CS2833: Modular Software Development

# Collections

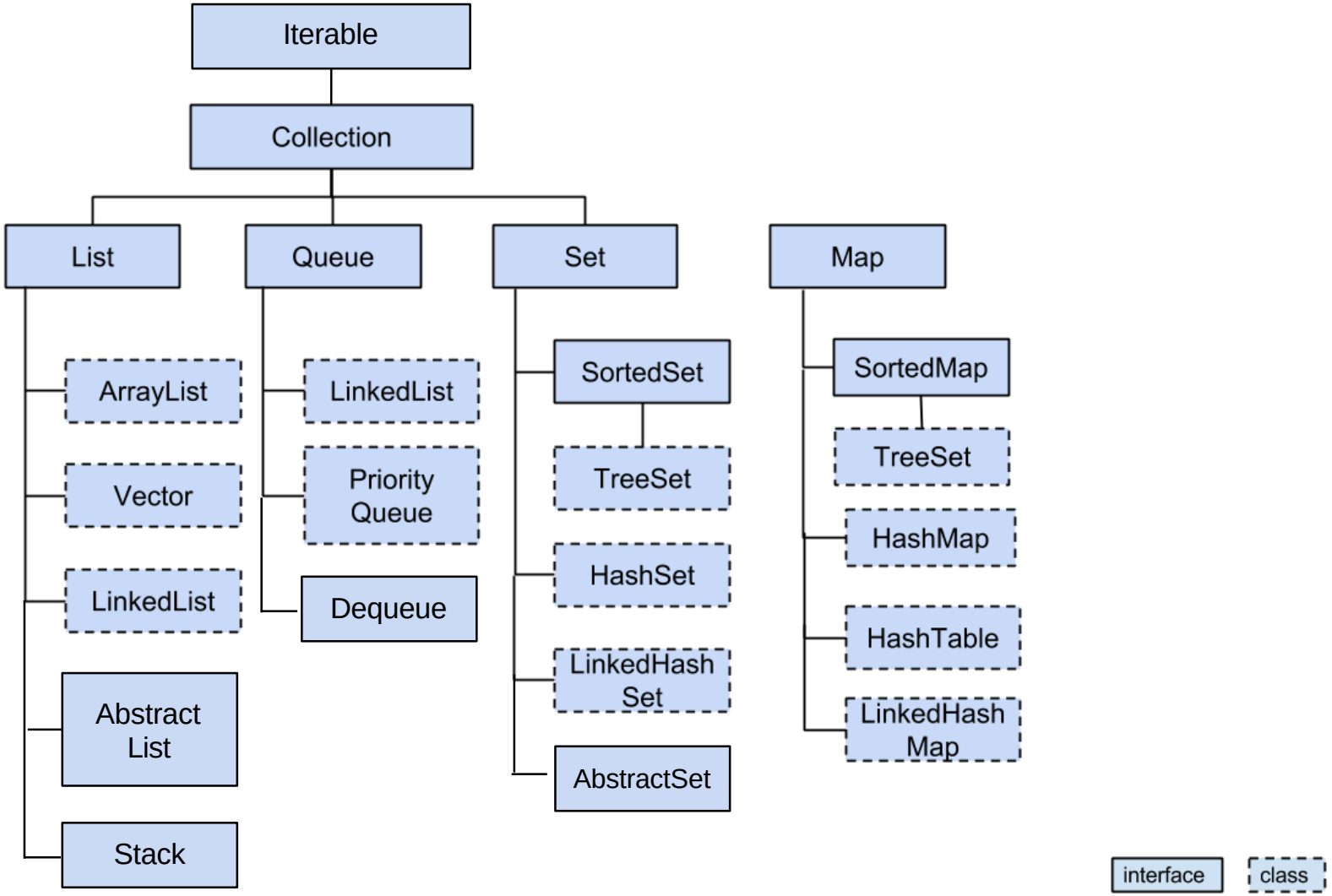
Department of Computer Science and Engineering
University of Moratuwa

# Collections Framework

- Collections are used to store, retrieve, manipulate, and communicate aggregate data
- All collections frameworks contain the following:

- **Interfaces** : These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation.
- **Implementations** : These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms** : These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

# Collection Structure



# Core Collection Interface

- **Collection -** The root of the collection hierarchy. Its a group of objects known as its elements. Java doesn't provide any direct implementation of this interface but provides implementation of more specific sub interfaces, such as Set and List.
- **Set -** A collection without duplicate elements. eg. represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.
- **List -** An ordered collection (sequence). Lists can contain duplicate elements.
- **Queue -** A collection used to hold multiple elements prior to processing.
- **Map -** An object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value.
  eg. Hashtable

# The Collection Interface

*List<String > list = new ArrayList <String>( );*

- This statement creates a new ArrayList ( an implementation of the List interface), initially containing all the elements in c. T
- The ArrayList has been up casted to a List.
- This approach won't always work, because some classes have additional functionality. For example, LinkedList has additional methods that are not in the List interface. Hence upcasting would loose these methods in ArrayList.

