Objects and Classes Chapter Notes

Before we start this chapter, I want to point out word choices, a word’s AKA; how a concept will also be called other things:

* Objects: classes, ADTs, instance of the class
* States: variables, fields
* Behaviors: methods

Contents

[What is an object? 2](#_Toc46837310)

[Object-Oriented Programming: 3](#_Toc46837311)

[UML Diagram: 3](#_Toc46837312)

[Generic Example of UML Diagram 3](#_Toc46837313)

[Applying UML Diagram to Code: 4](#_Toc46837314)

[Textbook example: 6](#_Toc46837315)

[Defining Classes for Objects 7](#_Toc46837316)

[Definition 7](#_Toc46837317)

[Instantiation 7](#_Toc46837318)

[Constructor 7](#_Toc46837319)

[Syntax 7](#_Toc46837320)

[Recap 8](#_Toc46837321)

[No-arg constructor 8](#_Toc46837322)

[Explicit constructor 8](#_Toc46837323)

[For this class 9](#_Toc46837324)

[Visibility Modifiers: 9](#_Toc46837325)

[Packages 9](#_Toc46837326)

[Types/Defining/Scope: 9](#_Toc46837327)

[Why do we want to use visibility modifiers? 10](#_Toc46837328)

[Data Field Encapsulation 10](#_Toc46837329)

[Default Values of an object 12](#_Toc46837330)

[Acccessors and Mutators AKA Setters and Getters 13](#_Toc46837331)

[this Reference 13](#_Toc46837332)

[Using the this Keyword 14](#_Toc46837333)

[Within a constructor, invoking another constructor: 14](#_Toc46837334)

[Object referring to its states: 14](#_Toc46837335)

[Using this to reference hidden data fields 15](#_Toc46837336)

[Immutable Objects and Classes 16](#_Toc46837337)

[Utilizing Objects/Classes 17](#_Toc46837338)

[Syntax 17](#_Toc46837339)

[Examples 17](#_Toc46837340)

[Accessing Object via Reference Variables 18](#_Toc46837341)

[Caution 18](#_Toc46837342)

[Where do we create objects? 18](#_Toc46837343)

[Separate Classes 19](#_Toc46837344)

[List object after the class that has main method in it 19](#_Toc46837345)

[Passing an object to a method is to pass the reference of the object 20](#_Toc46837346)

[Array of objects 20](#_Toc46837347)

[Scope of Variables 20](#_Toc46837348)

[Instance variable in Java 21](#_Toc46837349)

[static Variables in Java 21](#_Toc46837350)

[Recap 22](#_Toc46837351)

# What is an object?

An object is something in the real world that we want to represent in a program. We give these “somethings” - real world entities - different names, like object, class, and ADT (Abstract Data Type). We call them objects because we are trying to represent the facts of the entity; class because in Java we define a class to represent that entity; and ADT because as a whole, each is different but is a data type that we use in programming.

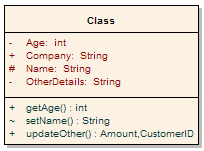
# Object-Oriented Programming:

Also seen as OOP for short. Object-oriented programming enables us to develop large-scale software and GUIs. It allows reusable soft-ware. It has 4 pillars, or fundamental concepts: Data Encapsulation, Abstraction, Inheritance, and Polymorphism. We will cover encapsulation in this chapter, abstraction in the next chapter, and inheritance and polymorphism in the following.

# UML Diagram:

UML diagrams aid us (as program designers) in creating an object/class/ADT.

The object/class/ADT is represented as a class name, state, and behaviors:



Class Name

State: the data

Behavior: the methods

Where the class name is the name of the object/class/ADT, state is the data (things we can define or describe) associated with the object/class/ADT, and behavior are the methods (the things it can do) of the object/class/ADT.

We can think of this as a 1x3 table.

The class cell consists of the class name in bold aligned in the middle of the cell.

The state cell consists of each state/variable that the object has aligned to the left. Each variable is presented in the fashion of identifier: dataType.

