Print Copy

Print functionality is for user convenience only, not all interactive content is fully printable.

Print Cancel
Rendering chapter...

©zyBooks 05/10/21 13:25 728163
Neha Maddali
IASTATECOMS228Spring2021

23.1 Derived classes

Derived class concept

Commonly, one class is similar to another class but with some additions or variations. Ex: A store inventory system might use a class called GenericItem that has itemName and itemQuantity data members. But for produce (fruits and vegetables), a ProduceItem class with data members itemName, itemQuantity, and expirationDate may be desired.

participation activity
23.1.1: Creating a ProduceItem from GenericItem.

Animation content:

undefined

Animation captions:

1. A GenericItem has data members itemName and itemQuantity and 3 member functions.
2. A ProduceItem is very similar to a GenericItem, but a ProduceItem also needs an expirationDate data member.
3. A ProduceItem needs the same data member functions as a GenericItem plus functions to get and set the expiration date.
4. If ProduceItem is implemented as an independent class, all the data/function members from GenericItem must be copied into ProduceItem, creating lots of duplicate code.
5. If ProduceItem is implemented as a derived class, ProduceItem need only implement what is different between a GenericItem and ProduceItem.

participation activity
23.1.2: Derived class concept.
1)
Creating an independent class that has the same members as an existing class creates duplicate code.

- ○ True
- ○ False

2)
Creating a derived class is generally less work than creating an independent class.

- ○ True
- ○ False

Inheritance

A derived class (or subclass) is a class that is derived from another class, called a base class (or superclass). Any class may serve as a base class. The derived class is said to inherit the properties of the base class, a concept called inheritance. An object declared 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.

A derived class is declared by placing the keyword extends after the derived class name, followed by the base class name. Ex: `class DerivedClass extends BaseClass { ... }`. The figure below defines the base class GenericItem and derived class ProduceItem that inherits from GenericItem.

Figure 23.1.1: Class ProduceItem is derived from class GenericItem.

GenericItem.java

```
public class GenericItem {
   private String itemName;
   private int itemQuantity;

   public void setName(String newName) {
      itemName = newName;
```

```
    }

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

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

ProduceItem.java

```
public class ProduceItem extends GenericItem {
   private String expirationDate;

   public void setExpiration(String newDate) {
      expirationDate = newDate;
   }

   public String getExpiration() {
      return expirationDate;
   }
}
```

participation activity
23.1.3: Using GenericItem and ProduceItem objects.

Animation content:

undefined

Animation captions:

1. miscItem is a GenericItem, which has 2 private fields and 3 public methods.
2. Since ProduceItem is derived from GenericItem, ProduceItem inherits GenericItem's class members and adds new members implemented in ProduceItem.
3. miscItem's name and quantity are set, and miscItem is printed to the screen.
4. perishItem's name, quantity, and expiration are set. Then perishItem is printed to the screen.

participation activity
23.1.4: Derived classes.
1)
A class that can serve as the basis for another class is called a _____ class.

Check Show answer
2)
In the figure above, how many total class members does GenericItem contain?

Check Show answer
3)
In the figure above, how many total class members are unique to ProduceItem?

Check Show answer
4)
Class Dwelling has fields door1, door2, door3. A class House is derived from Dwelling and has fields wVal, xVal, yVal, zVal. How many fields does an initialization `House h = new House();` create?

Check Show answer

Inheritance scenarios

Various inheritance variations are possible:

- A derived class can serve as a base class for another class. Ex: `class FruitItem extends ProduceItem {...}` creates a derived class FruitItem from ProduceItem, which was derived from GenericItem.

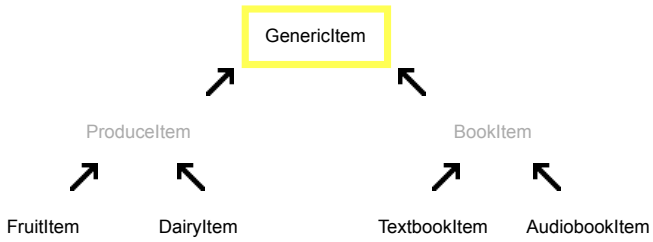
- A class can serve as a base class for multiple derived classes. Ex: `class FrozenFoodItem extends GenericItem {...}` creates a derived class FrozenFoodItem that inherits from GenericItem, just as ProduceItem inherits from GenericItem.

A class can only be derived from one base class directly. Ex: Inheriting from two classes as in `class House extends Dwelling, Property {...}` results in a compiler error.

participation activity
23.1.5: Interactive inheritance tree.

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

Inheritance tree



Selected class pseudocode

```
private:
    private String itemName;
    private int itemQuantity;

public:
    public void setName(String newName)
    public void setQuantity(int newQty)
    public void printItem()
```

Selected class code

```
public class GenericItem {
   private String itemName;
   private int itemQuantity;

   public void setName(String newName) {
      itemName = newName;
   }

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

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

participation activity
23.1.6: Inheritance scenarios.
Refer to the interactive inheritance tree above.

1)
The BookItem class acts as a derived class and a base class.

- ○ True
- ○ False

2)
ProduceItem and BookItem share some of the same class members.

- ○ True
- ○ False

3)
DairyItem and TextbookItem share some of the same class members.

- ○ True
- ○ False

4)
AudiobookItem inherits the field called readerName from BookItem.

- ○ True
- ○ False

5)
AudiobookItem inherits the method getTitle() from BookItem.

- ○ True
- ○ False

Example: Business and Restaurant

The example below defines a Business class with private fields name and address. The Restaurant class is derived from Business and adds a rating private field with a getter and setter.

Figure 23.1.2: Inheritance example: Business and Restaurant classes.

Business.java

```
public class Business {
   private String name;
   private String address;

   public void setName(String busName) {
      name = busName;
   }

   public void setAddress(String busAddress) {
      address = busAddress;
   }

   public String getDescription() {
      return name + " -- " + address;
   }
}
```

Restaurant.java

```
public class Restaurant extends Business {
   private int rating;

   public void setRating(int userRating) {
      rating = userRating;
   }

   public int getRating() {
      return rating;
   }
}
```

InheritanceExample.java

```
public class InheritanceExample {
   public static void main(String[] args) {
      Business someBusiness = new Business();
      Restaurant favoritePlace = new Restaurant();

      someBusiness.setName("ACME");
      someBusiness.setAddress("4 Main St");

      favoritePlace.setName("Friends Cafe");
      favoritePlace.setAddress("500 W 2nd Ave");
      favoritePlace.setRating(5);

      System.out.println(someBusiness.getDescription());
      System.out.println(favoritePlace.getDescription());
      System.out.println("  Rating: " + favoritePlace.getRating());
   }
}
```

```
ACME -- 4 Main St
Friends Cafe -- 500 W 2nd Ave
  Rating: 5
```

participation activity
23.1.7: Inheritance example.
Refer to the code above.
1)
How many methods are defined in Restaurant?