# Collection Bulk Operations

- **containsAll -** Returns true if the target Collection contains all of the elements in the specified Collection.
- **addAll -** Adds all of the elements in the specified Collection to the target Collection.
- **removeAll -** Removes from the target Collection all of its elements that are also contained in the specified Collection.
- **retainAll** - Removes from the target Collection all its elements that are not also contained in the specified Collection. That is, it retains only those elements in the target. Collection that are also contained in the specified Collection.
- **clear -** Removes all elements from the Collection.

# List Container

- Lists promise to maintain elements in a particular sequence.
- There are two types of List:

- **ArrayList -**

  Which excels at randomly accessing elements, but is slower when inserting and removing elements in the middle of a List.

- **LinkedList -**

  Which provides optimal sequential access, with inexpensive insertions and deletions from the middle of the List. A LinkedList is relatively slow for random access, but it has a larger feature set than the ArrayList .

# Iterators

- An iterator is an object whose job is to move through a sequence and select each object in that sequence without the client programmer knowing or caring about the underlying structure of that sequence.
- The Java Iterator can move in only one direction
- An iterator can:
  - Ask a Collection to hand you an Iterator using a method called ***iterator ()***, this Iterator will be ready to return the first element in the sequence
  - Get the next object in the sequence with ***next ()***
  - See if there are any more objects in the sequence with ***hasNext ()***
  - Remove the last element returned by the iterator with ***remove ()***
- The ListIterator is a more powerful subtype of Iterator that is produced only by List classes. While Iterator can only move forward, ListIterator is bidirectional

