Java Programming

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today’s Lecture

- Exception Handling

Exception Handling

- Exception – An indication of a problem that occurs during a program’s execution.
- No need to terminate program when an error occurs
- Exception handling allows the program to continue executing after dealing with the problem.

- Normal Error Handling Pseudocode

```
Perform a task
If the preceding task did not execute correctly
        Perform error processing

Perform next task
If the preceding task did not execute correctly
        Perform error processing

… and so on
```

Error Handling Overview

- Exception handling allows you to remove error-handling code from the "main line".

- In previous pseudocode you must check for errors even if they occur infrequently.

- Benefits of removing from "main line".
  - Improve program clarity
  - Enhances modifiability

- You choose which exceptions to handle.

Error Handling Overview

Error Handling Overview

- Exceptions are "thrown".
- When a method detects a problem and is unable to handle it that method "throws" an exception.
- If an exception was thrown then an error has occurred.
- For example…

```
public static int quotient( int numerator, int denominator ) {
          return numerator / denominator;
}
```

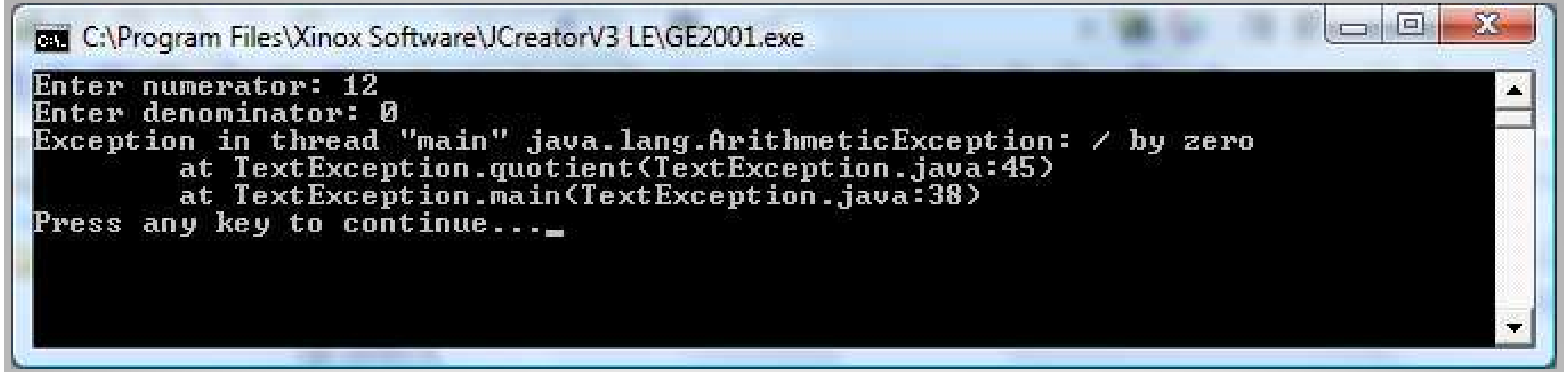
Divide by zero causes exception to be thrown

```
public static void main(String[] args) {
   Scanner scanner  = new Scanner(System.in);

   System.out.print("Enter numerator: ");
   int numerator = scanner.nextInt();
   System.out.print("Enter denominator: ");
   int denominator = scanner.nextInt();

   int result = quotient(numerator, denominator);
}
```

quotient method will throw an exception if dividing by zero

Divide By Zero

- Stack Trace
  - Gives the name of the exception that occurred.
  - Shows the method-call stack at the time the exception occurred.



```
Enter numerator: 12
Enter denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TextException.quotient(TextException.java:45)
        at TextException.main(TextException.java:38)
Press any key to continue..._
```

Divide By Zero

Divide By Zero

- Throw Point – The initial point where the exception occurs.

Where was the throw point in the division error example?

```
public static int quotient( int numerator, int denominator )
{
        return numerator / denominator;
}                          ↖ Throw Point

public static void main(String[] args)
{
        Scanner scanner  = new Scanner(System.in);
        int numerator = scanner.nextInt();
        int denominator = scanner.nextInt();

        int result = quotient(numerator, denominator);
}
```

Divide By Zero

Divide By Zero

- Dividing by zero with int data type causes a ***java.lang.ArithmeticException*** exception to be thrown.
- An ArithmeticException is NOT the only exception that can be thrown.
- There are many other types of exceptions that can be thrown.
- For example…

```
public static int quotient( int numerator, int denominator )
{
        return numerator / denominator;
}

public static void main(String[] args)
{
        Scanner scanner  = new Scanner(System.in);
        int numerator = scanner.nextInt();
        int denominator = scanner.nextInt();

        int result = quotient(numerator, denominator);
}
```

Both throw exceptions if not given an integer

Wrong Input Type

- Entering a string when an integer is required will cause a ***java.lang.InputMismatchException*** exception to be thrown.

