Chapter 10 - Inheritance

Section 10.1 - Derived classes

Commonly, one class is similar to another class but with some additions or variations. For example, a store inventory system might use a class called GenericItem having itemName and itemQuantity members. But for produce (fruits and vegetables), a class ProduceItem having itemName, itemQuantity, and expirationDate members may be desired. Note that ProduceItem is really a GenericItem with an additional feature, so ideally a program could define the ProduceItem class as being the same as the GenericItem class but with the addition of an expirationDate member.

Such similarity among classes is supported by indicating that a class is derived from another class, as shown below.

Figure 10.1.1: A derived class example: Class ProduceItem is derived from class GenericItem.

```java
public class GenericItem {
    public void setName(String newName) {
        itemName = newName;
        return;
    }

    public void setQuantity(int newQty) {
        itemQuantity = newQty;
        return;
    }

    public void printItem() {
        System.out.println(itemName + " " + itemQuantity);
        return;
    }

    private String itemName;
    private int itemQuantity;
}
```

```java
public class ProduceItem extends GenericItem { // ProduceItem derived from GenericItem
    public void setExpiration(String newDate) {
        expirationDate = newDate;
        return;
    }
}
```

https://zybooks.zyante.com/#/zybook/LehmanCMP167Spring2016/chapter/10/print
A class named GenericItem is defined as normal. In main(), a GenericItem reference variable miscItem is initialized, the item’s data fields set to "Smith Cereal" and "9", and the item’s printItem() member method called. A class named ProduceItem is also defined, that class was derived from the GenericItem class by appending extends GenericItem after the name ProduceItem, i.e.,

class ProduceItem extends GenericItem {
    // ...
}

The term derived class (or subclass) refers to a class that is derived from another class that is known as a base class (or superclass). Any class may serve as a base class; no changes to the declaration of that class are required. The derived class is said to inherit the properties of its base class, a concept commonly called inheritance. An object defined of a derived class type has access to all the public members of the derived class as well as the public members of the base class. The following animation illustrates the relationship between a derived class and a base class.
Programmers commonly draw class inheritance relationships using *Unified Modeling Language (UML)* notation ([Wikipedia: UML]).
Various class derivation variations are possible:

- A derived class can itself serve as a base class for another class. In the earlier example, `class FruitItem extends ProduceItem {...}` could be added.

- A class can serve as a base class for multiple derived classes. In the earlier example, `class FrozenFoodItem extends GenericItem {...}` could be added.

- A class can only be derived from one base class directly. For example, inheriting from two classes as in `class House extends Dwelling, Property {...}` results in a compiler error.
10.1.3: Interactive inheritance tree.

Click a class to see available functions and data for that class.

Inheritance tree

- Item
  - Produce
    - Fruit
  - Book
    - Textbook
    - Audiobook
  - Dairy
### 10.1.4: Derived classes basic.

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Your answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A class that can serve as the basis for another class is called a _____ class.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Class Dwelling has data members door1, door2, door3. A class House is derived from Dwelling and has data members wVal, xVal, yVal, zVal. The definition and initialization House h = new House(); creates how many data members?</td>
<td></td>
</tr>
</tbody>
</table>

Exploring further:
- [Oracle’s Java tutorials on inheritance](https://docs.oracle.com/javase/tutorial/)
Section 10.2 - Access by members of derived classes

The members of a derived class have access to the public members of the base class, but not to the private members of the base class. This is logical—allowing access to all private members of a class merely by creating a derived class would circumvent the idea of private members. Thus, adding the following member method to the earlier ProduceItem class yields a compiler error.

```
// ===== start =====
public int getID() {
    return idNum;
}
// ===== end =====
```
Recall that members of a class may have their access specified as public or private. A third access specifier is protected, which provides access to derived classes and other classes in the same package but not by anyone else. Packages are discussed in detail elsewhere, but for our purposes a package can just be thought of as the directory in which program files are located. Thus, classes in the same package are located in the same directory. The following illustrates the implications of the protected access specifier.

```java
public class ProduceItem extends GenericItem {
    ...
    public void displayProduceItem() {
        System.out.println(itemName + " " + itemQuantity + " (Expires: " + expirationDate + ")");
    }
    ...
}
```

