What is an abstract class?

*When we think of a class, we assume that programs will create objects of that type. Sometimes it’s useful to declare classes—called****abstract classes****—for which you*never*intend to create objects. Because they’re used only as superclasses in inheritance hierarchies, we refer to them as****abstract superclasses****. These classes cannot be used to instantiate objects, because, as we’ll soon see, abstract classes are*incomplete*. Subclasses must declare the “missing pieces” to become “concrete” classes, from which you can instantiate objects. Otherwise, these subclasses, too, will be abstract. We demonstrate abstract classes in (Deitel)*

*What is the purpose of an abstract class (this may require some research)?*

*An abstract class’s purpose is to provide an appropriate superclass from which other classes can inherit and thus share a common design*

*Classes that can be used to instantiate objects are called****concrete classes****. Such classes provide implementations of*every*method they declare (some of the implementations can be inherited).*

*Abstract superclasses are*too general*to create real objects—they specify only what is common among subclasses. We need to be more*specific*before we can create objects.*

*Concrete classes provide the specifics that make it reasonable to instantiate objects.*

*Not all hierarchies contain abstract classes. However, you’ll often write client code that uses only abstract superclass types to reduce the client code’s dependencies on a range of subclass types. For example, you can write a method with a parameter of an abstract superclass type. When called, such a method can receive an object of*any*concrete class that directly or indirectly extends the superclass specified as the parameter’s type.*

*Abstract classes sometimes constitute several levels of a hierarchy. For example, the Shape hierarchy of*[*Fig. 9.3*](http://proquest.safaribooksonline.com.portal.lib.fit.edu/9780133813036/ch09lev2sec2_html#ch09fig03)*begins with abstract class Shape. On the next level of the hierarchy are*abstract*classes TwoDimensionalShape and ThreeDimensionalShape. The next level of the hierarchy declares*concrete*classes forTwoDimensionalShapes (Circle, Square and Triangle) and for ThreeDimensionalShapes (Sphere, Cube and Tetrahedron).*

*(*Deitel)

Provide a unique Java code example of an abstract class.

**Declaring an Abstract Class and Abstract Methods**

You make a class abstract by declaring it with keyword **abstract**. An abstract class normally contains one or more **abstract methods**. An abstract method is an *instance method* with keyword abstract in its declaration, as in

Although we cannot instantiate objects of abstract superclasses, you’ll soon see that we can use abstract superclasses to declare variables that can hold references to objects of any concrete class derived from those abstract superclasses. We’ll use such variables to manipulate subclass objects polymorphically. You also can use abstract superclass names to invoke static methods declared in those abstract superclasses.

https://www.cs.berkeley.edu/~jrs/61b/lec/12

CS 61B: Lecture 12

Wednesday, February 19, 2014

Today's reading: Sierra & Bates, Chapter 8.

ABSTRACT CLASSES

================

An abstract class is a class whose sole purpose is to be extended.

public abstract class List {

protected int size;

public int length() {

return size;

}

public abstract void insertFront(Object item);

}

Abstract classes don't allow you to create objects directly. You can declare a

variable of type List, but you can't create a List object.

List myList; // Right on.

myList = new List(); // COMPILE-TIME ERROR.

However, abstract classes can be extended in the same way as ordinary classes,

and the subclasses are usually not abstract. (They can be, but usually they're

normal subclasses with complete implementations.)

The abstract List class above includes an abstract method, insertFront. An

abstract method lacks an implementation. One purpose of an abstract method is

to guarantee that every non-abstract subclass will implement the method.

Specifically, every non-abstract subclass of List must have an implementation

for the insertFront method.

public class SList extends List {

// inherits the "size" field.

protected SListNode head;

// inherits the "length" method.

public void insertFront(Object item) {

head = new SListNode(item, head);

size++;

}

}

If you leave out the implementation of insertFront in SList, the Java compiler

will complain that you must provide one. A non-abstract class may never

contain an abstract method, nor inherit one without providing an

implementation.

Because SList is not abstract, we can create SList objects; and because SLists

are Lists, we can assign an SList to a List variable.

List myList = new SList(); // Right on.

myList.insertFront(obj); // Right on.

What are abstract classes good for? It's all about the interface.

