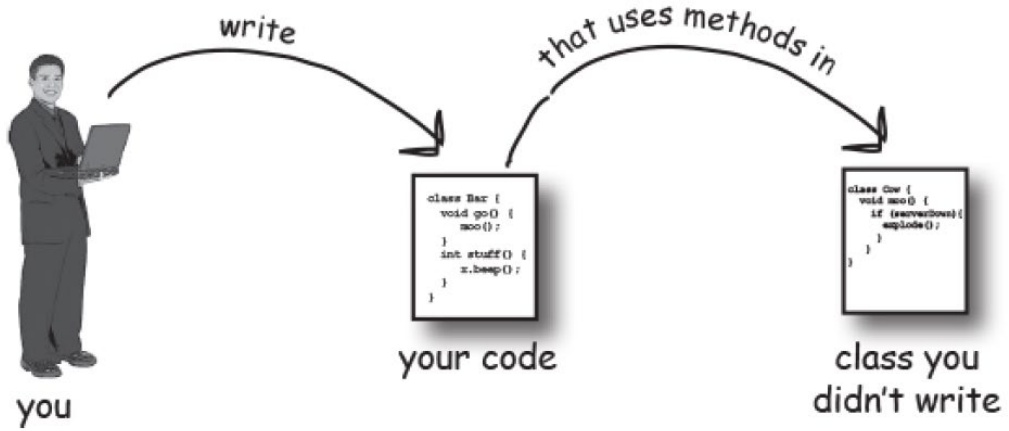
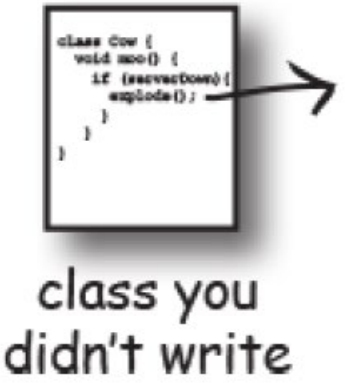
Exceptions

Lecture, 2023

University American College Skopje
School of Computer Science and Information Technology
Course: Programming Languages
Prepared by: Dejan Mitov

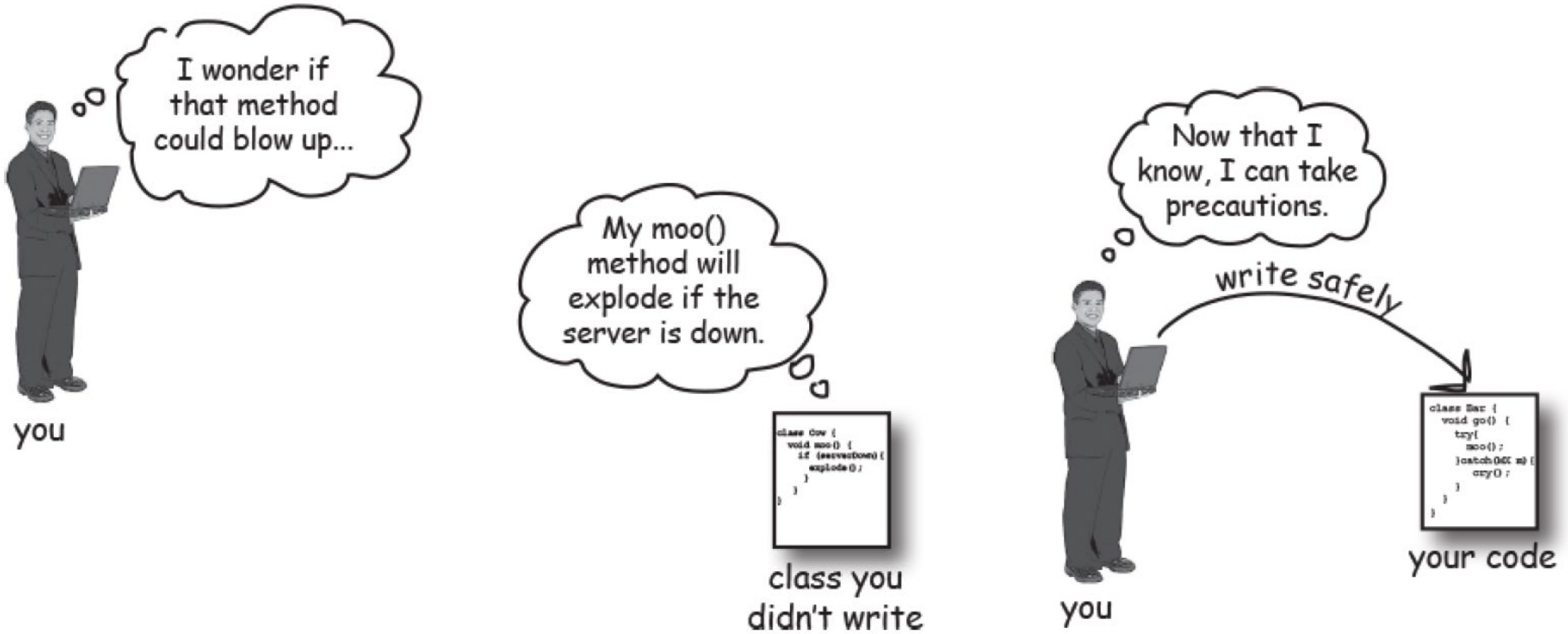
What happens when a method you want to call (probably in a class you didn't write) is risky?






```
void moo() {
    if (serverDown) {
     explode();
    }
}
```

You need to know that the method you're calling is risky.



Methods in Java use exceptions to tell the calling code, "Something Bad Happened. I failed."

- Java's exception-handling mechanism is a clean, well-lighted way to handle "exceptional situations" that pop up at runtime;
- It lets you put all your error-handling code in one easy-to-read place. It's based on the method you're calling telling you it's risky (i.e., that the method might generate an exception), so that you can write code to deal with that possibility.
- If you know you might get an exception when you call a particular method, you can be prepared for possibly even recover from the problem that caused the exception.

