Data Structures for Java

William H. Ford
William R. Topp

Chapter 2
Class Relationships

Bret Ford

Wrapper Classes

- Convert a value of primitive type to an object.
- Supply methods to access and display the value.
- Wrapper classes include Integer, Double, and Boolean.

Comparing Integer Objects

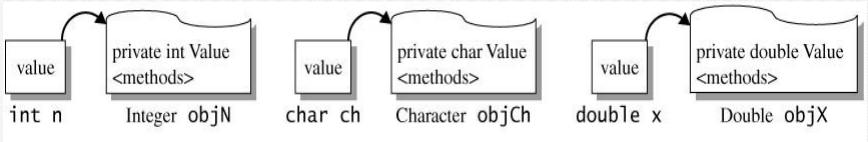
- Compare primitive values using ==, <, <=, >, >=, etc.
- Use equals() and compareTo() for comparing objects.

```
Example: Compare Integer objects objA and objB.
Integer objA = new Integer(35), objB = new Integer(50);

int t = objA.compareTo(objB); // t < 0 since 35 < 50
boolean b = objB.equals(new Integer("35")); // b is true
//assign to objMax the larger of the two Integer objects.

Integer objMin = (objA.compareTo(objB) > 0) ? objA : objB;
```

Wrapper Classes Integer, Character and Double



Wrapper classes integer, character, and double whose objects hold a value of primitive type.

Static Wrapper Class Members

- Integer.MIN_VALUE and Integer.MAX_VALUE are the minimum and maximum integer values
- Double has similar constants.
- Methods such as the Integer parseInt() method convert a numeric string to the corresponding primitive value.
- toString() provides a string representation for a primitive type.

Character Handling

- Character is a wrapper class for the primitive type char.
- Classifying a character

```
public static boolean isLetter(char ch);
public static boolean isDigit(char ch);
public static boolean isWhitespace(char ch);
```

- Testing and Modifying Case of a Character

```
public static boolean isUpperCase(char ch);
public static boolean isLowerCase(char ch);
public static char toUpperCase(char ch);
public static char toLowerCase(char ch);
```

Autoboxing and Auto-unboxing

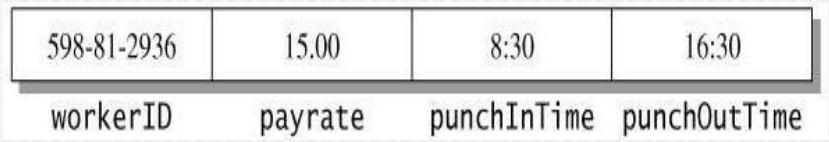
- Autoboxing automatically converts a primitive type to its associated wrapper type.
  - Example: `Integer[] arr = {5, 3, 2, 9, 35};`
- Auto-unboxing automatically converts from a wrapper type to the equivalent primitive type.
  - Example: `int n = arr[1];`

Object Composition

- Class (client class) contains one or more objects of another class (supplier class).
- Termed the “has-a” relationship.

TimeCard Class

- Maintains data for hourly workers.
  - `private String workerID;`
    `private double payrate;`
    `private Time24 punchInTime, punchOutTime;`



TimeCard Constructor

```
public TimeCard(String workerID,
                double payrate,
                int punchInHour,
                int punchInMinute)
{
   // initialize workerID and payrate
   this.workerID = workerID;
   this.payrate = payrate;

   // create Time24 object by calling
   // constructor Time24(hour,minute)
   punchInTime = new Time24(punchInHour,punchInMinute);

   // create Time24 object by calling
   // default constructor Time24()
   punchOutTime = new Time24();
}
```

payworker()

```
public String payWorker(int punchOutHour,
                        int punchOutMinute)
{
   // local variables for time
   // worked and hours worked
   Time24 timeWorked;
   // timeWorked converted to hours
   double hoursWorked;

   // numeric format object for
   // hours worked and pay
   DecimalFormat fmt =  new DecimalFormat("0.00");

   // update punchOutTime by calling setTime()
   punchOutTime.setTime(punchOutHour,punchOutMinute);
```

payworker() (continued)

