Module 61CSE215: Object-oriented Programming with Java

4. Object – Class – Method

Vietnamese-German University
Ngoc Tran, Ph.D.
ngoc.th@vgu.edu.vn

Binh Duong, 2025

Content

- Class
- Object
- Method

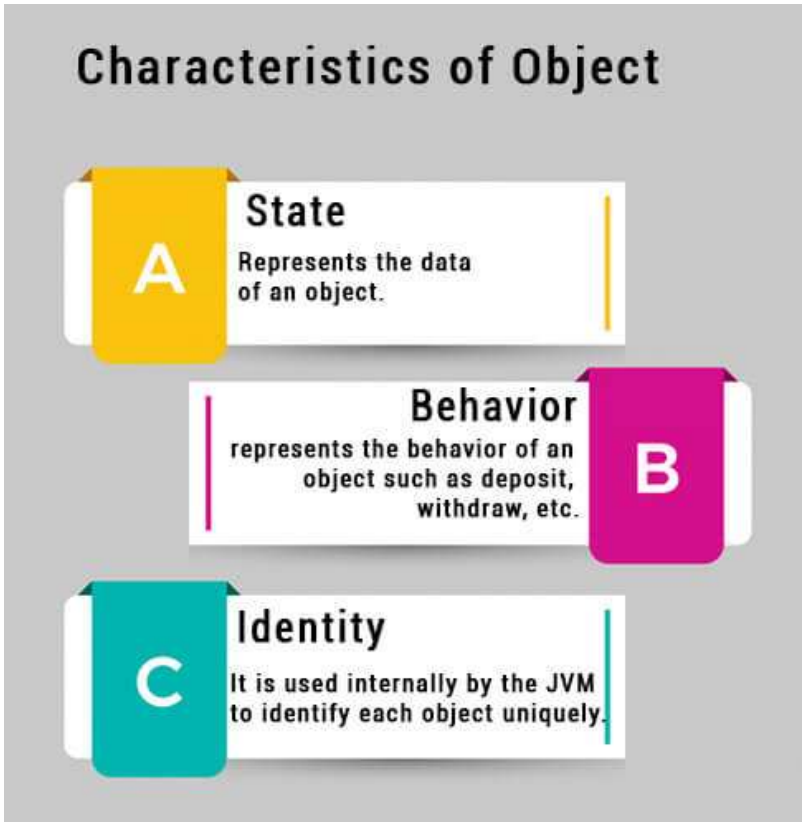
Object

- An entity that has state and behavior is known as an object
- It can be physical (tangible) or logical (intangible).
  - Examples of tangible objects are person, chair, bike, marker, table.
  - An example of an intangible object is the banking system.

Object

- An object has three characteristics: state, behavior and identity.
- E.g., dog is an object.

  - States: Dog has name Pie, her breed is Corgi, her size is small, her color is red and white and she is 1 year old.
  - Behaviors: eat, sleep, sit, run.



