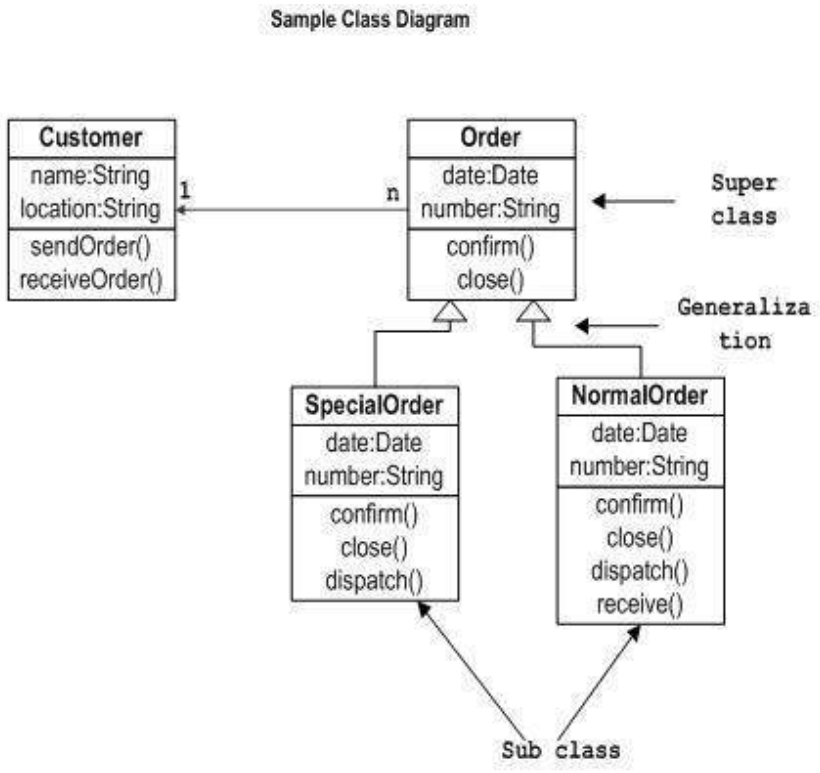

Java™
Basics

Table of contents

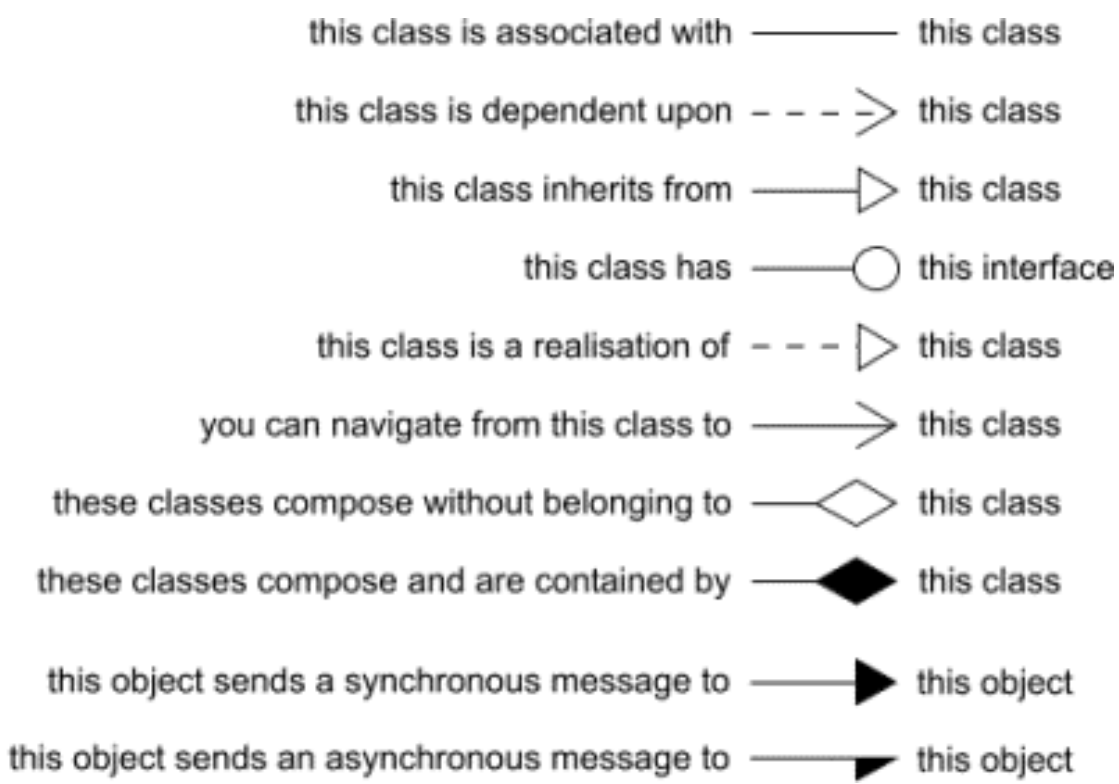
UML notations



The Class diagram is static diagram. It represents the static view of an application.

The Class diagram shows a collection of classes, interfaces, associations, collaborations and constraints.

The Class diagrams are widely used in the modelling of object oriented systems and they describe three different perspectives when designing a system, conceptual, specification, implementation.


this class is associated with
this class
this class is dependent upon
this class
this class inherits from
this class
this class has
this interface
this class is a realisation of
this class
you can navigate from this class to
this class
these classes compose without belonging to
this class
these classes compose and are contained by
this class
this object sends a synchronous message to
this object
this object sends an asynchronous message to
this object

Java basic data types

There are tow data types available in Java:

- Primitive Data Types
- Reference/Object Data types

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Primitive Data Types

Primitive data types are predefined by the language and named by a keyword.

Type	Size	Range	Default
boolean	1 bit	true or false	false
byte	8 bits	[-128, 127]	0
short	16 bits	[-32,768, 32,767]	0
char	16 bits	['\u0000', '\uffff'] or [0, 65535]	'\u0000'
int	32 bits	[-2,147,483,648 to 2,147,483,647]	0
long	64 bits	$[-2^{63}, 2^{63}-1]$	0
float	32 bits	32-bit IEEE 754 floating-point	0.0
double	64 bits	64-bit IEEE 754 floating-point	0.0

Reference/Object Data types

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed.
- Class object, and various type of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variables can be used to refer to any object of the declared type or any compatible type.
- Example: Animal **animal** = new Animal(“giraffe”); *(**animal** - variable)*

Access modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors.

The four access levels are:

- Visible to the package/default. No modifiers are needed.
- Visible to the class only (**private**).
- Visible to the world (**public**).
