

Object Oriented Programming

Prepared by Dr. Yasser Abdelhamid

IT223
Object Oriented Programming

Lecture 10



OUTLINE

- Exceptions
- Object Oriented design

EXCEPTIONS

EXCEPTION

❖ An exception is an event that disrupts the normal flow of the program.

EXAMPLE

```
1
2 public class ExceptionClass {
3     static int[] x = {1,2,3};
4     public static void main(String[] args) {
5         for(int i=0; i<=3;i++){
6             System.out.println("Value of index "+ i + " is "+ x[i]);
7         }
8     }
9 }
```

```
run:
Value of index 0 is 1
Value of index 1 is 2
Value of index 2 is 3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for
length 3
    at ExceptionClass.main(ExceptionClass.java:14)
[C:\Users\start\AppData\Local\NetBeans\Cache\12.5\executor-snippets\run.xml:111: The following
error occurred while executing this line:
[C:\Users\start\AppData\Local\NetBeans\Cache\12.5\executor-snippets\run.xml:68: Java returned:
1
BUILD FAILED (total time: 0 seconds)
```

WHAT IS AN EXCEPTION?

- ❖ An Exception is an **unwanted event** that **interrupts the normal flow** of the program.
- ❖ When an exception occurs program execution gets **terminated**.
- ❖ In such cases we get a **system generated error message**.
- ❖ By handling the exceptions we can **provide a meaningful message to the user about the issue rather than a system generated message**, which may not be understandable to a user.

WHY AN EXCEPTION OCCURS?

- ❖ There can be several reasons that can cause a program to **throw exception**.
- ❖ For example:
 - Opening a non-existing file in your program,
 - Network connection problem,
 - Bad input data provided by user etc.

EXCEPTION HANDLING

❖ If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user.