Source: Javatpoint.com

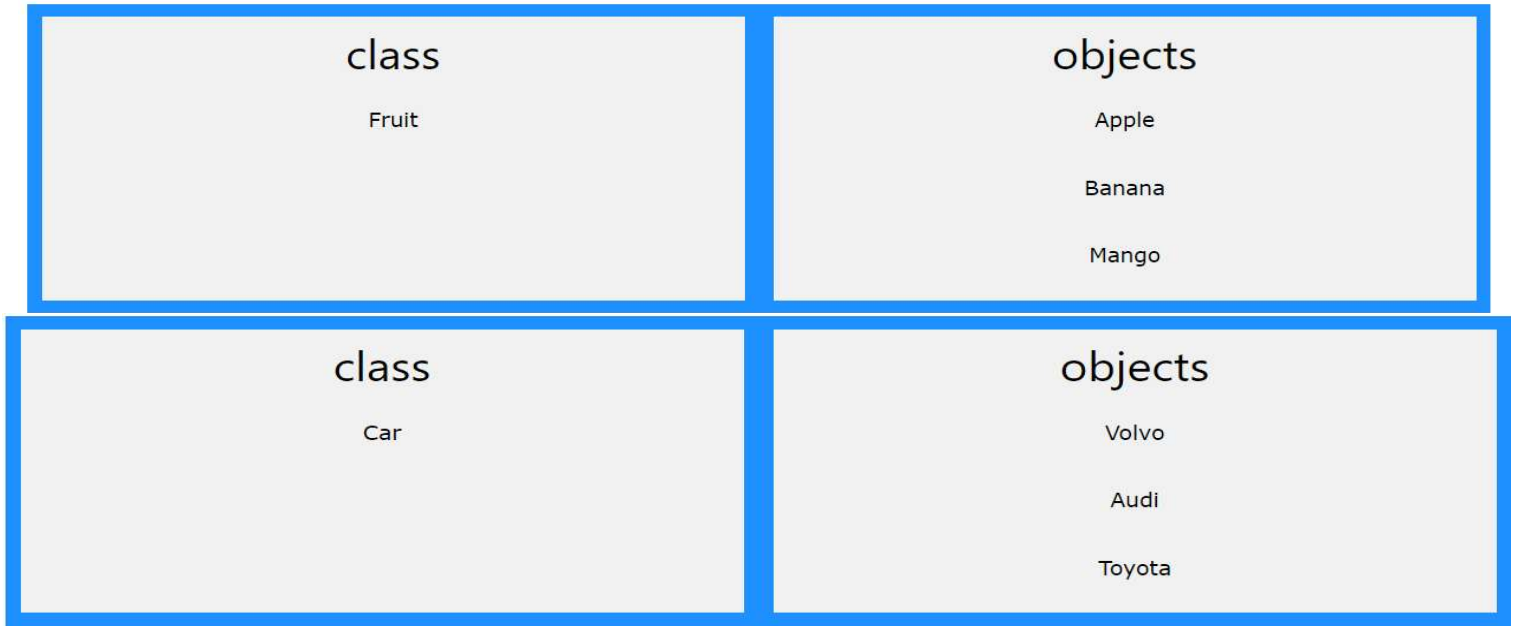
Class

- A class is a template or blueprint from which objects are created.
- Class is a basic concept of OOP which revolve around the real-life entities.
- Class in Java determines how an object will behave and what the object will contain.
- Syntax

```
class <class_name>{
        fields;
        methods;
}
```

Class vs Object

- A class is a template for objects, and an object is an instance of a class. Or, a class is a logical construct; an object has physical reality.
- When the individual objects are created, they inherit all the variables and methods from the class.



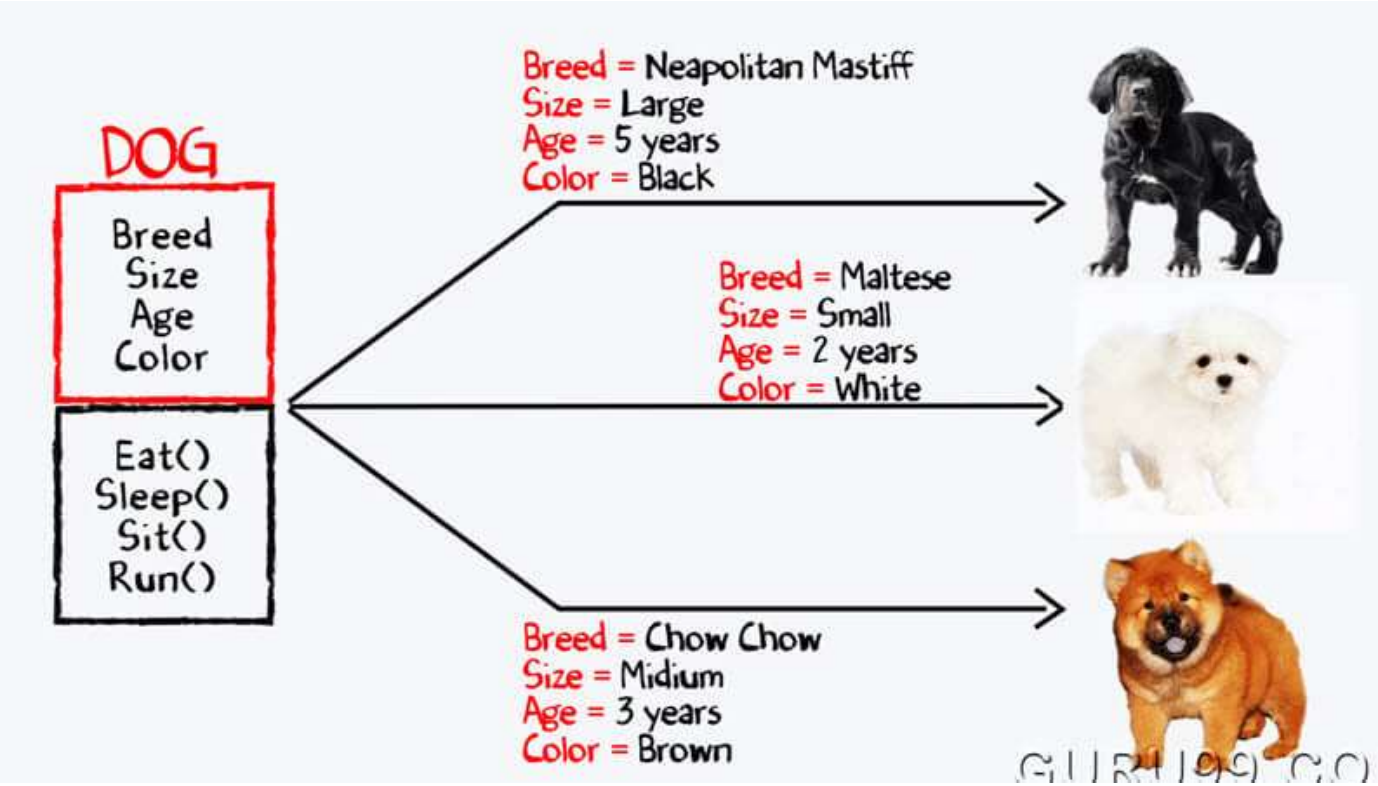
Class vs Object

- Object is an instance of a class.
- Syntax to declare an object.

