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.

- the methods are said to be *overloaded,* and

- the process is referred to as *method overloading.*

- Method overloading is one of the ways that Java implements polymorphism (one interface multiple methods).

- overloaded methods must differ in the type and/or number of their parameters.

- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

```
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()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of
ob.test(123.25): " + result);
}
}
```

Output:

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

```
// Automatic type conversions apply to
overloading.
class OverloadDemo {
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[]) {
OverloadDemo ob = new OverloadDemo();
int i = 88;
ob.test();
ob.test(10, 20);
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke
test(double)
}
}
```

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;
double height;
double 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; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// 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);
}
}
```

Output

:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

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

output:

ob1 == ob2: true

ob1 == ob3: false

```
class Box {
double width;
double height;
double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
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;
}
double volume() {
return width * height * depth;
}
}
```

```
class OverloadCons2 {
public static void main(String args[]) {
// create boxes using the various
constructors
Box mybox1 = new Box(10, 20, 15);
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 clone
vol = myclone.volume();
System.out.println("Volume of clone is " +
vol);
}
}
```

Argument Passing

There are two ways that a computer language can pass an argument to a subroutine.

call-by-value.

- This method copies the *value* of an argument into the formal parameter of the subroutine.
- Therefore, changes made to the parameter of the subroutine have no effect on the argument.

call-by-reference.

- In this method, 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.
- The changes made to the parameter will affect the argument used to call the subroutine.

call-by-value

```
// Simple 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);
}
}
```

Output:

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

call-by-reference

```
// Objects are passed by reference.
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 CallByRef {
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);
}
}
```

This program generates the following 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);
}
}
```

Output:

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

Understanding static

- Both methods and variables can be declared as **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.

- Methods declared as **static** have several restrictions:

  - They can only call other **static** methods.
  - They must only access **static** data.
  - They cannot refer to **this** or **super** in any way.

```
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);
}
}
```

Here is the output of the program:

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

- First, **a** is set to **3**, then the **static** block executes (printing a message),
- finally, **b** is initialized to **a * 4** or **12**.
- Then **main( )** is called, which calls **meth( )**, passing **42** to **x**.
- The three **println( )** statements refer to the two **static** variables **a** and **b**, as well as
- to the local variable **x**.