariedEmployee,"Earned",
salariedEmployee.earnings());
        System.out.printf("%s\n%s: %, .2f\n\n",hourlyEmployee,"Earned",
hourlyEmployee.earnings());
        System.out.printf("%s\n%s: %, .2f\n\n",commissionEmployee,"Earned",
commissionEmployee.earnings());
        System.out.printf("%s\n%s: %, .2f\n\n",basePlusCommissionEmployee,"Earned",
basePlusCommissionEmployee.earnings());
    }
}
```


Polymorphism case study: A payroll system

- Test Program

```
//4-element employee array
Employee employees[] = new Employee[4];
employees[0] = salariedEmployee;
employees[1] = hourlyEmployee;
employees[2] = commissionEmployee;
employees[3] = basePlusCommissionEmployee;
System.out.println("Employees processed polymorphically:\n");
for (Employee currentEmployee : employees) {
    System.out.println(currentEmployee); //invoke toString
    //determine if object is a BasePlusCommissionEmployee
    if (currentEmployee instanceof BasePlusCommissionEmployee){
        //downcast Employee ref to BasePlusCommissionEmployee ref
        BasePlusCommissionEmployee employee = (BasePlusCommissionEmployee)
currentEmployee;
        double oldBaseSalary = employee.getBaseSalary();
        employee.setBaseSalary(1.10*oldBaseSalary);
        System.out.printf("New base salary with 10%% increase is %, .2f\n",
employee.getBaseSalary());
    }
    System.out.printf("Earned %, .2f\n\n", currentEmployee.earnings());
}
}
```

Polymorphism case study: A payroll system

Employees processed individually:

Salaried Employee: John Smith
ID: 121
Weekly Salary: 200,000.00
Earned: 200,000.00

Hourly Employee: Karen Price
ID: 225
Hourly Wage: 4,000.00
Earned: 250,000.00

Commission Employee: Sue Jones
ID: 85
Gross Sales: 500,000.00
Commission Rate: 0.06
Earned: 30,000.00

Base Salaried: Commission Employee:
Bob Lewis
ID: 200
Gross Sales: 50,000.00
Commission Rate: 0.04
Base Salary: 150,000.00
Earned: 152,000.00

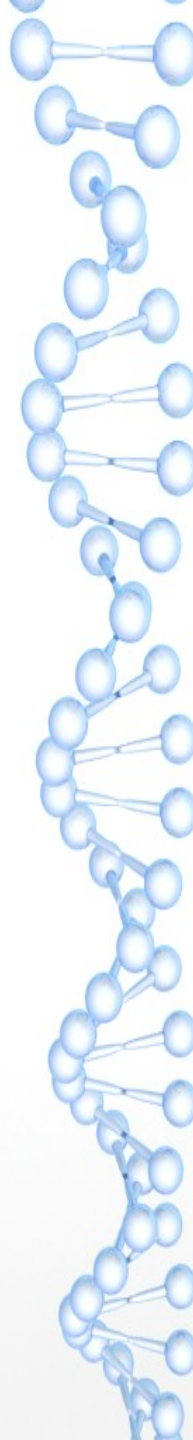
Employees processed polymorphically:

Salaried Employee: John Smith
ID: 121
Weekly Salary: 200,000.00
Earned 200,000.00

Hourly Employee: Karen Price
ID: 225
Hourly Wage: 4,000.00
Earned 250,000.00

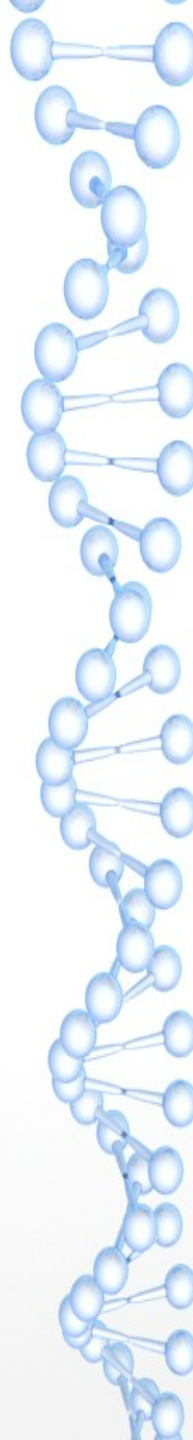
Commission Employee: Sue Jones
ID: 85
Gross Sales: 500,000.00
Commission Rate: 0.06
Earned 30,000.00

Base Salaried: Commission Employee: Bob Lewis
ID: 200
