Object-Oriented Essentials

By Warren Mansur, edited by Eric Braude

Programs are mostly unrealiable

“I’ve assigned [binary search] in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert [its] description into a program in the language of their choice; a high-level pseudocode was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take thirty minutes to examine their code, which the programmers did with test cases. In several classes and with over a hundred programmers, the results varied little: *ninety percent of the programmers found bugs in their programs* (and I wasn’t always convinced of the correctness of the code in which no bugs were found). I was amazed: given ample time, only about ten percent of professional programmers were able to get this small program right. But they aren’t the only ones to find this task difficult: in the history in Section 6.2.1 of his Sorting and Searching, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.” —Jon Bentley, Programming Pearls (1st edition), pp.35–36

Learning Objectives

- Understand motivations for object-oriented languages
- Interpret relationships between classes
- Use inheritance hierarchy
- Use interface hierarchy
- Apply polymorphism
- Apply downcasting
- Distinguish concrete classes from abstract classes
- Contrast abstract classes vs interfaces
- Use the Comparable interface

A Benefit of Strongly Typed Languages

Better able to practice safety:

Enforce what you intend

Agenda

1. **Objects and classes**
2. Inheritance
3. Constructors in Java
4. Polymorphism
5. Downcasting
6. Abstract classes
7. Interfaces

Motivations For Object-Orientation

Encapsulation
- Every data item is owned by an object
- Objects are decoupled and communicate with external interfaces
- Internal data manipulation is hidden

Modularity
- The application is divided into loosely coupled, logical modules

Abstraction
- Intent is separated from implementation
- Problems are solved the way humans think

Motivations For Object-Orientation

Inheritance
- Objects extend other objects
- Unnecessary rewriting is avoided

Polymorphism
- One name, many forms
- Behavior has one name but behaves differently according to the relevant object

Code Reuse
- Rewriting of very similar code is avoided

Class

- ➲the blueprint from which individual objects are created
- ➲defines the attributes
  - – the object’s data
- ➲defines methods
  - – which represent an object’s behavior

Object

- ➲an *instance* or *occurrence* of a class
- ➲has its own values for the attributes
- ➲behavior same as all other objects for same class

Agenda

1. Objects and classes
2. **Inheritance**
3. Constructors in Java
4. Polymorphism
5. Downcasting
6. Abstract classes
7. Interfaces

Inheritance Example: from *Car*

```
class Car
{
   private String model;
   private int  milesDriven;

   public Car(String model, int miles)
   {
        this.model = model;
        this.milesDriven = miles;
   }

   public int findPrice()
   {
      if (milesDriven < 50000)
         return 20000;
      else
         return 10000;
   }
}
```

Car Inheritance Ex.

```
class ElectricCar extends Car
{
   private int  energyEfficiency;

   public ElectricCar(String model, int miles, int eff)
   {
      super(model, miles);
      energyEfficiency = eff;
   }

   public int findPrice()
   // POSTCONDITION:
   // EITHER energyEfficiency < 3 AND super.findPrice() + $2,000 was returned
   // OR energyEfficiency >= 3 AND super.findPrice() + $2,500 was returned
   {
      int temp = super.findPrice();
      if (energyEfficiency < 3)
         return temp + 20000;
      else
         return temp + 25000;
   }
}
```

Car Inheritance Execution

```
public class Main
{
   public static void main(String[] args)
   /*
    * POSTCONDITION 1: price of Camry with 48,000 miles is on the console
    * POSTCONDITION 2: price of Camry electric with 4,000 miles is on the console
    */
   {
       Car c = new Car("Camry", 48000);
       System.out.println(c.findPrice());
       ElectricCar ec = new ElectricCar("Camryel", 4000, 5);
       System.out.println(ec.findPrice());
   }
}
```

Output

```
20000
45000
```

BOSTON UNIVERSITY
METROPOLITAN COLLEGE

Agenda

1. Objects and classes
2. Inheritance
3. **Constructors in Java**
4. Polymorphism
5. Downcasting
6. Abstract classes
7. Interfaces

Constructor Rules for Java Inheritance*

➲**Automatic Call to Parent Class Constructor**: when a subclass (derived class) object is created, the constructor of the superclass (base class) is invoked first, automatically.

