The Run-Time Stack

- A data structure maintained by the Java Runtime Environment.
- Keeps track of the methods that are currently running, and the data they are using.
- Each item on the stack is a stack frame, holding the following things:
  - Name of the stack frame’s method.
  - Space for the parameters of the stack frame’s method.
  - Space for local variables declared in the body of the stack frame’s method.
  - Space for the return value (if any) of the stack frame’s method.
  - The address of the method that called stack frame’s method.

Run Time Stack
package runtimestackdemo;
import acm.program.*;
import acm.graphics.*;
import java.awt.*;

public class RunTimeStackDemo extends GraphicsProgram {
    public static void main(String[] args) {
        new RunTimeStackDemo().start();
    }

    public void run() {
        Rectangle bounds = getBounds();
        double w = bounds.getWidth();
        double h = bounds.getHeight();
        drawMainHouse(w/4, h/4, w/4, h/4);
        drawGarage(w/2.5, 5*h/12, w/6, h/6);
    }

    private void drawMainHouse(double x, double y, double w, double h) {
        GRect mainDoor = new GRect(x+w/3, y+4*h/3, w/3, 2*h/3);
        add(mainDoor);
    }

    private void drawGarage(double x, double y, double w, double h) {
        GRect garageDoor = new GRect(x+w/6, y+4*h/3, 2*w/3, 2*h/3);
        add(garageDoor);
    }

    private void drawHouse(double x, double y, double w, double h) {
        GRect base = new GRect(x, y+h, w, h);
        GLine roofLeft = new GLine(x, y+h, x+w/2, y);
        GLine roofRight = new GLine(x+w, y+h, x+w/2, y);
        add(base);
        add(roofLeft);
        add(roofRight);
    }

    // . . . Omitted . . .
}

(Empty) Run Time Stack
run
this bounds w h

Run Time Stack

getBounds
this
run
this bounds w h

Run Time Stack

run
this bounds w h

Run Time Stack
**Run Time Stack**

`getWidth`  
`this`  
`run`  
`this`  
`bounds`  
`w`  
`h`  

**Run Time Stack**

`getHeight`  
`this`  
`run`  
`this`  
`bounds`  
`w`  
`h`
Run Time Stack

drawHouse  roofLeft  x  y
this  base  roofRight  w  h

drawMainHouse  x  y
this  mainDoor  w  h

run  this  bounds  w  h

Run Time Stack

add  gObject
this

drawHouse  roofLeft  x  y
this  base  roofRight  w  h

drawMainHouse  x  y
this  mainDoor  w  h

run  this  bounds  w  h

Run Time Stack
Run Time Stack

```plaintext
add gObject
this

drawHouse roofleft x y
this base rooRight w h
drawMainHouse x y
this mainDoor w h
run
this bounds w h
```

Run Time Stack

```plaintext
drawHouse roofleft x y
this base rooRight w h
drawMainHouse x y
this mainDoor w h
run
this bounds w h
```
```plaintext
<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Run Time Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>drawHouse</td>
<td>roofleft</td>
<td>this x y</td>
</tr>
<tr>
<td></td>
<td>base</td>
<td>this x y</td>
</tr>
<tr>
<td></td>
<td>roofRight</td>
<td>this x y</td>
</tr>
<tr>
<td>drawMainHouse</td>
<td>x y</td>
<td>run this x y h</td>
</tr>
<tr>
<td></td>
<td>mainDoor</td>
<td>run this x y h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>run this x y h</td>
</tr>
</tbody>
</table>
```

Run Time Stack
drawGarage
this
run
this

GRect
this
run
this

Run Time Stack
Run Time Stack

```plaintext
add gObject
this

drawGarage x y
this garageDoor w h
run
this bounds w h
```
Run Time Stack

```plaintext
drawGarage x y
this garageDoor w h
run
this bounds w h
```
Run Time Stack

```plaintext
drawHouse roofLeft x y
this base roofRight w h
drawGarage x y
this garageDoor w h
run
this bounds w h
```
GRect   x  y  w  h  retVal
this     w  h  
drawHouse roofLeft  x  y
this     x  y  w  h
base     w  h  
drawGarage roofRight  x  y
this     x  y  w  h
garageDoor w  h  
run       w  h  
this     w  h  
bounds   w  h  