The compiler needs to know that YOU know you're calling a risky method

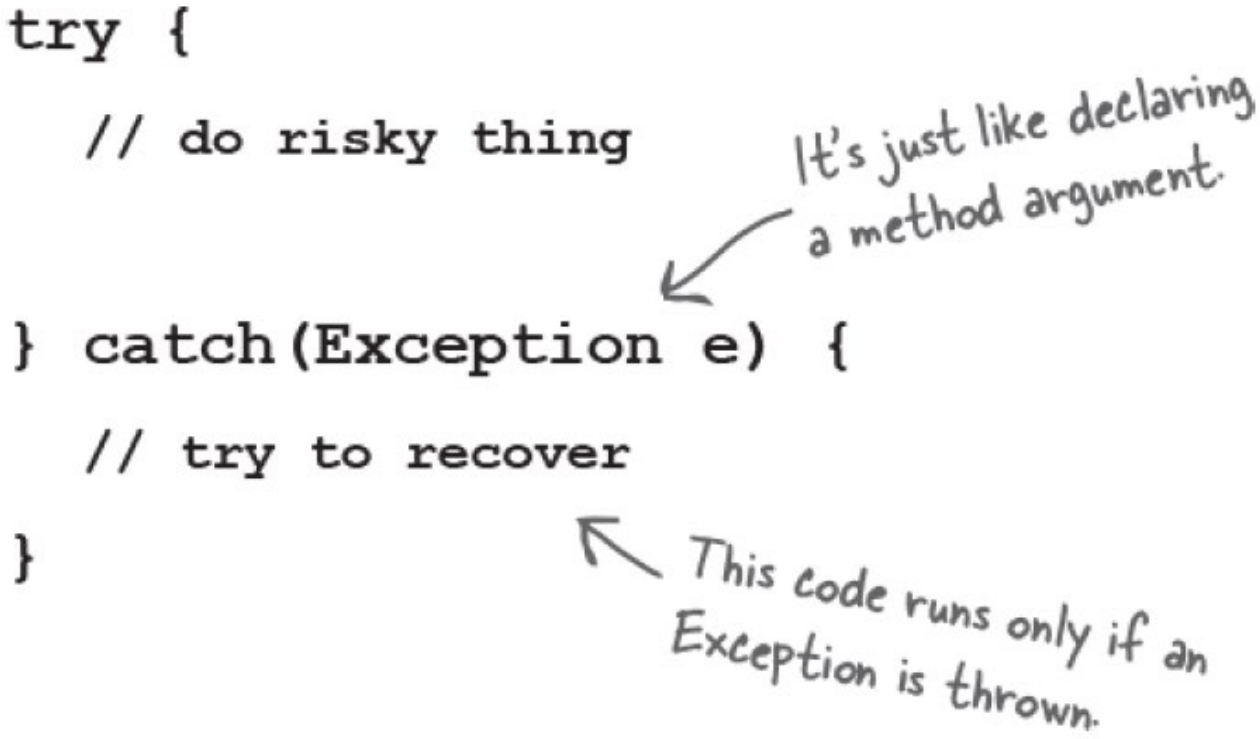
```
public void play() {
  try {
    Sequencer sequencer = MidiSystem.getSequencer();
    System.out.println("Successfully got a sequencer");
  } catch(MidiUnavailableException e) {
    System.out.println("Bummer");
  }
}

public static void main(String[] args) {
  MusicTest1 mt = new MusicTest1();
  mt.play();
}
}
```

Put the risky thing in a "try" block. It's the "risky" getSequencer method that might throw an exception.

Make a "catch" block for what to do if the exceptional situation happens—in other words, a MidiUnavailableException is thrown by the call to getSequencer().

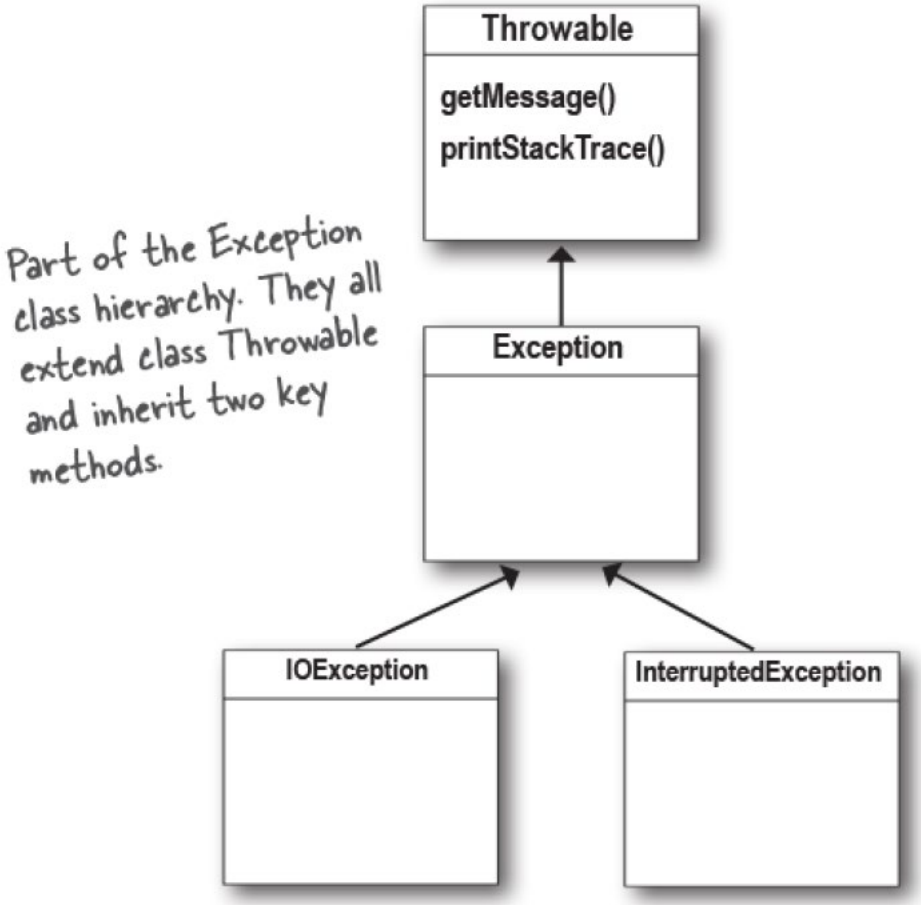
A try/catch block tells the compiler that you know an exceptional thing could happen in the method you're calling, and that you're prepared to handle it.

