Collections & Exceptions

Dr. Hadeer A. Hosny

Second term
2022-2021

Points to be covered:

- **File class**
- **Random class**
- **Arrays**
- **`compareTo()` method**
- **Collections**
- **Exceptions**

File class

File Class [java.io]

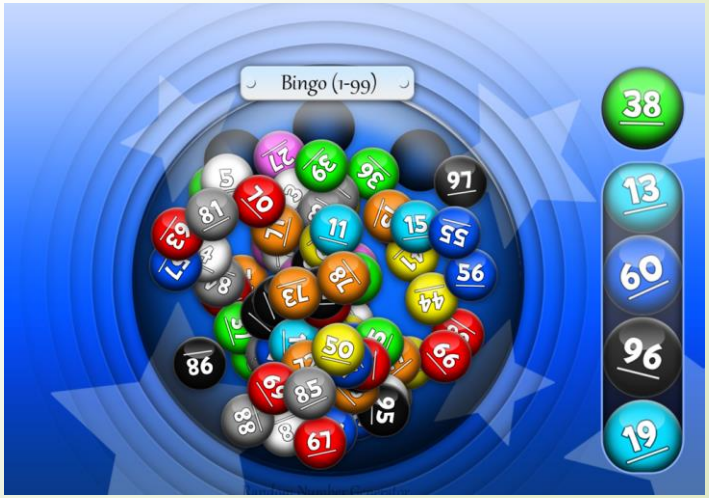
- A file name like "numbers.txt" has only String properties
- File has some very useful methods
  - exists: tests if a file already exists
  - canRead: tests if the OS will let you read a file
  - canWrite: tests if the OS will let you write to a file
  - delete: deletes the file, returns true if successful
  - length: returns the number of bytes in the file
  - getName: returns file name, excluding the preceding path
  - getPath: returns the path name—the full name

```
File numFile = new File("numbers.txt");
if (numFile.exists())
  System.out.println(numfile.length());
```

Random class

Java Random class

- Some applications, such as games require the use of randomly generated numbers.
- Java Random class is used to generate a stream of random numbers.
- The algorithms implemented by Random class use a protected utility method.



Java Random class

- **To use the Random class:**
  - **import java.util.Random;**
  - **Random randomNumbers = new Random();**

Constructor	Description
Random()	It creates a new random number generator.
Random(long seed)	This creates a new random number generator using a single long seed.

Java Random class methods

Method	Description
double nextDouble()	It returns the next pseudorandom, uniformly distributed double value between 0.0 and 1.0 from this random number generator's sequence.
float nextFloat()	It returns the next pseudorandom, uniformly distributed float value between 0.0 and 1.0 from this random number generator's sequence.
int nextInt()	It returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence.
int nextInt(int n)	It returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

Java Random class methods

long nextLong()	It returns the next pseudorandom, uniformly distributed long value from this random number generator's sequence.
void setSeed(long seed)	It sets the seed of this random number generator using a single long seed.
boolean nextBoolean()	It returns the next pseudorandom, uniformly distributed boolean value from this random number generator's sequence.

Java Random class example

```
import java.util.Random;

public class Test {
        public static void main(String args[]){
                Random random = new Random();
                random.setSeed(30);
                //It generates boolean value
                System.out.println(random.nextBoolean());
                //It generates double value
                System.out.println(random.nextDouble());
                //It generates float value
                System.out.println(random.nextFloat());
                //It generates int value
                System.out.println(random.nextInt());
                //It generates int value within specific limit
                System.out.println(random.nextInt(50));
        }
}
```

Output

```
true
0.29472606320906436
0.11627269
368153135
37
```

Arrays

Arrays

- Arrays are a little surprising (at first) in Java
- Arrays themselves are objects!
- There are two basic types of arrays
  - Arrays of primitives
  - Arrays of object references
- There are also multidimensional arrays which are essentially arrays of array [of arrays...]
- Remember since an array is an object it will have a reference

Arrays of Primitives

- There are actually several different legal syntaxes for creating arrays. We'll just demonstrate one. Let's make an array of ints:

```
int[] ia; // This is the reference
ia = new int[10]; // This makes the object
```

- The array object referenced by ia can now hold 10 ints. They are numbered from 0 to 9.
- To learn the size of the array we can do this:

```
ia.length
```

- Note: This is not a method call but is like an instance variable. It cannot be changed!!!

```
for(int i=0; i < ia.length; i++)
    ia[i] = i * 10;
```

Arrays of Objects

- Actually, Arrays of Object References.
- Make array of Box objects

```
Box[] ba; // Creates reference
ba = new Box[10]; // Makes array object
```

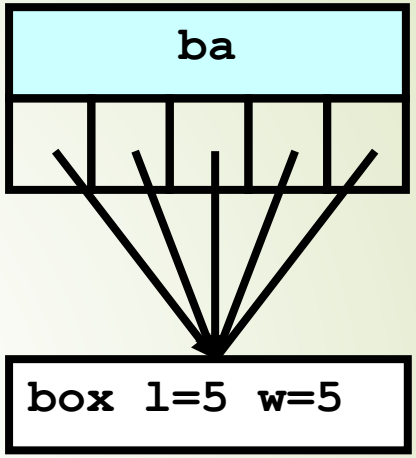
- We now have 10 references which can refer to box objects! They will all be initialized to null.
- Box class in example has two variables l and w
- Constructions like this are allowed

```
Box[] ba={new Box(2,1), new Box(4,3), new Box(1,6)};
```

