Lecture 07a - Polymorphism

CS202: Introduction to Object Oriented Programming
Victor Mejia
CSULA

Today's Topics:

- Quick Recap - Inheritance
- Polymorphism
- Dynamic Binding
- The Object class
- ArrayList

Quick Recap - Inheritance

- *Inheritance* is one of the main techniques of object-oriented programming (OOP)
- Using this technique, a very general form of a class is first defined and compiled, and then more specialized versions of the class are defined by adding instance variables and methods
  - The specialized classes are said to *inherit* the methods and instance variables of the general class

Quick Recap - Inheritance

- Inheritance is the process by which a new class is created from another class
  - The new class is called a *derived class*
  - The original class is called the *base class*
- A derived class automatically has all the instance variables and methods that the base class has, and it can have additional methods and/or instance variables as well
- Inheritance is especially advantageous because it allows code to be *reused*, without having to copy it into the definitions of the derived classes

Derived Classes

- Within Java, a class called **`Employee`** can be defined that includes all employees
- This class can then be used to define classes for hourly employees and salaried employees
  - In turn, the **`HourlyEmployee`** class can be used to define a **`PartTimeHourlyEmployee`** class, and so forth

Overriding a Method Definition

- Although a derived class inherits methods from the base class, it can change or *override* an inherited method if necessary
  - In order to override a method definition, a new definition of the method is simply placed in the class definition, just like any other method that is added to the derived class

Changing the Return Type of an Overridden Method

- Ordinarily, the type returned may not be changed when overriding a method
- However, if it is a class type, then the returned type may be changed to that of any descendent class of the returned type
- This is known as a *covariant return type*
  - *Covariant return types* are new in Java 5.0; they are not allowed in earlier versions of Java

Covariant Return Type

- Given the following base class:

```
public class BaseClass
{ . . .
  public Employee getSomeone(int someKey)
  . . .
```

- The following is allowed in Java 5.0:

```
public class DerivedClass extends BaseClass
{ . . .
  public HourlyEmployee getSomeone(int someKey)
  . . .
```

Changing the Access Permission of an Overridden Method

- The access permission of an overridden method can be changed from private in the base class to public (or some other more permissive access) in the derived class
- However, the access permission of an overridden method can not be changed from public in the base class to a more restricted access permission in the derived class

Changing the Access Permission of an Overridden Method

- Given the following method header in a base case:
  `private void doSomething()`
- The following method header is valid in a derived class:
  `public void doSomething()`
- However, the opposite is not valid
- Given the following method header in a base case:
  `public void doSomething()`
- The following method header is <u>not</u> valid in a derived class:
  `private void doSomething()`

The `this` Constructor

- Within the definition of a constructor for a class, `this` can be used as a name for invoking another constructor in the same class
  - The same restrictions on how to use a call to `super` apply to the `this` constructor
- If it is necessary to include a call to both `super` and `this`, the call using `this` must be made first, and then the constructor that is called must call `super` as its first action

The `this` Constructor

- Often, a no-argument constructor uses `this` to invoke an explicit-value constructor
  - No-argument constructor (invokes explicit-value constructor using `this` and default arguments):

```
public ClassName()
{
   this(argument1, argument2);
}
```

  - Explicit-value constructor (receives default values):

```
public ClassName(type1 param1, type2 param2)
{
   . . .
}
```

The `this` Constructor

```
public HourlyEmployee()
{
   this("No name", new Date(), 0, 0);
}
```

- The above constructor will cause the constructor with the following heading to be invoked:

```
public HourlyEmployee(String theName,
  Date theDate, double theWageRate,
  double theHours)
```

Tip: An Object of a Derived Class Has More than One Type

- An object of a derived class has the type of the derived class, and it also has the type of the base class
- More generally, an object of a derived class has the type of every one of its ancestor classes
  - Therefore, an object of a derived class can be assigned to a variable of any ancestor type

Tip: An Object of a Derived Class Has More than One Type

- An object of a derived class can be plugged in as a parameter in place of any of its ancestor classes
- In fact, a derived class object can be used anyplace that an object of any of its ancestor types can be used
- Note, however, that this relationship does not go the other way
  - An ancestor type can never be used in place of one of its derived types

