תכנות מונחה עצמים
תרגול 2

מייל: ofrit@ariel.ac.il

נכתב ע"י: עופרי תבור

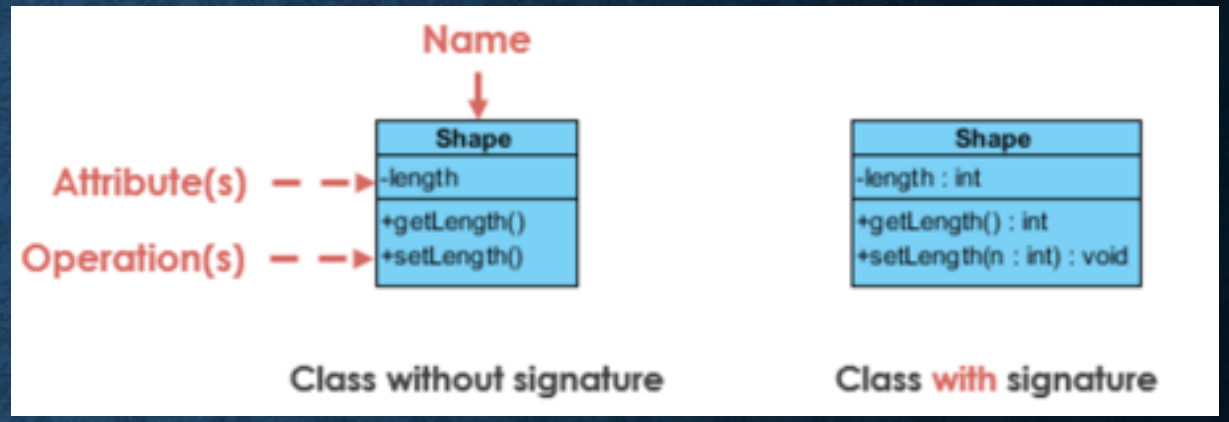
נושאים להיום:

- UML- דיאגרמת מחלקות
- טיפול בשגיאות וחריגים
- טיפול בקבצים
- Json
- Junit

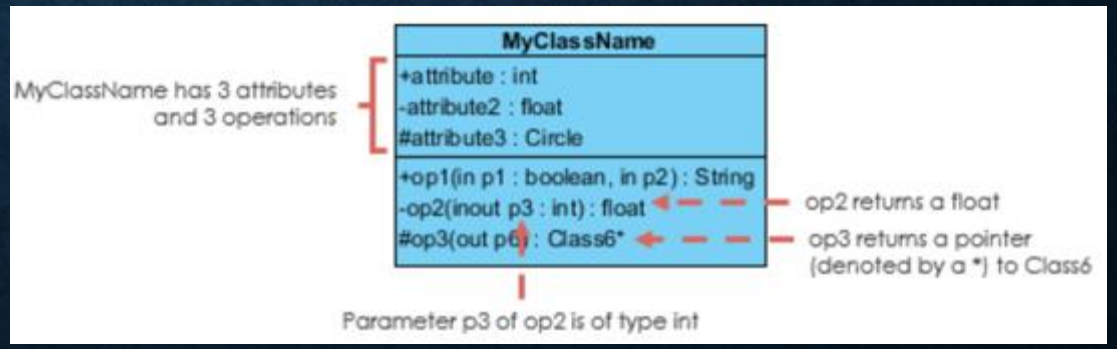
UML

UML Class Notation

- A class represent a concept which encapsulates state (**attributes**) and behavior (**operations**). Each attribute has a type. Each **operation** has a **signature**. *The class name is the **only mandatory information***.
- **Class Name:**
- The name of the class appears in the first partition.
- **Class Attributes:**
- Attributes are shown in the second partition.
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.



- **Class Operations (Methods):**
- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code

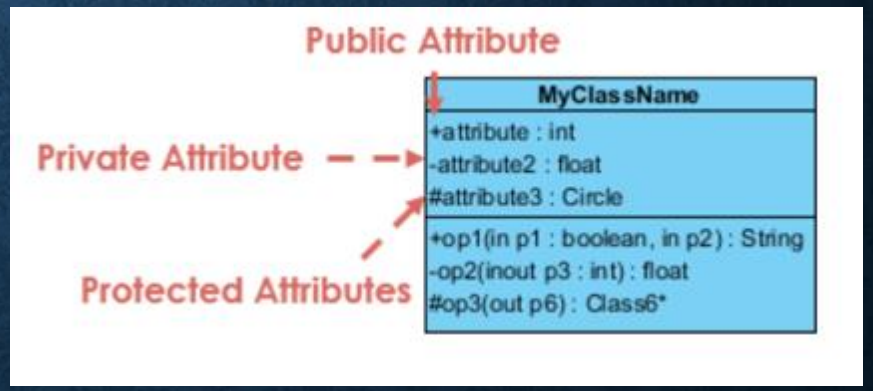


UML

Class Visibility

The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.

