Module 1

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Lecture 1 – Object-Oriented Fundamentals

Learning Objectives

After successfully completing this part of the module, students will be able to do the following:

1. Interpret relations between classes.
2. Use an inheritance hierarchy.
3. Use an interface hierarchy.
4. Give examples of polymorphism.
5. Demonstrate dynamic binding.
6. Appraise downcasting.
7. Distinguish between concrete classes and abstract classes.
8. Compare and contrast abstract classes and interfaces.
9. Use the Comparable interface.

Module 1 Study Guide and Deliverables

Topics:	Lecture 1: Object-Oriented Fundamentals
Readings:	• Module 1 online content • Deitel & Deitel, Chapter 9 and Chapter 10 • Instructor's live class slides
Discussions:	Please post your introduction as soon as possible.
Assignments:	Assignment 1 due Tuesday, May 17 at 6:00 AM ET
Live Classrooms:	• Tuesday, May 10, from 6:00 PM to 8:00 PM ET • Live Office: Schedule with facilitators as long as there are questions

Module Welcome and Introduction

met_cs622_18_su1_ebraude_mod1 video cannot be displayed here

Classes and Objects

A class in Java represents an entity or a concept in a self-contained way. A class contains two basic types of components:

- Data (or fields or attributes)
- Operations on data (methods)

An object is created as an instance of a class. It is the packaging of the data and methods in the same unit, along with the setting of specific access privileges of data and methods of a class that leads to encapsulation and "information hiding."

For example, the class Car (shown below) may represent a (highly simplified view of a) real-world car. In our example, class Car has two (private) fields – model and milesDriven – and two methods – one constructor method and a second method named findPrice().

```
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;
```

```
    }
}
```

Objects of the Car class can be created with statements such as

```
Car mycar = new  Car("Corolla", 95000);
```

A method is invoked (called) on an object, such as mycar, as follows:

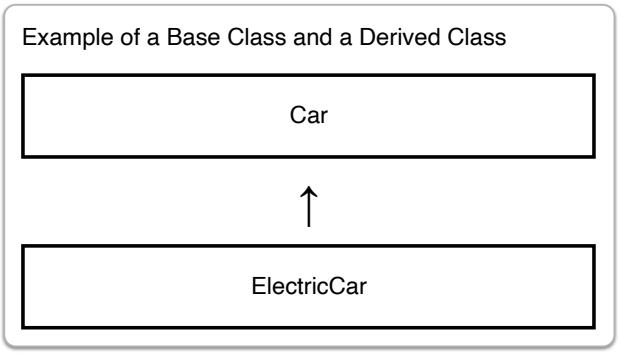
```
System.out.println(mycar.findPrice());
```

This module assumes that students are familiar with the basics of classes, objects, and object-oriented programming in Java. The remainder of this module provides a brief review of the major concepts in inheritance, polymorphism, abstract classes and interfaces.

Inheritance

Inheritance is a relationship between classes with common features. Inheritance facilitates code reuse and makes for superior program design, resulting in code that is more readable and maintainable. The main idea is that if two classes have something in common, then one class can probably be created (derived) from the other. For example, consider the relationship between cars and electric cars. Electric cars are a special type of car; all electric cars are cars but not all cars are electric cars. We can represent this relationship by creating a base class (also referred to as a parent class or a super class) named "Car" and a derived class (also called a child class or a sub-class) named "ElectricCar". We then say that the ElectricCar class inherits properties from the Car class (see the figure). The properties that are inherited by the subclass from the superclass include

- fields (attributes), and
- methods



The following program illustrates the inheritance relationship between classes Car and ElectricCar. (We assume that the Car class has already been defined.)

```
public class Cars
{
   public static void main(String[] args)
   {
       Car c = new Car("Camry", 48000);
       System.out.println(c.findPrice());

       ElectricCar ec = new ElectricCar("Camryel", 4000, 5);
       System.out.println(ec.findPrice());
```

```
    }
}


class ElectricCar extends Car
{
   private int  energyEfficiency;

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

   public int findPrice()
   {
      int temp = super.findPrice();
      if (energyEfficiency < 3)
         return temp + 20000;
      else
         return temp + 25000;
   }
}
```