----------------------------------------------------

| An abstract class lets you define an interface |

| - for multiple classes to share, |

| - without defining any of them yet. |

----------------------------------------------------

Let's consider the List class. Although the List class is abstract, it is an

ADT--even without any implementation!-- because it has an interface with public

method prototypes and well-defined behaviors. We can implement an

algorithm--for example, a list sorter--based on the List interface, without

ever knowing how the lists will be implemented. One list sorter can sort every

kind of List.

public void listSort(List l) { ... }

In another part of the universe, your project partners can build lots of

subclasses of List: SList, DList, TailList, and so on. They can also build

special-case List subclasses: for example, a TimedList that records the amount

of time spent doing List operations, and a TransactionList that logs all

changes made to the list on a disk so that all information can be recovered if

a power outage occurs. A library catalogue application that uses DLists can

send them to your listSort algorithm to be sorted. An airline flight database

that uses TransactionLists can send them to you for sorting, too, and you don't

have to change a line of sorting code. You may have written your list sorter

years before TransactionLists were ever thought of.

----------------- The list sorter is built on the foundation of a list

| Application | ADT, and the application is built on the foundation of

----------------- the list sorter. However, it's the application, and

| not the list sorter, that gets to choose what kind of

| calls list is actually used, and thereby obtains special

v features like transaction logging. This is a big

----------------- advantage of object-oriented languages like Java.

| List Sorter |

-----------------

|

| calls

v

-----------------

| List ADT |

-----------------

JAVA INTERFACES

===============

Java has an "interface" keyword which refers to something quite different than

the interfaces I defined in Lecture 8, even though the two interfaces are

related. Henceforth, when I say "interfaces" I mean public fields, public

method prototypes, and the behaviors of public methods. When I say "Java

interfaces" I mean Java's "interface" keyword.

A Java interface is just like an abstract class, except for two differences.

(1) In Java, a class can inherit from only one class, even if the superclass

is an abstract class. However, a class can "implement" (inherit from) as

many Java interfaces as you like.

(2) A Java interface cannot implement any methods, nor can it include any

fields except "final static" constants. It only contains method

prototypes and constants.

public interface Nukeable { // In Nukeable.java

public void nuke();

}

public interface Comparable { // In java.lang

public int compareTo(Object o);

}

public class SList extends List implements Nukeable, Comparable {

[Previous stuff here.]

public void nuke() {

head = null;

size = 0;

}

public int compareTo(Object o) {

[Returns a number < 0 if this < o,

0 if this.equals(o),

> 0 if this > o.]

}

}

Observe that the method prototypes in a Java interface may be declared without

the "abstract" keyword, because it would be redundant; a Java interface cannot

contain a method implementation.

The distinction between abstract classes and Java interfaces exists because of

technical reasons that you might begin to understand if you take CS 164

(Compilers). Some languages, like C++, allow "multiple inheritance," so that a

subclass can inherit from several superclasses. Java does not allow multiple

inheritance in its full generality, but it offers a sort of crippled form of

multiple inheritance: a class can "implement" multiple Java interfaces.

Why does Java have this limitation? Multiple inheritance introduces a lot of

problems in both the definition of a language and the efficient implementation

of a language. For example, what should we do if a class inherits from two

different superclasses two different methods or fields with the same name?

Multiple inheritance is responsible for some of the scariest tricks and traps

of the C++ language, subtleties that cause much wailing and gnashing of teeth.

Java interfaces don't have these problems.

Because an SList is a Nukeable and a Comparable, we can assign it to variables

of these types.

Nukeable n = new SList();

Comparable c = (Comparable) n;

The cast is required because not every Nukeable is a Comparable.

"Comparable" is a standard interface in the Java library. By having a class

implement Comparable, you immediately gain access to Java's sorting library.

For instance, the Arrays class in java.util includes a method that sorts arrays

of Comparable objects.

public static void sort(Object[] a) // In java.util

The parameter's type is Object[], but a run-time error will occur if any item

stored in a is not a Comparable.

Interfaces can be extended with subinterfaces. A subinterface can have

multiple superinterfaces, so we can group several interfaces into one.

public interface NukeAndCompare extends Nukeable, Comparable { }

We could also add more method prototypes and constants, but in this example

I don't.