Wrong Input Type

- The program STOPS executing when an exception occurs.

  Is this desirable behavior???

Divide By Zero

Handling Exceptions

- In the previous examples the program stopped when an exception occurred.
- It would be better to "handle" the exception and let the program keep running.

- Use a try/catch block to handle exceptions.
- Any code that can throw an exception should go inside the try/catch block.
- For example…

Handling Exceptions

```
public static int quotient( int numerator, int denominator )
{       return numerator / denominator;  }

public static void main(String[] args)
{
        try
        {
                Scanner scanner  = new Scanner(System.in);
                int numerator = scanner.nextInt();
                int denominator = scanner.nextInt();

                int result = quotient(numerator, denominator);
        }
        catch (…)
        {  …  }
}
```

Divide by zero causes exception to be thrown

Quotient call inside of try/catch block

What goes in the catch block?

```
public static int quotient( int numerator, int denominator )
{         return numerator / denominator;  }

public static void main(String[] args)
{
          try
          {
                    Scanner scanner  = new Scanner(System.in);
                    int numerator = scanner.nextInt();
                    int denominator = scanner.nextInt();

                    int result = quotient(numerator, denominator);
          }
          catch (ArithmeticException ae)
          {
                    System.err.println("Error");
          }
}
```

Divide by zero causes exception to be thrown

You "catch" exceptions here

```
public static int quotient( int numerator, int denominator )
{         return numerator / denominator;  }

public static void main(String[] args)
{
          try
          {
                    Scanner scanner  = new Scanner(System.in);
                    int numerator = scanner.nextInt();
                    int denominator = scanner.nextInt();

                    int result = quotient(numerator, denominator);
          }
          catch (ArithmeticException ae)
          {
                    System.err.println("Error");
          }
}
```

What happens if an input exception is thrown? →

- The program will crash with an InputMismatchException.
- Uncaught Exception – No matching catch block in the try statement that threw the exception.
- The only exception that the previous program handles is an ArithmeticException.

What can you do about this?

Handling Exceptions

```
public static int quotient( int numerator, int denominator )
{         return numerator / denominator;  }

public static void main(String[] args) {
          try {
                    Scanner scanner  = new Scanner(System.in);
                    int numerator = scanner.nextInt();
                    int denominator = scanner.nextInt();

                    int result = quotient(numerator, denominator);
          }
          catch (ArithmeticException ae) {
                    System.err.println("Error – Divide by zero.");
          }
          catch (InputMismatchException ime) {
                    System.err.println("Error – Incorrect input.");
          }
}
```

Add another catch block

- What will happen if an exception other than an ArithmeticException or an InputMismatchException occurs?

Handling Exceptions

- What will happen if an exception other than an ArithmeticException or an InputMismatchException occurs?

  Answer:

  Other exceptions area uncaught exceptions.

  The program will crash and output a stack trace for those exceptions.

Handling Exceptions

- In the previous example the exceptions are handled but it does not "recover" from the error.

- It would be better to give the user a chance to re-enter data.

Handling Exceptions

- Suppose that the user must enter a number and we do not want the program to crash if they enter something else like a string.

```
boolean continueLoop = true;
int num = 0;
Scanner keyboard = new Scanner(System.in);
do {
    try {
        System.out.println("Enter a number");
        num = keyboard.nextInt();

        continueLoop = false;

    } catch (InputMismatchException ime) {
        keyboard.nextLine(); // Consume the newline character
        System.out.println("Error - Enter a number");
    }
} while (continueLoop);
// Code to use num goes here...
System.out.println(num);
```

Initialize continueLoop to true

If the input succeeds, then it will go to the next statement and set continueLoop to false (this will cause the loop to end)

If a string was entered for nextInt it will throw an InputMismachException and be caught in the catch block. continueLoop will still be true, and the user will have to enter data again.