# Iterators

```
import java.util.*;
class TestJavaCollection1{
    public static void main(String args[]){
        ArrayList<String> list=new ArrayList<String>();//Creating arraylist
        list.add("Car");//Adding object in arraylist
        list.add("Bus");
        list.add("Jeep");
        list.add("Van");

        //Traversing list through Iterator
        Iterator itr=list.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```

# Exercise

Consider the four core interfaces; **Set** , **List** , **Queue** , and **Map** . For each of the following four assignments, specify which of the four core interfaces is best - suited

List
- Whimsical Toys Inc (WTI) needs to record the names of all its employees. Every month, an employee will be chosen at random from these records to receive a free toy.

Set
- WTI has decided that each new product will be named after an employee but only first names will be used, and each name will be used only once. Prepare a list of unique first names.

Map
- WTI decides that it only wants to use the most popular names for its toys. Count up the number of employees who have each first name.

Queue
- WTI acquires season tickets for the local lacrosse team, to be shared by employees. Create a waiting list for this popular sport.

CS2832: Modular Software Development

# Exception Handling

Department of Computer Science and Engineering
University of Moratuwa

# Java Exception

- Java exception is an object that describes the exceptional condition that has occurred in code
  - The exception object contains:
    - Information about the error
    - Error type
    - The state of the program when the error occurred
  - Later this exception is recognized and processed

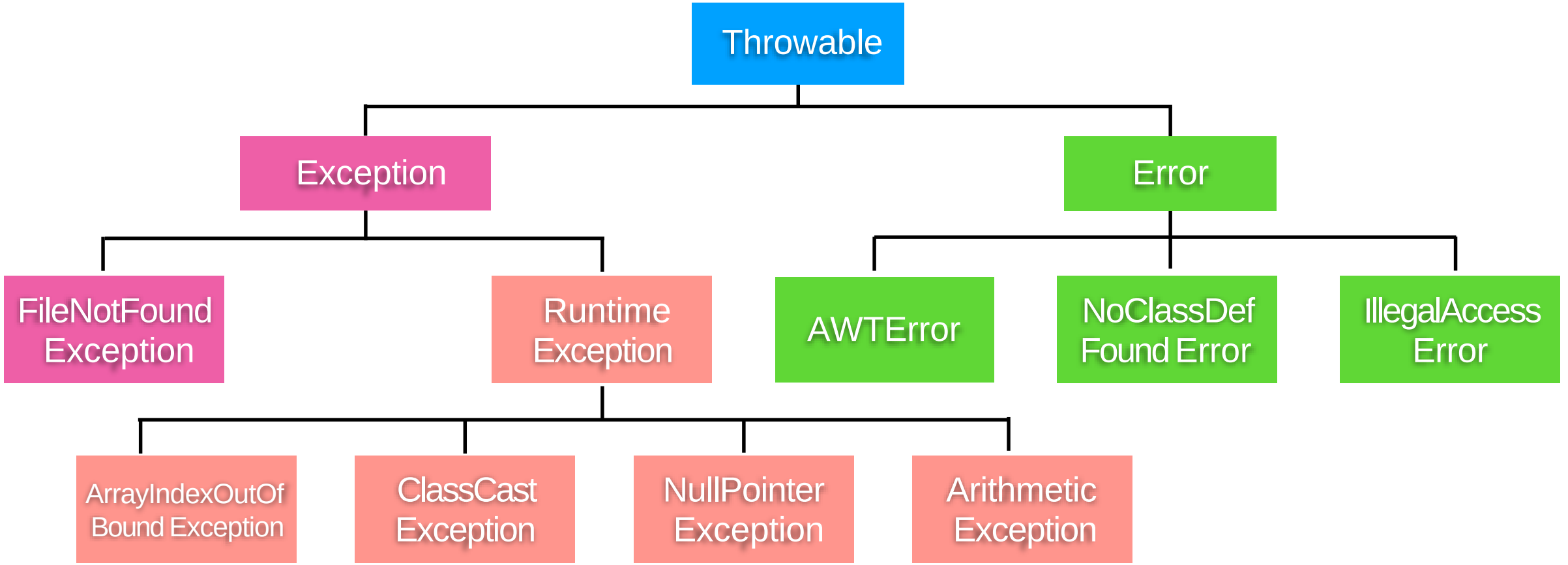
# Types of Exceptions

- Checked Exceptions
- Exceptional conditions that a program should anticipate and recover from Ex: I/O Error

- Runtime Exceptions
- Exceptional conditions that are internal to the application
- Could be a bug, logic error or improper use of an API
  Ex:NullPointerException

- Errors
- Exceptional conditions that are external to the application

# Exception Hierarchy



# Exception Handling

- Java exception handling is managed by the following five keywords:
  *try, catch, throws, throw, finally*
- When an error occurs, the following sequence occurs
  - Get the exception
  - Throws the exception - creating an exception object and handing it to the runtime system
  - Catch the exception
  - Handle the exception

# The try- catch Block

```
try {
        ……....
}
catch ( ExceptionType name) {
        ……..……..
}
```

```
public void readFile (String fileName) {
      BufferedReader br = null;
      try {
         br = new BufferedReader (new FileReader ( fileName ));
            String line = br.readLine ();
      }
       catch ( FileNotFoundException ex) {
            System.out.println ( ex.getMessage ());
       } catch ( IOException ex) {
          System.out.println ("IOException ");
       }
}
```

# The try- catch Block

- The try block contains code that could throw an exception

- If an exception occurs within the try block, corresponding handler will handle the exception

- The catch block contains code that is executed if and when the exception handler is invoked.
  - Error recovery
  - Prompt the user to make a decision
  - Propagate the error up to a higher- level handler
- A method to handle multiple exceptions is to have multiple try - catch/catch blocks

# The throws Clause

- If the method is not handling checked exceptions, then method must specify that it can throw those exceptions

  *public void readFile(String fileName ) **throws IOException** {…}*

  - An alternative to try - catch block

- Indicates to client programmers what exceptions they may have to deal with when they invoke the method
- Not every exception that can be thrown by a method need to be put in a throws clause

# The finally Block

- The finally block always executes when the try block exits
- Even if an exception occurs, finally block is executed

- However if the JVM while try/catch block is executing or it the thread executing try - catch block is interrupted or killed, the finally block might not be executed

- The finally block is primarily used for code cleanup rather than exception handling

# The finally Block