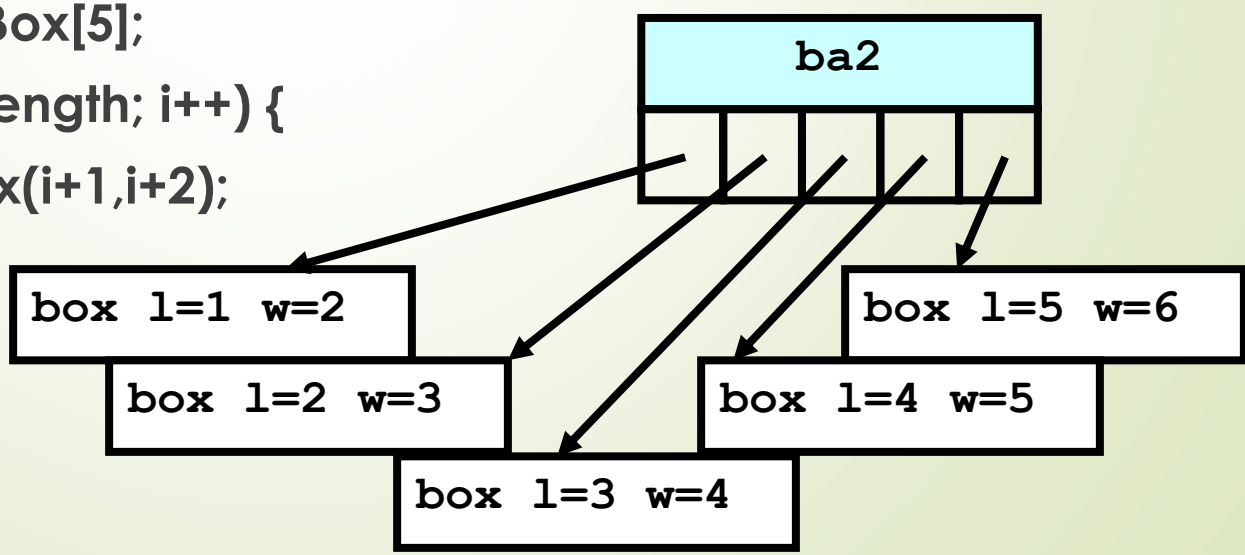
- It is very important to understand the necessity of creating new objects!

Arrays of Objects

```
Box[] ba = new Box[5];
Box b = new Box(5,5);
for(int i=0; i < ba.length; i++) {
    ba[i] = b;
}
```




```
Box[] ba2 = new Box[5];
for(int i=0; i < ba.length; i++) {
    ba[i] = new Box(i+1,i+2);
}
```



How many objects?

CompareTo() method

The `compareTo` method

- **The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.**
  - Example: in the `String` class, there is a method:

```
public int compareTo(String other)
```

- A call of `A.compareTo(B)` will return:

a value <	0	if A comes "before" B in the ordering,
a value >	0	if A comes "after" B in the ordering,
or	0	if A and B are considered "equal" in the ordering.

Using `compareTo`

- `compareTo` can be used as a test in an `if` statement.

```
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) {  // true
    ...
}
```

Primitives	Objects
`if (a < b) { ...`	`if (a.compareTo(b) < 0) { ...`
`if (a <= b) { ...`	`if (a.compareTo(b) <= 0) { ...`
`if (a == b) { ...`	`if (a.compareTo(b) == 0) { ...`
`if (a != b) { ...`	`if (a.compareTo(b) != 0) { ...`
`if (a >= b) { ...`	`if (a.compareTo(b) >= 0) { ...`
`if (a > b) { ...`	`if (a.compareTo(b) > 0) { ...`

compareTo

- **You can use an array or list of strings with Java's included binary search method because it calls `compareTo` internally.**

```
String[] a = {"al", "bob", "cari", "dan", "mike"};
int index = Arrays.binarySearch(a, "dan");  // 3
```

```
package test;

import java.util.Arrays;

class Test {

    public static void main(String[] args) {
        String[] a = {"al", "bob", "cari", "dan", "mike"};
        int index = Arrays.binarySearch(a, "dan");  // 3
        System.out.println(index);
    }

} // Test
```

```
run:
3
BUILD SUCCESSFUL (total time: 0 seconds)
```

Comparable

```
public interface Comparable<E> {
    public int compareTo(E other);
}
```

- A class can implement the `Comparable` interface to define a natural ordering function for its objects.

- A call to your `compareTo` method should return:

a value <	0	if `this` object comes "before" the `other` object,
a value >	0	if `this` object comes "after" the `other` object,
or	0	if `this` object is considered "equal" to the `other`.