$ javac ProduceItem.java

ProduceItem.java:12: itemName has private access in GenericItem
    System.out.println(itemName + " " + itemQuantity + " (Expires: " + expirationDate + ");
         ^
ProduceItem.java:12: itemQuantity has private access in GenericItem
    System.out.println(itemName + " " + itemQuantity + " (Expires: " + expirationDate + ");
         ^
2 errors
Being specified as protected, the member called baseName is accessible anywhere in the derived class. Note that the baseName member is also accessible in main()—the protected specifier also allows access to classes in the same package; protected members are private to everyone else.

To make ProduceItems.displayProduceItem() method work, we merely need to change the private members to protected members in class GenericItem. GenericItem’s class members itemName and
itemQuantity thus become accessible to a derived class like ProduceItem. A programmer may often want to make some members protected in a base class to allow access by derived classes, while making other members private to the base class.

The following table summarizes access specifiers.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>Accessible by self.</td>
</tr>
<tr>
<td>protected</td>
<td>Accessible by self, derived classes, and other classes in the same package.</td>
</tr>
<tr>
<td>public</td>
<td>Accessible by self, derived classes, and everyone else.</td>
</tr>
<tr>
<td>no specifier</td>
<td>Accessible by self and other classes in the same package.</td>
</tr>
</tbody>
</table>

Separately, the keyword "public" in a class declaration like `public class DerivedClass {...}` specifies a class's visibility in other classes in the program:

- **public**: A class can be used by every class in the program regardless of the package in which either is defined.
- **no specifier**: A class can be used only in other classes within the same package, known as `package private`.

Most beginning programmers define classes as public when learning to program.
## Participation Activity

**10.2.1: Access by derived class members.**

Assume `public class DerivedClass extends BaseClass {...}`

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Your answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BaseClass' public member method can be called by a member method of DerivedClass.</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>BaseClass' protected member method can be called by a member method of DerivedClass.</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>BaseClass' private field can be accessed by a member method of DerivedClass.</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>For <code>DerivedClass derivedObj = new DerivedClass();</code> in <code>main()</code>, derivedObj can access a protected member of BaseClass. Assume <code>main()</code> is defined in a class located in the same package as DerivedClass.</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>For <code>BaseClass baseObj = new BaseClass();</code> in <code>main()</code>, baseObj can access a protected member of BaseClass. Assume <code>main()</code> is defined in a class located in a <strong>different</strong> package as BaseClass.</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>

### Exploring further:

---

[https://zybooks.zyante.com/#/zybook/LehmanCMP167Spring2016/chapter/10/print](https://zybooks.zyante.com/#/zybook/LehmanCMP167Spring2016/chapter/10/print)
Section 10.3 - Overriding member methods

A derived class may define a member method having the same name as the base class. Such a member method overrides the method of the base class. The following example shows the earlier GenericItem/ProduceItem example where the ProduceItem class has its own printItem() member method that overrides the printItem() method of the GenericItem class.

Figure 10.3.1: ProduceItem's printItem() method overrides GenericItem's printItem() method.

GenericItem.java:
```java
public class GenericItem {
    public void setName(String newName) {
        itemName = newName;
        return;
    }

    public void setQuantity(int newQty) {
        itemQuantity = newQty;
        return;
    }

    public void printItem() {
        System.out.println(itemName + " " + itemQuantity);
        return;
    }

    protected String itemName;
    protected int itemQuantity;
}
```

ProduceItem.java:
```java
public class ProduceItem extends GenericItem {
    public void setExpiration(String newDate) {
        expirationDate = newDate;
        return;
    }

    public String getExpiration() {
        return expirationDate;
    }

    @Override
    public void printItem() {
        System.out.print(expirationDate + " " + itemQuantity);
        return;
    }
}
```

More on access specifiers from Oracle's Java tutorials
Overriding differs from overloading. In overloading, methods with the same name must have different parameter types. In overriding, a derived class member method takes precedence over base class member method with the same name and parameter types. Overloading is performed if derived and base member methods have different parameter types; the member method of the derived class does not hide the member method of the base class.