The above program demonstrates the fact that an electric car is a car and then some more. A subclass inherits everything (all attributes and all methods with the exception of constructors) that a superclass has, and typically adds some (possibly zero) new attributes and/or some (possibly zero) new methods. A subclass can also provide a new definition to a method that it inherits; this is referred to as "overriding." In the above example, the ElectricCar class inherits method findPrice() from the Car class and then overrides it. Sometimes, while overriding a particular method, a subclass may retain the computation done in the method in the superclass, and then add something to it. This is achieved by a special method call "super". In the above example, ElectricCar uses much of the code of Car, thus avoiding "reinventing the wheel."

Calling a Superclass Constructor

If all the constructors of a superclass expect argument(s), then any subclass constructor must explicitly invoke one of the superclass constructors with a super(…) call placed as the very first statement of the subclass constructor. See the following example.

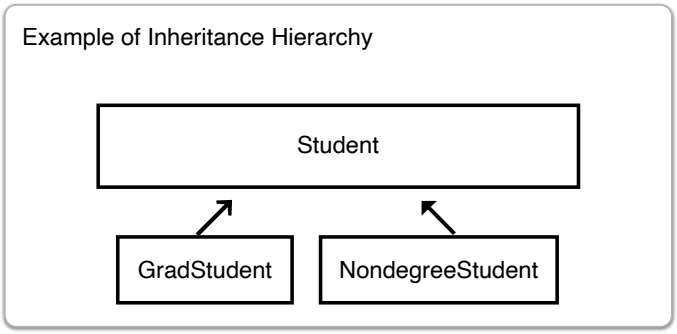
```
public class IsSuperMust
{
  public static void main(String[] args)
  {
      B b = new B(5);
  }
}

class A
{
  public  A() {System.out.println("A's no-arg constructor");}
  public A(int n) {System.out.println("A's constructor with int param");}
```

```
}

class B extends A
{
  private int bb;

  public B(int n)
  {
     System.out.println("B's constructor with int param");
     bb = n;
  }
}
```

In class B's constructor B(int n) in the above example, a programmer-written super() is NOT a must, because class A already has a no-argument constructor A() which gets invoked automatically by B's constructor. However, if A(int n) were the ONLY constructor in class A, a super(int …) would be required in B's constructor.

Polymorphism and Dynamic (Late) Binding

As another example of the inheritance hierarchy, let us now consider the hierarchy defined by students (the Student class) and two types of students, namely graduate students (the GradStudent class) and non-degree students (the NondegreeStudent class). See the following diagram.



The class Poly1 (below) illustrates the interaction between the classes. In an inheritance hierarchy, a polymorphic method is one that is overridden in at least one subclass. In the following program, method computeGrade() is polymorphic.

Java makes dynamic binding or late (run-time, as opposed to compile-time) binding. This means that the decision on which overridden version of the instance method will be called is made at run-time. In other words, the actual method that will be called is determined not by the compiler; the compiler just checks if it is legal to call the method, whereas the run-time environment determines which of the polymorphic versions will be called.