Comparable template

```
public class name implements Comparable<name> {

    ...

    public int compareTo(name other) {
        ...
    }
}
```

Comparable example

```
public class Point implements Comparable<Point> {
    private int x;
    private int y;
    …

    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;    // same x, smaller y
        } else if (y > other.y) {
            return 1;     // same x, larger y
        } else {
            return 0;     // same x and same y
        }
    }
}
```

Collections

Collections

- **collection: an object that stores data; "data structure"**
  - **the objects stored are called elements**
  - **some collections maintain an ordering; some allow duplicates**
  - **typical operations: add, remove, clear, contains (search), size**

  - **examples found in the Java class libraries:**
    - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`

  - **all collections are in the `java.util` package**

```
import java.util.*;
```

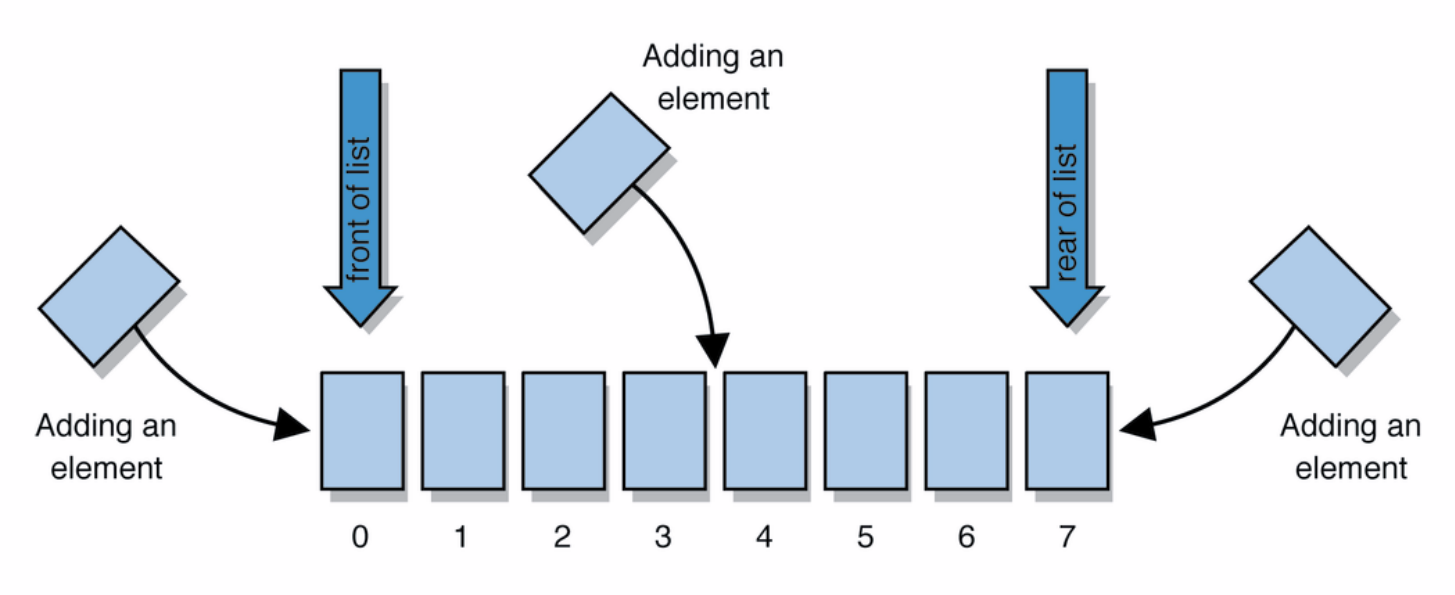
Collections

- The interface **java.util.Collection**, for instance, contains methods for manipulating a collection.
- **Some of the methods in this interface are:**
  1. **boolean add(Object object):** adds the supplied object to the collection.
  2. **boolean addAll(Collection collection):** adds all objects in the supplied collection to this collection.
  3. **void clear():**removes all of the elements from this collection.
  4. **boolean contains(Object object):** returns true if and only if this collection contains the supplied object.
  5. **int size():** returns the number of elements in this collection.
  6. **Methods for removing objects, checking if the collection is empty**, etc.

- The **List interface extends Collection**.
- A list is a collection of objects where the objects are put in a sequence. Thus, it has all the methods that pertain to a collection and the ones that are specific to lists such as **void add(int index, Object object)** which inserts the given object at the position specified by the index in this list.

Lists

- **list: a collection storing an ordered sequence of elements**
  - **each element is accessible by a 0-based index**
  - **a list has a size (number of elements that have been added)**
  - **elements can be added to the front, back, or elsewhere**
  - **in Java, a list can be represented as an `ArrayList` object or LinkedList object.**



Idea of a list

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)

```
[]
```

- You can add items to the list.
  - The default behavior is to add to the end of the list.

```
[hello, ABC, goodbye, okay]
```

- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
  - Think of an "array list" as an automatically resizing array object.
  - Internally, the list is implemented using an array and a size field.

