11

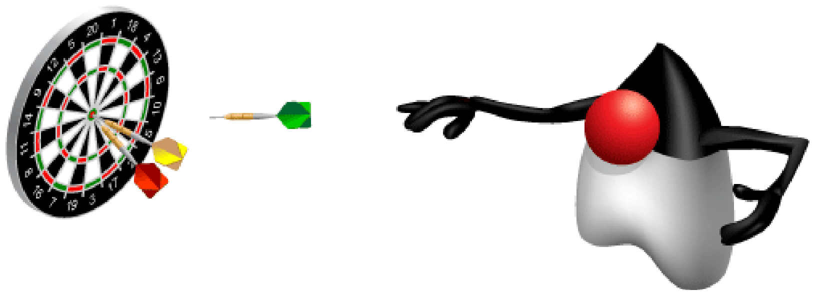
Exceptions and Assertions

ORACLE

Objectives

After completing this lesson, you should be able to:

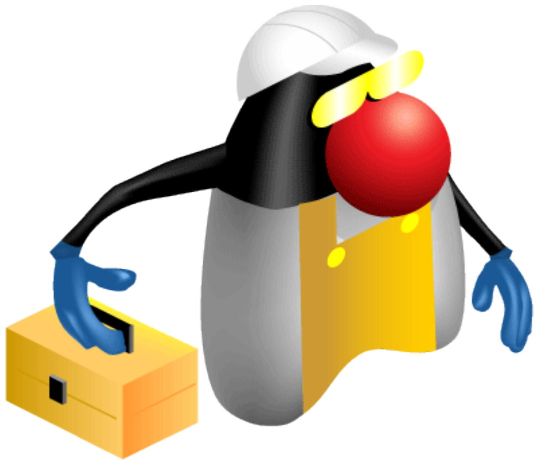
- Define the purpose of Java exceptions
- Use the `try` and `throw` statements
- Use the `catch`, multi-`catch`, and `finally` clauses
- Autoclose resources with a `try`-with-resources statement
- Recognize common exception classes and categories
- Create custom exceptions and auto-closeable resources
- Test invariants by using assertions



Error Handling

Applications sometimes encounter errors while executing. Reliable applications should handle errors as gracefully as possible. Errors:

- Should be an exception and not the expected behavior
- Must be handled to create reliable applications
- Can occur as the result of application bugs
- Can occur because of factors beyond the control of the application
  - Databases becoming unreachable
  - Hard drives failing



Exception Handling in Java

When you are using Java libraries that rely on external resources, the compiler will require you to “handle or declare” the exceptions that might occur.

- Handling an exception means that you must add in a code block to handle the error.
- Declaring an exception means that you declare that a method may fail to execute successfully.

`try-catch` Statement

The `try-catch` statement is used to handle exceptions.

```
try {
    System.out.println("About to open a file");
    InputStream in =
        new FileInputStream("missingfile.txt");
    System.out.println("File open");
} catch (Exception e) {
    System.out.println("Something went wrong!");
}
```

This line is skipped if the previous line failed to open the file.

This line runs only if something went wrong in the `try` block.

Exception Objects

A `catch` clause is passed as a reference to a `java.lang.Exception` object.

The `java.lang.Throwable` class is the parent class for `Exception` and it outlines several methods that you may use.

