Programming in Java

A Closer Look at Methods and Classes

Mahesh Kumar
(maheshkumar@andc.du.ac.in)

Course Web Page
(www.mkbhandari.com/mkwiki)

Outline

1. Overloading Methods
2. Overloading Constructors
3. Using Objects as Parameters
4. A Closer Look at Argument Passing
5. Returning Objects
6. Recursion
7. Introducing Access Control
8. Understanding static
9. Introducing final
10. Nested and Inner Classes

Outline

11. String Class
12. Using Command-Line Arguments
13. Varargs: Variable-Length Arguments [Self Study]

Overloading Methods

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- Method overloading is one of the ways that Java supports polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Overloading Methods

```
// Method overloading Demo
class OverloadDemo {
        void test() {
                System.out.println("No parameters");
        }

        // Overload test for one integer parameter.
        void test(int a) {
                System.out.println("a: " + a);
        }

        // Overload test for two integer parameters.
        void test(int a, int b) {
                System.out.println("a and b: " + a + " " + b);
        }

        // Overload test for a double parameter
        double test(double a) {
                System.out.println("double a: " + a);
                return a*a;
        }
}
```

Overloading Methods

```
// Method overloading Demo
class OverloadDemo {
        void test() {
                System.out.println("No parameters");
        }

        // Overload test for one integer parameter.
        void test(int a) {
                System.out.println("a: " + a);
        }

        // Overload test for two integer parameters.
        void test(int a, int b) {
                System.out.println("a and b: " + a + " " + b);
        }

        // Overload test for a double parameter
        double test(double a) {
                System.out.println("double a: " + a);
                return a*a;
        }
}
```

```
class Overload {
        public static void main(String args[ ]) {
                OverloadDemo ob = new OverloadDemo( );
                double result;

                // call all versions of test( ) - overloaded 4 times
                ob.test( );
                ob.test(10);
                ob.test(10, 20);
                result = ob.test(123.25);
                System.out.println("Result of ob.test(123.25): " + result);
        }
}
```

Overloading Methods

```
// Method overloading Demo
class OverloadDemo {
        void test() {
                System.out.println("No parameters");
        }

        // Overload test for one integer parameter.
        void test(int a) {
                System.out.println("a: " + a);
        }

        // Overload test for two integer parameters.
        void test(int a, int b) {
                System.out.println("a and b: " + a + " " + b);
        }

        // Overload test for a double parameter
        double test(double a) {
                System.out.println("double a: " + a);
                return a*a;
        }
}
```

```
class Overload {
        public static void main(String args[ ]) {
                OverloadDemo ob = new OverloadDemo( );
                double result;

                // call all versions of test( ) - overloaded 4 times
                ob.test( );
                ob.test(10);
                ob.test(10, 20);
                result = ob.test(123.25);
                System.out.println("Result of ob.test(123.25): " + result);
        }
}
```

Q What will be the output?

Overloading Methods

```
// Method overloading Demo
class OverloadDemo {
        void test() {
                System.out.println("No parameters");
        }

        // Overload test for one integer parameter.
        void test(int a) {
                System.out.println("a: " + a);
        }

        // Overload test for two integer parameters.
        void test(int a, int b) {
                System.out.println("a and b: " + a + " " + b);
        }

        // Overload test for a double parameter
        double test(double a) {
                System.out.println("double a: " + a);
                return a*a;
        }
}
```

```
class Overload {
        public static void main(String args[ ]) {
                OverloadDemo ob = new OverloadDemo( );
                double result;

                // call all versions of test( ) - overloaded 4 times
                ob.test( );
                ob.test(10);
                ob.test(10, 20);
                result = ob.test(123.25);
                System.out.println("Result of ob.test(123.25): " + result);
        }
}
```

Q What will be the output?

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

Overloading Methods – Automatic Type Conversions

- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.
- However, this match need not always be exact.
- In some cases, Java's automatic type conversions can play a role in overload resolution.

Byte –> Short –> Int –> Long – > Float –> Double

Widening or Automatic Conversion

Double –> Float –> Long –> Int –> Short –> Byte

Narrowing or Explicit Conversion

Overloading Methods – Automatic Type Conversions

```
// Automatic type conversions apply to overloading.

class OverloadATDemo {
        void test( ) {
                System.out.println("No parameters");
        }

        // Overload test for two integer parameters.
        void test(int a, int b) {
                System.out.println("a and b: " + a + " " + b);
        }

        // Overload test for a double parameter
        void test(double a) {
                System.out.println("Inside test(double) a: " + a);
        }
}
```