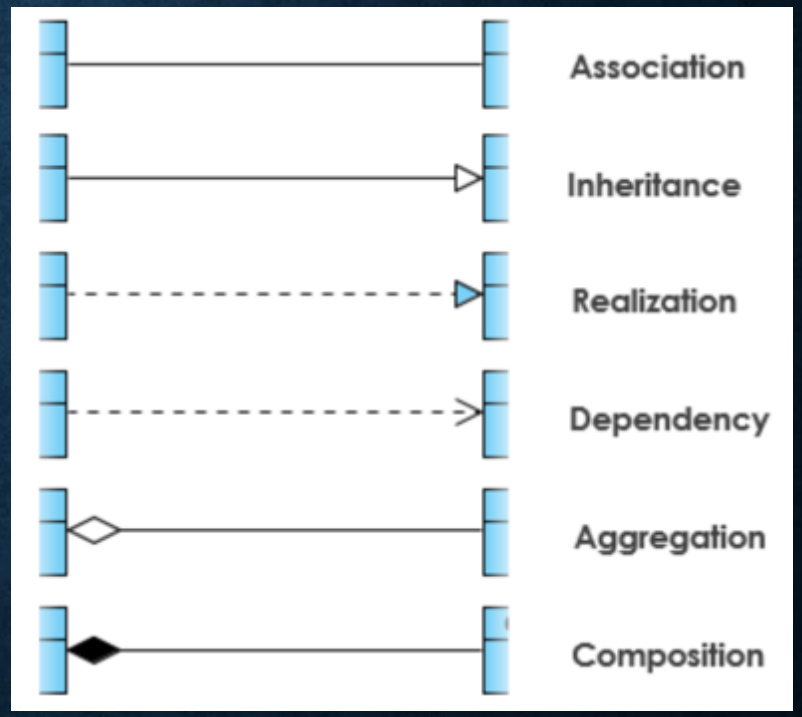
- + denotes public attributes or operations
- - denotes private attributes or operations
- # denotes protected attributes or operations



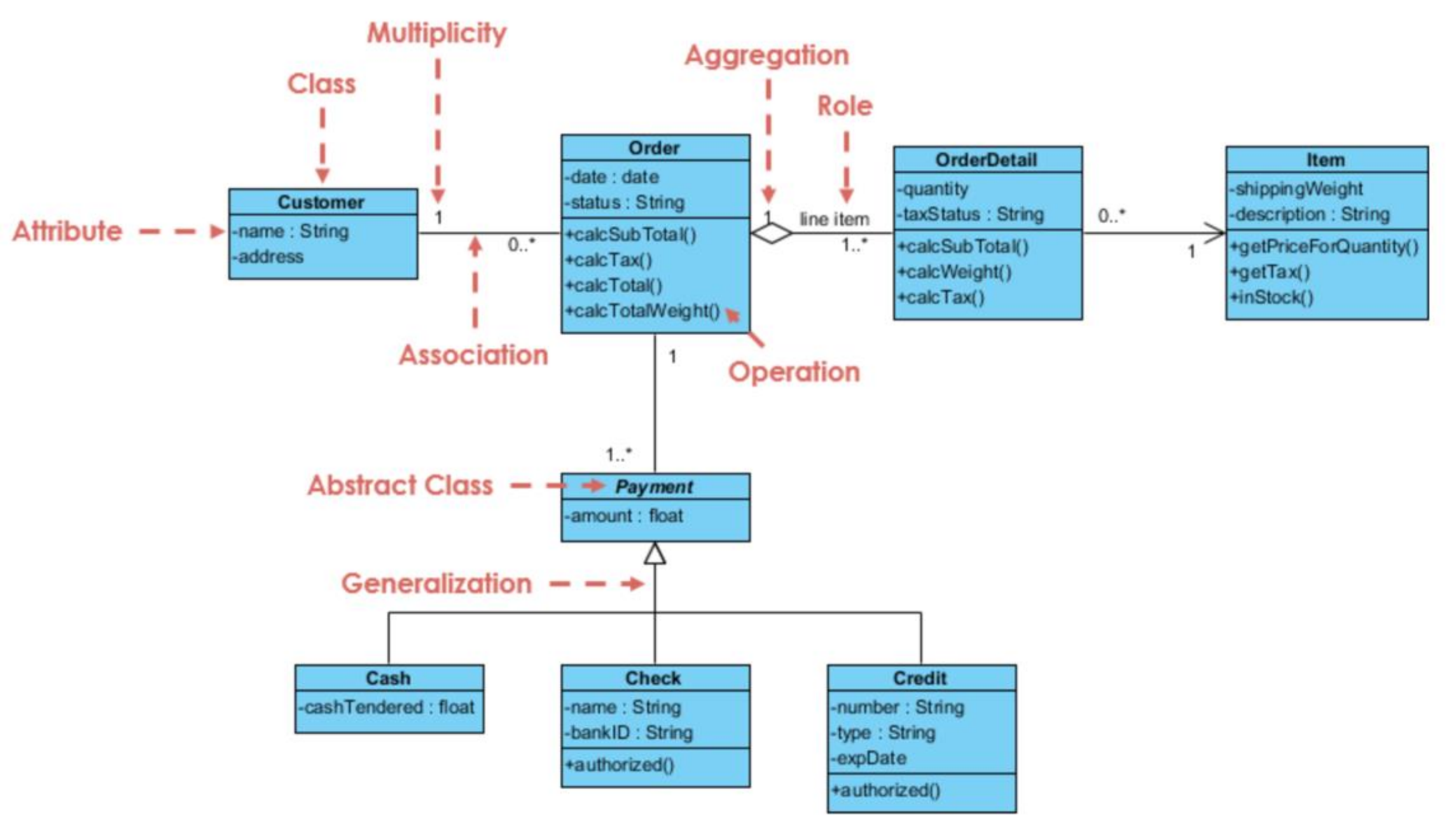
UML

Relationships between classes

UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. Can you describe what each of the relationships mean relative to your target programming language shown in the Figure below?



