Java is an object-oriented programming language.

Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has **attributes**, such as model, weight, color, and **methods**, such as toAccelerate() and toStop().

A Class is like an object constructor, or a "blueprint" for creating objects.

Create an Object

In Java, an object is created from a class. Class named MyClass can be used to create objects.
To create an object of MyClass, specify the class name, followed by the object name, and use the keyword new:

```
public class MyClass {
  int x = 5;
 // private int y;

  public static void main(String[] args) {
        // declare and create objects
        MyClass myObj = new MyClass();
        //myObj.y = 10;
        System.out.println(myObj.x);
  }
}
```

Using Multiple Classes

You can also create an object of a class and access it from another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the main() method (code to be executed)).

```
public class MyClass {
  int x = 5;
// private int y //we need a setters and getter
}

public class OtherClass {
  public static void main(String[] args) {
        MyClass myObj = new MyClass();
        System.out.println(myObj.x);
  }
}
```

Modify Attributes

```
public class MyClass {
  int x = 10;

  public static void main(String[] args) {
        MyClass myObj = new MyClass();
        myObj.x = 40;
        System.out.println(myObj.x);
  }
}
```

Access Methods(functions) With an Object

Functions are self-contained chunks of code that perform a specific task. You give a function a name that identifies what it does, and this name is used to “call” the function to perform its task when needed.

```
public class Car { // Create a Car class

  // Create a fullThrottle() method
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  // Create a speed() method and add a parameter
  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }

  // Inside main, call the methods on the myCar object
  public static void main(String[] args) {
        Car myCar = new Car(); // Create a myCar object
        myCar.fullThrottle(); // Call the fullThrottle() method
        myCar.speed(200);   // Call the speed() method
  }
}
//Outputs: The car is going as fast as it can!
// Max speed is: 200
```

Using Multiple Classes

```
public class Car {
        public void fullThrottle() {
        System.out.println("The car is going as fast as it can!");
 }

 public void speed(int maxSpeed) {
        System.out.println("Max speed is: " + maxSpeed);
 }
}

class OtherClass { //app runner class
 public static void main(String[] args) {
   Car myCar = new Car();     // Create a myCar object
   myCar.fullThrottle();      // Call the fullThrottle() method
   myCar.speed(200);          // Call the speed() method
 }
}
```

A **constructor** in Java is a **special method** that is used to **initialize** objects.(it allows you to initialize variables as soon as you create an object). Constructor is like a construction.
The constructor is called when an object of a class is created. It allows you to give value as soon as you create an object:

```
// Create a MyClass class
public class MyClass {
  private int x;  // Create a class attribute, only MyClass knows about x

  // Create a class constructor for the MyClass class, should be the same as class
  public MyClass() {
        System.out.println("This is the constructor");
        this.x = 5;  // Set the initial value for the class attribute x
 }

  public static void main(String[] args) {
        MyClass myObj = new MyClass(); // Create an object of class MyClass ( new
MyClass() is the default constructor)
        //Scanner sc = new Scanner(System.in);
        System.out.println(myObj.x); // Print the value of x
 }
}
```

Constructor Parameters

```
public class Car {
  private int modelYear;
  private String modelName;

  public Car(int year, String modelName) {
        modelYear = year;// if we are not assigning it will be null
        this.modelName = modelName;//this if we are using same name
  }

  public static void main(String[] args) {
        Car myCar = new Car(1989, "Mustang");// Create an object of class
        System.out.println(myCar.modelYear + " " + myCar.modelName);
  }
}// Outputs 1989 Mustang
```

Encapsulation

Encapsulation group the data and the code which handles that data together to protect or hide data from implementation details from the other classes.

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, we must:

- One way to hide the implementation details is to declare class variables/attributes as `private` (only accessible within the same class)
- To allow access to these private data members we need to provide **setter** and **getter** methods to access and update the value of a `private` variable

Get and Set

`private` variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public **getter** and **setter** methods.
The `get` method returns the variable value, and the `set` method sets the value.
Syntax for both is that they start with either `get` or `set`, followed by the name of the variable, with the first letter in upper case:

```
public class Person {
  private String name; // private = restricted access
  // Getter
  public String getName() {
    return name;
  }
```
The get method returns the value of the variable name.

```
  // Setter
  public void setName(String newName) {
    name = newName;
  }
```
The set method takes a parameter (newName) and assigns it to the name variable.
}

Why Encapsulation?

- Better control of class attributes and methods
- Class variables can be made **read-only** (if you commit the setmethod), or **write-only** (if you commit the getmethod)
- Flexible: the programmer can change one part of the code without affecting other parts
- Increased security of data

Java Inheritance (Subclass and Superclass)

Java supports inheritance, which means data inside parent or super class can be inherited into its child or subclass. Extend means using existing and adding some new features in thing or object. Java also use extends in same way. It is used to inherit the feature of parent class and we can add extra feature also to child class as well.
***Main purpose of Inheritance is to make code shorter, easy to right, easy to read and maintain*.**

Example without inheritance:

```
class Employee {
    public String name;
    public double salary;
    public Date birthDate;
    public String getDetails()
}
class Manager {
    public String name;
    public double salary;
    public Date birthDate;
    public String department;
    public String getDetails()
}
```

Example with inheritance:

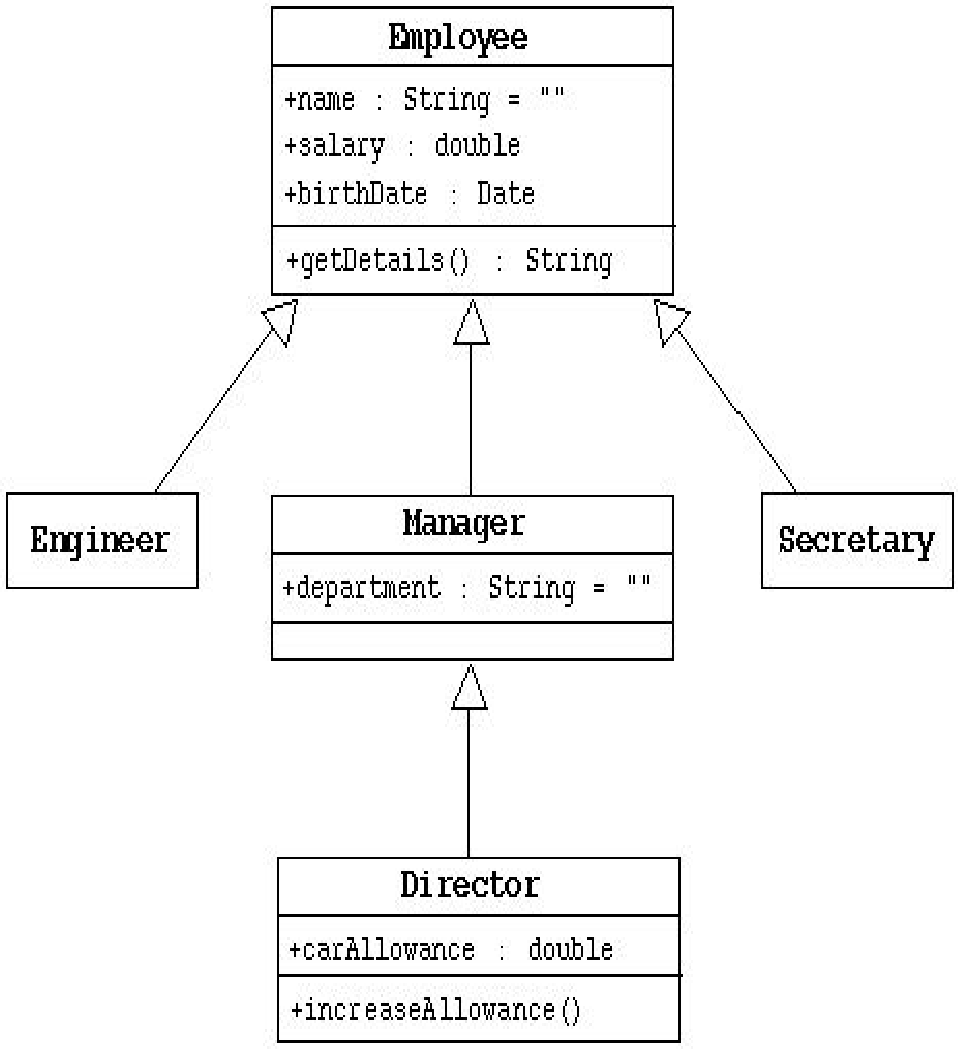
```
class Employee {
    public String name = "";
    public double salary;
    public Date birthDate;
    public String getDetails()
}

class Manager extends Employee{
    public String department;
}
```

Class, from which is inherited is called **extended/ parent/ base/ general/ super-class**

Class which inherits, is called **extending/ child/ derived/special/sub-Class**

UML Diagram



In Java **child class can extend only one parent class**.

Inheritance: **super**
Constructors are not inherited by default, **super** keyword is used to refer to parent class, for both properties and methods of the parent class.

```
class Employee {
  public String name;
  public double salary;
  public Date birthDate;

  public String getDetails() {
    return "Name: " + name +
    "Salary: " + salary;
  }
}

class Manager extends Employee {
  public String department;
  public String getDetails() {
    // call parent method
    return super.getDetails() +  "Department: " + department; //subclass method from super-calss method

  }
}
```

Java Polymorphism

Polymorphism is a concept by which we can perform a *single action in different ways*. Polymorphism comes from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means **many forms**, and it occurs by many classes that are related to each other by **inheritance**.

Pig and Dog objects will call the animalSound() method for both of them:

```
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound,");
  }
}

class Pig extends Animal {
  public void animalSound() {
        System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
        System.out.println("The dog says: bow wow");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();        // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object
        myAnimal.animalSound();
        myPig.animalSound();
        myDog.animalSound();
  }
}

// The animal makes a sound
// The pig says: wee wee
// The dog says: bow wow
```

Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-class or parent class.

```
class Animal {
   public void move() {
      System.out.println("Animals can move");
   }
}


class Dog extends Animal {
@Override
   public void move() {//can be method from different packages or classes
      System.out.println("Dogs can walk and run");
   }

}

public class TestDog {
   public static void main(String args[]) {
      Animal a = new Animal();   // Animal reference and object
      Animal b = new Dog();   // Animal reference but Dog object

      a.move();   // runs the method in Animal class
      b.move();   // runs the method in Dog class
   }

}

//Animals can move
//Dogs can walk and run
```

- Subclass can modify behavior which is inherited from superclass
- Subclass can define a method with the same "signature", with unique set of:
  - Passed parameters and their types
  - Returned type

Overloading

One class may contain multiple methods that have the same behaviour (i.e. perform the same tasks) but work on different types of data.

In Java, we can declare **multiple methods with the same name in a single class**, this is called **overloading** methods.

Overloading allows different methods to have the same name, but different signatures where the signature with different parameters.

```
    public class Sum {

        public int sum(int x, int y) {
                return (x + y);
        }
        // Overloaded sum(). This sum takes three int parameters
        public int sum(int x, int y, int z) {
                return (x + y + z);
        }
        // Overloaded sum(). This sum takes two double parameters
        public double sum(double x, double y) {
                return (x + y);
        }
        public static void main(String args[]) {
                Sum s = new Sum(); // Create a s object
                System.out.println(s.sum(10, 20));
                System.out.println(s.sum(10, 20, 30));
                System.out.println(s.sum(10.5, 20.5));
        }
}
```

What is the advantage?

We don’t have to create and remember different names for functions doing the same thing. For example, in our code, if overloading was not supported by Java, we would have to create method names like sum1, sum2, … or sum2Int, sum3Int, … etc.

Method **overloading** is when the same method is defined with different set of parameters or returned type

Method **overriding(re-implement)** is when subclass redefines implementation of method inherited from superclass with the same signature

The `instanceof` keyword checks whether an object is an instance of a specific class or an interface.