Overloading Methods – Automatic Type Conversions

```
// Automatic type conversions apply to overloading.

class OverloadATDemo {
        void test( ) {
                System.out.println("No parameters");
        }

        // Overload test for two integer parameters.
        void test(int a, int b) {
                System.out.println("a and b: " + a + " " + b);
        }

        // Overload test for a double parameter
        void test(double a) {
                System.out.println("Inside test(double) a: " + a);
        }
}
```

```
class Overload {
        public static void main(String args[ ]) {
                OverloadATDemo ob = new OverloadATDemo( );
                int i = 88;
                ob.test( );

                // this will invoke test(int,int)
                ob.test(10, 20);

                // this will invoke test(double)
                ob.test(i);

                // this will invoke test(double)
                ob.test(123.2);
        }
}
```

Overloading Methods – Automatic Type Conversions

```
// Automatic type conversions apply to overloading.

class OverloadATDemo {
        void test( ) {
                System.out.println("No parameters");
        }

        // Overload test for two integer parameters.
        void test(int a, int b) {
                System.out.println("a and b: " + a + " " + b);
        }

        // Overload test for a double parameter
        void test(double a) {
                System.out.println("Inside test(double) a: " + a);
        }
}
```

```
class Overload {
        public static void main(String args[ ]) {
                OverloadATDemo ob = new OverloadATDemo( );
                int i = 88;
                ob.test( );

                // this will invoke test(int,int)
                ob.test(10, 20);

                // this will invoke test(double)
                ob.test(i);

                // this will invoke test(double)
                ob.test(123.2);
        }
}
```

Q **What will be the output?**

Overloading Methods – Automatic Type Conversions

```
// Automatic type conversions apply to overloading.

class OverloadATDemo {
        void test( ) {
                System.out.println("No parameters");
        }

        // Overload test for two integer parameters.
        void test(int a, int b) {
                System.out.println("a and b: " + a + " " + b);
        }

        // Overload test for a double parameter
        void test(double a) {
                System.out.println("Inside test(double) a: " + a);
        }
}
```

```
class Overload {
        public static void main(String args[ ]) {
                OverloadATDemo ob = new OverloadATDemo( );
                int i = 88;
                ob.test( );

                // this will invoke test(int,int)
                ob.test(10, 20);

                // this will invoke test(double)
                ob.test(i);

                // this will invoke test(double)
                ob.test(123.2);
        }
}
```

Q **What will be the output?**

No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2

Overloading Constructors

```
// Here, Box defines three constructors to initialize the dimensions of a box various ways.
class Box {
        double width, height, depth;
        // constructor used when all dimensions specified
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }
        // constructor used when no dimensions specified
        Box( ) {
                width = -1;
                height = -1;
                depth = -1;
        }
        // constructor used when cube is created
        Box(double len) {
                width = height = depth = len;
        }
        // compute and return volume
        double volume( ) {
                return width * height * depth;
        }
}
```

Overloading Constructors

```
// Here, Box defines three constructors to initialize the dimensions of a box various ways.
class Box {
        double width, height, depth;
        // constructor used when all dimensions specified
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }
        // constructor used when no dimensions specified
        Box( ) {
                width = -1;
                height = -1;
                depth = -1;
        }
        // constructor used when cube is created
        Box(double len) {
                width = height = depth = len;
        }
        // compute and return volume
        double volume( ) {
                return width * height * depth;
        }
}
```

```
class OverloadCons {
        public static void main(String args[ ]) {
                // create boxes using the various constructors
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box( );
                Box mycube = new Box(7);

                double vol;

                // get volume of first box
                vol = mybox1.volume( );
                System.out.println("Volume of mybox1 is " + vol);

                // get volume of second box
                vol = mybox2.volume( );
                System.out.println("Volume of mybox2 is " + vol);

                // get volume of cube
                vol = mycube.volume( );
                System.out.println("Volume of mycube is " + vol);
        }
}
```

Overloading Constructors

```
// Here, Box defines three constructors to initialize the dimensions of a box various ways.
class Box {
        double width, height, depth;
        // constructor used when all dimensions specified
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }
        // constructor used when no dimensions specified
        Box( ) {
                width = -1;
                height = -1;
                depth = -1;
        }
        // constructor used when cube is created
        Box(double len) {
                width = height = depth = len;
        }
        // compute and return volume
        double volume( ) {
                return width * height * depth;
        }
}
```

```
class OverloadCons {
        public static void main(String args[ ]) {
                // create boxes using the various constructors
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box( );
                Box mycube = new Box(7);

                double vol;

                // get volume of first box
                vol = mybox1.volume( );
                System.out.println("Volume of mybox1 is " + vol);

                // get volume of second box
                vol = mybox2.volume( );
                System.out.println("Volume of mybox2 is " + vol);

                // get volume of cube
                vol = mycube.volume( );
                System.out.println("Volume of mycube is " + vol);
        }
}
```

Q What will be the output?