  ClassName ReferenceVariable = new ClassName();

- To create an object:
  - Defining a class describing the common features of all objects in the same classification of the requested object.
  - Declaring the object with the defined class.

Example


DOG
Breed
Size
Age
Color
Eat()
Sleep()
Sit()
Run()
Breed = Neapolitan Mastiff
Size = Large
Age = 5 years
Color = Black
Breed = Maltese
Size = Small
Age = 2 years
Color = White
Breed = Chow Chow
Size = Midium
Age = 3 years
Color = Brown

Class

- The class is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.
- The class forms the basis for object-oriented programming in Java.
- Any concept you wish to implement in a Java program must be encapsulated within a class.

Class & Object

- Class defines a new data type.
- These new types can be used to create objects of that type.
- A class is a template for an object, and an object is an instance of a class.
- As an object is an instance of a class, you will often see the two words object and instance used interchangeably.

General Form of a Class

- To define a class, specifying the data (attributes/variables) and the code (methods/functions) that operates on that data.
- Some simple classes contain code or data, most real-world classes contain both.
- As you will see, a class' code defines the interface to its data.

Class

- The data, or variables, defined within a class are called instance variables.
  - Data for one object is separate and unique from the data for another.
- The code is contained within methods.

```
class classname {
   type instance-variable1;
   type instance-variable2;
   // ...
   type instance-variableN;

   type methodname1(parameter-list) {
    // body of method
   }
   type methodname2(parameter-list) {
    // body of method
   }
   // ...
   type methodnameN(parameter-list) {
      // body of method
   }
}
```

Class

- The methods and variables defined within a class are called members of the class.
- The instance variables are acted upon and accessed by the methods defined for that class.
- Data for one object is separate and unique from the data for another.
- Most methods will not be specified as static or public.

A Simple Class

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

- A class defines a new type of data called **Box**.
- Use this name to declare objects of type **Box**.

```
public class BoxDemo {
    public static void main(String[] args) {
        Box mybox = new Box();
    }
}
```

Declaring Objects

- To obtain objects of a class is a two-step process.
  - First, you must declare a variable of the class type.
  - Second, you must acquire an actual, physical copy of the object and assign it to that variable, using the new operator.
  - E.g.:

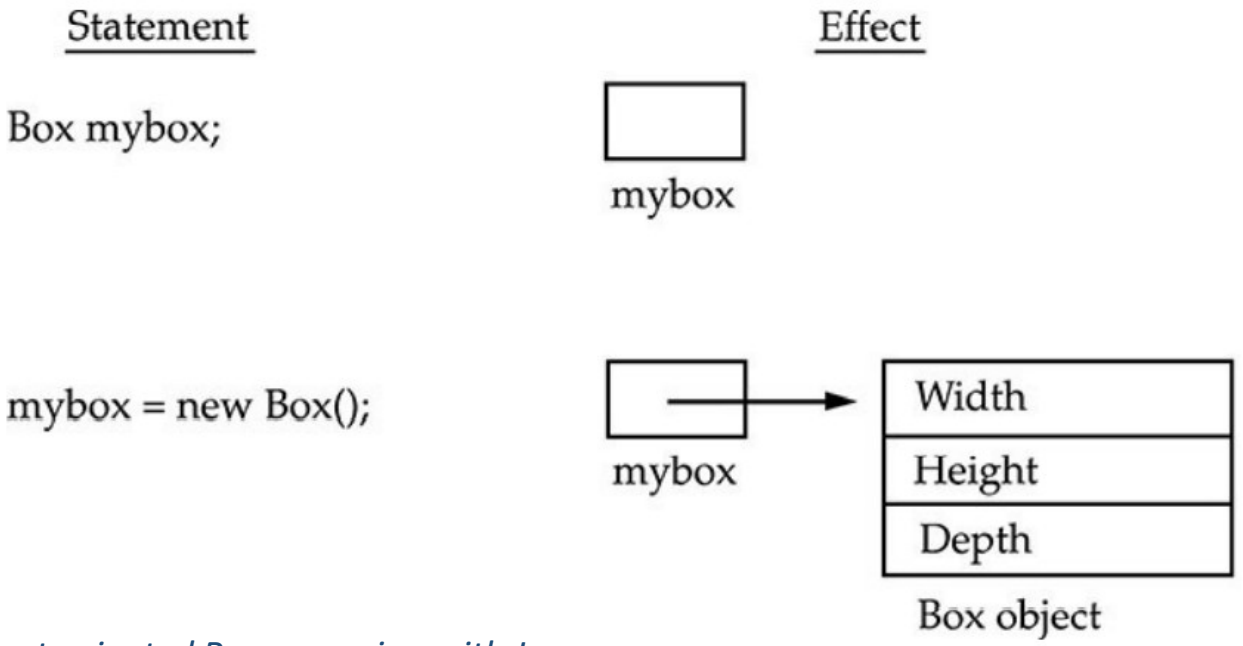
```
Box mybox = new Box();
```

Or

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

Declaring Objects

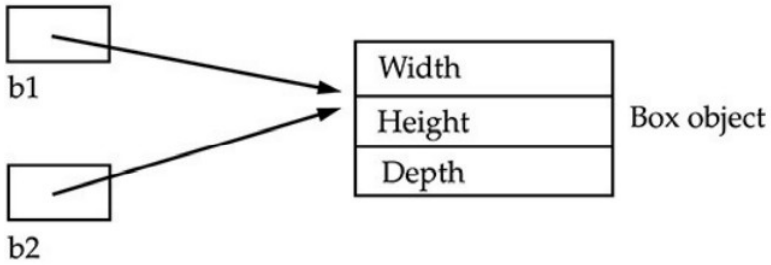
- The new operator dynamically allocates memory for an object and returns a reference to it. This reference is the address in memory of the object. This reference is then stored in the variable.



Assigning Object Reference Variables

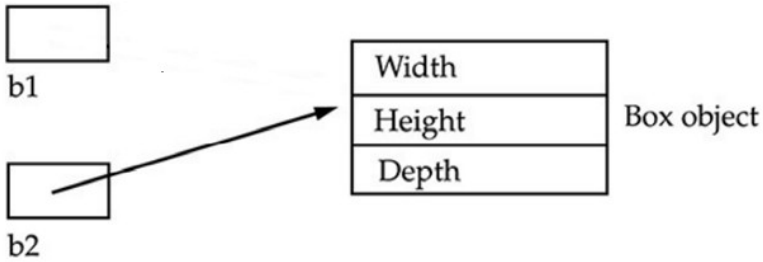
```
Box b1 = new Box();
Box b2 = b1;
```

- b1 and b2 will both refer to the same object.




```
Box b1 = new Box();
Box b2 = b1;
b1 = null;
```

- b1 has been set to null, but b2 still points to the original object.



Variable/Method Access

- Every **Box** object contains its own copies of the instance variables **width**, **height**, and **depth**.
- To access these variables or methods, use the dot (.) operator. The operator '.' links the name of the object with the name of an instance variable/method.
- E.g., `mybox.width = 10;//to assign the width variable of mybox the value 100`

Class BoxDemo

```
//This class declares an object of type Box
public class BoxDemo {
    public static void main(String[] args) {
        Box mybox = new Box();
        double vol;

        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;

        vol = mybox.width * mybox.height * mybox.depth;

        System.out.println("Volume is " + vol);
    }
}
```

Class

- Remarks
  - Call the file that contains this program BoxDemo.java, because the main( ) method is in the class **BoxDemo** (not the class called **Box**).
  - When compiling this program, you will find that two .class files have been created, one for **Box** and one for **BoxDemo**.
  - The Java compiler automatically puts each class into its own .class file. It is not necessary for both the **Box** and the **BoxDemo** class to be in the same source file. Each class is in different file, called Box.java and BoxDemo.java.
  - To run this program, you must execute BoxDemo.class.

Output: Volume is 3000.0

Class BoxDemo2

- mybox1's data is completely separate from the data contained in mybox2

The output produced by this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

```
/**
 * This program declares two Box objects.
 */
public class BoxDemo2 {

    public static void main(String[] args) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);

        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
    }
}
```

Methods

- Use methods to access the instance variables defined by the class.
- General form of a method:

```
type name(parameter-list) {
    // body of method
    type value;
    //…
    return value; //void method doesn't have this
}
```

- type specifies the type of data returned by the method.
- This can be any valid type, including class types that you create.
- If the method does not return a value, its return type must be void.

Returning a Value

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

    void volume() {
 System.out.print("Volume is ");
 System.out.println(width * height *
depth);
    }
}
```

```
 /* This program includes a method inside
the box class.
 */
public class BoxDemo3 {

    public static void main(String[] args) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        mybox1.volume();

        mybox2.volume();
    }
}
```

Returning a Value

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

    double computeVolume() {
        return width * height * depth;
    }
}
```

```
public class BoxDemo4 {

    public static void main(String[] args) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

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

        vol = mybox2.computeVolume();
        System.out.println("Volume is " + vol);
    }
}
```

Returning a Value

- The type of data returned by a method must be compatible with the return type specified by the method.
  - E.g., if the return type of some method is boolean, you could not return an integer.
- The variable receiving the value returned by a method must also be compatible with the return type specified for the method.