`Protected` and `Package` Access

- If a method or instance variable is modified by **`protected`** (rather than **`public`** or **`private`**), then it can be accessed *by name*
  - Inside its own class definition
  - Inside any class derived from it
  - In the definition of any class in the same package
- The **`protected`** modifier provides very weak protection compared to the **`private`** modifier
  - It allows direct access to any programmer who defines a suitable derived class
  - Therefore, instance variables should normally not be marked **`protected`**

`Protected` and `Package` Access

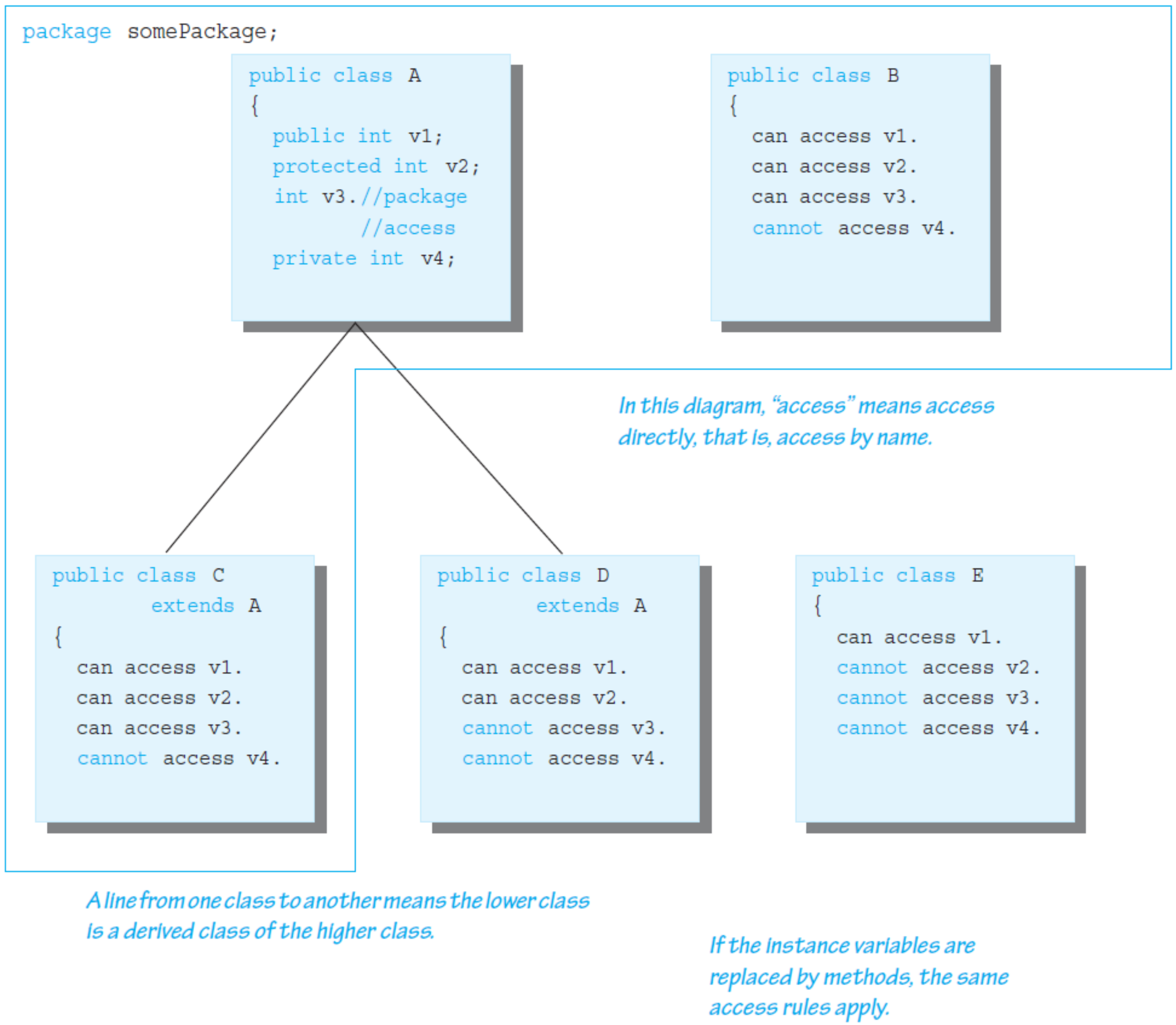
- An instance variable or method definition that is not preceded with a modifier has *package access*
  - Package access is also known as *default* or *friendly access*
- Instance variables or methods having package access can be accessed *by name* inside the definition of any class in the same package
  - However, neither can be accessed outside the package

