Chapter 16

How to work with exceptions

Objectives

Applied

1. Given a method that throws one or more exceptions, code a method that calls that method and catches or throws the exceptions.
2. Code a method that throws an exception to the calling method.
3. Use the methods of the Throwable class to get information about an exception.
4. Code a class that defines a custom exception, and then use that exception in an application.
5. Given Java code that uses any of the language elements presented in this chapter, explain what each statement does.

Objectives (cont.)

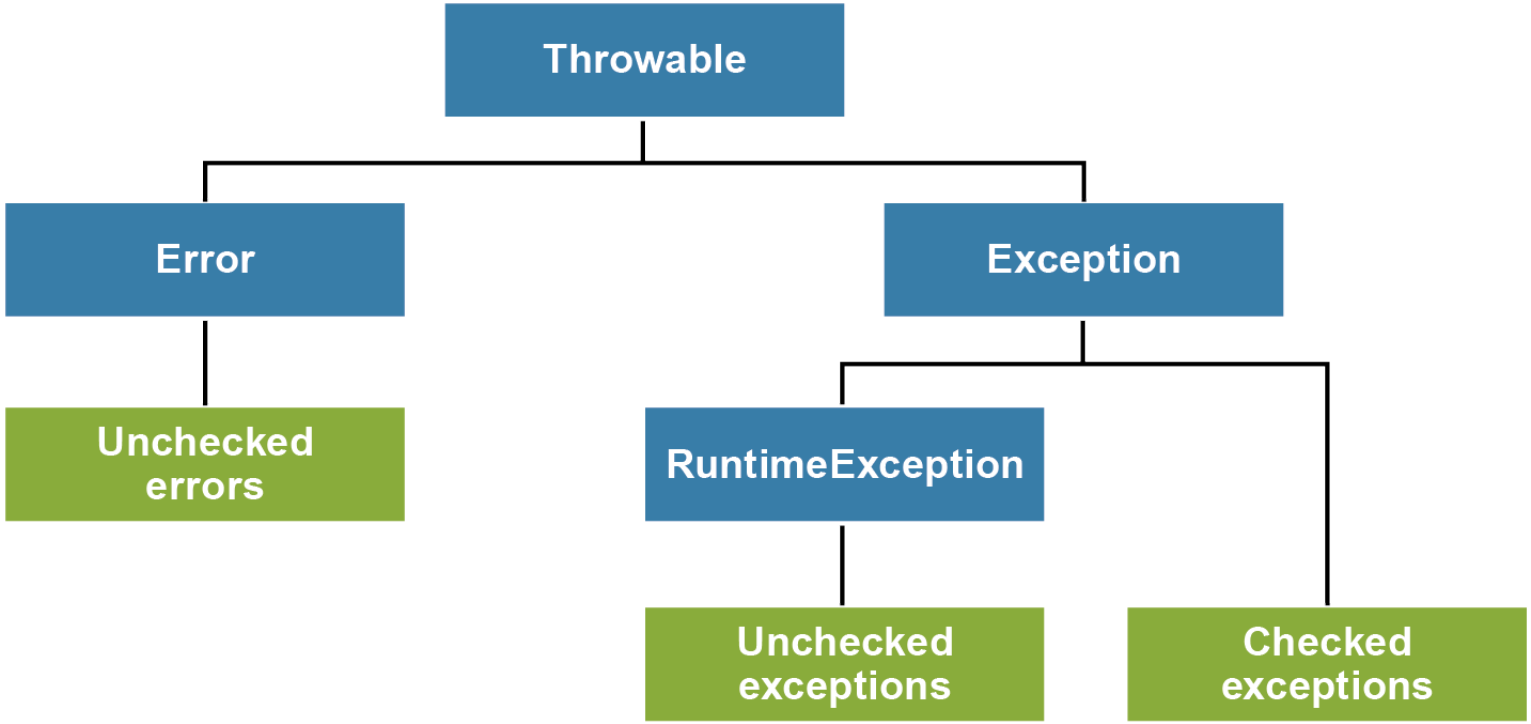
Knowledge

1. Describe the classes in the Throwable hierarchy.
2. Describe the difference between checked and unchecked exceptions and explain when you need to catch each.
3. Explain how Java propagates exceptions and how it uses the stack trace to determine what exception handler to use when an exception occurs.
4. Describe the order in which you code the catch clauses in a try statement.
5. Explain when the code in the finally clause of a try statement is executed and what that code typically does.
6. Explain what it means to swallow an exception and why that's almost never an acceptable practice.
7. Explain how a try-with-resources statement works.