Gross Sales: 50,000.00
Commission Rate: 0.04
Base Salary: 150,000.00
New base salary with 10% increase is
165,000.00
Earned 167,000.00



Polymorphism case study: A payroll system

- Program notes
 - Polymorphic behaviour is demonstrated starting with the employees array
 - We note that the values of the array are subclass objects derived directly or indirectly from class `Employees`
 - This is possible because of the is-a relationship
 - The loop is an iteration through array `employees` and invokes methods `toString` and `earnings` with `Employee` variable `currentEmployee`, which is assigned the reference to a different `Employee` during each iteration
 - The output illustrates that the appropriate method for each class is indeed invoked
 - These methods are invoked at run time (and not compile time) in a process known as dynamic binding or late binding



Polymorphism case study: A payroll system

- The instanceof operator
 - Compares an object to a specified type
 - Can be used to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface
 - In this program we use this to do special processing on BasePlusCommissionEmployee objects
 - When we encounter objects of this class, we increase their base salary by 10%
 - Object employee from class BasePlusCommissionEmployee is created from object currentEmployee (of class Employee) by downcasting currentEmployee
 - Only allowed when there is an is-a relationship
 - This cast is required if we are to invoke subclass BasePlusCommissionEmployee methods getBaseSalary and setBaseSalary on current Employee object



`final` Methods and Classes

- We have seen that a variable declared as `final` cannot change, i.e., they represent constant values
- A final method cannot be overridden in a subclass
 - Private and static methods are implicitly final, as they cannot be overridden in subclasses
- The compiler resolves calls to final methods at compile (not run) time
 - The process is known as ***static binding***



Outline

- Classes and objects
- Nested Classes
- Composition
- Package access & Static import
- Inheritance
- Polymorphism
- **Interfaces**



Interfaces

- Understanding Interfaces
 - We use interfaces all the time in the real world
 - Example: interface between a driver and a car with manual transmission includes
 - Steering wheel, gear shift, clutch pedal, brakes pedal, gas pedal