That compiler doesn't care how you handle it; it cares only that you say you're taking care of it.



An exception is an object... of type Exception

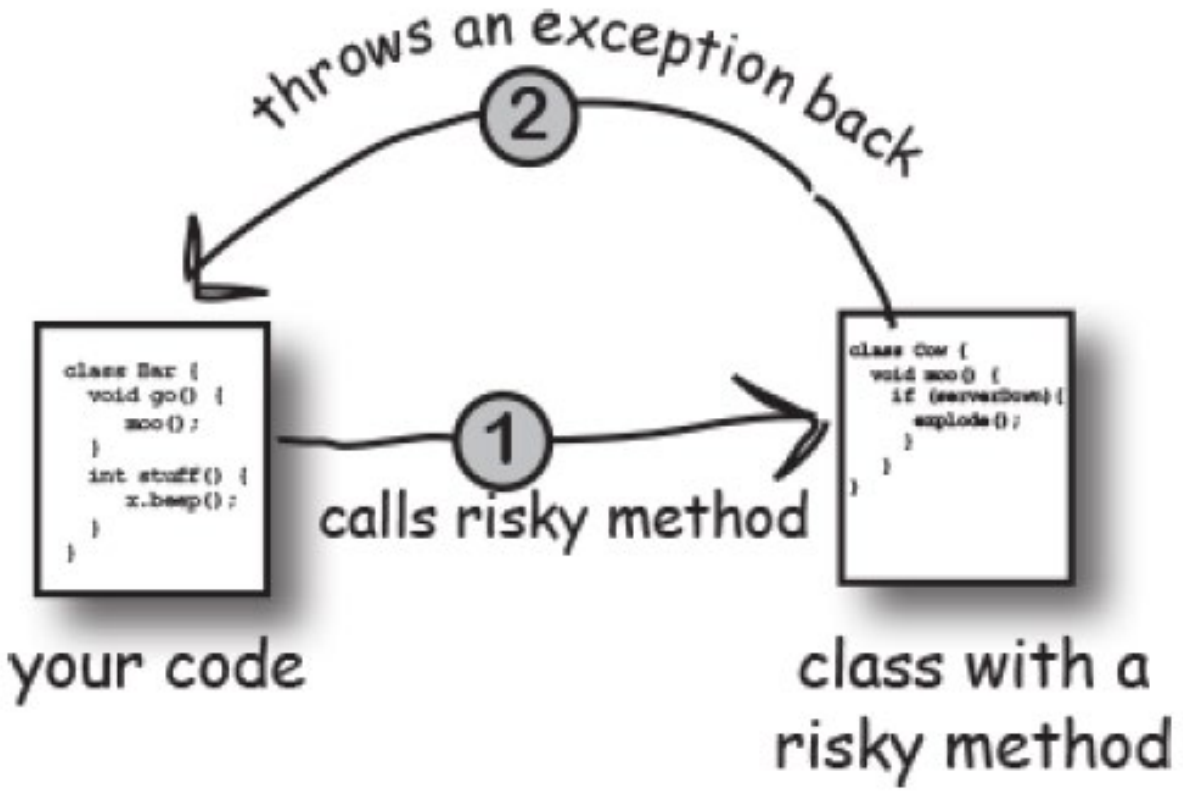
Extends Throwable and inherits getMessage() and printStackTrace()

