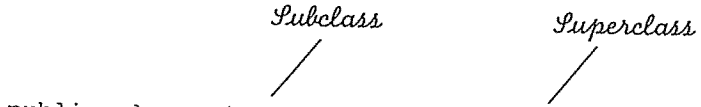
4.3. Inheritance

Inheritance allows a programmer to state that one class *extends* another class, inheriting its features. In Java terminology, a *subclass* extends a *superclass*. `extends` is a Java reserved word. For example:




```
public class HighSchool extends School
{
  ...
}
```

Inheritance implements the IS-A relationship between objects: an object of a subclass type IS-A(n) object of the superclass. A high school is a kind of school. A `HighSchool` object IS-A (kind of) `School` object. Technically this means that in your program you can use an object of a subclass whenever an object of its superclass is expected. For example:

```
School sch = new HighSchool(...);
```

If a constructor or a method in a client class expects a `School` type of parameter to be passed to it, you can call it with a `HighSchool` type of parameter. Objects of a subclass inherit the data type of the superclass.

In Java, a class can directly extend only one superclass — there is no *multiple inheritance* for classes. But more than one subclass can be derived from the same superclass:

```
public class HighSchool extends School ...
public class ElementarySchool extends School ...
public class DrivingSchool extends School ...
```

The IS-A relationship of inheritance is not to be confused with the HAS-A relationship between objects. That *X* "has a" *Y* simply means that *Y* is a data field (an instance variable) in *X*. For example, you might say that a `HighSchool` HAS-A `MarchingBand`, but not that a `HighSchool` IS-A `MarchingBand`.

(2)

Subclass methods

A subclass inherits all the public methods of its superclass, and you can call an inherited method of the same object without any "dot prefix." For example, if `School` has a method

```
public String getName() { ... }
```

then `HighSchool`'s method `registerForAP` can call it directly:

```
public class HighSchool extends School
{
  ...
  public void registerForAP()
  {
    String registrationForm = getName() + ...;
    ...
  }
  ...
}
```

`HighSchool`'s clients can call `getName`, too, for any `HighSchool` object:

```
HighSchool hs = new HighSchool(...);
String name = hs.getName();
```

A subclass can add its own methods. It can also *override* (redefine) a method of the superclass by providing its own version with exactly the same *signature* (the same name, and number, types, and order of parameters). For example, `School` may have a `toString` method, and `HighSchool`'s `toString` may override it.

Occasionally it may be necessary to make an explicit call to a superclass's public method from a subclass. This is accomplished by using the super-dot prefix. For example:

```
public class HighSchool extends School
{
  ...
  public String toString()
  {
    return super.toString() + collegeAcceptance() + ...;
  }
  ...
}
```

The superclass's <u>private</u> methods are not callable in its subclasses.

Subclass constructors

Constructors are not inherited: a subclass has to provide its own or rely on the default no-args constructor.

A subclass's constructors can explicitly call the superclass's constructors using the keyword `super`. For example:

```
public class School
{
  private String name;
  private int numStudents;

  public School(String nm, int num)
  {
    name = nm;
    numStudents = num;
  }
  ...
}

public class ElementarySchool extends School
{
  private int highestGrade;

  public ElementarySchool(String nm, int num, int grade)
  {
    super(nm, num); // calls School's constructor
    highestGrade = grade;
  }
  ...
}
```

If `super(...)` is used, it must be the first statement in the subclass's constructor (as in the above example). If `super` is not used, then superclass's no-args constructor is called by default, and it must exist, or the compiler will report an error.

Subclass's fields

A subclass inherits all the class (static) variables and instance variables of its superclass. However, the instance variables are usually declared private (always private in the AP subset) and so cannot be referenced directly in the subclass.

The superclass's private variables are not directly accessible in its subclass. So you must use public accessors and modifiers to get and set values instance variables declared in the superclass.

Superclass's public constants (`public static final` variables) are accessible everywhere.

