Basic Concepts of Java Programming: Part 4

Reusability properties: Super class & subclasses including multilevel hierarchy, process of constructor calling in inheritance, use of super and final keywords with super() method, dynamic method dispatch, use of abstract classes & methods, interfaces. Creation of packages, importing packages, member access for packages

UD

Aug 2023

Recapitulation: Super Class & Subclass

```
class Shape {
       double wid, ht; // Width & Height dimensions
       void showDim(){ System.out.println("Width: "+wid+" Height: "+ ht); }
}

class Triangle extends Shape {
       String colr; // Colour of traiangle
       double area() { return wid * ht / 2; }
       void showColr() { System.out.println("Triangle colour:"+colr ); }
}

class Shapes {
       public static void main( String[] args) {
          Triangle t1 = new Triangle (); Triangle t2 = new Triangle();
          t1.wid = 3.0; t1.ht = 4.0; t1.colr="RED";
          t2.wid = 4.0; t2.ht = 5.0; t2.colr="BLUE";
          System.out.println("Details of t1");
          t1.showColr(); t1.showDim(); System.out.println("Area: "+t1.area()
);
          System.out.println(); System.out.println("Details of t2");
          t2.showColr(); t2.showDim(); System.out.println("Area: "+t2.area()
);
       }
}
```

Recapitulation: Notes on Super Class & Subclass

```
class Shape {
        double wid, ht; // Width & Height of Traingle
        void showDim(){ System.out.println("Width: "+wid+" Height: "+ ht); }
}

class Triangle extends Shape {
        String colr; // Colour of traiangle
        double area() { return wid * ht / 2; }
        void showColr() { System.out.println("Triangle colour:"+colr ); }
}

class Shapes {
        publ
            T
            t
            t
            S
            t                                                          )
);
            S
            t                                                          )
);
        }
}
```

Note:

- Subclass is specialized version of Super Class, e.g. *Triangle* subclass from *Shape* super class
- Subclass can access appropriate members & methods under Super Class
- Superclass can be used to create any number of more specific subclasses, e.g. *Rectangle* etc
- Multiple inheritance not supported in Java;

Recapitulation: Private variables in Superclass

```
class Shape {
        private double wid, ht; // Width & Height of Traingle
        void showDim(){ System.out.println("Width: "+wid+" Height: "+ ht); }
}

class Triangle extends Shape {
        String colr; // Colour of traiangle
        double area() { return wid * ht / 2; }
        void showColr() { System.out.println("Triangle colour:"+colr ); }
}

class ShapesPvt {
        public static void main( String[] args) {
           Triangle t1 = new Triangle (); Triangle t2 = new Triangle();
           t1.wid = 3.0; t1.ht = 4.0; /* ? */ t1.colr="RED";
           t2.wid = 4.0; t2.ht = 5.0; /* ? */ t2.colr="BLUE";
           System.out.println("Details of t1");
           t1.showColr(); t1.showDim(); System.out.println("Area: "+t1.area()
);
           System.out.println(); System.out.println("Details of t2");
           t2.showColr(); t2.showDim(); System.out.println("Area: "+t2.area()
);
        }
}
```

Recapitulation: Private variables in Superclass

```
class Shape {
        private double wid, ht; // Width & Height of Traingle
        void showDim(){ System.out.println("Width: "+wid+" Height: "+ ht); }
}

class Triangle extends Shape {
        String colr; // Colour of traiangle
        double area() { return wid * ht / 2;
        void showColr() { System.out.println(
}

class ShapesPvt {
        public static void main( String[] arg
           Triangle t1 = new Triangle (); Tri
           t1.wid = 3.0; t1.ht = 4.0; /* ? */
           t2.wid = 4.0; t2.ht = 5.0; /* ? */
           System.out.println("Details of t1"
           t1.showColr(); t1.showDim(); Syste
);
           System.out.println(); System.out.println("Details of t2");
           t2.showColr(); t2.showDim(); System.out.println("Area: "+t2.area()
);
        }
}
```