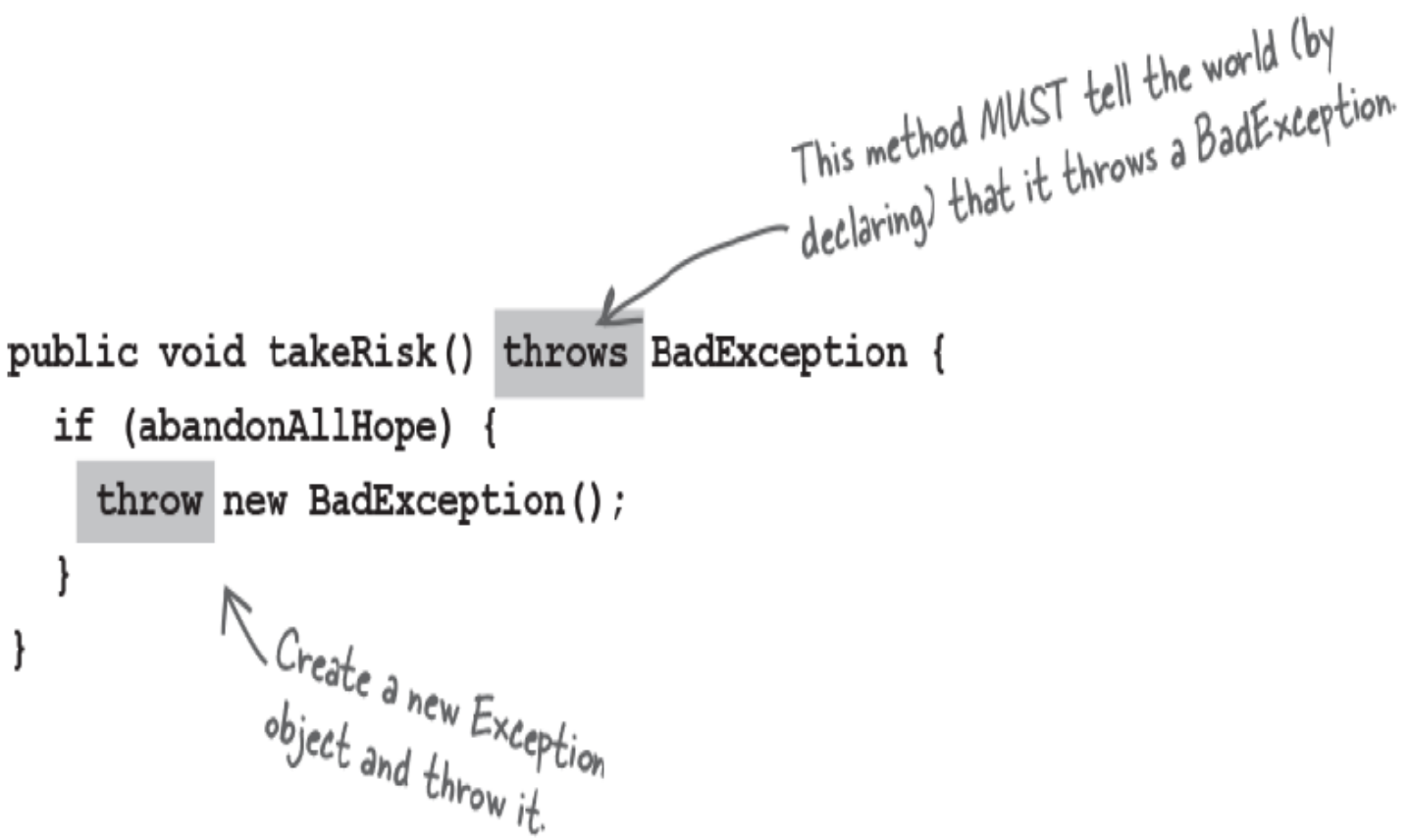


Stack Trace example ...

```
D:\java>java Hello
Exception in thread "main" java.lang.NoClassDefFoundError: Hello
Caused by: java.lang.ClassNotFoundException: Hello
        at java.net.URLClassLoader$1.run(URLClassLoader.java:200)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:307)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:301)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:252)
        at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:320)
Could not find the main class: Hello.  Program will exit.
```

If it's your code that catches the exception, then whose code throws it?





Risky, exception-throwing code

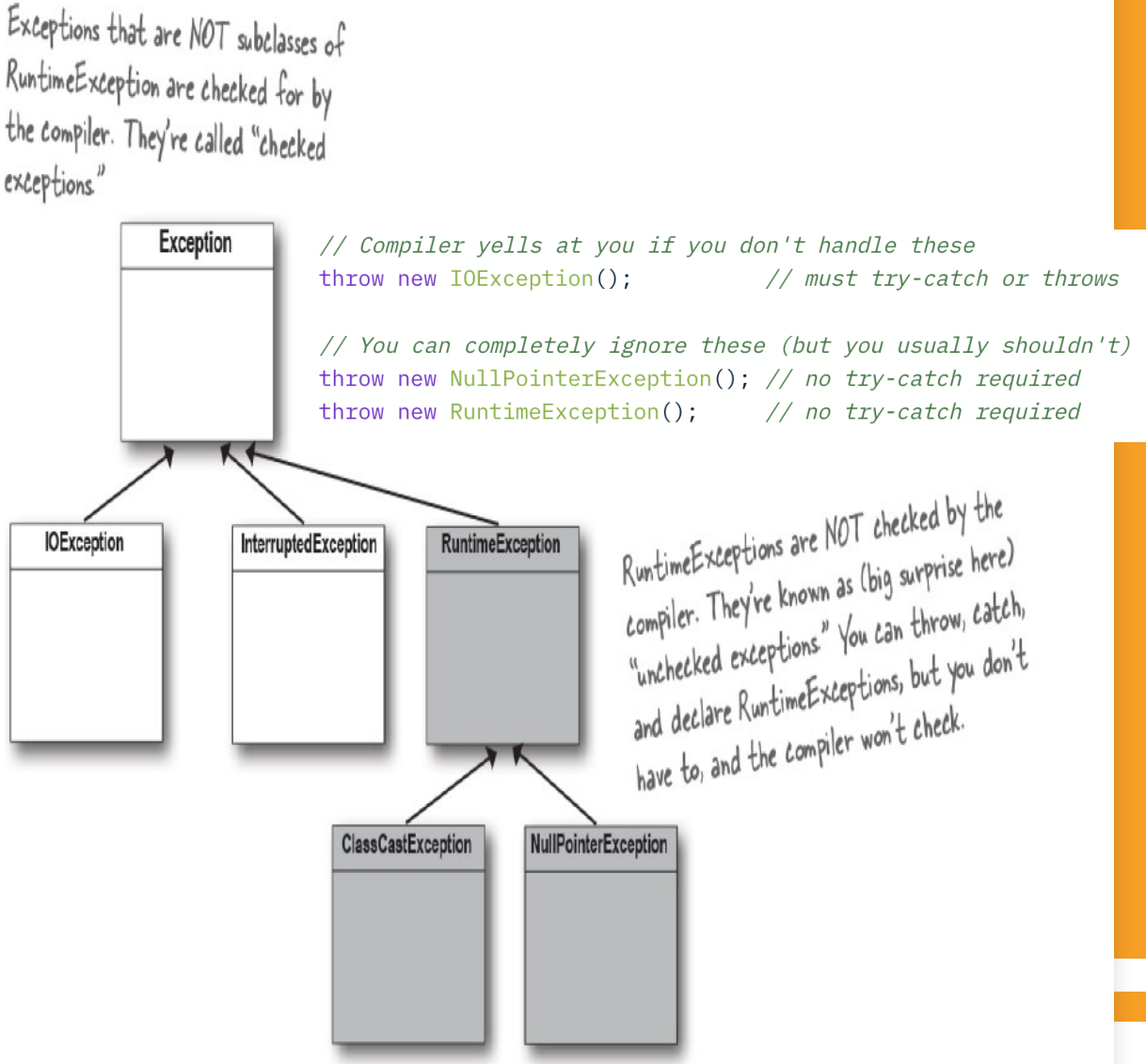
```
public void crossFingers() {
  try {
    anObject.takeRisk();
  } catch (BadException e) {
    System.out.println("Aaargh!");
    e.printStackTrace();
  }
}
```

If you can't recover from the exception, at LEAST get a stack trace using the printStackTrace() method that all exceptions inherit.