`ArrayList` methods

`add(`**value**`)`	appends value at end of list
`add(`**index, value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index, value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

ArrayList

`addAll(`**list**`)` `addAll(`**index, list**`)`	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
`contains(`**value**`)`	returns true if given value is found somewhere in this list
`containsAll(`**list**`)`	returns true if this list contains every element from given list
`equals(`**list**`)`	returns true if given other list contains the same elements
`iterator()` `listIterator()`	returns an object used to examine the contents of the list
`lastIndexOf(`**value**`)`	returns last index value is found in list (-1 if not found)
`remove(`**value**`)`	finds and removes the given value from this list
`removeAll(`**list**`)`	removes any elements found in the given list from this list
`retainAll(`**list**`)`	removes any elements *not* found in given list from this list
`subList(`**from, to**`)`	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
`toArray()`	returns the elements in this list as an array

Array List

- **Using the above classes, it is easy to create and use lists. The following simple class creates a sequence of String objects, stores them in a list, and prints the list.**

```
import java.util.*;
public class ListUseExample {
  public static void main(String[] s) {
    List list = new ArrayList();
    for (int count = 1; count <= 10; count++) {
      list.add(new String("String " + count));
    }
    for (int count = 0; count <= 9; count++) {
      System.out.println(list.get(count));
    }
  }
}
```

Exception

Exception

Exceptions

- Rarely does a program runs successfully at its very first attempt.
- It is common to make mistakes while developing as well as typing a program.
- Such mistakes are categorized as:
  - Syntax errors - compilation errors.
  - Semantic errors– leads to programs producing unexpected outputs.
  - Runtime errors – most often lead to abnormal termination of programs or even cause the system to crash.
- Exceptions
  - Are for handling errors
  - Example:
    - ArrayIndexOutOfBoundsException
    - NullPointerExeption

Common Java Exceptions

- ArithmeticException
- ArrayIndexOutOfBoundException
- ArrayStoreException
- FileNotFoundException
- IOException – general I/O failure
- NullPointerException – referencing a null object
- NumberFormatException- error in conversion from type to another type

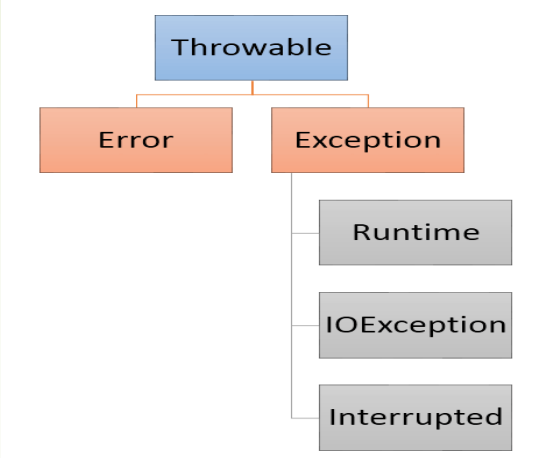
Error-Handling

- **Programming has two main tasks**
  - **Do the main computation or task at hand**
  - **Handle exceptional (rare) failure conditions that may arise**

Common Error handling

- Dividing a number by zero.
- Accessing an element that is out of bounds of an array.
- Trying to store incompatible data elements.
- Using negative value as array size.
- Trying to convert from string data to a specific data value (e.g.,
- converting string “abc” to integer value).
- File errors:
  - Opening a file in “read mode” that does not exist or no read permission
  - Opening a file in “write/update mode” which has “read only” permission.
- Any more ….

Exception Classes



- **Throwable**
  - Superclass for all exceptions
- **Two main types of exceptions**
  - **Exception**
    - This is something the caller/programmer should know about and handle
    - Must be declared in a *throws* clause
  - **Error**
    - defines the exception or the problems that are not expected to occur under normal circumstances by our program, example Memory error, Hardware error, JVM error, etc

Exception Subclasses

- **Exceptions are organized in a hierarchy**
  - **Subclasses are most specific**
  - **Higher level exceptions are less specific**
- **You can create your own subclasses of exceptions which are application specific**
  - **Rule of thumb: if your client code will need to distinguish a particular error and do something special, create a new exception subclass, otherwise, just use existing classes.**

Methods with Exceptions

- **Method *throws***
  - When a method does something that can result in an error, it should declare *throws* in the method declaration

```
public void fileRead(String f) throws IOException {
....
}
```

  - The calling method can transfer control to a exception handler by catching an exception - *try, catch*
  - Clean up can be done by - *finally*
- **Exception *throw***
  - *throw* can be used to signal an exception at runtime
  - Exceptions raised in try block can be caught and then they can be thrown again/propagated after performing some operations. This can be done by using the keyword "throw" as follows: throw exception-object; OR throw new Throwable_Subclass;