← Error: cannot access private members of a superclass

← Same error as above

← Same error as above

Recapitulation: Code correction with accessors

```
class Shape {
        private double wid, ht; // Width & Height of Traingle
        void showDim(){ System.out.println("Width: "+ wid +" Height: "+ ht ); }
        double getWidth() { return wid; } // (1) ← Introduced 4 Accessor methods
        double getHeight() { return ht; } // (2)
        void setWidth( double w ) { wid = w; } // (3)
        void setHeight( double h ) { ht = h; } // (4)
}

class Triangle extends Shape {
        String colr; // Colour of traiangle
        double area() { return getWidth() * getHeight() / 2; } // Accessor
        void showColr() { System.out.println("Triangle colour:"+colr ); }
}

class ShapesPvt {
        public static void main( String[] args) {
           Triangle t1 = new Triangle (); Triangle t2 = new Triangle();
           t1.setWidth(3.0); t1.setHeight(4.0);// Used Superclass'accessor method
           t1.colr="RED";
           t2.setWidth(4.0); t2.setHeight(5.0);// Used Superclass'accessor method
           t2.colr="BLUE";
           System.out.println("Details of t1");
           t1.showColr(); t1.showDim(); System.out.println( "Area: "+t1.area() );
           System.out.println(); System.out.println( "Details of t2" );
           t2.showColr(); t2.showDim(); System.out.println("Area: "+t2.area() );
        }
```

Recapitulation: Private variable in a subclass

```
class Shape {
        private double wid, ht; // Width & Height of Traingle
        void showDim(){ System.out.println("Width: "+ wid +" Height: "+ ht ); }
        double getWidth() { return wid; } // (1) ← Introduced 4 Accessor methods
        double getHeight() { return ht; } // (2)
        void setWidth( double w ) { wid = w; } // (3)
        void setHeight( double h ) { ht = h; } // (4)
}

class Triangle extends Shape {
        private String colr; // Colour of traiangle
        double area() { return getWidth() * getHeight() / 2; } // Accessor
        void showColr() { System.out.println("Triangle colour:"+colr ); }
}

class ShapesPvtSub {
        public static void main( String[] args) {
           Triangle t1 = new Triangle (); Triangle t2 = new Triangle();
           t1.setWidth(3.0)  ← Error: cannot access private      class'accessor method
           t1.colr="RED"; /     members of another class
           t2.setWidth(4.0)                                     class'accessor method
           t2.colr="BLUE"; // ← Same as above
           System.out.println("Details of t1");
           t1.showColr(); t1.showDim(); System.out.println( "Area: "+t1.area() );
           System.out.println(); System.out.println( "Details of t2" );
           t2.showColr(); t2.showDim(); System.out.println("Area: "+t2.area() );
        }
```

Recapitulation: Using Subclass constructor

```
class Shape {
        private double wid, ht; // Width & Height of Traingle
        void showDim(){ System.out.println("Width: "+ wid +" Height: "+ ht ); }
        double getWidth() { return wid; } // (1) ← Introduced 4 Accessor methods
        double getHeight() { return ht; } // (2)
        void setWidth( double w ) { wid = w; } // (3)
        void setHeight( double h ) { ht = h; } // (4)
}

class Triangle extends Shape {
        private String colr; // Colour of traiangle
        Triangle(String c, double w, double h) { colr=c; setWidth(w); setHeight(h); }
        double area() { return getWidth() * getHeight() / 2; } // Accessor
        void showColr() { System.out.println("Triangle colour:"+colr ); }
}

class ShapesPvtSub {
        public static void main( String[] args) {
           Triangle t1 = new Triangle("RED", 3.0, 4.0);
           Triangle t2 = new Triangle("BLUE", 4.0, 5.0);

           System.out.println("Details of t1");
           t1.showColr(); t1.showDim(); System.out.println( "Area: "+t1.area() );

           System.out.println(); System.out.println( "Details of t2" );
           t2.showColr(); t2.showDim(); System.out.println("Area: "+t2.area() );
        }
```