`Protected` and `Package` Access

- Note that package access is more restricted than `protected`
  - Package access gives more control to the programmer defining the classes
  - Whoever controls the package directory (or folder) controls the package access

Access Modifiers

Display 7.9 Access Modifiers



Tip: Static Variables Are Inherited

- Static variables in a base class are inherited by any of its derived classes
- The modifiers **`public`**, **`private`**, and **`protected`**, and package access have the same meaning for static variables as they do for instance variables

Access to a Redefined Base Method

- Within the definition of a method of a derived class, the base class version of an overridden method of the base class can still be invoked
  - Simply preface the method name with super and a dot

```
public String toString()
{
  return (super.toString() + "$" + wageRate);
}
```

- However, using an object of the derived class outside of its class definition, there is no way to invoke the base class version of an overridden method

You Cannot Use Multiple `supers`

- It is only valid to use `super` to invoke a method from a direct parent
  - Repeating `super` will not invoke a method from some other ancestor class
- For example, if the `Employee` class were derived from the class `Person`, and the `HourlyEmployee` class were derived form the class `Employee` , it would not be possible to invoke the `toString` method of the `Person` class within a method of the `HourlyEmployee` class

```
super.super.toString() // ILLEGAL!
```

The `Object` Class

The Class **`Object`**

- In Java, every class is a descendent of the class ***`Object`***
  - Every class has **`Object`** as its ancestor
  - Every object of every class is of type **`Object`**, as well as being of the type of its own class
- If a class is defined that is not explicitly a derived class of another class, it is still automatically a derived class of the class **`Object`**

The Class **`Object`**

- The class **`Object`** is in the package **`java.lang`** which is always imported automatically
- Having an **`Object`** class enables methods to be written with a parameter of type **`Object`**
  - A parameter of type **`Object`** can be replaced by an object of any class whatsoever
  - For example, some library methods accept an argument of type **`Object`** so they can be used with an argument that is an object of any class

The Class `Object`

- The class **`Object`** has some methods that every Java class inherits
  - For example, the **`equals`** and **`toString`** methods
- Every object inherits these methods from some ancestor class
  - Either the class **`Object`** itself, or a class that itself inherited these methods (ultimately) from the class **`Object`**
- However, these inherited methods should be overridden with definitions more appropriate to a given class
  - Some Java library classes assume that every class has its own version of such methods

The Right Way to Define `equals`

- Since the `equals` method is always inherited from the class `Object`, methods like the following simply overload it:

```
public boolean equals(Employee otherEmployee)
{ . . . }
```

- However, this method should be overridden, not just overloaded:

```
public boolean equals(Object otherObject)
{ . . . }
```

The Right Way to Define `equals`

- The overridden version of `equals` must meet the following conditions
  - The parameter `otherObject` of type `Object` must be type cast to the given class (e.g., `Employee)`
  - However, the new method should only do this if `otherObject` really is an object of that class, and if `otherObject` is not equal to `null`
  - Finally, it should compare each of the instance variables of both objects

A Better `equals` Method for the Class `Employee`

```
public boolean equals(Object otherObject)
{
  if(otherObject == null)
    return false;
  else if(getClass( ) != otherObject.getClass( ))
    return false;
  else
  {
    Employee otherEmployee = (Employee)otherObject;
    return (name.equals(otherEmployee.name) &&
      hireDate.equals(otherEmployee.hireDate));
  }
}
```

Tip: `getClass` Versus `instanceof`

- Many authors suggest using the `instanceof` operator in the definition of `equals`
  - Instead of the `getClass()` method
- The `instanceof` operator will return `true` if the object being tested is a member of the class for which it is being tested
  - However, it will return `true` *if it is a descendent of that class* as well
- It is possible (and especially disturbing), for the `equals` method to behave inconsistently given this scenario

Tip: **`getClass`** Versus **`instanceof`**

- Here is an example using the class **`Employee`**

```
. . . //excerpt from bad equals method
else if(!(OtherObject instanceof Employee))
  return false; . . .
```

- Now consider the following:

```
Employee e = new Employee("Joe", new Date());
HourlyEmployee h = new
  HourlyEmployee("Joe", new Date(),8.5, 40);
boolean testH = e.equals(h);
boolean testE = h.equals(e);
```

