3

Exception Handling and Assertions

Objectives

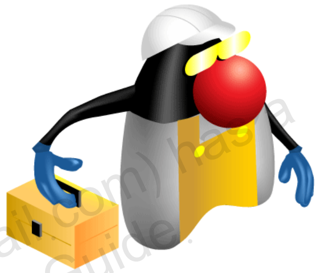
After completing this lesson, you should be able to:

- 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



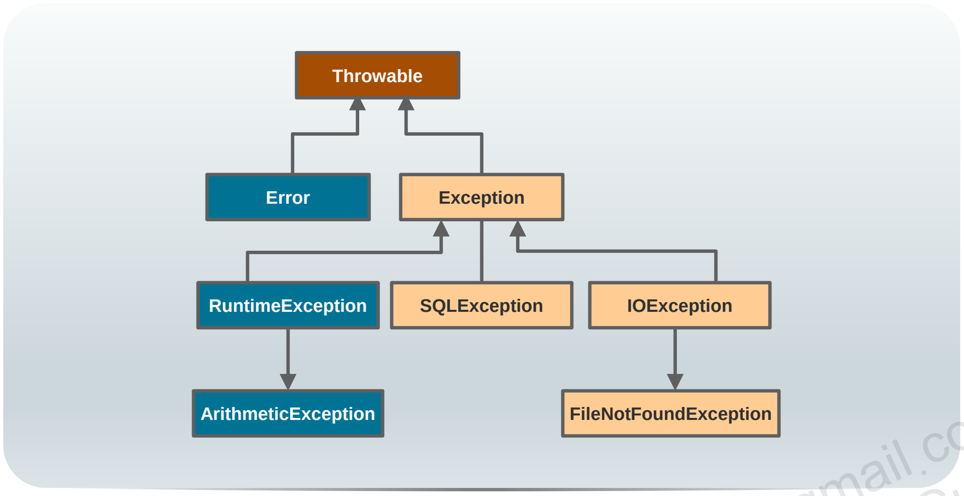
Returning a Failure Result

Some programming languages use the return value of a method to indicate whether or not a method completed successfully. For instance, in the C example `int x = printf("hi");`, a negative value for `x` would indicate a failure. Many of C's standard library functions return a negative value upon failure. The problem is that this example could also be written as `printf("hi");` where the return value is ignored. In Java, you also have the same concern; any return value can be ignored.

When a method you write in the Java language fails to execute successfully, consider using the exception-generating and handling features available in the language instead of using return values.

Exception Types

- An exception is an instance of a class derived directly or indirectly from the `java.lang.Throwable` class.
- Two predefined Java classes are derived from `Throwable`- `Error` and `Exception`.



Dealing with Exceptions

When an `Exception` object is generated and passed to a `catch` clause, it is instantiated from a class that represents the specific type of problem that occurred. These exception-related classes can be divided into two categories: checked and unchecked.

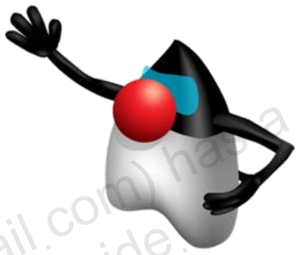
Unchecked Exceptions

`java.lang.RuntimeException` and `java.lang.Error` and their subclasses are categorized as unchecked exceptions. These types of exceptions should not normally occur during the execution of your application. You can use a `try-catch` statement to help discover the source of these exceptions. However, when an application is ready for production use, there should be a little code remaining that deals with `RuntimeException` and its subclasses. The Error subclasses represent errors that are beyond your ability to correct, such as the JVM running out of memory. Common `RuntimeException`s that you may have to troubleshoot include:

- `ArrayIndexOutOfBoundsException`: Accessing an array element that does not exist
- `NullPointerException`: Using a reference that does not point to an object
- `ArithmeticException`: Dividing by zero

Exception Types

- The exceptions derived from the `Error` class represent problems with the JVM and normally can't be recovered, and there is little that a programmer will do with these exceptions.
- The subclasses of `Exception` class support two types of exceptions:
  - Checked: These are exceptions that you need to handle within the code
  - Unchecked: These are exceptions that you don’t need to handle within the code



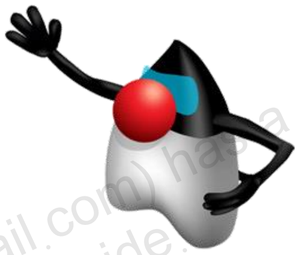
Checked exceptions include all exceptions derived from the `Exception` class and are not derived from the `RuntimeException` class. These must be handled in code or the code will not compile cleanly, resulting in compile-time errors.