- ○ 2
- ○ 3
- ○ 5

2)
How many methods can a Restaurant object call?

- ○ 2

○ 3
○ 5

3)
Which method call produces a syntax error?

○ `someBusiness.setRating(4);`
○ `favoritePlace.getRating();`
○ `favoritePlace.setRating(4);`

4)
What is the best way to declare a new DepartmentStore class?

○ `class DepartmentStore { ... }`
○ `class DepartmentStore extends Restaurant { ... }`
○ `class DepartmentStore extends Business { ... }`

Exploring further:

- Oracle's Java tutorials on inheritance.

challenge activity
23.1.1: Derived classes.
Start

Type the program's output

VehicleDerivation.java | Vehicle.java | Car.java

```
public class VehicleDerivation { public static
void main(String[] args) { Car myCar = new
Car(); myCar.setSpeed(50);
myCar.printCarSpeed(); } }
```

```
Moving at: 50
```

1 2 3

Check Next

challenge activity
23.1.2: Basic inheritance.
Assign courseStudent's name with Smith, age with 20, and ID with 9999. Use the printAll() member method and a separate println() statement to output courseStudents's data. Sample output from the given program:

```
Name: Smith, Age: 20, ID: 9999
```

Run

View your last submission

23.2 Access by members of derived classes

Member access

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 Restaurant class yields a compiler error.

Figure 23.2.1: Member methods of a derived class cannot access private members of the base class.

```
public class Business {
   private String name;
   private String address;

   ...

}
```

```
public class Restaurant extends Business {
   private int rating;

   ...

   public void displayRestaurant() {
      System.out.println(name);
      System.out.println(address);
      System.out.println("Rating: "+ rating);
   }
   ...
}
```

```
$ javac Restaurant.java
Restaurant.java:12: name has private access in Business
      System.out.println(name);
                         ^
Restaurant.java:12: address has private access in Business
      System.out.println(address);
                         ^
2 errors
```

participation activity
23.2.1: Access by derived class members.

Assume `public class Restaurant extends Business{...}`

1)
Business's public member method can be called by a member method of Restaurant.

- ○ True
- ○ False

2)
Restaurant's private fields can be accessed by Business.

- ○ True
- ○ False

Protected member access

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.

In the following example, the member called name is specified as protected and is accessible anywhere in the derived class. Note that the name 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.

Figure 23.2.2: Access specifiers—Protected allows access by derived classes and classes in the same package but not by others.
Code contains intended errors to demonstrate protected accesses.

Business.java:

```
public class Business{
   protected String name;    // Member accessible by self and derived classes
   private String address;   // Member accessible only by self

   public void printMembers() { // Member accessible by anyone
      // Print information ...
   }

}
```

Restaurant.java:

```
public class Restaurant extends Business{
   private int rating;

   public void displayRestaurant() {
      // Attempted accesses
      printMembers();             // OK
      name = "Gyro Hero";         // OK    ("protected" above made this possible)
      address = "5 Fifth St";     // ERROR
   }
```

```
   // Other class members ...
}
```

InheritanceAccessEx.java

```
public class InheritanceAccessEx {
   public static void main(String[] args) {
      Business business = new Business();
      Restaurant restaurant = new Restaurant();

      // Attempted accesses
      business.printMembers();          // OK
      business.name = "Gyro Hero";      // OK (protected also applies to other classes in the same package)
      business.address = "5 Fifth St";  // ERROR

      restaurant.printMembers();        // OK
      restaurant.name = "Gyro Hero";    // OK (protected also applies to other classes in the same package)
      restaurant.rating = 5; // ERROR

      // Other instructions ...
   }
}
```

To make Restaurant's displayRestaurant() method work, we merely need to change the private members to protected members in class Business. Business's class members name and address thus become accessible to a derived class like Restaurant. 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 23.2.1: Access specifiers for class members.

Specifier	Description
private	Accessible by self.
protected	Accessible by self, derived classes, and other classes in the same package.
public	Accessible by self, derived classes, and everyone else.
no specifier	Accessible by self and other classes in the same package.

participation activity
23.2.2: Protected access specifier.

Assume `public class Restaurant extends Business{...}`.

Suppose a new class, `public class SkateShop{...}`, is defined in the same package as Business.

1)
Business's protected fields can be accessed by a member method of Restaurant.

- ○ True
- ○ False

2)
Business's protected fields can be accessed by a member method of SkateShop.

- ○ True
- ○ False

Class definitions

Separately, the keyword "public" in a class definition 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
23.2.3: Access specifiers for class definitions.

Suppose a new class, `RestaurantReview`, is defined in a separate package. For the following cases, which specifier, if any, should be used for `_____ class Restaurant{...}`?

1)
Restaurant can be accessed by RestaurantReview.

- ○ no specifier
- ○ public
- ○ protected

2)

Restaurant cannot be accessed by RestaurantReview.

- ○ no specifier
- ○ public
- ○ protected

Exploring further:

- More on access specifiers from Oracle's Java tutorials

23.3 Overriding member methods

Overriding

When a derived class defines a member method that has the same name and parameters as a base class's method, the member method is said to override the base class's method. The example below shows how the Restaurant's getDescription() method overrides the Business's getDescription() method.

The @Override annotation is placed above a method that overrides a base class method so the compiler verifies that an identical base class method exists. An annotation is an optional command beginning with the "@" symbol that can provide the compiler with information that helps the compiler detect errors better. The @Override annotation causes 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 or misnames the overriding method. Good practice is to always include an @Override annotation with a method that is meant to override a base class method.

participation activity
23.3.1: Overriding member method example.

