APCS Inheritance & Abstract Classes Notes

***Inheritance* is used in a Java program when the objects manipulated by the program form a natural hierarchy using an *"is-a"* relationship.** For example, suppose we want to design a program for a bookstore that sells several different kinds of books, including children's books and textbooks. A children's book is a book, and so is a textbook. Therefore, we might want to define classes `ChildrensBook` and `Textbook` as subclasses of the `Book` class (and the `Book` class will then be the superclass of both the `ChildrensBook` and `Textbook` classes). A subclass is defined using the keyword extends as follows:

```
public class ChildrensBook extends Book {
   . . .
}

public class TextBook extends Book {
   . . .
 }
```

An advantage of defining classes this way is that **subclasses inherit all of the fields and methods of their superclasses** (except the constructors, which are discussed below). For example, since every book has a title, a regular price, a sale price, and the number of copies sold, there is no need to include those fields in the definitions of the `ChildrensBook` and `TextBook` classes; they will be inherited automatically. Similarly, there is no need to redefine the `sell`, `doDiscount`, `mostPopular`, and `newPrice` methods; every `ChildrensBook` and every `TextBook` will have those methods.

Note that the relationship between a book and a title is a *"has-a"* relationship (a book has a title), not an *"is-a"* relationship. That is why `title` is a field, not a subclass, of a `Book`.

Usually, a subclass will define some new fields and/or methods that are not defined by its superclass. For example, a `ChildrensBook` might include the age of the children for whom it is intended, and a `TextBook` might include the name of the course for which it is required:

```
public class ChildrensBook extends Book {
   private int childsAge; /*** new field ***/
   public int getAge() { return childsAge; } /*** new method ***/
}

public class TextBook extends Book {
   private String requiredBy; /*** new field ***/
   public String getRequiredBy( ) { return requiredBy; } /*** new method ***/
}
```

Constructors

As mentioned above, **the constructors of a superclass are not inherited by its subclasses. However, a subclass's constructors always call a superclass constructor, either explicitly or implicitly. A superclass constructor is called explicitly using *super*.** For example, we could define a constructor for the `ChildrensBook` class to include an explicit call to the `Book` constructor like this:

```
public ChildrensBook(String theTitle, double price, int age) {
   super(theTitle, price);
   childsAge = age;
}
```

When this `ChildrensBook` constructor is called, it first calls the `Book` constructor to initialize the `title`, `regularPrice`, `salePrice`, and `numSold` fields; it then initializes the `childsAge` field itself.

Note that if an explicit call to the superclass' constructor is included, it must be the first statement in the subclass constructor.

If a subclass's constructor does not include an explicit call to one of its superclass's constructors, then there will be an implicit call to the superclass' default constructor (i.e.,the compiler will add a call). If the superclass does not have a default constructor, this implicit call will cause a compile-time error. For example, if we failed to include an explicit call to super in the `ChildrensBook` or `TextBook` constructors, we would get a compile-time error since the `Book` class has no default constructor.

Using a Subclass Instead of a Superclass

Another advantage of using inheritance is that you can use a subclass object anywhere that a superclass object is expected. For example, because every textbook is a book, any method that has a parameter of type `Book` can be called with an argument of type `TextBook`; you do not have to write two versions of the method, one for `Book` parameters and the other' for `TextBook` parameters. For example, the following method computes the difference between a book's regular price and its sale price:

```
public double priceDifference( Book b ) {
   return(b.getPrice() - b.getSalePrice());
}
```

The method will work just fine if it is called with either a `Book` or a `TextBook`:

```
Book b = ...
TextBook tb = ...
double dl = priceDifference(b); // call with a Book argument
double d2 = priceDifference(tb); // call with a TextBook argument
```

Similarly, it is fine to assign from a `TextBook` to a `Book`, because a `Book` is expected on the right-hand side of the =, and a `TextBook` is a `Book`:

```
TextBook tb = ...
Book b = tb;  // assign from a TextBook to a Book
```

Although a subclass object can be used anywhere a superclass object is expected, the reverse is not true: in general, you cannot use a superclass object where a subclass object is expected. For example, you cannot call a method that has a `TextBook` parameter with a `Book` argument, and you cannot assign from a `Book` to a `TextBook`. To illustrate this, assume that the following method has been defined in the `TextBook` class:

```
public static boolean sameClass( TextBook tbl, TextBook tb2 ) {
   return ((tbl.requiredBy).equals(tb2.requiredBy));
}
```

The following code would cause two compile-time errors, as noted in the comments:

```
Book b = ...
TextBook tb = b;  // compile-time error! Can't assign from a Book to a TextBook
if(TextBook.sameClass( b, tb )) ...// compile-time error! Can't use a Book
                                   // argument when the corresponding parameter
                                   // is a TextBook
```