Tip: `getClass` Versus `instanceof`

- `testH` will be `true`, because `h` is an `Employee` with the same name and hire date as `e`
- However, `testE` will be `false`, because `e` is not an `HourlyEmployee`, and cannot be compared to `h`
- Note that this problem would not occur if the `getClass()` method were used instead, as in the previous `equals` method example

`instanceof` and `getClass`

- Both the `instanceof` operator and the `getClass()` method can be used to check the class of an object
- However, the `getClass()` method is more exact
  - The `instanceof` operator simply tests the class of an object
  - The `getClass()` method used in a test with `==` or `!=` tests if two objects *were created with* the same class

The `instanceof` Operator

- The `instanceof` operator checks if an object is of the type given as its second argument

  `Object instanceof ClassName`

  – This will return `true` if `Object` is of type `ClassName`, and otherwise return `false`

  – Note that this means it will return `true` if `Object` is the type of *any descendent class* of `ClassName`

The `getClass()` Method

- Every object inherits the same `getClass()` method from the `Object` class
  - This method is marked `final`, so it cannot be overridden
- An invocation of `getClass()` on an object returns a representation *only* of the class that was used with `new` to create the object
  - The results of any two such invocations can be compared with `==` or `!=` to determine whether or not they represent the exact same class

```
(object1.getClass() == object2.getClass())
```

A First Look at the `clone` Method

- Every object inherits a method named `clone` from the class `Object`
  - The method `clone` has no parameters
  - It is supposed to return a deep copy of the calling object
- However, the inherited version of the method was not designed to be used as is
  - Instead, each class is expected to override it with a more appropriate version

A First Look at the `clone` Method

- The heading for the `clone` method defined in the `Object` class is as follows:

  `protected Object clone()`
- The heading for a `clone` method that overrides the `clone` method in the `Object` class can differ somewhat from the heading above
  - A change to a more permissive access, such as from protected to public, is always allowed when overriding a method definition
  - Changing the return type from `Object` to the type of the class being cloned is allowed because every class is a descendent class of the class `Object`
  - This is an example of a covariant return type

A First Look at the `clone` Method

- If a class has a copy constructor, the **`clone`** method for that class can use the *copy constructor* to create the copy returned by the **`clone`** method

```
public Sale clone()
{
  return new Sale(this);
}
```

and another example:

```
public DiscountSale clone()
{
  return new DiscountSale(this);
}
```

Copy Constructor

```
public Sale(String theName, double thePrice)
{
    setName(theName);
    setPrice(thePrice);
}

public Sale(Sale originalObject)
{
    if (originalObject == null)
    {
        System.out.println("Error: null Sale object.");
        System.exit(0);
    }
    //else
    name = originalObject.name;
    price = originalObject.price;
}
```

The **hashCode** Method

- a hash code is an integer that is derived from an object
- hash codes should be scrambled
  - if x and y are two distinct objects, there should be a high probability that x.hashCode() and y. hashCode() are different
- The String class uses the following algorithm to compute the hash code:

```
int hash = 0;
for (int i = 0; i < length(); i++)
   has = 31 * hash + charAt(i)
```

The **hashCode** Method

```
String s = "Java";
StringBuilder sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());

String t = new String("Java");
StringBuilder tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

s	2301506
sb	1735600054
t	2301506
tb	21685669

The **hashCode** Method

```
String s = "Java";
StringBuilder sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());

String t = new String("Java");
StringBuilder tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

- **`s`** and **`t`** have the same hash code: for strings the hash codes are derived from their contents

- string builders **`sb`** and **`tb`** have different hash codes because no **`hashCode`** method has been defined for the **`StringBuilder`** class
  - default **`hashCode`** in the **`Object`** class derives the hash code from the object's memory address

The **hashCode** Method

- If you redefine the equals method, redefine the hashCode method for objects that users might insert into a hash table
  - data structure covered in CS203
- It should return an integer (can be negative)
- Just combine the hash codes of the instance fields

```
public int hashCode() {
  return 7 * name.hashCode()
       + 13 * new Double(price).hashCode();
}
```

The **hashCode** Method

- equals and hashCode must be compatible
    - if you define **`Sale.equals`** to compare name and price, **hashCode** needs to hash name and price also, not just name
- tip: if you have fields of array type, you can use the static **`Arrays.hashCode`** method to compute a hash code that is composed of the hash codes of the array elements
- an ArrayList object has a **`hashCode()`** method

Introduction to Polymorphism

- There are three main programming mechanisms that constitute object-oriented programming (OOP)
  - Encapsulation
  - Inheritance
  - Polymorphism
- Polymorphism is the ability to associate many meanings to one method name
  - It does this through a special mechanism known as *late binding* or *dynamic binding*

Introduction to Polymorphism

- Inheritance allows a base class to be defined, and other classes derived from it
  - Code for the base class can then be used for its own objects, as well as objects of any derived classes
- Polymorphism refers to a programming language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes.