Run Time Stack

GLine   x1  y1  x2  y2  retVal
this     x2  y2  
drawHouse roofLeft  x  y
this     x  y  w  h
base     w  h  
drawGarage roofRight  x  y
this     x  y  w  h
garageDoor w  h  
run       w  h  
this     w  h  
bounds   w  h  

Run Time Stack

GLine   x1  y1  x2  y2  retVal
this     x2  y2  
drawHouse roofLeft  x  y
this     x  y  w  h
base     w  h  
drawGarage roofRight  x  y
this     x  y  w  h
garageDoor w  h  
run       w  h  
this     w  h  
bounds   w  h  

Run Time Stack
add gObject
this
drawHouse  roofLeft  x  y
this  base  roofRight  w  h
drawGarage  x  y
this  garageDoor  w  h
run
this  bounds  w  h

Run Time Stack
```plaintext
run this drawHouse base x y w h
drawHouse roofLeft x y
roofRight w h

run this drawGarage x y
garageDoor w h

garageDoor bounds w h

garageDoor bounds w h
```

Run Time Stack
Run Time Stack

drawGarage
this garageDoor x y w h
run
this bounds w h

Run Time Stack

run
this bounds w h

Run Time Stack

(Empty) Run Time Stack
Averaging Program: Signaling Errors with Return Values

- User types in some numbers.
- Program reads the numbers.
- If user typed in at least one number, program prints out the average.
- If user typed in zero numbers, program prints out an error message.

```java
package averageretcode;
import java.io.*;
import java.text.*;
import java.util.StringTokenizer;

public class AverageReturnCode {
    public static void main(String[] args)
    throws IOException, ParseException {
        InputStreamReader inputStream = new InputStreamReader(System.in);
        BufferedReader inStream = new BufferedReader(inputStream);
        System.out.print("Enter Numbers: ");
        String currentLine = inStream.readLine();
        double avg = averageNumbers(currentLine);
        if (avg==-1.0) System.out.println("Average not defined.");
        else System.out.println("The average is: "+ avg);
    }

    // ... Omitted ...
}

private static double averageNumbers(String inputString) throws ParseException {
    StringTokenizer stringTokenizer = new StringTokenizer(inputString);
    NumberFormat numberFormatter = NumberFormat.getInstance();
    double total = 0.0;
    int count = 0;
    while (stringTokenizer.hasMoreTokens()) {
        Number currentNumber = numberFormatter.parse(stringTokenizer.nextToken());
        total += currentNumber.doubleValue();
        count++;
    }
    if (count==0)  return -1.0;
    else return total/count;
}
```

If the `averageNumbers` method returns `-1.0`, this means there are no numbers in the `String currentLine`. In this situation, the program prints out an error message.
Problems with Returning Error Codes

- If all return values are possible, no value can properly signal an error.
  - In the *Average* program, the number –1, can be an average.
  - If the user’s numbers average to –1, the program will report an error.
- Statements handling normal conditions are mixed with statements handling errors, resulting in a program that is hard to read.

General Form of *try*, *catch* and *finally* Blocks

```
try
  { <Statement1> … <StatementL> }
catch (<Exception-Class1> <Exception-Parameter1>)
  { <Statement1> … <StatementM1> }
  ...
catch (<Exception-ClassK> <Exception-ParameterK>)
  { <Statement1> … <StatementMK> }
finally
  { <Statement1> … <StatementN> }
```

Semantics of the *try*, *catch* and *finally* Blocks

- Execute the statements in the *try* block.
- If an exception described in a *catch* block is thrown during execution of the *try* block, then execute the *catch* block.
- Execute the *finally* block regardless of what happens during the *try* block.
Averaging Program: Signaling Errors with Exceptions

