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.

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.

In some cases, Java's automatic type conversions can play a role in overload resolution.

```
class OverloadDemo {
   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(i);       // this will invoke test(double)
       ob.test(123.2);    // this will invoke test(double)
    }
}
```

As you can see, this version of OverloadDemo does not define test(int). Therefore, when test( ) is called with an

integer argument inside Overload, no matching method is found. However, Java can automatically convert an integer

into a double, and this conversion can be used to resolve the call. Therefore, after test(int) is not found,

Java elevates i to double and then calls test(double).

Of course, if test(int) had been defined, it would have been called instead.

Java will employ its automatic type conversions only if no exact match is found.

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);
    }
}
```

Output:

ob1.a: 2

ob2.a: 12

As you can see, each time incrByTen( ) is invoked, a new object is created, and a reference to it is returned to the

calling routine. Since all objects are dynamically allocated using new, you don’t need to worry about an object going

out-of-scope because the method in which it was created terminates. The object will continue to exist as long as there

is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the

next time garbage collection takes place.