Dynamic Binding

- The process of associating a method definition with a method invocation is called *binding*
- If the method definition is associated with its invocation when the code is compiled, that is called *early binding*
- If the method definition is associated with its invocation when the method is invoked (at run time), that is called *late binding* or *dynamic binding*

Dynamic Binding

- Java uses dynamic binding for all methods (except private, **`final,`** and static methods)
- Because of dynamic binding, a method can be written in a base class to perform a task, even if portions of that task aren't yet defined
- For an example, the relationship between a base class called **`Sale`** and its derived class **`DiscountSale`** will be examined

The `Sale` and `DiscountSale` Classes

- The `Sale` class contains two instance variables
  - `name`: the name of an item (`String`)
  - `price`: the price of an item (`double`)
- It contains three constructors
  - A no-argument constructor that sets `name` to `"No name yet"`, and price to `0.0`
  - A two-parameter constructor that takes in a `String` (for `name`) and a `double` (for `price`)
  - A copy constructor that takes in a `Sale` object as a parameter

The **`Sale`** and **`DiscountSale`** Classes

- The **`Sale`** class also has a set of accessors (**`getName`**, **`getPrice`**), mutators (**`setName`**, **`setPrice`**), overridden **`equals`** and **`toString`** methods, and a static **`announcement`** method
- The **`Sale`** class has a method **`bill`**, that determines the bill for a sale, which simply returns the price of the item
- It has two methods, **`equalDeals`** and **`lessThan`**, each of which compares two sale objects *by comparing their bills* and returns a **`boolean`** value

The **`Sale`** and **`DiscountSale`** Classes

- The **`DiscountSale`** class inherits the instance variables and methods from the **`Sale`** class
- In addition, it has its own instance variable, **`discount`** (a percent of the **`price`**), and its own suitable constructor methods, accessor method (**`getDiscount`**), mutator method (**`setDiscount`**), overriden **`toString`** method, and static **`announcement`** method
- The **`DiscountSale`** class has its own **`bill`** method which computes the bill as a function of the **`discount`** and the **`price`**

The **`Sale`** and **`DiscountSale`** Classes

- The **`Sale`** class **`lessThan`** method
  - Note the **`bill()`** method invocations:

```
public boolean lessThan (Sale otherSale)
{
  if (otherSale == null)
  {
    System.out.println("Error: null object");
    System.exit(0);
  }
  return (bill( ) < otherSale.bill( ));
}
```

The **`Sale`** and **`DiscountSale`** Classes

- The **`Sale`** class **`bill()`** method:

```
public double bill( )
{
  return price;
}
```

- The **`DiscountSale`** class **`bill()`** method:

```
public double bill( )
{
  double fraction = discount/100;
  return (1 - fraction) * getPrice( );
}
```

The **`Sale`** and **`DiscountSale`** Classes

- Given the following in a program:

```
   . . .
Sale simple = new sale("floor mat", 10.00);
DiscountSale discount = new
           DiscountSale("floor mat", 11.00, 10);
   . . .
if (discount.lessThan(simple))
  System.out.println("$" + discount.bill() +
                " < " + "$" + simple.bill() +
                " because late-binding works!");
   . . .
```

  - Output would be:

```
$9.90 < $10 because late-binding works!
```

The **`Sale`** and **`DiscountSale`** Classes

- In the previous example, the **`boolean`** expression in the **`if`** statement returns **`true`**
- As the output indicates, when the **`lessThan`** method in the **`Sale`** class is executed, it knows which **`bill()`** method to invoke
  - The **`DiscountSale`** class **`bill()`** method for **`discount`**, and the **`Sale`** class **`bill()`** method for **`simple`**
- Note that when the **`Sale`** class was created and compiled, the **`DiscountSale`** class and its **`bill()`** method did not yet exist
  - These results are made possible by late-binding