Unchecked exceptions are all other exceptions. They include exceptions, such as division by zero and array subscripting errors. These do not have to be caught, but like the `Error` exceptions, if they are not caught, the program will terminate.

Exception Handling Techniques in Java

There are two general techniques you can use in Java:

1. Handling an exception: You must add in a code block to handle the error
   a. `try` block
   b. `try-with-resources` block
2. Declaring an exception: You declare that a method may fail to execute successfully.



The Handle or Declare Rule

To use many libraries, you require knowledge of exception handling. They include:

- File IO (NIO: `java.nio`)
- Database access (JDBC: `java.sql`)

Handling an exception means that you use a `try-catch` statement to transfer control to an exception-handling block when an exception occurs. Declaring an exception means to add a `throws` clause to a method declaration, indicating that the method may fail to execute in a specific way. In other words, handling means it is your problem to deal with and declaring means that it is someone else's problem to deal with.

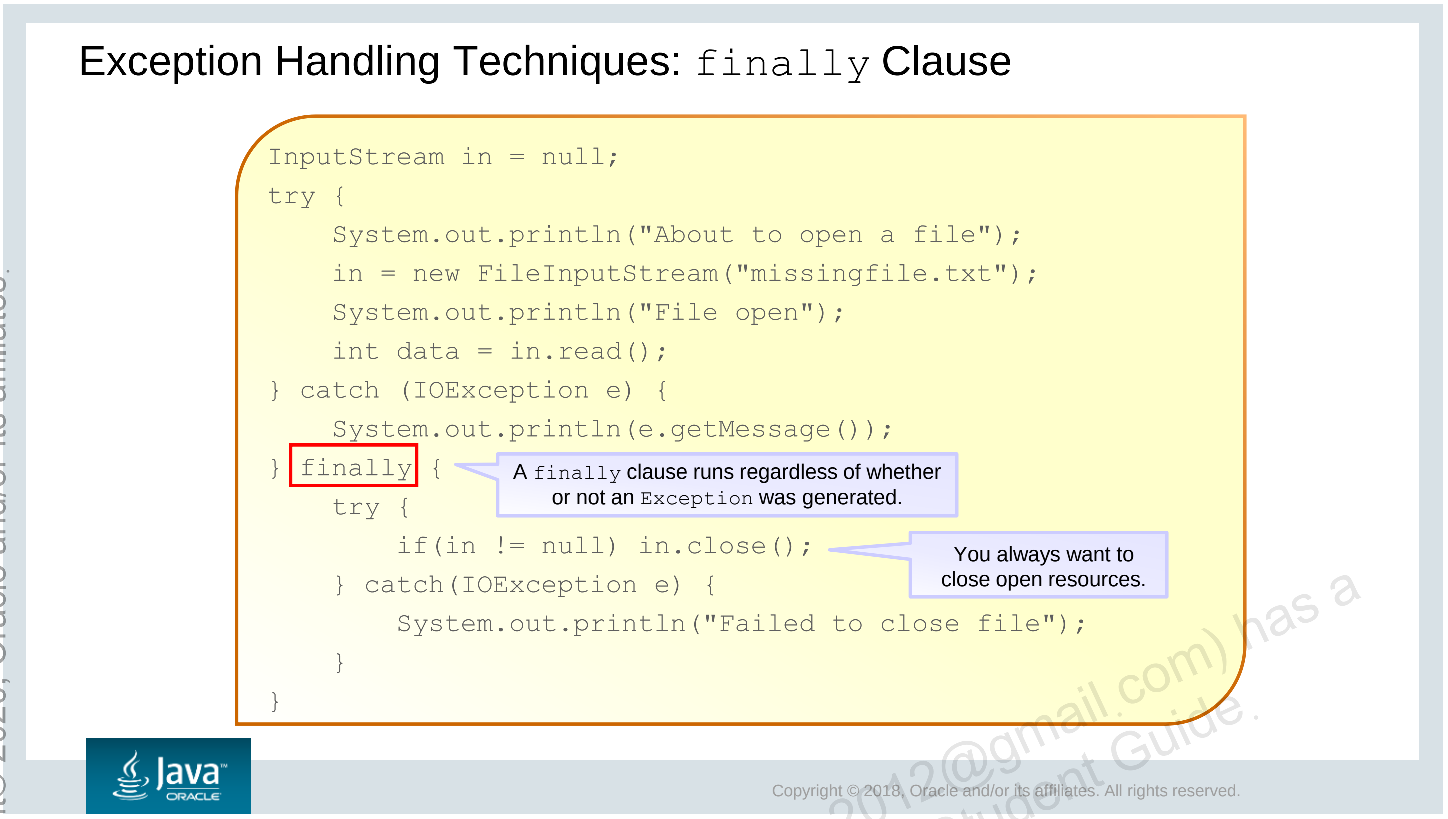
Exception Handling Techniques: `try` Block