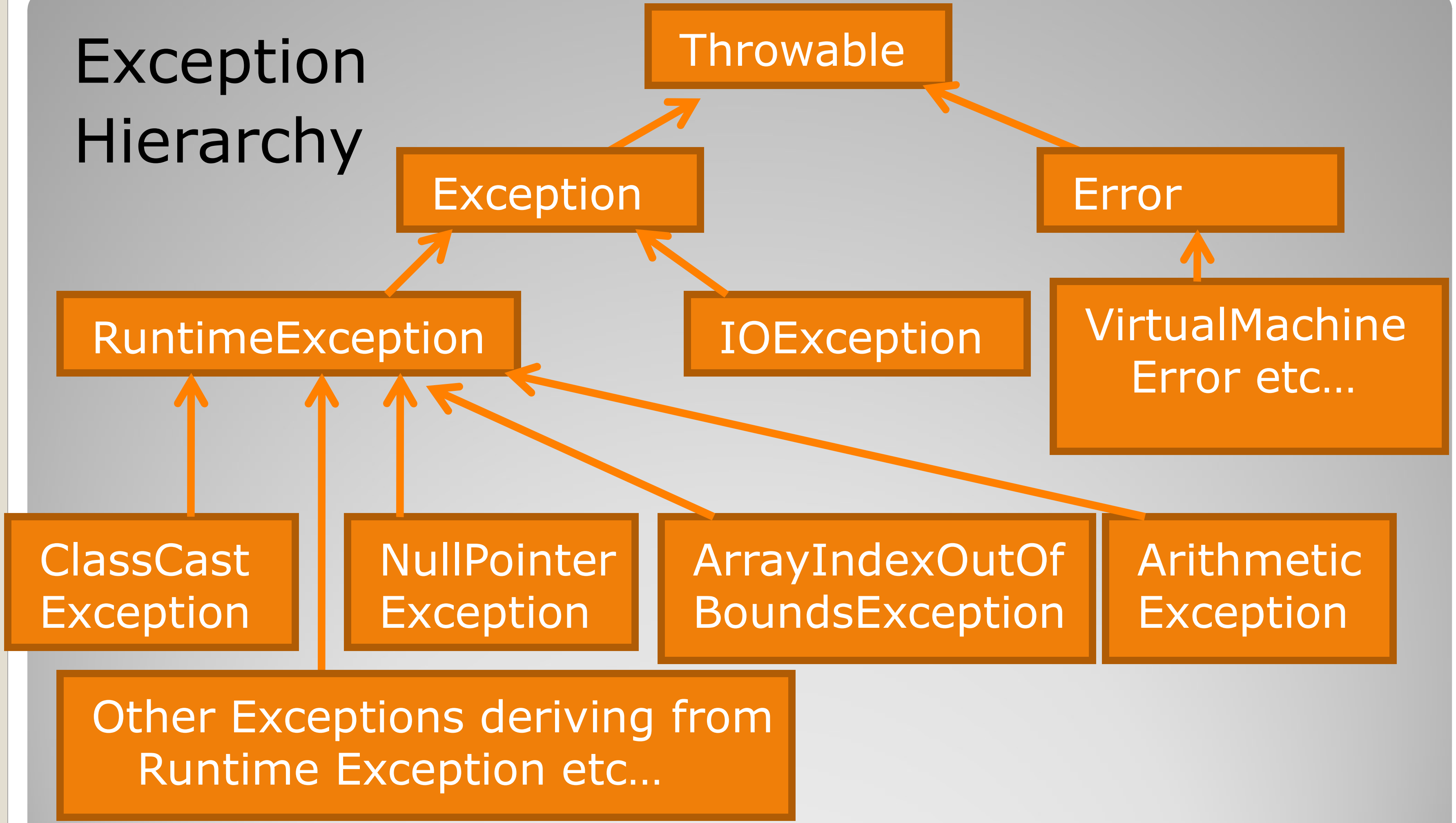
- **Throws** clause
- Part of a method declaration.
- Specifies the exceptions that a method throws.
- Quotient method contains a throws clause.

```
public static int quotient( int numerator, int denominator )
   throws AritmeticException
{
         return numerator / denominator;
}
```

Throws clause. Indicates that method quotient throws an ArithmeticException.

Java Exception Hierarchy

Exception Hierarchy



- Java compiler enforces a **"catch-or declare requirement"**.