- Visible to the package and all subclasses (**protected**).

Default Access Modifier – No keyword:

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

Variables and methods can be declared without any modifiers, as in the following examples:

```
String version = "1.5.1";

boolean processOrder() {

      return true;

}
```

Private Access Modifier – private:

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.

```
public class Logger {
	private String format;
	public String getFormat() {
		return this.format;
	}
	public void setFormat(String format) {
		this.format = format;
	}
}
```

Public Access Modifier – public:

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

```
public class Bicycle {
	private int cadence;
	private int gear;
	private int speed;
	public int getCadence() {
		return cadence;
	}
	public void setCadence(int newValue) {
		cadence = newValue;
	}
}
```

Protected Access Modifier – protected:

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The following parent class uses protected access control, to allow its child class override*openSpeaker()* method:

```
class AudioPlayer {
	protected boolean openSpeaker(Speaker sp) {
		// implementation details
	}
}
class StreamingAudioPlayer {
	boolean openSpeaker(Speaker sp) {
		// implementation details
	}
 }
```

Non-Access Modifiers

Java provides a number of non-access modifiers to achieve many other functionality.

- The ***static*** modifier for creating class methods and variables.
- The ***final*** modifier for finalizing the implementations of classes, methods, and variables.
- The ***abstract*** modifier for creating abstract classes and methods.
- The ***synchronized*** and ***volatile*** modifiers, which are used for threads.

Static Modifier:

Static Variables:

The *static* key word is used to create variables that will exist independently of any instances created for the class. Only one copy of the static variable exists regardless of the number of instances of the class.

Static variables are also known as class variables. Local variables cannot be declared static.

Static Methods:

The *static* key word is used to create methods that will exist independently of any instances created for the class.

Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

Example:

The static modifier is used to create class methods and variables, as in the following example:

```
public class InstanceCounter {
private static int numInstances = 0;
      protected static int getCount() {
          return numInstances;
      }
      private static void addInstance() {
          numInstances++;
      }
      InstanceCounter() {
          InstanceCounter.addInstance();
      }
      public static void main(String[] arguments) {
          System.out.println("Starting with " +
          InstanceCounter.getCount() + " instances");
          for (int i = 0; i < 500; ++i){
               new InstanceCounter();
      }
      System.out.println("Created " +
      InstanceCounter.getCount() + " instances");
      }
}
```

The final Modifier:

final Variables:

A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object.

However the data within the object can be changed. So the state of the object can be changed but not the reference.

With variables, the *final* modifier often is used with *static* to make the constant a class variable.

```
public class Test{
    final int value = 10;
    // The following are examples of declaring constants:
    public static final int BOXWIDTH = 6;
    static final String TITLE = "Manager";

    public void changeValue(){
        value = 12;
    //will give an error
    }
}
```

final Methods:

A final method cannot be overridden by any subclasses. As mentioned previously the final modifier prevents a method from being modified in a subclass.

The main intention of making a method final would be that the content of the method should not be changed by any outsider.

```
public class Test{
     public final void changeName(){
     // body of method
     }
}
```

final Classes:

The main purpose of using a class being declared as *final* is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

```
public final class Test {
    // body of class
}
```

The abstract Modifier:

abstract Class:

An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

A class cannot be both abstract and final. (since a final class cannot be extended). If a class contains abstract methods then the class should be declared abstract. Otherwise a compile error will be thrown.

An abstract class may contain both abstract methods as well normal methods.

```
abstract class Caravan{
     private double price;
     private String model;
     private String year;
     public abstract void goFast(); //an abstract method
     public abstract void changeColor();
}
```

abstract Methods:

An abstract method is a method declared with out any implementation. The methods body(implementation) is provided by the subclass. Abstract methods can never be final or strict.

Any class that extends an abstract class must implement all the abstract methods of the super class unless the subclass is also an abstract class.

The abstract method ends with a semicolon. Example: public abstract sample();

```
public abstract class SuperClass{
    abstract void m(); //abstract method
}
class SubClass extends SuperClass{
    // implements the abstract method
    void m(){
            .........
    }
}
```

The synchronized Modifier:

The synchronized key word used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

```
public synchronized void showDetails(){
                .......
}
```

The transient Modifier:

An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.

This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.

```
public transient int limit = 55; // will not persist
public int b; // will persist
```

The volatile Modifier:

The volatile is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.

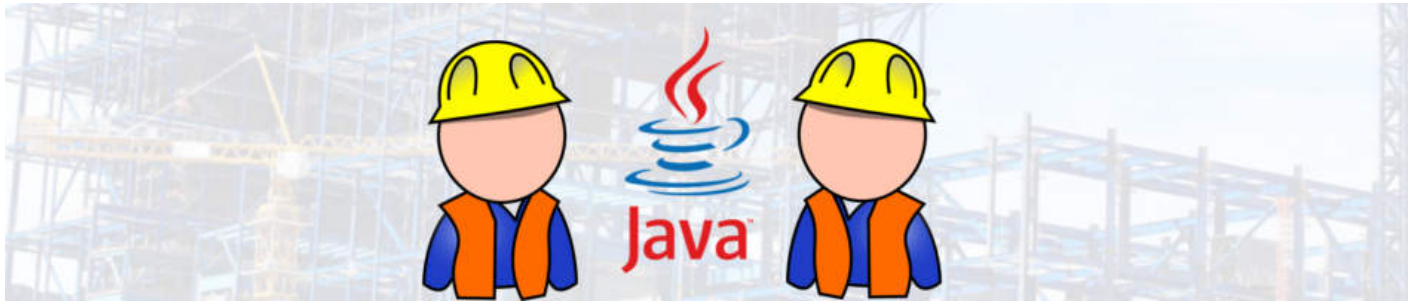
Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

```
public class MyRunnable implements Runnable{
	private volatile boolean active;

	public void run(){
		active = true;
		while (active){ // line 1
			// some code here
		}
	}

	public void stop(){
		active = false; // line 2
	}
}
```

Java constructor



A class contains constructors that are invoked to create objects from the class blueprint.