➲**Default Constructor**: If the superclass has a no-argument (default) constructor, it is called automatically when the subclass's constructor is invoked.

* Edited from chatGPT

Constructor Rules for Java Inheritance

➲ **Non-Default Constructor**: If the superclass does not have a no-argument constructor, or if you want a specific constructor of the superclass to be called, you must manually call it from the subclass's constructor.

This is done by using the super() call in the first line of the subclass constructor. The arguments you pass to super() must match the arguments of one of the superclass's constructors.

Constructor Rules for Java Inheritance

- ➲**No Inherited Constructors**: In Java, constructors are not members of a class and hence, they are not inherited by subclasses. Each class (subclass or superclass) has its own constructors.
- ➲**No Overriding of Constructors**: A subclass cannot override constructors from its superclass.
- ➲**Private Constructors**: If a superclass has only private constructors, it cannot be subclassed. This is a common technique to prevent a class from being subclassed or to make a class effectively "final".

Constructor Rules for Java Inheritance

- **Calling Superclass Constructor**: If you want to call a specific superclass constructor, you must use the super() call in the first line of your subclass's constructor. If you do not make a call to super(), the Java compiler will insert a no-argument super() call for you.

- **Constructor Overloading**: Similar to methods, constructors can be overloaded in both superclass and subclass. Overloading provides different ways of initializing an object.

Superclass Constructor Example

```
class Base
{
  public Base() {
     System.out.println("Base's no-arg constructor invoked.");
  }
  public Base(int n) {
      System.out.println("Base's constructor invoked with int param");
  }
  protected String getObjectDescription() {
     return "Object of class Base";
  }
}

public class Derived extends Base {
      public int derivedAttribute;
      public Derived(int aBAttributeValue) {
         System.out.println("Derived's constructor invoked with int parameter.");
         derivedAttribute = aBAttributeValue;
      }
}
```

Superclass Constructor Example

```
public class Main {

    public static void main(String[] args) {
        Derived aDerivedInstance =  new Derived(111);
        System.out.println(aDerivedInstance.getObjectDescription() +
                " with derivedAttribute = " + aDerivedInstance.derivedAttribute + ".");
    }
  }
```

Output

```
Base's no-arg constructor invoked.
Derived's constructor invoked with int parameter.
Object of class Base with derivedAttribute = 111.
```

Agenda

1. Objects and classes
2. Inheritance
3. Constructors in Java
4. **Polymorphism**
5. Downcasting
6. Abstract classes
7. Interfaces

An Origin of Polymorphism

Using the same verb in different contexts is a key aspect of language

run a race
run a company

Student Classes

```
public class Student {
    private int score;
    public Student(int aScore)
    {
        score = aScore;
    }
    public int getScore()
    {
        return score;
    }
    public void setScore(int aScore)
    {
        score = aScore;
    }
    public String computeGrade()
    // Returns "A"/B/C/D/F according to >= 90/80/70/60/otherwise
    {
        if (score >= 90) return "A";
        else if (score >= 80) return "B";
             else if (score >= 70) return "C";
                    else if (score >= 60) return "D";
                     else  return "F";
    }
}
```

```java
public class GradStudent extends Student {
   private String thesisTitle;
   public GradStudent(int aScore, String aTitle)
   {
      super(aScore);
      thesisTitle = aTitle;
   }
   public String getTitle()
   {
      return thesisTitle;
   }
   public void setTitle(String title)
   {
      thesisTitle = title;
   }
   public String computeGrade()
   // Returns "First/Second Class" according to getScore() >= 95
   {
       if (getScore() >= 95)
          return "First Class";
       else
          return "Second Class";
   }
}
```

GradStudent Class

NonDegreeStudent Subclass

```
public class NonDegreeStudent extends Student {

  public NonDegreeStudent(int aScore) {
    super(aScore);
  }

    public String computeGrade()
    {
      if (getScore() >= 75) return "Pass";
      else return "Fail";
    }
}
```