 - Same interface found in most manual transmission cars
 - Hence, knowing how to drive one manual transmission car, the driver can drive just about any manual transmission car



Interfaces

- Software objects also communicate via interfaces
 - Example: the payroll application we saw
 - Consisted of computations that determined what payments to make (in our case, payments relating to salaries and wages)
 - But the company could also engage in making payments unrelated to salaries and wages
 - For example, payments for the items listed in an invoice
 - Interfaces give us the capability to handle similar computations for unrelated objects
 - E.g., employee salaries and wages (employee objects) and invoice payments (invoice objects)



Interfaces

- About interfaces

- All interface members must be public
- Interfaces may not specify any implementation details such as concrete method declarations and instance variables
 - Hence
 - Interface methods are implicitly public abstract methods
 - All interface fields are implicitly public, static, and final
- To use an interface, a concrete class
 - Must specify that it implements the interface
 - Must declare each method in the interface with the signature specified in the interface declaration
- A class that does not implement all the methods of the interface is an abstract class and must be declared abstract
- Note
 - If the interface is in a different package, it needs to be imported into the implementing class, before the class definition

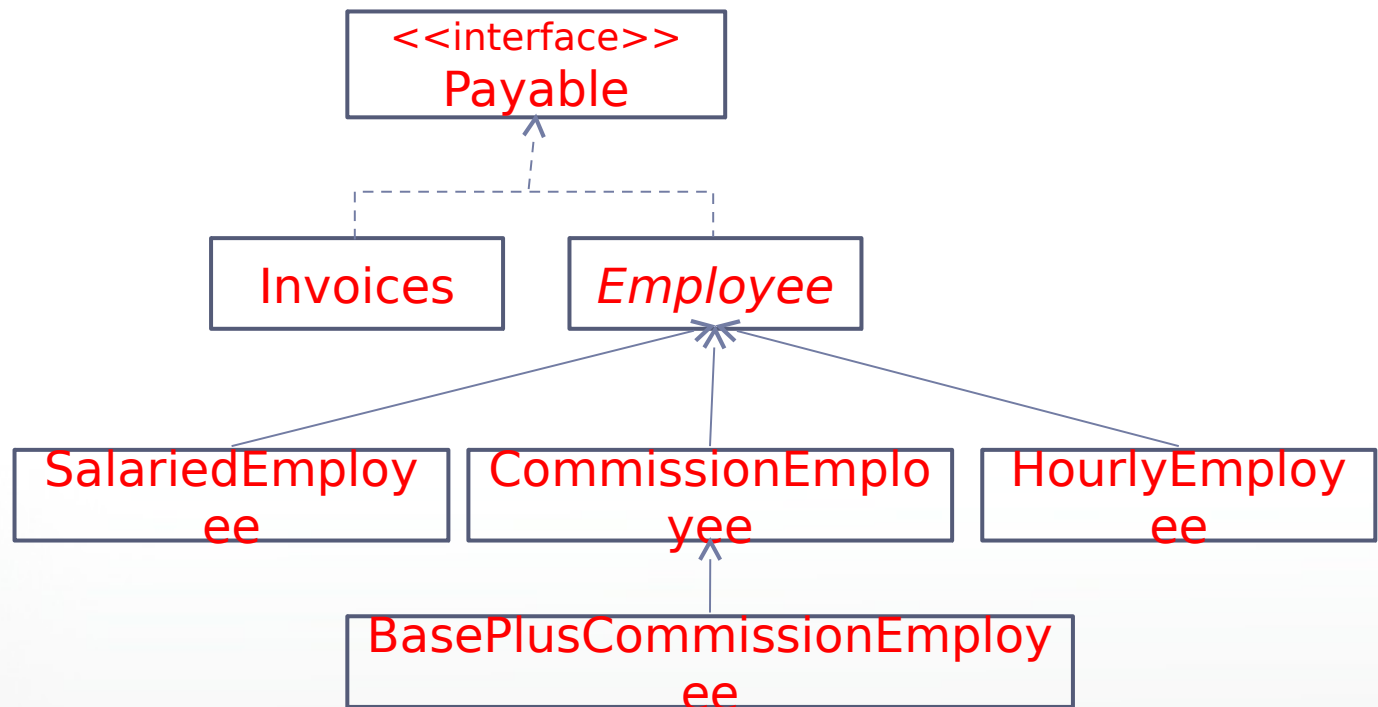


Interfaces

- Hence
 - A class implementing an interface is like signing the following contract with the compiler
 - I will declare all methods specified by the interface **or**
 - I will declare my class abstract

Interfaces Example

- A Payable Hierarchy
 - An application that can determine payments for both employees and invoices





Interfaces Example

- Interface Payable
 - Contains method `getPaymentAmount`
 - Method returns a double amount that must be paid for an object of any class that implements the interface
 - This is a general purpose version of method earnings of the Employee hierarchy
- Class Invoice
 - Implements interface Payable
- Class Employee
 - A modification of the old Class Employee to implement interface Payable



Interfaces Example