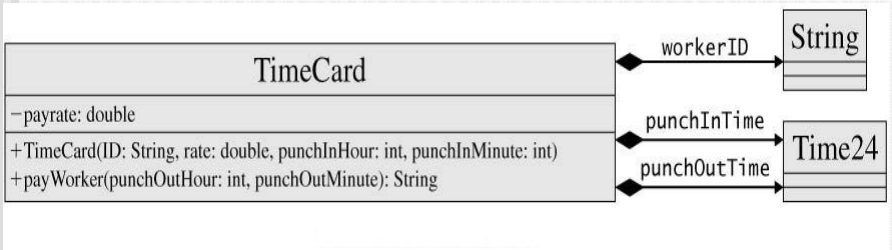
```
      // evaluate time worked with Time24
      // interval() method
      timeWorked =
            punchInTime.interval(punchOutTime);

      // hoursWorked is timeWorked as
      // fractional part of an hour
      hoursWorked = timeWorked.getHour() +
                    timeWorked.getMinute()/60.0;
```

payworker() (concluded)

```
        // return formatted string
        return "Worker:      " +
               workerID + "\n" +
               "Start time:  " + punchInTime +
               "End time: " +
               punchOutTime + "\n" +
               "Total time:  " +
               fmt.format(hoursWorked) +
               " hours" + "\n" +
               "At $" + fmt.format(payrate) +
               " per hour, pay is $" +
               fmt.format(payrate*hoursWorked);
    }
```

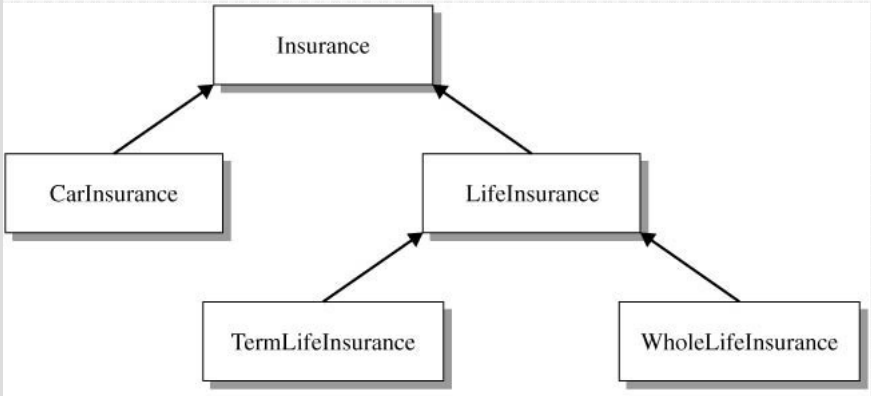
UML for the TimeCard Class



Inheritance in Java

- An “is a” relationship that involves sharing of attributes and methods among classes.
- A superclass defines a common set of attributes and operations.
- A subclass extends the resources in a superclass by adding its own data and methods.

Inheritance Hierarchy Tree

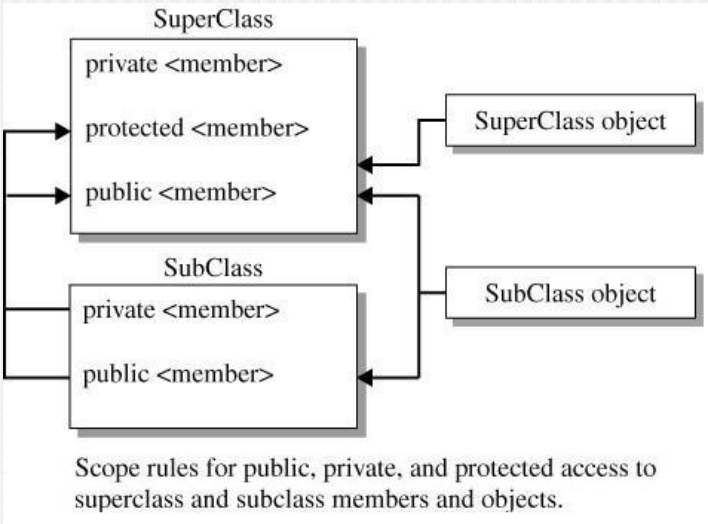


Hierarchy tree describing inheritance relations among insurance policies.

