What are Exceptions?

- An exception is an exceptional event that disrupts the normal flow of instructions during the execution of a program.

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

The Call Stack

Function call Seq. 
Recursive search for handler 
main() starts the method calls 
method 10 has an exception handler 
method 20 has no Exception handler 
method 30 has no exception handler 
method 40 generates the exception!!!
**Exception catch-or-specify requirement**

- **Catch** – A method can catch an exception by providing an exception handler for that type of exception.
- **Specify** – A method specifies that it can throw exceptions by using the `throws` clause in the method declaration.
- **Checked exceptions** – There are two kinds of exceptions:
  - runtime exceptions and
  - non-runtime exceptions.

**Exception catch-or-specify requirement**

- **Runtime exceptions** occur within the Java runtime system: arithmetic exceptions (e.g., 1/0), pointer exceptions (e.g., null.member), and indexing exceptions (e.g., `a = array[-1][Max+1];`).
- **A method does not have to catch or specify runtime exceptions, although it may.**

**Non-runtime exceptions** are exceptions that occur in code outside of the Java runtime system. For example, exceptions that occur during I/O are non-runtime exceptions.

The compiler requires that non-runtime exceptions are caught or specified; hence checked exceptions.

**Exceptions that can be thrown by a method** include:

- Any exception thrown directly by the method with the `throw` statement
- Any exception thrown indirectly by calling another method that throws an exception

---

**Exception catch-or-specify requirement**

**Exception catch-or-specify requirement**

**Exception catching and handling**

- **Exception handling mechanism** — the `try`, `catch`, and `finally` blocks
- The following example defines and implements a class named `ListOfNumbers`. Which creates a Vector that contains ten Integer elements numbered 0-9.
- The `ListOfNumbers` class also defines a method named `writeList` that writes the list of numbers into a text file called `OutFile.txt`.

```java
public class ListOfNumbers {
    private Vector vec;
    private static final int SIZE = 10;

    public ListOfNumbers ()
    {
        vec = new Vector(SIZE);
        for (int i = 0; i < SIZE; i++)
            vec.addElement(new Integer(i));
    }

    public void writeList()
    {
        try {
            // code that may throw an exception
            catch (Exception e) {
                // catch exception
            }
            finally {
                // code that always runs
            }
        }
    }
}
```
Exception catching and handling – Ex.

`public class ListOfNumbers {
    ...
    public void writeList()
        { PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
          for (int i = 0; i < SIZE; i++)
            out.println("Value at:" + i + "=" + vec.elementAt(i));
          out.close();
        }
}

Exception catching and handling – Ex.

• The call to `PrintWriter` constructor initializes an output stream on a file. If the file cannot be opened, the constructor throws an `IOException`.
• The Vector class’s `elementAt` method, which throws an `ArrayIndexOutOfBoundsException` if the value of its argument is too small (<0) or too large (>Max).
• Trying to compile `ListOfNumbers` class generates 1 error message about the exception thrown by the `FileWriter` constructor –

However, it does not display an error message about the exception thrown by `elementAt`, runtime exception (`ArrayIndexOutOfBoundsException`). Whereas the exception thrown by the constructor, (`IOException`), is a checked exception.

Exception handling – try block

You associate exception handlers with a try block by providing one or more catch blocks directly after the try:
```
try {
  statements
}
```

Each catch block is an exception handler and handles the type of exception indicated by its argument.
Exception handling – catch block

Two exception handlers for `writeList` method

```java
try {
    ...
} catch (ArrayIndexOutOfBoundsException e) {
    System.err.println("Caught ArrayIndexOutOfBoundsException:" + e.getMessage() );
} catch (IOException e) {
    System.err.println("Caught IOException:" + e.getMessage() );
}
```

Exception handling – catch block

- The handlers shown print an error message. Although simple, this might be the behavior you want.
- The exception gets caught, the user is notified, and the program continues to execute.
- However, exception handlers can do more. They can do error recovery, prompt the user to make a decision, or decide to exit the program.

Exception handling – finally block

- The runtime system always executes the statements within the finally block regardless of what happens within the try block.
- The runtime system always executes the statements within the finally block regardless of what happens within the try block.