Adding Superclass constructor

```
class Shape {
        private double wid, ht; // Width & Height of Traingle
        Shape(double w, double h) { wid=w; ht=h; } // Superclass constructor
        void showDim(){ System.out.println("Width: "+wid+" Height: "+ht); }
        double getWidth() { return wid; }
        double getHeight() { return ht; }
        void setWidth( double w ) { wid = w; }
        void setHeight( double h ) { ht = h; }
}

class Triangle extends Shape {
        private String colr; // Colour of traiangle
        Triangle(String c, double w, double h){
           super(w,h); // Calls Superclass constructor
           colr=c;
        }
        double area() { return getWidth() * getHeight() / 2; }
        void showColr() { System.out.println("Triangle colour:"+colr ); }
}

class ShapesSuperCons {
        public static void main( String[] args) {
            Triangle t1 = new Triangle("RED", 3.0, 4.0);
            Triangle t2 = new Triangle("BLUE", 4.0, 5.0);

            System.out.println("Details of t1");
            t1.showColr(); t1.showDim(); System.out.println( "Area: "+t1.area() );

            System.out.println(); System.out.println( "Details of t2" );
            t2.showColr(); t2.showDim(); System.out.println("Area: "+t2.area() );
        }
}
```

Adding Superclass constructor

```
class Shape {
        private double wid, ht; // Width & Height of Traingle
        Shape(double w, double h) { wid=w; ht=h; } // Superclass constructor
        void showDim(){ System.out.println("Width: "+wid+" Height: "+ht); }
        double getWidth() { return wid; }
        double getHeight() { return ht; }
        void setWidth( double w ) { wid = w; }
        void setHeight( double h ) { ht = h; }
}

class Triangle extends Shape {
        private String colr; // Colour of traiangle
        Triangle(String c, double w, double h){
           super(w,h); // Calls Superclass constructor
           colr=c;
        }
        double area() { return getWidth() * getHeight() / 2; }
        void showColr() { System.out.println("Triangle colour:"+colr ); }
}

class ShapesS
        publ
                                                          1.area() );
                                                          ;
                                                          .area() );
        }
}
```

Note:

- Super() must always be the first statement inside subclass constructor
- Constructor overloading can be used with varying number of parameters for superclass and subclass

What's the output of below code?

```
class A { int i; } // Superclass

class B extends A { // Subclass
    int i;

    B( int a, int b) {
        i=a; /* Need to access i in A; how? */
        i=b; /* i in B */
    }

    void show() {
        System.out.println("i in Superclass A: " + i); /* How? */
        System.out.println("i in Subclass B: " + i);
    }
}

class TestSuper {
    public static void main(String[] args) {
        B sc = new B(10, 90);
        sc.show();
    }
}
```

Output (not intended really!!):

```
i in Superclass A: 90
i in Subclass B: 90
```

Corrected code using super

```
class A { int i; } // Superclass

class B extends A { // Subclass
    int i;

    B( int a, int b) {
        super.i=a; /* i in A */
        i=b; /* i in B */
    }

    void show() {
        System.out.println("i in Superclass A: " + super.i);
        System.out.println("i in Subclass B: " + i);
    }
}

class TestSuper {
    public static void main(String[] args) {
        B sc = new B(10, 90);
        sc.show();
    }
}
```

Output (ok now):
i in Superclass A: 10
i in Subclass B: 90

Notes:

1) Use **super** somewhat like **this**, where the former is used to refer superclass and the later for current class
2) **super** in this form can also be used to call methods in Superclass

Recapitulation: Multilevel Hierarchy

- Multiple layers of inheritance
- Subclass inherits aspects of all Superclasses at previous layers

```
// Demonstrate when constructors are executed
// Create a super class
class A {       // Superclass
       A(){ System.out.println( "Constructing A" ); }
}

class B extends A {    //Subclass of A
       B(){ System.out.println( "Constructing B"
}

class C extends B {    //Subclass of B
       C(){ System.out.println( "Constructing C"
}

class OrderofConstruction {
    public static void main(String[] args) {
       C oc = new C();
    }
}
```

Output:

```
Constructing A
Constructing B
Constructing C
```

Note:

- Constructors are executed in order of derivation
- Superclass has no knowledge of any subclass.
- Initializations made in superclass is independent of that in subclass

Recapitulation: Type Compatibility

- Refer below code; class X and Y are having same code
- What is the expected output?

```
class X {
        int a;

        X(int i){ a = i ; }
}

class Y {
        int a;

        Y(int i){ a = i; }
}

class TestRef {
    public static void main(String[] args) {
        X x1 = new X(10);
        Y y1 = new Y(5);

        X x2;
        x2 = x1; System.out.println("x2.a: " + x2.a);
        x2 = y1; System.out.println("x2.a: " + x2.a);
    }
}
```

Output:
Compilation error for line: *x2 = y1;*

Note:
- Java is strongly typed language
- Object reference variable can refer only to objects of same type
- Compilation error in case of type mismatch; however there is an exception..