A subclass can add its own static or instance variables. For example, the class `ElementarySchool` above, a subclass of `School`, adds an instance variable

```
private int highestGrade;
```

27

Consider the following partial definitions:

```
public class MailingList
{
  private ArrayList<String> people;

  public MailingList() { people = new ArrayList<String>(); }
  public void add(String name) { people.add(name); }
  public ArrayList<String> getPeople() { return people; }

  < Other methods not shown >
}

public class Subscribers extends MailingList
{
  public Subscribers() { }  // Calls super() by default

  // Returns the number of names in people
  private int size()
  {
    return < expression >;
  }

  < Other methods not shown >
}
```

Which of the following should replace < *expression* > in the `size` method of the `Subscribers` class so that the method works as specified?

(A) `super.size();`
(B) `people.size();`
(C) `super.people.size();`
(D) `getPeople().size();`
(E) None of the above

The `MailingList` class HAS-A(n) `ArrayList<String>` `people` as an instance variable, but `MailingList` is not an `ArrayList`: it does not extend `ArrayList`. The programmer has not provided a `size` method for the `MailingList` class (a design mistake), so Choice A is wrong. Choices B or C might look plausible at first, but `people` is private in `MailingList`, so it is not directly accessible in `Subscribers`. But `MailingList` has a public method `getPeople`, and this method is inherited and accessible in the `Subscribers` class. `getPeople` returns an `ArrayList<String>`, which has a method `size` that returns the size of the list. The answer is D.

Also note that

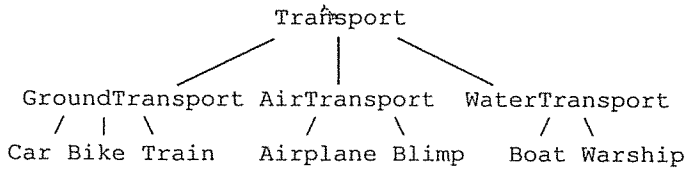
```
return getPeople().size();
```

is equivalent to

```
ArrayList<String> temp = getPeople();
return temp.size();
```

4.4. Class Hierarchies

If you have a class, you can derive one or several subclasses from it. Each of these classes can in turn serve as a superclass for other subclasses. You can build a whole tree-like hierarchy of classes, in which each class has one superclass. For example:

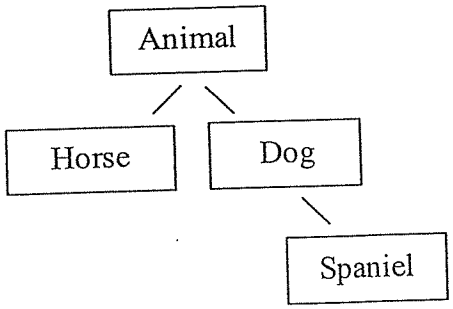


In fact, in Java all classes belong to one big hierarchy; it starts at a class called `Object`. If you do not specify that your class extends any particular class, then it extends `Object` by default. Therefore, every object IS-A(n) `Object`. The `Object` class provides a few common methods, including `equals` and `toString`, but these methods are not very useful and usually get redefined in classes lower in the hierarchy.

Class hierarchies exist to allow reuse of code from higher classes in the lower classes without duplication and to promote a more logical design. A class lower in the hierarchy inherits the data types of all classes above it.

For example, if we have classes

```
public class Animal { ... }
public class Dog extends Animal { ... }
public class Spaniel extends Dog { ... }
```



all of the following declarations are legal:

```
Spaniel s = new Spaniel(...);
Dog d = new Spaniel(...);
Animal a = new Spaniel(...);
```

But if you also define

```
public class Horse extends Animal { ... }
```

then

```
Horse x = new Spaniel(...);
```

is an error, of course: `Spaniel` does not extend `Horse`.