- 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).

The traditional technique to handle exceptions uses a combination of a `try`, `catch`, and `finally` blocks. A `try` block is used to surround code that might throw exceptions and is followed by zero or more catch blocks and then, optionally, by a single `finally` block. The `catch` blocks are added after a `try` block to "catch" exceptions. The statements in the `catch` block provide blocks of code to "handle" the error. A `finally` clause can optionally be used after the catch blocks. It is guaranteed to execute even if code within a `try` or a `catch` block throws or does not throw an exception.



Exception Handling Techniques: `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.

Closing Resources

When you open resources, such as files or database connections, you should always close them when they are no longer needed. Attempting to close resources inside the `try` block can be problematic because you can end up skipping the close operation. A `finally` block always runs regardless of whether or not an error occurred during the execution of the `try` block. If control jumps to a `catch` block, the `finally` block executes after the `catch` block.

Sometimes the operation that you want to perform in your `finally` block may itself cause an `Exception` to be generated. In that case, you may be required to nest a `try-catch` inside of a `finally` block. You may also nest a `try-catch` inside of `try` and `catch` blocks.

Exception Handling Techniques: `try`-with-resources

- The `try`-with-resources block:
  - Declares one or more resources. All the resources opened within the block are automatically closed upon exit from the block.
  - No need to close the resources explicitly in the code.
  - Any class that implements `java.lang.AutoCloseable` can be used as a resource.

```
try (BufferedReader reader = Files.newBufferedReader((Path) new FileReader("src.txt"));
     BufferedWriter writer = Files.newBufferedWriter((Path) new FileWriter("dest.txt"));
    {
     String input;
     while ((input = reader.readLine()) != null) {
     writer.write(input);
     writer.newLine();
    }
}
catch(URISyntaxException ex) {..}
catch(IOException ex){..}
```

Two resources declared within the try-with-resources block

Closeable Resources

The use of the previous technique can be cumbersome when multiple resources are opened and a failure occurs. It can result in multiple try-catch blocks that become hard to follow.

In Java 7, the try-with-resources block was introduced to address this situation. The `try`-with-resources statement can eliminate the need for a lengthy `finally` block. Resources opened by using the `try`-with-resources statement are always closed. If a resource should be autoclosed, its reference must be declared within the `try` statement's parentheses.

Resources declared with a try-with-resources block must be separated by semicolons, otherwise a compile-time error will be generated.

Exception Handling Techniques: `try`-with-resources Improvements

- Concise `try`-with-resources statements in JDK 9:

  If you already have a resource as a final or effectively final variable, you can use that variable in the `try`-with-resources statement without declaring a new variable in the `try`-with-resources statement.