Visibility for Members in an Inheritance Hierarchy

- Private members accessible only within a particular class.
- Protected members are accessible by defining class and all subclasses.
- Public members are accessible by the defining class, all subclasses, and any instance of the class.

Scope Rules in Inheritance Hierarchies



Scope rules for public, private, and protected access to superclass and subclass members and objects.

Employee Inheritance Hierarchy

- Superclass Employee specifies name and Social Security Number and associated methods.
- The subclasses SalaryEmployee and HourlyEmployee inherit (extend) Employee and add data and methods for handling salaried workers and hourly workers.

Employee Class

```
class Employee
{
   // instance variables are accessible by
   // subclass methods
   protected String empName;
   protected String empSSN;

   // create an object with initial values
   // empName and empSSN
   public Employee(String empName, String empSSN)
   {
      this.empName = empName;
      this.empSSN = empSSN;
   }
   // update the employee name
   public void setName(String empName)
   { this.empName = empName; }
```

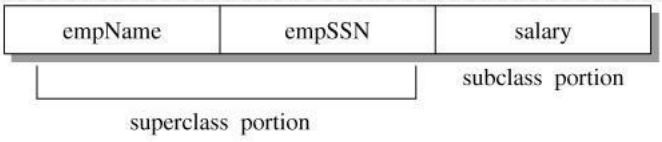
Employee Class (concluded)

```
    // returns a formatted string to display
    // employee information
    public String toString()
    { return "Name:     " + empName + '\n' +
             "SS#:      " + empSSN; }

    // method is declared in this
    // class for polymorphism
    public String payrollCheck()
    { return ""; }
}
```

SalaryEmployee Subclass

- SalaryEmployee class extends Employee and adds a salary instance variable along with methods that access the salary.



A resulting `SalaryEmployee` object consists of three data fields: the name,Social Security number, and salary. You can view the object as having two parts, the superclass portion and the subclass portion. The superclass portion consists of data in the superclass and the subclass portion consists of data in the subclass.

SalaryEmployee API

class SALARYEMPLOYEE extends Employee	
	Constructor
	SalaryEmployee(String empName, String empSSN, double salary) Creates an object with arguments empName and empSSN initializing the superclass portion of the object.
	Methods
double	**getSalary**() Returns the salary paid the employee during each pay period. .
String	**payrollCheck**() Returns a string that describes a pay check. The format includes the employee name, social security number, and salary.
void	**setSalary**(double salary) Assigns the specified argument as the new salary.
String	**toString**() Returns a string that describes the object. The format includes the name, social security number, status ("salaried") and the salary.

Keyword super

- Use super(args) in a subclass constructor to call the superclass constructor. Must be the first statement in the constructor.
- For methods with the same name in the subclass and the superclass, call the superclass method using the form
  super.method(args)

SalaryEmployee Constructor

```
public SalaryEmployee(String empName,
                      String empSSN,
                      double salary)
{
   // call the Employee
   // superclass constructor
   super(empName, empSSN);
   this.salary = salary;
}
```