Recapitulation: Type Compatibility (contd.)

- Superclass reference can refer to a subclass object

```
class X {
        int a;
        X(int i) { a = i; }
}

class Y extends X {
        int b;
        Y(int i, int j){
                super(j);
                b = i; }
}

class SuperSubRef {
    public static void main(String[] args) {
        X x1 = new X(10);
        Y y1 = new Y(5, 6);

        X x2;
        x2 = x1; System.out.println("x2.a: " + x2.a);
        x2 = y1; System.out.println("x2.a: " + x2.a);
        x2.a = 20; System.out.println("x2.a: " + x2.a);
        x2.b = 27; System.out.println("x2.a: " + x2.a);
    }
}
```

Recapitulation: Type Compatibility (contd.)

- Superclass reference can refer to a subclass object

```
class X {
        int a;
        X(int i) { a = i; }
}

class Y extends X {
        int b;
        Y(int i, int j){
                super(j);
                b = i; }
}

class SuperSubRef {
    public static void main(String
        X x1 = new X(10);
        Y y1 = new Y(5, 6);

        X x2;
        x2 = x1; System.out.println
        x2 = y1; System.out.println
        x2.a = 20; System.out.print
        x2.b = 27; System.out.print
    }
}
```

Output:

Compilation error for the line: *x2.b = 27;*

When removed and compiled freshly:

10

6

20

Note:

- Y derived from X; hence Y object can be assigned to X reference (x2 in this example)
- X2 can't access b even when it refers to a Y object (e.g. y1) ← Superclass has no knowledge of what a subclass adds to it
- Superclass reference can refer to subclass object ← Reverse is not true

Using **final**

- **Prevent method overriding**
- **Prevent class inheritance**

  e.g. encapsulate control of a hardware device
- **Prevent changes in member variable**

  e.g. named constant

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

class B extends A {
        void meth() {
            System.out.println("What do you expect?");
        }
}
```

Output:

```
A.java:8: error: meth() in B cannot override meth() in A
        void meth() {
             ^
  overridden method is final
1 error
```

So… **final prevents method overriding**

Using final: Prevents class inheritance

- Explicitly declare a class as **final**
  → **Implicitly** declares underlying methods as final too
- Illegal to declare a class as both **abstract** and **final**

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

class B extends A {
        void meth() {
            System.out.println("What do you expect?");
        }
}
```

Output:

```
A.java:7: error: cannot inherit from final A
class B extends A {
                ^
1 error
```

Using final with Data Members

- Used for creating named constants with initial value
- Cannot be changed throughout the program's lifetime

```
// Return error message
class ErrorMsg {
    final int OUTERR=0, INERR=1, DISKERR=2, INDXERR=3;// Error codes
    String[] msgs = { "Error: Output Error", "Error: Input Error",
"Error: Disk Full", "Error: Index out of range" };

    String getErrorMsg (int i) {
      if (i >= 0 && i < msgs.length)
          return msgs[i];
      else
          return( "Invalid error code" );
    };

    public static void main( String[] args) {
      ErrorMsg err = new ErrorMsg();
      System.out.println(err.getErrorMsg( err.INDXERR ));
    }
}
```

Output:

```
Error: Index out of range
```

Using final with Data Members

Estimate output for the following program:

```
// Return error message
class ErrorMsg {
    final int OUTERR=0, INERR=1, DISKERR=2, INDXERR=3; /Error codes
    String[] msgs = { "Error: Output Error", "Error: Input Error",
"Error: Disk Full", "Error: Index out of range" };

    String getErrorMsg (int i) {
      if (i >= 0 & i < msgs.length)
          return msgs[i];
      else
          return( "Invalid error code" );
    };

    public static void main( String[] args) {
      ErrorMsg err = new ErrorMsg();
      System.out.println(err.getErrorMsg( err.INDXERR ));
      err.INDXERR = 99;
    }
}
```