Notice that the annotation @Override appears above the printItem() method definition in the ProduceItem class. Annotations are optional notes beginning with the '@' symbol that can provide the compiler with useful information in order to help the compiler detect errors better. The override annotation lets the compiler know that the programmer intends to define a method that will override a method in a base class. This annotation will cause the compiler to produce an error when a programmer mistakenly specifies parameters that are different from the parameters of the method that should be overridden. A good practice is to always include an override annotation with methods that are meant to override methods in a base class.

The following shows an example of how the override annotation helps the compiler detect inconsistencies in the manner in which ProduceItem overrides GenericItem’s printItem() method in what would otherwise be valid code.
The overriding function can still call the overridden method by using the \texttt{super} keyword, as in \texttt{super.printItem()}, as follows.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image1.png}
\caption{The override annotation helps the compiler detect incorrect method overriding.}
\end{figure}

\begin{verbatim}
public class ProduceItem extends GenericItem {
    // Other methods ...

    @Override
    public void printItem(int someInt) {
        System.out.println(itemName + " " + itemQuantity + 
            " (Expires: " + expirationDate + ")");
        return;
    }

    // Other fields ...
}
\end{verbatim}

$ javac ProduceItem.java
ProduceItem.java:11: method does not override or implement a method from a supertype
    ^
    1 error

The super keyword is used to access class members of an object's base class—i.e., \texttt{superclass} -- instead of the object's own class members. Without the use of the super keyword, the call to \texttt{printItem()} would refer to itself (a \texttt{recursive} call), so the method would call itself, and that call would call itself, etc., never actually printing anything (an error in this case).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image2.png}
\caption{Method calling overridden method of base class (i.e., superclass).}
\end{figure}

\begin{verbatim}
public class ProduceItem extends GenericItem {
    // Other methods ...

    @Override
    public void printItem() {
        super.printItem();
        System.out.println(" (Expires: " + expirationDate + ")");
        return;
    }

    // Other fields ...
}
\end{verbatim}
### 10.3.1: Override.

Assume myItem is defined and initialized as GenericItem, and myProduce as ProduceItem, with classes GenericItem and ProduceItem defined as above.

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Your answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>myItem.printItem()</code> calls the printItem() method for which class?</td>
<td>GenericItem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProduceItem</td>
</tr>
<tr>
<td>2</td>
<td><code>myProduce.printItem()</code> calls the printItem() method for which class?</td>
<td>GenericItem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProduceItem</td>
</tr>
<tr>
<td>3</td>
<td>Provide a statement within printItem() method of the the ProduceItem class to call the printItem() method of ProduceItem's base class.</td>
<td>printItem();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>@Override printItem();</td>
</tr>
<tr>
<td></td>
<td></td>
<td>super.printItem();</td>
</tr>
<tr>
<td>4</td>
<td>If ProduceItem did NOT have its own printItem() method defined, the printItem() method of which class would be called?</td>
<td>GenericItem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ProduceItem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A call to PrintItem() yields an error.</td>
</tr>
</tbody>
</table>
Section 10.4 - The Object class

Java's built-in **Object class** serves as the base class for all other classes and does not have a superclass—i.e., the Object class is located at the root of the Java class hierarchy. Thus, all classes, including user-defined classes, implement Object's methods. In the following discussion, note the subtle distinction between the term "Object class" and the generic term "object", which can refer to the instance of any class. Some common methods defined within the Object class are presented below. Refer to [Oracle's Java Object class specification](https://docs.oracle.com/javase/8/docs/api/index.html) for a more detailed description of all available methods.
- **toString()** -- Returns a String representation of the Object. By default, the toString() method returns a String containing the name of the class of which the object is an instance (e.g., the Object class) followed by the object's hexadecimal address in memory.