• The call to `averageNumbers` is placed within a `try` block.
• If the user enters no numbers, then:
  – The `averageNumbers` method throws an `AveragingException`.
  – The exception is caught by a `catch` block, which prints out an error message.
• Otherwise, `averageNumbers` returns the numerical average via its return value.

```java
package averageexceptions;
import acm.program.*;

public class AverageExceptions extends ConsoleProgram {
    public void run() {
        println("Number Averaging Program: ");
        try {
            double avg = averageNumbers();
            println("The average is: "+avg);
        } catch (AveragingException e) {
            println("Error!");
        }
    }

    public static void main(String[] args) {
        new AverageExceptions().start();
    }
}
```

```java
private double averageNumbers() throws AveragingException {
    int count = 0;
    double total = 0.0;
    do {
        String str = readLine("Enter a number: ");
        if (str.equals("")) break;
        total += new Double(str);
        count++;
    } while (true);
    if (count==0)
        throw new AveragingException();
    else return total/count;
}
```

```java
class AveragingException extends Exception {
}
```
Two Channels of Communication from Called Method to Calling Method

• One channel is the normal return value mechanism.
  – E.g. When `averageNumbers` returns a computed numerical average.
• The other channel is the throwing of an exception.
  – E.g., When `averageNumbers` throws an `AveragingException`.

ExceptionsDemo: Illustrating Transfer of Control

• The `run` method calls the `levelOne` method in a `try` block catching `RedException` and `GreenException`.
• The `levelOne` method calls the `levelTwo` method in a `try` block catching `BlueException` and `YellowException`.
• The `levelTwo` method randomly chooses to throw a `Red`, `Green`, `Blue`, `Yellow` or `Black` Exception.

```java
import java.util.Random;
public class ExceptionsDemo {
  public static void main(String[] args) throws BlackException {
    try {
      levelOne();
    } catch (RedException e) {
      System.out.println("Caught a RedException in run method.");
      System.out.println(e.getMessage());
    } catch (GreenException e) {
      System.out.println("Caught a GreenException in run method.");
      System.out.println(e.getMessage());
    } catch (BlueException e) {
      System.out.println("Caught a BlueException in run method.");
      System.out.println(e.getMessage());
    } catch (YellowException e) {
      System.out.println("Caught a YellowException in run method.");
      System.out.println(e.getMessage());
    } finally { System.out.println("Finishing up run method."); }
  }
// . . . Omitted . . . 
```

The `main` method declares that it may throw a `BlackException`.
The `levelOne` method declares that it may throw a `RedException`, `GreenException` or `BlackException`.

The `levelTwo` method declares that it may throw a `Red`, `Blue`, `Green`, `Yellow` or `Black` Exception.

`class RedException extends Exception {
    public RedException(String msg) { super("I'm a RedException. " + " " + msg); }
}

class BlueException extends Exception {
    public BlueException(String msg) { super("I'm a BlueException. " + " " + msg); }
}

class GreenException extends Exception {
    public GreenException(String msg) { super("I'm a GreenException. " + " " + msg); }
}

class YellowException extends Exception {
    public YellowException(String msg) { super("I'm a YellowException. " + " " + msg); }
}

class BlackException extends Exception {
    public BlackException(String msg) { super("I'm a BlackException. " + " " + msg); }
}

private static void levelOne() throws RedException, GreenException, BlackException {

    try {
        levelTwo();
    }
    catch (BlueException e) {
        System.out.println("Caught a BlueException in levelOne method.");
        System.out.println(e.getMessage());
    }
    catch (YellowException e) {
        System.out.println("Caught a YellowException in levelOne method.");
        System.out.println(e.getMessage());
    }
    finally {
        System.out.println("Finishing up levelOne method.");
    }
}

private static void levelTwo() {

    Random randomizer = new Random();
    int number = Math.abs(randomizer.nextInt()) % 5;
    String message = "I was thrown from levelTwo method.",
    switch (number) {
    case 0 : System.out.println("Throwing a RedException in levelTwo method.");
            throw new RedException(message);
    case 1 : System.out.println("Throwing a GreenException in levelTwo method.");
            throw new GreenException(message);
    case 2 : System.out.println("Throwing a BlueException in levelTwo method.");
            throw new BlueException(message);
    case 3 : System.out.println("Throwing a YellowException in levelTwo method.");
            throw new YellowException(message);
    case 4 : System.out.println("Throwing a BlackException in levelTwo method.");
            throw new BlackException(message);
    }
}
Scenario #1:

First `run` executes its `try` block. The associated `catch` and `finally` blocks are recorded in the stack frame.

```
run  args  Catch Red  Catch Green
      finally [...]
```

Next `levelOne` executes its `try` block. The associated `catch` and `finally` blocks are again recorded in the stack frame.

```
levelOne  Catch Blue  Catch Yellow
      finally [...]
run  args  Catch Red  Catch Green
      finally [...]
```

Now `levelTwo` throws a `BlueException`.

```
levelTwo  message  
      number  randomizer  
  levelOne  Catch Blue  Catch Yellow
      finally [...]
run  args  Catch Red  Catch Green
      finally [...]
```
Control transfers immediately to the BlueException catch block in levelOne. After the catch block is finished, the associated finally block in levelOne is executed, and levelOne returns.

When control transfers back to run, the try block of run finishes, the catch blocks of run are skipped, and the finally block of run is executed. Then the program exits.

> java ExceptionsDemo
Throwing a BlueException in levelTwo method.
Caught a BlueException in levelOne method.
I'm a BlueException. I was thrown from levelTwo method.
Finishing up levelOne method.
Finishing up run method.
Scenario #2:

First **run** executes its **try** block. The associated **catch** and **finally** blocks are recorded in the stack frame.

Next **levelOne** executes its **try** block. The associated **catch** and **finally** blocks are again recorded in the stack frame.

Now **levelTwo** throws a **RedException**.
Control transfers immediately to the **finally** block in `levelOne`, skipping the **catch** blocks in `levelOne`. After the **finally** block is finished, `levelOne` is immediately terminated.

```
> java ExceptionsDemo
Throwing a RedException in levelTwo method.
Finishing up levelOne method.
Caught a RedException in run method.
I'm a RedException. I was thrown from levelTwo method.
Finishing up run method.
```

Then control transfers immediately to the **RedException catch** block in `run`. After the **catch** block is finished, the associated **finally** block in `run` is executed, and the program exits.
Scenario #3:

First `run` executes its `try` block. The associated `catch` and `finally` blocks are recorded in the stack frame.

```plaintext
run  args  |  Catch Red  |  Catch Green
            finally (...)  
```

Run Time Stack

Next `levelOne` executes its `try` block. The associated `catch` and `finally` blocks are again recorded in the stack frame.

```plaintext
levelOne  Catch Blue  |  Catch Yellow
run  args  |  Catch Red  |  Catch Green
            finally (...)  
```

Run Time Stack

Now `levelTwo` throws a `BlackException`.

```plaintext
levelTwo  message
number  randomizer
levelOne  Catch Blue  |  Catch Yellow
run  args  |  Catch Red  |  Catch Green
            finally (...)  
```

Run Time Stack
Control transfers immediately to the `finally` block in `levelOne`, skipping the `catch` blocks in `levelOne`. After the `finally` block is finished, `levelOne` is immediately terminated.

```
levelOne  Catch Red  Catch Yellow
  finally (...)  
run  args  Catch Red  Catch Green
  finally (...)  
```

Run Time Stack

---

Then control transfers to the `finally` block in `run`, skipping the `catch` blocks in `run`. After the `finally` block is finished, `run` is immediately terminated, and the program exits. The Java Run Time Environment catches the `BlackException` and prints out its message.

```
run  args  Catch Red  Catch Green
  finally (...)  
```

Run Time Stack

---

> java ExceptionsDemo
Throwing a `BlackException` in `levelTwo` method.
Finishing up `levelOne` method.
Finishing up `run` method.
Exception in thread 'run' `BlackException`:
  I'm a `BlackException`. I was thrown from `levelTwo` method.
  at ExceptionsDemo.levelTwo(ExceptionsDemo.java:66)
  at ExceptionsDemo.levelOne(ExceptionsDemo.java:31)
  at ExceptionsDemo.run(ExceptionsDemo.java:9)
Guessing Game: Iterating try and catch Statements

