

Exception Handling

Objectives

- © At the end of the lesson, the student should be able to:
 - Define exceptions
 - Handle exceptions using a simple try-catch-finally block
 - Use throw and throws clause
 - create and use user Defined Exception

Exceptions

@ An exception

- is an event that interrupts the normal processing flow of a program. This event is usually some error of some sort.
- This causes our program to terminate abnormally.

Exceptions

@ The term *exception* is shorthand for the phrase “exceptional event “

@ Definition:

- “ *An exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions.”
- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.

Exceptions

- Ⓢ Many kinds of errors can cause exceptions
- Ⓢ problems ranging from serious hardware errors, such as a hard disk crash to simple programming errors, such as trying to access an out-of-bounds array element

Examples

- ⌚ Some examples of exceptions:
 - **ArrayIndexOutOfBoundsException** exceptions, which occurs if we try to access a non-existent array element
 - **NumberFormatException**, which occurs when we try to pass as a parameter a non-number in the `Integer.parseInt()` method.

Throwing an Exception

- ④ When an exception occurs within a Java method, the method creates an **exception object** and hands it off to the runtime system.
- ④ The exception object contains information about the exception, including its type and the state of the program when the error occurred.
- ④ Creating an exception object and handing it to the runtime system is called ***throwing an exception.***

Catching an Exception

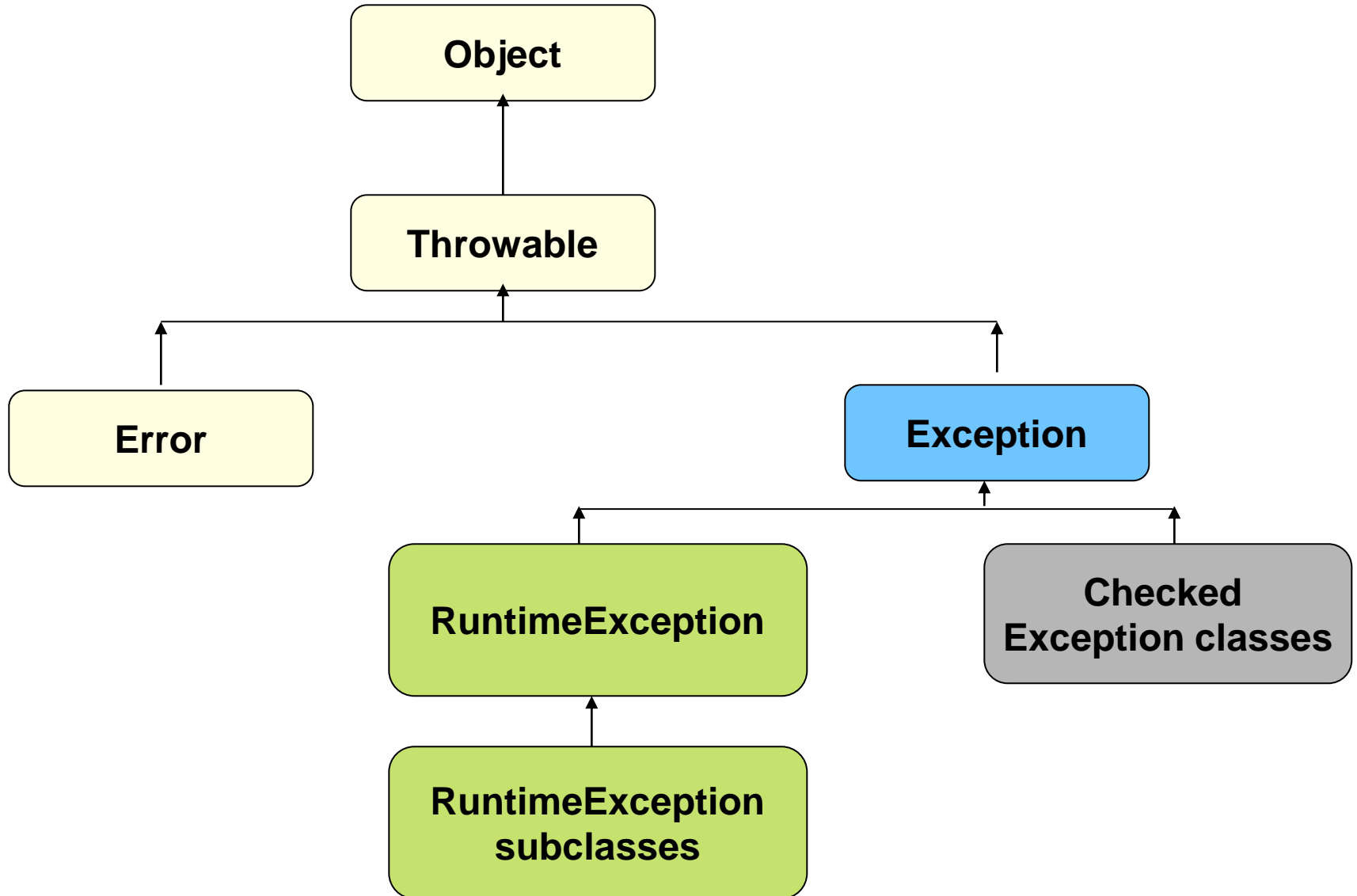
- ⓐ After a method throws an exception, the runtime for an appropriate exception handler
- ⓐ An exception handler is considered appropriate if the type of the exception thrown is the same as the type of exception handled by the handler
- ⓐ Handling the Exception by the handler is called ***“catching an exception”***
- ⓐ In case the runtime system does not find an exception handler, the runtime system (and consequently the Java program) terminates.