Returning a Value

- The underlined statement in slide 19 can be rewritten more efficiently without the `vol` variable:

```
System.out.println("Volume is" + mybox1.volume());
```

Adding a Method That Takes Parameters

```
int square(){
        return 10 * 10;
}
```

- Parameters allow a method to be generalized.

```
int square(int i){
      return i * i;
}
```

- From main()

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

- A parameter is a variable defined by a method that receives a value when the method is called. E.g., in square( ), i is a parameter.
- An argument is a value that is passed to a method when it is invoked. E.g., square(100) passes 100 as an argument

Parameterized Method

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

    double computeVolume() {
        return width * height * depth;
    }

    void setDim(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
}
```

```
public class BoxDemo5 {

    public static void main(String[] args) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);

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

        vol = mybox2.computeVolume();
        System.out.println("Volume is " + vol);
    }
}
```

Constructors

- A constructor initializes an object immediately upon creation.
- It has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called when the object is created, before the new operator completes.
- Constructors have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself.
- Its job is to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

Constructors

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

    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }

    double computeVolume() {
        return width * height * depth;
    }
}
```

Parameterized Constructors

- The Box( ) constructor initializes all boxes with the same predetermined dimensions.

➔ Each object can be initialized by specifying in the parameters to its constructor.

- E.g.: to construct Box objects of various dimensions, we set the parameters in the input of the constructor.

Parameterized Constructors

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

    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }

    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    double computeVolume() {
       return width * height * depth;
        }
    }
```

Parameterized Constructors

```
/*
 * Box uses a parameterized constructor to initialize the dimensions of a box.
 */
public class BoxDemo7 {

    public static void main(String[] args) {

        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);

        double vol;

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

        vol = mybox2.computeVolume();
        System.out.println("Volume is " + vol);
    }
}
```

Parameterized Constructors

```
/*
 * Box uses a parameterized constructor to initialize the dimensions of a box.
 */
public class BoxDemo7 {

    public static void main(String[] args) {

        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);

        double vol;

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

        vol = mybox2.computeVolume();
        System.out.println("Volume is " + vol);
    }
}
```

The output from this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

The `this` Keyword

- `this` can be used inside any method to refer to the current object on which the method was invoked.

```
Box(double width, double height, double depth) {
        this.width = width;
    this.height = height;
    this.depth = depth;
}
```

- The use of `this` is redundant but correct.

The `this` Keyword

Instance Variable Hiding

- It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- A local variable has the same name as an instance variable, the local variable hides the instance variable.
- In example (slide 29), the function Box( ) uses width, height, and depth for parameter names, and uses `this` to access the instance variables by the same name.

Garbage Collection

- In C++, dynamically allocated objects must be manually released by use of a `delete` operator.
- Java takes a different approach; it handles deallocation for you automatically.
  - The Java runtime environment has **a garbage collector** that periodically frees the memory used by objects that are no longer referenced.

A Stack Class

- Is an example of encapsulation.
- A stack stores data using first-in, last-out ordering.
- Implements a stack for up to ten integers:

A Stack Class

```
class Stack {
    int stck[] = new int[10];
    int tos; //top of stack

    Stack() {
        tos = -1;
    }

    void push(int item) {
        if (tos == 9) {
            System.out.println("Stack is full.");
        } else {
            stck[++tos] = item;
        }
    }
```

```
    int pop() {
        if (tos < 0) {
            System.out.println("Stack underflow.");

            return 0;
        } else {
            return stck[tos--];
        }
    }
}
```

A Stack Class

```
public class TestStack {

    public static void main(String[] args) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        for (int i = 0; i < 10; i++)
            mystack1.push(i);
        for (int i = 10; i < 20; i++)
            mystack2.push(i);

        System.out.println("Stack in mystack1:");
        for (int i = 0; i < 10; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for (int i = 0; i < 10; i++)
            System.out.println(mystack2.pop());
    }
}
```

A Stack Class

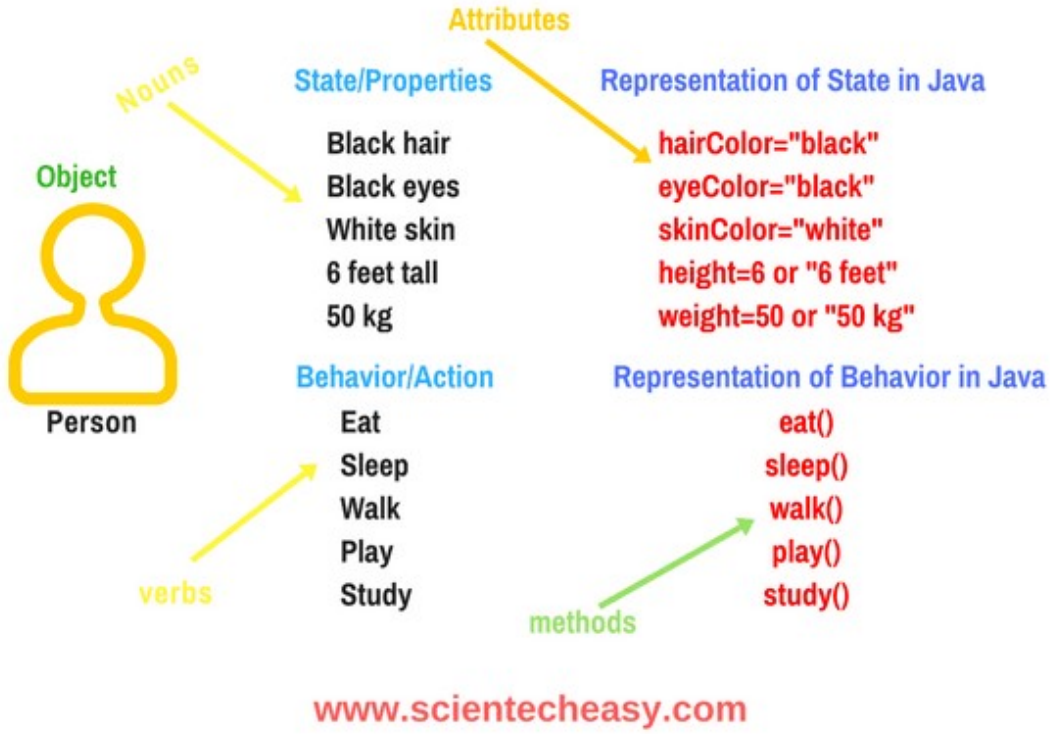
```
public class TestSta

