JAVA Methods and Classes

3C Section
Department of CSE, BMSCE
Prepared by
Syed Akram

Method overloading

- Methods of the **same name can be declared in the same class**, as long as they have different sets of parameters (determined by the number, types and order of the parameters) – this is called method overloading.
- When an overloaded method is called, the Java compiler selects the appropriate method by examining the number, types and order of the arguments in the call.
- Method overloading is commonly used to create several methods with the same name that perform the same or similar tasks, but on different types or different numbers of arguments.

- For example, Math methods abs, min and max are overloaded with four versions each:

1. One with two double parameters.
2. One with two float parameters.
3. One with two int parameters.
4. One with two long parameters.

```
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;
    }
  }
```

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

```
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);
    }
}
```

```
// 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 Overload2 {
      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
```

Constructor Overloading

```
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;
    }
    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 Parameter:

```
// 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
```

```
// Here, Box allows one object to initialize another.
class Box {
        double width;
        double height;
        double depth;
        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;
        }
```

```
        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;
        }
}  // End of class
```

```
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);
        }
}
```

```
public class Overloading
{
    int square(int intVal)
    {
        System.out.println("Called square with argument: "+intVal);
        return intVal*intVal;
    }
    double square(double doubleVal)
    {
        System.out.println("Called square with argument: "+doubleVal);
        return doubleVal*doubleVal;
    }
}
```

```
public class OverloadingTest
{
    public static void main(String[] args)
    {
        Overloading first=new Overloading();
        System.out.println("Square of integer 8 equals to "+first.square(8));
        System.out.println("Square of double 8.5 equals to "+first.square(8.5));
    }
}
```

```
run:
Called square with argument: 8
Square of integer 8 equals to 64
Called square with argument: 8.5
Square of double 8.5 equals to 72.25
BUILD SUCCESSFUL (total time: 3 seconds)
```

- The compiler distinguishes overloaded methods by their signature –
  - a combination of the method’s name and the number, types and order of its parameters.
- If the compiler looked only at method names during compilation
  - the code in previous example would be ambiguous.
- Internally, the compiler uses longer method names that include the original method name, the types of each parameter and the exact order of the parameters to determine whether the methods in a class are unique in that class.
- Overloaded method calls cannot be distinguished by return type.

- Overloaded method declarations with identical signatures cause compilation errors, even if the return types are different.
- That is understandable, because the return type is not necessarily apparent when you call a method.

- Constructor Overloading
- **Using Objects as Parameters**
- **Call by Value & Call by reference**
  - ***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**
- **Recursion**
- **Access Control**

Call 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);
        }
}
```

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

Call by Reference

```
//Call 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)
        }
}
```

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
```

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());
    }
}
// ob.c = 100; // Error! In this line
```

Understanding static

- Static means “pertaining to the class in general”, *not* to an individual object
- If you 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.
  - 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.**
- 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.**
- **Variable**

  static int num;
- **If we wish to call a method from outside**
  - *classname.method( )*

```
public class JustAdd {
    int x;
    int y;
    int z;

    public static void main(String args[]) {
        x = 5;
        y = 10;
        z = x + y;
    }
}
```