If you can't yet recognize them, no problem this section is meant to help you to understand UML class relationships. A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:


Multiplicity
Class
Aggregation
Role
Attribute
Association
Operation
Abstract Class
Generalization
Customer
-name : String
-address
Order
-date : date
-status : String
+calcSubTotal()
+calcTax()
+calcTotal()
+calcTotalWeight()
OrderDetail
-quantity
-taxStatus : String
+calcSubTotal()
+calcWeight()
+calcTax()
Item
-shippingWeight
-description : String
+getPriceForQuantity()
+getTax()
+inStock()
Payment
-amount : float
Cash
-cashTendered : float
Check
-name : String
-bankID : String
+authorized()
Credit
-number : String
-type : String
-expDate
+authorized()
1
0..*
1
line item
1..*
0..*
1
1
1..*

אז איך יוצרים UML?

- ניתן להתקין פלאגאין לIntelliJ שנקרא PLANTUML
- בעזרת התוסף ניתן ליצור דיאגרמות שונות
- מומלץ להשתמש ב-class diagram על מנת ליצור UML- לא מחייב
- להעשרה ופרטים נוספים על אופן הכתיבה עצמו לחצו כאן : https://plantuml.com/class-diagram

Exceptions

When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an exception (throw an error).

באילו חריגות כבר נתקלנו?

- NullPointerException- ניתקל כאשר נפנה לשדה או שיטה של אובייקט שהוא null.
למשל במעבר איטרטיבי על רשימה מקושרת אם לא נגדיר כמו שצריך מתי לעצור את לולאת הwhile זה יקרה.
- ArrayIndexOutOfBoundsException- ניתקל כאשר נפנה לתא שלא קיים במערך.
- ArithmeticException- ניתקל אם נבצע חלוקה ב-0.

אם נזרקה חריגה בתוכנית שלי, האם זה דבר רע?

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

try-catch block

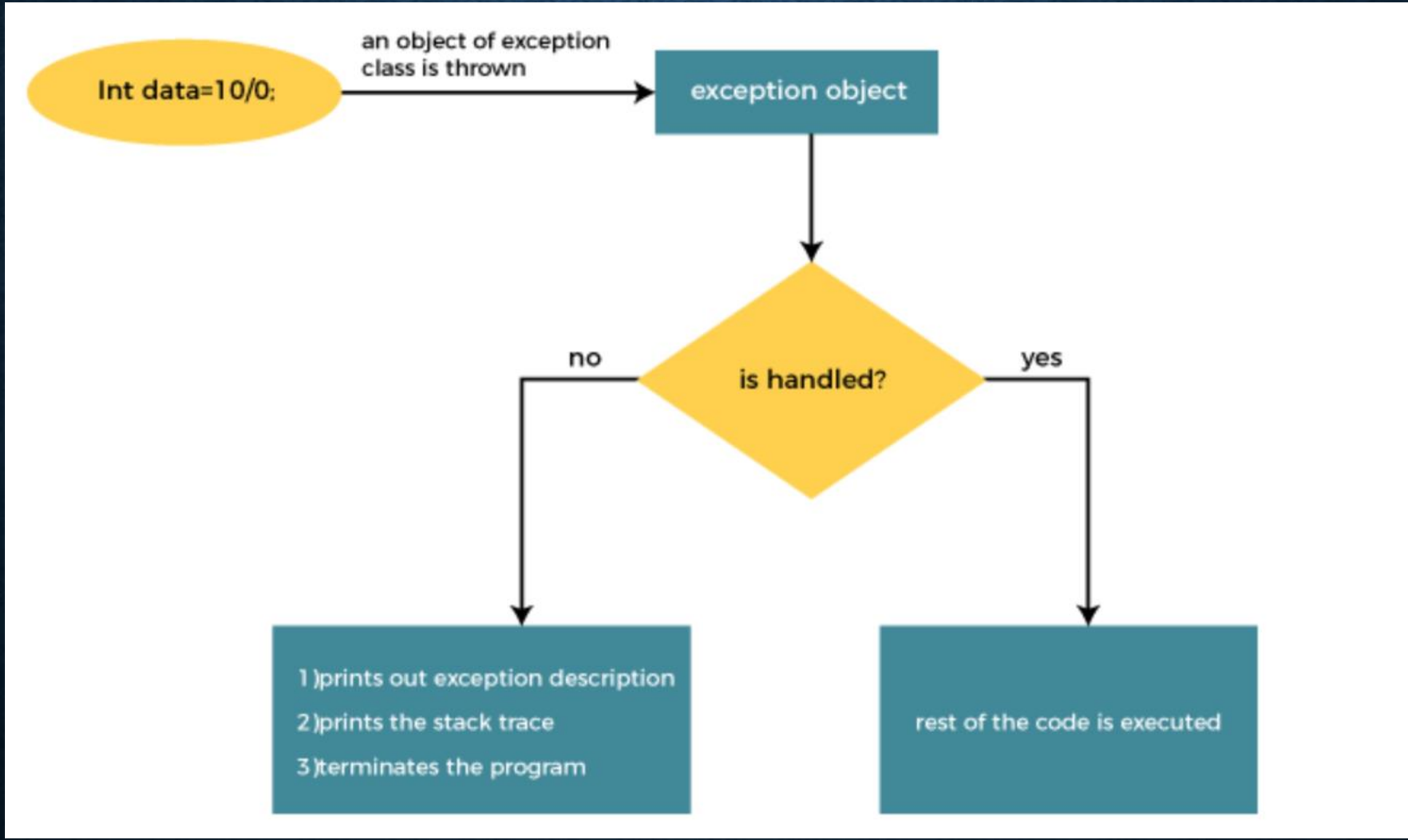
try block

- Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.
- If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.
- Java try block must be followed by either catch or finally block.