- **equals(otherObject)** -- Compares an Object to another otherObject and returns true if both variables reference the same object. Otherwise, the equals() method returns false. By default, the equals() method tests the equality of the two Object references, not the equality of their contents.

The following example illustrates the use of the toString() method with objects of various types, including a user-defined class that overrides the toString() method in order to represent a decimal integer in a numeral system of any base less than 10 (e.g., binary).

**Figure 10.4.1: Using the Object class's toString() method with various class types.**

```
public class IntegerWithBase {
    private int decimalValue;
    private int baseFormat;

    public IntegerWithBase(int inDecimal, int inBase) {
        this.decimalValue = inDecimal;
        this.baseFormat = inBase;
    }

    @Override
    public String toString() {
        int quotientVal = 0;
        int remainderVal = 0;
        int dividendVal = 0;
        String resultVal = "";

        dividendVal = decimalValue;

        if (baseFormat > 1) {
            // Loop iteratively determines each digit
            do {
                quotientVal = dividendVal / baseFormat;
                remainderVal = dividendVal % baseFormat;

                // Append remainder to the result as the new digit
                resultVal = remainderVal + resultVal;

                dividendVal = quotientVal;
            } while (quotientVal > 0);
        } else {
            resultVal = String.valueOf(decimalValue);
        }

        return resultVal;
    }
}
```

tempNum = 100
tempNum (base 4) = 1210
myObj = java.lang.Object@1148ab5c
The main() method creates three different objects (i.e., an Integer object, an IntegerWithBase object, and an Object object) and prints the String representation of each object to the console by calling toString(). The program’s output demonstrates the differences in implementation among the three objects’ toString() methods. While the Object class’s toString() method prints the object’s type followed by the object’s memory address, the built-in Integer class overrides toString() in order to print its internal integer value. Similarly, the IntegerWithBase class overrides toString() in order to print the integer value in a given numeral system. Note that although the above program explicitly invokes each object’s toString() method, the Java compiler allows the programmer to omit calls to toString() if the object is concatenated with a String or if the object is an argument to the println() or print() methods, which automatically invoke an argument’s toString() method. Thus, statements such as

```
System.out.println("tempNumInBase4 = " + tempNumInBase4);
```

are valid as well.

The IntegerWithBase class defines a constructor that allows the user to specify an integer’s decimal value and the base in which to represent the number when the program calls the toString() method. For example, the above statement

```
IntegerWithBase tempNumInBase4 = new IntegerWithBase(100, 4);  // Create an IntegerWithBase object
```

creates an IntegerWithBase object that can represent the integer 100 in the base-4 numeral system. The IntegerWithBase class overrides Object’s toString() method with an iterative algorithm that computes the digits in the new numeral system and returns the corresponding String. First, the toString() method initializes the variable called dividendVal to the original value of the integer (e.g., 100). Then, every iteration of the while loop performs integer division of the dividendVal by the baseFormat (e.g., 4). The resulting remainderVal becomes the next digit in the new numeral system representation and the quotientVal becomes the new dividendVal for the next iteration. The while loop terminates when the quotientVal becomes zero, and then the toString() method returns the resultVal.
Notice that the IntegerWithBase class does not handle base values greater than 10 appropriately. For example, creating the object

\[
\text{IntegerWithBase tempNumInBase16 = new IntegerWithBase(255,16);}
\]

in order print the value 255 in hexadecimal (base 16) results in the output "1515" as opposed to a value such as "FF". The problem lies with the range of characters used to represent a digit. One possible solution involves using alphabetical characters to represent digits with a value greater than nine.
10.4.1: Modifying IntegerWithBase to print bases greater than 10.