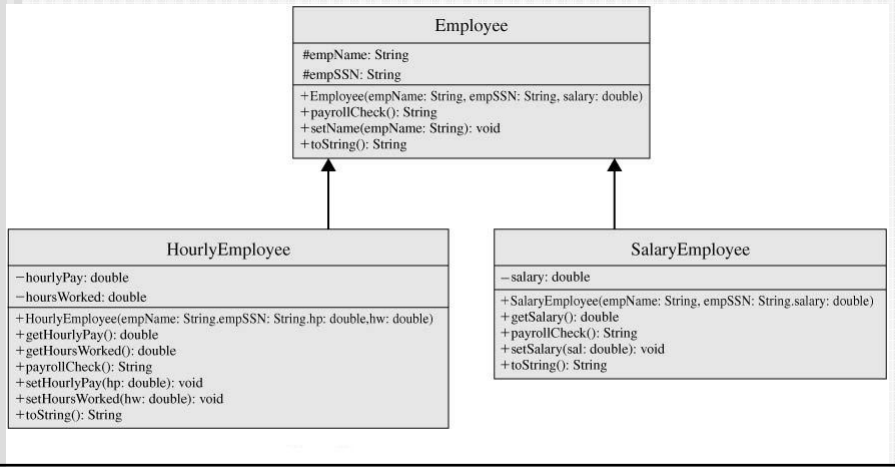
SalaryEmployee toString()

```
public String toString()
{
   DecimalFormat fmt =
         new DecimalFormat("#.00");
   return super.toString() + '\n' +
          "Status:   Salary" + '\n' +
          "Salary:   $" + fmt.format(salary);
}
```

HourlyEmployee Subclass

- Declares private instance variables `hourlyPay` and `hoursWorked` of type double

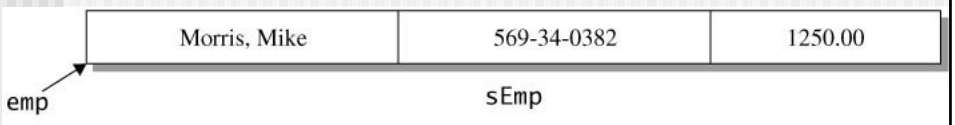
UML for the Employee Hierarchy



Assignment in an Inheritance Hierarchy

- Can assign any subclass reference to a superclass reference.

```
Employee emp;
SalaryEmployee sEmp =
   new SalaryEmployee("Morris, Mike", "569-34-0382",
                      1250.00);
emp = sEmp;
```



Employee variable emp references the SalaryEmployee object sEmp.

Assignment in an Inheritance Hierarchy (concluded)

- Superclass variable that references a subclass object can be used to call any public method in the superclass.
- May not be used to call a method defined only in the subclass.

```
// invalid! setSalary() is not defined in Employee
emp.setSalary(1500.00);

// valid!  uses reference emp to call an Employee method
emp.setName("Morris, Michael");
```

Static Binding

- Static binding associates a method with the class type of a reference variable.

Example:

```
emp.setName("Harrison, Pamela"),
sEmp.setSalary(1500.0)
```

Overriding Methods

- When the superclass and the subclass have methods with the same signature, we say that the subclass method "overrides" the superclass method.

```
// emp = sEmp sets emp to point at sEmp ("Michael Morris").
// the runtime system executes payrollCheck SalaryEmployee
System.out.println(emp.payrollCheck());
```

```
Output:
Pay Morris, Mike (569-34-0382)  $1250.00
```

Polymorphism

- When the superclass and one or more subclasses define methods with the same signature, the runtime system executes the subclass method under the following conditions
  - A subclass object is assigned to a superclass reference variable.
  - The method call uses the superclass reference variable.

Polymorphism (concluded)

- Rather than using static binding which would associate the method with the superclass reference variable, the compiler directs the runtime system to determine the subclass type referenced by the variable and then calls the corresponding subclass method. This is dynamic binding since the association between reference variable and method is established at runtime.

Polymorphism Example

```
// declare a subclass object
HourlyEmployee hEmp =
   new HourlyEmployee("Holmes, Julie",
                      "837-68-2198",
                      12.00, 30);
// assign subclass object hEmp to superclass
// reference variable
emp = hEmp;
// create a pay check using polymorphism
System.out.println(emp.payrollCheck());
```

```
Output:
   Pay Holmes, Julie (837-68-2198) $360.00
```

Upcasting

- Upcasting occurs when a subclass object reference is assigned to a superclass reference.
  - Superclass reference variable may call any public method in the superclass and any public method in the subclass for which polymorphism applies.