The behavior cell lists all the methods that is associated with the object. Each method is presented in the fashion of methodIdentifier(parameterList): returnValue.

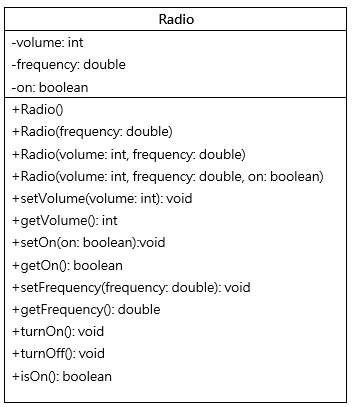
## Generic Example of UML Diagram

|  |
| --- |
| **ObjectName** |
| variableName: dataType |
| Constructor()  methodName(): void  methodName(variableName: datatype, … ): dataType |

## Applying UML Diagram to Code:

Once you have the UML diagram, you transfer it to code to define the object/class.

This UML Diagram defines a Radio class with the states of volume, frequency, and on:

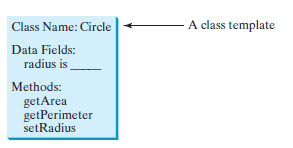


Code:

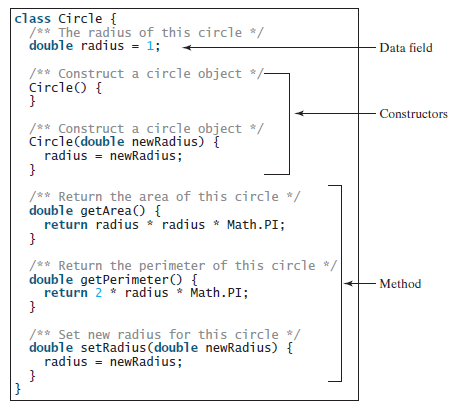
11 public class Radio {  
12   
13 private int volume = 1; //1-10  
14 private double frequency = 82.0; //82-127  
15 private boolean on = false;  
16   
17 public Radio(){  
18 this(true, 1, 82.0);  
19 //setOn(true);  
20 //setVolume(1);  
21 //setFrequency(126);  
22 }  
23   
24 public Radio(double frequency){  
25 this(true, 1, frequency);  
26 //setOn(true);  
27 //setVolume(1);  
28 //setFrequency(frequency);  
29 }  
30   
31 public Radio(int volume, double frequency){  
32 this(true, volume, frequency);  
33 //setOn(true);  
34 //setVolume(volume);  
35 //setFrequency(frequency);  
36 }  
37   
38 public Radio(boolean on, int volume, double frequency){  
39 setOn(on);  
40 setVolume(volume);  
41 setFrequency(frequency);  
42 }  
43   
44 public void setVolume(int volume){  
45 if(on && volume >= 1 && volume <= 10)  
46 this.volume = volume;  
47 }  
48   
49 public int getVolume(){  
50 if(on)  
51 return this.volume;  
52 else{  
53 System.out.println("sorry, the radio is off");  
54 return 0;  
55 }  
56 }  
57   
58 public void setFrequency(double frequency){  
59 if (on && frequency >= 82 && frequency <= 127)  
60 this.frequency = frequency;  
61 }  
62   
63 public double getFrequency(){  
64 if(on)  
65 return this.frequency;  
66 else{  
67 System.out.println("sorry, the radio is off");  
68 return 0;  
69 }  
70 }  
71   
72 public void setOn(boolean on){  
73 this.on = on;  
74 }  
75   
76 public boolean getOn(){  
77 return this.on;  
78 }  
79   
80 public void turnOn(){  
81 this.on = true;  
82 }  
83   
84 public void turnOff(){  
85 this.on = false;  
86 }  
87   
88 public boolean isOn(){  
89 return this.on;  
90 }  
91   
92 }

### Textbook example:

UML Diagram:



Code:



# Defining Classes for Objects

Objects are reference data types in Java.