Animation content:

undefined

Animation captions:

1. The Business class defines a getDescription() method that returns a string with the business name and address.
2. Restaurant derives from Business and uses the @Override annotation above a member method that has the same name, parameters, and return type as the base class method getDescription().
3. The Restaurant object favoritePlace calls the Restaurant's getDescription(), which overrides the base class's getDescription().

Overriding vs. overloading

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 a 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.

participation activity
23.3.2: Overriding.
Refer to the code above.
1)
If a Restaurant object calls getDescription(), the Restaurant's getDescription() is called instead of Business's getDescription().

- ○ True
- ○ False

2)
If a Business object calls getDescription(), the Restaurant's getDescription() is called instead of Business's getDescription().

- ○ True
- ○ False

3)
Removing Business's getDescription() method in the example above causes a syntax error.

- ○ True
- ○ False

4)
Changing Restaurant's `String getDescription()` to `String getDescription(int num)` causes a syntax error.

- ○ True
- ○ False

5)
Changing Business's name and address data members from protected to private in the example above causes a syntax error.

- ○ True
- ○ False

Calling a base class method

An overriding method can call the overridden method by using the super keyword. Ex: `super.getDescription()`. The super keyword is a reference variable used to call the parent class's methods or constructors.

Figure 23.3.1: Method calling overridden method of base class.

```
public class Restaurant extends Business{
   ...

   @Override
   public String getDescription() {
      return super.getDescription() + "\n  Rating: " + rating;
   }

   ...
}
```

A common error is to leave off super when wanting to call a base class method. Without the use of the super keyword, the call to getDescription() refers to itself (a *recursive* call), so getDescription() would call itself, which would call itself, etc., never actually printing anything.

participation activity
23.3.3: Override example.

Choose the correct replacement for the missing code below so ProduceItem's printItem() overrides GenericItem's printItem().

GenericItem.java

```
public class GenericItem {
   __(A)__: String itemName;
   __(A)__: int itemQuantity;
   ...

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

ProduceItem.java

```
public class ProduceItem extends GenericItem {
   private String expirationDate;

   public void setExpiration(String newDate) {
      expirationDate = newDate;
   }

   public String getExpiration() {
      return expirationDate;
   }

   @Override
   public __B()__ {
      __(C)__;
      System.out.println("  (expires " + expirationDate + ")");
   }
}
```

1)
(A)

- ○ private
- ○ public

2)
(B)

- ○ void printItem(int itemNumber)
- ○ void printItem()
- ○ String printItem()

3)
(C)

- ○ printItem()
- ○ printItem(super)
- ○ super.printItem()

challenge activity
23.3.1: Overriding member methods.

Start

Type the program's output

ClassOverridingEx.java | Computer.java | Laptop.java

```
public class ClassOverridingEx { public static void main(String[] args) { Laptop
myLaptop = new Laptop(); myLaptop.setComputerStatus("30%", "connected");
myLaptop.setWiFiStatus("bad"); myLaptop.printStatus(); } }
```

```
WiFi: bad
CPU: 30%
```

1 2 3

Check Next

challenge activity
23.3.2: Basic derived class member override.
Define a method printAll() for class PetData that prints output as follows with inputs "Fluffy", 5, and 4444. Hint: Make use of the base class' printAll() method.

```
Name: Fluffy, Age: 5, ID: 4444
```

Run

View your last submission

23.4 The Object class

The Object class

The built-in Object class serves as the base class for all other classes and does not have a base class. All classes, including user-defined classes, are derived from Object and 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. Two common methods defined within the Object class are toString() and equals().

- The toString() method returns a String representation of the Object. By default, toString() returns a String containing the object's class name followed by the object's hash code in hexadecimal form. Ex: java.lang.Object@372f7a8d.
- The equals(otherObject) method compares an Object to otherObject and returns true if both variables reference the same object. Otherwise, equals() returns false. By default, equals() tests the equality of the two Object references, not the equality of the Objects' contents.

Figure 23.4.1: Business class is derived from Object.

```
public class Business {
   protected String name;
   protected String address;

   void setName(String busName) {
      name = busName;
   }

   void setAddress(String busAddress) {
      address = busAddress;
   }

   String getDescription() {
      return name + " -- " + address;
   }
}
```

participation activity
23.4.1: Behavior of toString() method.

Animation content:

```
undefined
```

Animation captions:

1. The Object class is the base class for all other classes.

2. Java's Integer class overrides the Object's toString() method, but the user-defined Business class does not.
3. Object's toString() returns a string consisting of the object's class name (java.lang.Object), the '@' character, and an unsigned hexadecimal representation of the object's hash code (1148ab5c).
4. Integer's toString() method returns the object's associated integer value.
5. Business does not override toString() so Object's toString() method is called and outputs the object's class name (Business), the '@' character, and an unsigned hexadecimal representation of the object's hash code (19469ea2).

participation activity
23.4.2: The Object class and overriding the toString() method.
1)
User-defined classes are not derived from the Object class.

- ○ True
- ○ False

2)
All classes can access Object's public and protected methods like toString() and equals(), even if the methods are not explicitly overridden.

- ○ True
- ○ False

3)
The built-in Integer class overrides the toString() method in order to return a String representing an Integer's value.

- ○ True
- ○ False

4)
The Object class's toString() method returns a String containing only the Object instance's type.

- ○ True
- ○ False

Overriding toString() in the base class

The figure below shows a Business class that overrides Object's toString() method and returns a String containing the business name and address. The Restaurant class derives from Business but does not override toString(). So when a Restaurant object's toString() method is called, the Business class's toString() method executes.

Figure 23.4.2: Base class Business overrides toString().

Business.java

```
public class Business {
   protected String name;
   protected String address;

   void setName(String busName) {
      name = busName;
   }

   void setAddress(String busAddress) {
      address = busAddress;
   }

   @Override
   public String toString() {
      return name + " -- " + address;
   }
}
```

Restaurant.java

```
public class Restaurant extends Business {
   private int rating;

   public void setRating(int userRating) {
      rating = userRating;
   }

   public int getRating() {
      return rating;
   }
}
```

The toString() method is called automatically by the compiler when an object is concatenated to a string or when print() or println() is called. Ex: `System.out.println(someObj)` calls `someObj.toString()` automatically.

participation activity

23.4.3: toString() in base class only.

Refer to the code below to determine what each code snippet outputs.