Objectives (cont.)

8. Explain how a multi-catch block works.
9. Describe three situations where you might want to throw an exception from a method.
10. Describe two situations where you might create a custom exception class.
11. Explain what exception chaining is and when you might use it.

The Throwable hierarchy



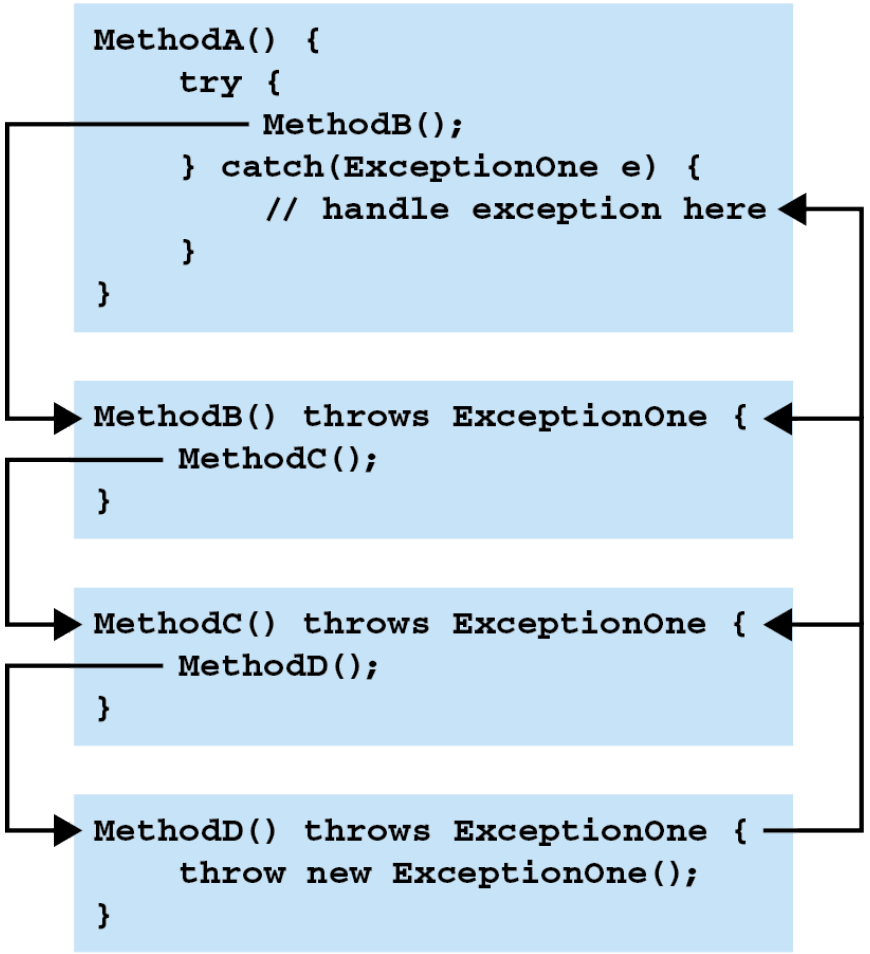
Common checked exceptions

```
ClassNotFoundException
IOException
    EOFException
    FileNotFoundException
```

Common unchecked exceptions

```
ArithmeticException
IllegalArgumentException
    NumberFormatException
IndexOutOfBoundsException
    ArrayIndexOutOfBoundsException
    StringIndexOutOfBoundsException
NoSuchElementException
    InputMismatchException
NullPointerException
```

How Java propagates exceptions



Two ways to handle checked exceptions

- Throw the exception to the calling method
- Catch the exception and handle it

The syntax of the try statement

```
try { statements }
[catch (MostSpecificExceptionType e) { statements }] ...
[catch (LeastSpecificExceptionType e) { statements }]
[finally { statements }]
```

A method that catches two types of exceptions and uses a finally clause

```
public static String readFirstLine(String path) {
    BufferedReader in = null;
    try {
        in = new BufferedReader(
             new FileReader(path));
                    // may throw FileNotFoundException
        String line = in.readLine();
                    // may throw IOException
        return line;
    } catch (FileNotFoundException e) {
        System.out.println("File not found.");
        return null;
    } catch(IOException e) {
        System.out.println("I/O error occurred.");
        return null;
```

A method that catches two types of exceptions and uses a finally clause (cont.)

```
        } finally {
            try {
                if (in != null) {
                    in.close();     // may throw IOException
                }
            } catch (IOException e) {
                System.out.println("Unable to close file.");
            }
        }
    }
```