Abstract classes

Classes closer to the top of the hierarchy are more abstract — the properties and methods of their objects are more general. As you proceed down the hierarchy, the classes become more specific and the properties of their objects are more concretely spelled out. Java syntax allows you to define a class that is officially designated `abstract`. For example:

```
public abstract class Solid { ... }
```

An abstract class can have constructors and methods; however, some of its methods may be declared `abstract` and left without code. For example:

```
public abstract class Solid
{
  ...
  public abstract double getVolume();
  ...
}
```

This indicates that every `Solid` object has a method that returns its volume, but the actual implementation may depend on the specific type of solid. For example, the `Sphere` and `Cube` subclasses of `Solid` will define `getVolume` differently.

A class in which all the methods are defined is called *concrete*. Naturally, abstract classes appear near the top of the hierarchy and concrete classes sit below. You cannot instantiate an abstract class, but you can declare variables or arrays of its type. For example:

```
Solid s1 = new Sphere(radius);
Solid s2 = new Cube(side);
Solid[] solids = { new Sphere(100), new Cube(100) };
```

Questions 28-29 refer to the following partial class definitions:

```
public abstract class Account
{
  public Account() { ... }
}

public class BankAccount extends Account

 private double balance;

 public BankAccount(double amount)
 {
   super(); // optional: default
   balance = amount;
 }


public class CheckingAccount extends BankAccount

private String customerName;

public CheckingAccount(String name, double amount)
{
  < missing statements >
}
...
```

(8)

28

Which of the following is an acceptable replacement for < *missing statements* > in `CheckingAccount`'s constructor?

I.
```
balance = amount;
customerName = name;
```

II.
```
super(amount);
customerName = name;
```

III.
```
super(name, amount);
```

(A) I only
(B) II only
(C) I and II only
(D) II and III only
(E) I, II and III

`balance` is private in `BankAccount`; it is not accessible in `CheckingAccount`, so Option I cannot be right. `BankAccount` does not have a constructor with two parameters, so Option III cannot be right either. Option II is the way to go. The answer is B.

29

Which of the following declarations are valid?

I.
```
Account acct = new BankAccount(10.00);
```

II.
```
CheckingAccount acct = new BankAccount(10.00);
```

III.
```
BankAccount acct = new CheckingAccount("Amy", 10.00);
```

(A) I and II only
(B) II and III only
(C) I and III only
(D) I, II, and III
(E) None of the three

It may appear that Option I is wrong because an abstract class `Account` cannot be instantiated. But in fact we are not instantiating `Account` — we are instantiating `BankAccount` and assigning the newly created `BankAccount` object to an `Account` variable. Since a `BankAccount` IS-A(n) `Account`, we are okay. We only have to make sure that `BankAccount` has a constructor that takes one `double` parameter (which it does). Options II and III are clearly problematic as a pair: either a `BankAccount` IS-A `CheckingAccount` or a `CheckingAccount` IS-A `BankAccount`, but not both. Here `CheckingAccount` extends `BankAccount`, so Option III is okay (again, provided `CheckingAccount` has a constructor that takes a `String` and `double` parameters). The answer is C.

4.5. Polymorphism

Polymorphism is a mechanism that ensures that the correct method is called for an object disguised as a more generic type. In our "solids" example, if we write

```
double volume = solids[0].getVolume();
```

the compiler does not know whether `solids[0]` is a `Sphere` or a `Cube`. The decision of which `getVolume` method to call is postponed until run time. In Java implementation, each object holds a pointer to a table of entry points to its methods; thus the object itself "knows" what type of object it is. This technique is called *dynamic method binding* — which method to call is decided at run time, not compile time.

Polymorphism is a feature of Java and other object-oriented languages; all you have to do is understand it and use it correctly.

One common situation when polymorphism comes into play occurs when different types of objects are mixed together in an array or list, as shown in the above example. This code

```
for (Solid solid : solids)
  totalVolume += solid.getVolume();
```

works no matter what `Solids` are stored in the `solids` array because the appropriate `getVolume` method is called for each element of the array. This is true even if several different types of `Solids` are in the `solids` array, such as `Sphere`, `Cube`, or `Pyramid`.