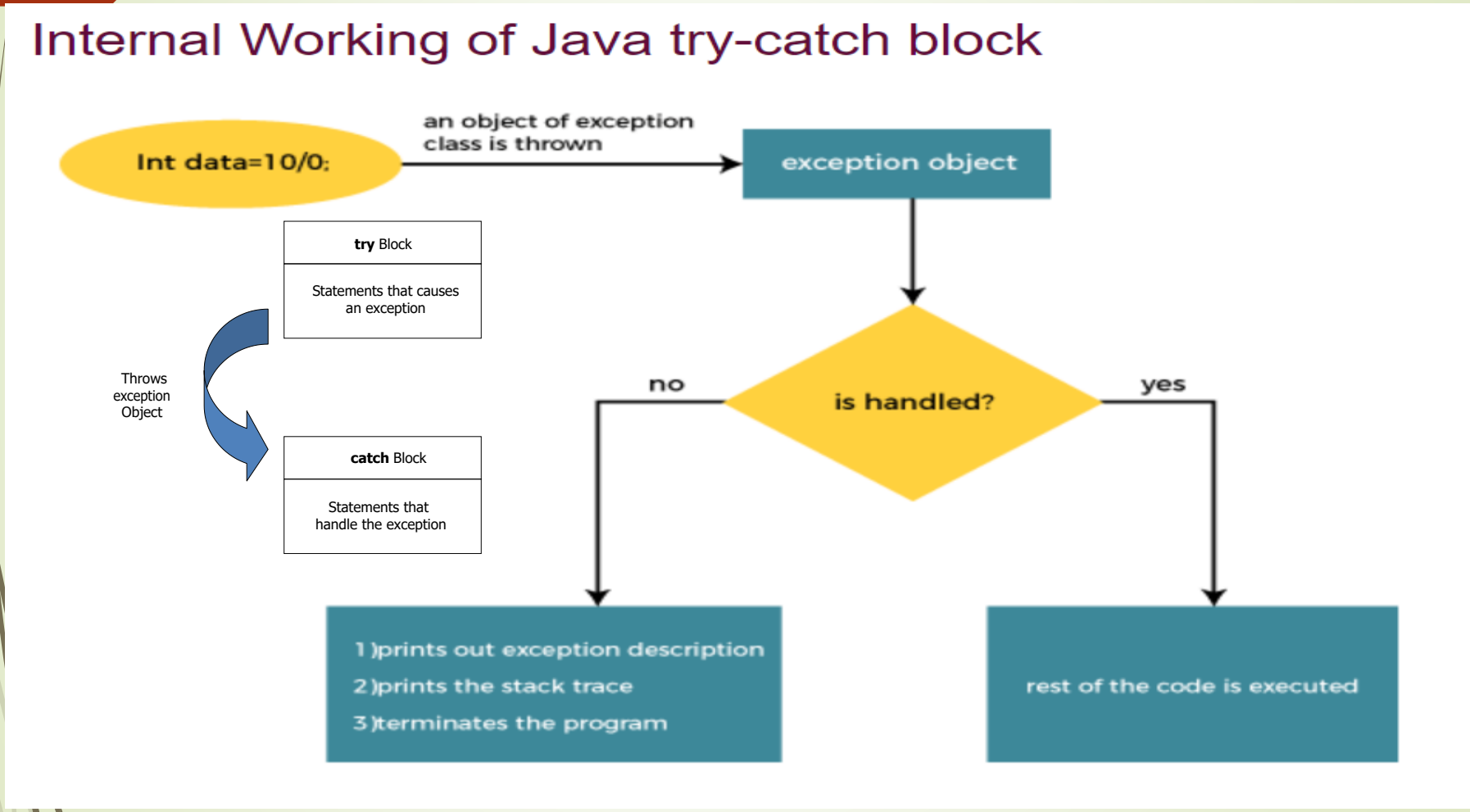
“Handling” Exceptions

- **Three possible options**
  - **Do nothing approach**
    - **Always a bad idea! Do not use this!!**
  - **Pass-the-buck-approach**
    - **Declare the exception in a *throws***
    - **This passes the exception along to the caller to handle**
  - **Do-Something-approach**
    - **Use *try-catch* block to test if an exception can happen and then so something useful**
- **Which one to use:**
  - **Depends on the application!**

try / catch

- **Idea**:
  - "**try**" to do something
  - If it fails "**catch**" the exception
  - Do something appropriate to deal with the error
- **Note**:
  - **A *try* may have multiple *catches*!**
    - Depending upon the different types of exceptions that can be thrown by all the statements inside a try block
  - **Exceptions** are tested in the same order as the catch blocks
    - Important when dealing with exceptions that have a superclass-subclass relationship

try / catch


Internal Working of Java try-catch block
Int data=10/0;
an object of exception class is thrown
exception object
try Block
Statements that causes an exception
Throws exception Object
catch Block
Statements that handle the exception
is handled?
no
yes
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

With exception handling

```
class WithExceptionHandling{
        public static void main(String[] args){
                int a,b;  float r;
                a = 7;   b = 0;
                try{
                        r = a/b;
                        System.out.println("Result is " + r);
                }

                catch(ArithmeticException e){
                        System.out.println(" B is zero);
                }
                System.out.println("Program reached this line");
                                                        }
                                }
```