Define a new private method within IntegerWithBase called toAlphaNumDigit() that takes an integer value as an argument and returns a char representing the digit. For argument values between 0 and 9, the method should simply return the unicode value for that argument (i.e., a char value between 48 and 57). For argument values greater than or equal to 10, the method should return unicode values corresponding to a lower-case letter in the alphabet (i.e., a char value between 97 and 122). Thus, the statement `toAlphaNumDigit(15);` for example, should return the char value 102, which corresponds to the letter “f”.

Use this private method to convert the remainder values computed within toString() to the appropriate characters. For example, creating the object `IntegerWithBase tempNumInBase16 = new IntegerWithBase(255,16);` should output “ff”.

```java
public class IntegerWithBase {
    private int decimalValue;
    private int baseFormat;

    public IntegerWithBase(int inDecimal, int inBase) {
        this.decimalValue = inDecimal;
        this.baseFormat = inBase;
    }

    @Override
    public String toString() {
        int quotientVal = 0;
        int remainderVal = 0;
        int dividendVal = 0;
        String resultVal = "";
        dividendVal = decimalValue;
```
10.4.2: The Object class and overriding the toString() method.

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Your answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>User-defined classes are not derived from the Object class.</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>All classes can access Object's public and protected methods (e.g., toString() and equals()) even if such methods are not explicitly overridden.</td>
<td>True</td>
</tr>
<tr>
<td>3</td>
<td>The built-in Integer class overrides the toString() method in order to return a String representing an Integer's value.</td>
<td>True</td>
</tr>
<tr>
<td>4</td>
<td>The Object class's toString() method returns a String containing only the Object instance's type.</td>
<td>True</td>
</tr>
</tbody>
</table>

Exploring further:
- Oracle's Java Object class specification.
- Oracle's Java class hierarchy.

Section 10.5 - Polymorphism
**Polymorphism** refers to determining which program behavior to execute depending on data types. Method overloading is a form of **compile-time polymorphism** wherein the compiler determines which of several identically-named methods to call based on the method's arguments. Another form is **runtime polymorphism** wherein the compiler cannot make the determination but instead the determination is made while the program is running.

One scenario requiring runtime polymorphism involves derived classes. Commonly, a programmer wishes to create a collection of objects that combines base and derived class types, such as an ArrayList named inventoryList whose elements can each be a reference to an object of type GenericItem, ProduceItem, or FrozenFoodItem (the latter two types derived from GenericItem). Such an ArrayList can be initialized as

```
ArrayList<GenericItem> inventoryList = new ArrayList<GenericItem>();
```

and references to any of those objects may be added, as shown below.

---

**Figure 10.5.1: Runtime polymorphism.**

The JVM can dynamically determine the correct method to call based on the object's type.

---

**GenericItem.java:**

```java
public class GenericItem {
    public void setName(String newName) {
        itemName = newName;
        return;
    }

    public void setQuantity(int newQty) {
        itemQuantity = newQty;
        return;
    }

    public void printItem() {
        System.out.println(itemName + " " + itemQuantity);
        return;
    }

    protected String itemName;
    protected int itemQuantity;
}
```

**ProduceItem.java:**

```java
public class ProduceItem extends GenericItem { // ProduceItem derived from GenericItem
    public void setExpiration(String newDate) {
        expirationDate = newDate;
        return;
    }

    public String getExpiration() {
        return expirationDate;
    }

    @Override
}
```
The program uses a Java feature relating to **derived/base class reference conversion** wherein a reference to a derived class can be converted to a reference to the base class (without explicit

```java
public override void printItem() {
    System.out.println(itemName + " " + itemQuantity
        + " (Expires: " + expirationDate + ")");
    return;
}

private String expirationDate;
```