    public static vo
        Stack mystac
        Stack mystac

        for (int i =
            mystack1
        for (int i =
            mystack2

        System.out.p
        for (int i =
            System.o

        System.out.p
        for (int i =
            System.o
    }
}
```

This program generates the following output:

```
Stack in mystack1:      Stack in mystack2:
9                       19
8                       18
7                       17
6                       16
5                       15
4                       14
3                       13
2                       12
1                       11
0                       10
```

Exercise

See the figure below and create class Person and declare 4 objects which describe your 4 colleagues.



Take a close look at Methods and Classes…

Overloading Methods

- 2 or more methods within the same class that share the same name, as long as their parameter declarations are different.
- Method overloading is one of the ways that Java supports polymorphism.
- 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.
- 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. Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.

Overloading Methods

```
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }

    void test(int a) {
        System.out.println("a: " + a);
    }

    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }

    double test(double a) {
        System.out.println("double a: " + a);
        return a * a;
    }
}
```

Overloading Methods

```
class OverloadDemo {
    void test() {
        System.out.println("No para
    }

    void test(int a) {
        System.out.println("a: " + 
    }

    void test(int a, int b) {
        System.out.println("a and b
    }

    double test(double a) {
        System.out.println("double 
        return a * a;
    }
}
```

```
/**
 * Demonstrate method overloading.
 */
public class Overload {
    public static void main(String[] args) {
        OverloadDemo ob = new OverloadDemo();
        double result;

        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of
ob.test(123.25): " + result);
    }
}
```

Overloading Methods

```
class OverloadDemo {
    void test() {
        System.out.println("No para
    }

    void test(int a) {
        System.out.println("a: " + 
    }

    void test(int a, int b) {
        System.out.println("a and b
    }

    double test(double a) {
        System.out.println("double 
        return a * a;
    }
}
```

```
/**
 * De
 */
publi
    p
        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
```

Overloading Methods

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

```
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }

    void test(int a, int b) {
        System.out.println("a: " + a);
    }

    double test(double a) {
        System.out.println("Inside test (double) a: " + a);
        return a * a;
    }
}
```

Overloading Methods

```
class OverloadDemo {
    void test() {
        System.out.println("No parameters");
    }

    void test(int a, int b) {
        System.out.println("a:
    }

    double test(double a) {
        System.out.println("Ins
        return a * a;
    }
}
```

```
public 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);

    }
}
```

Overloading Method

```
class OverloadDemo {
    void test() {
        System.out.println("No pa
    }

    void test(int a, int b) {
        System.out.println("a:
    }

    double test(double a) {
        System.out.println("Ins
        return a * a;
    }
}
```

This program generates the following output:

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

```
public 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);

    }
}
```

Overloading Constructors

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

//constructor used when all dimension
specified
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
//constructor used when no dimensions
specified
    Box() {
        width = -1;
        height = -1;
        depth = -1;
    }
```

```
//constructor used when cube is created
Box(double len) {
    width = height = depth = len;
 }

//compute and return volume
double volume(){
       return width*height*depth;
}
```

Overloading Constructors

```
public class OverloadCons {
    public static void main(String[] args) {
        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 secondbox
        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);
    }
}
```

Overloading Constructors

```
public class OverloadCons {
    public static void main(String[] args) {
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
```

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
```

```
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

```
//Box allows one object to initialize another.
class Box2 {
    double width;
    double height;
    double depth;

    Box2(Box2 ob) {//pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    Box2(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    Box2() {
        width = -1;
        height = -1;
        depth = -1;
    }

    Box2(double len) {
        width = height = depth = len;
    }

    double volume() {
        return width * height * depth;
    }
}
```

Using Objects as Parameters

```
public class OverloadCons {
    public static void main(String[] args)
{
        Box2 mybox1 = new Box2(10, 20, 15);
        Box2 mybox2 = new Box2();
        Box2 mycube = new Box2(7);
//create copy of mybox1
        Box2 myclone = new Box2(mybox1);

        double vol;
//get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume of
mybox1 is " + vol);
//get volume of secondbox
        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);
//get volume of clone
        vol = myclone.volume();
        System.out.println("Volume of
mycube is " + vol);
    }
}
```

Argument Passing

- 2 ways that a computer language can pass an argument to a subroutine:
  1. **call-by-value:**
     - copies the value of an argument into the formal parameter of the subroutine
     - changes made to the parameter of the subroutine have no effect on the argument
  2. **call-by-reference:**
     - is a reference to an argument (not the value of the argument) is passed to the parameter.
     - changes made to the parameter will affect the argument used to call the subroutine.