catch block

- Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

Internal Working of Java try-catch block



Internal Working of Java try-catch block

Problem without exception handling

```java
public class TryCatchExample1 {

  public static void main(String[] args) {

    int data=50/0; //may throw exception

    System.out.println("rest of the code");

  }

}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Solution by exception handling

```java
public class TryCatchExample2 {

  public static void main(String[] args) {
    try
    {
    int data=50/0; //may throw exception
    }
      //handling the exception
    catch(ArithmeticException e)
    {
      System.out.println(e);
    }
    System.out.println("rest of the code");
  }
```

```
java.lang.ArithmeticException: / by zero
rest of the code
```

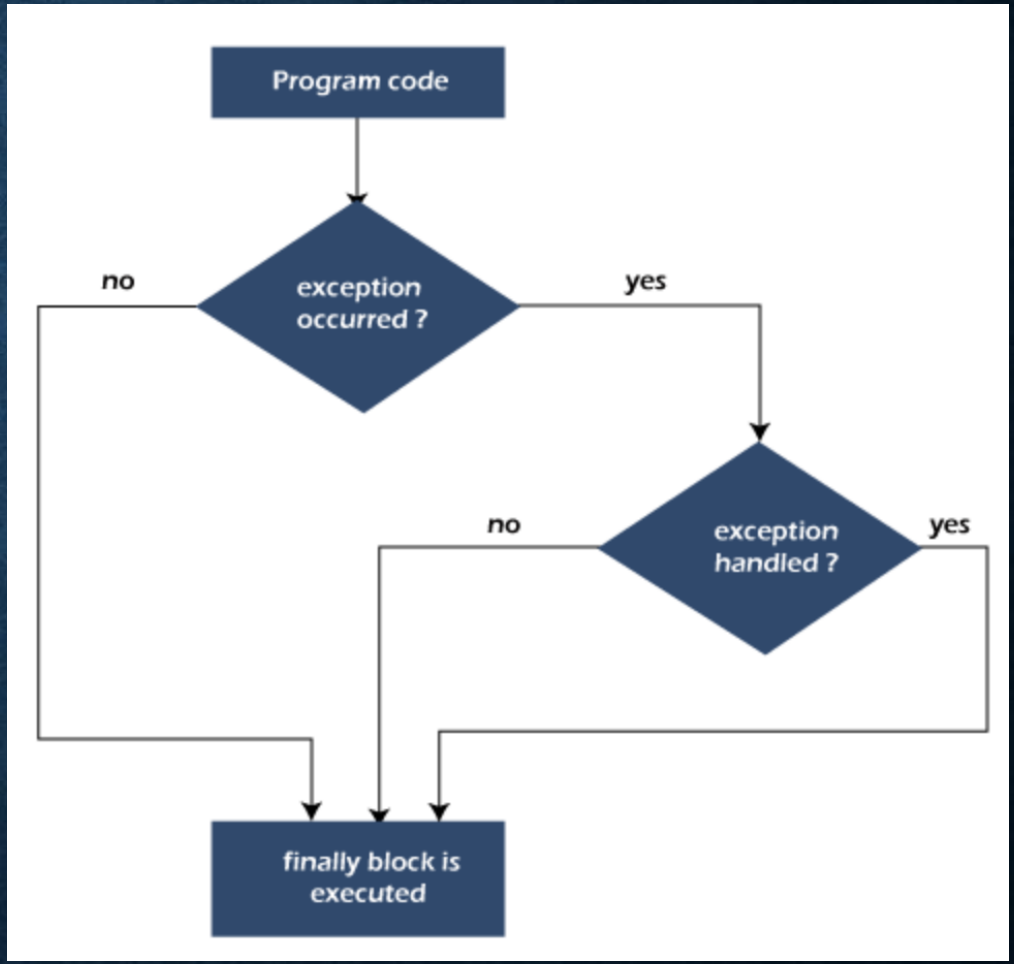
What will be the output?

```
public static double Divide(int a, int b){
    return (double)(a/b);
}

public static void main(String[] args) {
    int a=5, b=0;
    try {
        System.out.println(a+"/"+b+"="+Divide(a,b));
    }
    catch(ArithmeticException e){
        System.out.println("Arithmetic Exception caught!");
        System.out.println("You can't divide by 0!");
    }
}
```

finally block

finally block is a block used to execute important code such as closing the connection, etc.
Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.
The finally block follows the try-catch block.



Why use Java finally block?
- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

```java
class TestFinallyBlock {
  public static void main(String args[]){
  try{
//below code do not throw any exception
   int data=25/5;
   System.out.println(data);
  }
//catch won't be executed
  catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");
}

System.out.println("rest of phe code...");
  }
}
```

```
finally block is always executed
rest of the code...
```

```java
public class TestFinallyBlock2{
    public static void main(String args[]){

    try {

      System.out.println("Inside try block");

      //below code throws divide by zero exception
     int data=25/0;
     System.out.println(data);
    }