```
Business aaaBus = new Business();
Restaurant tacoRest = new Restaurant();

aaaBus.setName("AAA Business");
aaaBus.setAddress("5 Race St");

tacoRest.setName("Tom's Tacos");
tacoRest.setAddress("600 Pleasure Ave");
tacoRest.setRating(5);
```

1)

```
System.out.println(aaaBus.toString());
```

- ○ Business@372f7a8d
- ○ AAA Business -- 5 Race St
- ○ Tom's Tacos -- 600 Pleasure Ave

2)

```
System.out.println(tacoRest);
```

- ○ Restaurant@372f7a8d
- ○ AAA Business -- 5 Race St
- ○ Tom's Tacos -- 600 Pleasure Ave

3)

```
String somePlace = aaaBus + " #2";
System.out.println(somePlace);
```

- ○ Syntax error
- ○ AAA Business -- 5 Race St
- ○ AAA Business -- 5 Race St #2

4)

```
String somePlace = aaaBus;
System.out.println(somePlace);
```

- ○ Syntax error
- ○ AAA Business
- ○ AAA Business -- 5 Race St

Overriding toString() in the derived class

Both the base class Business and derived class Restaurant override toString() in the figure below. The Restaurant toString() uses the super keyword to call the base class toString() to get a string with the business name and address. Then toString() concatenates the rating and returns a string containing the name, address, and rating.

Figure 23.4.3: Derived class Restaurant overrides toString().

Business.java

```
public class Business {
   protected String name;
   protected String address;

   void setName(String busName) {
      name = busName;
   }

   void setAddress(String busAddress) {
      address = busAddress;
   }

   @Override
   public String toString() {
      return name + " -- " + address;
   }
}
```

Restaurant.java

```
public class Restaurant extends Business {
   private int rating;

   public void setRating(int userRating) {
      rating = userRating;
   }

   public int getRating() {
      return rating;
   }

   @Override
   public String toString() {
      return super.toString() + ", Rating: " + rating;
   }
}
```

participation activity
23.4.4: toString() in base and derived classes().

Refer to the code below to determine what each code snippet outputs.

```
Business aaaBus = new Business();
Restaurant tacoRest = new Restaurant();

aaaBus.setName("AAA Business");
aaaBus.setAddress("5 Race St");

tacoRest.setName("Tom's Tacos");
tacoRest.setAddress("600 Pleasure Ave");
tacoRest.setRating(5);
```

1)

```
System.out.println(aaaBus.toString());
```

- ○ Business@372f7a8d
- ○ AAA Business -- 5 Race St
- ○ Tom's Tacos -- 600 Pleasure Ave

2)

```
System.out.println(tacoRest);
```

- ○ Restaurant@372f7a8d
- ○ Tom's Tacos -- 600 Pleasure Ave
- ○ Tom's Tacos -- 600 Pleasure Ave, Rating: 5

3)

```
String somePlace = aaaBus + " &\n" + tacoRest;
System.out.println(somePlace);
```

- ○ Syntax error
- ○ AAA Business -- 5 Race St &
  Tom's Tacos -- 600 Pleasure Ave, Rating: 5
- ○ AAA Business &
  Tom's Tacos

Exploring further:

- Oracle's Java Object class specification.
- Oracle's Java class hierarchy.

challenge activity
23.4.1: Enter the output of the println() statements.
Start

Type the program's output

CallCar.java | Car.java

public class CallCar { public static void main(String[] args) { Car car1 = new Car(); Car car2 = new Car(); car1.setBrand("Nissan"); car1.setModel("Rogue"); car2.setBrand("Audi"); car2.setModel("A4"); System.out.println(car2); System.out.println(car1); } }

```
Audi A4
Nissan Rogue
```

1 2

Check Next

23.5 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. Programmers commonly create a collection of objects of both base and derived class types. Ex: the statement `ArrayList<GenericItem> inventoryList = new ArrayList<GenericItem>();` declares an ArrayList that can contain references to objects of type GenericItem or ProduceItem. ProduceItem derives from GenericItem.

Figure 23.5.1: Runtime polymorphism.
The JVM can dynamically determine the correct method to call based on the object's type.

GenericItem.java:

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

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

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

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

ProduceItem.java:

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

   public String getExpiration() {
      return expirationDate;
   }

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

   private String expirationDate;
}
```

ItemInventory.java:

```
import java.util.ArrayList;

public class ItemInventory {
   public static void main(String[] args) {
      GenericItem genericItem1;
      ProduceItem produceItem1;
```

```
      ArrayList<GenericItem> inventoryList = new ArrayList<GenericItem>(); // Collection of "Items"
      int i;                                                               // Loop index

      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()
      }
   }
}
```

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

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

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 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 `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 ProduceItem 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.

participation activity
23.5.1: Polymorphism.

Consider the GenericItem and ProduceItem classes defined above.

1)
An item of type `ProduceItem` may be added to an ArrayList of type `ArrayList<GenericItem>`.

- ○ True
- ○ False

2)
The JVM automatically performs runtime polymorphism to determine the correct method to call.

- ○ True
- ○ False

Exploring further:

- More on Polymorphism from Oracle's Java tutorials

challenge activity
23.5.1: Polymorphism and method overloading.

Start

Type the program's output

CallWatch.java	Watch.java	SmartWatch.java

import java.util.ArrayList; public class CallWatch { public static void main(String[] args) { SmartWatch watch1; Watch watch2; SmartWatch watch3; ArrayList<Watch> watchList = new ArrayList<Watch>(); int i; watch1 = new SmartWatch(); watch1.setHours(10); watch1.setMins(32); watch1.setPercentage(10); watch2 = new Watch(); watch2.setHours(17); watch2.setMins(39); watch3 = new SmartWatch(); watch3.setHours(9); watch3.setMins(18); watch3.setPercentage(100); watchList.add(watch1); watchList.add(watch2); watchList.add(watch3); for(i = 0; i < watchList.size(); ++i) { watchList.get(i).printItem(); } } }

```
10:32 10%
17:39
9:18 100%
```

1 2

Check Next

challenge activity
23.5.2: Basic polymorphism.
Write the printItem() method for the base class. Sample output for below program:

```
Last name: Smith
First and last name: Bill Jones
```

Run

23.6 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 program below uses the Business class and other built-in classes to create and output a single ArrayList of differing types.

