Java Enterprise Edition

ANUDIP FOUNDATION

Declaring Classes

Objective: • Declaration • Types • fields	**Materials Required:** 1. Eclipse IDE/InteliJ/STC
Theory:40 mins	**Practical :20mins**
Total Duration: 60 mins	

Declaring Classes

You've seen classes defined in the following way:

```
class MyClass {
   // field, constructor, and
   // method declarations
}
```

This is a *class declaration*. The *class body* (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

The preceding class declaration is a minimal one. It contains only those components of a class declaration that are required. You can provide more information about the class, such as the name of its superclass, whether it implements any interfaces, and so on, at the start of the class declaration. For example,

```
class MyClass extends MySuperClass implements YourInterface {
   // field, constructor, and
   // method declarations
}
```

means that MyClass is a subclass of MySuperClass and that it implements the YourInterface interface.

You can also add modifiers like *public* or *private* at the very beginning—so you can see that the opening line of a class declaration can become quite complicated. The modifiers *public* and *private*, which determine what other classes can access MyClass, are discussed later in this lesson. The lesson on interfaces and inheritance will explain how and why you would use the *extends* and *implements* keywords in a class declaration. For the moment you do not need to worry about these extra complications.

In general, class declarations can include these components, in order:

1. Modifiers such as *public*, *private*, and a number of others that you will encounter later.
2. The class name, with the initial letter capitalized by convention.
3. The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class can only *extend* (subclass) one parent.
4. A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*. A class can *implement* more than one interface.
5. The class body, surrounded by braces, {}.

OR

Declaring Class:

Class:

A class is a prescription for a particular kind of object — it defines a new type. You use the definition of a class to create objects of that class type — that is, to create objects that incorporate all the components specified as belonging to that class.

Note: The String class is a comprehensive definition for a String object, with all the operations you are likely to need specified. Whenever you create a new String object, you are creating an object with all the characteristics and operations specified by the class definition. Every String object has all the methods that the String class defines built in. This makes String objects indispensable and string handling within a program easy.

A class definition is very simple. There are just two kinds of things that you can include in a class defi nition:

Fields: These are variables that store data items that typically differentiate one object of the class from another. They are also referred to as data members of a class.

Methods: These define the operations you can perform for the class — so they determine what you can do to, or with, objects of the class. Methods typically operate on the fields — the data members of the class.

The fields in a class definition can be of any of the primitive types, or they can be references to objects of any class type, including the one that you are defining. The methods in a class definition are named, self-contained blocks of code that typically operate on the fields that appear in the class definition. Note, though, that this doesn't necessarily have to be the case, as you might have guessed from the main() methods you have written in all the examples up to now.

Fields in a Class Definition

An object of a class is also referred to as an instance of that class. When you create an object, the object contains all the fields that were included in the class definition. However, the fields in a class definition are not all the same — there are two kinds.

One kind of field is associated with the class and is shared by all objects of the class. There is only one copy of each of these kinds of fields no matter how many class objects are created, and they exist even if no objects of the class have been created. This kind of variable is referred to as a class variable because the field belongs to the class and not to any particular object, although as I've said, all objects of the class share it. These fields are also referred to as static fields because you use the static keyword when you declare them.

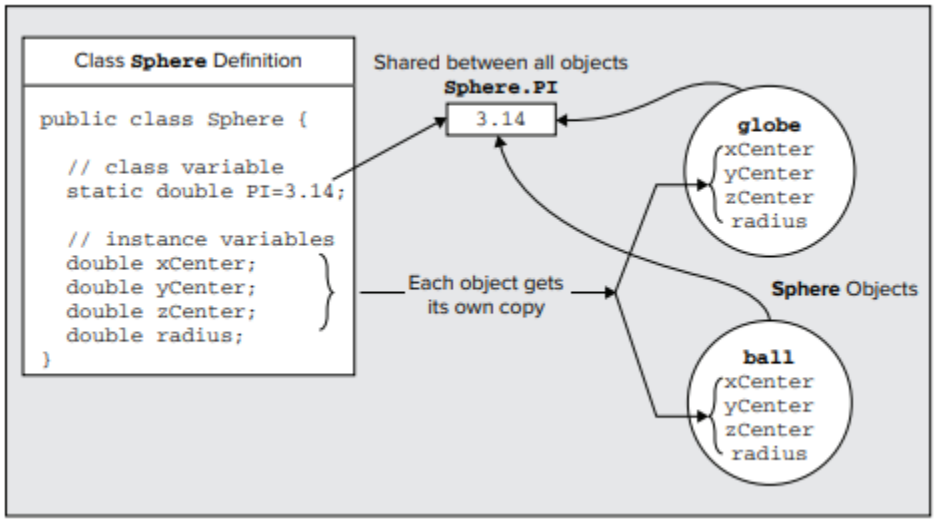
The other kind of field in a class is associated with each object uniquely — each instance of the class has its own copy of each of these fields, each with its own value assigned. These fields differentiate one object