Using Objects as Parameters

```
// Objects may be passed to methods.
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
        }

        // return true if o is equal to the invoking object
        boolean equalTo(Test o) {
                if(o.a == a && o.b == b)
                        return true;
                else
                        return false;
        }
}
```

Using Objects as Parameters

```
// Objects may be passed to methods.
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
        }

        // return true if o is equal to the invoking object
        boolean equalTo(Test o) {
                if(o.a == a && o.b == b)
                        return true;
                else
                        return false;
        }
}
```

```
class PassOb {
        public static void main(String args[ ]) {

                Test ob1 = new Test(100, 22);
                Test ob2 = new Test(100, 22);
                Test ob3 = new Test(-1, -1);

                System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
                System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
        }
}
```

Using Objects as Parameters

```
// Objects may be passed to methods.
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
        }

        // return true if o is equal to the invoking object
        boolean equalTo(Test o) {
                if(o.a == a && o.b == b)
                        return true;
                else
                        return false;
        }
}
```

```
class PassOb {
        public static void main(String args[ ]) {

                Test ob1 = new Test(100, 22);
                Test ob2 = new Test(100, 22);
                Test ob3 = new Test(-1, -1);

                System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
                System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
        }
}
```

Q What will be the output?

Using Objects as Parameters

```
// Objects may be passed to methods.
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
        }

        // return true if o is equal to the invoking object
        boolean equalTo(Test o) {
                if(o.a == a && o.b == b)
                        return true;
                else
                        return false;
        }
}
```

```
class PassOb {
        public static void main(String args[ ]) {

                Test ob1 = new Test(100, 22);
                Test ob2 = new Test(100, 22);
                Test ob3 = new Test(-1, -1);

                System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
                System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
        }
}
```

Q What will be the output?

```
ob1 == ob2: true
ob1 == ob3: false
```

Copy Constructor

```
// Here, Box allows one object to initialize another.
class Box {
        double width;
        double height;
        double depth;

        // Constructor taking an object of type Box.
        Box(Box ob) {
                width = ob.width;
                height = ob.height;
                depth = ob.depth;
        }

        // constructor used when all dimensions specified
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }

        // constructor used when no dimensions specified
        Box( ) {
                width = -1;
                height = -1;
                depth = -1;
        }

        // constructor used when cube is created
        Box(double len) {
                width = height = depth = len;
        }

        // compute and return volume
        double volume( ) {
                return width * height * depth;
        }
}
```

Copy Constructor

```
class OverloadCons2 {
    public static void main(String args[ ]) {
        // create boxes using the various constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box( );
        Box mycube = new Box(7);

        // create copy of mybox1
        Box myclone = new Box(mybox1);

        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);

        // get volume of cube
        vol = mycube.volume( );
        System.out.println("Volume of cube is " + vol);

        // get volume of clone
        vol = myclone.volume( );
        System.out.println("Volume of clone is " + vol);
    }
}
```

Q What will be the output?

A Closer Look at Argument Passing

- In general, there are two ways that a computer language can pass an argument to a subroutine.

1. *Call-by-value*
   - Copies the value of an argument into the formal parameter of the subroutine.
   - Changes made to the parameter of the subroutine have no effect on the argument.

2. *Call-by-Reference*
   - A reference to an argument (not the value of the argument) is passed to the parameter.
   - Inside the subroutine, this reference is used to access the actual argument specified in the call.
   - Changes made to the parameter will affect the argument used to call the subroutine.

A Closer Look at Argument Passing

```
// Primitive types are passed by value.
class Test {
        void meth(int i, int j) {
                i *= 2;
                j /= 2;
        }
}

class CallByValue {
        public static void main(String args[ ]) {
                Test ob = new Test();

                int a = 15, b = 20;

                System.out.println("a and b before call: " + a + " " + b);

                ob.meth(a, b);

                System.out.println("a and b after call: " +a + " " + b);
        }
}
```

A Closer Look at Argument Passing

```
// Primitive types are passed by value.
class Test {
        void meth(int i, int j) {
                i *= 2;
                j /= 2;
        }
}

class CallByValue {
        public static void main(String args[ ]) {
                Test ob = new Test();

                int a = 15, b = 20;

                System.out.println("a and b before call: " + a + " " + b);

                ob.meth(a, b);

                System.out.println("a and b after call: " +a + " " + b);
        }
}
```

Q What will be the output?