```
package employee;  
public interface Payable {  
    double getPaymentAmount();  
}
```

Interfaces Example

- Class Invoice

```
package employee;

public class Invoice implements Payable {
    private String partNumber;
    private String partDescription;
    private int quantity;
    private double pricePerItem;

    public Invoice(String part, String description, int count, double price) {
        this.partNumber = part;
        this.partDescription = description;
        setQuantity(count);
        setPricePerItem(price);
    }

    public String getPartDescription() {
        return partDescription;
    }

    public void setPartDescription(String partDescription) {
        this.partDescription = partDescription;
    }

    public String getPartNumber() {
        return partNumber;
    }
}
```

Interfaces Example

• Class
Invoice
contd.

```
public void setPartNumber(String partNumber) {
    this.partNumber = partNumber;
}
public double getPricePerItem() {
    return pricePerItem;
}
public void setPricePerItem(double price) {
    this.pricePerItem = (price < 0.0) ? 0.0 : price;
}
public int getQuantity() {
    return quantity;
}
public void setQuantity(int count) {
    this.quantity = (count < 0) ? 0 : count;
}
public String toString() {
    return String.format("%s: \n%s: %s (%s) \n%s: %d \n%s: %, .2f",
        "Invoice", "Part Number", getPartNumber(), getPartDescription(),
        "Quantity", getQuantity(), "Price per Item", getPricePerItem());
}
public double getPaymentAmount() {
    return getQuantity() * getPricePerItem();
}
}
```

Interfaces Example

```
package employee;

public abstract class Employee implements Payable {
    private String firstName; private String lastName; private int id;
    public Employee (String first, String last, int id){
        setFirstName(first);  setLastName(last);    setId(id);
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String toString() {
        return String.format("%s %s\nID: %s",getFirstName(),
                               getLastName(),getId());
    }
}
```

· Class
Employee

Interfaces Example

- Class SalariedEmployee
- Student should modify classes HourlyEmployee, CommissionEmployee, and BasePlusCommissionEmployee similarly

```
package employee;

public class SalariedEmployee extends Employee {
    private double weeklySalary;

    public SalariedEmployee(String first,
        String last, int id, int salary) {
        super(first, last, id);
        setWeeklySalary(salary);
    }

    public double getWeeklySalary() {
        return weeklySalary;
    }

    public void setWeeklySalary(int salary) {
        weeklySalary = salary < 0.0 ? 0.0 : salary;
    }

    public String toString() {
        return String.format("Salaried Employee: "
            + "%s\n%s: %, .2f", super.toString(),
            "Weekly Salary", getWeeklySalary());
    }

    public double getPaymentAmount() {
        return getWeeklySalary();
    }
}
```



Interfaces Example

• PayableInterfaceTest

```
package employee;
public class PayableInterfaceTest {
    public static void main(String[] args) {
        Payable payableObjects[] = new Payable[4];
        payableObjects[0] = new Invoice("01234", "Seat", 2, 10500);
        payableObjects[1] = new Invoice("56789", "Door", 3, 45000);
        payableObjects[2] = new SalariedEmployee("John", "Smith", 121, 200000);
        payableObjects[3] = new SalariedEmployee("Lisa", "Barnes", 231, 250000);
        System.out.println("Invoices and Employees processed polymorphically:\n");
        for (Payable currentPayable : payableObjects) {
            System.out.printf("%s \n%s: %, .2f\n\n", currentPayable.toString(),
                "Payment Due", currentPayable.getPaymentAmount());
        }
    }
}
```

Interfaces Example

- PayableInterfaceTest

Invoices and Employees processed polymorphically:

Invoice:

Part Number: 01234 (Seat)

Quantity: 2

Price per Item: 10,500.00

Payment Due: 21,000.00

Invoice:

Part Number: 56789 (Door)

Quantity: 3

Price per Item: 45,000.00

Payment Due: 135,000.00

Salaried Employee: John Smith

ID: 121

Weekly Salary: 200,000.00

Payment Due: 200,000.00

Salaried Employee: Lisa Barnes

ID: 231

Weekly Salary: 250,000.00

Payment Due: 250,000.00



Declaring Constants with Interfaces

- Constants so declared are
 - public, static, final
- One main use of interfaces to declare constants
 - Declare a set of constants that can be used in many class declarations, as illustrated below

```
public interface Constants {  
    int ONE = 1;  
    int TWO = 2;  
    int THREE = 3;  
}
```



Java API Interfaces

- The Java API supports several useful interfaces that can simply be implemented and used
- Examples
 - Comparable
 - Serializable
 - Runnable
 - GUI event-listener interfaces
 - SwingConstants