Output:

```
error: cannot assign a value to final variable INDXERR
```

Using final with static

- **final** can be used on method parameters and local variables
- Refers constant through its class name rather than object

```
// Return error message
class ErrorMsg {
    final static int OUTERR=0, INERR=1, DISKERR=2, INDXERR=3; /Error
codes
    String[] msgs = { "Error: Output Error", "Error: Input Error",
"Error: Disk Full", "Error: Index out of range" };

    String getErrorMsg (int i) {
      if (i >= 0 & i < msgs.length)
          return msgs[i];
      else
          return( "Invalid error code" );
    };

    public static void main( String[] args) {
      ErrorMsg err = new ErrorMsg();
      System.out.println(err.getErrorMsg( ErrorMsg.INDXERR ));
    }
}
```

Note:

- final static member accessed by referring class name
- Accessing method within same class, hence class reference **optional**

Dynamic Method Dispatch

- Mechanism by which a call to an overridden method is resolved at run time rather than compile time
- Used for implementing **run-time polymorphism**

```
class Shape { void who() {System.out.println("Shape");} }
class Triangle extends Shape { void who() {System.out.println("Triangle");} }
class Rectangle extends Shape { void who() {System.out.println("Rectangle");} }
class DynDispTest {
        public static void main(String[] args) {
                Shape s1 = new Shape();
                Triangle t1 = new Triangle();
                Rectangle r1 = new Rectangle();
                Shape sref;

                sref = s1; sref.who(); // Dynamic binding / resolcved run-time
                sref = t1; sref.who(); // Dynamic binding / resolcved run-time
                sref = r1; sref.who(); // Dynamic binding / resolcved run-time
        }
}
```

Abstract class

- A class that contains one or more abstract methods, e.g.

```
abstract class Shape2D {
    // Declare member variables: width, height, name;
    :::
    // Define parametrized constructors and default constructor
    :::
    // Accessor methods: getWid(), getHt(), setWid(), setHt() etc.
    :::
    // Define abstract area()
    abstract double area();
}
class Triangle extends Shape2D {
    // Declare specific member variable: colour
    :::
    // Define constructors
    :::
    double area() { return getWd()*getHt() / 2 };
    :::
}
```

- Abstract superclass Shape2D defines only a generalized form of area() and details are filled by subclass Triangle
- abstract used only for normal methods (no static methods/constructors)
- Abstract class cannot be instantiated for incomplete implementation

Interfaces

- Specifies what must be done, not how will be done, e.g.

```
public interface Series {
    int getNext(); // return next number in series
    void reset();  // restart series from starting value
    void setStart(int x); // set starting value
}
```

- Interface Series defined; 1 or more classes can implement, e.g.

```
class ByTwos implements Series { //Implement Series interface
    int start, val;
    ByTwos() { start=0; val=0; }
    //Implement methods specified by Series interface; must be
       public; return type & signature must match
    public int getNext() { val += 2; return val; } // Even numbers
    public void reset() { val = start; }
    public void setStart(int x) { start=x; val=x; }
}
class SeriesDemo {
    public static void main(String[] args) {
       ByTwos ob = new ByTwos(); // Print even series now
       for (int i=0; i<5; i++) System.out.println(ob.getNext());
       ob.setStart(20); System.out.println("Starting now from 20:");
       for (int i=0; i<5; i++) System.out.println(ob.getNext());
    }
}
```

Interfaces

- Specifies what must be done, not how will be done, e.g.

```
public interface Series {
    int getNext(); // return next number in series
    void reset();  // restart series from starting value
    void setStart(int x); // set s
}
```

- Interface **Series** defined; 1 or

```
class ByTwos implements Series {
    int start, val;
    ByTwos() { start=0; val=0; }
    //Implement methods specified
       public; return type & signa
    public int getNext() { val +=
    public void reset() { val = st
    public void setStart(int x) {
}
class SeriesDemo {
    public static void main(String
       ByTwos ob = new ByTwos(); /
       for (int i=0; i<5; i++) Sys
       ob.setStart(20); System.out
       for (int i=0; i<5; i++) Sys
    }
}
```

Did you notice?

- Here, interface and 2 classes were put under the same file
- File name must be **Series.java** (as **Series** is declared as **public**). However, SeriesDemo having main() method should be run for execution.