The instanceof keyword compares the instance with type. The return value is either `true` or `false`.

```
     public class MyClass {

          public static void main(String[] args) {

          MyClass myObj = new MyClass();

     System.out.println(myObj instanceof MyClass); // returns true

  }

}
```

Downcasting In Java

Abstraction in Java

Abstract is a simple class that usually contains at least one abstract method, and without any body.
Abstraction is a process of hiding the implementation details and showing only functionality to the user.
Another way, it shows only essential things to the user and hides the internal details.

```
public abstract class Animal { // Abstract class exist to be extended, they can't be instantiated
   public abstract void animalSound();//Abstract method (body is empty)

  public void sleep() {// Regular method
    System.out.println("Zzz");
  }
}
```

By adding this abstract modifier to the class, is preventing us from being able to create an instance of Animal object from this class.

```
class Pig extends Animal {// Subclass (inherit from Animal)
@Override
public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}
class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig();
    myPig.animalSound();
    myPig.sleep();
   }
}
```

Java Interface

Interface contains only abstract methods and interfaces which can't be instantiated.

Like a class, an interface can have methods and variables, but the methods declared in interface are by default - only method signature and no body.

```
  public  interface Animal{
          // by default, it's like a template.
         public void animalSound()
    }

class Pig implements Animal {// Subclass (inherit from Animal)
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}
```

Interface are used when

- We need to declare methods which we need to implement in several classes.
- When we need to expose class methods without exposing class itself

Interface vs Abstract class vs Concrete class

UML (Unified Modeling Language)

UML is a graphical language.

UML is the current standard for programming in object-oriented programming languages. When creating classes and other objects with relationships between each other, UML is what is used to visually describe these relationships.

To help system and developers for visualizing, and documenting software systems.

UML helps to plan a program before the programming takes place. This can help reduce overhead/costs during the implementation stage of any program. Additionally:

- UML model diagram is easy to change.
- UML helps to organize, plan and visualize a program
- Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

A UML is a visual representation of the relationships between classes and entities in a computer program. To understand a program, it is essential to understand what each class object does, the information it stores and how it relates to other classes in the program. By showing this information in a diagram, it is easy to understand and visualize a program's relationships.

UML Diagram Types:

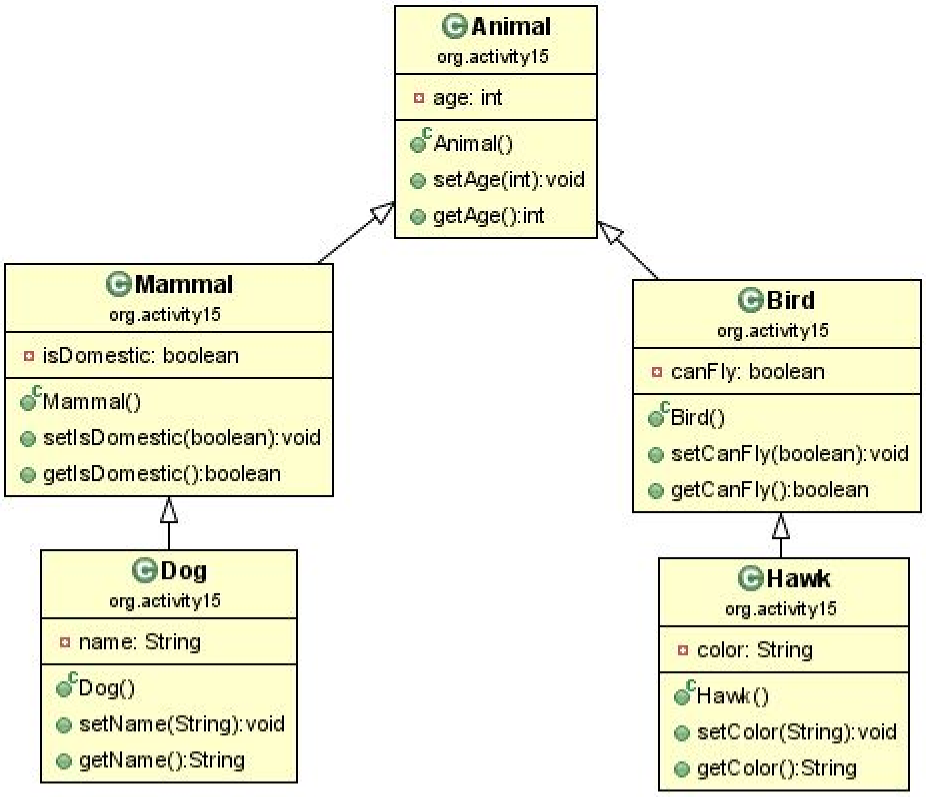
There are two main categories:

- Structure Diagrams
  - Class Diagram
  - Component Diagram
  - Deployment Diagram
  - Object Diagram
  - Package Diagram
  - Profile Diagram
  - Composite Structure Diagram
- Behavioral Diagrams
  - Use Case Diagram
  - Activity Diagram

- State Machine Diagram
- Sequence Diagram
- Communication Diagram
- Interaction Overview Diagram
- Timing Diagram

1. Class diagram

Class diagrams are the most important UML diagram and are very important in software development. Class diagrams are the best way to illustrate a system's structure in a detailed way, showing its attributes, operations as well as its relationships.



The main purpose of the class diagram is to analysis and design of the static view of an application and describe the responsibilities of a system.