**Object** = real world entity

State = data field = variable = properties or attributes (defines the object)

Behavior = methods (what the object does)

**Class** = a construct that define objects of the same type

Definition:

A class is a template, blueprint, or contract that defines what an object’s data fields and methods will be.

The relationship between classes and objects is analogous to that between an apple-pie recipe and apple pies: You can make as many apple pies as you want from a single recipe.

An object is an instance of the class.

The terms *object* and *instance* are often interchangeable.

Instantiation:

Creating an instance of a class is called instantiation.

Constructor:

Regarding the behaviors, there is a special method called a constructor. The constructor does what it sounds like, it constructs an object; it is the instantiationator. The constructor has to have the same name as the class (otherwise it will not compile).

The constructor is a unique method. Its purpose is to create objects, as such, the states can be assigned values passed through the parameter list. You can define as many constructors as you want, as long as the parameter list is different. The constructor does not have a return type in the method header.

Java constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.

FYI: constructors are listed right after the states in your object programs.

Syntax:

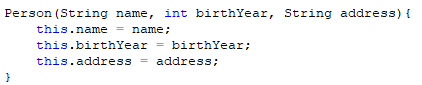
visibilityModifier ClassName (parameterList)

There are two types of constructors:

1. Default constructor; it is also called a no-arg constructor (because there are no arguments for this constructor



1. Parameterized constructor; also called explicit constructors (as these have been explicitly defined)



Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

If you do not create/define a constructor, then the compiler will create/define a default (also called no-args) constructor at compile time.

If you do create/define a constructor, then the compiler will not define the default/no-args constructor.

Constructor chaining is a concept that we will learn more about when we cover inheritance.

Recap:

No-arg constructor: This constructor method does not have a parameter list. If your class does not define any constructors, then (behind the scene), the compiler will automatically create this method/constructor in your .class file.

If you have default values for the state(s) in your class, then when that object is instantiated, those default values will be assigned to that object.

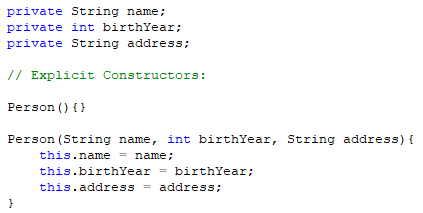
Explicit constructor: This/these is/are the methods that you are defining within your class. You can create as many as you deem necessary, remember that you want your objects to be user friendly.

When you define an explicit constructor, you can have values passed in to be assigned to the states.

* Constructors = invoked to construct objects from the class = instantiation
* Constructor methods have the same name of the class
* Default constructor: the constructor defined by the compiler when a class is defined without constructors
  + *Will automatically be created if no constructors are explicitly defined*
* No-arg constructor: no parameters = Default constructor
* Constructors do NOT have a return type
* Constructors are invoked by using the new operator

For this class, our objects will always have at least 2 explicitly defined constructors: 1) the no-args and 2) with all the states being assigned values.

Example:



Visibility Modifiers:

Visibility modifiers are keywords and add a component to the states and behaviors of the object that we are defining, this component lets us control how other objects have access to the states and behaviors; it is the scope. This scope provides ways that we can protect the state and behaviors.

Packages:

Before we go farther, let us get an understanding of packages. A package, as the name suggests, is a pack (or group) of classes, interfaces and other packages. In java, we use packages to organize our classes and interfaces. There are two types of packages in Java: built-in packages and the packages we can create (also known as user defined package).

In a rough sense, we can think of packages as a folder system. The main folder that holds files and other sub-folders, those sub-folders can hold other files and sub-folders, and so on.

If we look at Java’s packages, we can see the java language as the main folder and util and io as some of the sub-folders.

## Types/Defining/Scope:

We are already exposed to one of these visibility modifiers: public. Using public as a visibility modifier means that the state and/or the behavior is available to all – it is like we think of the word public – it is out there for all to see and use. We are going to add to this three other visibility modifiers, protected, default, and private. We have actually been using *default* all along for the variables you have been using – you just didn’t know it! Below is a chart showing the scope of each visibility modifier.