Why to handle Exception?

☺ By using exceptions to manage errors, Java programs have the following advantages over traditional error management techniques:

- **Advantage 1:** Separating Error Handling Code from "Regular" Code
- **Advantage 2:** Propagating Errors Up the Call Stack
- **Advantage 3:** Grouping Error Types and Error Differentiation

Java Exception Class hierarchy



Types of Exception

- Ⓒ Java Exceptions can be categorized into two broad categories
 - Checked Exceptions
 - Unchecked Exceptions (runtime exceptions)

Runtime exceptions

(Unchecked Exception)

@ Runtime exceptions are those exceptions that occur within the Java runtime system.

@ Examples:

- **ArithmeticExceptions**
- **NullPointerExceptions**
- **ArrayIndexOutOfBoundsException**

@ Such type of exceptions cannot be checked/detected by the compiler and hence are also called unchecked exceptions.

Checked Exceptions

- ④ *Checked exceptions* are not runtime exceptions and are checked by the compiler
- ④ the compiler checks that these exceptions are caught or specified.
- ④ Checked exceptions do ***not*** extend the **RuntimeException** class.
- ④ Checked exceptions ***must*** be *handled* by the programmer to avoid a compile-time error.
- ④ Example
 - **IOException**

Handling exceptions in Java

© Java exception handling is managed via five keywords:

- **try**
- **catch**
- **finally**
- **throw**
- **throws,**

try-catch-finally block

- ② To handle exceptions in Java, we use a try-catch-finally block
 - What we do in our programs is that we place the statements that can possibly generate an exception inside this block.

try-catch-finally block

@ The general form of a try-catch-finally block is,

```
try {  
    // write the statements that can generate an exception in the  
    // block  
}  
catch(<exceptionType> <varName>) {  
    //write the action your program will do if an exception  
    // of a certain type occurs  
}  
.....  
catch(<exceptionType> <varName>) {  
    //write the action your program will do if an exception  
    // of a certain type occurs  
}  
finally {  
    // add more cleanup code here  
}
```