    //handles the Arithmetic Exception / Divide by zero exception
    catch(ArithmeticException e){
      System.out.println("Exception handled");
      System.out.println(e);
    }

    //executes regardless of exception occured or not
    finally {
      System.out.println("finally block is always executed");
    }

    System.out.println("rest of the code...");
    }
  }
```

```
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

Java throw keyword

The Java throw keyword is used to throw an exception explicitly.
We specify the **exception** object which is to be thrown.
The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.
We can throw either checked or unchecked exceptions in Java by throw keyword.
It is mainly used to throw a custom exception.
We will discuss custom exceptions later in this section.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

Java throw keyword

```java
public class TestThrow1 {
  //function to check if person is eligible to vote or not
  public static void validate(int age) {
    if(age<18) {
      //throw Arithmetic exception if not eligible to vote
      throw new ArithmeticException("Person is not eligible to vote");
    }
    else {
      System.out.println("Person is eligible to vote!!");
    }
  }
  //main method
  public static void main(String args[]){
    //calling the function
    validate(13);
    System.out.println("rest of the code...");
 }
```

```
at TestThrow1.validate(TestThrow1.java:8)
at TestThrow1.main(TestThrow1.java:18)
```

Java throw keyword

```java
class UserDefinedException extends Exception
{
   public UserDefinedException(String str)
   {
      // Calling constructor of parent Exception
      super(str);
   }
}
// Class that uses above MyException
public class TestThrow3
{
   public static void main(String args[])
   {
      try
      {
         // throw an object of user defined exception
         throw new UserDefinedException("This is user-defined exception");
      }
      catch (UserDefinedException ude)
      {
         System.out.println("Caught the exception");
         // Print the message from MyException object
         System.out.println(ude.getMessage());
      }
   }
```

```
Caught the exception
This is user-defined exception
```

Java throws keyword

The **Java throws keyword** is used to declare an exception.
It gives an information to the programmer that there may occur an exception.
So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.
Exception Handling is mainly used to handle the checked exceptions.
If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

```
Exception in thread "main" java.io.IOException: device error
    at M.method(Testthrows4.java:4)
    at Testthrows4.main(Testthrows4.java:10)
```

throw VS throws

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Java Files

The File class from the java.io package, allows us to work with files.

To use the File class, create an object of the class, and specify the filename or directory name:

```
import java.io.File;  // Import the File class

File myObj = new File("filename.txt"); // Specify the filename
```

Method	Type	Description
canRead()	Boolean	Tests whether the file is readable or not
canWrite()	Boolean	Tests whether the file is writable or not
createNewFile()	Boolean	Creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	Tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

Create a File

To create a file in Java, you can use the createNewFile() method. This method returns a Boolean value: true if the file was successfully created, and false if the file already exists.
Note that the method is enclosed in a try...catch block.
This is necessary because it throws an IOException if an error occurs (if the file cannot be created for some reason)

```java
import java.io.File;  // Import the File class
import java.io.IOException;  // Import the IOException class to handle errors

public class CreateFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
```

To create a file in a specific directory (requires permission), specify the path of the file and use double backslashes to escape the "\" character (for Windows). On Mac and Linux you can just write the path, like: /Users/name/filename.txt

```java
File myObj = new File("C:\\Users\\MyName\\filename.txt");
```

Write to a File

In the following example, we use the FileWriter class together with its write() method to write some text to the file we created in the example above.
Note that when you are done writing to the file, you should close it with the close() method:

Example

```
import java.io.FileWriter;   // Import the FileWriter class
import java.io.IOException;  // Import the IOException class to handle errors

public class WriteToFile {
  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("filename.txt");
      myWriter.write("Files in Java might be tricky, but it is fun enough!");
      myWriter.close();
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
```