a and b before call: 15 20
a and b after call: 15 20

A Closer Look at Argument Passing

```
// Objects are passed through their references.
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
        }
        // pass an object
        void meth(Test o) {
                o.a *= 2;
                o.b /= 2;
        }
}
class PassObjRef {
        public static void main(String args[ ]) {
                Test ob = new Test(15, 20);
                System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);

                ob.meth(ob);

                System.out.println("ob.a and ob.b after call: " +ob.a + " " + ob.b);
        }
}
```

A Closer Look at Argument Passing

```
// Objects are passed through their references.
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
        }
        // pass an object
        void meth(Test o) {
                o.a *= 2;
                o.b /= 2;
        }
}
class PassObjRef {
        public static void main(String args[ ]) {
                Test ob = new Test(15, 20);
                System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);

                ob.meth(ob);

                System.out.println("ob.a and ob.b after call: " +ob.a + " " + ob.b);
        }
}
```

Q What will be the output?

ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10

Returning Objects

```
// Returning an object.
class Test {

        int a;

        Test(int i) {
                a = i;
        }

        Test incrByTen( ) {
                Test temp = new Test(a+10);
                return temp;
        }
}
```

```
class RetOb {
        public static void main(String args[ ]) {
                Test ob1 = new Test(2);
                Test ob2;
                ob2 = ob1.incrByTen( );

                System.out.println("ob1.a: " + ob1.a);
                System.out.println("ob2.a: " + ob2.a);

                ob2 = ob2.incrByTen();
                System.out.println("ob2.a after second increase: "+ ob2.a);
        }
}
```

Q What will be the output?

Returning Objects

```
// Returning an object.
class Test {

        int a;

        Test(int i) {
                a = i;
        }

        Test incrByTen( ) {
                Test temp = new Test(a+10);
                return temp;
        }
}
```

```
class RetOb {
        public static void main(String args[ ]) {
                Test ob1 = new Test(2);
                Test ob2;
                ob2 = ob1.incrByTen( );

                System.out.println("ob1.a: " + ob1.a);
                System.out.println("ob2.a: " + ob2.a);

                ob2 = ob2.incrByTen();
                System.out.println("ob2.a after second increase: "+ ob2.a);
        }
}
```

Q What will be the output?

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

Recursion

- Java supports *recursion.*
- Recursion is the process of defining something in terms of itself.
- As it relates to Java programming, recursion is the attribute that allows a method to call itself.
- A method that calls itself is said to be recursive.

Recursion

```
// A simple example of recursion.
class Factorial {
        // this is a recursive method
        int fact(int n) {
                int result;
                if(n==1)
                        return 1;
                result = fact(n-1) * n;
                return result;
        }
}

class Recursion {
        public static void main(String args[ ]) {
                Factorial f = new Factorial( );
                System.out.println("Factorial of 3 is " + f.fact(3));
                System.out.println("Factorial of 4 is " + f.fact(4));
                System.out.println("Factorial of 5 is " + f.fact(5));
        }
}
```

Q What will be the output?

Recursion

```
// A simple example of recursion.
class Factorial {
        // this is a recursive method
        int fact(int n) {
                int result;
                if(n==1)
                        return 1;
                result = fact(n-1) * n;
                return result;
        }
}

class Recursion {
        public static void main(String args[ ]) {
                Factorial f = new Factorial( );
                System.out.println("Factorial of 3 is " + f.fact(3));
                System.out.println("Factorial of 4 is " + f.fact(4));
                System.out.println("Factorial of 5 is " + f.fact(5));
        }
}
```

Q What will be the output?

Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120

Recursion

```
// A simple example of recursion.
class Factorial {
        // this is a recursive method
        int fact(int n) {
                int result;
                if(n==1)
                        return 1;
                result = fact(n-1) * n;
                return result;
        }
}

class Recursion {
        public static void main(String args[ ]) {
                Factorial f = new Factorial( );
                System.out.println("Factorial of 3 is " + f.fact(3));
                System.out.println("Factorial of 4 is " + f.fact(4));
                System.out.println("Factorial of 5 is " + f.fact(5));
        }
}
```