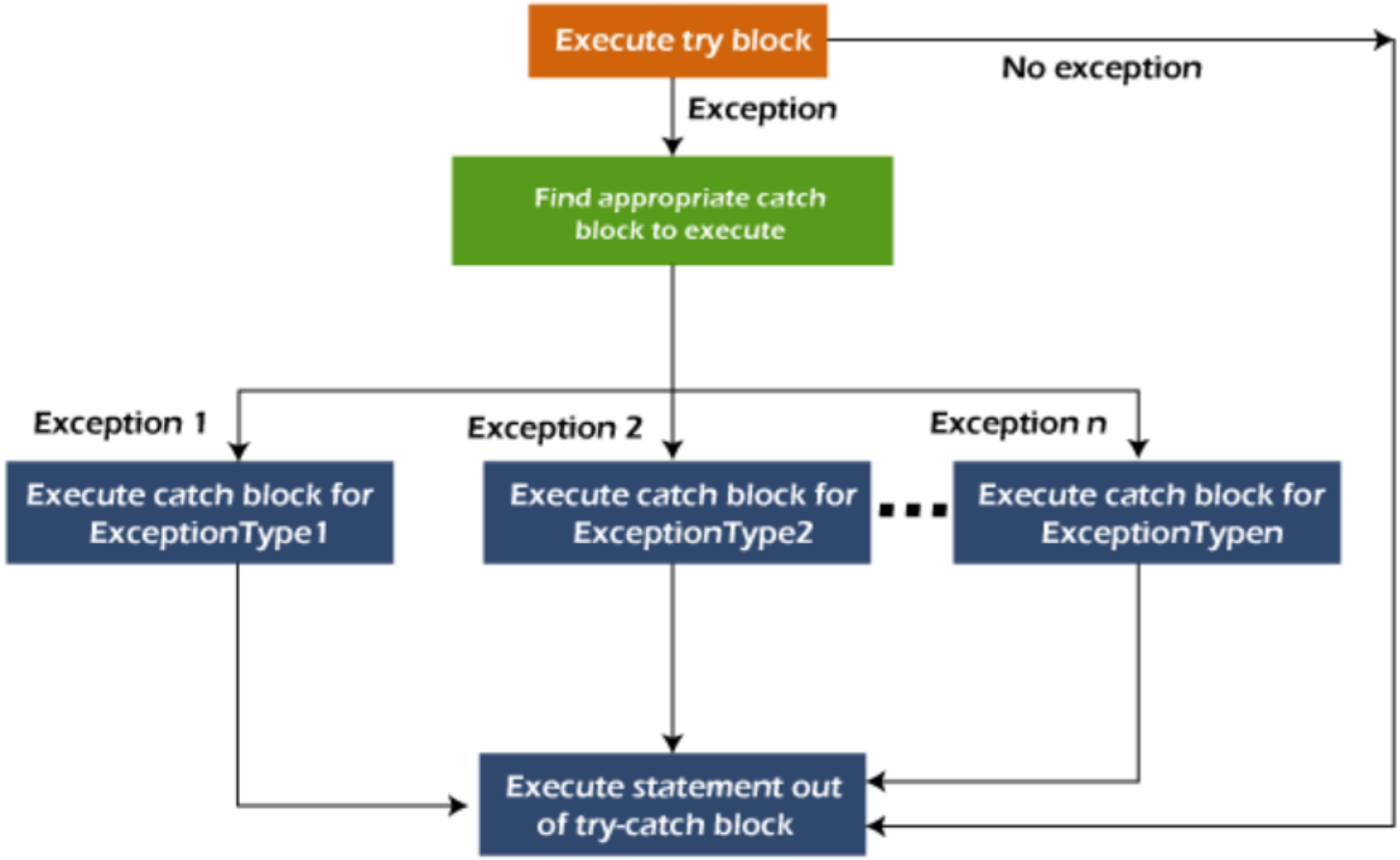
Program reach here

Exception Patterns

- **Multiple catch clauses**
  - **Possible to have multiple catch clauses for a single try statement**
    - **Essentially checking for different types of exceptions that may happen**
  - **Evaluated in the order of the code**
    - ***Bear in mind the Exception hierarchy when writing multiple catch clauses!***
    - **If you catch Exception first and then IOException, the IOException will never be caught!**

```
try {
        // statements
}
catch( Exception-Type1 e)
{
        // statements to process exception 1
}
..
..
catch( Exception-TypeN e)
{
        // statements to process exception N
}
```

Flowchart of Multi-catch Block



Exception Patterns

- **Multiple catch clauses example**

```
private void load(File file)  {
    try {
        // file opening
    }
        catch (IOException e) {
        System.err.println("IO err:" + e.getMessage());
    }
        catch (Exception e) {
        System.err.println("XML parse err:" + e.getMessage());
    }

}
```

Example


```
public class MultipleCatchBlock1 {
    public static void main(String[] args) {
        try{      int a[]=new int[5];      a[5]=30/0;      }
            catch(ArithmeticException e)
            {   System.out.println("Arithmetic Exception occurs");    }
            catch(ArrayIndexOutOfBoundsException e)
        {System.out.println("ArrayIndexOutOfBounds Exception occurs");   }
            catch(Exception e)
            {    System.out.println("Parent Exception occurs");   }
          System.out.println("rest of the code");
                                                    }
                                              }
```

```
Arithmetic Exception occurs
rest of the code
```