Your code that calls the risky method

One method will catch what another method throws. An exception is always thrown back to the caller.

The method that throws has to declare that it might throw the exception.

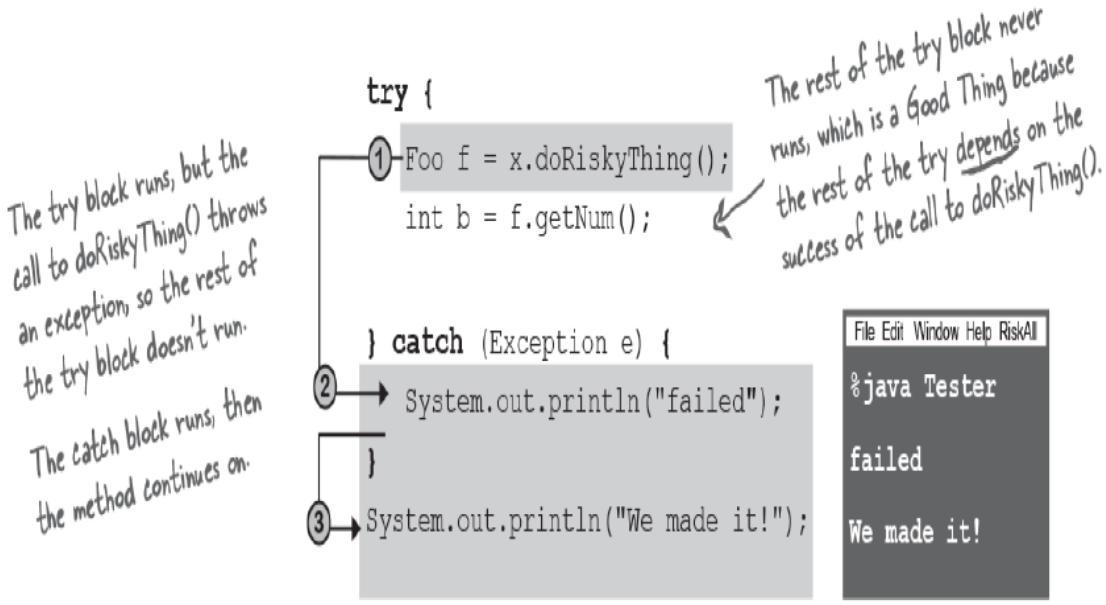


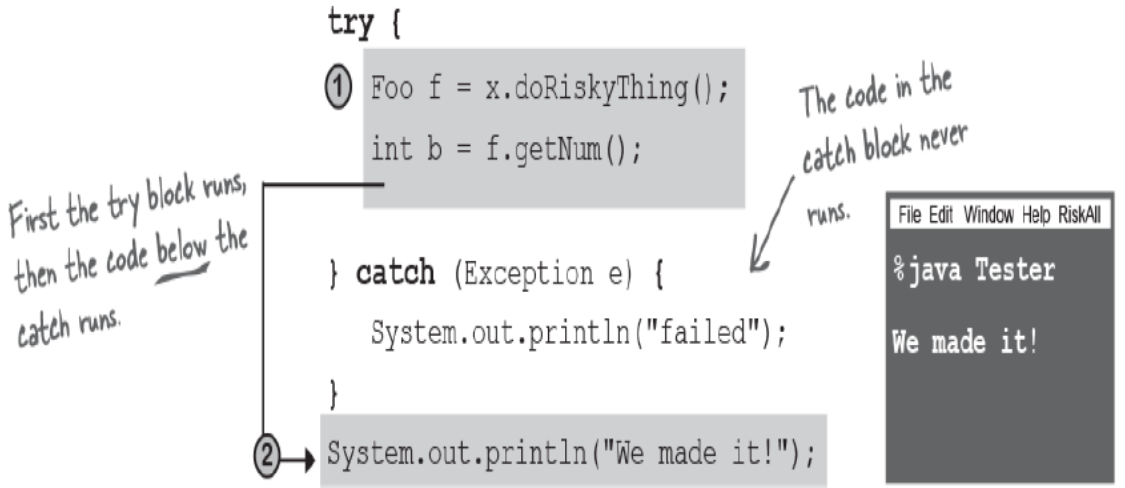
The compiler guarantees

- If you throw an exception in your code, you must declare it using the throws keyword in your method declaration.
- If you call a method that throws an exception (in other words, a method that declares it throws an exception), you must acknowledge that you're aware of the exception possibility. One way to satisfy the compiler is to wrap the call in a try/catch.

Flow control in try/catch blocks

- When you call a risky method, one of two things can happens:
  - The risky method either succeeds, and the try block completes,
  - or the risky method throws an exception back to your calling method.





```
try {
  turnOvenOn();
  x.bake();
  turnOvenOff();
} catch (BakingException e) {
  e.printStackTrace();
  turnOvenOff();
}
```

```
try {
  turnOvenOn();
  x.bake();
} catch (BakingException e) {
  e.printStackTrace();
} finally {
  turnOvenOff();
}
```

Finally: for the things you want to do no matter what

- Without finally, you have to put the turnOvenOff() in both the try and the catch because you have to turn off the oven no matter what. A finally block lets you put all your important cleanup code in one place instead of duplicating.

- A finally block is where you put code that must run regardless of an exception.