try-catch-finally block

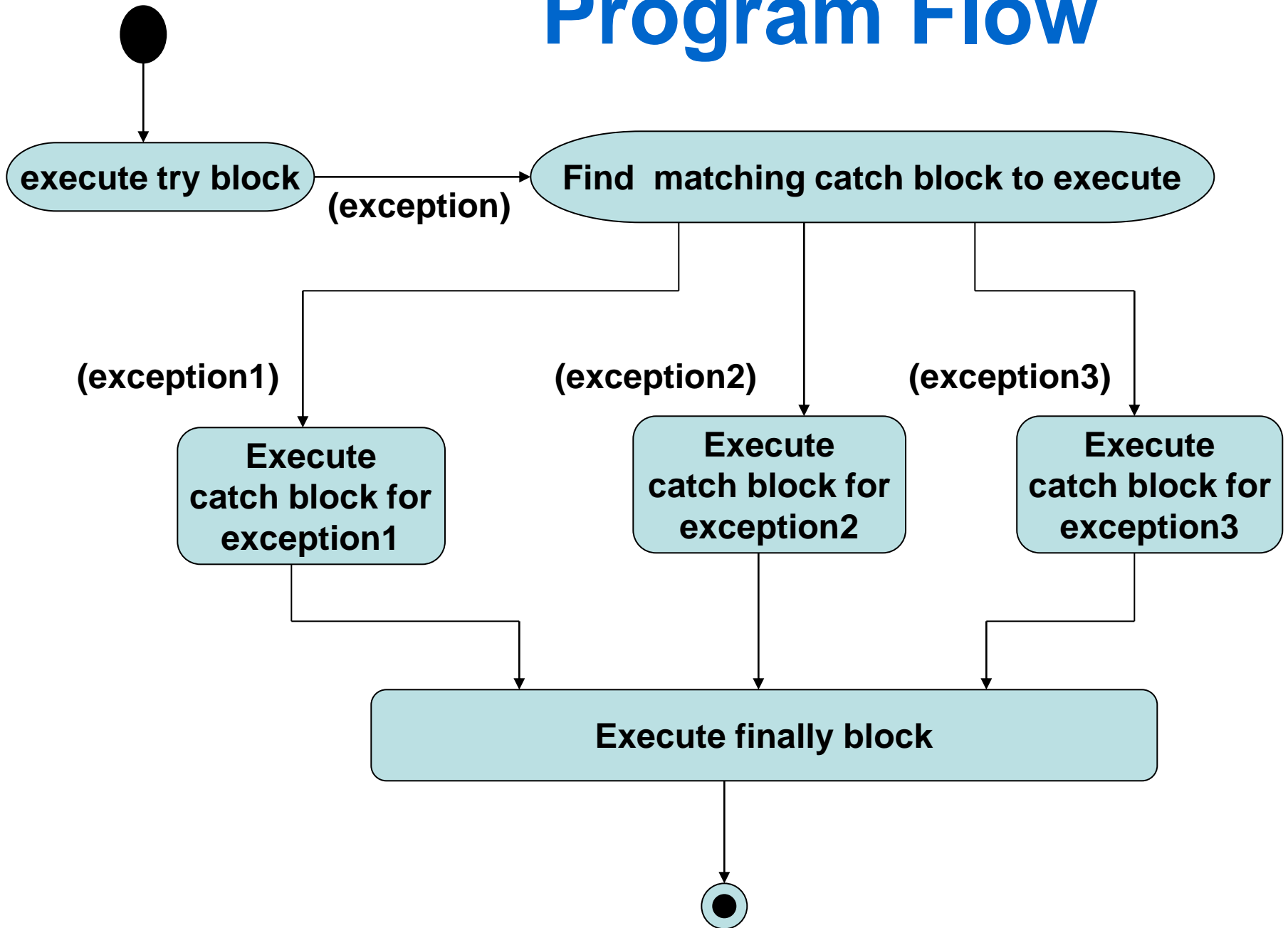
@ Key aspects about the syntax of the try-catch-finally construct:

- For each try block, there can be one or more catch blocks, but only one finally block.
- The catch blocks and finally blocks must always appear in conjunction with the try block, and in the above order.
- A try block must be followed by AT LEAST one catch block OR one finally block, or both.
- Each catch block defines an exception handle. The header of the catch block takes exactly one argument, which is the exception its block is willing to handle. The exception must be of the **Throwable** class or one of its subclasses.

try-catch-finally block

- Ⓢ A single try block may have more than one catch block
- Ⓢ A try -catch block can be nested inside a try block.
- Ⓢ A catch block handles only the exception for which it is meant for.
- Ⓢ If an exception occurs in a inner try block and is not handled by the inner catch blocks the exception can be handled by the outer catch block(s)
- Ⓢ Whether an exception occurs in a try block or not the finally block always executes.

Program Flow



Example: try-catch

```
public class ExceptionExample {  
    public static void main( String[] args ){  
        try{  
            System.out.println( args[1] );  
        }  
        catch( ArrayIndexOutOfBoundsException exp ){  
            System.out.println("Exception caught!");  
        }  
    }  
}
```

throw

- ④ Using the “throw” clause the programmer can throw an exception depending upon a certain condition desired by the programmer.
- ④ The “throw” clause can also be used to throw user defined exceptions.

```
throw new ExceptionType();
```

throw - An Example

```
class ThrowDemo
{
    public static void main(String[] args)
    {
        double balance = 3000;
        try
        {
            double amount = Double.parseDouble(args[0]);
            if (balance < amount)
                throw new BalanceException();
        }
        catch (BalanceException e)
        {
            System.out.println( "You do not have
                                sufficient balance");
        }
        System.out.println( "Hello World!");
    }
}
```

throws

- @ If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- @ This is done by including a **throws** clause in the method's declaration.
- @ The general form of a method declaration that includes a **throws** clause:

```
type method-name (parameter-list) throws exception-list  
{  
  // body of method  
}
```