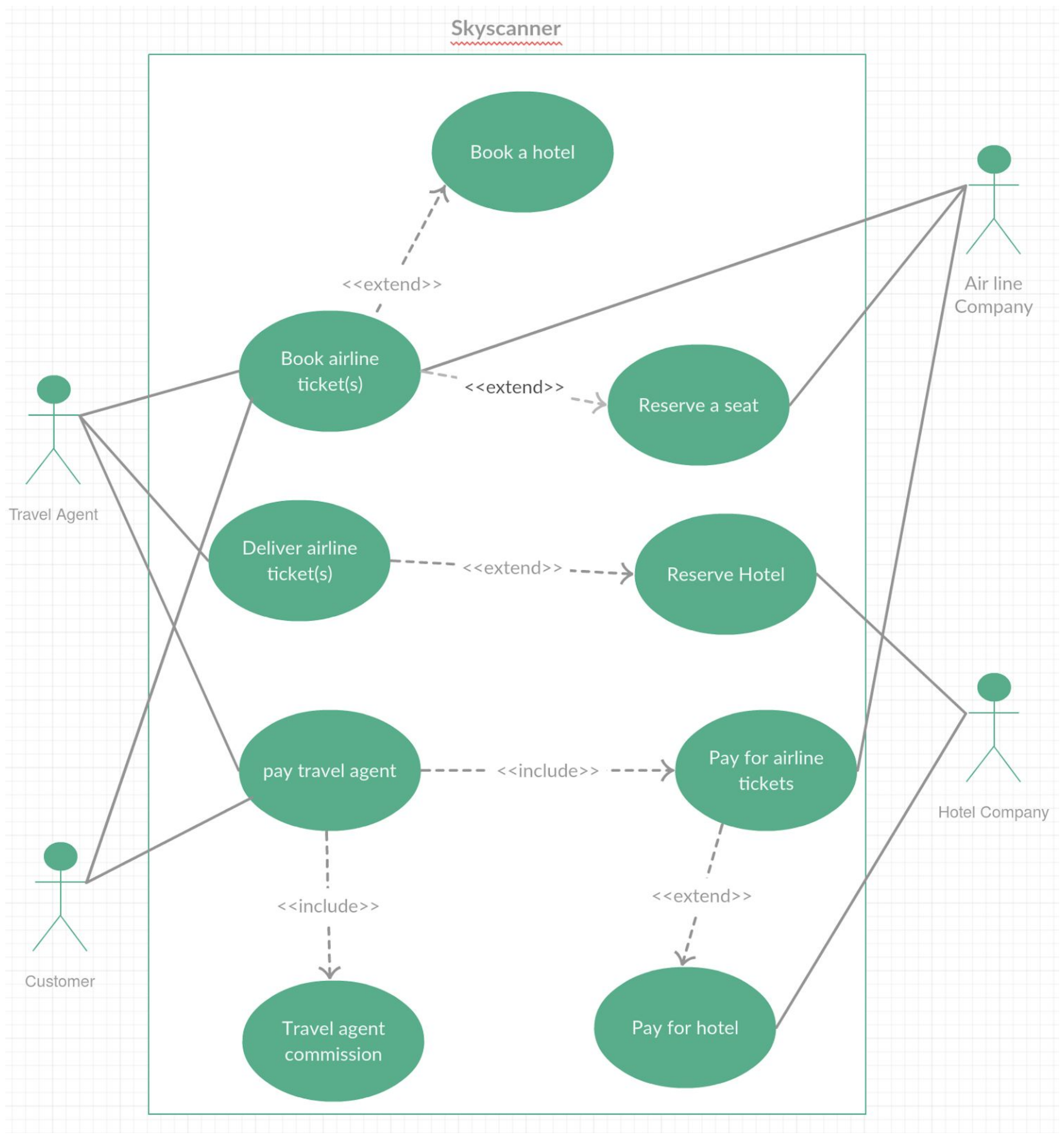
- Class diagrams give you a sense of orientation,
- Provide detailed structure info of your systems. At the same time they offer a quick overview of the properties and relationships.
- Class diagrams are simple and fast to read.
- With the right software they are also easy to create.

2. Use Class diagram

Use Case diagrams show the various activities the users can perform on the system.

Use case can evolve at each iteration from a method of capturing requirements, to development guidelines to programmers. Use case perform a test case and finalize into user documentation.

The purpose of use case diagram is to show dynamic aspects of the system in very high abstraction. Usually developers are using **Use Class diagram** when you have already functional app and you want to add more features to it.



Actor — human user, other internal or external application

Use case — used function of the presented system

Association — association between different functionalities of use cases

Generalization — parent/child general/special relationship between use cases includes -> reuse of functionality

extends -> new and/or optional functionality

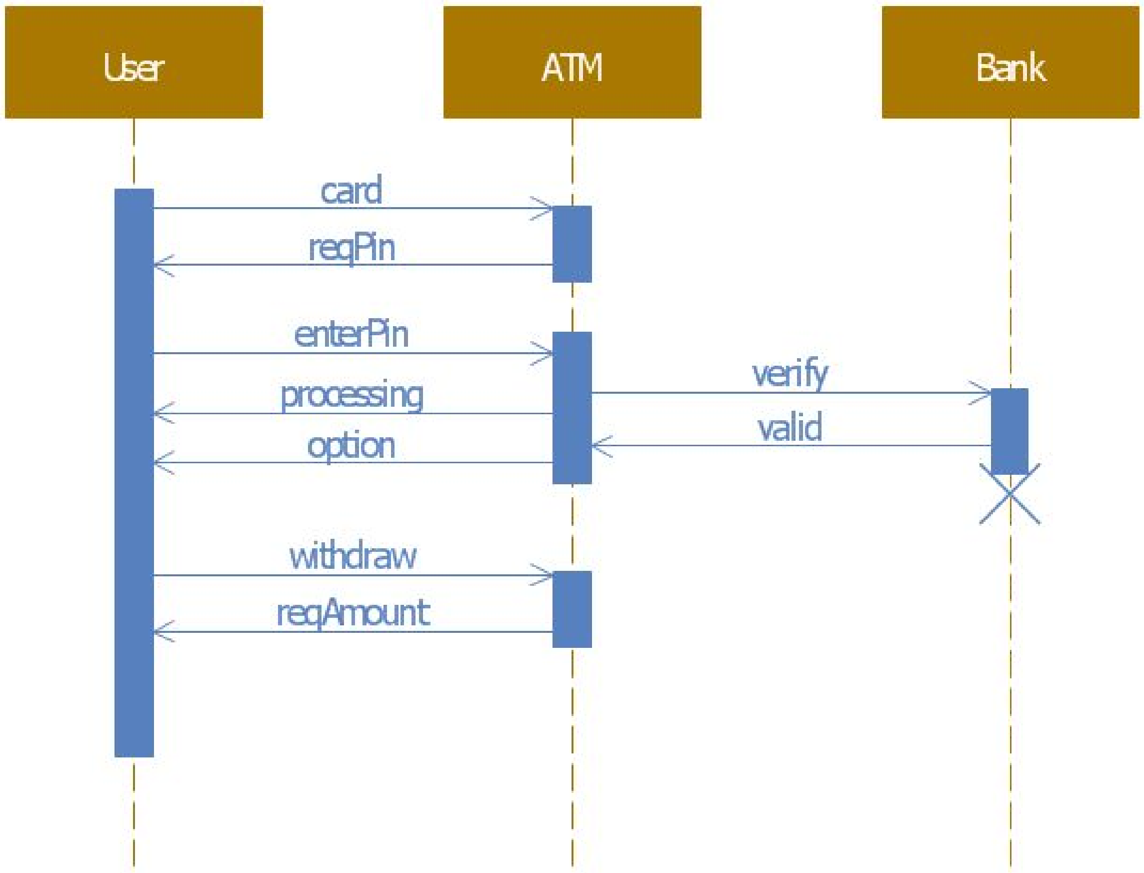
"Book a hotel" may not be required, but may optionally be required based on your decision of your use case.