In the following program, the value of r, and thus the actual reference type assigned to the variable s is not known until run-time. The compiler checks whether or not the computeGrade() method can be called in a given context; the run-time environment figures out which of the three polymorphic versions of computeGrade() will be called.

```
// Polymorphism

public class Poly1
```

```
{
  public static void main(String[] args)
  {
     Student s = null;
     int r = (int)(Math.random() * 3);
     if (r == 0)
           s = new Student(105, 78);
     else if (r == 1)
                s = new GradStudent(240, 91, "Computer Science");
          else
                s = new NondegreeStudent(100, 70);

     System.out.println("Student's score = " + s.getScore());
     System.out.println("Student's grade = " + s.computeGrade());
  }
}


class Student
{
  private int id;
  private int score;

  public Student(int iden, int sco)
  {
     id = iden;
     score = sco;
  }

  public int getScore()
  {
    return score;
  }

  public void setScore(int sco)
  {
    score = sco;
  }

  public String computeGrade()
  {
    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";
  }
}

class GradStudent extends Student
{
  private String thesisTitle;
```

```
    public GradStudent(int iden, int sc, String title)
    {
       super(iden, sc);
       thesisTitle = title;
    }

    public String getTitle()
    {
      return thesisTitle;
    }

    public void setTitle(String title)
    {
      thesisTitle = title;
    }

    public String computeGrade()
    {
      if (getScore() >= 95) return "First Class";
      else return "Second Class";
    }
  }

  class NondegreeStudent extends Student
  {
    public NondegreeStudent(int iden, int sc)
    {
       super(iden, sc);
    }

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

Different runs of the above program are expected to produce different outputs, with a particular run producing one of three possible results. Output from several sample runs are shown below.

Student's score = 91
Student's grade = Second Class

Student's score = 78
Student's grade = C

Student's score = 91
Student's grade = Second Class

Student's score = 91
Student's grade = Second Class

Student's score = 78
Student's grade = C

Student's score = 70
Student's grade = Fail

Downcasting

The following program illustrates a critically important aspect of potential type mismatches in using inheritance hierarchies. Consider the following statements in this program:

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

The setTitle and getTitle methods invoked on the g object do not work, because the type of the g object is Student, and class Student does not have any setTitle or getTitle method. Note that the g reference is set to a GradStudent, but that doesn't negate the fact that g's declared type is Student. At compile-time, the compiler checks that any method invoked on a Student object (such as g) is indeed a method in the Student class. This must not be confused with polymorphism. The method setTitle and getTitle are NOT polymorphic in the program under consideration. These two methods are present in the GradStudent class but NOT in the Student class.

The problem can be solved by using an explicit cast:

```
((GradStudent)g).setTitle("My Thesis")
```

and

```
((GradStudent)g).getTitle())
```

This is an example of downcasting, or casting from a parent (superclass) to a child (subclass). In the present program, the downcast is guaranteed to work because g was indeed set to a GradStudent reference. In general, however, before using a downcast such as this one, we need to test whether the object is indeed an instance of the subclass. This is the reason why the following is a popular idiom in Java:

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

```
// Type issues: downcasting
public class TestStudent2
{
  public static void main(String[] args)
  {
     Student s = new Student();
     s.setScore(76);
     System.out.println("Student's score = " + s.getScore());
     System.out.println("Student's grade = " + s.computeGrade());

     Student g = new GradStudent();
```

```
        g.setScore(99);
 //     g.setTitle("My Thesis"); // doesn't compile
        System.out.println("GradStudent's score = " + g.getScore());
        System.out.println("GradStudent's grade = " + g.computeGrade());
//      System.out.println("GradStudent's thesis title = " + g.getTitle());
                           // doesn't compile

        System.out.println("GradStudent's thesis title = " +
                        ((GradStudent)g).getTitle());  // downcasting
    }
}


class Student
{
  private int id;
  private int score;


  public int getScore()
  {
    return score;
  }

  public void setScore(int sco)
  {
    score = sco;
  }

  public String computeGrade()
  {
    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";
  }
}


class GradStudent extends Student
{
  private String thesisTitle;

  public String getTitle()
  {
    return thesisTitle;
  }

  public void setTitle(String title)
  {
    thesisTitle = title;
```

```
  }


  public String computeGrade()
  {
    if (getScore() >= 95) return "First Class";
    else return "Second Class";
  }
}
```

The type mismatch problem in the above program would not exist if g were declared a GradStudent object, as follows:

```
GradStudent g = new  GradStudent();
```

But the original style (`Student g = new GradStudent()`) is generally preferred because that allows the benefits of polymorphism to be exploited.

The issue of downcasting is further explored in the following program which illustrates a common pitfall in these situations, ClassCastException.