Polymorphism

```java
public class Main {
public static void main(String[] args)
{
ArrayList<Student> students = new ArrayList<>();
students.add(new Student(78));
students.add(new GradStudent(78, "Computer Science"));
students.add(new NonDegreeStudent(78));
for(Student s : students) { // demonstration of polymorphism
System.out.println("Student's score = " + s.getScore());
System.out.println("Student's grade = " + s.computeGrade());
}
}
}
```

Output

```
Student's score = 78
Student's grade = C
Student's score = 78
Student's grade = Second Class
Student's score = 78
Student's grade = Pass
```

Agenda

1. Objects and classes
2. Inheritance
3. Constructors in Java
4. Polymorphism
5. **Downcasting**
6. Abstract classes
7. Interfaces

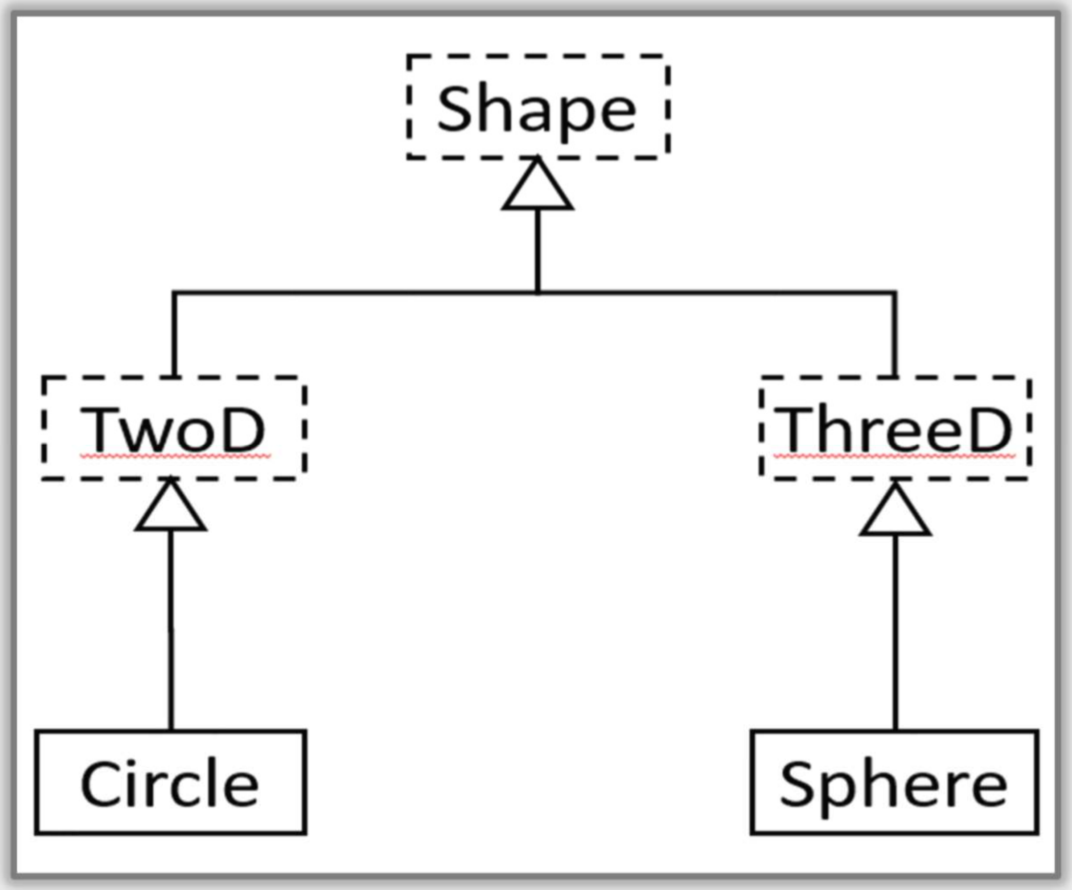
Downcasting Example

```
Student g = new GradStudent();
g.setTitle("My Thesis");  // doesn't compile
System.out.println("Thesis title = " + g.getTitle());  // doesn’t compile


// Downcast (if known it’s a grad student)
((GradStudent)g).setTitle("My Thesis");


// Downcast using instanceof first

if (g instanceof GradStudent)
    ((GradStudent)g).setTitle("My Thesis");
```