3. Sequence diagram

A Sequence diagram **-** event diagrams

A Sequence diagram is an interaction diagram that shows how processes operate with one another and what is their order.

A sequence diagram is a good way to visualize and validate various runtime scenarios. These can help to predict how a system will behave and to discover responsibilities a class may need to have in the process of modeling a new system.



Benefits of using UML **sequence diagrams:**

- Help you to discover architectural, interface and logic problems early
- **Collaboration tool -** allow you to discuss the design in specific terms
- **Documentation -** the diagrams can abstract much of the implementation detail and provide a high level view of system behavior.

UML Diagram Guide

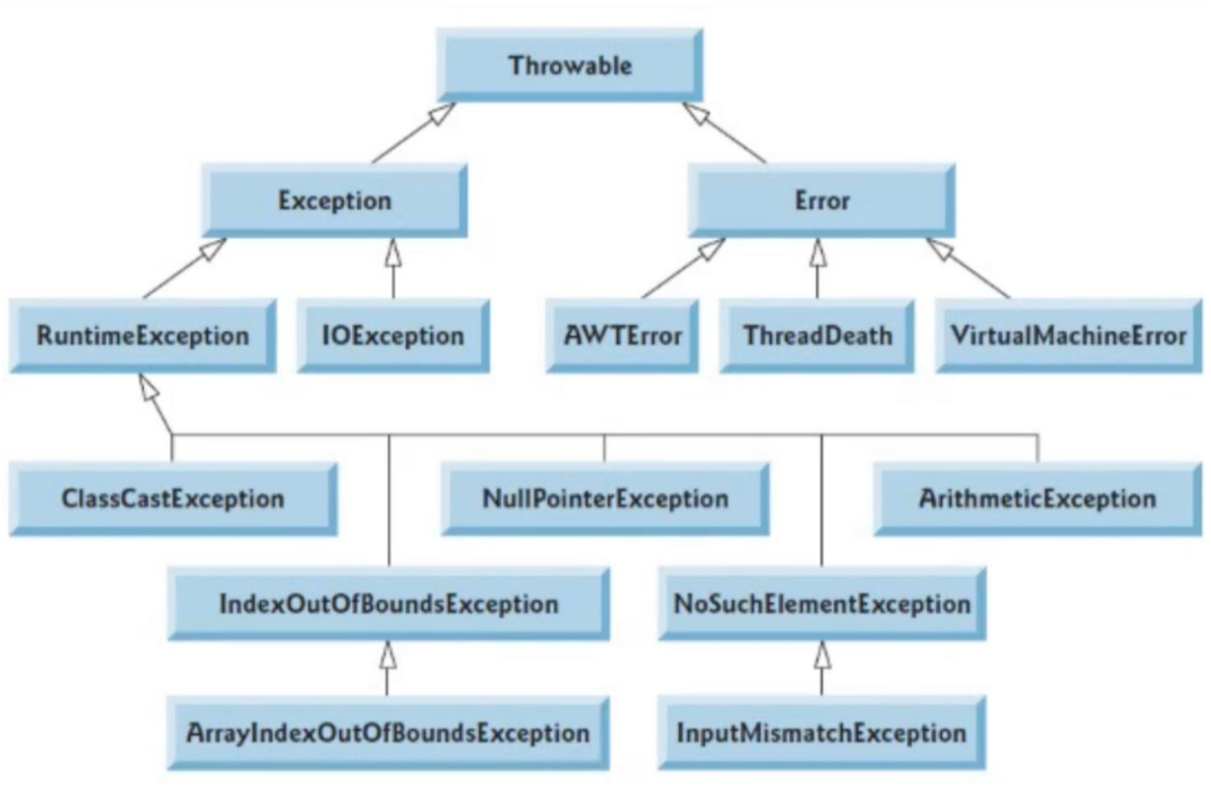
Exceptions and Assertions

Exception is an event that performs in that way, which we didn't expect, so something went wrong !

Exception examples:

- Trying to open non-existing file
- Interrupted network connection
- Passed values outside the valid range
- Trying to load non-existing .class file

Throwable class is the superclass of all errors and exceptions in the Java language.



Exception:

- ArithmeticException — can appear during arithmetic operations
- InputMismatchException — can appear, when passed parameters are with wrong values

- **Throwable** is the base class that defines everything that can be thrown.
- **Exception** is the common case. It is about problems that occur during the execution of your program.
- **Error** is the "rare" case: it signifies problems that are outside the control of the usual application:
  - JVM errors, out of memory, problems verifying bytecode: these are things that you *should not handle* because if they occur things are already so bad that your code is likely to bad.

throws keyword is used to declare that a method may throw one or some exceptions.

When you want to specific exception to be thrown you need to go for "**Try/Catch**" block.
If your method does not bother you with what operation it is doing and exceptions may come there, then you will go for "Throws" block which says this method may throws exceptions.

Assertions

The Java ***assert*** keyword allows developers to quickly verify certain state of a program.

An assertion is a statement in which ensures the correctness of any goals which have been done in the program. When an assertion is executed, it is assumed to be true.
If the assertion is false, the JVM will throw an Assertion error. Assertion statements are used along with boolean expressions.

```
public class Example {
   public static void main(String[] args) {
     Connection conn = getConnection();
     assert conn != null : "Connection is null";
   }

}
```

the code is checking that a connection to an external resource returns a non-null value. If that value is *null,* the JVM will **automatically throw an *AssertionError***.

Using Java Assertions

Collection

A *collection* represents a group of objects, known as its elements, that can hold references to other objects.

Collection interface is generally to deal with data structure which defines certain method to be implemented by its subclass or extends by an interface.