**ItemInventory.java:**

```java
import java.util.ArrayList;

public class ItemInventory {

    public static void main(String[] args) {
        GenericItem genericItem1;
        ProduceItem produceItem1;
        ArrayList<GenericItem> inventoryList = new ArrayList<GenericItem>(); // Colle
        int i = 0;

        genericItem1 = new GenericItem();
        genericItem1.setName("Smith Cereal");
        genericItem1.setQuantity(9);

        produceItem1 = new ProduceItem();
        produceItem1.setName("Apple");
        produceItem1.setQuantity(40);
        produceItem1.setExpiration("May 5, 2012");

        genericItem1.printItem();
        produceItem1.printItem();

        // More common: Collection (e.g., ArrayList) of objs
        // Polymorphism -- Correct printItem() called
        inventoryList.add(genericItem1);
        inventoryList.add(produceItem1);
        System.out.println("\nInventory: ");
        for (i = 0; i < inventoryList.size(); ++i) {
            inventoryList.get(i).printItem(); // Calls correct printItem()
        }
        return;
    }
}
```

Smith Cereal 9
Apple 40 (Expires: May 5, 2012)

Inventory:
Smith Cereal 9
Apple 40 (Expires: May 5, 2012)
casting). Such conversion is in contrast to other data type conversions, such as converting a double to an int (which is an error unless explicitly cast). Thus, the above statement

```java
inventoryList.add(produceItem1);
```

uses this feature, with a ProduceItem reference being converted to a GenericItem reference (inventoryList is an ArrayList of GenericItem references). The conversion is intuitive; recall in an earlier animation that a derived class like ProductItem consists of the base class GenericItem plus additional members, so the conversion yields a reference to the base class part (so really there's no change).

However, an interesting question arises when printing the ArrayList's contents. For a given element, how does the program know whether to call GenericItem's printItem() or ProduceItem's printItem()? The Java virtual machine automatically performs runtime polymorphism, i.e., it dynamically determines the correct method to call based on the actual object type to which the variable (or element) refers.

<table>
<thead>
<tr>
<th>Participation Activity</th>
<th>10.5.1: Polymorphism.</th>
</tr>
</thead>
</table>

Consider the GenericItem and ProduceItem classes defined above.

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Your answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>An item of type <code>ProduceItem</code> may be added to an <code>ArrayList</code> of type</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td><code>ArrayList&lt;GenericItem&gt;</code></td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>The JVM automatically performs runtime polymorphism to determine the</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>correct method to call</td>
<td>False</td>
</tr>
</tbody>
</table>

Exploring further:

- [More on Polymorphism](https://docs.oracle.com/javase/tutorial/java/IandI/poly.html) from Oracle's Java tutorials
- [More on abstract classes and methods](https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html) from Oracle's Java tutorials
10.5.1: Basic polymorphism.

Write the printItem() method for the base class. Sample output for below program:

Last name: Smith
First and last name: Bill Jones

```java
class BaseItem {
    String firstName;
    String lastName;

    public BaseItem() {
        firstName = null;
        lastName = null;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public void printItem() {
        if (firstName == null)
            System.out.println(lastName);
        else
            System.out.println(firstName + " "+ lastName);
    }
}

class DerivedItem extends BaseItem {
    String base;

    public DerivedItem() {
        super(); // Use the constructor of the base class
    }

    public void setBase(String base) {
        this.base = base;
    }

    public void printItem() {
        super.printItem();
        System.out.println(base);
    }
}

public class BaseItemPtr {
    public static void main(String[] args) {
        BaseItemPtr itemList = new BaseItemPtr();
        BaseItem baseItem = new BaseItem();
        baseItem.setLastName("Smith");
        itemList.add(baseItem);

        BaseItem baseItem2 = new BaseItem();
        baseItem2.setLastName("Jones");
        itemList.add(baseItem2);

        DerivedItem derivedItem = new DerivedItem();
        derivedItem.setBase("Bill");
        derivedItem.setBase("Bill");
        itemList.add(derivedItem);

        for (int i = 0; i < itemList.size(); ++i) {
            itemList.get(i).printItem();
        }
        return;
    }
}
```

Section 10.6 - ArrayLists of Objects

Because all classes are derived from the Object class, programmers can take advantage of runtime polymorphism in order to create a collection (e.g., ArrayList) of objects of various class types and perform operations on the elements. The following program adds objects of seemingly differing types (e.g., Object, Integer, IntegerWithBase, Double, and String) into a single ArrayList and prints the contents.