Figure 23.6.1: Business.java.

```
public class Business {
   protected String name;
   protected String address;

   public Business() {}

   public Business(String busName, String busAddress) {
      name = busName;
      address = busAddress;
   }

   @Override
   public String toString() {
      return name + " -- " + address;
   }
}
```

participation activity
23.6.1: ArrayList of Objects.

Animation content:

undefined

Animation captions:

1. objList is an ArrayList of Object elements. All objects derive from Object, so objList can store any type of object.
2. Five new objects of various class types are added to the ArrayList. Each derived class reference is automatically converted to a base class (Object) reference.
3. PrintArrayList() takes an ArrayList of Objects as an argument and outputs every element of the ArrayList.
4. get(i) returns each Object element. Runtime polymorphism allows the correct version of toString() to be called based on the actual class type of each element.

Note that a method operating on a collection of Object elements may only invoke the methods defined 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.

participation activity
23.6.2: ArrayLists of Object elements and runtime polymorphism principles.
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

challenge activity
23.6.1: Enter the output of the ArrayList of Objects.
Start

Type the program's output

ArrayListManager.java | Business.java | Coffee.java

```
import java.util.ArrayList; public class ArrayListManager { public static void printArrayList(ArrayList<Object> objList) { int i; for (i = 0; i < objList.size(); ++i) { System.out.println(objList.get(i)); } } public static void main(String[] args) { ArrayList<Object> objList = new ArrayList<Object>(); String myString = new String("Echo"); Coffee myCoffee = new Coffee("Latte", "French"); Business myBusiness = new Business("Dinoco", "5 Main St"); objList.add(myString); objList.add(myCoffee); objList.add(myBusiness); printArrayList(objList); } }
```

```
Echo
Latte, French
Dinoco -- 5 Main St
```

1

Check Try again

Exploring further:

- Oracle's Java Object class specification.
- More on Polymorphism from Oracle's Java tutorials

23.7 Abstract classes: Introduction (generic)

Abstract classes

Object-oriented programming (OOP) is a powerful programming paradigm, consisting of several features. Three key features include:

- ***Classes:*** A class encapsulates data and behavior to create objects.
- ***Inheritance:*** Inheritance allows one class (a subclass) to be based on another class (a base class or superclass). Ex: A Shape class may encapsulate data and behavior for geometric shapes, like setting/getting the Shape's name and color, while a Circle class may be a subclass of a Shape, with additional features like setting/getting the center point and radius.
- ***Abstract classes:*** An abstract class is a class that guides the design of subclasses but cannot itself be instantiated as an object. Ex: An abstract Shape class might also specify that any subclass must define a computeArea() method.

participation activity
23.7.1: Classes, inheritance, and abstract classes.

Animation content:

`undefined`

Animation captions:

1. A class provides data/behaviors for objects.
2. Inheritance creates a Circle subclass that implements behaviors specific to a circle.
3. The abstract Shape class specifies "Compute area" is a required behavior of a subclass. Shape does not implement "Compute area", so a Shape object cannot be created.

4. The Circle class implements "Compute area". The Circle class is a non abstract, which is also called a concrete class, and Circle objects can be created.

participation activity
23.7.2: Classes, inheritance, and abstract classes.

Consider the example above.

1)
The Shape class is an abstract class, and the Circle class is a concrete class.

- ○ True
- ○ False

2)
The Shape class can be instantiated as an object.

- ○ True
- ○ False

3)
The Circle class can be instantiated as an object.

- ○ True
- ○ False

4)
The Circle class must implement the computeArea() method.

- ○ True
- ○ False

Example: Biological classification

An example of abstract classes in action is the classification hierarchy used in biology. The upper levels of the hierarchy specify features in common across all members below that level of the hierarchy. As with concrete classes that implement all abstract methods, no creature can actually be instantiated except at the species level.

participation activity
23.7.3: Biological classification uses abstract classes.

Animation content:

```
undefined
```

Animation captions:

1. Each level of the biological hierarchy specifies behaviors common to that level.
2. At this level of the hierarchy, a lot of behavior for the organism is known but the organism is not yet specified.
3. At the final level (species), the organism can be fully described, just as a concrete class can be fully instantiated.

participation activity
23.7.4: Abstract classes.
1)
Consider a program that catalogs the types of trees in a forest. Each tree object contains the tree's species type, age, and location. This program will benefit from an abstract class to represent the trees.

- ○ True
- ○ False

2)
Consider a program that catalogs the types of trees in a forest. Each tree object contains the tree's species type, age, location, and estimated size based on age. Each species uses a different formula to estimate size based on age. This program will benefit from an abstract class.

- ○ True
- ○ False

3)
Consider a program that maintains a grocery list. Each item, like eggs, has an associated price and weight. Each item belongs to a category like produce, meat, or cereal, where each category has additional features, such as meat having a "sell by" date. This program will benefit from an abstract class.

- ○ True
- ○ False

23.8 Abstract classes

Abstract and concrete classes

An abstract method is a method that is not implemented in the base class, thus all derived classes must override the function. An abstract method is denoted by the keyword `abstract` in front of the method signature. A method signature defines the method's name and parameters. Ex:

`abstract double computeArea();` declares an abstract method named computeArea().

An abstract class is a class that cannot be instantiated as an object, but is the superclass for a subclass and specifies how the subclass must be implemented. An abstract class is denoted by the keyword `abstract` in front of the class definition. Any class with one or more abstract methods must be abstract.

A concrete class is a class that is not abstract, and hence *can* be instantiated.

participation activity
23.8.1: A Shape class with an abstract method is an abstract class.

Animation content:

undefined

Animation captions:

1. The Shape class has the abstract computeArea() method.
2. The Shape class is abstract due to having an abstract method, and must contain the abstract keyword in the declaration.
3. An abstract class cannot be instantiated.

participation activity
23.8.2: Shape class.
1)
Shape is an abstract class.

- ○ True
- ○ False

2)
The Shape class defines and provides code for non-abstract methods.

- ○ True
- ○ False

3)
Any class that inherits from Shape must implement the computeArea(), getPosition(), setPosition(), and movePositionRelative() methods.

- ○ True
- ○ False

Ex: Shape classes

The example program below manages sets of shapes. Shape is an abstract class, and Circle and Rectangle are concrete classes. The Shape abstract class specifies that any derived class must define a method computeArea() that returns type double.