```
public class TestStudent3
{
  public static void main(String[] args)
  {
     Student[] starray = {new Student(6),  new GradStudent(), new Student(4)};

     for (Student st: starray)
        System.out.println("Id = " + st.getId());


     Student s = new GradStudent();
     // s.showTitle(); // does not compile

     if (s instanceof GradStudent)
      ((GradStudent)s).showTitle();

     Student s5 = new Student(1000);
     ((GradStudent)s5).showTitle();   //run-time class-cast exception
  }
}


class Student
{
  private int id = 7;

  public Student(int n)
  {
    System.out.println("Called with n = " + n);
    id = n;
  }

  public int getId() {return id;}
```

```
}


class GradStudent extends Student
{
  private String thesisTitle = "A Masterpiece";

  public GradStudent()
  {
    super(55555);
  }


  public void showTitle()
  {
    System.out.println(thesisTitle);
  }
}
```

In the above code, s.showTitle() does not compile, while

```
if (s instanceof GradStudent)
  ((GradStudent)s).showTitle();
```

works fine. And ((GradStudent)s5).showTitle() results in a ClassCastException because s5 is not an instance of GradStudent. (Exception handling is covered in Module 2.) The above program produces the following output:

```
Called with n = 6
Called with n = 55555
Called with n = 4
Id = 6
Id = 55555
Id = 4
Called with n = 55555
A Masterpiece
Called with n = 1000
Exception in thread "main" java.lang.ClassCastException: Student cannot be cast to GradStudent
        at TestStudent3.main …
```

Type Issues

We explore the issue of type (mis)match further with the following example where, again, the hierarchy consists of a Student superclass and a Grad subclass. (The insides of classes Student and Grad are left empty to simplify the code and focus exclusively on type matching.)

```
public class ArgumentTypeMatchSubclassSuperclass
{
  public static void main(String[] args)
  {
```

```
        Student s = new Grad();
        m(s);       // only one choice: m(Student)


        Grad g = new Grad();
        m(g);       // two choices, but m(Grad) closer match
      }

      public static void m(Student st)
      {
        System.out.println("m(Student) called …");
      }

      public static void m(Grad gr)
      {
        System.out.println("m(Grad) called …");
      }
    }

    class Student
    {
    }

    class Grad extends Student
    {
    }
```

The above program shows two overloaded static methods m(Student) and m(Grad). The output is:

```
m(Student) called …
m(Grad) called …
```

The reason is that the m(s) call matches m(Student), and the m(g) call matches both m(Student) and m(Grad) but chooses the latter because it is a closer match.

Now, if we delete the m(Grad) method (keeping only the m(Student)), both the invocations will use m(Student), and the program's output will be:

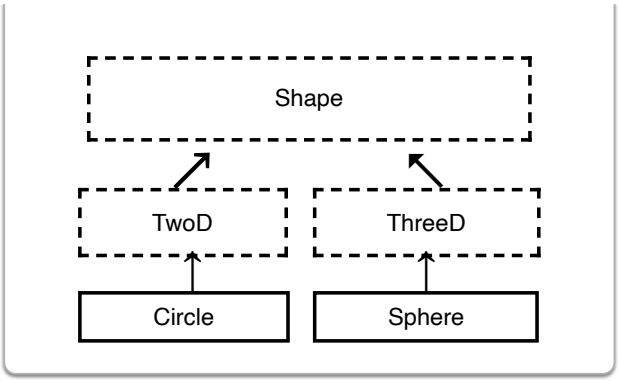
```
m(Student) called …
m(Student) called …
```

If, however, we delete the m(Student) method (keeping only the m(Grad)), the program will not compile, because in that case the m(s) invocation would not be able to find a matching argument (note that not all Students are Grad). (The m(g) call would be OK, though.)

Abstract Class

Consider the inheritance hierarchy shown in the following diagram and the class ShapeTwoLevel that uses this hierarchy.

Example of Abstract Class