The syntax of the try-with-resources statement

```
try (statement[;statement] ...) { statements }
[catch (MostSpecificExceptionType e) { statements }] ...
[catch (LeastSpecificExceptionType e) { statements }]
```

A method that catches two types of exceptions and automatically closes the specified resource

```
public static String readFirstLine(String path) {
    try (BufferedReader in = new BufferedReader(
                             new FileReader(path))) {
        String line = in.readLine();
        return line;
    } catch (FileNotFoundException e) {
        System.out.println("File not found.");
        return null;
    } catch (IOException e) {
        System.out.println("I/O error occurred.");
        return null;
    }
}
```

Four methods available from all exceptions

```
getMessage()
toString()
printStackTrace()
printStackTrace(outputStream)
```

How to print exception data to the error output stream

```
catch(IOException e) {
    System.err.println(e.getMessage() + "\n");
    System.err.println(e.toString() + "\n");
    e.printStackTrace();
    return null;
}
```

Resulting output for a FileNotFoundException

```
c:\murach\java\produx.txt (The system cannot find the file
specified)

java.io.FileNotFoundException: c:\murach\java\produx.txt (The
system cannot find the file specified)

java.io.FileNotFoundException: c:\murach\java\produx.txt (The
system cannot find the file specified)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:131)
  at java.io.FileInputStream.<init>(FileInputStream.java:87)
  at java.io.FileReader.<init>(FileReader.java:58)
  at ProductApp.readFirstLine(ProductApp.java:70)
  at ProductApp.main(ProductApp.java:10)
```

How to print exception data to the standard output stream

```
catch(IOException e) {
    System.out.println(e.getMessage() + "\n");
    System.out.println(e.toString() + "\n");
    e.printStackTrace(System.out);
    return null;
}
```

The syntax of the multi-catch block

```
catch (ExceptionType | ExceptionType
    [| ExceptionType]... e) { statements }
```

A method that does not use a multi-catch block

```
public static String readFirstLine(String path) {
    try (BufferedReader in = new BufferedReader(
                             new FileReader(path))) {
        String line = in.readLine();
                             // may throw IOException
        return line;
    } catch (FileNotFoundException e) {
        System.err.println(e.toString());
        return null;
    } catch (EOFException e) {
        System.err.println(e.toString());
        return null;
    } catch(IOException e) {
        e.printStackTrace();
        return null;
    }
}
```

A method that uses a multi-catch block

```
public static String readFirstLine(String path) {
    try (BufferedReader in = new BufferedReader(
                             new FileReader(path))) {
        String line = in.readLine();
                             // may throw IOException
        return line;
    } catch (FileNotFoundException | EOFException e) {
        System.err.println(e.toString());
        return null;
    } catch(IOException e) {
        e.printStackTrace();
        return null;
    }
}
```

The syntax for declaring a method that throws exceptions

```
modifiers returnType methodName([parameterList])
    throws exceptionList {}
```

A method that throws an IOException

```
public static String readFirstLine(String path)
        throws IOException {
    BufferedReader in = new BufferedReader(
                        new FileReader(path));
    String line = in.readLine();
    return line;
}
```

A method that throws two exceptions

```
public static String readFirstLine(String path)
        throws FileNotFoundException, IOException {
    BufferedReader in = new BufferedReader(
                        new FileReader(path));
    String line = in.readLine();
    return line;
}
```

Compiler error if you don't catch or throw a checked exception

```
error: unreported exception IOException; must be
caught or declared to be thrown
```

The syntax of the throw statement

```
throw throwableObject;
```

Common constructors of the Throwable class

```
Throwable()
Throwable(message)
```

A method that throws an unchecked exception

```
public double calculateFutureValue(double monthlyPayment,
        double monthlyInterestRate, int months) {
    if (monthlyPayment <= 0) {
        throw new IllegalArgumentException(
            "Monthly payment must be > 0");
    }
    if (monthlyInterestRate <= 0) {
        throw new IllegalArgumentException(
            "Interest rate must be > 0");
    }
    if (months <= 0) {
        throw new IllegalArgumentException(
            "Months must be > 0");
    }

    // code to calculate and return future value
}
```

Code that throws an IOException for testing purposes

```
try {
    // code that reads the first line of a file

    if (true) {
        throw new IOException("I/O exception test");
    }
    return firstLine;
} catch (IOException e) {
   // code to handle IOException goes here
}
```