Agenda

1. Objects and classes
2. Inheritance
3. Constructors in Java
4. Polymorphism
5. Downcasting
6. **Abstract classes**
7. Interfaces

Abstract Class (No Instance)



Abstract Shape Classes

```
abstract class Shape
{
  public abstract double getArea();
}
abstract class TwoD extends Shape
{
}
abstract class ThreeD extends Shape
{
  public abstract double getVol();
}
class Circle extends TwoD
{
 private double radius;
 public Circle(double r)
 {
   radius = r;
 }
 public double getArea()
 {
   return Math.PI * radius * radius;
 }
}
```

Concrete Shape Classes

```
class Sphere extends ThreeD
{
 private double radius;

 public Sphere(double r)
 {
   radius = r;
 }

 public double getArea()
 {
   return 4.0 * Math.PI * radius * radius;
 }

 public double getVol()
 {
   return (4.0/3.0) * Math.PI * radius * radius * radius;
 }
}
```

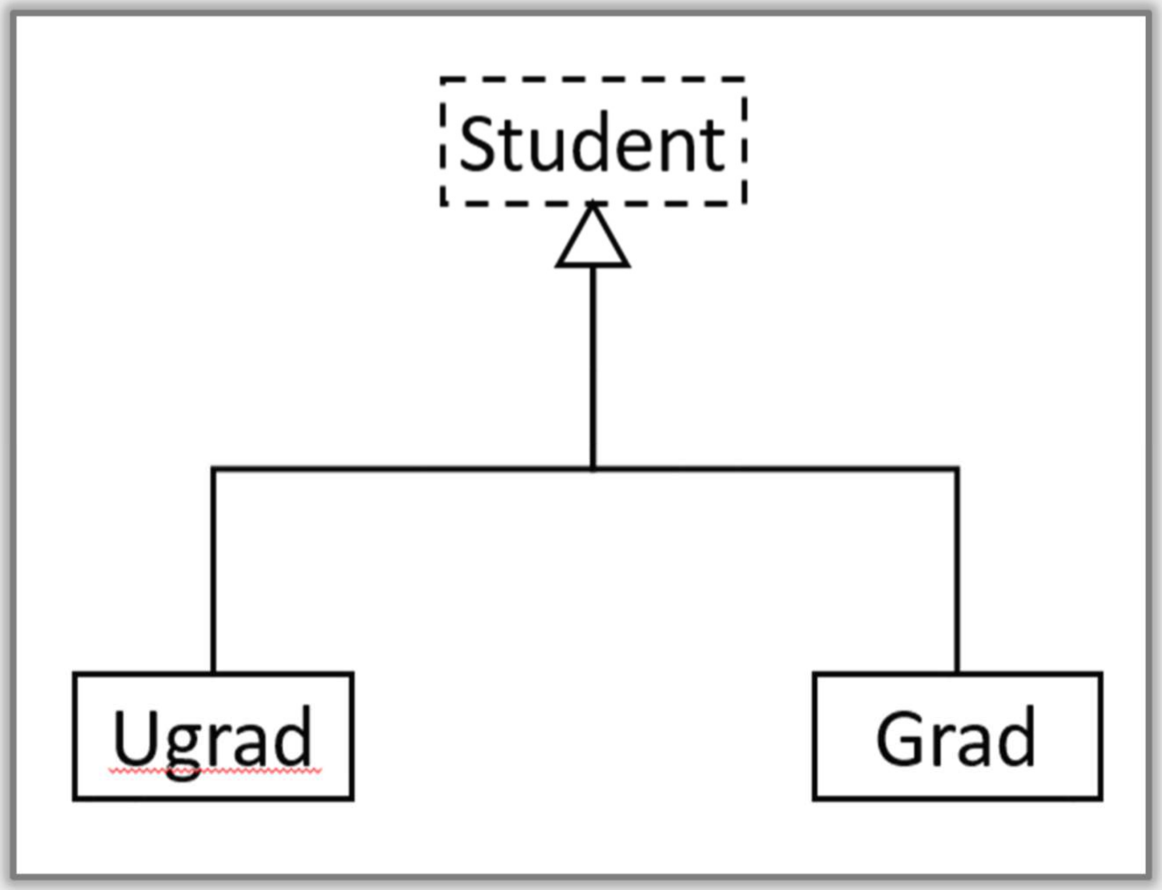
```java
public class Main
{
  public static void main(String[] args)
  {
    Shape[] shArr = {new Circle(1.0), new Sphere(1.0), new Sphere(10.0)};

    for (Shape elem : shArr)
    {
      if (elem != null)
      {
          System.out.println("Area = " + elem.getArea());
          if (elem instanceof ThreeD)
                System.out.println("Volume = " + ((ThreeD)elem).getVol());
      }
    }
  }
}
```