Another situation for polymorphism occurs when a method takes a more generic type of parameter and a client class passes a more specific type of parameter to the method. For example, one of the overloaded versions of `System.out`'s `print` method takes

(10)

30

Given

```
public class Person
{
  private String name;

  public Person(String nm) { name = nm; }
  public String getName() { return name; }
  public String toString { return getName(); }
}

public class OldLady extends Person
{
  private int age;

  public OldLady(String nm, int yrs) { super(nm); age = yrs; }
  public String getName() { return "Mrs. " + super.getName(); }
  public int getAge() { return age; }
}
```

what is the output of the following statements?

```
Person p = new OldLady("Robinson", 92);
System.out.println(p + ", " + ((OldLady)p).getAge());
```

(A) `Mrs. Robinson, 92`
(B) `Robinson, 92`
(C) `Robinson`
(D) `ClassCastException`
(E) No output due to infinite recursion

1. The variable p is disguised as a `Person` type, but it is actually an `OldLady`.
2. `p + ", "` calls p's `toString` method. `OldLady` inherits `toString` from `Person`, so `Person`'s `toString` is called.
3. `toString` in turn calls `getName`. Which one? This is the trickiest part. Both `Person` and `OldLady` have a `getName` method, but, due to polymorphism, `OldLady`'s `getName` will be called (notwithstanding the fact that we call it from `Person`'s `toString` method).
4. `OldLady`'s `getName` takes `"Mrs. "` and appends to it the result of `super.getName()`. The latter explicitly calls `Person`'s `getName`, which simply returns the name.
5. Finally we cast p back to the `OldLady` type — we need this to call its `getAge` method. A `Person` does not have `getAge`, and the compiler does not keep track of what type we assigned to p. `getAge`'s result is appended to the output string.

Choice B tries to make you forget about polymorphism or suggests that polymorphism does not apply here. It does. Choice C is an awkward attempt to confuse you about p's data type. Deep inside, p is not just a `Person` but an `OldLady`, so it does have a `getAge` method once we cast it to `OldLady`. If p were only a `Person`, the cast to `OldLady` would cause a `ClassCastException`, as suggested in Choice D. Choice E hints that `getName` infinitely calls itself. This does not happen here: `OldLady`'s `getName` explicitly calls `Person`'s `getName` as indicated by the super-dot prefix. We would have infinite recursion only if we forgot the super-dot prefix. The answer is A.

4.6. Interfaces

In Java, an interface is even more "abstract" than an abstract class. An interface has no constructors or instance variables and no code at all — just headers for methods. All its methods are public and abstract. For example:

```
public interface Fillable
{
  void fill(int x);
  int getCurrentAmount();
  int getMaximumCapacity();
}
```

(No need to repeat "public abstract" for each method in an interface — it is understood.)

A class "implements" an interface by supplying code for all the interface methods. `implements` is a reserved word. For example:

```
public class Car implements Fillable
{
  private int fuelTankCapacity;
  private int fuelAmount;
  ...
  public void fill(int gallons) { fuelAmount += gallons; }
  public int getCurrentAmount() { return fuelAmount; }
  public int getMaximumCapacity() { return fuelTankCapacity; }
}

public class VendingMachine implements Fillable
{
  private int currentStock;
  ...
  public void fill(int qty) { currentStock += qty; }
  public int getCurrentAmount() { return currentStock; }
  public int getMaximumCapacity() { return 20; }
}
```

For a concrete class to implement an interface, it must define all the methods required by the interface. It must also explicitly state that it implements the interface. A class that claims it implements an interface but does not define some of the interface methods must be declared `abstract`.