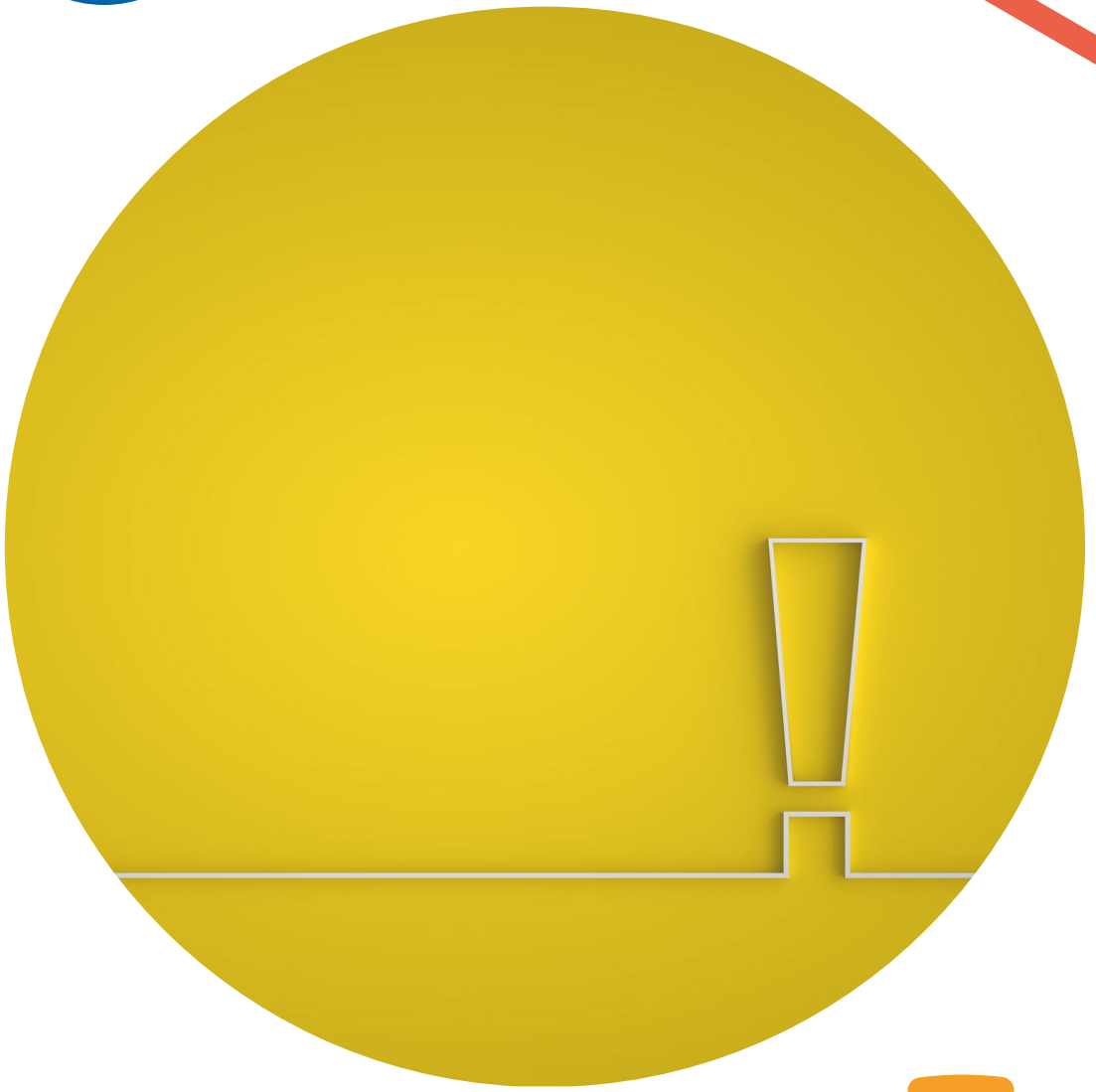
Multiple Exception Handling

- It is possible that the tryblock contain code that causes exceptions of various kinds to occur
- Therefore, it is legal to have multiple catchblocks, each to handle an exception of a different kind
- The first catch block that contains the appropriate exception class will be executed
- Therefore, the last catch block should contain code for handling an Exception type exception
  - i.e. if none of the specialized exception handlers does not get invoked, the generic Exception class handler will get invoked

```java
public class Exceptions
{
Run | Debug
public static void main(String[] args)
{
    System.out.println(x:"An attempt to divide by zero:");
    try
    {
    System.out.println(0/0);
    }
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println(x:"An ArrayIndexOutOfBoundsExceptionoccured: ");
        e.printStackTrace();
    }
    catch(ArithmeticException e)
    {
        System.out.println(x:"An ArithmeticException occured:");
        e.printStackTrace();
    }
    catch(Exception e)
    {
        System.out.println(x:"An Exception occured: ");
        e.printStackTrace();
    }
    System.out.println(x:"An attempt to divide by zero was made");
}
}
```

Key takeaways

- A method can throw an exception when something fails at runtime.
- An exception is always an object of type Exception.
- All Exceptions the compiler cares about are called “checked exceptions,” which really means compiler-checked exceptions.
- Only RuntimeExceptions are excluded from compiler checking.
- All other exceptions must be acknowledged in your code.
- A method throws an exception with the keyword throw, followed by a new exception object: throw new NoCaffeineException();
- Methods that might throw a checked exception must announce it with a throws SomeException declaration.
- If your code calls a checked-exception-throwing method, it must reassure the compiler that precautions have been taken.
- If you’re prepared to handle the exception, wrap the call in a try/catch, and put your exception handling/recovery code in the catch block.



User Defined Exceptions

Exception Throwing on purpose

- The occurrence of exceptions is called exception throwing
- If a certain part of the code results in exceptions occurring, it is said that that part throws an exception
  - E.g. the line System.out.println(0/0);throws an ArithmeticException
- But, since exceptions can also be instantiated by the programmer (using the new statement), they can thrown by the programmer as well

The throw keyword

- Exceptions are thrown using the throw keyword
- Example:
  - Exception e = new Exception();
  - throw e;
- or simply
  - throw new Exception();
- The last line is preferred, because the exceptions can be handled only inside catch blocks and there is no need to reference them before

