Module 1

Overloading methods

Overloading constructors

Using objects as parameters

Argument Passing

Access Control

Understanding Static

Nested and Inner Classes

Overloading Methods

- define **two or more methods within the same class that share the same name, as long as their parameter declarations are different.**

- 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**.

- When an overloaded method is invoked, Java uses **the type and/or number of arguments** as its guide to determine which version of the overloaded method to actually call.

- Thus, overloaded methods **must differ in the type and/or number of their parameters.**

- While overloaded methods may have different return types, **the return type alone is insufficient to distinguish two versions of a method**.

```
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);
  } }
```

```
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);
    ob.test(123.2);
  }
}
```

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

Overloading constructors

```
class Box {
  double width;
  double height;
  double depth;
    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
  }
    Box(double len) {
    width = height = depth = len;
  }
   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;

    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);

    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
  }
}
```

Using Objects as Parameters

```
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));
  }
}
```

```
class Box {
  double width;
  double height;
  double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }
// all other constructors same as before
```

```
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);

    double vol;
  ...
}
}
```

A Closer Look at Argument Passing

- **call-by-value:** This approach copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

- **call-by-reference:** In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

```
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);
  }
}
```

```
class Test {
  int a, b;
  Test(int i, int j) {
    a = i;
    b = j;
  }
    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);
  } }
```

- 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

Access Control

- Java's access specifiers are **public, private, and protected.**
- Java also defines a **default access level**.
- **protected applies only when inheritance is involved**

- *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*

```
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();

    ob.a = 10;
    ob.b = 20;

     ob.c = 100;

    ob.setc(100);

    System.out.println("a, b, and c: " + ob.a + "
" +   ob.b + " " + ob.getc());
  }
}
```

Understanding 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.**
- declare both methods and variables to be static.
- The most common example of a static member is main( ).
- 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

- 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.**

```
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);          }
}
```

```
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);
  } }
```

Outside of the class in which they are defined, static methods and variables can be used independently of any object.
to call a static method from outside its class,

classname.method( )

Here, *classname is the name of the class in which the static method is declared*

Arrays Revisited

there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its **length instance variable**. All arrays have this variable, and it will always hold the size of the array.

```
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);
  }   }
```

Nested and Inner Classes

- 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.
- ***A nested class has access to the members, including private members, of the class in which it is nested***.
- ***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.

- 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.

- 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.

```
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // this is an innner 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();
  }
}
```

```
class Outer {
  int outer_x = 100;
  void test() {
    Inner inner = new Inner();
    inner.display();
  }
    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);
  } }

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  } }
```

```
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();
  }   }
```

WAP to implement a Stack for integers