all are wrong

```
class StaticDemo {

  int a, b;

  StaticDemo(int i, int j) {
   a=i;b=j;
  }

  void disp() {
      System.out.println(a+"---"+b);
  }

  public static void main(String args[]) {
       disp();
       System.out.println(a+"-"+b);

  }
}
```

```
StaticDemo.java:14: error: non-static method disp() cannot be referenced from a static context
                disp();
                ^
StaticDemo.java:15: error: non-static variable a cannot be referenced from a static context
                System.out.println(a+"-"+b);
                                   ^
StaticDemo.java:15: error: non-static variable b cannot be referenced from a static context
                System.out.println(a+"-"+b);
```

```
class StaticDemo {

  int a, b;

  StaticDemo(int i, int j) {
   a=i;b=j;
  }

  void disp() {
      System.out.println(a+"---"+b);
  }

  public static void main(String args[]) {
       StaticDemo obj = new StaticDemo(10,20);
       obj.disp();
       System.out.println(obj.a+"-"+obj.b);

  }
}
```

```
10—20
10-20
```

```
class StaticDemo {

  int a, b;

  StaticDemo(int i, int j) {
   a=i;b=j;
  }

  void disp() {
      System.out.println(a+"---"+b);
  }

  public static void main(String args[]) {
       StaticDemo obj = new StaticDemo(10,20);
       obj.disp();
       System.out.println(a+"-"+b);

  }
}
```

```
StaticDemo.java:16: error: non-static variable b cannot be referenced from a static context
                System.out.println(a+"-"+b);
```

```
class StaticDemo {

  int a, b;
  static int st = 100;
  StaticDemo(int i, int j) {
   a=i;b=j;
  }

  void disp() {
      System.out.println(a+"---"+b);
  }

  public static void main(String args[]) {
       StaticDemo obj = new StaticDemo(10,20);
       obj.disp();
       System.out.println("Static Variable:" +StaticDemo.st);

  }
}
```

```
10—20
Static Variable: 100
```

```
class StaticDemo {

  int a, b;
  static int st = 100;
  StaticDemo(int i, int j) {
   a=i;b=j;
  }

  void disp() {
      System.out.println(a+"---"+b);
  }

  public static void main(String args[]) {
       StaticDemo obj = new StaticDemo(10,20);
       obj.disp();
       System.out.println("Static Variable:" +st);

  }
}
```

```
10—20
Static Variable: 100
```

```
class Demo {

  int a, b;
  static int st = 100;
  Demo(int i, int j) {
   a=i;b=j;
  }

  void disp() {
      System.out.println(a+"---"+b);
  }
}

class StaticDemo{

  public static void main(String args[]) {
       Demo obj = new Demo(10,20);
       obj.disp();
       System.out.println("Static Variable:" +Demo.st);

  }
}
```

```
10—20
Static Variable: 100
```

```
public class JustAdd {
  int x;
  int y;
  int z;

  public static void main(String args[]) {
      JustAdd myAdd = new JustAdd()
      myAdd.doItAll()
    }

  void doItAll() {
    x = 5;
    y = 10;
    z = x + y;
  }
}
```

```
public class Main {
   public static void main( String[] args ) {

      // accessing the methods of the Math class
      System.out.println("Absolute value of -12 =  " +Math.abs(-12));
      System.out.println("Value of PI = " + Math.PI);
      System.out.println("Value of E = " + Math.E);
      System.out.println("2^2 = " + Math.pow(2,2));
   }
}
```

```
Absolute value of -12 = 12
Value of PI = 3.141592653589793
Value of E = 2.718281828459045
2^2 = 4.0
```

```
class StaticTest {

    // non-static method
    int multiply(int a, int b){
        return a * b;
    }

    // static method
    static int add(int a, int b){
        return a + b;
    }
}
```

```
public class Main {

  public static void main( String[] args ) {

      // create an instance of the StaticTest class
      StaticTest st = new StaticTest();

      // call the nonstatic method
      System.out.println(" 2 * 2 = " + st.multiply(2,2));

      // call the static method
      System.out.println(" 2 + 3 = " + StaticTest.add(2,3));
  }
}
```

```
2 * 2 = 4
2 + 3 = 5
```

Static Rules

- *static* variables and methods belong to the class in general, not to individual objects
- *The absence* of the keyword *static* before non-local variables and methods means *dynamic* (one per object/instance)
- A dynamic method can access all dynamic *and* static variables and methods in the same class
- A static method can not access a dynamic variable *(How could it choose or which one?)*
- A static method can not call a dynamic method (because it might access an instance variable)

Introducing final

- Final variable prevents from modifying the variable.
- coding convention to choose all uppercase identifiers for **final variables.**
- **final do not occupy memory on a per-instance basis.**
- For Eg.

```
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;
```