(Hiding/protecting/exposing data)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Modifier | UML Symbol | Accessed from the same class | Accessed from the same package | Accessed from a subclass in a different package | Accessed from a different package |
| public | + | 🗸 | 🗸 | 🗸 | 🗸 |
| protected | # | 🗸 | 🗸 | 🗸 |  |
| default |  | 🗸 | 🗸 |  |  |
| private | - | 🗸 |  |  |  |

## Why do we want to use visibility modifiers?

These modifiers allow us to say where the class, state, and/or behaviors can be accessed. This provides us a way we can protect the state, this gives us a way to ensure the integrity of the data – a main goal. We do not want any ol’ program to manipulate the data, if the data is to be changed, then it has to go through the proper channels to change it. We keep the integrity of the data, it keeps the program’s validity; valid results = valid program, invalid results = worthless program.

Data Field Encapsulation: Making data fields private protects data and makes the class easy to maintain.

This is one of the fundamentals of OOP concepts! (The 4, sometimes people say there are 3, pillars of OOP are: encapsulation, abstraction, inheritance, and polymorphism.) It is very important to understand this concept.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation you have to:

* use the private modifier for the fields/states so that the state can only be accessed within its own class
* Define getters/accessors & setters/mutators to access & modify the fields/states as needed for each state/field. This ensures that the data keeps its integrity; you can only access and/or change a state/field by going through the proper channels.

Benefits of Encapsulation:

* Protects the data
* Makes the class easy to maintain
* Allows a state/field to be make read-only or write-only
* Keeps the integrity of the class/object

Example in text book: CircleWithPrivateDataFields.java

Example:

public class Person {

private String name;

private int birthYear;

private String address;

// Explicit Constructors:

Person(){}

Person(String name, int birthYear, String address){

this.name = name;

this.birthYear = birthYear;

this.address = address;

}

// setters/mutators & getters/accessors:

public void setName(String name){

this.name = name;

}

public String getName(){

return this.name;

}

public void setBirthYear(int birthYear){

this.birthYear = birthYear;

}

public int getBirthYear(){

return this.birthYear;

}

public void setAddress(String address){

this.name = address;

}

public String getAddress(){

return this.address;

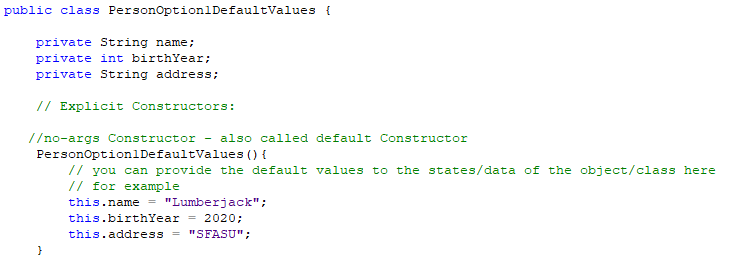
}

}// end of Person Object/Class

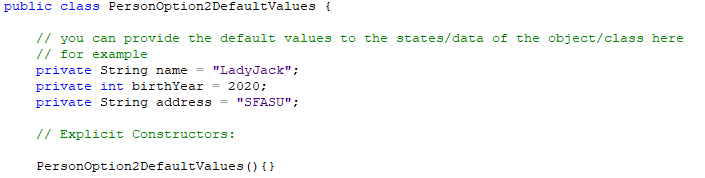
Default Values of an object:

You, as the designer, can provide default values to an object. There are several ways to do this.

1. In the default/no-args constructor, assign the values there to the states/variables.



1. Assign the default values when the states/variables are declared.



If you do not provide default values to the states/variables and an object is invoked using the default/no-args constructor, then the Java default values of that data type will be assigned.

**null Value**: if a data field (state) of a reference type does not reference any object, the data field holds a special literal: null

The default value of a data field is null for a reference type, 0 for a numeric type, false for a boolean type, and '\u0000' for a char type.