Figure 10.6.1: Printing an ArrayList of Object elements
IntegerWithBase.java:

```java
public class IntegerWithBase {
    private int decimalValue;
    private int baseFormat;

    public IntegerWithBase(int inDecimal, int inBase) {
        this.decimalValue = inDecimal;
        this.baseFormat = inBase;
    }

    @Override
    public String toString() {
        int quotientVal = 0;
        int remainderVal = 0;
        int dividendVal = 0;
        String resultVal = "";

        dividendVal = decimalValue;

        if (baseFormat > 1) {
            // Loop iteratively determines each digit
            do {
                quotientVal = dividendVal / baseFormat;
                remainderVal = dividendVal % baseFormat;

                // Append remainder to the result as the new digit
                resultVal = remainderVal + resultVal;
                dividendVal = quotientVal;
            } while (quotientVal > 0);
        } else {
            resultVal = String.valueOf(decimalValue);
        }

        return resultVal;
    }
}
```

ArrayPrinter.java:

```java
import java.util.ArrayList;

public class ArrayPrinter {
    // Method prints an ArrayList of Objects
    public static void PrintArrayList(ArrayList<Object> objList) {
        int i = 0;

        for (i = 0; i < objList.size(); ++i) {
            System.out.println(objList.get(i));
        }

        return;
    }

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

12

1010

3.14

Hello!
The statement `ArrayList<Object> objList = new ArrayList<Object>();` initializes an `ArrayList` of `Object` elements used to store different objects. The program then adds five new objects of various class types to the `ArrayList` and prints the contents of the `ArrayList`. Adding an object of a type derived from `Object` (e.g., `Double`) into an `ArrayList` of `Object` elements is possible due to Java's automatic conversion of derived class references to base class references. Thus, a statement such as `objList.add(new Double(3.14));` converts the reference to the new `Double` object into an `Object` reference.

The `PrintArrayList()` method takes an `ArrayList` of `Objects` as an argument, iterates through every element of the `ArrayList`, and prints the String representation of each element using the `toString()` method. Runtime polymorphism enables the Java virtual machine to dynamically determine the correct version of `toString()` to call based on the actual class type of each element. Notice that the statement `System.out.println(objList.get(i));` does not need to explicitly call each element’s `toString()` method because each element is concatenated with a String literal.

Finally, note that a method operating on a collection of `Object` elements may only invoke the methods declared by the base class (e.g., the `Object` class). Thus, a statement that calls the `toString()` method on an element of an `ArrayList` of `Objects` called `objList`, such as `objList.get(i).toString()`, is valid because the `Object` class defines the `toString()` method. However, a statement that calls, for example, the `Integer` class’s `intValue()` method on the same element (i.e., `objList.get(i).intValue()`) results in a compiler error even if that particular element is an `Integer` object.
### 10.6.1: ArrayLists of Object elements and runtime polymorphism principles.

Consider the `IntegerWithBase` and `ArrayPrinter` classes defined above.

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Your answer</th>
</tr>
</thead>
</table>
| 1  | An item of any class type may be added to an `ArrayList` of type `ArrayList<Object>`.
|    |                                                                         | Yes         |
|    |                                                                         | No          |
| 2  | Assume that an `ArrayList` of type `ArrayList<Object>` called `myList` contains only three elements of type `Double`. Is the statement `myList.get(0).doubleValue();` valid?  
  Note that the method `doubleValue()` is defined in the `Double` class but not the `Object` class. | Yes         |
|    |                                                                         | No          |
| 3  | The above program's `PrintArrayList()` method can dynamically determine which implementation of `toString()` to call. | Yes         |
|    |                                                                         | No          |

Exploring further:
- [Oracle's Java Object class specification](https://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.2.2)
- [More on Polymorphism](https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html) from Oracle's Java tutorials

### Section 10.7 - Is-a versus has-a relationships
The concept of inheritance is commonly confused with the idea of composition. Composition is the idea that one object may be made up of other objects, such as a MotherInfo class being made up of objects like firstName (which may be a String object), childrenData (which may be an ArrayList of ChildInfo objects), etc. Defining that MotherInfo class does not involve inheritance, but rather just composing the sub-objects in the class.

In contrast, a programmer may note that a mother is a kind of person, and all persons have a name and birthdate. So the programmer may decide to better organize the program by defining a PersonInfo class, and then by creating the MotherInfo class derived from PersonInfo, and likewise for the ChildInfo class.