Figure 23.8.1: Shape is an abstract class. Circle and Rectangle are concrete classes that extend the Shape class.

Shape.java implements the Shape base class

```
public abstract class Shape {

   protected Point position;

   abstract double computeArea();

   public Point getPosition() {
      return this.position;
   }

   public void setPosition(Point position) {
      this.position = position;
   }

   public void movePositionRelative(Point position) {
      double x = this.position.getX() + position.getX();
      double y = this.position.getY() + position.getY();

      this.position.setX(x);
      this.position.setY(y);
   }
}
```

Point.java holds the x, y coordinates for a point

```
public class Point {

   private double x;
   private double y;

   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }

   public double getX() {
      return x;
   }

   public double getY() {
      return y;
   }

   public void setX(double x) {
      this.x = x;
   }

   public void setY(double y) {
      this.y = y;
   }
}
```

Circle.java defines a Circle class

Rectangle.java defines a Rectangle class

```
public class Circle extends Shape {

   private double radius;

   public Circle(Point center, double radius) {
      this.radius = radius;
      this.position = center;
   }

   @Override
   public double computeArea() {
      return (Math.PI * Math.pow(radius, 2));
   }
}
```

```
public class Rectangle extends Shape {

   private double length, height;

   Rectangle(Point upperLeft, double length, double height) {
      this.position = upperLeft;
      this.length = length;
      this.height = height;
   }

   @Override
   public double computeArea() {
      return (length * height);
   }
}
```

TestShapes.java tests the Shape class

```
public class TestShapes {
   public static void main(String[] args) {
      Circle circle1 = new Circle(new Point(1.0, 1.0), 1.0);
      Circle circle2 = new Circle(new Point(1.0, 1.0), 2.0);

      Rectangle rectangle = new Rectangle(new Point(0.0, 1.0), 1.0, 1.0);

      // Print areas
      System.out.println("Area of circle 1 is: " + circle1.computeArea());
      System.out.println("Area of circle 2 is: " + circle2.computeArea());
      System.out.println("Area of rectangle is: " + rectangle.computeArea());
      System.out.println();

      // Print positions
      System.out.println("Circle 1 is at: (" + circle1.getPosition().getX() +
                         ", " + circle1.getPosition().getY() + ")");

      System.out.println("Rectangle is at: (" + rectangle.getPosition().getX() +
                         ", " + rectangle.getPosition().getY() + ")");
      System.out.println();

      // Move shapes
      circle1.setPosition(new Point(3.0, 1.0));
      rectangle.movePositionRelative(new Point(1.0, 1.0));

      // Print positions
      System.out.println("Circle 1 is at: (" + circle1.getPosition().getX() +
                         ", " + circle1.getPosition().getY() + ")");

      System.out.println("Rectangle is at: (" + rectangle.getPosition().getX() +
                         ", " + rectangle.getPosition().getY() + ")");
      System.out.println();
   }
}
```

```
Area of circle 1 is: 3.141592653589793
Area of circle 2 is: 12.566370614359172
Area of rectangle is: 1.0

Circle 1 is at: (1.0, 1.0)
Rectangle is at: (0.0, 1.0)

Circle 1 is at: (3.0, 1.0)
Rectangle is at: (1.0, 2.0)
```

participation activity
23.8.3: Shape classes.
1)
Since the Circle and Rectangle classes both implement the computeArea() method, Circle and Rectangle are both abstract.

- ○ True
- ○ False

2)
An instance of the ____ class cannot be created.

- ○ Shape
- ○ Point
- ○ Circle

3)
The getPosition() method of the Circle class is implemented in the _____ class.

- ○ Circle
- ○ Rectangle
- ○ Shape

4)
If the Circle class omitted the computeArea() implementation, could Circle objects be instantiated?

- ○ Yes
- ○ No

challenge activity
23.8.1: Abstract classes.
Start

Type the program's output

TestAnimal.java | Animal.java | Domestic.java | Wild.java

public class TestAnimal { public static void main(String[] args) { Domestic myDomestic = new Domestic("Fuzzy", 2, "Gus"); Wild myWild = new Wild("Hobbes", 3, "Tiger"); myDomestic.printInfo(); System.out.println(); myWild.printInfo(); } }

```
Fuzzy, 2 years
Owner: Gus

Hobbes, 3 years
Species: Tiger
```

1

Check Try again

23.9 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.

Figure 23.9.1: Composition.
The 'has-a' relationship. A MotherInfo object 'has a' String object and 'has a' ArrayList of ChildInfo objects, but no inheritance is involved.

```
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;

   ...
}
```

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.

Figure 23.9.2: 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.

```
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;
  ...
}
```

participation activity
23.9.1: 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.

1)
Pear / Fruit

- ○ Is-a
- ○ Has-a

2)
House / Door

- ○ Is-a
- ○ Has-a

3)
Dog / Owner

- ○ Is-an
- ○ Has-an

4)
Mug / Cup

- ○ Is-a
- ○ Has-a

UML diagrams

Programmers commonly draw class inheritance relationships using Unified Modeling Language (UML) notation (IBM: UML basics).

participation activity
23.9.2: UML derived class example: ProduceItem derived from GenericItem.

Animation content:

undefined

Animation captions:

1. A class diagram depicts a class' name, data members, and methods.
2. A solid line with a closed, unfilled arrowhead indicates a class is derived from another class.
3. The derived class only shows additional members.

23.10 UML

UML class diagrams

The Universal Modeling Language (UML) is a language for software design that uses different types of diagrams to visualize the structure and behavior of programs. A structural diagram visualizes static elements of software, such as the variables and methods used in the program. A behavioral diagram visualizes dynamic behavior of software, such as the flow of an algorithm.

A UML class diagram is a structural diagram that can be used to visually model the classes of a computer program, including member variables and methods.

participation activity

23.10.1: UML class diagrams show class names, members, types, and access.

Animation content:

undefined

Animation captions:

1. One box exists for each class. The class name is centered at the top.
2. Class members are listed in the box below. Member variables have a name followed by a colon and the type.
3. Each member method's name and return type is listed similarly.
4. Private and public access is noted to the left of each member. A minus (-) indicates private and a plus (+) indicates public.

participation activity
23.10.2: UML class diagrams.

Refer to the animation above.