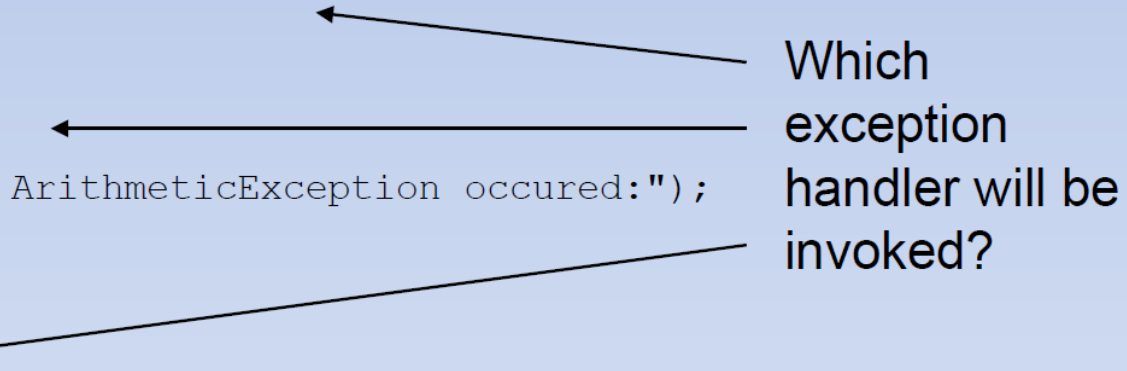
```java
public class Exceptions
{
    public static void main(String[] args)
    {
        System.out.println("Programmaticalthrowing of exceptions:");
    try{
        throw new ArithmeticException();
    }
    catch(ArrayIndexOutOfBoundsExceptione){
        System.out.println("An ArrayIndexOutOfBoundsExceptionoccured: ");
        e.printStackTrace();
    }
    catch(ArithmeticExceptione){
        System.out.println("An ArithmeticExceptionoccured:");
        e.printStackTrace();
    }
    catch(Exception e){
        System.out.println("An Exception occured: ");
        e.printStackTrace();
    }
    finally{
        System.out.println("The exception has been handled!");
    }
    System.out.println("Programmatical throwing of exceptions was made");
    }
}
```

Define Exceptions

- The Exception class (or any of its subclasses) can further be extended, to generate user defined exception classes
- In the previous example, the exception that was thrown did not have a default message
  - E.g. the message used upon division by zero is"/ by zero"
- When the user defines an exception class, a good idea is to create a constructor for it, which will accept a String parameter, which will utilize the appropriate constructor of the class
  - The Exception class has a constructor which accepts a String parameter, containing the message that the exception will show when thrown
  - Therefore, every subclass of the Exception class can have user defined messages which are given to its instances

```
class UserDefinedException extends Exception
{
  UserDefinedException()
  {
  }

  UserDefinedException(String message)
  {
     super(message);
  }
}
```

```
public class Exceptions
{
    public static void main(String[] args)
    {
          System.out.println("Programmatical throwing of exceptions:");
          try
          {
            throw new UserDefinedException("A UserDefinedException was instantiated!");
          }
          catch(ArrayIndexOutOfBoundsException e)
          {
                  System.out.println("An ArrayIndexOutOfBoundsException occured: ");
                  e.printStackTrace();
          }
          catch(ArithmeticException e)
          {
                  System.out.println("An ArithmeticException occured:");
                  e.printStackTrace();
          }
          catch(Exception e)
          {
                  System.out.println("An Exception occured: ");
                  e.printStackTrace();
          }
          finally
          {
                  System.out.println("The exception has been handled!");
          }
          System.out.println("Programmatical throwing of exceptions was made");
    }
}
```



A Method Throwing an Exception the throws Keyword

- A method must be able to handle all exceptions that would occur during its execution
  - i.e. appropriate exception handlers must be defined
- Otherwise, it must be declared that the method itself throws an exception of a certain type
- This is done using the throws keyword upon method declaration
- If the method is declared to throw an exception, it doesn't need to contain try-catch-finally blocks
- Still, a method invoking another method, which in turn throws exceptions, must contain try-catch-finally blocks with appropriate exception handlers, so that the program would compile without errors

```
class UserDefinedException extends Exception{
    UserDefinedException()
    {}

    UserDefinedException(String message){
          super(message);
    }
}

public class Exceptions{

    static void throwException() throws UserDefinedException
    {
          throw new UserDefinedException("A UserDefinedException was instantiated!");
    }

    public static void main(String[] args)
    {
          System.out.println("Programmatical throwing of exceptions:");

          try{
                    throwException();
          }
          catch(Exception e){
                    System.out.println("An Exception occured: ");
                    e.printStackTrace();
          }
          finally{
                    System.out.println("The exception ahs been handled!");
          }
          System.out.println("Programmatical throwing of exceptions was made");
    }
}
```

Exceptions are Expensive!

- Exceptions occupy considerable processing power and memory and should be used sparingly
- It is much better to predict and alleviate the situations that would result in exceptions being thrown than to use try-catch-finally
- Exceptions should be used only when there is no other way to prevent the occurrence of certain situations upon run time
  - E.g. when an integer is requested to be input, and a String is input instead

Console I/O

- Exceptions are important when using console I/O
- There is no way to read data from the console (i.e. keyboard) without exception handling
  - This is because the method which performs the reading itself throws an exception
  - Since every object is converted to a String using the toString() method of the Object class, console input does not produce exceptions, but there is no way to guarantee “safe” input using the keyboard

Console I/O

- To enable console input, package java.io.*must be imported first
- This package contains all classes and methods necessary for console input
  - The System.out.println()method is a part of the java.lang.*package, which is an integral part of Java and is always implicitly imported
- Among them is the IOExceptionclass, which is instantiated and thrown during CMD input

Input Data Stream Readers

- CMD input in Java is done using input data stream readers
  - Input data stream readers are also objects, which are instantiated from their corresponding classes
- When the keyboard is read from, it is important to connect the input stream reader to the appropriate data stream reader, in this case the console (i.e. keyboard)
- The specifics will be shown during the exercises