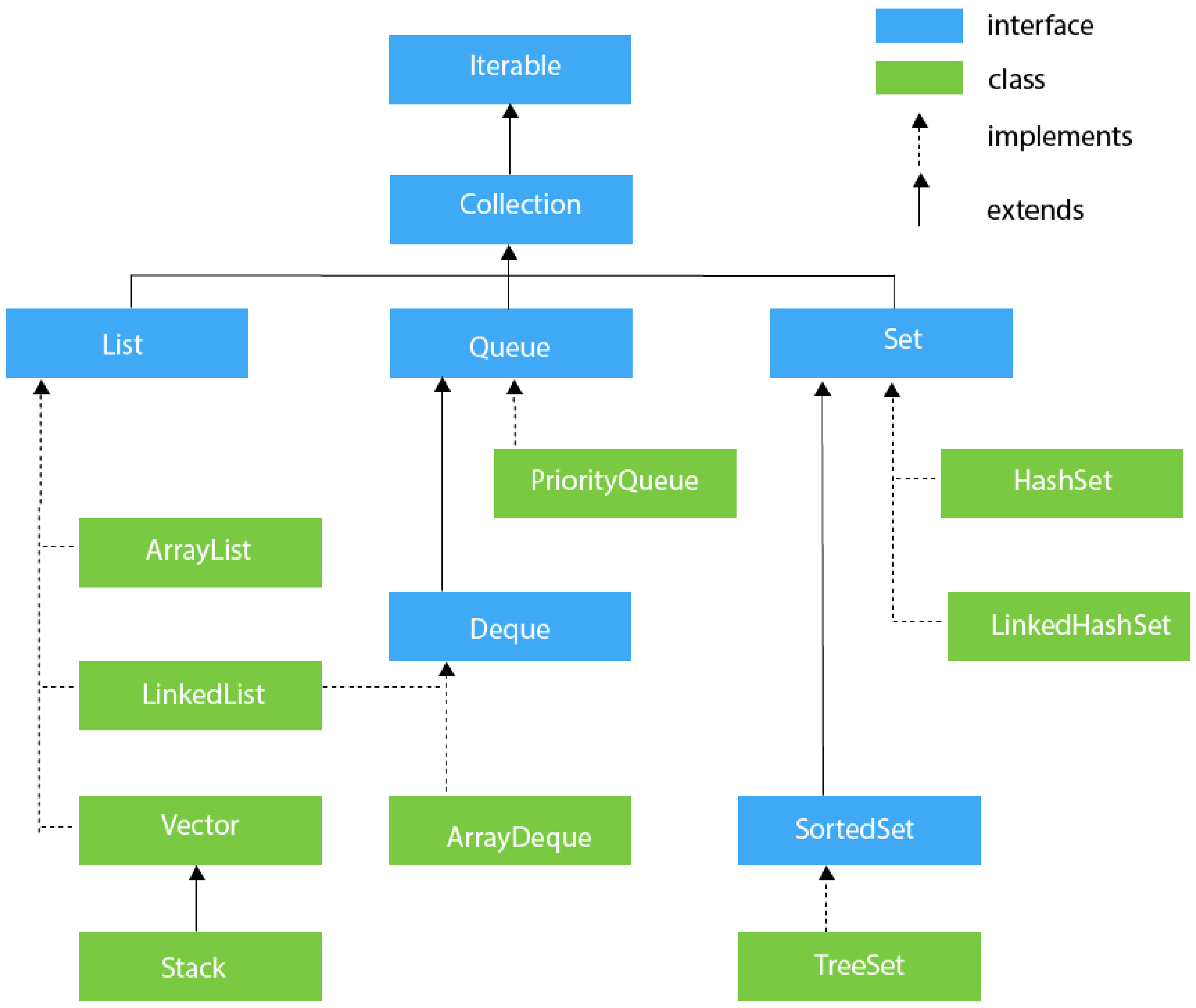
Simply speaking java introduce collection interface to deal with multiple objects which has similar type.

Collection size not fixed it means as we are inserting and deleting the elements the size of the collection dynamically increased or decreased.

Collection interface is generally to deal with data structures, it means we need to write new logic for performing the operations like insert, delete, search, sorting etc.

All the collection classes are available in "java.util"(utility) package.

Hierarchy of Collection Framework



List:

List interface is the child interface of Collection interface.

This category is used to store group of individuals elements where the elements can be duplicated.

To work with this category we have to use following implementations class of list interface.

ArrayList, **LinkedList**

LinkedList - It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order.

```
  public class JavaCollection{

        public static void main(String args[]){

                LinkedList<String> name=new LinkedList<String>();

                name.add("Tom");

                name.add("Sam");

                name.add("Mike");

                name.add("Mike");

        Iterator<String> itr=name.iterator();

        while(itr.hasNext()){

        System.out.println(itr.next());

   }

  }

}
```

Iterators are used in Collection framework in Java to retrieve elements one by one

Set:

This category is used to store a group of individual elements. But they elements can't be duplicated.

To work with this category we have to use following implementations class of Set interface.

LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted. That is, when cycling through a LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.

```
public static void main(String args[]) {
      // create a hash set
      LinkedHashSet hs = new LinkedHashSet();
       // add elements to the hash set
      hs.add("B");
      hs.add("A");
      hs.add("D");
      hs.add("E");
      hs.add("C");
      hs.add("F");
      hs.add("F");
      System.out.println(hs);
   }
}
//output B, A, D, E, C, F
```

Map:

This category is used to store the element in the form **key: value** pairs where the keys can't be duplicated.

To work with this category we have to use following implementation classes of Map interface.

HashMap, LinkedHashMap, TreeMap, HashTable

- Both List, Set are extending from Collection Interface.

-Map is part of collection Framework but not extending from collection Interfaces.

JavaScript Object Notation - is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.