```
finally {
        if ( br != null) {
                br.close();
        }
        else{
                System.out.println (“Not Open”);
        }
}
```

# The throw Clause

- It is possible for the program to throw an exception explicitly rather than by the JRE
- The exception that is thrown can be either a standard system exception or manually created exception
- The flow of execution stops immediately after the throw statement, it looks for a catch block which can handle the exception type

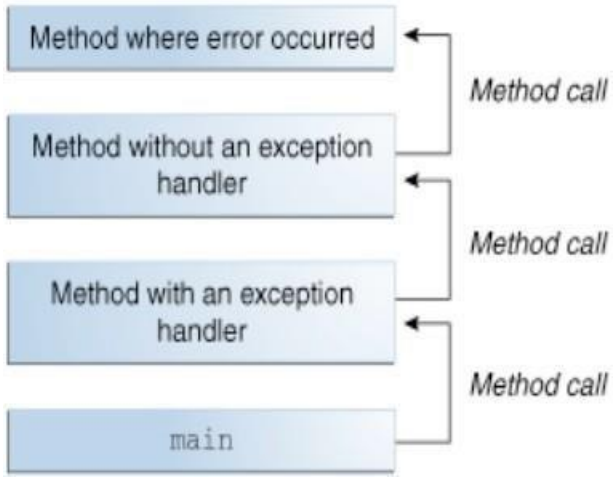
throws

```
void simpleMethod() throws ArithmeticException, NullPointerException{
 //Body of the method
}
```

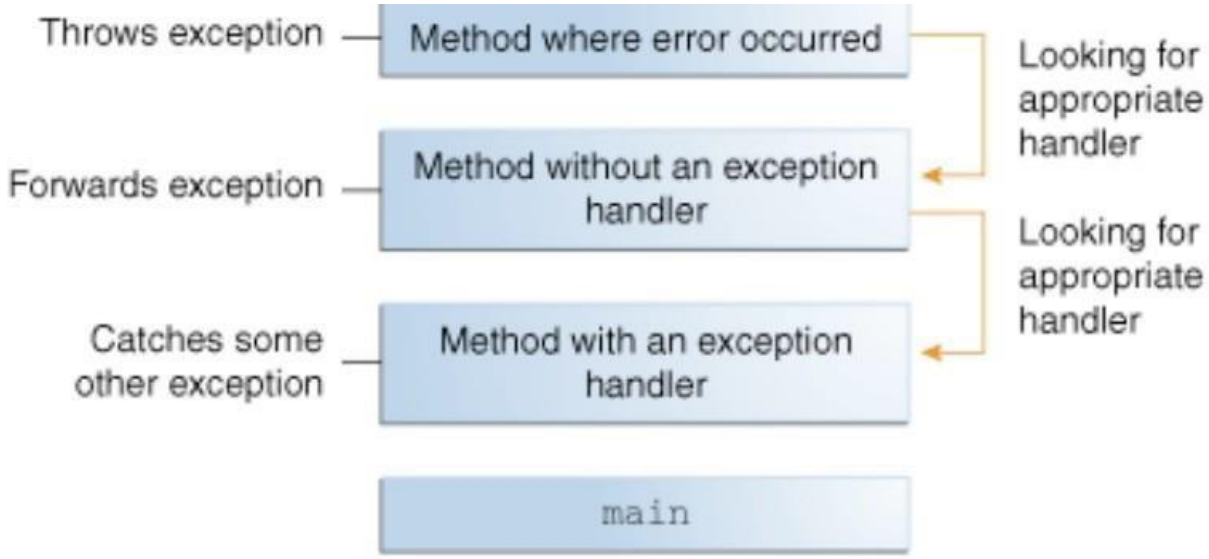
throw

```
void simpleMethod() {
        throw new ArithmeticException("divided by zero!!");
}
```

# Exception Handling




When an exception is occurred, default exception handler tries the nearest method, to see if it has an appropriate exception handler. If not it goes to the next method and so on. If no matching exception handler is found, the default exception handler halts the program and prints out the stack trace.



# Creating Your Own Exceptions

- To create your own exception class, you must inherit from an existing exception class
- As a general practice you only sub class the Exception class. This ensures, that the compiler checks if user defined classes and methods are dealt with properly.
- However if you are writing system or hardware related utilities, then you can use sub classing from either Error or RuntimeException classes

# Creating your own Exception

```
class InvalidNumberException extends Exception{
            InvalidNumberException(String s){
            super(s);
 }
}


class MyClass{
  public void validate(int number) throws  InvalidNumberException{
    if(number<18)
     throw new  InvalidNumberException("not valid");
    else
     System.out.println(“The number if correct!");
   }


}
```

# Advantages of Exception Handling

- Separation of program logic from error handling
- Error propagation
- Grouping and differentiating error types

# Questions?

Thank You