## Abstract Classes and Methods

1. **Abstract Classes**
2. An *abstract* class is one from which no objects can be created.

(Classes from which objects *may* be created are commonly called "concrete" classes).

1. The purpose of an abstract class is to create a common form for all classes to be derived from it.

It establishes a basic form so you can see what is in common among all classes derived from it, and use this as a guide when deriving new classes.

1. You designate a class to be abstract by placing the keyword **abstract** before the keyword **class** in the declaration, e.g.

**public abstract class** BankAccount

D. Although no objects of an abstract class type may be created, abstract classes may have instance variables and a constructor! Here is why:

Recall that when an object of a subclass is created, the superclass “sub-object” must be created first, via a call from the subclass constructor to the superclass constructor. (See the *Inheritance* document online)

So, an abstract superclass with instance variables inherited by its subclasses would need a constructor – not to create an object of the superclass type, but to initialize the superclass sub-object (the inherited instance variables) when a subclass object is created.

1. **Abstract Methods**
2. Recall that methods declared in Java *interfaces* are not allowed to have an implementation (i.e., no method body). Such methods are commonly called *abstract methods*.
3. Abstract classes may also have abstract methods. In fact, if a class has one or more abstract *methods,* the class *must* be declared abstract.
4. Unlike interfaces, however, abstract classes may also have non-abstract (i.e. “concrete”) methods.
5. Whereas all *interface* methods are implicitly abstract, an abstract class must explicitly designate methods to be abstract by using keyword **abstract** in the declaration, e.g.,

**public abstract void** play() ; // note missing body

1. All abstract methods must be overridden in each subclass, unless the subclass is also abstract.
2. If an abstract method is *not* overriden in a given subclass, then the compiler considers that subclass to *also* be abstract and requires it to be so declared.
3. **What If a Superclass Method is *not* Overridden?**
4. If a method is not overridden in a particular subclass, then that subclass uses its superclass' version of the method (assuming it *is* overridden in the superclass).
5. If the superclass *also* does not override the method, then the version from *its* superclass is used, and so on up the inheritance hierarchy.
6. This process may ultimately lead all the way up the hierarchy to superclass **Object.**