If you know that a particular `Book` variable is actually pointing to a `TextBook` object,then you can use a class cast to tell the compiler that it is OK to use that variable where a `TextBook` is expected. For example:

```
Book b = new TextBook(...); // b points to a TextBook object
TextBook tb;
tb = (TextBook)b; // no compile-time error
if(TextBook.sameClass( (TextBook)b, tb )) ... // no compile-time error
```

Although the use of a class cast prevents a compile-time error, a runtime check is still performed to make sure that the `Book` variable really is pointing to a `TextBook` object. If not, an exception is thrown. For example:

```
Book b = new Book(...);  // b points to a Book object
TextBook tb;
tb = (TextBook)b;  // runtime error!  b points to a Book, not a TextBook
if(TextBook.sameClass( (TextBook)b, tb )) ..// runtime error!  b points to a
                                           // Book!, not a TextBook
```

Overloading and Overriding Methods

Just as a class can define *overloaded* methods (methods with the same name but different signatures), a subclass can overload a method of its superclass by defining a method with the same name but a different signature. For example, the designer of the `ChildrensBook` class might want a second version of the `mostPopular` method that finds the most popular book for children of a particular age:

```
public static Book mostPopular( Book[] bookList, int age) {
// precondition: bookList.length > 0
// postcondition: returns the book intended for children of
//                the given age that has sold the most copies
   . . .
}
```

In addition to overloading methods defined by its superclass, a subclass can also *override* a superclass method; that is, it can define a new version of the method specialized to work on subclass objects. A superclass method is overridden when the subclass defines a method with exactly the same name, the same number of parameters, and the same types of parameters as the superclass.

For example, a special formula might be used to compute the sale price of children's books (different from the formula used for other kinds of books). In this case, the `ChildrensBook` class might override the `Book` class's definition of the `doDiscount` method as follows:

```
// lower the sale price for a childrens book (by 20%)
public void doDiscount( ) {
   salePrice = salePrice * .8;
}
```

As discussed above, a variable of type `Book` may actually point to a `Book` object, a `TextBook` object, or a `ChildrensBook` object. The type of the object actually pointed to (not the declared type of the variable) is what determines which version of an overridden method is called (this is known as dynamic dispatch). For example:

```
Book b = new Book(...);
Book tb = new TextBook(...);
Book cb = new ChildrensBook(...);
b.doDiscount();  // b points to a Book object, so the Book
                 // class's doDiscount method is called
tb.doDiscount(); // tb points to a TextBook object;
                 // the doDiscount method was not overridden
                 // in the TextBook class, so the Book class's
                 // doDiscount method is called
cb.doDiscount(); // cb points to a ChildrensBook object;
                 // the doDiscount method was overridden in the
                 // ChildrensBook class, so the ChildrensBook
                 // class's doDiscount method is called
```

In this example, variables `b`, `tb`, and `cb` are all declared to be of type `Book`. However, `tb` is initialized to point to a `TextBook`, and `cb` is initialized to point to a `ChildrensBook`. The calls `b.doDiscount()` and `tb.doDiscount()` cause the `Book` class' `doDiscount` method to be called (because `b` points to a `Book`, and because `tb` points to a `TextBook` and the `TextBook` class does not override the `doDiscount` method). The call `cb.doDiscount()` causes the `ChildrensBook` class's `doDiscount` method to be called (because `cb` points to a `ChildrensBook`, and that class does override the `doDiscount` method).

Abstract Methods and Classes

Suppose you want to define a class hierarchy in which some method needs to be provided by all subclasses, but there is no reasonable default version (i.e., it is not possible to define a version of the method in the superclass that makes sense for the subclasses). For example, you might define a `Shape` class with three subclasses: `Circle`, `Square`, and `Rectangle`. A `Circle` will have fields that specify the coordinates of its center and its radius. A `Square` will have fields that specify the coordinates of its upper-left comer and the length of one side. A `Rectangle` will have fields that specify the coordinates of its upper-left comer, its height, and its width.

It will be useful to have a `Draw` method for all `Shape`s; however, there is no reasonable `Draw` method that will work for a `Circle`, a `Square`, and a `Rectangle`. This is a time to use an abstract method: a method that is declared in a class but defined only in a subclass. (For our example, the `Draw` method will be the abstract method; it will be declared in the `Shape` class, and it will be defined in each of the three subclasses: `Circle`, `Square`, and `Rectangle`.)