- **Checked exception** – A method that throws a checked exception must be called under one of the following circumstances:
  - **The method call must be inside of a try statement that has a catch block for that exception.**

  or
  - **The method call must be inside of another method that "throws" that exception.**

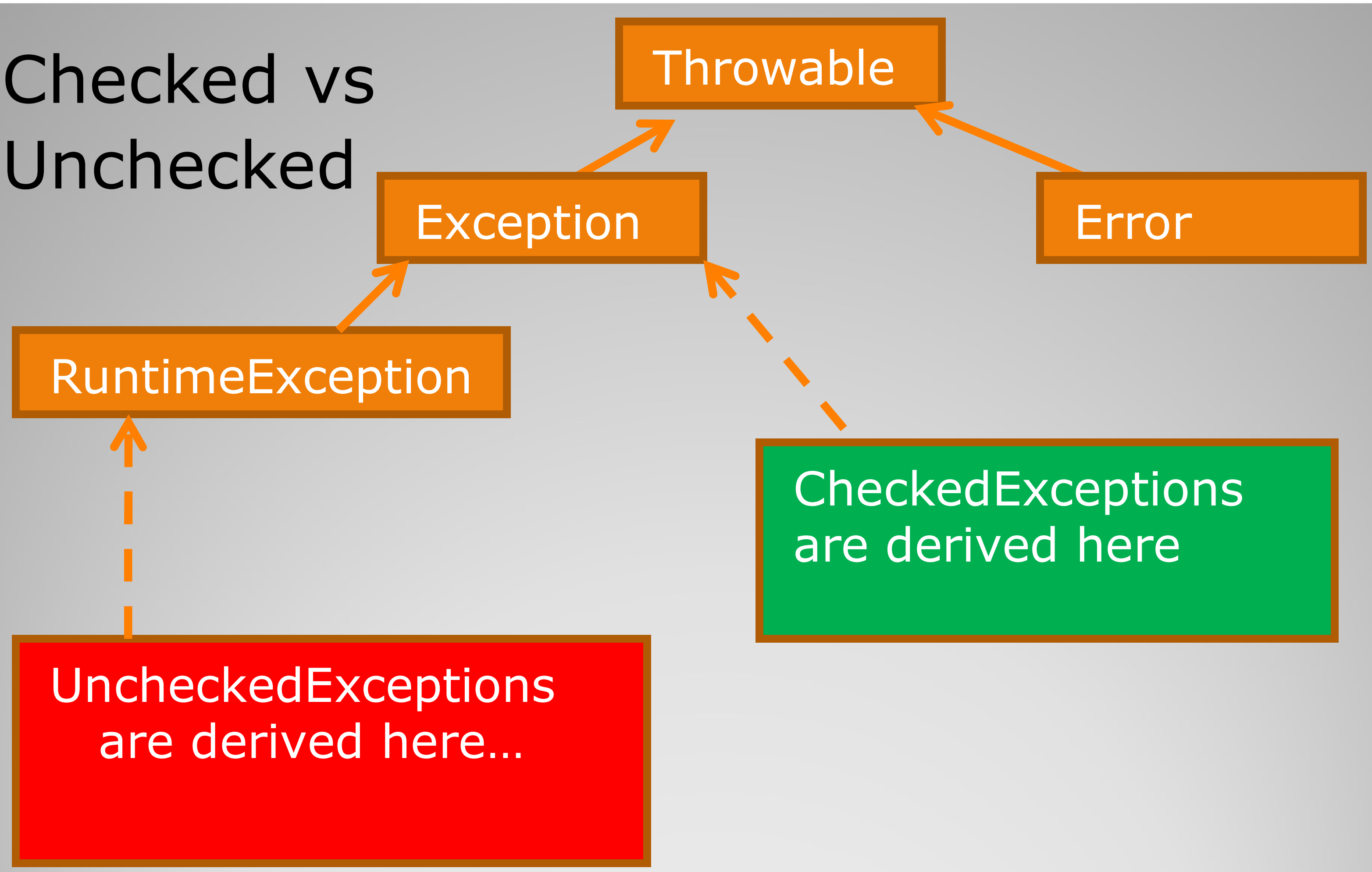
Checked Vs Unchecked Exceptions

Checked Vs Unchecked Exceptions

- **Unchecked exception** – A method that throws an unchecked exception has no restrictions for calling it.
- Unchecked exceptions are derived from the RuntimeException class.
- Checked exceptions are derived from the Exception class but NOT the RuntimeException class.

Java Exception Hierarchy

Checked vs Unchecked


Throwable
Exception
Error
RuntimeException
CheckedExceptions are derived here
UncheckedExceptions are derived here…

- ArithmeticException is an unchecked exception.
- IOException is a checked exception.
- Checked exceptions are typically caused by conditions that are out of the control of the program.
  - For example, cannot open a file.

Checked Vs Unchecked Exceptions

- Some exceptions are derived from other exceptions.
- A try statement can catch both a base class exception and an exception derived from that base class.
- For example, ArithmeticException is derived from Exception (though not directly).