- For example, given resource declarations like:

```
// A final resource
final Resource resource1 = new Resource("resource1");
// An effectively final resource
Resource resource2 = new Resource("resource2");
```

- The old way to write the code to manage these resources would be something like:

```
// Original try-with-resources statement from JDK 7 or 8
       try (Resource r1 = resource1;
            Resource r2 = resource2) {
       // Use of resource1 and resource 2 through r1 and r2.
    }
```

Resources are declared within the try-with-resources statement

Exception Handling Techniques: `try`-with-resources Improvements

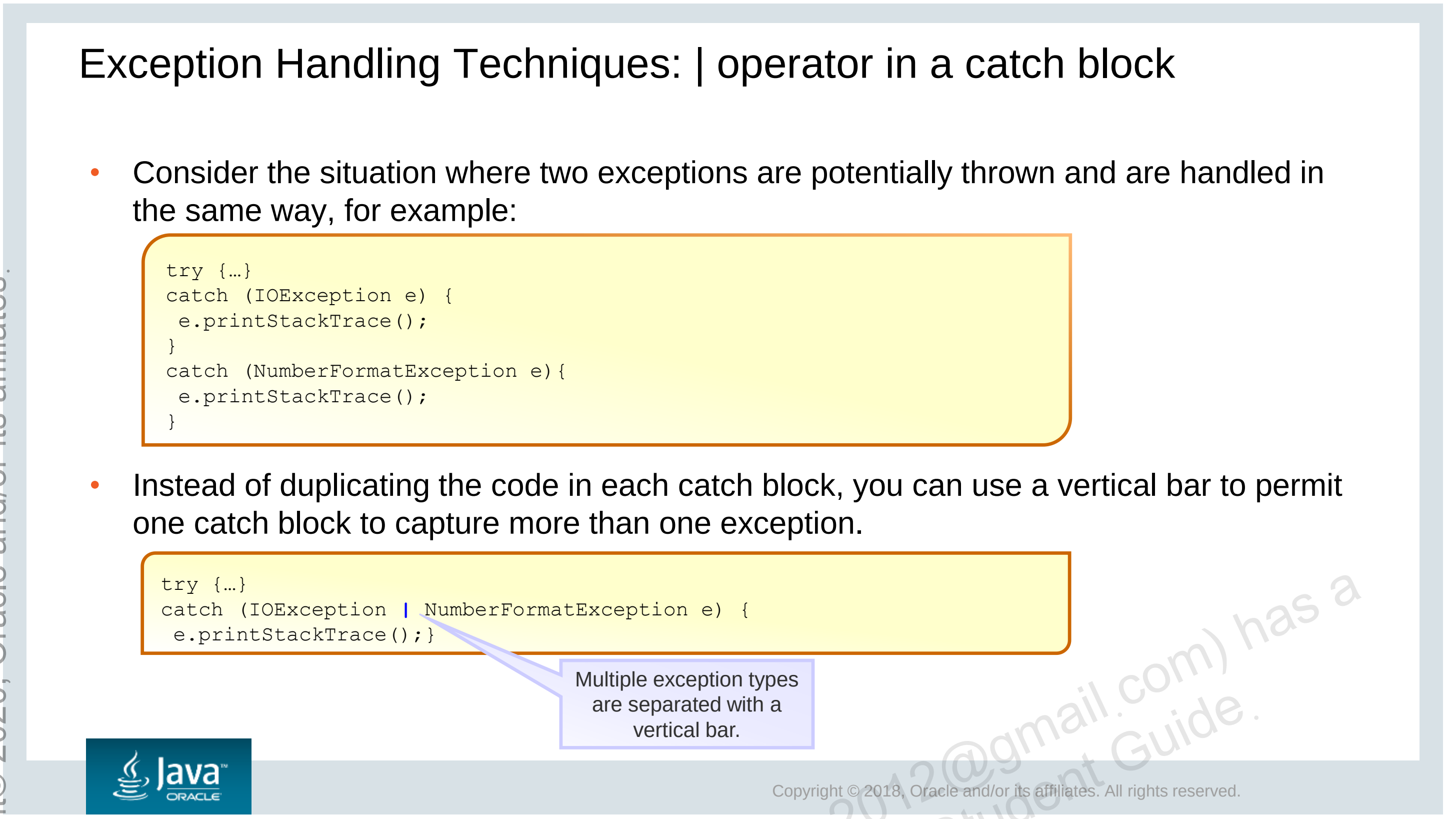
- `try`-with-resources statement in JDK 9 and later versions.

```
 BufferedReader reader = Files.newBufferedReader((Path) new FileReader("src.txt"));
 BufferedWriter writer = Files.newBufferedWriter((Path) new FileWriter("dest.txt"));

 try (reader ; writer) {
    String input;
    while ((input = reader.readLine()) != null) {
    writer.write(input);
    writer.newLine();
  }
}

catch(URISyntaxException ex) {
}
catch(IOException ex){
}
```