```
public class ShapeTwoLevel
{
  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());
      }
    }
  }
}

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;
  }
}

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;
  }
}
```

This example illustrates abstract classes. An abstract class is a class that represents a concept that is meant to be augmented and realized by subclasses. An abstract class is never instantiated (it is not meant to be instantiated by definition). A non-abstract, or concrete, class that inherits from the abstract class is responsible for implementing the abstract concept. In the present example, Shape, TwoD and ThreeD are abstract classes, while Circle and Sphere are concrete classes. An abstract class may contain abstract methods. An abstract method has no method implementation (method body); it has only the header (declaration). Any concrete subclass of an abstract class must implement (override) all the abstract methods of the abstract class. If a subclass fails to implement *all* the abstract methods of its superclass(es), it must be declared abstract.

The output of the program is shown below. Note that for a sphere, both area and volume are computed.

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

If a class contains at least one abstract method, it must be declared abstract. However, an abstract class does not necessarily have to have an abstract method, as demonstrated in the following program. This makes sense, because the processing specified in the concrete method(s) of an abstract class may need to be shared by all the subclasses of the abstract (super)class, but at the same time, there may be valid reasons why we do not want any instance of the (abstract) class to be created.

In the following program, abstract class Student has no abstract method.

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

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

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");
    }
}
```

If every student, regardless of his/her status, needs to have the "pass" comment appear in the grade computation, then it makes sense to put that part (just that part alone) in the superclass.

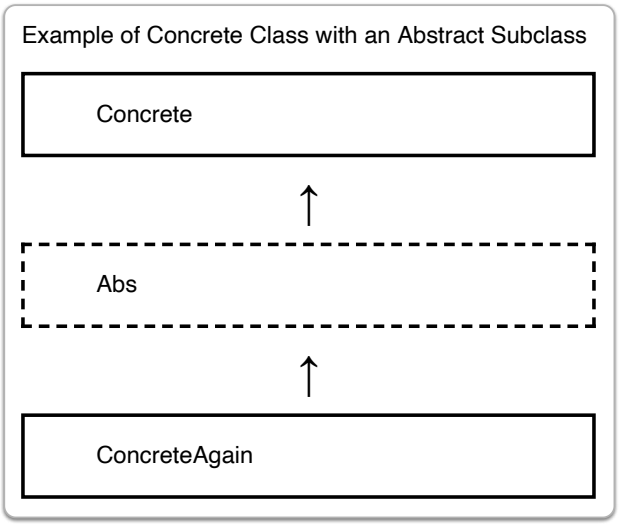
Note that no need exists for a Student object to be created, because there never is a Student (the Student class is conceptual or abstract, not physical) – every single Student must be either a Grad or a UGrad.

Here is the output of the program:

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

Concrete Superclass with Abstract Subclass

It is possible for a concrete class to have an abstract subclass. See the diagram below and the accompanying code (class ConcreteClassCanHaveAbstractSubclass).




```
public class ConcreteClassCanHaveAbstractSubclass
{
  public static void main(String[] args)
  {
      ConcreteAgain ca = new ConcreteAgain();
      ca.f();
  }
}

class Concrete
{
   private int i = 100;
   public void show() {System.out.println("i = " + i);}
}

abstract class Abs extends Concrete
{
  abstract void f();
}

class ConcreteAgain extends Abs
{
  void f()
  {
      System.out.println("hi");
      show();
  }
}
```

The output of the program is:

```
hi
i = 100
```

Test Yourself 1.1

Direction: Please read the question, think carefully, figure out your own answer first, and then click "Show Answer" to compare yours to the suggested answer.

Question: If the following statement is inserted as the first statement in method main() in class ConcreteClassCanHaveAbstractSubclass in the notes,

```
Concrete c = new Concrete();
```

what will be the change in the output of the program?

Suggested answer: No change.