Shape Output

Output

```
Area = 3.141592653589793
Area = 12.566370614359172
Volume = 4.1887902047863905
Area = 1256.6370614359173
Volume = 4188.790204786391
```

Student Diagram with Abstract



Student Classes

```
abstract class Student
{
    public void computeGrade()
    {
       System.out.println("Student: pass");
    }
}
class Ugrad extends Student
{
    public void computeGrade()
    {
       super.computeGrade();
       System.out.println("Ugrad: with honors");
    }
}
class Grad extends Student
{
    public void computeGrade()
    {
       System.out.println("Grad: with distinction");
    }
}
```

Student Output

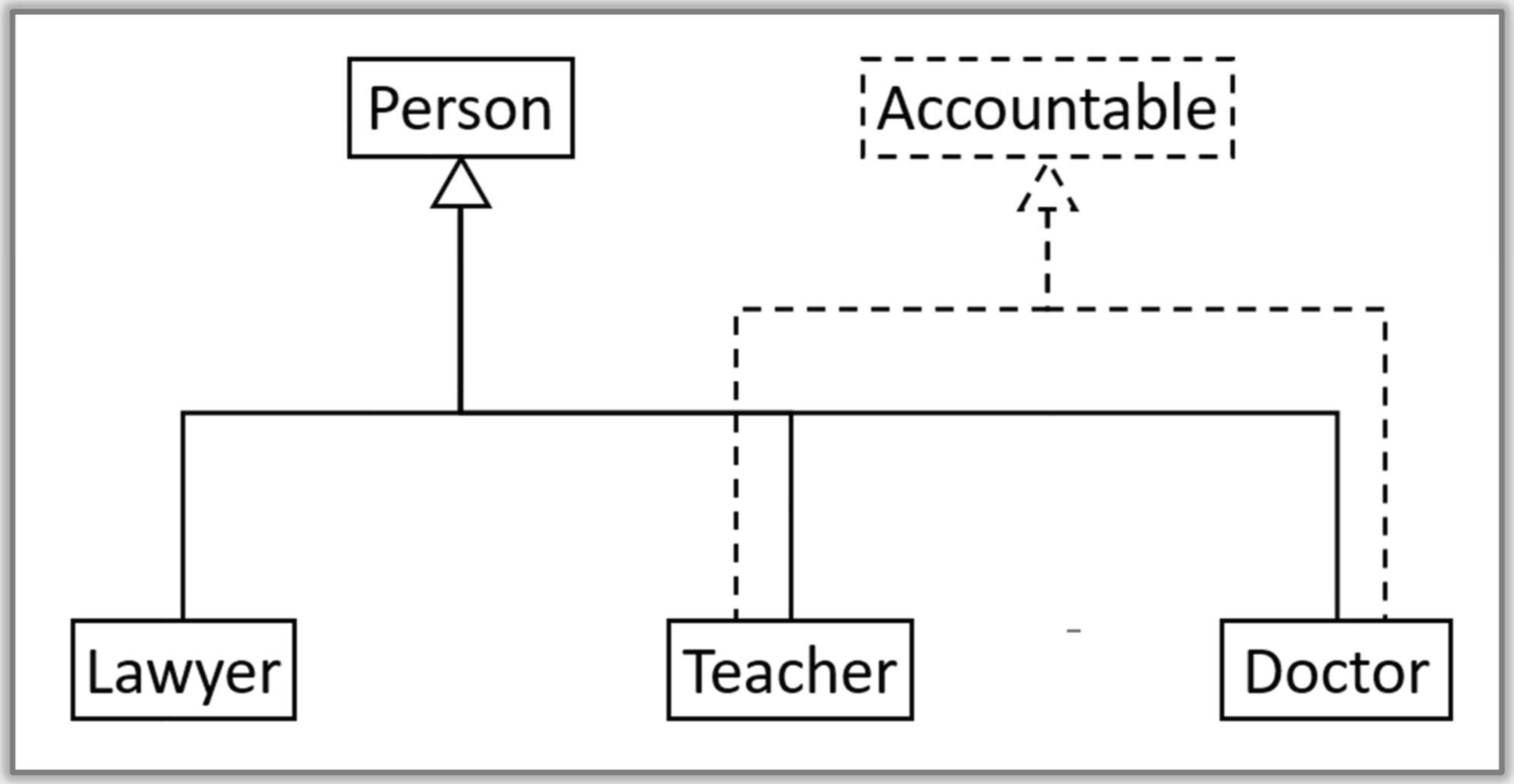
```java
public class Main
{
     public static void main(String[] args)
    {
       Student stuArr[] = {new Ugrad(), new Grad(), new Grad()};

       for (Student elem : stuArr)
           if (elem != null)
              elem.computeGrade();
    }
}
```

```
                    Output
Student: pass
Ugrad: with honors
Grad: with distinction
Grad: with distinction
```