JSON

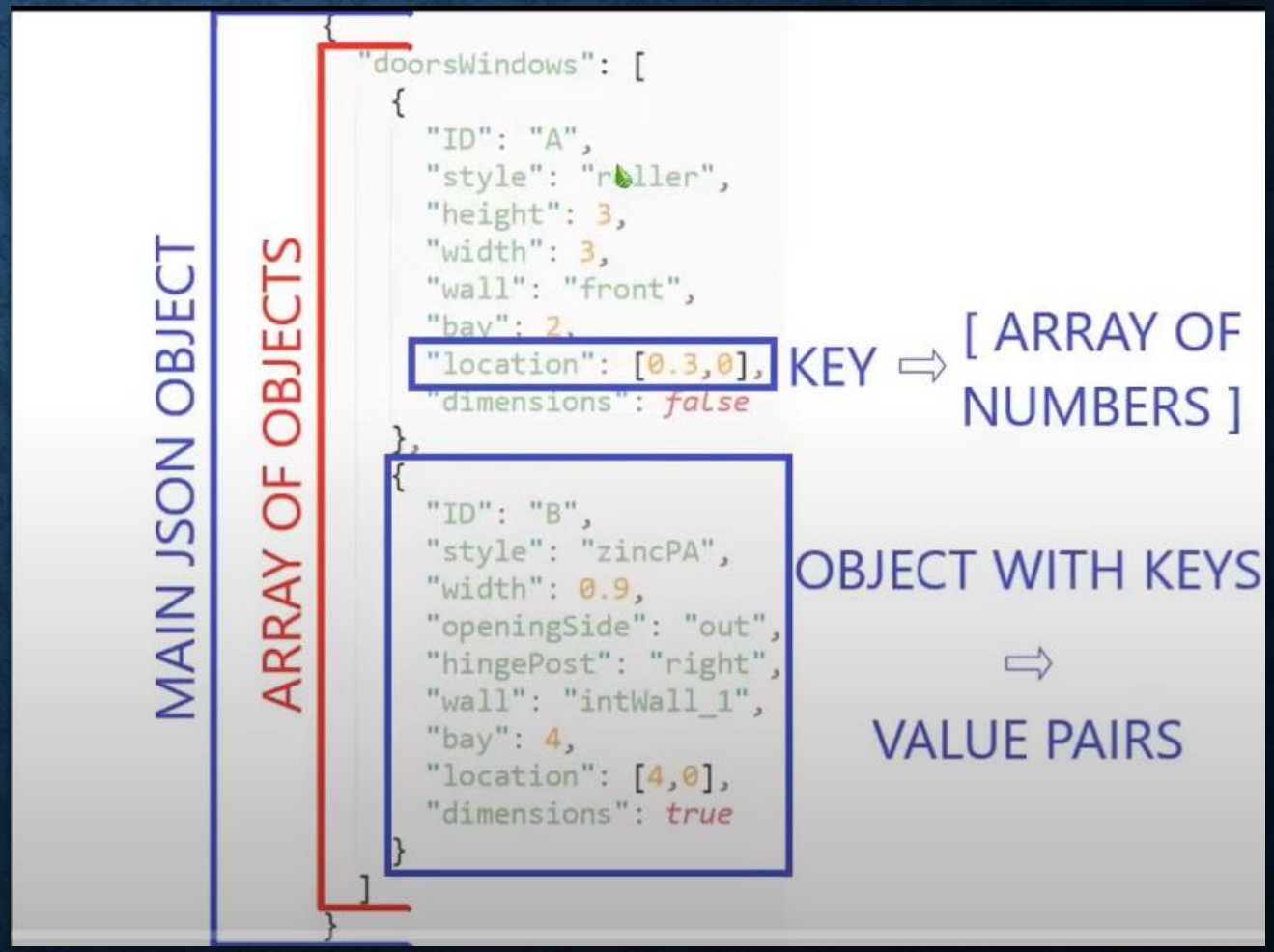
The following JSON example define an employee's object, with an array of 3 employees:

JSON Example

```
{"employees":[
  { "firstName":"John", "lastName":"Doe" },
  { "firstName":"Anna", "lastName":"Smith" },
  { "firstName":"Peter", "lastName":"Jones" }
]}
```

JSON Syntax rules

- Data is name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays



אז למה שנרצה להשתמש בJSON?

- קריאה וכתיבה יחסית בקלות.
- מבנה נוח לשימוש לאחר הקריאה- מבוסס מפתח וערך.
- ניתן לשימוש בשפות תכנות רבות.
- ניתן לקריאה בקלות על ידי בני אדם.
- משמש להעברת הודעות ומסרים

החסרונות של JSON

- כבד מבחינת משקל.

JSON הוא ייצוג טקסטואלי של הפרטים השמורים בו וכתוצאה מכך משקלו נקבע לפי משקל סך כל התווים הנמצאים בו כך שאפשר יחסית בקלות להגיע לJSON ששוקל כמה עשרות MB ואפילו יותר.

מה ניתן לשמור בקובץ JSON?

- מספרים(שלמים, ולא שלמים)
- מחרוזות
- ביטויים בוליאניים
- מערכים/רשימות
- Null
- אובייקטים פשוטים ומורכבים(כאלה שמורכבים מאובייקטים אחרים)