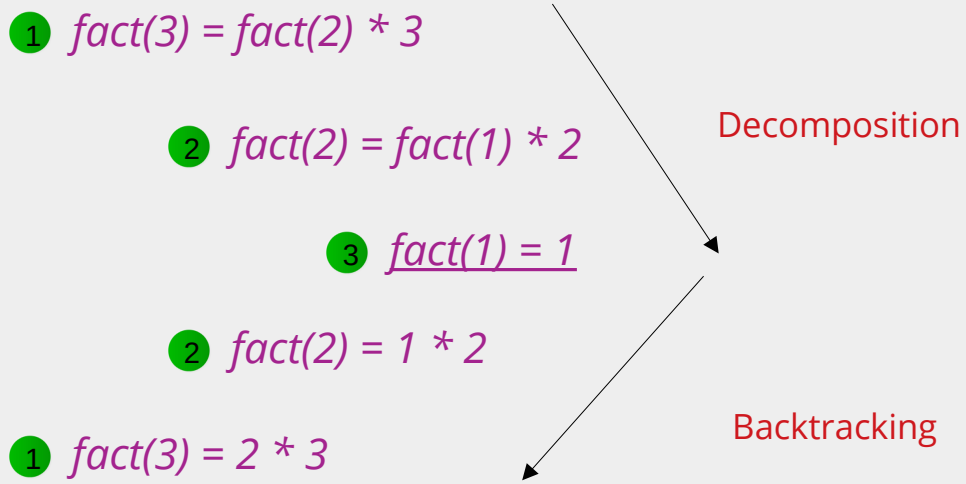
1. *fact(3) = fact(2) * 3*
2. *fact(2) = fact(1) * 2*
3. *fact(1) = 1*

Decomposition

Recursion

```
// A simple example of recursion.
class Factorial {
        // this is a recursive method
        int fact(int n) {
                int result;
                if(n==1)
                        return 1;
                result = fact(n-1) * n;
                return result;
        }
}

class Recursion {
        public static void main(String args[ ]) {
                Factorial f = new Factorial( );
                System.out.println("Factorial of 3 is " + f.fact(3));
                System.out.println("Factorial of 4 is " + f.fact(4));
                System.out.println("Factorial of 5 is " + f.fact(5));
        }
}
```



Recursion

- When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start
- As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method.
- Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional method calls.
- Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted.
- If this occurs, the Java run-time system will cause an exception.

Recursion

- The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives.
- For example: Quick Sort Algorithm, some types of AI-related algorithms
- When writing recursive methods, you must have an if statement somewhere to force the method to return without the recursive call being executed(**base condition**). If you don't do this, once you call the method, it will never return.
- In a nutshell, the **two properties** a recursive method must have:-
  1. *Base Condition: There must be at least one base condition, when met the function stops calling itself recursively.*
  2. *Progressive approach: The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base condition.*

Introducing Access Control

- Access/Visibility specifiers/modifiers/control are the mechanism by which you can precisely control access to the various members of a class.
- How a member can be accessed is determined by the access modifier attached to its declaration.
- Java supplies a rich set of access modifiers. Some aspects of access control are related mostly to inheritance or packages.
- Java's access modifiers are:
  1. *public*
  2. *private*
  3. *protected*
  4. *default (package-private)*

Introducing Access Control

1. ***public***

   The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package

2. ***private***

   The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

3. ***protected***

   The access level of a protected modifier is within the package and outside the package through sub class. If you do not make the sub class, it cannot be accessed from outside the package.

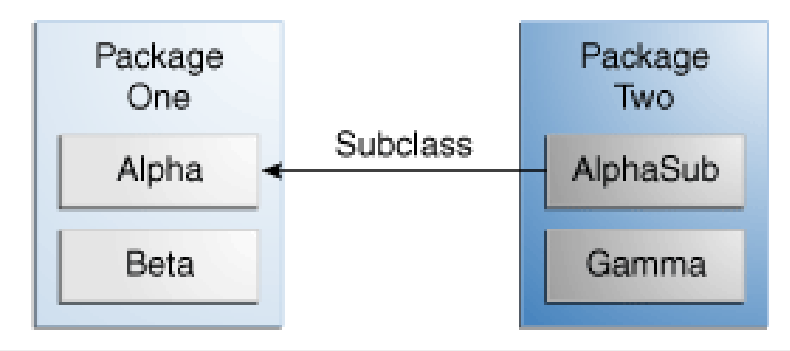
4. ***default (package-private)***

   The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Introducing Access Control

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Introducing Access Control



Classes and Packages used to Illustrate Access Levels

Visibility

Modifier	Alpha	Beta	Alphasub	Gamma
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

The table shows where the **members of the Alpha class are visible** for each of the access modifiers that can be applied to them.