**Format for HW Project 6**

- **Open-ended**: You pick a (practical but simple) project that you write the Java package to solves the problem
  - E.g.,
    - Get a list of *.gif files from the user and animate them as a movie 500 milliseconds apart
    - Graphics Package – draw an object of interest (circle, rectangle, line, cube, star, etc.) with pre-defined parameters (center, size, line-width, etc.) with a desired color
    - Bouncing-ball application
    - Scientific Calculator

- **Completely structured**: I select one problem that everyone works on!
  - Either way we’ll want to use OOP design, exception handling, GUI widgets, packaging, etc.
  - E.g.,
    - Get a list of *.gif files from the user and animate them as a movie 500 milliseconds apart
    - Graphics Package – draw an object of interest (circle, rectangle, line, cube, star, etc.) with pre-defined parameters (center, size, line-width, etc.) with a desired color
    - Bouncing-ball application
    - Scientific Calculator

Exception handling – finally block

- This poses a somewhat complicated problem because `writeList`’s try block can exit in one of three ways.
  - The new `FileWriter` statement fails and throws an `IOException`.
  - The `vec.elementAt(i)` statement fails and throws an `ArrayIndexOutOfBoundsException`.
  - Everything succeeds and the try block exits normally.

```java
finally {
    if (out !=null){
        System.out.println("Closing PrintWriter");
        out.close();
    } else
        System.out.println("PrintWriter not open");
}
```
Exception handling

- The try block in this method has three exit possibilities.
  - The new `FileWriter` statement fails and throws an `IOException`.
  - The `vec.elementAt(i)` statement fails and throws an `ArrayIndexOutOfBoundsException`.
  - Everything succeeds and the try statement exits normally.
- Let’s look at what happens in the `writeList` method during each of these exit possibilities.

Specifying Exceptions thrown by methods

- The throws clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method.
- The clause goes after the method name and argument list and before the brace that defines the scope of the method.

```java
public void writeList() throws IOException, ArrayIndexOutOfBoundsException {
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < size; i++)
        out.println("Value at:" + i + "=" + vec.elementAt(i));
    out.close();
}
```

To specify that `writeList` can throw two exceptions, you add a throws clause to the method declaration for the `writeList` method.

Specifying Exceptions thrown by methods

- The throws clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method.
- The clause goes after the method name and argument list and before the brace that defines the scope of the method.

```java
public void writeList() throws IOException, ArrayIndexOutOfBoundsException {
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < size; i++)
        out.println("Value at:" + i + "=" + vec.elementAt(i));
    out.close();
}
```

Remember that `ArrayIndexOutOfBoundsException` is a runtime exception, so you don’t have to specify it in the throws clause.

Throwing Exceptions

- Sometimes, it’s appropriate for your code to catch exceptions that can occur within it. In other cases, however, it’s better to let a method farther up the call stack handle the exception.
- For example, if you were providing the `ListOfNumbers` class as part of a package of classes, you probably couldn’t anticipate the needs of all the users of your package.
- In this case, it’s better to not catch the exception and to allow a method farther up the call stack to handle it.

```java
public void writeList() throws IOException, ArrayIndexOutOfBoundsException {
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < size; i++)
        out.println("Value at:" + i + "=" + vec.elementAt(i));
    out.close();
}
```

writeList() throws two exceptions (IOException, ArrayIndexOutOfBoundsException).

The throw clause

- The throw statement requires a single argument: a throwable object, instance of any subclass of the Throwable class.
- Ex. The method removes the top element from the stack and returns the object:

```java
public Object pop() throws EmptyStackException {
    Object obj;
    if (size == 0)
        throw new EmptyStackException();
    obj = objectAt(--size);
    return obj;
}
```

Specifying Exceptions thrown by methods

- The throws clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method.
- The clause goes after the method name and argument list and before the brace that defines the scope of the method.

```java
public void writeList() throws IOException, ArrayIndexOutOfBoundsException {
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < size; i++)
        out.println("Value at:" + i + "=" + vec.elementAt(i));
    out.close();
}
```

To specify that `writeList` can throw two exceptions, you add a throws clause to the method declaration for the `writeList` method.

Specifying Exceptions thrown by methods

- The throws clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method.
- The clause goes after the method name and argument list and before the brace that defines the scope of the method.

```java
public void writeList() throws IOException, ArrayIndexOutOfBoundsException {
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < size; i++)
        out.println("Value at:" + i + "=" + vec.elementAt(i));
    out.close();
}
```

Remember that `ArrayIndexOutOfBoundsException` is a runtime exception, so you don’t have to specify it in the throws clause.