❖ Example:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at
ExceptionDemo.main(ExceptionDemo.java:5)
ExceptionDemo : The class name
main : The method name
ExceptionDemo.java : The filename
java:5 : Line number
```

❖ Instead of presenting this message, we can display a friendly message to the user specifying what went wrong, and giving him the chance to continue running the program without break.

EXCEPTION HANDLING MECHANISM

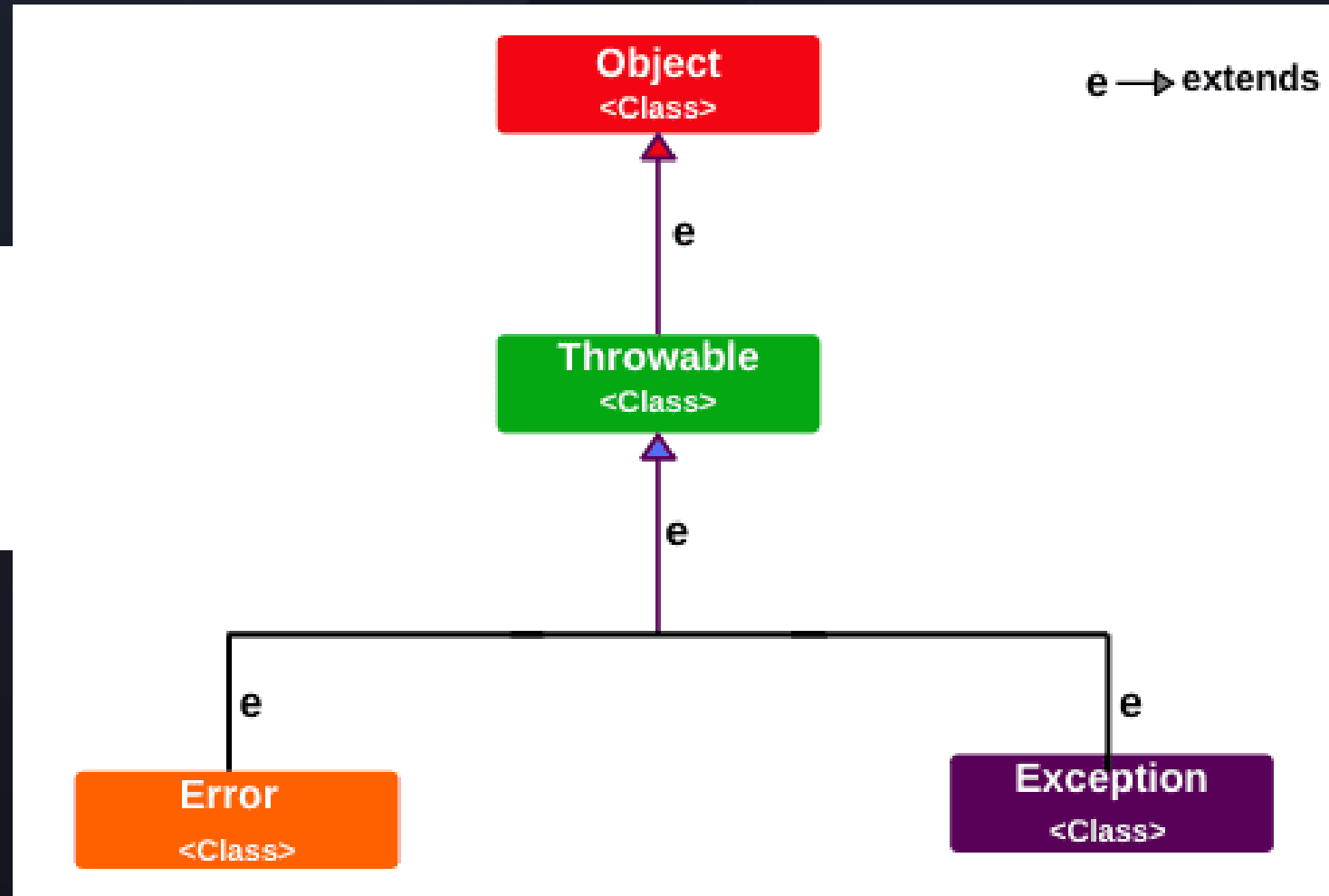
❖ Exception Handling is a mechanism to handle runtime errors so that when an exception occurs, the program can still go back to the normal flow.



HIERARCHY OF JAVA EXCEPTION CLASSES

`java.lang.Throwable` class is the root class of Java Exception hierarchy inherited by two subclasses:

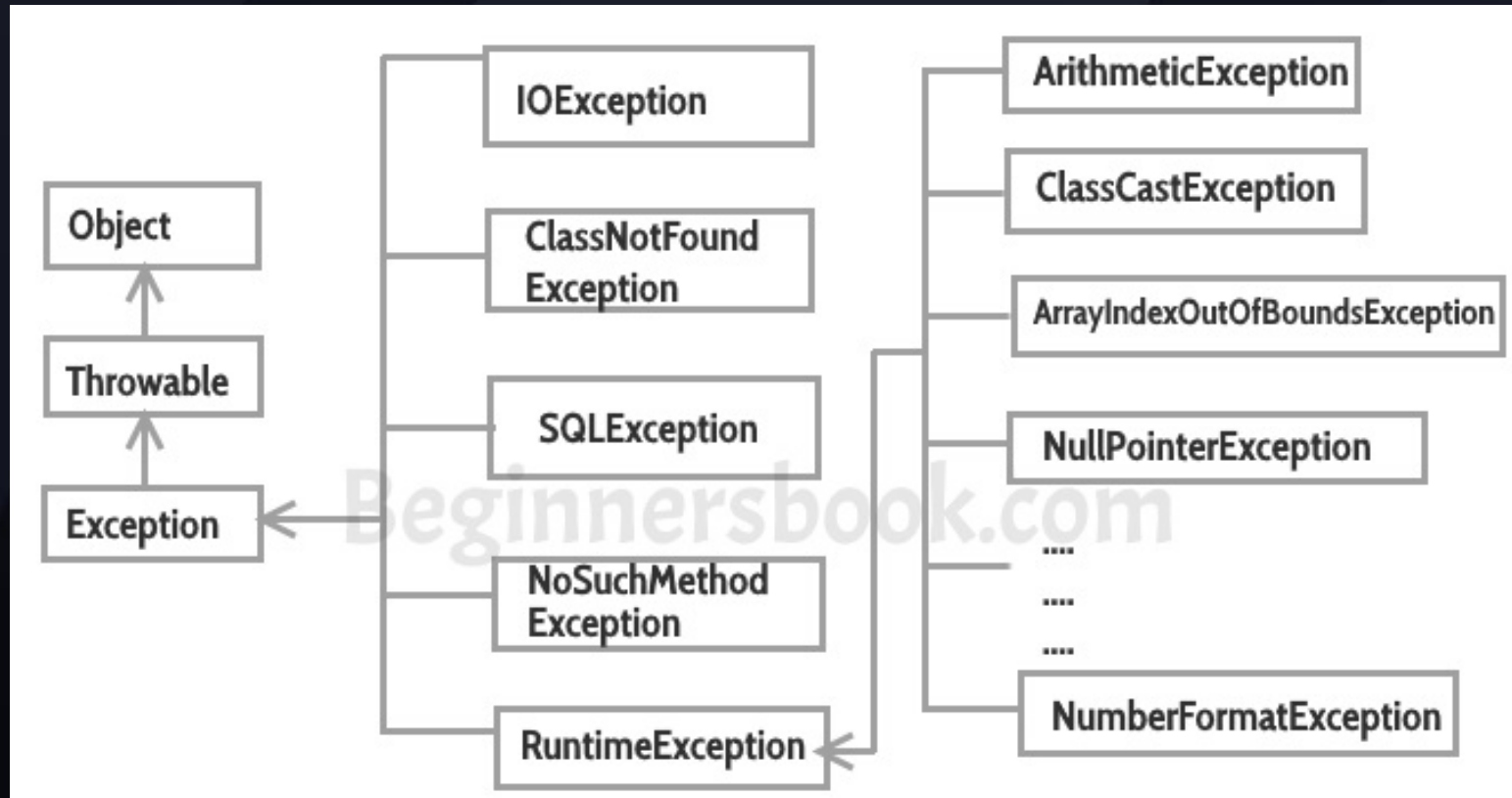
- `Exception` and
- `Error`.



DIFFERENCE BETWEEN ERROR AND EXCEPTION

- ❖ **Errors** indicate that something **severe** enough has gone wrong, **the application should crash** rather than try to handle the error.
- ❖ **Exceptions** are events that occurs in the code. A programmer can handle such conditions and take **necessary corrective actions**.

TYPES OF JAVA EXCEPTIONS



CHECKED EXCEPTIONS

- ❖ All exceptions **other than Runtime Exceptions** are known as Checked exceptions.
- ❖ The **compiler checks them during compilation** to see whether the programmer has handled them or not.
- ❖ If these exceptions are not handled/declared in the program, you will get compilation error.
 - For example:
 - SQLException, IOException, ClassNotFoundException etc.

UNCHECKED EXCEPTIONS

- ❖ **Runtime Exceptions** are also known as Unchecked Exceptions.
- ❖ These exceptions **are not checked at compile-time** so compiler does not check whether the programmer has handled them or not but **it's the responsibility of the programmer** to handle these exceptions and **provide a safe exit**.
 - For example:
 - `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException` etc.

TRY CATCH BLOCK

❖ Exceptions are handled in java using **Try-Catch code blocks**.