Interface

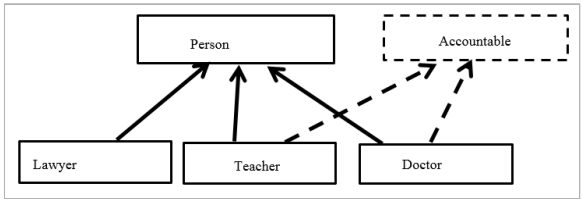
An interface is a construct that is similar to a class but contains only constants and abstract methods. An interface defines a type (much the same way that a concrete or abstract class does). The type defined by an interface is an abstract or conceptual type, in the sense that no instance (object) of an interface can be created. Like the inheritance hierarchy, interfaces and classes create an interface hierarchy. Just like a (sub)class "extends" another (super)class, a class "implements" an interface. Interfaces are an excellent mechanism for exploiting common characteristics in *unrelated* classes.

The methods in an interface must be public abstract. A class that "implements" an interface must provide code (method body) for each abstract method in the interface, or else the class will have to be abstract.

The fundamental difference between an interface and an abstract class is that the abstract class is used amongst *related* classes, while an interface is meant to be used amongst *unrelated* classes. (An interface cannot contain any non-constant field, but an abstract class can.)

We define and use an interface named Accountable in the following example. The entities represented in different classes may be accountable (in the English language sense) to different parties. The present example represents a hypothetical system of accountability using the interface Accountable.

Example of Interface




```
public class InterfaceAccountable
```

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

      for (Person elem: persons)
          if (elem instanceof Accountable)
               ((Accountable)elem).showAccountability();
          else elem.print();
   }
}

interface Accountable
{
  public abstract void showAccountability();
}

class Person
{
   public void print()
   {
       System.out.println("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
{
}
```

The program produces the following output:

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

The above example made the Lawyer, Doctor and Teacher classes inherit from the Person class; that made Lawyers, Doctors, and Teachers related. But this relationship was not necessary for the interface to work. As mentioned earlier, a number of

absolutely unrelated classes may each implement the same interface. Of course, one may ask how, in that case, the unrelated classes continue to be "unrelated" – am implicit relationship exists through the (shared) need to use the same interface. At that stage, the question boils down to one of the English language semantics. A possible example of "unrelated" classes implementing the same interface could be the classes Person, Company, and Country (where none is in a subclass/superclass relationship to any other) being Accountable to appropriate parties.

Comparable Interface

The Comparable interface provided by Java is very helpful in comparing two object instances. This interface defines the compareTo() method:

```
public interface Comparable
{
    int compareTo(Object obj);
}
```

This method is not static; it compares the implicit (this) object with the argument object and returns a negative integer, zero, or a positive integer depending on whether the implicit object is less than, equal to, or greater than the argument object.

(Comparable<E>, the generic version of Comparable, is encountered later in this course.)

Study the following example:

```
public class TestComp
{
  public static void main(String[] args)
  {
    MyShape s = new MyShape();
    Student stu  = new Student();

    MyShape s2 = new MyShape();

    System.out.println("Result = " + s.compareTo(stu));
    System.out.println("Result = " + s.compareTo(s2));
  }
}

class MyShape implements Comparable
{
   private int r;

   public int compareTo(Object o)
   {
     if (!(o instanceof MyShape))
       return -100;

     if (r < ((MyShape)o).r) return -1;
     else if (r > ((MyShape)o).r) return 1;
          else return 0;
   }
```

```
}

class Student
{
}
```

Note the two different calls to compareTo, and observe what happens when a MyShape object is compared to a Student object. The program produces the following output:

```
Result = -100
Result = 0
```

Using Comparable as a Type

Using Comparable in type declarations is a popular strategy (see the following program), but note that the following statement is illegal:

```
Comparable wrong = new Comparable(); // Invalid
```

In contrast,

```
Object aaa = new Object(); // OK
```

is fine, because Object is a concrete class.