Throwing Exceptions

- Sometimes, it’s appropriate for your code to catch exceptions that can occur within it. In other cases, however, it’s better to let a method farther up the call stack handle the exception.
- For example, if you were providing the `ListOfNumbers` class as part of a package of classes, you probably couldn’t anticipate the needs of all the users of your package.
- In this case, it’s better to not catch the exception and to allow a method farther up the call stack to handle it.

```java
public void writeList() throws IOException, ArrayIndexOutOfBoundsException {
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < size; i++)
        out.println("Value at:" + i + "=" + vec.elementAt(i));
    out.close();
}
```

writeList() throws two exceptions (IOException, ArrayIndexOutOfBoundsException).
The *throw* clause

- The objects that inherit from the `Throwable` class include direct descendants and indirect descendants (objects that inherit from children or grandchildren of the `Throwable` class).
- Most significant subclasses.

```
Object
    Throwable
        Error
            ...  
        Exception
            ...  
        RuntimeException
            ...  
```

Exceptions vs. Errors

- When a dynamic linking failure or other “hard” failure in the Java VM occurs, the Java VM throws an `Error`. Typical programs should not catch `Errors`. Also, typical programs never throw `Errors`.
- Most programs throw and catch objects that derive from the `Exception` class. An Exception indicates that a non serious system problem occurred.
- The `Exception` class has many descendants defined in the Java platform. E.g., `IllegalAccessError` signals that a particular method could not be found, and `NegativeArraySizeException` indicates an attempted to create an array with a negative size.

Extending the `Exception` class

- You may use other’s Exception classes, but consider writing your own exception classes if:
  - Do you need an exception type that isn’t represented by those in the Java platform?
  - Would it help your users if they could differentiate your exceptions from those thrown by classes written by others?
  - Does your code throw many related exceptions?
  - Will your users have access to those exceptions if you’re using others Exceptions?
  - Should your package be independent and self-contained?

Why use Exceptions?

1. To separate the details of what to do when something out of the ordinary happens. E.g.,
   ```java
   readFile { try { open the file; determine its size; allocate that much memory; read the file into memory; close the file; } catch (fileOpenFailed) doSomething1; catch (sizeDeterminationFailed) doSomething2; catch (memoryAllocFailed) doSomething3; catch (readFailed) doSomething4; catch (fileCloseFailed) doSomething5; }
   ```

2. To allow error propagation up the call-stack. E.g.,
   ```java
   method1 { try { call method2; } catch (Exception e) doErrorProcessing; }
   method2 throws Exception { call method3; }
   method3 throws Exception { call readFile; }
   ```
Why use Exceptions?

3. To allow grouping / differentiating error types. E.g.,
This handler will catch all I/O exceptions, including FileNotFoundException, EOFException, and so on.
catch (IOException e) {
   e.printStackTrace(); //output goes to System.err
   e.printStackTrace(System.out);//send trace to stdout
}
This handler handles any Exception
catch (Exception e){ // a (too) general exception handler
   ...
}

An Exception Handling Example: Divide by Zero

Example program
- User enters two integers to be divided
- We want to catch division by zero errors
- Exceptions
  - Objects derived from class Exception
  - Look in Exception classes in java.lang
  - Nothing appropriate for divide by zero
  - Closest is ArithmeticException
  - Extend and create our own exception class

Public class DivideByZeroException
{
   extends ArithmeticException
define our own exception class (exceptions are thrown objects)
   Default constructor (default message)
   customize message constructor
}

An Exception Handling Example: Divide by Zero

An Exception Handling Example: Divide by Zero

Class DivideByZeroException (extends Arithmetic Exception)

1. Constructor
2. Super

An Exception Handling Example: Divide by Zero

Code that may throw exception in try block
Covered in more detail in following sections

Error handling code in catch block
If no exception thrown, catch blocks skipped
2. Process GUI events

```java
try
    input2 = new JTextField(10);
    c.add(input2);
    input2.addActionListener(this);

    c.add(new JLabel("RESULT ", SwingConstants.RIGHT));
    output = new JTextField();
    c.add(output);

    setSize(425, 100);
    show();

catch (NumberFormatException nfe) {
    JOptionPane.showMessageDialog(this, "You must enter two integers", "Invalid Number Format", JOptionPane.ERROR_MESSAGE);
}
```

Notice enclosing try block. If an exception is thrown in the block (even from a method call), the entire block is terminated.

```
catch (DivideByZeroException dbze) {
    JOptionPane.showMessageDialog(this, dbze.toString(), "Attempted to Divide by Zero", JOptionPane.ERROR_MESSAGE);
}
```

Method `quotient` throws an `DivideByZeroException` exception (object) if denominator == 0.

```
public static void main(String args[]) {
    DivideByZeroTest app = new DivideByZeroTest();
    app.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            e.getWindow().dispose();
            System.exit(0);
        }
    });
}
```

### Outcome

**Try Blocks**

- Exceptions that occur in a `try` block are usually caught by handler specified by following `catch` block.

  ```java
  try {
      code that may throw exceptions
  } catch (ExceptionType ref) {
      exception handling code
  }
  ```

  - Can have any number of `catch` blocks
  - If no exceptions thrown, `catch` blocks skipped

**Throwing an Exception**

- `throw` indicates exception has occurred (throwing an exception).
  - `Operand` - Object of any class derived from `Throwable`.
  - `Derived from Throwable`:
    - `Exception` - most programmers deal with.
    - `Error` - serious, should not be caught.