Argument Passing

- In Java:
  - When a primitive type is passed to a method, it is passed by value.
  - When objects are passed to a method, it is passed by reference.

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);
  }
}
```

The output from this program is shown here:

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

the operations that occur inside meth( ) have no effect on the values of a and b used in the call; their values here did not change to 30 and 10.

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 PassObjRef {
  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
```

the actions inside meth( ) have affected the object used as an argument.

Returning Objects

- A method can return any type of data, including class types.

```
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);
  }
}
```

Returning Objects

- A method can return any type of data, including class types.

```
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
```

Access Control

- Encapsulation links data with the code that manipulates it.
- Encapsulation provides another important attribute: access control.
- Through encapsulation, you can control what parts of a program can access the members of a class.
- By controlling access, you can prevent misuse.
  - E.g., allowing access to data only through a well-defined set of methods.
  - When correctly implemented, a class creates a "**black box**" which may be used, but the inner workings of which are not open to tampering.

Access Control

- How a member can be accessed is determined by the access modifier attached to its declaration, i.e., public, private, and protected.

Visibility	Public	Protected	*Default*	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, *through inheritance*	No	No
From any non-subclass class outside the package	Yes	No	No	No

Access Control

- How a member can be accessed is determined by the access modifier attached to its declaration, i.e., public, private, and protected.

why *main( )* has always been preceded by the public modifier?

As *main()* is called by code that is outside the program

Access Control

```
//This program demonstrates the
difference between public and private.
class Test5 {
    int a; //default access
    public int b; // public access
    private int c; // private access

//method to access c
    void setc(int i) { //set c's value
        c = i;
    }

    int getc() { //get c's value
        return c;
    }
}
```

```
public class AccessTest {

    public static void main(String[] args)
{
        Test5 ob = new Test5();
//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);
        System.out.println("a, b, and c: "
+ ob.a + " " + ob.b + " " + ob.getc());
    }
}
```

static

- Instance variables declared as `static` are `global` variables.
- All instances of the class share the same `static` variable.
- A `static` block that gets executed exactly once, when the class is first loaded.
- Methods declared as `static` have **restrictions**:
  - They can only directly call other static methods of their class.
  - They can only directly access static variables of their class.
  - They cannot refer to `this` or `super` in any way.

static

```
//Demonstrate static variables, methods, and blocks.
public 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 { //static block
        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
```

static

```
class Student{
     int rollno;
     String name;
     static String college = "CSE";

     static void change(){
       college = "BBDIT";
     }

     Student(int r, String n){
       rollno = r;
       name = n;
     }
     void display(){
       System.out.println(rollno+" "+name+"
        "+college);
   }
}
```

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

     Student.change();

     Student s1 = new Student(111,"Karan");

     Student s2 = new Student(222,"Aryan");

     Student s3 = new Student(333,"Sonoo");


     s1.display();
     s2.display();
     s3.display();
     }
}
```

static

What are displayed at the console after running this program?

```
class Student{
     int rollno;
     String name;
     static String college = "CSE";

     static void change(){
     college = "BBDIT";
     }

     Student(int r, String n){
     rollno = r;
     name = n;
     }
     void display(){
       System.out.println(rollno+" "+name+"
        "+college);
   }
}
```

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```

```
ass TestStaticMethod{
    public static void main(String
 args[]){
    Student.change();

    Student s1 = new Student(111,"
Karan");
    Student s2 = new Student(222,"
Aryan");
    Student s3 = new Student(333,"
Sonoo");

    s1.display();
    s2.display();
    s3.display();
    }
}
```

static

- Outside of the class in which they are defined, static methods and variables can be used independently of any object.
- To do so, only specify the name of their class followed by the dot operator.

```
classname.method( )
```

where `classname` is the name of the class in which the static method is declared.

static

- A static variable can be accessed by the following syntax

```
classname.variable
```

```
class StaticDemo {
    static int a = 42;
    static int b = 99;

    static void callme() {
        System.out.println("a = " + a);
    }
}

public 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
```

final

- Prevents its contents from being modified, making it, essentially, a constant.
- This means that you must initialize a final field when it is declared.
  1. Give it a value when it is declared (this approach is probably the most common).
     - E.g.

```
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;
```

  2. Assign it a value within a constructor.

`final`

- Comments on the example at slide 79
  - Subsequent parts of your program can now use FILE_OPEN, etc., as if they were constants, without fear that a value has been changed.
  - It is a common coding convention to choose all uppercase identifiers for `final` fields.
- Method parameters and local variables can be declared final as well.
  - Declaring a parameter `final` prevents it from being changed within the method.
  - Declaring a local variable `final` prevents it from being assigned a value more than once.
- The keyword `final` can also be applied to methods, but its meaning is substantially different than when it is applied to variables.
  - Final class cannot be inherited

final

- What is the output of running the following code?