```
try
{
    //statement1;
    //statement2;
    //statement3;
    //...
}
catch(exception-type)
{
    //code that handles exception.
}
```

Enclosing statements in try block *may* throw an exception.

catch block catches and handles a *type* of an exception.

Try-catch block

TRY BLOCK

- ❖ The try block contains **set of statements where an exception might occur.**
- ❖ A try block is always **followed by a catch block**, which handles the exception that occurs in associated try block.
- ❖ A try block **must be followed by catch blocks or finally block or both.**

```
try{  
    //statements that may cause an exception  
}
```

CATCH BLOCK

- ❖ A catch block is where you **handle the exceptions**.
- ❖ This block **must follow the try block**.
- ❖ A single try block **can have several catch blocks** associated with it.
- ❖ When an exception occurs in try block, the **corresponding catch block** that handles that particular exception executes.

```
try
{
    //statements that may cause an exception
}
catch (exception(type) e(object))
{
    //error handling code
}
```

GENERIC EXCEPTION HANDLER

- ❖ The generic exception handler can **handle all the exceptions** but **you should place it at the end of catch blocks**.
- ❖ If you place it before all the catch blocks then it will display its generic message. which may not be meaningful to the user.

```

class Example1 {
    public static void main(String args[]) {
        int num1, num2;
        try {
            /* We suspect that this block of statement can throw exception */
            num2 = 62 / num1;
            System.out.println(num2);
            System.out.println("Hey I'm at the end of try block");
        }
        catch (ArithmeticException e) {
            /* This block will only execute if any Arithmetic exception occurs in try block */
            System.out.println("You should not divide a number by zero");
        }
        catch (Exception e) {
            /* This is a generic Exception handler which means it can handle all the exceptions.
            This will execute if the exception is not handled by previous catch blocks. */
            System.out.println("Exception occurred");
        }
        System.out.println("out of try-catch block");
    }
}

```

Output:
 You should not divide a number by zero
 Out of try-catch block...

CATCHING MULTIPLE EXCEPTIONS

```
class Example{
    public static void main(String args[]){
        try{
            int arr[]=new int[7];
            arr[4]=30/0; // divide by zero
            System.out.println("Last Statement of try block");
        }
        catch(ArithmeticException e){
            System.out.println("You should not divide a number by zero");
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Accessing array elements outside of the limit");
        }
        catch(Exception e){
            System.out.println("Some Other Exception");
        }
        System.out.println("Out of the try-catch block");
    }
}
```

Output:

You should not divide a number by zero
Out of the try-catch block

EXAMPLE:2

```
class Example{
    public static void main(String args[]){
        try{
            int arr[]=new int[7];
            arr[10]=10/5;
            System.out.println("Last Statement of try block");
        }
        catch(ArithmeticException e){
            System.out.println("You should not divide a number by zero");
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Accessing array elements outside of the limit");
        }
        catch(Exception e){
            System.out.println("Some Other Exception");
        }
        System.out.println("Out of the try-catch block");
    }
}
```

Output:

Accessing array elements outside of the limit
Out of the try-catch block

NESTED TRY CATCH BLOCK IN JAVA - EXCEPTION HANDLING

```
class Nest{
    public static void main(String args[]){
        //Parent try block
        try{
            //Child try block1
            try{
                System.out.println("Inside block1");
                int b =45/0;
                System.out.println(b);
            }
            catch(ArithmeticException e1){
                System.out.println("Exception: e1");
            }
            //Child try block2
            try{
                System.out.println("Inside block2");
                int b =45/0;
                System.out.println(b);
            }
            catch(ArrayIndexOutOfBoundsException e2){
                System.out.println("Exception: e2");
            }
            System.out.println("Just other statement");
        }
        ...
    }
}
```

```
catch(ArithmeticException e3){
    System.out.println("Arithmetic Exception");
    System.out.println("Inside parent try catch block");
}
catch(ArrayIndexOutOfBoundsException e4){
    System.out.println("ArrayIndexOutOfBoundsException");
    System.out.println("Inside parent try catch block");
}
catch(Exception e5){
    System.out.println("Exception");
    System.out.println("Inside parent try catch block");
}
System.out.println("Next statement..");
}
```

```
Inside block1
Exception: e1
Inside block2
Arithmetic Exception
Inside parent try catch block
Next statement..
```

JAVA FINALLY BLOCK - EXCEPTION HANDLING

- ❖ A finally block contains all the crucial statements that **must be executed whether exception occurs or not.**
- ❖ The statements present in this block will always execute regardless of whether exception occurs in try block or not such as **closing a connection, stream, file,** etc.

JAVA FINALLY BLOCK - EXCEPTION HANDLING