1)
The Square class' size member is _____.

- ○ public
- ○ private

2)
The computeArea() method takes a double as a parameter.

- ○ True
- ○ False

3)
Both the Circle and Square class have a member variable named center.

- ○ True
- ○ False

4)
A UML class diagram is a behavioral diagram.

- ○ True
- ○ False

5)
A UML class diagram describes everything that is needed to implement a class.

- ○ True
- ○ False

UML for inheritance

UML uses an arrow with a solid line and an unfilled arrow head to indicate that one class inherits from another. The arrow points toward the superclass.

UML uses italics to denote abstract classes. In particular, UML uses italics for the abstract class' name, and for each abstract method in the class. As a reminder, a superclass does not have to be abstract. Also, any class with an abstract method must be abstract.

participation activity
23.10.3: UML uses italics for abstract classes and methods.

Animation content:

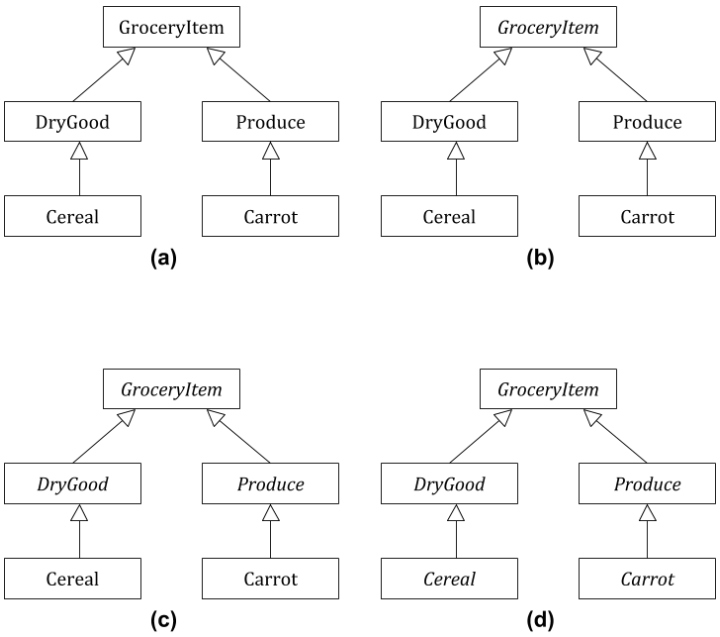
undefined

Animation captions:

1. Shape is an abstract class, so the class name and abstract method are in italics.
2. The solid-lined arrow with an unfilled arrow head indicates that the Circle class inherits from Shape.
3. Circle is a concrete class, so the class name is shown in regular font. Note that Circle implements computeArea().

participation activity
23.10.4: UML for inheritance.

Match the UML diagram to the best description for that diagram. Each of the questions concerns a different implementation of a grocery store inventory system. The figures may look the same, but note the use of italics.



- (a)
- (c)
- (b)
- (d)

GroceryItem is abstract and requires all subclasses to implement specific methods. DryGood and Produce can be created as classes.
GroceryItem is abstract and requires all subclasses to implement specific methods. DryGood and Produce are also abstract, as they require subclasses to implement methods specific to each class.
All classes are concrete.
All classes are abstract.
Reset

23.11 Interfaces

Java provides interfaces as another mechanism for programmers to state that a class adheres to rules defined by the interface. An interface can specify a set of abstract methods that an implementing class must override and define. In an interface, an abstract method does not need the `abstract` keyword in front of the method signature.

To create an interface, a programmer uses the keyword interface. The following code illustrates two interfaces named Drawable and DrawableASCII.

Figure 23.11.1: Creating an interface.

```
import java.awt.Graphics2D;

public interface Drawable {
   public void draw(Graphics2D graphicsObject); }
}
```

```
public interface DrawableASCII {
   public void drawASCII(char drawChar);
```

Drawable declares a draw() method for drawing using Java Swing components, which are discussed elsewhere. DrawableASCII declares a drawASCII() method for drawing using ASCII characters.

Any class that implements an interface must:

- List the interface name after the keyword implements
- Override and implement the interface's abstract methods

Although inheritance and polymorphism allow a class to override methods defined in the superclass, a class can only inherit from a single superclass. A class can implement multiple interfaces using a comma separated list. Each Interface a class implements means the class will adhere to the rules defined by the interface. Ex: Square can implement both the Drawable and DrawableASCII interfaces.

Figure 23.11.2: Implementing an interface.

```
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.Rectangle;

public class Square implements Drawable, DrawableASCII {
   private int sideLength;
```

```
   public Square(int sideLength) {
      this.sideLength = sideLength;
   }

   @Override
   public void draw(Graphics2D graphicsObject) {
      Rectangle shapeObject = new Rectangle(0, 0, this.sideLength, this.sideLength);
      Color colorObject = new Color(255, 0, 0);
      graphicsObject.setColor(colorObject);
      graphicsObject.fill(shapeObject);
   }

   @Override
   public void drawASCII(char drawChar) {
      int rowIndex;
      int columnIndex;

      for (rowIndex = 0; rowIndex < this.sideLength; ++rowIndex) {
         for (columnIndex = 0; columnIndex < this.sideLength; ++columnIndex) {
            System.out.print(drawChar);
         }
         System.out.println();
      }
   }
}
```

participation activity
23.11.1: Comparison of interfaces and abstract classes.

Interfaces and abstract classes can seem superficially similar but they have different purposes. The following questions will help clarify these differences. Choose whether an interface or abstract class is the best choice for each situation.

1)
A class that provides only static final fields.

- ○ Interface
- ○ Abstract class

2)
A class that provides variables/fields.

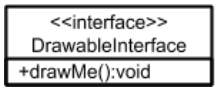
- ○ Interface
- ○ Abstract class

3)
A class that provides an API that must be implemented and no other code.

