**Overloading Methods**

1. **Method Overloading using add method**
   1. Create multiple add methods
2. **Auto conversion of datatype**
   1. Create method with double but pass int
3. **Overloading Methods**
   1. **Example**

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);

} }

1. **Using Objects as parameters**
   1. **Example**

// 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));

}

}

* 1. **In Box class**

class Box {

double width;

double height;

double depth;

// Notice this constructor. It takes an object of type Box.

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;

}

// 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 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);

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

} }

**Argument passing**

1. **Call by value**
   1. In Java, when you pass a primitive type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:
   2. Example:

// 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);

} }

1. **Call by reference**
   1. When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.
   2. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
   3. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument. For example, consider the following program:
   4. Example:

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);

} }

1. **Returning Objects**
   1. A method can return any type of data, including class types that you create. For example, in the following program, the incrByTen( ) method returns an object in which the value of a is ten greater than it is in the invoking object.
   2. Example:

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);

} }

1. **Recursion**
   1. 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.
   2. Example: Factorial

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. **Introducing final**
   1. A variable can be declared as final. Doing so prevents its contents from being modified. This means that you must initialize a final variable when it is declared. For example:

final int FILE\_NEW = 1;

final int FILE\_OPEN = 2;

final int FILE\_SAVE = 3;

final int FILE\_SAVEAS = 4;

final int FILE\_QUIT = 5;

* 1. Subsequent parts of your program can now use FILE\_OPEN, etc., as if they were constants, without fear that a value has been changed.
  2. It is a common coding convention to choose all uppercase identifiers for final variables. Variables declared as final do not occupy memory on a per-instance basis. Thus, a final variable is essentially a constant.
  3. The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This second usage of final is described in the next chapter, when inheritance is described.

1. **Understanding static**
   1. 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.
   2. To create such a member, precede its declaration with the keyword 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.
   3. Methods declared as static have several restrictions:
      1. They can only call other static methods.
      2. They must only access static data.
      3. They cannot refer to this or super in any way. (The keyword super relates to inheritance
   4. 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. The following example shows a class that has a static method, some static variables, and a static initialization block:

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

1. **Introducing Nested and Inner Classes**
   1. It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A.
   2. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class. 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.
   3. There are two types of nested classes: static and non-static. A static nested class is one that has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.
   4. The most important type of nested class is the inner class. An inner class is a non-static nested class. It 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.
   5. The following program illustrates how to define and use an inner class. The class named Outer has one instance variable named outer\_x, one instance method named test( ), and defines one inner class called Inner.
   6. Example:

// Demonstrate 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("display: outer\_x = " + outer\_x);

}

} }

class InnerClassDemo {

public static void main(String args[]) {

Outer outer = new Outer();

outer.test();

}

}