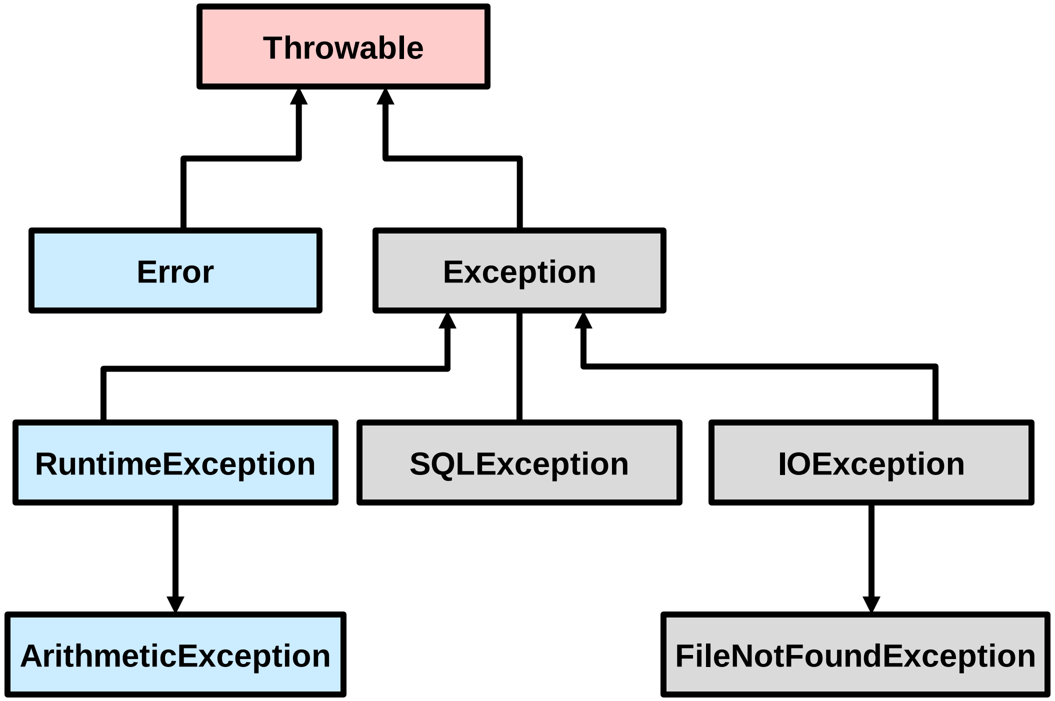
```
try{
    //...
} catch (Exception e) {
    System.out.println(e.getMessage());

}
```

Exception Categories

The `java.lang.Throwable` class forms the basis of the hierarchy of exception classes. There are two main categories of exceptions:

- Checked exceptions, which must be "handled or declared"
- Unchecked exceptions, which are not typically "handled or declared"



Handling Exceptions

You should always catch the most specific type of exception. Multiple catch blocks can be associated with a single try.

```
try {
    System.out.println("About to open a file");
    InputStream in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int data = in.read();
    in.close();
} catch (FileNotFoundException e) {
    System.out.println(e.getClass().getName());
    System.out.println("Quitting");
} catch (IOException e) {
    System.out.println(e.getClass().getName());
    System.out.println("Quitting");
}
```

Order is important. You must catch the most specific exceptions first (that is, child classes before parent classes).

`finally` Clause

```
InputStream in = null;
try {
    System.out.println("About to open a file");
    in = new FileInputStream("missingfile.txt");
    System.out.println("File open");
    int data = in.read();
} catch (IOException e) {
    System.out.println(e.getMessage());
} finally {
    try {
        if(in != null) in.close();
    } catch(IOException e) {
        System.out.println("Failed to close file");
    }
}
```

A `finally` clause runs regardless of whether or not an `Exception` was generated.

You always want to close open resources.

`try`-with-resources Statement

- The `try`-with-resources statement is a `try` statement that declares one or more resources.
- Any class that implements `java.lang.AutoCloseable` can be used as a resource.

```
System.out.println("About to open a file");
try (InputStream in =
      new FileInputStream("missingfile.txt")) {
    System.out.println("File open");
    int data = in.read();
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Catching Multiple Exceptions

Using the multi-`catch` clause, a single catch block can handle more than one type of exception.

```
ShoppingCart cart = null;
try (InputStream is = new FileInputStream(cartFile);
     ObjectInputStream in = new ObjectInputStream(is)) {
     cart = (ShoppingCart)in.readObject();
} catch (ClassNotFoundException | IOException e) {
    System.out.println("Exception deserializing " + cartFile);
    System.out.println(e);
    System.exit(-1);
}
```

Multiple exception types are separated with a vertical bar.

Declaring Exceptions

You may declare that a method throws an exception instead of handling it.

```
public static int readByteFromFile() throws IOException {
    try (InputStream in = new FileInputStream("a.txt")) {
        System.out.println("File open");
        return in.read();
    }
}
```

Notice the lack of `catch` clauses. The `try`-with-resources statement is being used only to close resources.

Handling Declared Exceptions

The exceptions that methods may throw must still be handled. Declaring an exception just makes it someone else's job to handle them.