- @ *exception-list* is a comma-separated list of the exceptions that a method can throw.

throws

- Ⓢ A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.
- Ⓢ All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

throws – An Example

```
class ReadFileDemo{
public static void main(String[] args) throws
    IOException,FileNotFoundException
{
    FileInputStream fis = new FileInputStream( "note.txt");
    int i;
    while ((i=fis.read())!=-1)
    {
        System.out.print((char)i);
    }
}
}
```

The method `read()` throws **IOException** and
`FileInputStream()` throws **FileNotFoundException**

User Defined Exceptions

- Ⓢ Java's built-in Exceptions handle most of the Exceptions
- Ⓢ we can also define our own exception class
- Ⓢ To create a user defined Exception we need to create a subclass of Exception
- Ⓢ In-turn which automatically becomes subclass of Throwable also

Methods Inherited by User Defined Exception class

- ② The **Exception** class does not define any methods of its own.
- ② It inherits those methods provided by **Throwable class**
- ② all exceptions, including those we create, have the methods defined by **Throwable class**

Methods of Throwable class

Method Summary	
Throwable	getCause() Returns the cause of this throwable or null if the cause is nonexistent or unknown.
String	getMessage() Returns the detail message string of this throwable.
Throwable	initCause(Throwable cause) Initializes the <i>cause</i> of this throwable to the specified value.
void	printStackTrace() Prints this throwable and its backtrace to the standard error stream.
void	printStackTrace(PrintStream s) Prints this throwable and its backtrace to the specified print stream.
void	printStackTrace(PrintWriter s) Prints this throwable and its backtrace to the specified print writer.
String	toString() Returns a short description of this throwable

Example: User defined Exception(1)

```
class InsufficientBalanceException extends Exception{  
  
    InsufficientBalanceException( String message){  
        super(name);  
    }  
  
};
```

Example: User defined Exception(2)

```
class Withdraw {  
    double balance=500;  
    public void withdraw( double amount){  
        try {  
            if (balance<amount)  
                throw new InsufficientBalanceException(  
                    "Insufficient balance");  
            System.out.println (" Transaction Successful!!");  
        }  
        catch (InsufficientBalanceException e){  
            System.out.println (e.getMessage());  
        }  
    }  
}
```

}

Example: User defined Exception(3)

```
class MyCustomException{  
    public static void main( String[] args){  
  
        Withdraw w= new Withdraw();  
        w.withdraw(700);  
        w.withdraw(300);  
        w.withdraw(600);  
    }  
};
```

Output:

Insufficient Balance
transaction Succesfull
Insufficient Balance

Source of Exceptions

@ Exceptions can be thrown by

- JVM

- @ JVM exceptions

- @ Those exceptions and errors exclusively or most logically thrown by JVM

- @ NullPointerException, StackOverflowError

- Programmatically

- @ Exceptions that are thrown by application and/or API Programmers

- @ NumberFormatException

Sources of Common Exception

@ `ArrayIndexOutOfBoundsException`

- Thrown when attempting to access an with an invalid index
- Thrown by JVM

@ `ClassCastException`

- Thrown when Attempting to cast a reference variable to type that fails Is-A test
- Thrown by JVM

@ `IllegalArgumentException`

- Thrown when a method receives an argument formatted differently than the method expects
- Thrown Programmatically

Sources of Common Exception

@ IllegalStateException

- Thrown when the state of the environment does not match the operation being attempted
 - @ E.g. using a Scanner which is closed
- Thrown programmatically

@ NullPointerException

- Thrown when attempting to access an object with a reference variable whose current value is **null**
- Thrown by JVM

@ NumberFormatException

- Thrown when a method that converts a String to a number receives a number that it cannot convert
- Thrown programmatically

Sources of Common Exception

@ AssertionError

- Thrown when a statements boolean test fails
- Thrown programmatically

@ ExceptionInitializerError

- Thrown when attempting to initialize a static variable or initialization block
- Thrown by JVM

@ StackOverflowError

- Typically thrown when a method recurses deeply
- Thrown by JVM

@ NoClassDefFoundError

- Thrown when JVM cannot find a class it needs, because of a commandline error, a classpath issue or a missing .class file
- Thrown by JVM

Summary

- ② Defined what exceptions are and some sample exceptions we encountered along the way.
- ② How to handle exceptions by using the try-catch-finally block.