Downcasting

- A superclass reference variable may not be used to call a method defined exclusively in a subclass. The programmer must use a cast to change the reference type of the variable to that of the subclass.

Downcasting Example

```
SalaryEmployee sEmp =
   new SalaryEmployee(“Bonner, Al”, “667-21-7128”
                      2500.0);
Employee emp;

Emp = sEmp;

((SalaryEmployee)emp).setSalary(3000.0);
```

The instanceof Operator

- Sometimes the choice of an downcast cannot be made until runtime. The instanceof operator allows the determination of subclass type.
- The method payIncrease() provides a good example of using the instanceof operator. Its first argument can be either a SalaryEmployee or an HourlyEmployee.

```
// give employee a percentage pay increase of pct
public void payIncrease(Employee emp, double pct)
```

payIncrease()

```
public static void payIncrease(Employee emp,
                               double pct)
{
   // use instanceof to determine the
   // object type for emp if SalaryEmployee,
   // access and update salary
   if (emp instanceof SalaryEmployee)
      ((SalaryEmployee)emp).setSalary(
         (1.0 + pct) *
         ((SalaryEmployee)emp).getSalary());
   else
         ((HourlyEmployee)emp).setHourlyPay(
            (1.0 + pct) *
         ((HourlyEmployee)emp).getHourlyPay());
}
```

Abstract Classes

- Define an abstract method in a class by preceding the signature with the keyword abstract and replacing the method body by ";". Place the keyword abstract in the class header.

```
abstract class ClassName
{
   // abstract class may contain data and concrete methods
      . . .
   // abstract class must contain at least one abstract method
   abstract public returnType methodName(<parameters>);
}
```

Abstract Classes (concluded)

- Each subclass of an abstract class must override all of the abstract methods in the superclass.
- A program cannot create an instance of an abstract class.
- Provides only resources for a subclass and method declarations that can be used with polymorphism.

The Scanner Class

- A Scanner object partitions text from an input stream into tokens by means of its "next" methods.
- Declare a Scanner object as follows:

```
Scanner keyIn = new Scanner(System.in);
```

Scanner Methods

- String line = sc.nextLine(); // whole line
- int i = sc.nextInt(); // i = 17
- String str = sc.next(); // str = "deposit"
- double x = sc.nextDouble(); // x = 450.75
- boolean b = sc.nextBoolean(); // b = false
- char ch = sc.next().charAt(0); // ch = 'A'

Testing for Scanner Tokens

```
// loop reads tokens in the line
while (sc.hasNext())
{
    token = sc.next();
    System.out.println("In loop next token = " + token);
}
```

```
Output:
      In loop next token = 17
      In loop next token = deposit
      In loop next token = 450.75
      In loop next token = false
      In loop next token = A
```

Scanner Class API

class SCANNER	java.util
	Constructors
	Scanner((InputStream source) Creates a Scanner object that produces values read from the specified input stream (Typically standard input System.in that denotes the keyboard)
	Scanner(Readable source) Creates a Scanner object that produces values read from the specified input stream (Typically a FileReader that denotes a file)

Scanner Class API continued

	Methods
void	**close()** Close the scanner.
boolean	**hasNext**() Returns true if the scanner has another token in the input stream
boolean	**hasNextBoolean**() Returns true if the next token in the input stream can be interpreted as a boolean value
boolean	**hasNextDouble**() Returns true if the next token in the input stream can be interpreted as a double value
boolean	**hasNextInt**() Returns true if the next token in the input stream can be interpreted as an int value

Scanner Class API (concluded)

String	**next**() Finds and returns the next complete token in the input stream as a String.
boolean	**nextBoolean**() Scans the next token in the input stream into a boolean value and returns that value.
double	**nextDouble**() Scans the next token in the input stream into a double value and returns that value.
int	**nextInt** () Scans the next token in the input stream into an int value and returns that value.