Java Lecture-8

Abstract Classes

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.

- That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

- You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier.

- These methods are sometimes referred to as *subclasser responsibility.*

- Syntax to declare a method abstract:

  abstract *type name*(*parameter-list*);

- Any class that contains one or more abstract methods must also be declared abstract.

- To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.

- There can be no objects of an abstract class.

- That is, an abstract class cannot be directly instantiated with the **new** operator.

- Also, you cannot declare abstract constructors, or abstract static methods.

- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared **abstract** itself.

```
// A Simple demonstration of abstract.
abstract class A {
  abstract void callme();

  // concrete methods are still allowed in abstract classes
  void callmetoo() {
    System.out.println("This is a concrete method.");
  }
}

class B extends A {
  void callme() {
    System.out.println("B's implementation of callme.");
  }
}

class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();

    b.callme();
    b.callmetoo();
  }
}
```

Using final with Inheritance

The keyword **final** has three uses.

- First, it can be used to create the equivalent of a named constant.

  The other two uses of final apply to inheritance.

- Using **final** to Prevent Overriding

- Using **final** to Prevent Inheritance

Using **final** to Prevent Overriding

To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration.

```
class A {
  final void meth() {
    System.out.println("This is a final method.");
  }
}

class B extends A {
  void meth() { // ERROR! Can't override.
    System.out.println("Illegal!");
  }
}
```

- Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to *inline* calls to them because it “knows” they will not be overridden by a subclass.

- When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call.

- Inlining is an option only with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called *late binding*.

- However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called *early binding*.

Using final to Prevent Inheritance

- Sometimes you will want to prevent a class from being inherited.
- To do this, precede the class declaration with **final**.
- Declaring a class as **final** implicitly declares all of its methods as **final**, too.
- As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

```
final class A {
  //...
}

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
  //...
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.