```
class Example
{
    public static void main(String args[]) {
        try{
            int num=121/0;
            System.out.println(num);
        }
        catch(ArithmeticException e){
            System.out.println("Number should not be divided by zero");
        }
        /* Finally block will always execute
        * even if there is no exception in try block
        */
        finally{
            System.out.println("This is finally block");
        }
        System.out.println("Out of try-catch-finally");
    }
}
```

Output:

Number should not be divided by zero
This is finally block
Out of try-catch-finally

JAVA FINALLY BLOCK - EXCEPTION HANDLING

```
class JavaFinally
{
    public static void main(String args[])
    {
        System.out.println(JavaFinally.myMethod());
    }
    public static int myMethod()
    {
        try {
            return 112;
        }
        finally {
            System.out.println("This is Finally block");
            System.out.println("Finally block ran even after return statement");
        }
    }
}
```

Output:

```
This is Finally block
Finally block ran even after return statement
112
```

OBJECT ORIENTED DESIGN

UNDERSTAND THE REQUIREMENTS

- ❖ Gain a thorough understanding of the project requirements and goals.
- ❖ Identify the key functionalities, user interactions, and system constraints.
- ❖ This understanding will guide your design decisions.

IDENTIFY OBJECTS AND THEIR RESPONSIBILITIES

- ❖ Identify the main objects or entities in the system based on the requirements.
- ❖ Determine the responsibilities or behaviors that each object should have.
- ❖ Consider what actions or operations objects need to perform to fulfill their roles.

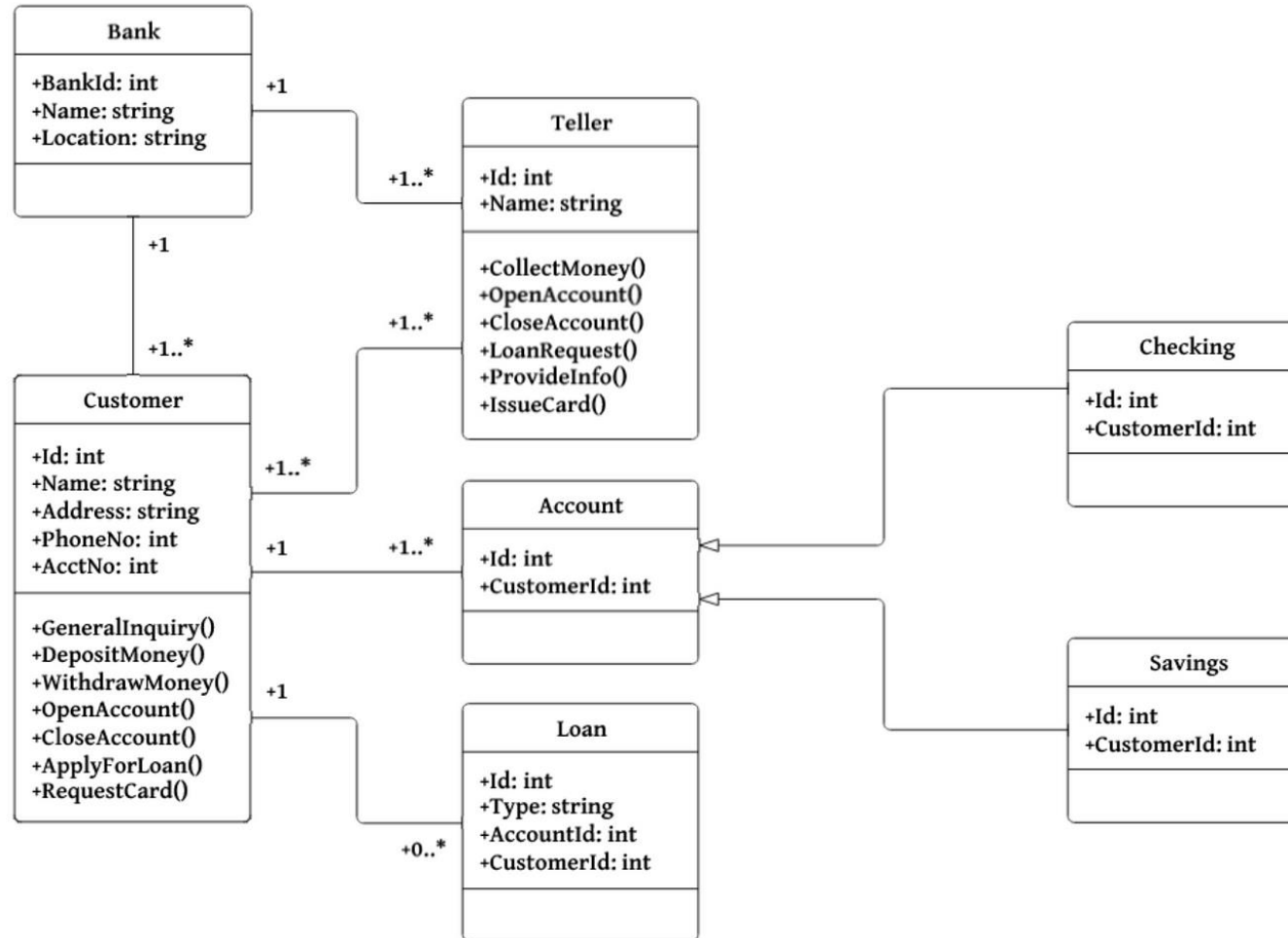
DEFINE RELATIONSHIPS BETWEEN OBJECTS

- ❖ Determine the relationships between objects.
- ❖ Identify associations, aggregations, compositions, and generalizations (inheritance) between objects.
- ❖ These relationships define how objects interact and collaborate to achieve system functionality.

CREATE CLASS DIAGRAMS

- ❖ Create class diagrams to visualize the objects, their attributes, and methods.
- ❖ Use UML (Unified Modeling Language) notation to represent classes, relationships, and multiplicity.
- ❖ Refine and iterate on the class diagrams as the design evolves.