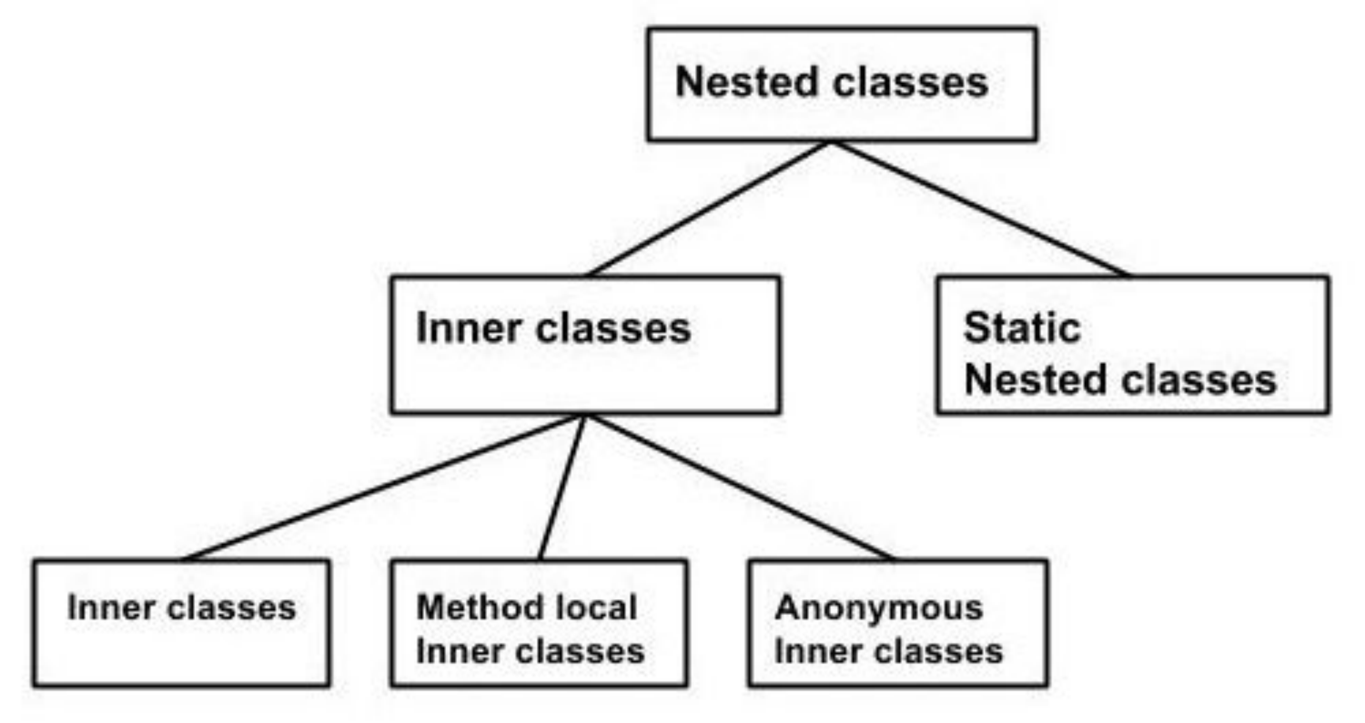
Array Revisited

```
// This program demonstrates the length array member.
class Length {
      public static void main(String args[]) {
        int a1[] = new int[10];
        int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
        int a3[] = {4, 3, 2, 1};
        System.out.println("length of a1 is " + a1.length);
        System.out.println("length of a2 is " + a2.length);
        System.out.println("length of a3 is " + a3.length);
      }
}
```

OUTPUT:

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

Introducing Nested and Inner Classes



```
class TestMemberOuter {
    private int data=30;
    class Inner{
      void msg(){
           System.out.println("data is "+data);
      }
    }

    public static void main(String args[]){
      TestMemberOuter obj=new TestMemberOuter();
      TestMemberOuter.Inner in=obj.new Inner();
      in.msg();
    }
}
```

```
data is 30
```

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

```
OUTPUT:
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
```

```java
public class OuterClassMethodDemo {
  // instance method of the outer class
  void my_Method() {
    int num = 23;

    // method-local inner class
    class MethodInner_Demo {
      public void print() {
        System.out.println("This is method inner class "+num);
      }
    } // end of inner class

    // Accessing the inner class
    MethodInner_Demo inner = new MethodInner_Demo();
    inner.print();
  }

  public static void main(String args[]) {
    OuterClassMethodDemo outer = new OuterClassMethodDemo();
    outer.my_Method();
  }
}
```

```
This is method inner class 23
```

```
abstract class AnonymousInner {
  public abstract void mymethod();
}

public class Outer_class {

  public static void main(String args[]) {
    AnonymousInner inner = new AnonymousInner() {
      public void mymethod() {
        System.out.println("This is an example of anonymous inner class");
      }
    };
    inner.mymethod();
  }
}
```

```
This is an example of anonymous inner class
```

References