A class can extend only one class, but it can implement several interfaces. For example:

```
public class Car extends Vehicle implements Fillable, Sellable { ... }
```

Each interface adds a secondary data type to the objects of the class that implements it. If a class *C* implements interface *I*, objects of *C* can be disguised as type *I* objects and polymorphism applies (the same way as for subclasses). For example, we can have a method:

```
/** Fills all objects in the array a to capacity
 */
public void fillUp(Fillable[] a)
{
  for (Fillable f : a)
    f.fill(f.getMaximumCapacity() - f.getCurrentAmount());
}
```

The formula works polymorphically for different types of `Fillable` objects that might be stored in the array `a`.

If class *C* implements interface *I*, all subclasses of *C* automatically implement *I*.

Interfaces help us write more general methods, facilitating code reuse.

The `Comparable<T>` *interface*

The library (built-in) interface `java.lang.Comparable<T>` is widely used for designating objects that can be compared in some way.

`Comparable<T>` is not in the AP subset, but it is better to understand what it is about, because `String`, `Integer`, and `Double` implement it and you are responsible for using their `compareTo` methods.

We compare objects, for example, when we arrange them in order (sorting) or perform a binary search. Objects of a class that implements `Comparable` are said to have a *natural ordering* defined.

Starting with Java 5.0, the `Comparable<T>` interface is a "generic" interface, that is, it works with objects of a specific type. The `Comparable<T>` interface specifies only one method:

```
int compareTo(T other);
```

The method returns a positive integer if this object is "greater than" the other, zero if they are "equal," and a negative integer if this object is "less than" the other. (Sort of like `this` "minus" `other`.) It is up to the programmer to define what "smaller" and "greater" might mean when comparing objects of his class and whether the value returned by `compareTo` has any meaning besides telling which object is larger or smaller.

`compareTo` takes one parameter of the type *T* — and it is usually assumed that the parameter belongs to the same class that implements `Comparable<T>`. For example:

```
public class Flight implements Comparable<Flight>
{
  ...
  public int compareTo(Flight other)
  {
    return getDepartureTime() - other.getDepartureTime();
  }
}
```

`String`, `Integer`, and `Double` all implement `Comparable`, but, naturally, they do so in different ways. Strings are compared lexicographically. The comparison is case-sensitive, and all uppercase letters precede (are "smaller than") all lowercase letters. `Integer` and `Double` objects are compared as usual, based on their numeric values.

The `Comparable<T>` interface does not specify an `equals` method, and a class that implements `Comparable<T>` will compile without an `equals` method defined. But it is better to override the `equals` method inherited from `Object` and make it consistent with `compareTo`, to avoid possible errors later. For example:

```
public boolean equals(Object x)
{
  return x instanceof Flight && compareTo((Flight)x) == 0;
}
```

The Boolean operator `instanceof` is not in the AP subset.

Note that the parameter to your `equals` method should have the type `Object`, so that your `equals` indeed overrides `Object`'s `equals`.

(A)

Free Response Example - pgs. A-D (Interfacing)
2016 Free Response Questions - pgs. 2-11

4.7. "Design" Question

The free-response section of the exam may include a "design" question, which asks you to design and write a short class a subclass of a given class — see, for example, free-response Question 2 in the 2010 AP CS exam and our solution (Chapter 6).

In writing a subclass of a given class avoid duplicating code of the superclass's constructors or methods —use `super(...)` or `super.someMethod(...)` calls whenever possible.

In designing and writing your class, pay special attention to the following:

- Use reasonable and consistent style with proper indentation and generous spacing between statements.
- Do not include comments — they are not required and you will waste time.
- Choose meaningful names for your class, its methods and their parameters, and fields and local variables. A class name starts with an uppercase letter; the names of all methods, their parameters, and instance variables start with a lowercase letter. Make sure that all constructors have the same name as the class.
- Make all instance variables private and group them together at the top of the class.
- Make all methods public, unless there is a specific hint in the question that some of them are "helper" methods used only inside this class — then make them private.
- Specify appropriate return types for all methods. Recall that constructors do not have a return type, not even `void`.
- Provide public "accessor" methods if specified in the question. An accessor returns the value of the respective instance variable. Accessor names may start with "get." For example:

```
public double getBalance() { ... }
```