Resources declared outside as final or effectively final are used within the try-with-resources statement



Exception Handling Techniques: | operator in a catch block

- Consider the situation where two exceptions are potentially thrown and are handled in the same way, for example:

```
try {…}
catch (IOException e) {
 e.printStackTrace();
}
catch (NumberFormatException e){
 e.printStackTrace();
}
```

- Instead of duplicating the code in each catch block, you can use a vertical bar to permit one catch block to capture more than one exception.

```
try {…}
catch (IOException | NumberFormatException e) {
 e.printStackTrace();}
```

Multiple exception types are separated with a vertical bar.

The Benefits of Multi-`catch`

Sometimes you want to perform the same action regardless of the exception being generated. The new multi-`catch` clause reduces the amount of code you must write by eliminating the need for multiple `catch` clauses with the same behaviors.

Exception Handling Techniques: Declaring Exceptions

- You may declare that a method throws an exception instead of handling it:
  - Is used when the current method is not the appropriate place to handle the exception.
  - Allows the exception to be propagated higher into the sequence of method calls.
- For example:

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

`readByteFromFile()` may encounter some condition where it may throw `IOException`. Instead of dealing with the exception in `readByteFromFile()`, the `throws` keyword in the method definition results in the exception being passed to the code that called this method.

Using the `throws` clause, a method may declare that it throws one or more exceptions during execution.

If an exception is generated while executing the method, the method stops executing and the exception is thrown to the caller. Uncaught exceptions are propagated to the next higher context until they are caught or they are thrown from main, where an error message and stack trace will be printed.

Overridden methods may declare the same exceptions, fewer exceptions, or more specific exceptions, but not additional or more generic exceptions.

A method may declare multiple exceptions with a comma-separated list.

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

Technically, you do not need to declare `FileNotFoundException` because it is a subclass of `IOException`, but it is a good practice to do so.

The method will skip all of the remaining lines of code in the method and immediately return to the caller.