[source: 4]

Introducing Access Control

```
// This program demonstrates the difference between public and private.

class Test {
        int a;                  // default access
        public int b;           // public access
        private int c;          // private access

        // methods to access c
        void setc(int i) {      // set c's value
                c = i;
        }

        int getc( ) {           // get c's value
                return c;
        }
}
```

Introducing Access Control

```
// This program demonstrates the difference between public and private.

class Test {
        int a;                  // default access
        public int b;           // public access
        private int c;          // private access

        // methods to access c
        void setc(int i) {      // set c's value
                c = i;
        }

        int getc( ) {           // get c's value
                return c;
        }
}
```

```
class AccessTest {
        public static void main(String args[ ]) {
                Test ob = new Test();

                // These are OK, a and b may be accessed directly
                ob.a = 10;
                ob.b = 20;

                // This is not OK and will cause an error
                ob.c = 100;             // Error

                // You must access c through its methods
                ob.setc(100);           // OK

                System.out.println("a, b, and c: " + ob.a + " " +
                ob.b + " " + ob.getc( ) );
        }
}
```

Understanding Static

- There will be times when you will want to define a class member that will be used independently of any object of that class
- Normally, a class member must be accessed only in conjunction with an object of its class.
- However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword static.

Understanding Static

- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be static.
- The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist.
- Instance variables declared as static are, essentially, global variables.
- When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

Understanding Static

- Methods declared as static have several restrictions:
  1. They can only directly call other static methods.
  2. They can only directly access static data.
  3. They cannot refer to this or super in any way.

- If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded.

Understanding Static

```
// Demonstrate static variables, methods, and blocks.

class UseStatic {

        static int a = 3;
        static int b;

        static void meth(int x){
                System.out.println("x = " + x);
                System.out.println("a = " + a);
                System.out.println("b = " + b);
        }
        static {
                System.out.println("Static block initialized.");
                b = a * 4;
        }

        public static void main(String args[ ]) {
                meth(42);
        }
}
```

1. As soon as the UseStatic class is loaded, all of the static statements are run.
2. First, a is set to 3, then the static block executes, which prints a message and then initializes b to a*4 or 12.
3. Then main( ) is called, which calls meth( ), passing 42 to x.
4. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Q What will be the output?

Understanding Static

```
// Demonstrate static variables, methods, and blocks.

class UseStatic {

        static int a = 3;
        static int b;

        static void meth(int x){
                System.out.println("x = " + x);
                System.out.println("a = " + a);
                System.out.println("b = " + b);
        }
        static {
                System.out.println("Static block initialized.");
                b = a * 4;
        }

        public static void main(String args[ ]) {
                meth(42);
        }
}
```

1. As soon as the UseStatic class is loaded, all of the static statements are run.
2. First, a is set to 3, then the static block executes, which prints a message and then initializes b to a*4 or 12.
3. Then main( ) is called, which calls meth( ), passing 42 to x.
4. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Q What will be the output?

```
Static block initialized.
x = 42
a = 3
b = 12
```

Understanding Static

```
// Accessing Static members outside the class

class StaticDemo {
        static int a = 42;
        static int b = 99;

        static void callme( ) {
                System.out.println("a = " + a);
        }
}

class StaticByName {
        public static void main(String args[ ]) {
                StaticDemo.callme( );
                System.out.println("b = " +StaticDemo.b );
        }
}
```

Q What will be the output?

Understanding Static

```
// Accessing Static members outside the class

class StaticDemo {
        static int a = 42;
        static int b = 99;

        static void callme( ) {
                System.out.println("a = " + a);
        }
}

class StaticByName {
        public static void main(String args[ ]) {
                StaticDemo.callme( );
                System.out.println("b = " +StaticDemo.b );
        }
}
```

Q What will be the output?

a = 42
b = 99

Introducing Final

- A variable can be declared as final.
- A final variable contents can not be modified, making it, essentially, a constant.
- It must be initialized when it is declared. You can do this in one of two ways:
  1. Assign it a value when it is declared (commonly used).

```
final int FILE_NEW  = 1;          //1.  Can be used as constant in the subsequent parts of your program
final int MAX_MARKS  = 100;    // 2. Common coding convention to use all uppercase identifiers for final fields
final int SPEEDLIMIT=60;
```

  2. Assign it a value within a constructor (blank final variable).

Introducing Final

- In addition to fields, both method parameters and local variables can be declared final.
- Declaring a parameter final prevents it from being changed within the method.
- Declaring a local variable final prevents it from being assigned a value more than once.
- The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables. (Will be discussed in Inheritance)

Nested and Inner Classes

- It is possible to define a class within anther class, such classes are known as nested classes.
- The scope of a nested class is bounded by the scope of its enclosing class.
  - if class B is defined within class A, then B does not exist independently of A.
- A nested class has access to the members, including private members, of the class in which it is nested but the reverse is not true.
- A nested class that is declared directly within its enclosing class scope is a member of its enclosing class.
- It is also possible to declare a nested class that is local to a block.

Nested and Inner Classes

- There are two types of nested classes: static and non-static.

