**Revisiting Overloading Methods: Order of Precedence**

Now that we've learned about wrapper classes, we can discuss one more aspect about overloading methods: how a method is selected when multiple options could potentially work.  Specifically, when invoking a method with a primitive argument, it will use the following order of precedence when trying to find a matching method:

1. The exact primitive type of the argument
2. If the argument is a byte, short, char, int, or long:
   1. The next larger size whole number primitive available (byte, short, int, and long)
   2. The smallest size decimal primitive available (float, double)
3. If the argument is a float:
   1. A double primitive
4. A wrapper class type (has to be an exact match)
5. Varargs (exact primitive type)
6. If the argument is a byte, short, char, int, or long:
   1. The next larger size whole number varargs available (byte, short, int, and long)
   2. The smallest size decimal varargs available (float, double)
7. If the argument is a float:
   1. A double varargs
8. Varargs (exact Wrapper class)

For Example:

public class MethodPicker {

// Method A

public void pickMe(int x) {

System.out.println ("int");

}

// Method B

public void pickMe(long x) {

System.out.println ("long");

}

// Method C

public void pickMe(Integer x) {

System.out.println ("Integer");

}

// Method D

public void pickMe(int... x) {

System.out.println ("int...");

}

// Method E

public void pickMe(long... x) {

System.out.println ("long...");

}

public static void main(String[] args) {

MethodPicker mp = new MethodPicker();

int x = 5;

mp.pickMe(x);

}

}

... the code "mp.pickMe(x)" would choose method A (exact primitive type) first, but if that wasn't there, it would then choose method B (a larger primitive type).  Likewise if both A and B weren't there, it would then choose method C (the exact wrapper type).  Method D (varargs of the exact primitive type) would be the next option available, followed by method E (varargs of a larger primitive type).

If method C was written as:

public void pickMe(Long x) {

}

... it would not be called since it's a Long and the primitive was an int (it will only match wrappers that are the exact type of the primitive).

**What about an array?**

Although varargs and arrays are accessed similarly, their creation from method arguments is different.  Specifically, an array parameter is not automatically created from a one or more values... therefore, this method:

public void pickMe(int[] x) {

}

...would never be invoked from a single value  mp.pickMe(5);

or from multiple, such as values mp.pickMe(5,2,6,10); either.  You would need to pass in an actual array for a method with an array parameter to be invoked.

**One final rule**

An array and varargs of the same type are treated as if they are same parameter.  In other words, having the following two methods appear together in the same class is illegal and would not compile:

public void pickMe(int[] myArray) {

}

public void pickMe(int... myVarargs) {

}