**However, Java assigns no default value to a local variable inside a method.**

Will get a compile error!

# Acccessors and Mutators AKA Setters and Getters

Each object’s state(s) can have specific methods that allow the state a value to be assigned (mutator/setter) or a method that returns the state’s value(accessor/getter).

Examples:

public void setOn(boolean on){

this.on = on;

}

public boolean getOn(){

return this.on;

}

And yes, these methods ALWAYS follow this format!

* Setters: void setIdentifier (datatype identifier)
  + always have a return type of void (they set a value – there is nothing to return)
  + always start with set\_\_\_\_\_identifierOfState\_\_\_\_\_
  + always take the ONE argument – that one specific state
* Getters: datatype getIdentifier ()
  + always have the return type of that specific state
  + always start with get\_\_identifierOfState\_\_\_\_
  + never have a parameter!

Recall that a method does one thing, and does that one thing well!

FYI: the setter and getters are listed right after the constructors in your object program. After the setters and getters are the rest of the behaviors for the object.

# this Reference

Within an instance method or a constructor, this is a reference to the *current object* — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using this.

The keyword this can be used in several situations:

* It can also be used inside a constructor to invoke another constructor of the same class.
* The this keyword is the name of a reference that an object can use to refer to itself. You can use the this keyword to reference the object's instance members.
* The this reference is needed to reference hidden data fields or invoke an overloaded constructor.

## Using the this Keyword

### Within a constructor, invoking another constructor:

In the below example, the first constructor (with no-args) is passing default values to the second constructor. In the second constructor, the values are being assigned in the setters.

public Radio(){

this(true, 1, 82.0);

}

public Radio(boolean on, int volume, double frequency){

setOn(on);

setVolume(volume);

setFrequency(frequency);

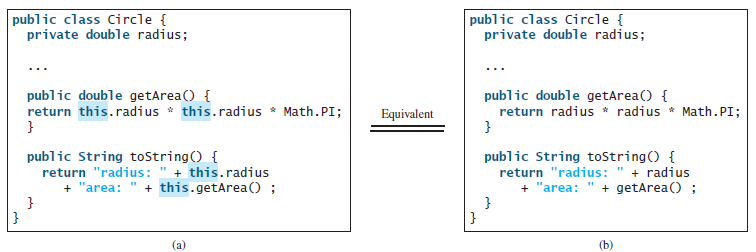
}

If a class has multiple constructors, it is better to implement them using this(arg‑list) as much as possible.

In general, a constructor with not or fewer arguments can invoke a constructor with more argument using this(arg‑list).

This syntax often simplifies coding and makes the class easier to read and maintain.

### Object referring to its states:



Using this to reference hidden data fields:

A data-field name is often used as the parameter name in a setter method for the data field. When you do this, then the data field is hidden in the setter method. You need to reference the hidden data-field name in the method in order to set a new value to it.

A hidden static variable can be accessed by using the ClassName.staticVariable reference.

A hidden instance variable (an object) can be access by using the keyword this.

In this example, to ensure that the JVM knows it is to take the value that is being passed in (identifier is volume) and assign it to the object that is invoking this method, its state (also with the identifier of volume).

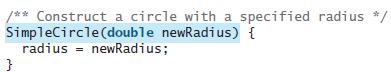
public void setVolume(int volume){

if(on && volume >= 1 && volume <= 10)

this.volume = volume;

}

In this way, we can use the same identifier name to mean the same thing, without losing the integrity of the data’s value. AND is much more appropriate and keeps the correct context of the identifiers. Instead of saying, as in the below example, newRadius to refer to radius, you keep the context of radius.



“Correct” Way:

SimpleCircle(double radius){

this.radius = radius;

}

**Using this with a Field**

The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter.

