Arrays and Strings: Creating an array, one and two dimensional arrays, string array and methods, Classes: String and String Buffer classes, Wrapper classes: Basics types, using super, Multilevel hierarchy, abstract and final classes, Object class, Packages and interfaces, Access protection, Extending Interfaces, packages.

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.
- Such a class determines the nature of the methods that the subclasses must implement.
- 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* because they have no implementation specified in the superclass.
- Thus, a subclass must override them—it cannot simply use the version defined in the superclass.

To declare an abstract method, use this general form:

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

- Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references.
- Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

```
// Using abstract methods and classes.
abstract class Figure {
  double dim1;
  double dim2;

  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  // area is now an abstract method
  abstract double area();
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // override area for rectangle
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
  }
}

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}
```

```
class AbstractAreas {
  public static void main(String args[]) {
  // Figure f = new Figure(10, 10); // illegal now
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref; // this is OK, no object is created

    figref = r;
    System.out.println("Area is " + figref.area());

    figref = t;


     System.out.println("Area is " + figref.area());
    }
  }
```

Using final with Inheritance

Using final to Prevent Overriding

- While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring.

- To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden. The following fragment illustrates final:

```
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!");
  }
}
```

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**.