- User repeatedly tries to guess an integer in the range 1..100.
- Put the code for reading the user’s input in a try block.
- Use `Integer.valueOf(…)` to process the user’s input.
- This method throws a `NumberFormatException` if the string does not represent an integer.
- Catch the exception and ask the user to enter his/her guess once again.
- Repeat the try and catch process until the user types in a valid integer.

```java
import java.io.*;
import java.util.Random;
import java.lang.NumberFormatException;
public class GuessingGameExceptions {  
    public static void main(String[] args) throws IOException  
    {  
        InputStreamReader inputStream = new InputStreamReader(System.in);
        BufferedReader inStream = new BufferedReader(inputStream);
        System.out.println("I am thinking of a number from 1 to 100.");
        System.out.print("Can you guess it?  "); System.out.flush();
        Random randomizer = new Random();
        int secretNumber = 1 + Math.abs(randomizer.nextInt()) % 100;
        int guess;
        do {guess = getGuess(inStream);  
            if (guess < secretNumber) System.out.print("Too low. Guess again: ");
            else if (guess > secretNumber) System.out.print("Too high. Guess again: ");
            else System.out.println("You got it!");
            System.out.flush();} while (guess != secretNumber);
        System.exit(0);  
    }  
    // . . . Omitted . . . 
}

private static int getGuess(BufferedReader inStream) throws IOException  
{  
    String line;
    int guess = 0;
    boolean success = false;
    do  
    {  
        try  
        {  
            line = inStream.readLine();
            guess = Integer.valueOf(line).intValue();
            success = true;
        }
        catch (NumberFormatException e)  
        {  
            System.out.print("Please enter an integer: ");
        }
    } while (!success);
    return guess;
}
```
Solving a Quadratic Equation Specified by Command Line Parameters

- User runs the solver program by typing: "java Quadratic <a> <b> <c>".
- The code for parsing the <a>, <b> and <c> parameters, and the code for solving the equation is placed in a try block.
- The code throws exceptions if parameters are missing, or if parameters don’t represent numbers, or if they don’t specify a valid quadratic equation.
- These exceptions are caught by catch blocks that print out error messages and exit.

```java
import java.lang.ArrayIndexOutOfBoundsException;
import java.lang.NumberFormatException;
import java.lang.ArithmeticException;

public class Quadratic
{
    static double a, b, c, r1, r2;

    public static void main(String[] args)
    {
        try
        {
            parseParameters(args);
            printEquation();
            findRoots();
            printRoots();
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Please supply three arguments.");
        }
        catch (NumberFormatException e)
        {
            System.out.println("Please supply numeric arguments.");
        }
        catch (ArithmeticException e)
        {
            System.out.println(e.getMessage());
        }
        // . . . Omitted . . .
    }

    private static void parseParameters(String[] args)
    {
        a = Double.valueOf(args[0]).doubleValue();
        b = Double.valueOf(args[1]).doubleValue();
        c = Double.valueOf(args[2]).doubleValue();
    }

    private static void printEquation()
    {
        System.out.println("Equation: " + a + "x^2 + " + b + "x + " + c + " = 0");
    }
}
```
private static void findRoots() {
    if (a==0.0) throw new ArithmeticException("Not a Quadratic Equation.");
    double discriminant = b*b - 4*a*c;
    if (discriminant<0.0) throw new ArithmeticException("Negative Discriminant.");
    r1 = root1(a,b,discriminant);
    r2 = root1(a,b,discriminant);
}
private static double root1(double a, double b, double discriminant) {
    return (-b + Math.sqrt(discriminant))/(2*a);
}
private static double root2(double a, double b, double discriminant) {
    return (-b - Math.sqrt(discriminant))/(2*a);
}
private static void printRoots() {
    System.out.println("Root 1 = " + r1 + "   Root 2 = " + r2);
}

**Catch or Declare**

• Suppose a method $M$ can throw an exception of class $E$.
  – Either directly with a throw statement.
  – Or indirectly by calling a method that can throw the exception.
• Java requires that the programmer do one of two things:
  – **Catch** the exceptions of class $E$ by placing the throwing code in a try block.
  – Declare that the method $M$ throws exception class $E$.
• There is an (er…) exception. This rule does not apply to:
  – ArithmeticException
  – NullPointerException
  – IndexOutOfBoundsException
  – NumberFormatException
  – and so-called “run time exceptions”.