Creating 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);
}
}
```

Custom exceptions are never thrown by standard Java class libraries. To take advantage of a custom exception class, you must throw it yourself. For example:

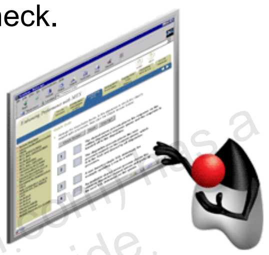
```
throw new InvalidPasswordException();
```

A custom exception class may override methods or add new functionality. The rules of inheritance are the same, even though the parent class type is an exception.

Because exceptions capture information about a problem that has occurred, you may need to add fields and methods depending on the type of information that needs to be captured. If a string can capture all the necessary information, you can use the `getMessage()` method that all `Exception` classes inherit from `Throwable`. Any `Exception` constructor that receives a string will store it to be returned by `getMessage()`.

Assertions

- You can 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
  - Don't use assertions to check the parameters of a public method.
  - Don't use methods that can cause side effects in the assertion check.



Why Use Assertions

You can use assertions to add code to your applications, which would ensure that the application is executing as expected. Using assertions, you test for failure of various conditions; if they do, you terminate the application and display debugging-related information. Assertions should not be used if the checks to be performed should always be executed because assertion checking may be disabled.

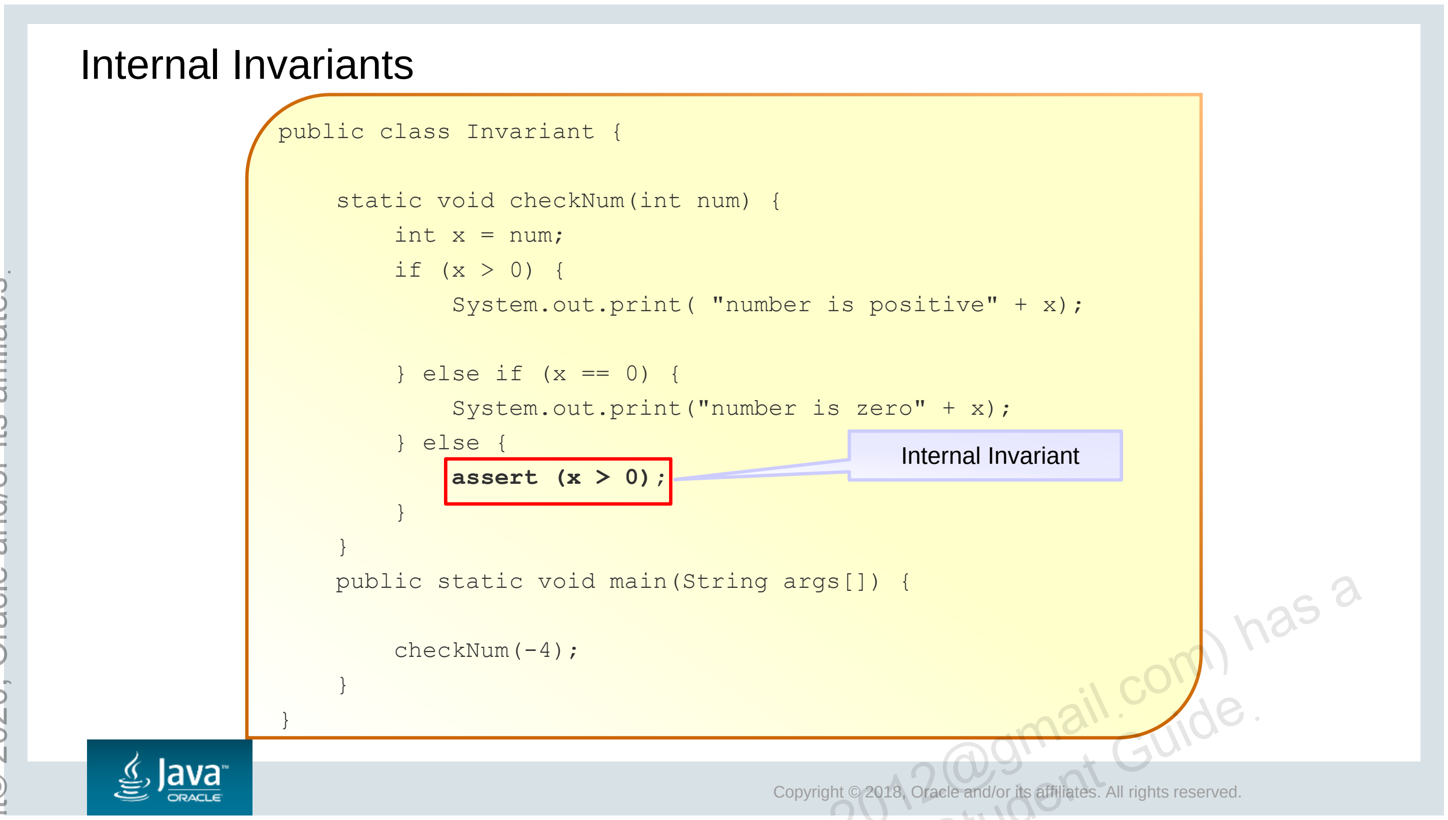
Assertion Syntax

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.

The `assert` Statement

`AssertionError` is a subclass of `Error` and, therefore, falls in the category of unchecked exceptions.