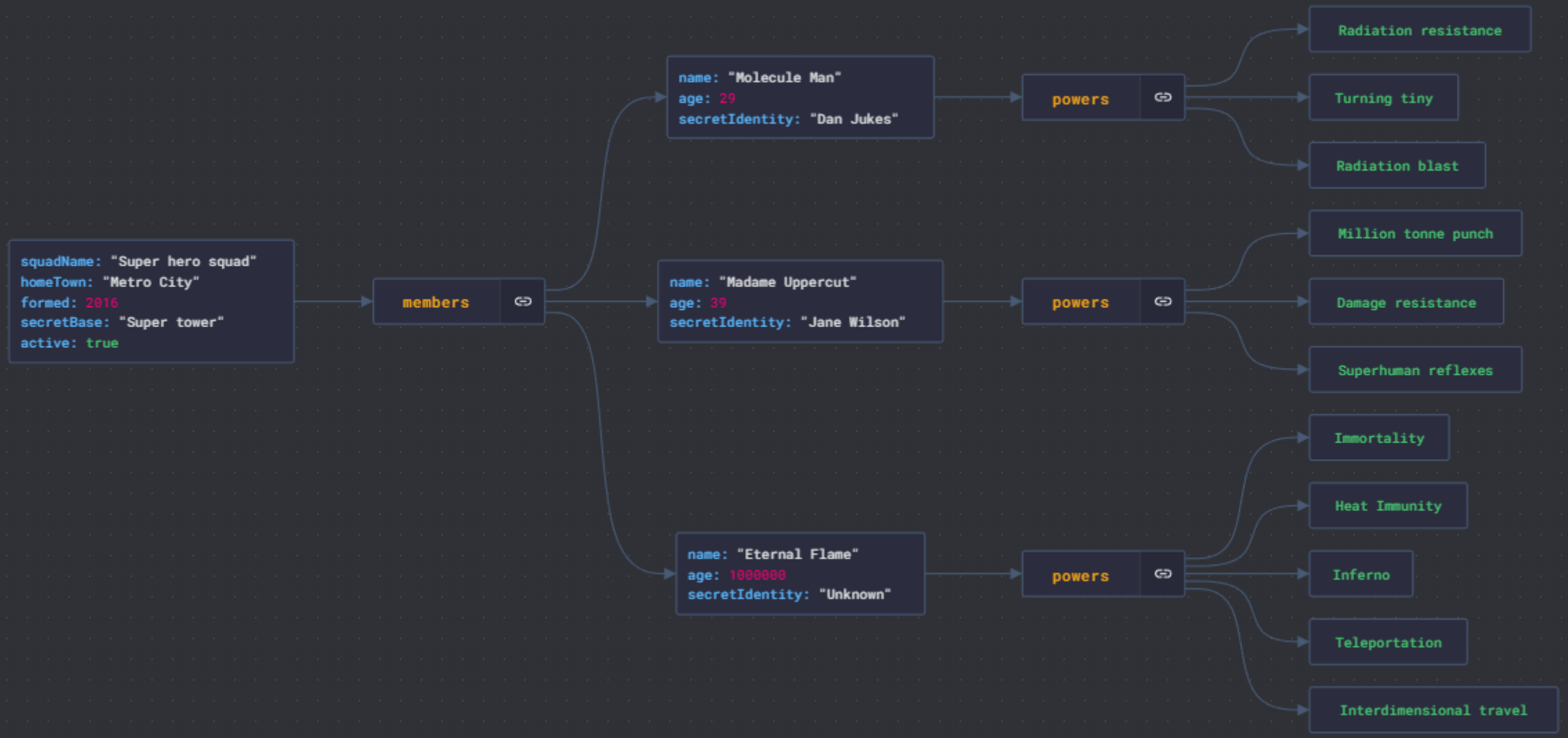
```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": [
        "Radiation resistance",
        "Turning tiny",
        "Radiation blast"
      ]
    },
    {
      "name": "Madame Uppercut",
      "age": 39,
      "secretIdentity": "Jane Wilson",
      "powers": [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ]
    },
    {
      "name": "Eternal Flame",
      "age": 1000000,
      "secretIdentity": "Unknown",
      "powers": [
        "Immortality",
        "Heat Immunity",
        "Inferno",
        "Teleportation",
        "Interdimensional travel"
      ]
    }
  ]
}
```

העשרה וידע כללי

אחד היתרונות של JSON הוא העובדה שהוא קריא ומובן למשתמשים, אך האם יש דרך לפשט אותו ויזואלית?
קיים אתר בשם jsoncrack שמאפשר ייצוג ויזואלי של תוכן הJSON שיהפוך אותו לעוד יותר ברור

בשקף הבא נתבונן בייצוג הוויזואלי של הJSON המצורף ונראה כמה שזה פשוט להבנה

https://jsoncrack.com/editor


squadName: "Super hero squad"
homeTown: "Metro City"
formed: 2016
secretBase: "Super tower"
active: true
members
name: "Molecule Man"
age: 29
secretIdentity: "Dan Jukes"
powers
Radiation resistance
Turning tiny
Radiation blast
name: "Madame Uppercut"
age: 39
secretIdentity: "Jane Wilson"
powers
Million tonne punch
Damage resistance
Superhuman reflexes
name: "Eternal Flame"
age: 1000000
secretIdentity: "Unknown"
powers
Immortality
Heat Immunity
Inferno
Teleportation
Interdimensional travel

מסקנה שכדאי לזכור

- נשים לב כי הייצוג הוויזואלי של קובץ JSON הוא בעצם מבנה נתונים שאנחנו כבר יכולים לזהות. מדובר בעץ שלכל קודקוד אין הגבלה על כמות הבנים.
- כמו בעצים אחרים שנתקלתם בהם במבני נתונים(לא עץ B+), הנתונים שמורים לכל אורך העץ ולא רק בעלים.
- מכאן שכמו בבעיות של עצים אנחנו נשתמש ברקורסיה על מנת לפתור את הבעיות, בשאלות רבות הקשורות לJSON נרצה לשקול להשתמש ברקורסיה כדי לפתור את הבעיה

Junit

JUNIT syntax

- המתודות שנכתוב במחלקת טסט יהיו בעצם טסטים. כל מתודה שמסומנת מעליה האנוטציה @Test היא מתודה של טסט.
- ישנם מלא אנוטציות שאפשר להוסיף למתודות במחלקה, ויש לזכור שהמחלקה הזאת היא מחלקה רגילה לכל דבר, גם לה אפשר ליצור תכונות ואפילו לאתחל ולעדכן אותן בין טסט לטסט.
- אנוטציות נוספות שאפשר להוסיף הן @BeforeEach @AfterEach @BeforeAll @AfterAll @Disabled @Timeout
- https://junit.org/junit5/docs/current/user-guide/

JUNIT syntax

- @Test – מייצג שהמתודה היא טסט
- @AfterEach – המתודה תרוץ אחרי כל אחד מהטסטים
- @BeforeEach – המתודה תרוץ לפני כל אחד מהטסטים
- @BeforeAll – מתודה שתרוץ לפני כל הטסטים
- @AfterAll - מתודה שתרוץ אחרי כל הטסטים
- @Timeout – המתודה שמה זמן מקסימלי שבו הטסט יכול לרוץ ואם הוא חורג מהזמן הזה הטסט נכשל.
- @Order(x)- המתודה תהיה המתודה ה-x לרוץ לפי הסדר(מאפשר לקבוע את סדר הטסטים)

Assert class

The org.junit.Assert class provides methods to assert the program logic.