```
public class ArrayOfObjects
{
            public static void main(String[] args)
            {
          Object[] objArr = {new Integer(100), new String("Java"), new String("good")};

          for (Object elem: objArr)
                   System.out.println(elem);

                Comparable[] arr = {new Integer(100), new String("Java"), new String("good")};
                // Integers and Strings  implement Comparable

          for (Comparable elem: arr)
                   System.out.println(elem);
            }
}
```

The output is:

```
100
Java
good
100
Java
good
```

Generic Interface

Generic interfaces, or interfaces that require a reference type to be used as a parameter to the interface, are extremely useful. (We discuss generics in detail in Module 3.) We define a generic interface GenericInterface<E> in the following example and illustrate two different uses of the interface in GenericInterface<Circle> and GenericInterface<SomeClass>, where Circle and SomeClass are concrete classes.

```
public class GenericInterfaceDemo
{
   public static void main(String[] args)
   {
     Circle c1 = new Circle(),
            c2 = new Circle();
     c1.myprint(c2);

     SomeClass s1 = new SomeClass(),
               s2 = new SomeClass();
     s1.myprint(s2);
   }
}


interface GenericInterface<E>
{
   public void myprint(E e);
}

class Circle implements GenericInterface<Circle>
{
   private double radius = 3.14;

   public void myprint(Circle c)
   {
     System.out.println("The input-parameter Circle has radius = " +  c.radius);
     System.out.println("And this.radius is " + this.radius);
   }
}

class SomeClass implements GenericInterface<SomeClass>
{
  public void myprint(SomeClass s)
  {
      System.out.println("this object = " + this);
      System.out.println("And input object = " + s);
  }
}
```

The output is:

```
The input-parameter Circle has radius = 3.14
And this.radius is 3.14
```

```
this object = SomeClass@2a139a55
And input object = SomeClass@15db9742
```

Module 1 Practice Questions

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer first, and then click "Show Answer" to compare yours to the suggested answer.

Test Yourself 1.2

What is the output of the following program (answer only by reading the code, without executing)?

```
public class TestSuper
{
 public static void main(String[] args)
 {
   C c = new C();
   c.method();
 }
}

class A
{
  public void nonpolymorphicMethodA()
  {
    System.out.println("A's nonpoly method");
  }

  public void polymorphicMethod()
  {
    System.out.println("A's poly method");
  }
}

class B extends A
{
  public void nonpolymorphicMethodB()
  {
    System.out.println("B's nonpoly method");
  }

  public void polymorphicMethod()
  {
    System.out.println("B's poly method");
  }
```

```
  public void m()
  {
    super.polymorphicMethod();
    super.nonpolymorphicMethodA();
  }
}

class C extends B
{
  public void method()
  {
    super.polymorphicMethod(); // no way to get A's
                               // polymorphicMethod()
    super.nonpolymorphicMethodA();
  }
}
```

Suggested answer:

B's poly method

A's nonpoly method

Test Yourself 1.3

What is the output of the following program? (Answer only by reading the code, without executing.)

```
public class TT
{
  public static void main(String[] args)
  {
    Student s = new Doctoral();
    s.f();
  }
}

interface  Edible
{
  public void f();
}

class Student
{
  public void f() {System.out.println("Hi there …");}
}

class Grad extends Student implements Edible
{
  public void show() {System.out.println("Grad …");}
}
```

```
class Doctoral extends Grad
{
}
```

Suggested answer: Hi there …

Test Yourself 1.4

What is the output of the following code? (Answer only by reading the code, without executing.)

```
public class TestConstructor
{
  public static void main(String[] args)
  {
    Alpha a1 = new Alpha();
    a1.show();

    Alpha a2 = new Alpha(99999);
    a2.show();
  }
}


class Alpha
{
   private int i = 172;
   private int j;

   public Alpha() {i = 100; j = 100;}
   public Alpha(int n) {j = n;}
   public void show() { System.out.println("i = " + i + " j = " + j);}
}
```

Suggested answer:

```
i = 100 j = 100
i = 172 j = 99999
```