EXAMPLE



ENCAPSULATE DATA AND BEHAVIOR

- ❖ Encapsulate the data (attributes) and behavior (methods) within each class.
- ❖ Define appropriate access modifiers (public, private, protected) for class members to ensure encapsulation and data integrity.
- ❖ Consider applying principles such as information hiding and the single responsibility principle.

APPLY DESIGN PATTERNS

- ❖ Consider applying design patterns to solve common design problems and improve the structure of your design.
- ❖ Design patterns provide proven solutions and can enhance flexibility, maintainability, and extensibility.
- ❖ Choose the appropriate patterns based on the system requirements and constraints.

REFINE AND OPTIMIZE THE DESIGN

- ❖ Continuously refine and optimize the design.
- ❖ Review the class relationships, responsibilities, and interactions to ensure cohesion and loose coupling.
- ❖ Eliminate redundancy and ensure the design follows best practices and design principles.

CONSIDER DESIGN TRADE-OFFS

- ❖ Consider trade-offs between different design aspects, such as performance, maintainability, and scalability.
- ❖ Evaluate the impact of design decisions on system performance, memory usage, and code complexity.
- ❖ Seek a balance between the competing design considerations.

VALIDATE THE DESIGN

- ❖ Validate the design by reviewing it with stakeholders, domain experts, or fellow developers.
- ❖ Seek feedback and incorporate necessary changes or improvements.
- ❖ Use design reviews to catch potential issues early and ensure the design aligns with the project requirements.

DOCUMENT THE DESIGN

- ❖ Document the design using appropriate documentation techniques. Create clear and concise documentation that describes the design decisions, rationale, and key design elements.
- ❖ Use diagrams, comments, and other supporting materials to enhance understanding and maintainability.

GENERAL

- ❖ Remember that object-oriented design is an iterative process.
- ❖ Refine the design as you gain more insights and feedback.
- ❖ Keep the design flexible enough to accommodate changes and future enhancements.
- ❖ Regularly review and refine the design as the project progresses to maintain a high-quality, scalable, and maintainable codebase.

SUMMARY

- Exceptions
- Object Oriented design



Thank You

www.midocean.ae