- Provide "modifier" methods if specified in the question. Modifiers set the values of one or several instance variables. Modifier names often start with "set". For example:

```
public void setPrice(double amount) { ... }
```

(B)

Part (b) may ask you to write a fragment of code from a client class that uses your class written in Part (a). In answering this part of the question, pay special attention to the following:

- Use constructors and call public methods of your Part (a) class with appropriate numbers, types, and order of parameters, consistent with what you wrote in Part (a). Use this opportunity to double-check the code you wrote in Part (a).
- Never refer directly to private instance variables of your Part (a) class in the client class; rather, call accessors or modifiers.
- It is allowed (and often desirable) to reuse in the client class the same names for variables as you used for similar formal parameters in methods in Part (a). For example, if in Part (a) you wrote

```
public void setPrice(double amount) { ... }
```

then in Part (b) you may write

```
double amount = ...;
...
item.setPrice(amount);
```

#1

Free Response Question – Interfacing & Program Design

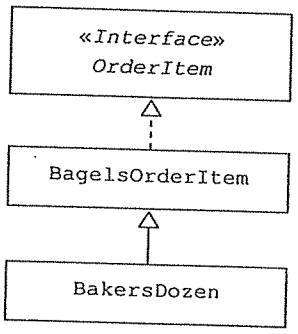
Simply Bagels bakery relies on a Java application for handling its orders. The designer of the application started the project by defining an interface `OrderItem`, shown below.

```
public interface OrderItem
{
  /** Returns the price of this item */
  double getPrice();

  /** Returns the number of units ordered */
  int getQuantity();

  /** Returns the total cost for this order item */
  double getCost();
}
```

She then wrote a class `BagelsOrderItem`, which implements the `OrderItem` interface, and a class `BakersDozen`, a subclass of `BagelsOrderItem`. This arrangement is summarized in the class diagram below.



In this question you will write the `BagelsOrderItem` and `BakersDozen` classes.

(a) Write a class `BagelsOrderItem` that implements the `OrderItem` interface. Implement your class in accordance with the encapsulation principle. Provide one constructor that takes two parameters: `double price` and `int quantity`. Provide accessor methods that return the item's price and quantity. `BagelsOrderItem`'s `getCost` method should return price times quantity. Write the entire `BagelsOrderItem` class, including all necessary instance variables and methods.

(b) Write a class `BakersDozen`, a subclass of `BagelsOrderItem`. A `BakersDozen` object represents an order of 13 bagels, which includes one free bagel.* Provide one constructor that takes one parameter, `double price` (the quantity is known: it is 13). Override `BagelsOrderItem`'s `getCost` method to reflect one free bagel. Do not duplicate the instance variables of `BagelsOrderItem` class in `BakersDozen`. Write class `BakersDozen` below.

For additional practice, write the class `ShoppingCart`. Your class should extend `ArrayList<OrderItem>`. Add only one method, `totalDue`, which returns the sum of the costs of all the items in the list. `ShoppingCart` does not have any explicitly defined constructors or instance variables. Note that a `ShoppingCart` can contain `BagelsOrderItem` objects, `BakersDozen` objects, perhaps other kinds of `OrderItem` objects. For example, the segment of code

```
ShoppingCart cart = new ShoppingCart();
cart.add(new BagelsOrderItem(0.25, 3));
cart.add(new BakersDozen(0.35));
System.out.printf("Total due: $%.02f", cart.totalDue());
```

should display `Total due: $4.95`.

* According to Wikipedia, the oldest known source for the expression "baker's dozen" dates to the 13th century in one of the earliest English statutes, instituted during the reign of Henry III (1216–1272), called the Assize of Bread and Ale. Bakers who were found to have shortchanged customers (some variations say that they would sell hollow bread) could be subject to severe punishment including judicial amputation of a hand. To guard against losing a hand to an axe, a baker would give 13 for the price of 12 in order to be certain of not being called a cheat.