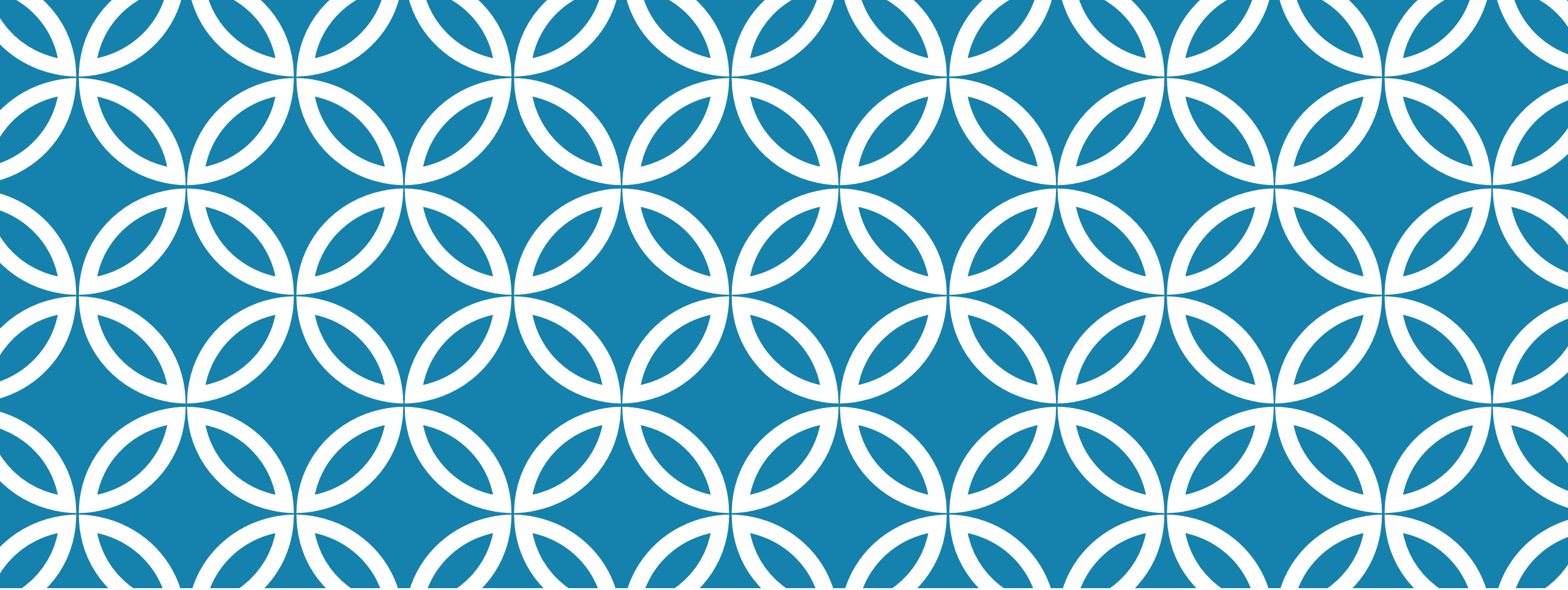

CHAPTER 7 METHODS AND CLASSES

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. When this is the case 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 supports polymorphism.

```
// Demonstrate method overloading.
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);
  }
}
```

This program generates the following output:

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

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

Refer next slide for program:

When test() is called with an integer argument inside overload, no matching method is found. Then java elevates i to double and then call test(double)

```
// 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 and
return type
  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)
  }
}
```

This program generates the following output:

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

Overloading Constructors

In addition to overloading normal methods, you can also overload constructor methods. Consider following example,

```
/* 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);
  }
}
```

The output produced by this program is shown here:

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

As you can see, the proper overloaded constructor is called based upon the parameters specified when **new** is executed.

USING OBJECTS AS PARAMETERS

We can pass objects as parameters to methods,

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

This program generates the following output:

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

We can construct a new object so that it is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter. Consider the following example,

```
// Here, Box allows one object to initialize another.

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);
    }
}
```

A closer look at argument passing

In java when you pass a primitive type to a method, It is passed by value

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

The output from this program is shown here:

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

Objects are passed to methods by use of 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
```

REMEMBER

When a primitive type is passed to a method, it is done by use of call-by-value.

Objects are implicitly passed by use of call-by-reference.

RETURNING OBJECTS

A METHOD CAN RETURN OBJECT OF A CLASS

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

The output generated by this program is shown here:

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

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.

RECURSION

Recursion is the process of defining something in terms of itself.

A methods that calls itself is said to be recursive. Here is how a factorial can be computed by use of a recursive method.

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

The output from this program is shown here:

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

INTRODUCING ACCESS CONTROL

Java's access specifiers are

- Public
- private
- protected

Java also defines a default access level.

protected applies only when inheritance is involved.

When a member of a class is public, then that member cab be accessed by any other code.

When member of a class is specified as private, then that member can only be accessed by other members of its class.

When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

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

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.

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

Here is the output of the program:

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

Here is an example. Inside **main( )**, the **static** method **callme( )** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
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);
  }
}
```

Here is the output of this program:

```
a = 42
b = 99
```

INTRODUCING FINAL

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;
```

Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed.

INTRODUCING NESTED AND INNER CLASSES

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.

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.

The most important type of nested class is the inner class. An inner class is a non-static nested class.