- When exception thrown:
  - Control exits current `try` block.
  - Proceeds to `catch` handler (if exists).
**Throwing an Exception**

- Exceptions
  - Can still throw exceptions without explicit `throw` statement
  - `ArrayIndexOutOfBoundsException`
  - Terminates block that threw exception
    - Not required to terminate program

**Catching an Exception**

- `catch` blocks
  - Contain exception handlers
  - Format:
    ```java
    catch( Exception e ) {
    error handling code
    }
    ```

  ```java
  catch ( DivideByZeroException dbze ) {
  JOptionPane.showMessageDialog( this, dbze.toString(),
  "Attempted to Divide by Zero", JOptionPane.ERROR_MESSAGE );
  }
  ```

- To catch all exceptions, catch an exception object:
  ```java
  catch( Exception e )
  ```

**Catching an Exception**

- Catching exceptions
  - First handler to catch exception does
  - All other handlers skipped
  - If exception not caught
    - Searches enclosing `try` blocks for appropriate handler

  ```java
  try{
  try{
  throw Exception2
  } catch ( Exception1 ){...}
  } catch( Exception2 ){...}
  ```

  - If still not caught, non-GUI based applications terminate

**Rethrowing an Exception**

- Rethrowing exceptions
  - Use if handler cannot process exception
  - Rethrow exception with the statement:
    ```java
    throw e;
    ```

  - Detected by next enclosing `try` block

  - Handler can always rethrow exception, even if it performed some processing

**Throws Clause**

- Throws clause
  - Lists exceptions that can be thrown by a method

  ```java
  public double quotient( int numerator, int denominator )
  throws DivideByZeroException
  ```

  ```java
  int g( float h ) throws a, b, c {
  // method body
  }
  ```

  - Method can throw listed exceptions or derived types
**Throws Clause**

- **Run-time exceptions**
  - Derive from `RunTimeException`
  - Some exceptions can occur at any point
    - `ArrayIndexOutOfBoundsException`
    - `NullPointerException` (Create object reference without attaching object to reference)
    - `ClassCastException` (Invalid casts)
  - Most avoidable by writing proper code

- **Checked exceptions**
  - Must be listed in `throws` clause of method
  - All non-`RuntimeExceptions`

- **Unchecked exceptions**
  - Can be thrown from almost any method
    - Tedium to write `throws` clause every time
    - No `throws` clause needed
  - Errors and `RuntimeExceptions`

**Catch-or-declare requirement**

- If method calls another method that explicitly throws checked exceptions
  - Exceptions must be in original method's `throws` clause
- Otherwise, original method must catch exception
- Method must either catch exception or declare it in the `throw` clause

**Constructors, Finalizers and Exception Handling**

- **What to do with an error in constructor?**
  - Constructor cannot return value
  - How do we inform program of error?
  - Possible solutions
    - Hope someone tests defective object
    - Set some variable outside constructor
    - Thrown exception informs program of a failed constructor
- Exceptions thrown in constructors
  - Object marked for garbage collection
  - `finalize`
  - No particular order

**Exceptions and Inheritance**

- **Inheritance**
  - Exception classes can have a common superclass
  - `catch ( Superclass ref )`
    - Catches subclasses
    - "Is a" relationship
    - Polymorphic processing
    - Easier to catch superclass than catching every subclass

**finally Block**

- **Resource leaks**
  - Programs obtain and do not return resources
  - Automatic garbage collection avoids most memory leaks
  - Other leaks can still occur

- **finally block**
  - Placed after last `catch` block
  - Can be used to return resources allocated in `try` block
  - Always executed, irregardless whether exceptions thrown or caught
  - If exception thrown in `finally` block, processed by enclosing `try` block
  - If there was an original exception, it is lost
// UsingExceptions.java
// Demonstration of method call stack for caught Exception object.
// Method printStackTrace
// Prints method call stack for caught Exception object.
// Most recent method on top of stack.
// Helpful for testing/debugging.
// Constructors
// Exception()
// Exception(String informationString)
// informationString may be accessed with method getMessage.

// Using printStackTrace and getMessage

Exception thrown in method3
java.lang.Exception: Exception thrown in method3
at UsingExceptions.method3(UsingExceptions.java:28)
at UsingExceptions.method2(UsingExceptions.java:23)
at UsingExceptions.method1(UsingExceptions.java:18)
at UsingExceptions.main(UsingExceptions.java:8)