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

An invariant is something that should always be `true`. An internal invariant is a “fact” that you believe to be true at a certain point in the program.

In the code snippet in the slide, the `assert` statement determines whether the number is less than zero and, if so, it throws an `AssertionError`.

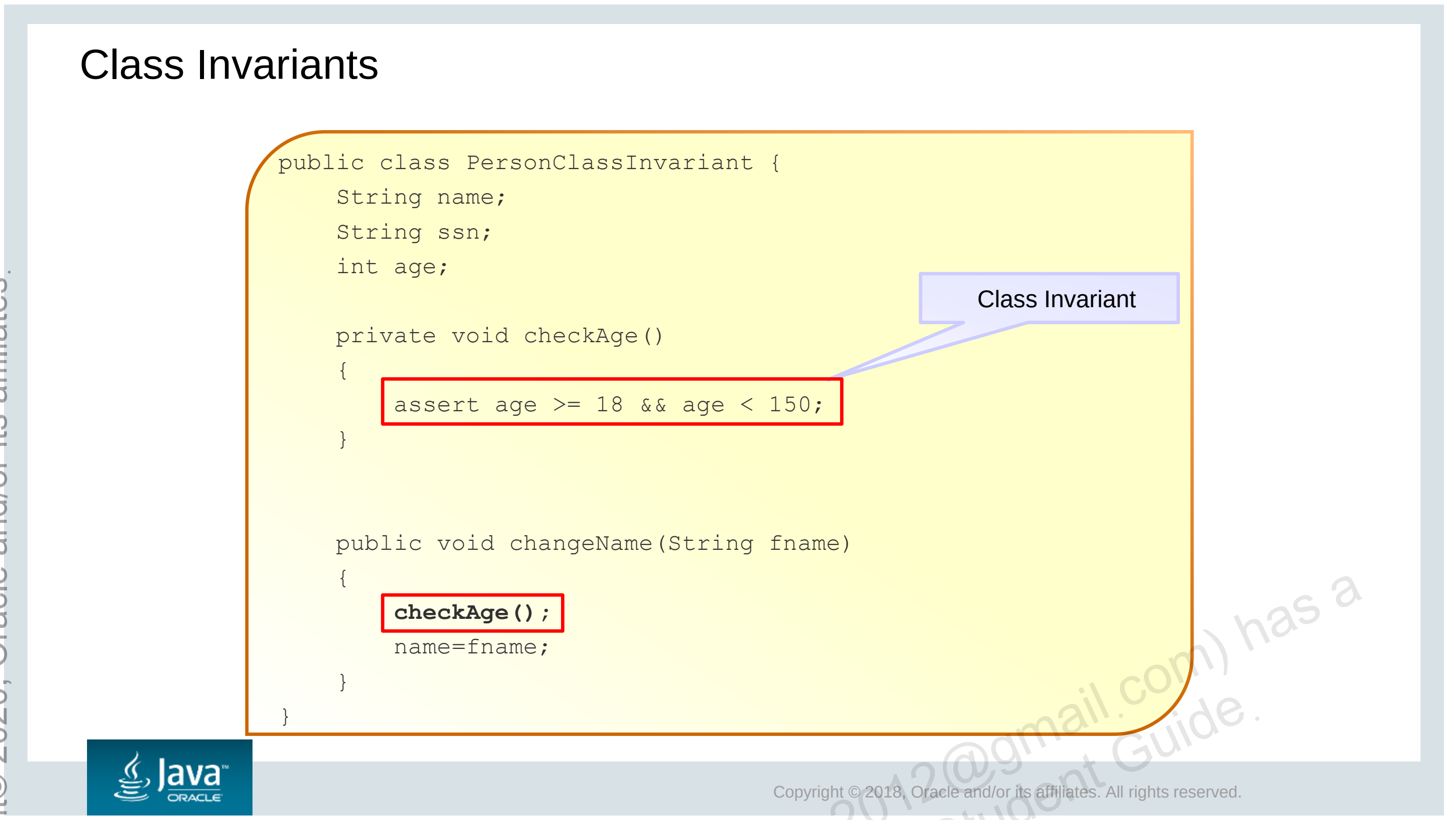
Control Flow Invariants

```
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

Assertion can be used in a `switch` statement with no `default` case, when the programmer is sure that one of the `switch` cases will be executed every time he or she can omit the `default` case as in the example in the slide.

To test this assumption, the programmer can add an `assert` statement in the `default` case. By using the assert statement, you can check the assumption about the applications flow of control. Assertion can be placed at any location where the control will not be reached.



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

A class invariant is one that an object must satisfy in order to be a valid member of a class.

Controlling Runtime Evaluation of Assertions

- Assertion checks are disabled by default. You can enable assertions with either of the following commands:

```
java -ea MyProgram
```

```
java -enableassertions MyProgram
```

Enabling Assertions in Netbeans

1. In Netbeans, right-click the project and select **Properties**.
2. In the window that appears, select **Run**.
3. Enter **`-enableassertions`** in VM Options.

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 3: Overview

This practice covers extending exception and using `throw` and `throws` clause.

In this practice, you write code to deal with both checked and unchecked exceptions.

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