Output:

```
2
4
6
8
10
Starting now from 20:
22
24
26
28
30
```

Interfaces (contd)

- Provides 100% abstraction
- Since an interface defines a type, you can declare a reference variable of interface type, e.g.

```
ByTwos ob2s   = new ByTwos();
ByThrees ob3s = new ByThrees();
Series iref; // An interface reference
iRef = ob2s; // Refers to a ByTwos object
/ :::
iRef = ob3s; // Refers to a ByThrees object
```

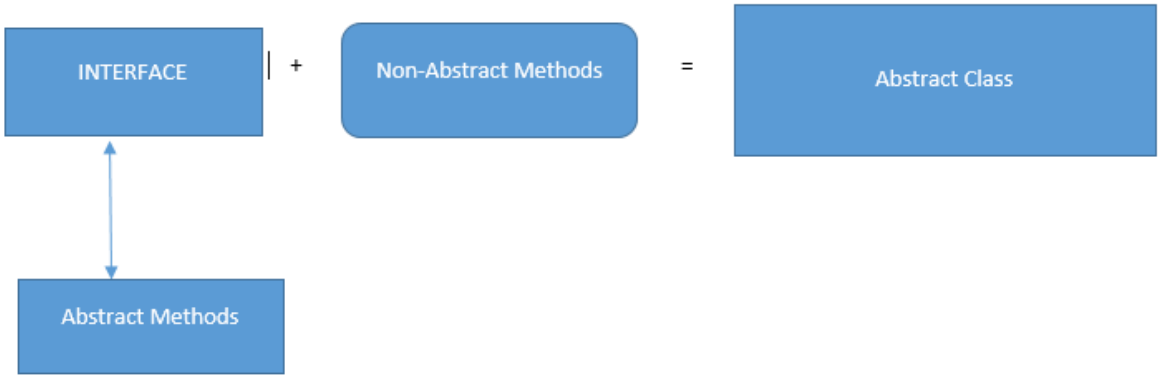
- A class can implement 1 or more interfaces, e.g.

```
interface IfA { void doSomething(); }
interface IfB { void doSomethingElse(); }
class MyClass implements IfA, IfB { // Must override interface methods
      public void doSomething() {System.out.println("Something");
      public void doSomethingElse() {System.out.println("Something else");
}
```

- For IfA & IfB having same method, there will be just one version of method under the implementing class
- Interface can also include variables; must be initialized with access specifier and treated as constants
- Interfaces can be extended; a class implementing such interface must provide implementations for all methods as per interface inheritance chain
- Interface can be declared within another interface; called as member interface or nested interface

Abstract Class vs. Interfaces

#	Abstract Class	Interface
1	Can extend only one class or one abstract class at a time	Can implement any number of interfaces at a time (allowing multiple inheritance)
2	Can extend another concrete / regular class or abstract class	Can only extend another interface
3	Can have both abstract and concrete methods	Can have only abstract methods
4	Keyword “abstract” is mandatory to declare abstract method	Keyword “abstract” is optional to declare a method as an abstract
5	Can have protected and public abstract methods	Can only have public abstract methods
6	Can have static, final or static final variable with any access specifier	Can only have public static final (constant) variable



Recapitulation: Packages

- Provides a mechanism for organizing related pieces of a program as a unit. Also, it encapsulates classes with suitable access control
- Avoid name collisions in a large Java program
- Demonstration of a short package:

```
package bookpack; // Current file part of bookpack package

class Book {
        private String title, author;
        private int pubYear;
        Book(String t, String a, int y) {title=t; author=a; pubYear=y;}
        void show() {System.out.println(title+"|"+author+"|"+pubYear);}
}
public class BookDemo {//From parent directory, run: java bookpack.Bookdemo
    public static void main(String...args) {
        Book[] books=new Book[2];
        books[0]=new Book("Java Fundamentals", "Schildt & Skrien", 2013);
        books[1]=new Book("Programming With Java", "Balagurusamy", 2014);
        for (int i=0; i<books.length; i++) {
                books[i].show();
                System.out.println();
        } // Compilation: javac -d . BookDemo.java
    } // Alternatively, CLASSPATH must be set to include the path to bookpack
} // Alternatively, use -classpath option with Java command
```

- Import class(es) from a package, e.g.

```
import myPack.MyClass; // specific class import
import myPack.*; //entire content of myPack will be imported
```

- **Static import** enables direct reference of static members. Use **sqrt()** directly after: `import static java.lang.Math.sqrt;`