Pitfall: No Dynamic Binding for Static Methods

- When the decision of which definition of a method to use is made at compile time, that is called *static binding*
  - This decision is made based on the *type of the variable naming the object*
- Java uses static, not late, binding with private, **`final`**, and static methods
  - In the case of **`private`** and **`final`** methods, late binding would serve no purpose
  - However, in the case of a static method invoked using a calling object, it does make a difference

Pitfall: No Dynamic Binding for Static Methods

- The **Sale** class **announcement()** method:

```
public static void announcement( )
{
  System.out.println("Sale class");
}
```

- The **DiscountSale** class **announcement()** method:

```
public static void announcement( )
{
  System.out.println("DiscountSale class");
}
```

Pitfall: No Dynamic Binding for Static Methods

- In the previous example, the the **`simple`** (**`Sale`** class) and **`discount`** (**`DiscountClass`**) objects were created
- Given the following assignment:

```
simple = discount;
```

  - Now the two variables point to the same object
  - In particular, a **`Sale`** class variable names a **`DiscountClass`** object

Pitfall: No Dynamic Binding for Static Methods

- Given the invocation:

```
simple.announcement();
```

  - The output is:

```
Sale class
```

- Note that here, **`announcement`** is a static method invoked by a calling object (instead of its class name)
  - Therefore the type of **`simple`** is determined by its variable name, not the object that it references

The **`final`** Modifier

- A *method* marked **`final`** indicates that it cannot be overridden with a new definition in a derived class
  - If **`final`**, the compiler can use early binding with the method

```
public final void someMethod() { . . . }
```

- A *class* marked **`final`** indicates that it cannot be used as a base class from which to derive any other classes

Dynamic Binding with `toString`

- If an appropriate `toString` method is defined for a class, then an object of that class can be output using `System.out.println`

```
Sale aSale = new Sale("tire gauge", 9.95);
System.out.println(aSale);
```

  - Output produced:

```
tire gauge Price and total cost = $9.95
```

- This works because of late binding

Dynamic Binding with `toString`

- One definition of the method `println` takes a single argument of type `Object`:

```
public void println(Object theObject)
{
  System.out.println(theObject.toString());
}
```

  - In turn, It invokes the version of `println` that takes a `String` argument
- Note that the `println` method was defined before the `Sale` class existed
- Yet, because of late binding, the `toString` method from the `Sale` class is used, not the `toString` from the `Object` class

An Object knows the Definitions of its Methods

- The type of a class variable determines which method names can be used with the variable
  - However, the object named by the variable determines which definition with the same method name is used
- A special case of this rule is as follows:
  - The type of a class parameter determines which method names can be used with the parameter
  - The argument determines which definition of the method name is used

Upcasting and Downcasting

- *Upcasting* is when an object of a derived class is assigned to a variable of a base class (or any ancestor class)

```
Sale saleVariable; //Base class
DiscountSale discountVariable = new
   DiscountSale("paint", 15,10); //Derived class
saleVariable = discountVariable; //Upcasting
System.out.println(saleVariable.toString());
```

- Because of late binding, **`toString`** above uses the definition given in the **`DiscountSale`** class

Upcasting and Downcasting

- *Downcasting* is when a type cast is performed from a base class to a derived class (or from any ancestor class to any descendent class)
  - Downcasting has to be done very carefully
  - In many cases it doesn't make sense, or is illegal:

```
discountVariable =                //will produce
     (DiscountSale)saleVariable;//run-time error
discountVariable = saleVariable //will produce
                                //compiler error
```

  - There are times, however, when downcasting is necessary, e.g., inside the `equals` method for a class:

```
Sale otherSale = (Sale)otherObject;//downcasting
```

Pitfall: Downcasting

- It is the responsibility of the programmer to use downcasting only in situations where it makes sense
  - The compiler does not check to see if downcasting is a reasonable thing to do
- Using downcasting in a situation that does not make sense usually results in a run-time error

Tip: Checking to See if Downcasting is Legitimate

- Downcasting to a specific type is only sensible if the object being cast is an instance of that type
  - This is exactly what the **`instanceof`** operator tests for:
    `object instanceof ClassName`
  - It will return true if ***`object`*** is of type ***`ClassName`***
  - In particular, it will return true if ***`object`*** is an instance of any descendent class of ***`ClassName`***