Agenda

1. Objects and classes
2. Inheritance
3. Constructors in Java
4. Polymorphism
5. Downcasting
6. Abstract classes
7. **Interfaces**

Interfaces Allow Multiple "Inheritance"



Accountable Classes

```
interface Accountable
{
  public abstract void showAccountability();
}
class Person
{
   public void print()
   {
       System.out.print("I am a Person. ");
   }
}
class Teacher extends Person implements Accountable
{
  public void showAccountability()
  {
       System.out.println("I am a Teacher; I am accountable to my students.");
  }
}
class Doctor extends Person implements Accountable
{
  public void showAccountability()
  {
       System.out.println("I am a Doctor; I am accountable to my patients.");
  }
}
class Lawyer extends Person {}
```

Accountable Output

```java
public class Main
{
   public static void main(String[] args)
   {
      Person[] persons = {new Teacher(), new Doctor(), new Lawyer()};

      for (Person elem: persons)
          elem.print();
          if (elem instanceof Accountable)
          {
               Accountable accountable = (Accountable)elem;
               accountable.showAccountability();
          }
    }
}
```

```
Output

I am a person. I am a Teacher; I am accountable to my students.
I am a person. I am a Doctor; I am accountable to my patients.
I am a Person.
```

```java
public class DriversLicense implements Comparable<DriversLicense> {
  private String firstName;
  private String lastName;
  private int licenseID;

  public DriversLicense(String first, String last, int licenseID) {
    this.firstName = first;
    this.lastName = last;
    this.licenseID = licenseID;
  }

  public int compareTo(DriversLicense otherLicense) {
    int retVal;

    if (licenseID == otherLicense.licenseID)
    retVal = 0;
    else if (licenseID < otherLicense.licenseID)
    retVal = -1;
    else
    retVal = 1;


    return retVal;
  }
}
```

Comparable Interface

```java
public class Main {
  public static void main(String[] args) {
    DriversLicense SallysLicense = new DriversLicense("Sally", "Small", 5);
    DriversLicense BobsLicense = new DriversLicense("Bob", "Glass", 2);
    DriversLicense ElainasLicense = new DriversLicense("Elaina", "Afilia", 5);
    DriversLicense VishnusLicense = new DriversLicense("Vishnu", "Santhana", 10);

    System.out.println("Sally's license compared to Bob's license: "
      +  SallysLicense.compareTo(BobsLicense));
    System.out.println("Sally's license compared to Elaina's license: "
        +  SallysLicense.compareTo(ElainasLicense));
    System.out.println("Sally's license compared to Vishnu's license: "
        +  SallysLicense.compareTo(VishnusLicense));
  }
}
```

Output

```
Sally's license compared to Bob's license: 1
Sally's license compared to Elaina's license: 0
Sally's license compared to Vishnu's license: -1
```

DriversLicense Output

Summary: use strongly typed language …

–in the struggle to manage complexity–

to enforce what you intend