For example, the Point class was written like this

public class Point {

public int x = 0;

public int y = 0;

**//constructor**

**public Point(int a, int b) {**

**x = a;**

**y = b;**

**}**

}

but it SHOULD have been written like this:

public class Point {

public int x = 0;

public int y = 0;

**//constructor**

**public Point(int x, int y) {**

**this.x = x;**

**this.y = y;**

**}**

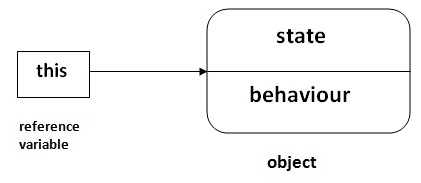
}

Each argument to the constructor shadows one of the object's fields — inside the constructor **x** is a local copy of the constructor's first argument. To refer to the Point field **x**, the constructor must use **this.x**.

**Usage of java this keyword**

Here is given the 6 usage of java this keyword.

1. this keyword can be used to refer current class instance variable.
2. this() can be used to invoke current class constructor.
3. this keyword can be used to invoke current class method (implicitly)
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this keyword can also be used to return the current class instance.



# Immutable Objects and Classes

You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed.

To define an object as immutable you do not create the setter/mutator methods.

If a class is immutable, then all its data fields must be private and it cannot contain public setter methods for any data fields.

The String class is immutable.

A class with all private data fields and no mutators is not necessarily immutable.

For a class to immutable, it *must* meet the following requirements:

* All data fields must be private
* There cannot be any mutator/setter methods for data fields
* No accessor/getter methods can return a reference to a data field that is mutable

# Utilizing Objects/Classes

To use an object, we have to create an instance of the object – instantiate an object.

We do this as we have been doing all along when we create a reference data type (like Scanner or File), by using the new keyword.

Syntax:

DataType identifier = new DataType();

Examples:

Radio radioNoArgsExample = new Radio();

Radio radioArgsExample = new Radio (true, 3, 90.1);

Circle circle1 = new Circle (2);

The **new** keyword is used to allocate memory at runtime.

new 🡪 instantiates a class by allocating memory for a new object and returning a reference to that memory location (a heap)

Instantiating a class = creating an object (instance = object)

new 🡪 invokes the object constructor

In memory:







Garbage = auto collected by JVM

# Accessing Object via Reference Variables

Once an object has been instantiated, we can use the reference variable (the identifier) to access the states and the behaviors associated with it. An object’s data and methods can be accessed through the dot (.) operator via the object’s reference variable (the identifier).

Referencing object's data: objectRefVar.data

Example: myCircle.radius

Invoking object's method: objectRefVar.methodName(arguments)

Example: myCircle.getArea()

Caution:

Recall that you use Math.methodName(arguments) (e.g., Math.pow(3, 2.5)) to invoke a method in the Math class. Can you invoke getArea() using SimpleCircle.getArea()? The answer is no. All the methods used before this chapter are static methods, which are defined using the static keyword. However, getArea() is non-static. It must be invoked from an object *using* objectRefVar.methodName(arguments) (e.g., myCircle.getArea()).

More explanations will be given in the section on “Static Variables, Constants, and Methods” in the textbook.

# Where do we create objects?

We want to keep our defined object code clean and to be able to reuse that code, we also cannot have a main method in our object class.

FYI: the textbook calls the program that uses the objects Test\*.java

We have two choices of how we want to set this up.

1. Separate classes
2. List the object after the class that has main method in it

Examples:

## Separate Classes

In this example, the program that has main is its own file and the program that defines the Radio class is its own file.

public class UseRadio {

public static void main(String[] args) {

Radio radio1 = new Radio(-5, 90.1);

Radio radio2 = new Radio(5, 90.1);

}// end main

}// end UseRadio

public class Radio {

// providing default values for a newly instantiated object

private int volume = 1; //1-10

private double frequency = 82.0; //82-127

private boolean on = true;

…(the rest of the code here)…

}// end Radio

## List object after the class that has main method in it

In this example, the file is saved as UseRadio.java. The Radio class is listed after the class that has main and Radio class does NOT use the public visibility modifier. It also limits the access to the Radio class and it cannot be used outside of this package.