```java
public class ChildInfo {
    public String firstName;
    public String birthDate;
    public String schoolName;

    ...
}

public class MotherInfo {
    public String firstName;
    public String birthDate;
    public String spouseName;
    public ArrayList<ChildInfo> childrenData;

    ...
}
```
**Figure 10.7.2: Inheritance.**

The 'is-a' relationship. A MotherInfo object 'is a' kind of PersonInfo. The MotherInfo class thus inherits from the PersonInfo class. Likewise for the ChildInfo class.

```java
public class PersonInfo {
    public String firstName;
    public String birthdate;

    ...
}

public class ChildInfo extends PersonInfo {
    public String schoolName;

    ...
}

public class MotherInfo extends PersonInfo {
    public String spousename;
    public ArrayList<ChildInfo> childrenData;

    ...
}
```

### 10.7.1: Is-a vs. has-a relationships.

Indicate whether the relationship of the everyday items is an is-a or has-a relationship. Derived classes and inheritance are related to is-a relationships, not has-a relationships.

<table>
<thead>
<tr>
<th>#</th>
<th>Question</th>
<th>Your answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fruit / apple</td>
<td>Is-a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Has-a</td>
</tr>
<tr>
<td>2</td>
<td>House / window</td>
<td>Is-a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Has-a</td>
</tr>
</tbody>
</table>
Section 10.8 - Java example: Employees and overriding class methods

10.8.1: Inheritance: Employees and overriding a class method.

The classes below describe a superclass named EmployeePerson and two derived classes, EmployeeManager and EmployeeStaff, each of which extends the EmployeePerson class. The main program creates objects of type EmployeeManager and EmployeeStaff and prints those objects.

1. Run the program, which prints manager data only using the EmployeePerson class' printInfo method.

2. Modify the EmployeeStaff class to override the EmployeePerson class' printInfo method and print all the fields from the EmployeeStaff class. Run the program again and verify the output includes the manager and staff information.

3. Modify the EmployeeManager class to override the EmployeePerson class' printInfo method and print all the fields from the EmployeeManager class. Run the program again and verify the manager and staff information is the same.

```
public class EmployeeMain {
    public static void main(String [] args) {
        // Create the objects
        EmployeeManager manager = new EmployeeManager(25);
        EmployeeStaff staff1 = new EmployeeStaff("Michele");

        // Load data into the objects using the Person class' method
        manager.setData("Michele", "Sales", "03-03-1975", 70000);
        staff1.setData("Bob", "Sales", "02-02-1980", 50000);

        // Print the objects
        manager.printInfo();
        staff1.printInfo();
        return;
    }
}
```
10.8.2: Employees and overriding a class method (solution).

Below is the solution to the problem of overriding the EmployeePerson class' printInfo() method in the EmployeeManager and EmployeeStaff classes. Note that the Main and Person classes are unchanged.

```java
public class EmployeeMain {

    public static void main(String[] args) {

        // Create the objects
        EmployeeManager manager = new EmployeeManager(25);
        EmployeeStaff staff1 = new EmployeeStaff("Michele");

        // Load data into the objects using the Person class' method
        manager.setData("Michele", "Sales", "03-03-1975", 70000);
        staff1.setData("Bob", "Sales", "02-02-1980", 50000);

        // Print the objects
        manager.printInfo();
        staff1.printInfo();
    }
}
```

Run