1. *static*
   - A static nested class is one that has the static modifier applied.
   - it must access the non-static members of its enclosing class through an object.
   - it cannot refer to non-static members of its enclosing class directly.
   - it is rarely used (due to above restrictions)

2. *non-static*
   - An inner class is a non-static nested class.
   - It (inner class) has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

Nested and Inner Classes

```
// Demonstration of how to define and use an inner class.

class Outer {
        int outer_x = 100;
        void test( ) {
                Inner inner = new Inner( );
                inner.display( );
        }
        // this is an inner class
        class Inner {
                void display( ) {
                        System.out.println("outer_x = " + outer_x);
                }
        }
}

class InnerClassDemo {
        public static void main(String args[ ]) {
                Outer outer = new Outer( );
                outer.test( );
        }
}
```

Q What will be the output?

Nested and Inner Classes

```
// Demonstration of how to define and use an inner class.

class Outer {
        int outer_x = 100;
        void test( ) {
                Inner inner = new Inner( );
                inner.display( );
        }
        // this is an inner class
        class Inner {
                void display( ) {
                        System.out.println("outer_x = " + outer_x);
                }
        }
}

class InnerClassDemo {
        public static void main(String args[ ]) {
                Outer outer = new Outer( );
                outer.test( );
        }
}
```

Q What will be the output?

outer_x = 100

Nested and Inner Classes

```
// Demonstration of how to define and use an inner class.

class Outer {
        int outer_x = 100;
        void test( ) {
                Inner inner = new Inner( );
                inner.display( );
        }
        // this is an inner class
        class Inner {
                void display( ) {
                        System.out.println("outer_x = " + outer_x);
                }
        }
}

class InnerClassDemo {
        public static void main(String args[ ]) {
                Outer outer = new Outer( );
                outer.test( );
        }
}
```

1. class Inner is defined within the scope of class Outer.
2. Any code in class Inner can directly access the variable **outer_x**.
3. An instance method named display( ) is defined inside Inner.
4. display( ) method displays **outer_x** on the standard output stream.
5. The main( ) method of InnerClassDemo creates an instance of class Outer and invokes its test( ) method.
6. test( ) method creates an instance of class Inner and the display( ) method is called.

Nested and Inner Classes

```
// Demonstration of how to define and use an inner class.

class Outer {
        int outer_x = 100;
        void test( ) {
                Inner inner = new Inner( );
                inner.display( );
        }
        // this is an inner class
        class Inner {
                void display( ) {
                        System.out.println("outer_x = " + outer_x);
                }
        }
}

class InnerClassDemo {
        public static void main(String args[ ]) {
                Outer outer = new Outer( );
                outer.test( );
        }
}
```

7. An instance of Inner can be created only in the context of class Outer.

8. An inner class has access to all of the members of its enclosing class, but the reverse is not true.

9. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class. (as shown in the next example)

Nested and Inner Classes

```
/* This program will not compile. Members of the inner class are known only
within the scope of the inner class and may not be used by the outer class. */
class Outer {
        int outer_x = 100;
        void test( ) {
                Inner inner = new Inner( );
                inner.display( );
        }
        // this is an inner class
        class Inner {
                int y = 10; // y is local to Inner
                void display( ) {
                        System.out.println("outer_x = " + outer_x);
                }
        }
        void showy( ) {
                System.out.println(y); // error, y not known here!
        }
}
class InnerClassDemo {
        public static void main(String args[ ]) {
                Outer outer = new Outer( );
                outer.test();
        }
}
```

Nested and Inner Classes

```
// Define an inner class within a for loop.
class Outer {
        int outer_x = 100;
        void test( ) {
                for(int i=0; i<10; i++) {
                        class Inner {
                                void display( ) {
                                        System.out.println("display: outer_x = " + outer_x);
                                }
                        }
                        Inner inner = new Inner( );
                        inner.display( );
                }
        }
}

class InnerClassDemo {
        public static void main(String args[ ]) {
                Outer outer = new Outer( );
                outer.test();
        }
}
```

1. So far, inner classes declared as members within an outer class scope. (previous examples)
2. It is possible to define inner classes within any block scope.
3. For example, a nested class within the block defined by a method or even within the body of a for loop. (example in this slide)

Q What will be the output?

Exploring the String Class

- String is probably the most commonly used class in the Java's class library.
  - String is an object that represents a sequence of characters.
- The java.lang.String class is used to create a Java string object.
- There are two ways to create a String object:
  1. By string literal: Java String literal is created by using double quotes.