Code that rethrows an exception

```
try {
    // code that throws IOException goes here
} catch (IOException e) {
    System.out.println(
        "IOException thrown in readFirstLine() method.");
    throw e;
}
```

Code for the DAOException class

```
public class DAOException extends Exception {
    public DAOException() {
    }

    public DAOException(String message) {
        super(message);
    }
}
```

A method that throws the DAOException

```
public static Product getProduct(String productCode)
        throws DAOException {
    try {
        Product p = readProduct(productCode);
                                // may throw IOException
        return p;
    } catch (IOException e) {
        throw new DAOException(
            "An error occurred while reading " +
            "the product.");
    }
}
```

Code that catches the DAOException

```
try {
    Product p = getProduct("1234");
} catch (DAOException e) {
    System.out.println(e.getMessage());
}
```

Resulting output

```
An error occurred while reading the product.
```

When to define your own exceptions

- When a method requires an exception that isn't provided by any of Java's exception types.
- When using a built-in Java exception would inappropriately expose details of a method's operation.

A constructor of the Throwable class for exception chaining

```
Throwable(cause)
```

A custom exception class that uses exception chaining

```
public class DAOException extends Exception {
    public DAOException() {}

    public DAOException(Exception cause) {
        super(cause);
    }
}
```

Code that throws a DAOException with chaining

```
catch (IOException e) {
    throw new DAOException(e);
}
```

Code that catches a DAOException

```
catch (DAOException e) {
    System.err.println(e);
}
```

Resulting output

```
DAOException: java.io.FileNotFoundException:
c:\murach\produx.txt (The system cannot find the
file specified)
```

An interface that uses a custom exception

```
package murach.db;

import java.util.List;

public interface DAO<T> {
    T get(String code) throws DAOException;
    List<T> getAll() throws DAOException;
    boolean add(T t) throws DAOException;
    boolean update(T t) throws DAOException;
    boolean delete(T t) throws DAOException;
}
```

A class that uses custom exceptions

```
package murach.db;

import java.io.*;
import java.nio.file.*;
import java.util.*;

import murach.business.Product;

public final class ProductTextFile
        implements DAO<Product> {
    private List<Product> products = null;
    private Path productsPath = null;
    private File productsFile = null;
    private final String FIELD_SEP = "\t";
```

A class that uses custom exceptions (cont.)

```
public ProductTextFile() throws DAOException {
    productsPath = Paths.get("products.txt");
    productsFile = productsPath.toFile();
    products = this.getAll();
}

@Override
public List<Product> getAll() throws DAOException {
    // if the products file has already been read,
    // don't read it again
    if (products != null) {
        return products;
    }

    products = new ArrayList<>();
    try (BufferedReader in =
        new BufferedReader(
        new FileReader(productsFile))) {
```

A class that uses custom exceptions (cont.)

```
            // read products from file into array list
            String line = in.readLine();
            while (line != null) {
                String[] columns = line.split(FIELD_SEP);
                String code = columns[0];
                String description = columns[1];
                String price = columns[2];
                Product p = new Product(
                    code, description,
                    Double.parseDouble(price));
                products.add(p);

                line = in.readLine();
            }
        } catch (IOException e) {
            throw new DAOException(e);
        }
        return products;
    }
```

A class that uses custom exceptions (cont.)

```
@Override
public Product get(String code) throws DAOException {
    for (Product p : products) {
        if (p.getCode().equals(code)) {
            return p;
        }
    }
    return null;
}

@Override
public boolean add(Product p) throws DAOException {
    products.add(p);
    return this.saveAll();
}
```

A class that uses custom exceptions (cont.)

```
@Override
public boolean delete(Product p)
        throws DAOException {
    products.remove(p);
    return this.saveAll();
}

@Override
public boolean update(Product newProduct)
        throws DAOException {
    // get the old product and remove it
    Product oldProduct =
        this.get(newProduct.getCode());
    int i = products.indexOf(oldProduct);
    products.remove(i);
    // add the updated product
    products.add(i, newProduct);

    return this.saveAll();
}
```

A class that uses custom exceptions (cont.)

```
    private boolean saveAll() throws DAOException {
        try (PrintWriter out =
                new PrintWriter(
                new BufferedWriter(
                new FileWriter(productsFile)))) {

                // write all products in the array list
                // to the file
                for (Product p : products) {
                    out.print(p.getCode() + FIELD_SEP);
                    out.print(p.getDescription() +
                        FIELD_SEP);
                    out.println(p.getPrice());
                }
                return true;
        } catch (IOException e) {
            throw new DAOException(e);
        }
    }
}
```