Here's the syntax:

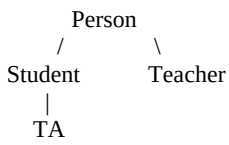
```
public abstract class Shape {
   abstract public void Draw(); // no body, just the method header
}

public class Circle extends Shape {
   public void Draw() {
      // code for Circle's Draw method goes here
   }
}

public class Square extends Shape {
   public void Draw() {
      // code for Square's Draw method goes here
   }
}

public class Rectangle extends Shape {
   public void Draw() {
      // code for Rectangle's Draw method goes here
   }
}
```

Note that if a class includes an abstract method, the class must be declared abstract (otherwise you get a compile-time error). Also, an abstract class cannot be instantiated (you cannot create an instance of the class itself, only of one of its subclasses). For example:

```
Shape s;             // OK -- just a pointer to a Shape,
                     // no attempt to create a Shape object
s = new Circle();    // OK -- Circle is not an abstract class
s = new Shape();     // Error! Can't instantiate an abstract class
```

Interfaces

Some objects have more than one *"is-a"* relationship. For example, consider designing classes to represent some of the people associated with a university. One way to think of the hierarchical relationship among those people is as shown below:



This diagram says that a TA (a teaching assistant) is a student, a student is a person, and a teacher is also a person. However, although a TA is certainly a student, in some ways, a TA is also a teacher (e.g., a TA teaches a class and gets paid). Java does not allow you to make the `TA` class a subclass of both the `Student` class and the `Teacher` class. One solution to this problem is to make `TA` a subclass of the `Student` class and to use an *interface* to define what TAs and teachers have in common.

An interface is similar to a class, but it can only contain:

- **public, static, final fields (i.e., constants)**
- **public, abstract methods (i.e., just method headers, no bodies)**

Here's an example:

```
public interface Employee {
   void raiseSalary( double d );
   double getSalary();
}
```

Note that both methods are implicitly public and abstract (those keywords can be provide but are not necessary).

A class can implement one or more interfaces (in addition to extending one class). It must provide bodies for all of the methods declared in the interface, or else it must be abstract. For example:

```
public class TA implements Employee extends Student {
   public void raiseSalary( double d ) {
      // actual code here
   }
   public double getSalary() {
      // actual code here
   }
}
```

Many classes can implement the same interface (e.g., both the `TA` class and the `Teacher` class can implement the `Employee` interface). Interfaces provide a way to group similar objects. For example, you could write a method with a parameter of type `Employee`, and then call that method with either a `TA` or a `Teacher` object. If you hadn't used the `Employee` interface (e.g., if you simply wrote `raiseSalary` and `getSalary` methods for the `TA` and `Teacher` classes), writing such a method would be very clumsy.

More on 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 (is also an 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 you have a constructor or a method (elsewhere, in a client class), which expects a `School` type of parameter passed to it, you can call it with a `HighSchool` type of argument. 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 you can derive different subclasses 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 member (an instance variable) in *X*. For example, you might say that a `HighSchool` HAS-A `MarchingBand`, but not a `HighSchool` IS-A `MarchingBand`.

Subclass methods

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

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

`HighSchool`'s `apRegister` can call it directly:

```
public class HighSchool extends School
{
   public void makeApForm()
   { 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, return type, and number and types 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' public (or protected) method from a subclass. This is accomplished using the `super.` prefix. For example:

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

The superclass' private methods are not callable in the subclass.

Subclass constructors

♦ Constructors are not inherited: a subclass has to provide its own. ♦

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

```
public class School
{
   public School (String name, int numStudents)
   {
      myName = name;
      myNumStudents = numStudents;
   }
}

public class ElementarySchool extends School
   public ElementarySchool(String name, int numStudents, int highestGrade)
   {
      super (name, numStudents); II Calls School's constructor
      myHighestGrade = highestGrade;
   }
   private int myHighestGrade;
}
```

♦ If super is used, it must be the first statement in the subclass's constructor. ♦

Subclass variables

A subclass inherits all the class (static) and instance variables of its superclass. However, the instance variables are usually private (always private in the AP subset).

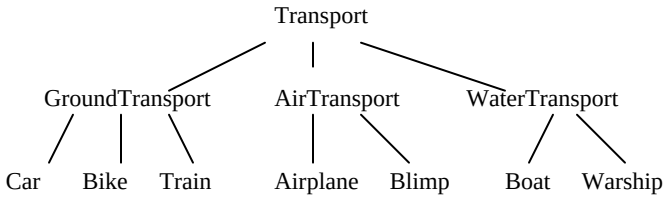
♦ The superclass's private variables are not directly accessible in its subclass. So you might as well forget that they are inherited — instead, use public accessors and modifiers to get and set values of the superclass's private instance variables. ♦

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

A subclass can add its own static or instance variables.

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 do belong to one big hierarchy; it starts at a class `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(...);
```

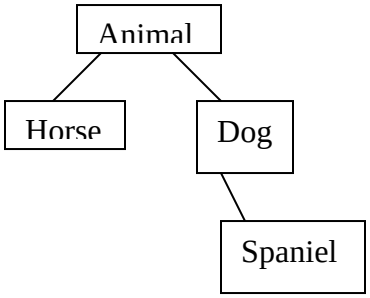
But if you also define

```
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 code 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) };
```

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 headings 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 interfaces — 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
{
   . . .
   public void fill(int gallons) { fuelAmount += gallons; }
   public int getCurrentAmount() { return fuelAmount; }
   public int getMaximumcapacity() { return fuelTankCapacity; }
}

public class VendingMachine implements Fillable
{
    . . .
   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:

```
public class C extends B implements I1, I2, I3 { ... }
```

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