public class UseRadio {

public static void main(String[] args) {

Radio radio1 = new Radio(-5, 90.1);

Radio radio2 = new Radio(5, 90.1);

}// end main method

}// end UseRadio class

class Radio {

// providing default values for a newly instantiated object

private int volume = 1; //1-10

private double frequency = 82.0; //82-127

private boolean on = true;

public Radio(){}

public Radio(boolean on, int volume, double frequency){

setOn(on);

setVolume(volume);

setFrequency(frequency);

}

…(the rest of the code here)…

}// end Radio

# Passing an object to a method is to pass the reference of the object

Why??? Because Java is ***pass-by-value***

Call stack for methods – heaps for objects

# Array of objects

An array of objects is actually an array of reference variables to the objects.

So, invoking an array of objects involves 2 levels of referencing.

When an array of objects is created using the new operator, each element in the array is a reference variable with a default value of null.

See Listing 9.11 TotalArea.java for detailed explanations.

# Scope of Variables

Scope is in regards to where and how a variable or method can be accessed.

In a class definition, there are three kinds of variables.

* Instance variables: variables declared outside of any method. Any method in the class definition can access these variables
* Parameter variables: variables on the parameter list. Only the method where the parameter appears can access these variables. This is how information is passed to the object.
* Local variables: variables declared inside a body of code, such as a method. Only the method where the parameter appears can access these variables. These variables are used to store intermediate results.
  + - Scope of a local variable begins at the point of declaration, and lasts to the end of the body where it is declared.

An instance variable can be made a static variable by adding the keyword static:

int number = 3;

static int number = 3;

The scope of instance and static variables is the entire class, regardless of where the variables are declared.

(Do not confuse a local variable with an instance variable.)

## Instance variable in Java

A variable that is created inside the class but outside the method, is known as instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object is created (instantiated). That is why, it is known as instance variable.

Instance variables:

* belong to the instances and have memory storage that is independent of the one another (each in their own heap)
* dependent
* can only be accessed by the instances after they are created
  + Meaning you have to create an instance (an object) of the class to access it.
  + Example:

public class Scope {

int number3 = 3;// instance variable

public static void main(String[] args) {

System.out.println ("instance variable number3: " +

new Scope().number3);

}

}

* reference is passed to the methods
  + therefore the changes are applied to the object

## static Variables in Java

The **static** keyword in Java is used for memory management mainly. static keyword can be applied to variables, methods, blocks and nested class.

\*\*The static keyword *belongs to the* ***class*** rather than an instance of the class. \*\*

The static can be:

1. variable (also known as class variable)

2. method (also known as class method)

3. block

4. nested class

1) Java static variable

If you declare any variable as static, it is known as a static variable.

* The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees, college name of students etc.
* The static variable gets memory only once in class area at the time of class loading. Making the program’s memory efficient

2) Java static method

If you apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* static method can access static data member and can change the value of it.

There are two main restrictions for the static method. They are:

1. The static method cannot use non static data member or call non-static method directly.

2. this and super cannot be used in static context.

static:

* has scope of the class
* Variables are shared by all the instances of the same class
* Non-dependent
* Methods and data can be accessed from a reference variable or class name
  + Think of the Math class
* Use the class name to improve readability
  + ClassName.method()
  + ClassName.staticVariable()

Recap:

* Instance and static variables in a class are referred to as the class's variables or data fields.
* Local variables are declared and used inside a method - locally.
* The class's variables and methods can appear in any order in the class.
* The exception: when a data field is initialized based on a reference to another data field.
* You can declare a class's variable only once, but you can declare the same variable identifier in a method many time in different non-nesting blocks.
  + If you do that (re-use identifiers) then the class is hidden to the local and the local takes precedence.
  + To avoid confusion and mistakes DO NOT do that – do not use the identifiers of instance or static variables as local variable identifiers, except for method parameters.