from another, giving an object its individuality — the particular name, address, and telephone number in a given Person object, for example. These are referred to as non-static fields or instance variables because you specify them without using the static keyword, and each instance of a class type has its own independent set.

Because this is extremely important to understand, let's summarize the two kinds of fi elds that you can include in your classes:

1. **Non-static fields**, also called instance variables: Each object of the class has its own copy of each of the non-static fi elds or instance variables that appear in the class defi nition. Each object has its own values for each instance variable. The name instance variable originates from the fact that an object is an instance or an occurrence of a class, and the values stored in the instance variables for the object differentiate the object from others of the same class type. You declare an instance variable within the class definition in the usual way, with a type name and a variable name, and it can have an initial value specified.
2. **Static fields,** also called class variables: A given class has only one copy of each of its static fields or class variables, and these are shared between and among all the objects of the class. Each class variable exists even if no objects of the class have been created. Class variables belong to the class, and they can be referenced by any object or class method, not just methods belonging to instances of that class. If the value of a static field is changed, the new value is available equally in all the objects of the class. This is quite different from non-static fields, where changing a value for one object does not affect the values in other objects. A static field must be declared using the keyword static preceding the type name.

Fig: shows a schematic of a class, Sphere, that has one class variable, PI, and four instance variables, radius, xCenter, yCenter, and zCenter. Each of the objects, globe and ball, has its own set of variables with the names radius, xCenter, yCenter, and zCenter, but both share a single copy of the class variable PI.



Methods in a Class Definition

The methods that you define for a class provide the actions that can be carried out using the variables specified in the class definition. Analogous to the variables in a class definition, there are two varieties of methods — instance methods and class methods. You can execute class methods even when no objects of a class exist, whereas instance methods can be executed only in relation to a particular object, so if no objects exist, you have no way to execute any of the instance methods defined in the class. Again, like class variables, class methods are declared using the keyword static, so they are sometimes referred to as static methods. You saw in the previous chapter that the valueOf() method is a static member of the String class.

Because static methods can be executed when there are no objects in existence, they cannot refer to instance variables. This is quite sensible if you think about it — trying to operate with variables that might not exist is bound to cause trouble. In fact the Java compiler won't let you try. If you reference an instance variable in the code for a static method, it doesn't compile — you just get an error message. The main() method, where execution of a Java application starts, must always be declared as static, as you have seen. The reason for this should be apparent by now. Before an application starts execution, no objects exist, so to start execution, you need a method that is executable even though there are no objects around — a static method therefore. The Sphere class might well have an instance method volume() to calculate the volume of a particular object. It might also have a class method objectCount() to return the current count of how many objects of type Sphere have been created. If no objects exist, you could still call this method and get the count 0.

NOTE:

Note that although instance methods are specific to objects of a class, there is only ever one copy of each instance method in memory that is shared by all objects of the class, as it would be extremely expensive to replicate all the instance methods for each object. A special mechanism ensures that each time you call a method the code executes in a manner that is specific to an object, but I'll defer explaining how this is possible until a little later in this chapter

DEFINING CLASSES

To define a class you use the keyword class followed by the name of the class followed by a pair of braces enclosing the details of the definition. Let's consider a concrete example to see how this works in practice. The definition of the Sphere class that I mentioned earlier could be:

```
class Sphere {
  static final double PI = 3.14;       // Class variable that has a fixed value
  static int count = 0;                // Class variable to count objects

  // Instance variables
  double radius;                       // Radius of a sphere

  double xCenter;                      // 3D coordinates
  double yCenter;                      // of the center
  double zCenter;                      // of a sphere

  // Plus the rest of the class definition...
}
```

NOTE

Whenever you want to make sure that a variable will not be modified, you just need to declare the variable with the keyword final. By convention, variables that are constants have names in capital letters.