```
class Bike9{
 final int speedlimit = 90;//final variable
 void run(){
       speedlimit = 400;
 }
 public static void main(String args[]){
       Bike9 obj = new  Bike9();
       obj.run();
 }
}//end of class
```

final

- What is the output of running the following code?

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){
       System.out.println("running safely with 100kmph");
   }

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

final

- Final class cannot be inherited.
- E.g., What is the output of running the following code?

```
final class Bike{}

class Honda1 extends Bike{
  void run(){
       System.out.println("running safely with 100kmph");
  }

  public static void main(String args[]){
       Honda1 honda = new Honda1();
       honda.run();
  }
}
```

Arrays Revisited

- Arrays are implemented as objects.
- The size of an array is found in its `length` instance variable.

```
//This program demonstrates the length array member.
public 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);
    }
}
```

Arrays Revisited

- Arrays are implemented as obje
- The size of an array is found in

This program displays the following output:

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

```
//This program demonstrates the length array member.
public 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

- A class is defined 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.
  - E.g., 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 directly 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, i.e., inside an if statement block.

Nested and Inner Classes

```
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }
//this is an inner class
    class Inner {
       int y = 10;

void display() {

System.out.println("display: outer_x
= " + outer_x);
        }
    }
```

Output from this application is shown here:

```
display: outer_x = 100
```

```
    void showy() {
        // System.out.println(y);
Inner inner = new Inner();
Systemt.out.println(inner.y);
    }
}

public class InnerClassDemo {

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

Nested and Inner Classes

- There are two types of nested classes: `static` and non-`static`.
- A `static` nested class:
  - has the `static` modifier applied.
  - must access the non-`static` members of its enclosing class through an object.
  - cannot refer to non-`static` members of its enclosing class directly. *Because of this restriction, static nested classes are seldom used.*

Nested and Inner Classes

```
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
    }
}

public class InnerClassDemo {

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

Nested and Inner Classes

```
//Define an inner class within a for loop.
class Outer2 {
    int outer_x = 100;

    void test() {
        for (int i = 0; i < 10; i++) {
            class Inner2 {
                void display() {

System.out.println("display: outer_x = " +
outer_x);
                }
            }
            Inner2 inner = new Inner2();
            inner.display();
        }
    }
}
```

```
public class InnerClassDemo2 {

    public static void
main(String[] args) {
        Outer2 outer = new
Outer2();
        outer.test();
    }
}
```

Nested and Inner Classes

```
//Define an inner class within a for loop.
class Outer2 {
    int outer_x = 1

    void test() {
        for (int i
            class I
                voi


System.out.println(
outer_x);
            }
        }
        Inner2 inner = new Inner2();
        inner.display();
    }
}
```

```
ss InnerClassDemo2 {

 static void
g[] args) {
ter2 outer = new

ter.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
```

Nested and Inner Classes

```
class OuterClass {
  int x = 10;

  class InnerClass {
    public int myInnerMethod() {
      return x;
    }
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass myOuter = new OuterClass();
    OuterClass.InnerClass myInner = myOuter.new InnerClass();
    System.out.println(myInner.myInnerMethod());
  }
}

// Outputs 10
```

Private Member Access

- Within the definition of a method of the outer class
  - It is legal to refer to a private instance variable of the inner class on an object of the inner class
  - It is legal to invoke a (nonstatic) method of the inner class as long as an object of the inner class is used as a calling object
- Within the definition of the inner or outer classes, the modifiers public and private are equivalent

Overloading Methods?

If a method is invoked in an inner class

- If the inner class has no such method, then it is assumed to be an invocation of the method of that name in the outer class
- If both the inner and outer class have a method with the same name, then it is assumed to be an invocation of the method in the inner class
- If both the inner and outer class have a method with the same name, and the intent is to invoke the method in the outer class, then the following invocation must be used:

```
OuterClassName.this.methodName();
```

References

1. Herbert Schildt, “Java - The Complete Reference”, 11th edition, Oracle Press, 2019. ISBN: 978-1-26-044024-9.
2. Kathy Sierra and Bert Bates, “Head First Java”, 2nd Edition, O’reilly Media Publisher, 2005. ISBN: 0596009208.
3. Ian F. Darwin, “Java Cookbook”, 4th Edition, O'Reilly Media, 2020. ISBN: 9781492072584.
4. Ben Evans, David Flanagan, “Java in a Nutshell”, 7th Edition, O'Reilly Media, 2018. ISBN: 9781492037255.
5. Richard L. Halterman, “Object-oriented Programming In Java”, 2008.
6. Java tutorials, https://www.w3schools.com/java/.

Project Exercise

1. With your selected project topic, list objects and their behaviors and states.
2. List classes from generalizing the above objects.
3. Write variables and methods for each classes.
4. Add constructors for classes in your project.