Methods of Assert class:

The common methods of Assert class are as follows:

- **void assertEquals(boolean expected,boolean actual)**: checks that two primitives/objects are equal. It is overloaded.
- **void assertTrue(boolean condition)**: checks that a condition is true.
- **void assertFalse(boolean condition)**: checks that a condition is false.
- **void assertNull(Object obj)**: checks that object is null.
- **void assertNotNull(Object obj)**: checks that object is not null.

```java
package com.javatpoint.logic;
public class Calculation {

  public static int findMax(int arr[]){
    int max=0;
    for(int i=1;i<arr.length;i++){
      if(max<arr[i])
        max=arr[i];
    }
    return max;
  }
}
```

```java
package com.javatpoint.testcase;

import static org.junit.Assert.*;
import com.javatpoint.logic.*;
import org.junit.Test;

public class TestLogic {

  @Test
  public void testFindMax(){
    assertEquals(4,Calculation.findMax(new int[]{1,3,4,2}));
    assertEquals(-1,Calculation.findMax(new int[]{-12,-1,-3,-4,-2}));
  }
}
```

Uncommon and useful assertions

- void assertDoesNotThrow(Executable executable; ->יכשל אם תזרק חריגה כלשהי
- <T extends Throwable> assertThrows(Class <T> expectedType, Executable executable)->
  יכשל אם לא נזרקה חריגה או נזרקה חריגה שונה מזו שהוגדרה
- Void assertTimeout(Duration duration, Executable executable)->יכשל אם תהיה חריגה מהטיימאאוט

```
@Test
void exceptionTesting() {
    Exception exception = assertThrows(ArithmeticException.class, () ->
        calculator.divide(1, 0));
    assertEquals("/ by zero", exception.getMessage());
}

@Test
void timeoutNotExceeded() {
    // The following assertion succeeds.
    assertTimeout(ofMinutes(2), () -> {
        // Perform task that takes less than 2 minutes.
    });
}
```

Rules for test writing

- כל טסט יבדוק פונקציה אחת בלבד
- כל טסט יבדוק חלק אחד בפונקציונאליות של פונקציה- ניתן לפרק לתתי מקרים במידת הצורך
- לחשוב מה עלול לגרום לתוכנה להכשל ולא מה יראה שהיא עובדת- אם זה לא בהכרח נכון, זה לא נכון.
- שימוש בבדיקות הקשורות לחריגות במידת הצורך- יש מקומות שנרצה להראות שהפונקציה שלנו זורקת חריגה עבור ביצוע פקודה לא חוקית למשל.
- בפונקציות שעשויות לזרוק חריגה, נרצה תמיד לבדוק שלא נזרקות חריגות.

•איכות > כמות

תזכרו שמי שבודק אתכם יכול לבדוק אתכם על קלט שלא חשבתם עליו ולהפיל לכם את המערכת, נסו לכתוב טסטים איכותיים ככל האפשר ולהמנע מטסטים שלא באמת עוזרים לכם להראות כמה המערכת שבניתם טובה.

איך לשפר את הטסטים?

- ניתן לבנות פונקציות שיסייעו לטסט ויהפכו אותו ליותר איכותי ויותר מדויק.
- נניח ויש לנו פרויקט הסובב סביב עצי חיפוש בינאריים, מבנה נתונים שאנחנו מכירים טוב.
- המצאנו מעבר מיוחד על העץ וראינו שעל 2-3 קלטים יחסית פשוטים אנחנו מקבלים פלט נכון אבל , אך האם זה אומר שלכל קלט נקבל פלט נכון?
- נוכל ליצור גנרטור שיוצר עצי חיפוש בינארים עם n קודקודים המכיל ערכים בתחום כלשהו באופן אקראי.
- ואז זה כבר חוזר לאינפי, עבור n גדול מספיק ומספר k גדול מספיק של עצי חיפוש בינארי כך שבכל הקלטים אנחנו מקבלים שהפלט תקין, כנראה שהפונקציה שלנו נכונה
- ואם נמצא פלט שגוי, נוכל לבדוק על איזה קלט אנחנו מקבלים פלט שגוי ובעזרת הדיבאגר להבין איפה בקוד הבעיה נמצאת.

יתרונות השיטה:

- אובר הכללה- יכול לתפוס מקרי קצה שלא חשבתם עליהם
- רנדומיזציה- מאפשר לכסות הרבה מאוד מקרים שונים מבלי לפצלם להרבה מקרי קצה
- חוכמת ההמונים- אם עבור כמות מאוד גדולה של קלטים אקראיים הפונקציה שלנו תקינה, סביר להניח שהיא תקינה
- פשוטה ונוחה- פונקציה אחת שמייצרת את הקלט האקראי שניתן להשתמש בה בכל אחד מהטסטים.

```java
public BST Generate(int nodeCount, int minV, int maxV){
    BST bst=new BST();
    HashSet<Integer> keep=new HashSet<>();
    Random rnd=new Random();
    while(nodeCount>0){
        int number = rnd.nextInt( bound: maxV-minV)+minV;
        if(!keep.contains(number)){
            bst.insert(number);
            keep.add(number);
            nodeCount--;
        }
    }
    return bst;
}
```