- ○ Interface
- ○ Abstract class

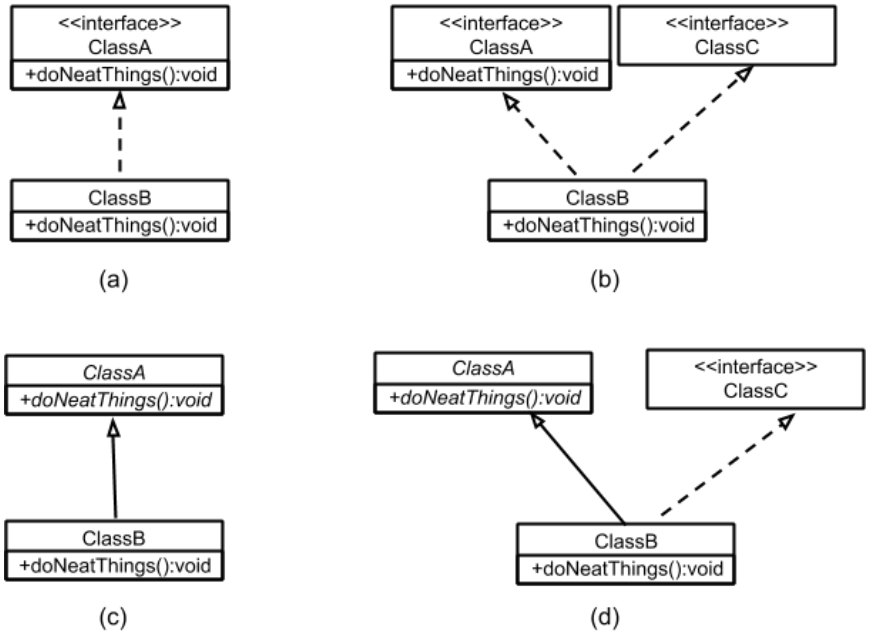
UML Diagrams denote interfaces using the keyword interface, inside double angle brackets, above the class name. Classes that implement the interface have a dashed line with an unfilled arrow pointing at the interface. Following UML conventions is important for clear communication between programmers.

Figure 23.11.3: UML for DrawableInterface.



participation activity
23.11.2: UML interfaces.

Match the UML diagram from above to the code block that it describes.



- (b)
- (a)
- (d)
- (c)

```
public abstract class ClassA {
   public abstract void doNeatThings();
}

public interface ClassC {

}

public class ClassB extends ClassA implements ClassC {
   @Override
   public void doNeatThings() {
      System.out.println("Does neat things!");
   }
}

public interface ClassA {
   public void doNeatThings();
}

public class ClassB implements ClassA {
   @Override
   public void doNeatThings() {
      System.out.println("Does neat things!");
   }
}

public abstract class ClassA {
   public abstract void doNeatThings();
}

public class ClassB extends ClassA {
   @Override
   public void doNeatThings() {
      System.out.println("Does neat things!");
   }
}

public interface ClassA {
   public void doNeatThings();
}

public interface ClassC {

}
```

```
public class ClassB implements ClassA, ClassC {
   @Override
   public void doNeatThings() {
      System.out.println("Does neat things!");
   }
}
```

Reset

challenge activity

23.11.1: Enter the output of the class implementing interfaces.

Start

Type the program's output

PrintMedia.java | MultiMedia.java | Show.java | Media.java

```
public class PrintMedia { public static void
main(String[] args) { Media med = new Media();
med.setDirector("Charlie Brooker");
med.setDuration(45); med.printDuration();
med.printDirector(); } }
```

```
45 minutes
Director: Charlie Brooker
```

1 2

Check Next

23.12 Java example: Employees and instantiating from an abstract class

zyDE 23.12.1: Employees example: Abstract class and interface.

The classes below describe an abstract class named EmployeePerson and two derived concrete classes, EmployeeManager and EmployeeStaff, both of which extend the EmployeePerson class. The main program creates objects of type EmployeeManager and EmployeeStaff and prints them.

1. Run the program. The program prints manager and staff data using the EmployeeManager's and EmployeeStaff's printInfo methods. Those classes override EmployeePerson's getAnnualBonus() method but simply return 0.
2. Modify the EmployeeManager and EmployeeStaff getAnnualBonus methods to return the correct bonus rather than just returning 0. A manager's bonus is 10% of the annual salary and a staff's bonus is 7.5% of the annual salary.

Current file:

EmployeeMain.java

Load default template...

Pre-enter any input for program, then press run.

Run

zyDE 23.12.2: Employees example: Abstract class and interface (solution).

Below is the solution to the above problem. Note that the EmployeePerson class is unchanged.

Current file:

EmployeeMain.java

Load default template...

Pre-enter any input for program, then press run.

Run

23.13 Java example: Employees and overriding class methods

zyDE 23.13.1: 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.

Current file:
EmployeeMain.java
Load default template...
Run

zyDE 23.13.2: 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.

Current file:
EmployeeMain.java
Load default template...
Run

23.14 LAB: Pet information (derived classes)

This section's content is not available for print.

23.15 LAB: Instrument information (derived classes)

This section's content is not available for print.

23.16 LAB: Course information (derived classes)

This section's content is not available for print.

23.17 LAB: Book information (overriding member methods)

This section's content is not available for print.

23.18 LAB: Plant information (ArrayList)

Given a base `Plant` class and a derived `Flower` class, complete main() to create an ArrayList called **myGarden**. The ArrayList should be able to store objects that belong to the `Plant` class or the `Flower` class. Create a method called printArrayList(), that uses the printInfo() methods defined in the respective classes and prints each element in **myGarden**. The program should read plants or flowers from input (ending with -1), adding each `Plant` or `Flower` to the **myGarden** ArrayList, and output each element in **myGarden** using the printInfo() method.

Ex. If the input is:

```
plant Spirea 10
flower Hydrangea 30 false lilac
flower Rose 6 false white
plant Mint 4
-1
```

the output is:

```
Plant Information:
   Plant name: Spirea
```

```
   Cost: 10

Plant Information:
   Plant name: Hydrangea
   Cost: 30
   Annual: false
   Color of flowers: lilac

Plant Information:
   Plant name: Rose
   Cost: 6
   Annual: false
   Color of flowers: white

Plant Information:
   Plant name: Mint
   Cost: 4
```

lab activity
23.18.1: LAB: Plant information (ArrayList)
10 / 10
Current file:
PlantArrayListExample.java
Load default template...
Develop mode | Submit mode
Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click Run program and observe the program's output in the second box.
Enter program input (optional)
If your code requires input values, provide them here.
Run program
Input (from above) →
PlantArrayListExample.java
(Your program)
→ Output (shown below)
Program output displayed here
Signature of your work What is this?
2/24.. W - - - |0 |10 |10 ..2/24