JSON is build on 2 structures:

- A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.

```
dat:[

{

 "first_name": "John",

 "lastName": "Smith",

 "age": 27,

 "address": {

  "streetAddress": "21 2nd Street",

  "city": "New York",

  "state": "NY",

 },[

 "phoneNumbers": {

  {

   "type": "home",

   "number": "212 555-1234"

  },



   "type": "office",

   "number": "646 555-4567"

  },

  {

   "type": "mobile",
      "number": "123 456-7890"
    }
  ],
}
]
```

```
import java.util.*;
class HashMap1{
 public static void main(String args[]){
   HashMap<Integer,String> hm=new HashMap<Integer,String>();
    System.out.println("Initial list of elements: "+hm);
      hm.put(100,"Tom");
      hm.put(101,"Tedd");
      hm.put(102,"Tom");

      System.out.println("After invoking put() method ");
      for(Map.Entry m:hm.entrySet()){
       System.out.println(m.getKey()+" "+m.getValue());
      }

      HashMap<Integer,String> map=new HashMap<Integer,String>();
      map.put(103,"Rose");
      map.putAll(hm);
      System.out.println("After invoking putAll() method ");
      for(Map.Entry m:map.entrySet()){
           System.out.println(m.getKey()+" "+m.getValue());
          }
     }
}
```

Java I/O

Java I/O (Input and Output) is used *to process input* and *produce output*.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented

2) System.in: This is used to feed the data to user's program and usually a keyboard.

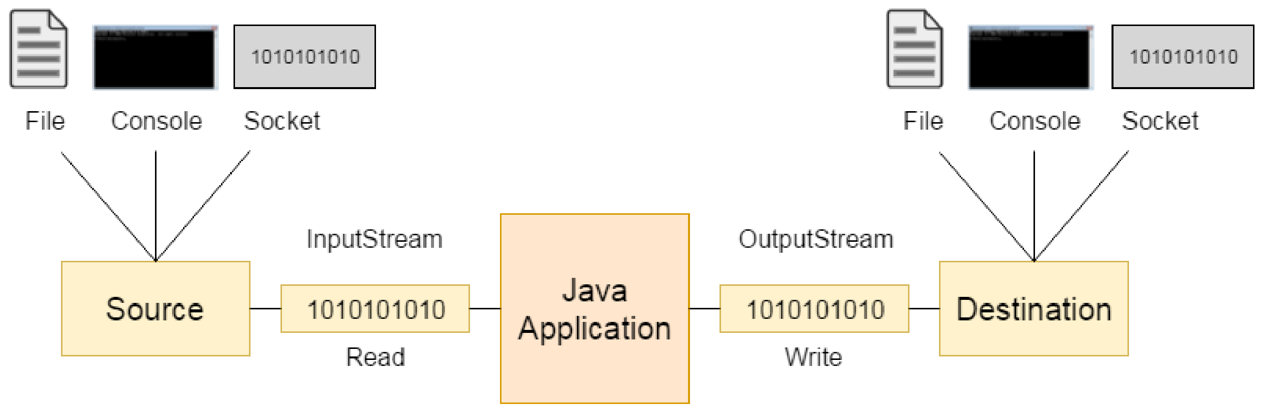
3) System.err: This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented.

InputStream

Java application uses an input stream to read data from a source

OutputStream

Java application uses an output stream to write data to a destination



A stream is an ordered sequence of bytes of length.

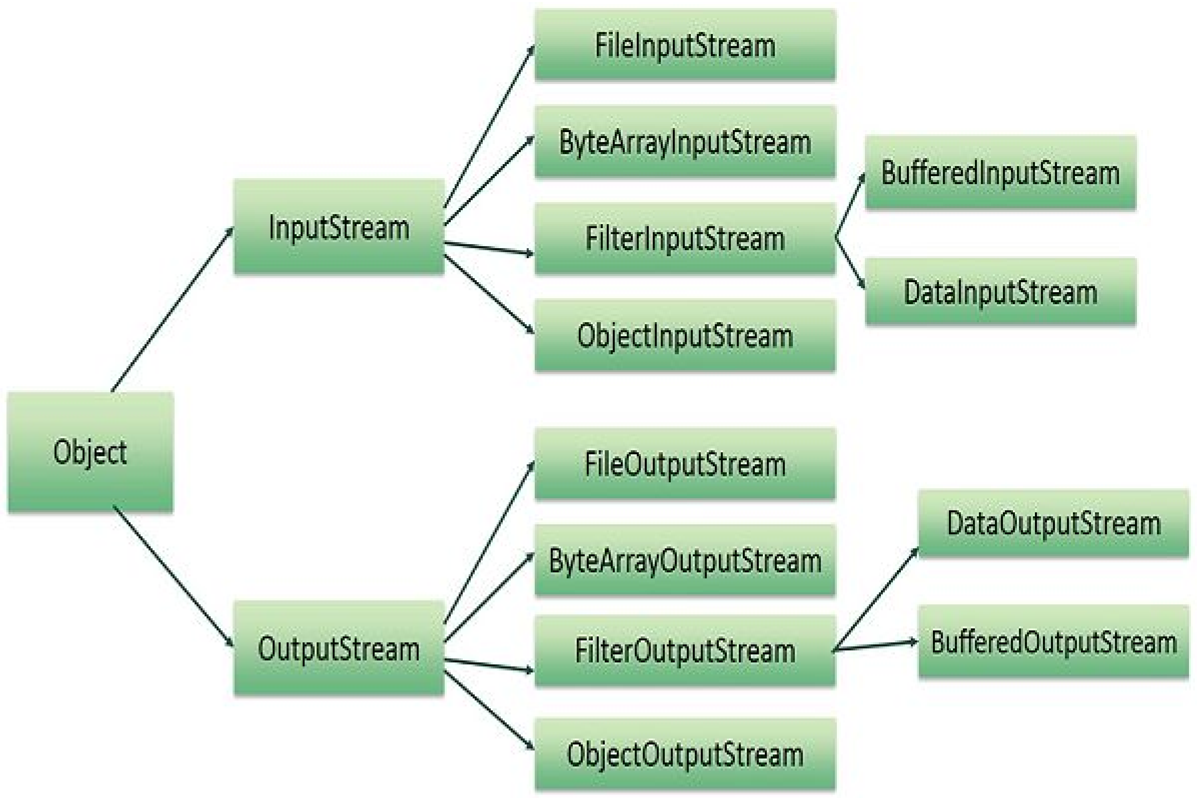
Input streams move bytes of data into a Java program from some generally external source.

Output streams move bytes of data from Java to some generally external target.

BufferedReader and BufferedWriter

BufferedWriter output stream,writes data, while buffering characters to provide for the efficient writing of single characters, strings and arrays.