```
public static void main(String[] args) {
   try {
        int data = readByteFromFile();
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```

Method that declared an exception

Throwing Exceptions

The `throw` statement is used to throw an instance of exception.

```
import java.io.FileNotFoundException;
class DemoThrowsException {
public void readFile(String file) throws
FileNotFoundException {
  boolean found = findFile(file);
  if (!found)
   throw new FileNotFoundException("Missing file");
   else {
          //code to read file
        }
    }
   boolean findFile(String file) {
        //code to return true if file can be located
 } }
```

Custom Exceptions

You can create custom exception classes by extending `Exception` or one of its subclasses.

```
class InvalidPasswordException extends Exception {

 InvalidPasswordException() {
    }
 InvalidPasswordException(String message) {
        super(message);
    }
InvalidPasswordException(String message, Throwable cause) {
        super(message, cause);
}
}
```

Assertions

- Use assertions to document and verify the assumptions and internal logic of a single method:
  - Internal invariants
  - Control flow invariants
  - Class invariants
- Inappropriate uses of assertions
  - Do not use assertions to check the parameters of a public method.
  - Do not use methods that can cause side effects in the assertion check.

Assertion Syntax

There are two forms of the `assert` statement:

- **`assert booleanExpression;`**
  - This statement tests the boolean expression.
  - It does nothing if the boolean expression evaluates to `true`.
  - If the boolean expression evaluates to `false`, this statement throws an `AssertionError`.
- **`assert booleanExpression : expression;`**
  - This form acts just like **`assert booleanExpression;`**.
  - In addition, if the boolean expression evaluates to `false`, the second argument is converted to a string and is used as descriptive text in the `AssertionError` message.

Internal Invariants

```
public class Invariant {

    static void checkNum(int num) {
        int x = num;
        if (x > 0) {
            System.out.print( "number is positive" + x);

        } else if (x == 0) {
            System.out.print("number is zero" + x);
        } else {
            assert (x > 0);
        }
    }
    public static void main(String args[]) {

        checkNum(-4);
    }
}
```

Internal Invariant

Control Flow Invariants

```
switch (suit) {
     case Suit.CLUBS: // ...
       break;
     case Suit.DIAMONDS: // ...
       break;
     case Suit.HEARTS: // ...
       break;
     case Suit.SPADES: // ...
       break;
    default:
    assert false : "Unknown playing card suit";
    break;
}
```

Control Flow Invariant

Class Invariants

```
public class PersonClassInvariant {
    String name;
    String ssn;
    int age;

    private void checkAge()
    {
        assert age >= 18 && age < 150;
    }


    public void changeName(String fname)
    {
        checkAge();
        name=fname;
    }
}
```

Class Invariant

Controlling Runtime Evaluation of Assertions

- If assertion checking is disabled, the code runs as fast as it would if the check were not there.
- Assertion checks are disabled by default. Enable assertions with either of the following commands:

```
java -enableassertions MyProgram
```

```
java -ea MyProgram
```

- Assertion checking can be controlled on class, package, and package hierarchy basis. See: http://download.oracle.com/javase/7/docs/technotes/guides/language/assert.html

Summary

In this lesson, you should have learned how to:

- Define the purpose of Java exceptions
- Use the `try` and `throw` statements
- Use the `catch`, multi-`catch`, and `finally` clauses
- Autoclose resources with a `try`-with-resources statement
- Recognize common exception classes and categories
- Create custom exceptions and auto-closeable resources
- Test invariants by using assertions

Practice 11-1 Overview: Catching Exceptions

This practice covers the following topics:

- Adding `try-catch` statements to a class
- Handling exceptions

Practice 11-2 Overview: Extending `Exception` and Using `throw` and `throws`

This practice covers the following topics:

- Extending the `Exception` class
- Throwing exceptions using `throw` and `throws`

Quiz

A `NullPointerException` must be caught by using a `try-catch` statement.

a. True

b. False

Quiz

Which of the following types are all checked exceptions (`instanceof`)?

a. `Error`

b. `Throwable`

c. `RuntimeException`

d. `Exception`

Quiz

Which keyword would you use to add a clause to a method stating that the method might produce an exception?

a. `throw`

b. `thrown`

c. `throws`

d. `assert`

Quiz

Assertions should be used to perform user-input validation.

a. True

b. False