```
// 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();
  }
}
```

Output from this application is shown here:

```
display: outer_x = 100
```

Consider second example,

```
// This program will not compile.
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("display: 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();
  }
}
```

Consider next example,

```
// 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();
  }
}
```

The output from this version of the program is shown here.

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer x = 100
```

STATIC NESTED CLASSES

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

Exploring the String Class

- every string you create is actually an object of type **String**
- Even string constants are actually **String** objects.
- For example, in the Statement

  System.out.println("This is a String, too");

- Strings can be constructed in a variety of ways:
  - String myString = "this is a test";

- Java defines one operator for **String** objects: **+**. It is used to concatenate two strings.
  - String myString = "I" + " like " + "Java.";

```
// Demonstrating Strings.
class StringDemo {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1 + " and " + strOb2;

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

Exploring the String Class

- **String** class contains several methods that you can use:

  boolean equals(String *object*)

  int length( )

  char charAt(int *index*)

```
// Demonstrating some String methods.
class StringDemo2 {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    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");
  }
}
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

```
// Demonstrate String arrays.
class StringDemo3 {
  public static void main(String args[]) {
    String str[] = { "one", "two", "three" };

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

Here is the output from this program:

```
str[0]: one
str[1]: two
str[2]: three
```

USING COMMAND LINE ARGUMENTS

- Sometimes you will want to pass information into a program when you run it.
- This is accomplished by passing *command-line arguments* to **main( )**.
- A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy. they are stored as strings in a **String** array passed to the **args** parameter of **main( )**.
- The first command-line argument is stored at **args[0]**, the second at **args[1]**, and so on.
- For example, the following program displays all of the command-line arguments that it is called with:

```
// 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]);
  }
}
```

Try executing this program, as shown here:

> java CommandLine this is a test 100 -1

When you do, you will see the following output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

VARARGS: VARIABLE LENGTH ARGUMENTS

- Beginning with JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments.
- This feature is called *varargs* and it is short for *variable-length arguments*.
- A method that takes a variable number of arguments is called a *variable-arity method*, or simply a *varargs method*.

```
// Use an array to pass a variable
// number of arguments to a method.

class PassArray {
  static void vaTest(int v[]) {
    System.out.print("Number of args: "
                     + v.length + " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }
  public static void main(String args[])
  {
    // Notice how an array must be
// created to hold the arguments.
    int n1[] = { 10 };
    int n2[] = { 1, 2, 3 };
    int n3[] = { };

    vaTest(n1); // 1 arg
    vaTest(n2); // 3 args
    vaTest(n3); // no args
  }
}
```

The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

```
// Demonstrate variable-length
// arguments.
class VarArgs {

  // vaTest() now uses a vararg.
  static void vaTest(int ... v) {
    System.out.print("Number of args: "
        + v.length + " Contents: ");

    for(int x : v)
      System.out.print(x + " ");
     System.out.println();
  }

public static void main(String args[])
  {

    // Notice how vaTest() can be
called with a
    // variable number of arguments.
    vaTest(10);      // 1 arg
    vaTest(1, 2, 3); // 3 args
    vaTest();        // no args
  }
}
```

The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

Remember:

- A method can have “normal” parameters along with a variable-length parameter.
- However, the ***variable-length parameter must be the last parameter*** declared by the method.
- For example, this method declaration is perfectly acceptable:

```
int doIt(int a, int b, double c, int ... vals) {
```

- There is one more restriction to be aware of: ***there must be only one varargs parameter***.

Overloading Vararg Methods

```
// Varargs and overloading.
class VarArgs3 {

  static void vaTest(int ... v) {
    System.out.print("vaTest(int ...): " +
                     "Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
                     "Number of args: " + v.length +
                     " Contents: ");
```

```
    for(boolean x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(String msg, int ... v) {
    System.out.print("vaTest(String, int ...): " +
                     msg + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }
```

```
  public static void main(String args[])
  {
    vaTest(1, 2, 3);
    vaTest("Testing: ", 10, 20);
    vaTest(true, false, false);
  }
}
```

The output produced by this program is shown here:

```
vaTest(int ...): Number of args: 3 Contents: 1 2 3
vaTest(String, int ...): Testing: 2 Contents: 10 20
vaTest(boolean ...) Number of args: 3 Contents: true false false
```

VARARGS AND AMBIGUITY

```
// Varargs, overloading, and ambiguity.
//
// This program contains an error and will
// not compile!
class VarArgs4 {

  static void vaTest(int ... v) {
    System.out.print("vaTest(int ...): " +
                     "Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }
```

```
  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
                     "Number of args: " + v.length +
                     " Contents: ");

    for(boolean x : v)
      System.out.print(x + " ");

    System.out.println();
  }


  public static void main(String args[])
  {
    vaTest(1, 2, 3);  // OK
    vaTest(true, false, false); // OK

    vaTest(); // Error: Ambiguous!
  }
}
```

In this program, the overloading of **vaTest( )** is perfectly correct. However, this program will not compile because of the following call:

vaTest(); // Error: Ambiguous!

Because the vararg parameter can be empty, this call could be translated into a call to **vaTest(int ...)** or **vaTest(boolean ...)**. Both are equally valid. Thus, the call is inherently ambiguous.

Here is another example of ambiguity.
The following overloaded versions of **vaTest( )** are inherently ambiguous even though one takes a normal parameter:

```
static void vaTest(int ... v) { // ...

static void vaTest(int n, int ... v) { // ...
```

Although the parameter lists of **vaTest( )** differ, there is no way for the compiler to resolve the following call:

```
vaTest(1)
```

DISCLAIMER

- These slides are not original and have been prepared from various sources for teaching purpose.

Sources:

- Herbert Schildt, Java™: The Complete Reference

Thank You