BufferedReader input stream, reads data from a memory area known as a buffer and it reads text from a character-input stream which is inside lines and arrays.



Scanner vs InputStreamReader

BufferedReader/InputStreamReader you can read the whole document character by character. With scanner this is no possible. It means that with the InputStreamReader you can have more control over the content of the document.

In terms of performance, Scanner is definitely the slower one. It's made for parsing, not reading huge blocks of data. InputStreamReader has a large enough buffer, can perform with BufferedReader which is few times faster than Scanner.

Threads

Most modern operating systems support threads, and the concept of threads has been around in various forms for many years. Java is the first mainstream programming language to explicitly include threading within the language itself.

UI example:

Main Thread **Background Thread**

For example, in UI application, you want to draw screens at the same time and you also want to capture user's action e.g. pressing keys and command and you may want to download or uploading something from the network.

If you do all this task in one thread, this may cause a problem for your application because UI seems to be frozen while you doing another task. By using multiple threads in Java you can execute each of this task independently.

A single process frequently may contain multiple threads. Threads can also directly communicate with each other since they share the same variables.

Every Java program has at least one thread — the main thread. When a Java program starts, the JVM creates the main thread and calls the program's main() method within that thread.

In one word, we use Threads to make Java application faster by doing multiple things at the same time. In technical terms, Thread helps you to achieve parallelism(it means that an application splits its tasks up into smaller subtasks) in Java program.

Reasons for using Multithreading in Java:

1) To make a task run parallel to another task e.g. drawing and event handling.
2) To take full advantage of CPU power.
3) For reducing response time
4) To sever multiple clients at the same time.

Thread.sleep() method can be used to pause the execution of current thread for specified time in milliseconds. The argument value for milliseconds can't be negative, else it throws IllegalArgumentException.

Thread.sleep() interacts with the thread scheduler to put the current thread in wait state for specified period of time. Once the wait time is over, thread state is changed to runnable state and wait for the CPU for further execution. So the actual time that current thread sleep depends on the thread scheduler that is part of operating system.

Networking

Networking refers to how the connected computers communicate.

Java.net package which contains the class for is helping establish connections between computers and then send messages between them.

Computers may also communicate across a private network (called intranets). In office workers don't usually have a printer at their desks, they share a printer. When they print document, the document is sent to the printer over the company’s intranet.

Same machine are communicating with each other.. When discussing networking, a machine is usually referred to as a host.

A common network configuration that you've probably heard of is client/server, meaning that one (or more) hosts on the network and clients that connect to the server. The browser is a client, web addresses connects to the server that has files for the web site.

Computer on a network which includes the internet communicate with each other using transport protocols TCP.

Transmission control protocol (TCP) is a network communication protocol designed to send data over the Internet.

Each application that needs data from the network is assigned a port (this includes clients connecting to a server). When data arrives, the port is used to route the data to the application that's waiting for it.

IPV4 addresses uses a 32-bit address scheme that allows for over 4 billion unique addresses. But now we have computers, tablets, game consoles, smart TVs, smart Internet, and each device has to have a unique IP address. Four billion IP addresses weren't enough, and so IPv6 was born. It uses a 128-bit address scheme, which allows much more IP addresses than IPV4 does. IPv4 addresses are written as **four integers**, separated by dots. IPV6 addresses are written in **hexadecimal and separated by colons**.

IP stands for Internet Protocol. TCP/IP - refers to using the TCP protocol with IP addresses, which doesn't necessarily connected to the Internet. Two applications running on the same host communicate with each other. When the client and server are on the same host, usually the IP address 127.0.0.1, which is referred to as localhost, is used to identify the host.

When communicating using TCP/IP, the sequence of events is as follows:

- 1.The client opens a connection to the server
- 2.The clients sends a request to the server
- 3.The server sends a response to the client
- 4.The client closes the connection to the server.

Steps 2 and 3 may be repeated multiple times before the connection is closed.

When using the networking API, you'll send requests, and receive responses. A **socket** is an endpoint for communication between two machines.

The client will have a socket, and the server will have a socket. When you have multiple clients connecting to the same server, each client will have its own socket.

Java provide the IP address and port when you create the socket for us. You don't have to understand how TCP/IP works, Java will do for us - specific connection will establish between the client and the server, and the data has to be sent as packets, which must be in specific data format.

The server, and the client app.

JDBC

JDBC stands for Java Database Connectivity. **JDBC**is a Java API to connect and execute the query with the database.
JDBC driver supports SQL(Structured Query Language) and any other DB.

JDBC interface API provides:

- Connection to DB
- Creation SQL statements
- Execution SQL statements performed by DB
- Retrieve returned results from DB

```
import java.sql.*;

public class TeacherManager {

    protected Connection conn;

    private static Logger log = Logger.getLogger(TeacherManager.class);

    public TeacherManager() {


        if (conn == null) {
            try {

//Before connection to database it is necessary to load the driver
                    Class.forName("com.mysql.jdbc.Driver");
                    conn = DriverManager.getConnection("jdbc:mysql:
//localhost/?autoReconnect=true&useSSL=false", "root",
                              "abcd1234");//creating connection

//Database operations are performed using Statement object:
PreparedStatement stmt = conn.prepareStatement();

//Execute the statement and store results in ResultSet object
ResultSet rs = stmt.executeQuery("select firstname, lastname from
teachers");
//Result handling Results which return more than 1 object are stored in
ResultSet object

// ResultSet next moves iterator to next record, and if there is no any, it
returns false. Otherwise it points to first record
while (rs.next()) {
    results.add(new Teacher(rs.getInt("ID"), rs.getString("firstname"),
rs.getString("lastname")));
}
rs.close();
```

```
            conn.setAutoCommit(false);
        } catch (Exception e) {
            log.debug(e.getMessage());
        }
    }
  }
}
```

SQL is a standard language for accessing and manipulating databases.

SQL statements are simplify English language statements who tells to SQL DB server what those statements should do.

As Java is strong typed language, it has three methods with different names and returned types:

- boolean **execute**()
- Result set **executeQuery**() — Select operation
- Int number **executeUpdate**() — row count for inserted, updated or deleted records, or 0 if statement return nothing

Select statement

Select statement is used to retrieve records forme from one or more tables

https://www.w3schools.com/sql/sql_select.asp