     For example: String university = "Delhi University";

  2. By new keyword: Java String is created by using a keyword "new".

     For example: String college = new String("ANDC, Govindpuri, Kalkaji");
- Java use the cocept of "string constant pool" for efficient memory management while handling strings.

Exploring the String Class

- Important things to remember about strings:
  1. Every string you create is actually an object of type String, including string constants.
     - System.out.println("**This is a String, too**"); // the string "This is a String, too" is a String object.
  2. Objects of type String are immutable; once a String object is created, its contents cannot be altered. (What if you need to change a string?)
     - Create a new string that contains the modification (concatination, substring, replace, etc..)
     - Use peer classes of String, called StringBuffer and StringBuilder, which allows strings to be altered.

Exploring the String Class

```
// Demonstrating Strings.

class StringDemo {

        public static void main(String args[ ]) {

                String strOb1 = "First String";

                String strOb2 = "Second String";

                // + operator is used to concatenate two strings
                String strOb3 = strOb1 + " and " + strOb2;

                System.out.println(strOb1);
                System.out.println(strOb2);
                System.out.println(strOb3);
        }
}
```

Exploring the String Class

```
// Demonstrating Strings.

class StringDemo {

        public static void main(String args[ ]) {

                String strOb1 = "First String";

                String strOb2 = "Second String";

                // + operator is used to concatenate two strings
                String strOb3 = strOb1 + " and " + strOb2;

                System.out.println(strOb1);
                System.out.println(strOb2);
                System.out.println(strOb3);
        }
}
```

Q What will be the output?

First String
Second String
First String and Second String

Exploring the String Class

- The String class contains several methods, some of them are:

1. equals( ) - To test two strings for equality
   boolean equals(secondStr)
   For example: if(strOb1.equals(strOb2)){ ... }

2. length( ) - To obtain the length of a string
   int length( )
   For example: strOb1.length( )

3. charAt( ) - To obtain the character at a specified index within a string
   char charAt(index)
   For example: strOb1.charAt(3)

Exploring the String Class

```
// Demonstrating some String methods.

class StringDemo2 {
        public static void main(String args[ ]) {

                String strOb1 = "Hello";
                String strOb2 = "World!";
                String strOb3 = strOb1;

                System.out.println("Length of strOb1: " + strOb1.length( ));

                System.out.println("Char at index 3 in strOb1: " + strOb1.charAt(3));

                if(strOb1.equals(strOb2))
                        System.out.println("strOb1 == strOb2");
                else
                        System.out.println("strOb1 != strOb2");

                if(strOb1.equals(strOb3))
                        System.out.println("strOb1 == strOb3");
                else
                        System.out.println("strOb1 != strOb3");
        }
}
```

Exploring the String Class

```
// Demonstrating some String methods.

class StringDemo2 {
        public static void main(String args[ ]) {

                String strOb1 = "Hello";
                String strOb2 = "World!";
                String strOb3 = strOb1;

                System.out.println("Length of strOb1: " + strOb1.length( ));

                System.out.println("Char at index 3 in strOb1: " + strOb1.charAt(3));

                if(strOb1.equals(strOb2))
                        System.out.println("strOb1 == strOb2");
                else
                        System.out.println("strOb1 != strOb2");

                if(strOb1.equals(strOb3))
                        System.out.println("strOb1 == strOb3");
                else
                        System.out.println("strOb1 != strOb3");
        }
}
```

Q What will be the output?

```
Length of strOb1: 5
Char at index 3 in strOb1: l
strOb1 != strOb2
strOb1 == strOb3
```

Exploring the String Class

```
// Demonstrate String arrays.

class StringDemo3 {
        public static void main(String args[ ]) {

                String str[ ] = { "ANDC", "Ramanujan", "Deshbandhu" };

                for(int i=0; i<str.length; i++) {
                        System.out.println("str[" + i + "]: " + str[i]);
                }
        }
}
```

Q What will be the output?

```
str[0]: ANDC
str[1]: Ramanujan
str[2]: Deshbandhu
```

Using Command-Line Arguments

```
// Display all command-line arguments.
class CommandLine {
        public static void main(String args[ ]) {
                for(int i=0; i<args.length; i++)
                        System.out.println("args[" + i + "]: " + args[i]);
        }
}
```

Execute the program as:

Java CommandLine Java Python JavaScript Kotlin

Output of the program will be:

```
args[0]: Java
args[1]: Python
args[2]: JavaScript
args[3]: Kotlin
```

References

- **Reference for this topic**

1. **Book-** Java: The Complete Reference, Tenth Edition: Herbert Schildt
2. **Web-** https://www.tutorialspoint.com/java/index.htm
3. **Web-** https://www.javatpoint.com/java-tutorial
4. **Web-** https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html