Exceptions upon Reading

- All methods for reading (from any stream) throw corresponding exceptions that must be handled
  - The methods for reading are declared to throw exceptions
  - Specifically, IOException exceptions
- This is because there is no guarantee that the data input through the stream will have valid values
  - E.g. inputting a character that is not from the keyboard
- Therefore, to have a functioning program that reads data from the keyboard
  - The methods invoking the reading methods (e.g. public static void main()) must be declared as throwing exceptions, using the throws keyword; or
  - The methods for reading must be placed inside corresponding try-catch-finally blocks
  - The latter approach offers more flexibility, since it can precisely specify where the exception occured 29

Parsing

- All data input through the keyboard are accepted as Strings
- To obtain other data types (e.g. integers) through the keyboard, the data input as Strings must be parsed an converted to appropriate data types
- This is done using the static methods of the appropriate wrapper classes of the corresponding primitive data types
  - E.g. Integer.parseInt()converts a String to an int, Double.parseDouble()converts a String to a double etc

Exceptions upon Parsing

- The parsing methods are also declared to throw exceptions
  - Specifically, NumberFormatExceptions
- This is because there is always the possibility that a flawed number format, which needs to be parsed, will be input
  - E.g. a decimal number instead of an integer
- Therefore, the method invoking these parsing methods must handle their exceptions, or to declare itself as throwing exceptions (using the throws keyword)

Case studies

- Inappropriate Use of Exceptions: Overusing Exceptions for Control Flow
- E-Commerce Transaction Processing
- Banking Software for Loan Approval
- Inventory Management System

Inappropriate Use of Exceptions: Overusing Exceptions for Control Flow

- Scenario: User Input Validation in a Command-Line Interface
- Imagine a command-line interface (CLI) application for a simple quiz game, where the user is prompted to enter answers to multiple-choice questions. Each answer is supposed to be a single character ('A', 'B', 'C', or 'D').
- Inadequate Use of Exceptions:
  - An inexperienced developer might decide to use exceptions for handling user input validation. For instance, they might throw an `IllegalArgumentException` if the user inputs an invalid character and use a try-catch block to prompt the user again.

Why This Is Inefficient

- Performance Overhead: Exceptions are expensive in terms of system resources. Throwing and catching exceptions, especially in a loop, can lead to significant performance degradation, particularly in a high-throughput environment.
- Semantic Inaccuracy: Exceptions are meant for exceptional conditions, i.e., error scenarios that are not expected to occur as part of the normal program flow. User input errors are a common occurrence and should be anticipated as part of regular control flow.
- Readability and Maintainability: Using exceptions for control flow can make the code harder to read and maintain. It mixes error handling logic with business logic, leading to potential confusion for other developers who may work on the code later.

```
while (true) {
    try {
        System.out.print("Enter your answer (A, B, C, D): ");
        String input = userInput.nextLine().toUpperCase();
        if (!input.matches("[ABCD]")) {
            throw new IllegalArgumentException("Invalid input.");
        }
        // Process the valid input
        break;
    } catch (IllegalArgumentException e) {
        System.out.println("Please enter a valid option: A, B, C, or D.");
    }
}
```

Better Approach: Standard Conditional Checks

A more efficient and clearer way to handle this scenario is to use standard conditional statements (like `if-else`) for input validation.

```
while (true) {
    System.out.print("Enter your answer (A, B, C, D): ");
    String input = userInput.nextLine().toUpperCase();
    if (input.matches("[ABCD]")) {
        // Process the valid input
        break;
    } else {
        System.out.println("Please enter a valid option: A, B, C, or D.");
    }
}
```

E-Commerce Transaction Processing

- Scenario:
  - In an e-commerce application, transactions involve several steps like item selection, payment processing, and order confirmation. Each of these steps can encounter issues like invalid input, payment gateway errors, or database connectivity issues.

- Use of Exceptions:
  - Input Validation Exceptions: When a user inputs invalid data (like an incorrect product code or quantity), an `IllegalArgumentException` can be thrown and caught to provide user-friendly error messages.
  - Payment Processing Exceptions: Payment gateways can fail due to network issues or invalid payment details. Catching exceptions like `IOException` or custom exceptions like `PaymentProcessingException` allows the application to rollback the transaction and inform the user of the failure, enhancing user experience and trust.
  - Database Connectivity Exceptions: When the application cannot connect to the database due to server downtime, an `SQLException` can be caught. This enables the application to try a reconnect, log the error for maintenance, and alert the user appropriately.

Banking Software for Loan Approval

- Scenario:
  - A banking application that processes loan applications needs to handle various checks like credit score validation, document verification, and regulatory compliance.

- Use of Exceptions:
  - Credit Score Check Exceptions: If the service that checks credit scores is unavailable, a custom exception like `CreditScoreServiceUnavailableException` can be thrown. This allows the application to queue the request and retry once the service is back, ensuring no customer application is unfairly rejected.
  - Document Verification Exceptions: Missing or invalid documents can lead to a `DocumentVerificationException`. This exception can trigger an automated email to the customer requesting the correct documents, improving process efficiency.
  - Regulatory Compliance Exceptions: Compliance checks might fail due to changing regulations or data issues. Catching a `ComplianceCheckException` allows the bank to hold the application process and conduct a manual review, ensuring legal compliance.

Inventory Management System

- Scenario:
  - An inventory management system in a retail business needs to handle stock updates, supplier orders, and inventory audits. These operations are prone to errors like stock mismatches or supplier communication issues.

- Use of Exceptions:
  - Stock Update Exceptions: When updating stock levels, a `StockLevelException` might occur if the new level goes below a critical threshold. This exception can trigger an automatic reorder process or alert the inventory manager.
  - Supplier Communication Exceptions: Issues in communicating with suppliers, such as network failures, can result in an `IOException`. Catching this exception allows the system to retry the communication or switch to an alternative communication method.
  - Inventory Audit Exceptions: Discrepancies during inventory audits can trigger an `InventoryMismatchException`. This can initiate a more thorough audit process and alert relevant departments to investigate potential issues like theft or data entry errors.

Conclusion

In all these cases, Java exceptions provide a structured way to handle error conditions, allowing business applications to respond appropriately to various challenges. This not only ensures the smooth operation of critical business processes but also enhances the overall reliability and user experience of the application.

Questions?