Catch Block Order

```
public static int quotient( int numerator, int denominator )
{         return numerator / denominator;  }

public static void main(String[] args) {
          try {
                    Scanner scanner  = new Scanner(System.in);
                    int numerator = scanner.nextInt();
                    int denominator = scanner.nextInt();

                    int result = quotient(numerator, denominator);
          }
          catch (ArithmeticException ae) {
                    System.err.println("Error – Divide by zero.");
          }
          catch (Exception e) {
                    System.err.println("Generic error handling");
          }
}
```

Catches both Arithmetic Exception and Exception.
What happens?

- Use the first matching catch block that is compatible with the thrown exception.
- So the ArithmeticException catch will run in the previous example.
- What happens if we change the order. For example…

Catch Block Order

```
public static int quotient( int numerator, int denominator )
{         return numerator / denominator;  }

public static void main(String[] args) {
          try {
                    Scanner scanner  = new Scanner(System.in);
                    int numerator = scanner.nextInt();
                    int denominator = scanner.nextInt();

                    int result = quotient(numerator, denominator);
          }
          catch (Exception e) {
                    System.err.println("Generic error handling");
          }
          catch (ArithmeticException ae) {
                    System.err.println("Error – Divide by zero.");
          }
}
```

Exception comes first in this example.
What happens?

- Compile error. Compiler will not let this happen.
- ArithmeticException is a type of Exception so it is compatible with the Exception catch block.
- ***The ArithmeticException catch block would be unreachable if this were allowed.***
- You must be careful about the order of the catch blocks.

Catch Block Order

finally Block

- **finally** blocks are used for "clean-up" code.
- Some resources need to be released or "cleaned-up" when they are no longer needed.
- finally blocks are used for releasing resources that are no longer needed.

```
try
{
          // Code that throws exceptions
}
catch (…) {
          // Catch code
}
catch (…) {
          // Other catch code
}
finally
{
          // Finally code goes here
          // This is for resource clean-up
}
```

When does the finally block execute?

finally Block

finally Block

- finally block runs after try and catch are run.

- Java ***guarantees*** (kind of) that the finally block will execute whether or not an exception is thrown in the try block.

- finally block runs if:
  - try block exits by normally reaching its ending curly brace.
  - try block exits by using return, break, continue.
  - Exception is thrown within the try block and caught.
  - Exception is thrown but not caught.

- finally block does NOT run if:
  - try block exits by a call to System.exit (terminates JVM).

System.exit

- **System.exit(0)** immediately terminates the JVM which ends the program.
- The finally block does NOT run when System.exit executes.

```
try {
   int num1, num2, quotient;
   num1 = 10;
   num2 = 0;
   System.exit(0);
   quotient = Divide(num1, num2);
}
catch (Exception e)
{
   System.err.println("Handled exception");
}
```

System.exit(0) immediately terminates the JVM. The finally block does not run.

- Throwing an exception starts the exception process.
- It is like throwing a ball (the exception is the ball).
- When throwing a ball, the ball starts moving when someone throws it.

throw statement

- Use the **throw statement** to actually throw an exception.

Throwing vs Exceptions

- Catching an exception is fundamentally different than throwing.
- The person catching the ball will receive the ball the thrower sent.

try/catch block

- Use a **try/catch block** to catch an exception.
- The try/catch block receives the exception that was thrown using the throw statement.

Throwing vs Exceptions

Creating and Throwing An Exception

- You can create methods that generate and throw exceptions.
- For example…

```
public static void main(String [] args) {
        try {
                int quotient;
                quotient = Divide(10, 0);
        }
        catch (Exception e)
        {
                System.err.println("Handled in main");
        }
}

public static int Divide(int num, int den) throws ArithmeticException {
        // If den is 0 then throw an exception
        if (den == 0) {
                ArithmeticException e;
                e = new ArithmeticException();
                throw e;
        }
        return num / den;
}
```

Catch exception. This try/catch block will catch the exception that is thrown by Divide.

Divide() can throw an ArithmeticException

Get the exception ready to throw by creating a new instance of the ArithmeticException class

Actually throw the exception

Throwing An Exception

Create Your Own Exception Class

- You can create own exception class and use it in programs just like the prewritten exception classes.
- Just create a class that is derived from Exception.
- You can put information specific to your exception in the class. For example, a "bad" value that a user entered can be stored in the class.
- We can create instances of MyException and throw them similar to what we did with ArithmeticException in the previous slide.

```
class MyException extends Exception {
   // Store the bad value that was entered in a variable
   private int badValue;
   // Other member variables and methods go here…
}
```

- Homework

- Final Exam

Important Dates

- In case it didn’t happen…
- Take attendance

Attendance