A **java constructor** has the same name as the name of the class to which it belongs. Constructor's syntax does not include a return type, since constructors never return a value.

Constructors may include parameters of various types. When the constructor is invoked using the new operator, the types must match those that are specified in the constructor definition.

Read more about Java constructors
https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html

http://www.javatpoint.com/constructor

Below is an example of a cube class containing 2 constructors. (one default and one parameterized constructor).

```
public class Cube1 {
	int length;
	int breadth;
	int height;
	public int getVolume() {
		return (length * breadth * height);
	}
	Cube1() {
		Length = 10;
		breadth = 10;
		height = 10;
	}
	Cube1(int l, int b, int h) {
		length = l;
		breadth = b;
		height = h;
	}
	public static void main(String[] args) {
		Cube1 cubeObj1, cubeObj2;
		cubeObj1 = new Cube1();
		cubeObj2 = new Cube1(10, 20, 30);
		System.out.println("Volume of Cube1 is : " + cubeObj1.getVolume());
		System.out.println("Volume of Cube1 is : " + cubeObj2.getVolume());
	}
}
```

Using the *this* keyword

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

```
        //this with a field

public class Point {
        public int x = 0;
        public int y = 0;

//constructor
        public Point(int x, int y) {
                this.x = x;
                this.y = y;
        }
}
```

```
            // this with a Constructor
public class Rectangle {
        private int x, y;
        private int width, height;

        public Rectangle() {
            this(0, 0, 1, 1);
        }
        public Rectangle(int width, int height) {
            this(0, 0, width, height);
        }
        public Rectangle(int x, int y, int width,
            int height) {
            this.x = x;
            this.y = y;
            this.width = width;
            this.height = height;
        }
    ...
}
```

Using the *void* keyword

Void is used when you are creating a class that will not return any value. Java always needs to know what to expect. So if you will get a string after to perform the action you would need to label string, if you expect a number you would label int, double, or whatever type of number will come back .

```
public void name() {
a= a + b;
}
```

void because value of **a** are just changing.

Using the *return* keyword

Returning a Value from a Method

A method returns to the code that invoked it when it

- completes all the statements in the method,
- reaches a return statement, or
- throws an exception (covered later),

whichever occurs first.

```
// a method for computing the area of the rectangle
public int getArea() {
return width * height;
}
```

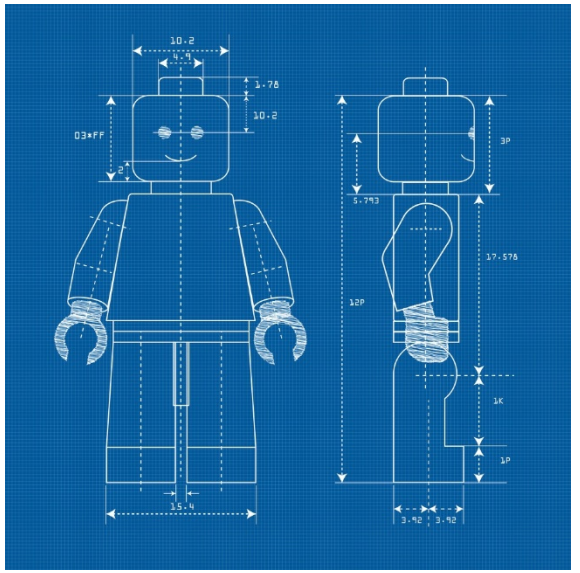
https://docs.oracle.com/javase/tutorial/java/javaOO/returnvalue.html

Objects & Classes

- **Object -** Objects have states and behaviors.

Example: A dog has states - color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.

- **Class -** A class can be defined as a template/blue print that describes the behaviors/states that object of its type support.



Creating an Object

There are three steps when creating an object from a class:

- Declaration: A variable declaration with a variable name with an object type.
- Instantiation: The 'new' key word is used to create the object.
- Initialization: The 'new' keyword is followed by a call to a constructor. This call initializes the new object.


```
public class Puppy{
    public Puppy(String name){
        // This constructor has one parameter, name.
    System.out.println("Passed Name is :" + name );
    }
    public static void main(String []args){
        // Following statement would create an object myPuppy
        Puppy myPuppy = new Puppy( "tommy" );
    }
}
```