Example


```
public class MultipleCatchBlock3 {
    public static void main(String[] args) {
        try{    int a[]=new int[5];
        a[5]=30/0;
        System.out.println(a[10]);    }
        catch(ArithmeticException e)
        {     System.out.println("Arithmetic Exception occurs");    }
        catch(ArrayIndexOutOfBoundsException e)
        {  System.out.println("ArrayIndexOutOfBounds Exception occurs");  }
        catch(Exception e)
        {     System.out.println("Parent Exception occurs");   }
        System.out.println("rest of the code");
                                    }
                                }
```

Output:

```
Arithmetic Exception occurs
rest of the code
```

Nested try

- **Nested try**
  - **In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.**
  - **For example, the inner try block can be used to handle ArrayIndexOutOfBoundsException while the outer try block can handle the ArithemeticException (division by zero).**
- **Why use nested try block**
  - **Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.**

Nested try

Syntax:

```
....
//main try block
try
{
   statement 1;
   statement 2;
//try catch block within another try block
   try
   {
      statement 3;
      statement 4;
//try catch block within nested try block
      try
      {
         statement 5;
         statement 6;
    }
      catch(Exception e2)
      {
//exception message
      }

   }
   catch(Exception e1)
   {
//exception message
   }
}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
....
```

Example

```
public class NestedTryBlock{
 public static void main(String args[]){
 //outer try block
  try{
  //inner try block 1
   try{
    System.out.println("going to divide by 0");
    int b =39/0;
   }
   //catch block of inner try block 1
   catch(ArithmeticException e)
   {
    System.out.println(e);
   }
```

```
//inner try block 2
   try{
   int a[]=new int[5];
     //assigning the value out of array bounds
    a[5]=4;
    }
     //catch block of inner try block 2
   catch(ArrayIndexOutOfBoundsException e)
   {
     System.out.println(e);
   }

        System.out.println("other statement");
  }
  //catch block of outer try block
  catch(Exception e)
  { System.out.println("handled the exception (outer
catch)");
  }
     System.out.println("normal flow..");
 }
}
```

```
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..
```

finally clause

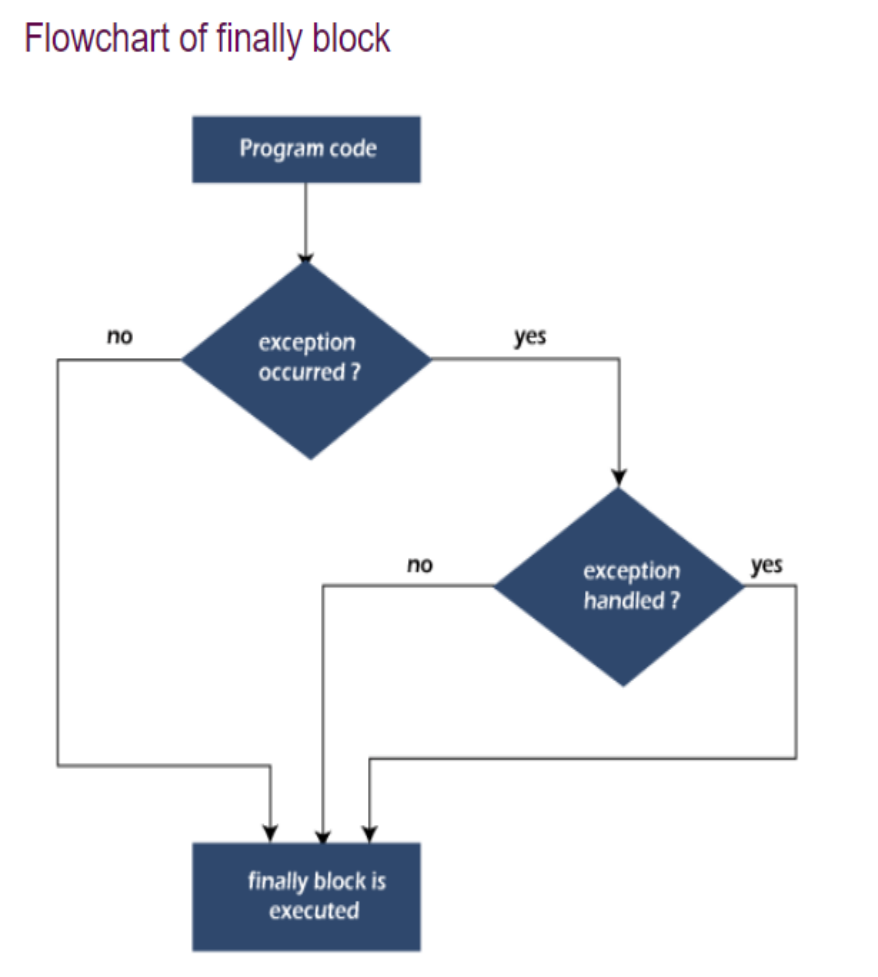
- **Try-catch-finally**
  - **Finally section includes code that is always executed before the block exits**
    - **Executes if try or catch happens**
  - **Usually used for**
    - **Doing cleanup**
      - **Closing files and releasing system resources.**
  - **A return statement in the try clause will execute the finally clause before returning**
    - **This is stylistically not good since it is confusing to the reader**

```
try {
        // statements
}
catch( Exception-Type1 e)
{
        // statements to process exception 1
}
..
..
finally {
  ....
}
```

finally clause

Three cases where the finally clause executes:

- no exception occurs in the try block
- an exception occurs in the try block and is caught
- an exception occurs in the try block but doesn't match any catch



Finally example

```
public void processFile() {
    processing = true;
    try {
        ...
    }
    catch (IOException e) {
        e.printStackTrace();
    }
    finally {
        processing = false;
    }
}
```

Example

```
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 the code...");
  }
}
```

```
5
finally block is always executed
rest of the code...
```

Exception *throw*

- 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.

Exception *throw*

```
class WithExceptionCatchThrow{
        public static void main(String[] args){
                int a,b;  float r;  a = 7;   b = 0;
                try{
                    r = a/b;
                    System.out.println("Result is " + r);
                }
                catch(ArithmeticException e){
                        System.out.println(" B is zero);
                        throw e;
                }
                System.out.println("Program is complete");
        }
}
```

Program Does Not reach here when exception occurs

Exception *throw*

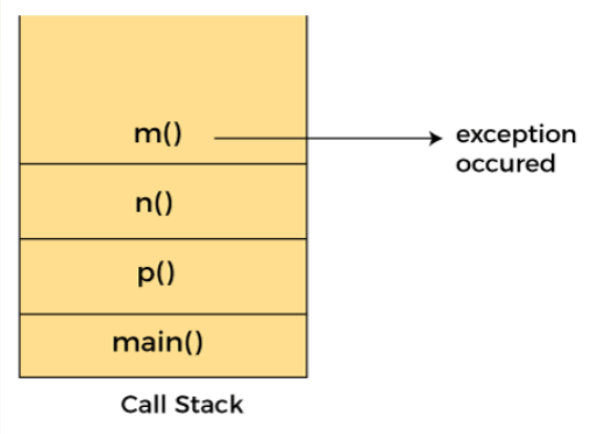
```
class WithExceptionCatchThrowFinally{
        public static void main(String[] args){
                int a,b; float r; a = 7; b = 0;
                try{
                    r = a/b;
                    System.out.println("Result is " + r);
                }
Program reaches here  catch(ArithmeticException e){
                        System.out.println(" B is zero);
                        throw e;
                }
                finally{
                 System.out.println("Program is complete");
                }
        }
}
```

Java Exception Propagation

- **An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.**
- **This is called exception propagation.**

Java Exception Propagation

```
class TestExceptionPropagation1{
  void m(){
    int data=50/0;
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation1 obj=new TestExceptionPropagation1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```



Output:

```
exception handled
normal flow...
```

Java throws

- 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.

- **Syntax of Java throws**

```
return_type method_name() throws exception_class_name{
//method code
}
```

Example


```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();   }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}   }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");   }
}
```

```
exception handled
normal flow...
```

Examples 1

```
class JavaException {
   public static void main(String args[]) {
      try {
         int d = 1;
         int n = 20;
         int fraction = n / d;
         int g[] = {1 };
         g[20] = 100;
      } catch(Exception e){
         System.out.println("In the catch block due to Exception = "+e);
  } catch (ArithmeticException e) {
         System.out.println("In the catch block due to Exception = " + e);
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("In the catch block due to Exception = " + e);
      }
      System.out.println("End Of Main");
   }
}
```

```
2   class JavaException {
3
4       public static void main(String args[]) {
5           try {
6               int d = 1;
7               int n = 20;
8               int fraction = n / d;
                int g[] = {1};
10              g[20] = 100;
11          } catch (Exception e) {
12              System.out.println("In the catch block due to Exception =
            } catch (ArithmeticException e) {
14              System.out.println("In the catch block due to Exception =
            } catch (ArrayIndexOutOfBoundsException e) {
16              System.out.println("In the catch block due to Exception =
```

Syntax error

Example 2

```
try {
  Student student = new UndergraduateStudent();
  GraduateStudent graduateStudent = (GraduateStudent) student;
  // process the object
} catch (ClassCastException cce) {
  // Object is not of type graduate student.
  // do some operation to recover from the error
}
```

Example 3

```
try {
  if (myObject.getField1().equals(someObject)) {
    int index = myObject.getIndex();
    int value = Integer.parseInt(JOptionPane.showInputDialog(null,
                                                "Enter a number"));

      myArray[index] = value;
    } catch (NullPointerException npe) {
      System.out.println("Null pointer " + npe);
      System.exit(0);
    } catch (ArrayIndexOutOfBoundsException aiofbe) {
      System.out.println("Array index out of range " + aiofbe);
      return;
    } catch (NumberFormatException nfe) {
      System.out.println("Invalid entry; exception " + nfe);
      return;
    }
  }
```