

CMSC 202

Exceptions

Error Handling

- In the ideal world, all errors would occur when your code is compiled. That won't happen.
- Errors which occur when your code is running must be handled by some mechanism that allows the originator (detector) of the error to pass information to the recipient of the error who will know how to deal with the error.

Procedural Error Handling

- In procedural languages, error handling was usually by convention. Programmers simply agreed on some standard way to report errors.
- Usually a function returned a value which had to be checked by the caller
 - Not part of the language
 - Completely unenforceable
 - Programmers tended to ignore them
 - Did you even know that C's `printf()` has a return value?
 - Checking all return values would result in unreadable code

Bank Account Deposit Example

```
public static void main(String[] args) {  
    BankAccount account = new BankAccount(100.00);  
  
    Scanner input = new Scanner(System.in);  
    // get amount to deposit  
    System.out.println("Enter a Number: ");  
    int amt = input.nextInt();  
  
    if (amt > 0) {  
        account.deposit(amt);  
    }  
    else {  
        // reprompt the user?  
        // exit the system?  
        // user a default amount?  
    }  
}
```

Bank Account Deposit Example

In a banking application written in a procedural approach, we might write a function that verifies that the amount to deposit is acceptable. This function returns an int which must be interpreted by the calling code.

```
public static final int BAD_AMOUNT = 1;
public static final int OK_AMOUNT = -1;

private static int verifyAmount(double amount) {
    if (amount < 0) {
        return BAD_AMOUNT;
    }
    return OK_AMOUNT;
}
```

Errors By Convention

In main, the code calls the function and checks the return value

```
if (verifyAmount(amt) == BankAccount.OK_AMOUNT) {  
    account.deposit(amt);  
}  
else{  
    // reprompt the user?  
    // exit the sytem?  
    // user a default amount?  
}
```

```
public void deposit(double amount) {  
    this.balance += amount;  
}
```

Better Error Handling

- **Separation of error detection from error handling**
 - Class implementer detects the error
 - Class user decides what to do about the error
 - Exit the program
 - Output a message and continue the program
 - Retry the function that had the error
 - Ask the user what to do
 - Many other possibilities
 - Reduces complexity of code
 - Code that works when nothing unusual happens is separated from the code that handles **exceptional situations**
- Enforced by language

“Exceptional”

- What’s an exceptional situation?
- As defined in the Sun Java tutorial:
 - An **exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- The program encounters a situation it doesn’t know how to handle
- Different than a “normal problem”
 - Program has enough information to know what to do next

Exception Handling

- Removes error handling code from the code that caused the error
- Makes it possible to catch all kinds of errors, errors of a certain type, or errors of related types
- Is used in situations in which the system cannot recover.
- Is used when the error will be dealt with by a different part of the program (i.e., different scope) from that which detected the error
- Can be slow, but we don't care because errors occur very infrequently

Introduction to Exception Handling

- Java library software (or programmer-defined code) provides a mechanism that signals when something unusual happens
 - This is called *throwing an exception*
- In another place in the program, the programmer must provide code that deals with the exceptional case
 - This is called *handling the exception*

try-throw-catch

- The basic way of handling exceptions in Java consists of the *try-throw-catch* trio

try-throw-catch example

```
public void deposit(double amount) {  
    try{  
        if(amount < 0)  
            throw new Exception("Deposit Negative Amount");  
        this.balance += amount;  
    }  
    catch(Exception e){  
        // do something about the exception  
    }  
}
```

- This example is essentially a if-else expression, but to see the true separation of error detection from error handling we should look at how the try-throw-catch mechanism works.

The `try` block

- Code which calls a method that might throw an exception is placed inside a `try` block
 - The `try` block contains the code for the basic algorithm
 - It tells what to do when everything goes smoothly
 - It is called a `try` block because it "tries" to execute the case where all goes as planned
- Code which handles the exception is placed into a `catch` block
 - `Catch` block immediately follows the `try` block

Better Bank Account Deposit Code

```
public static void main(String[] args) {  
  
    BankAccount account = new BankAccount(100.00);  
    try{  
        Scanner input = new Scanner(System.in);  
        // get amount to deposit  
        System.out.println("Enter a Number: ");  
        double amt = input.nextDouble();  
        account.deposit(amt);  
    }  
    catch (Exception e) {  
        // do something with the exception  
    }  
}
```

try-throw-catch Mechanism

- When an exception is thrown, the execution of the surrounding **try** block is stopped
 - Normally, the flow of control is transferred to another portion of code known as the **catch** block
- The value thrown is the argument to the **throw** operator, and is always an object of some exception class
 - The execution of a **throw** statement is called *throwing an exception*

try-throw-catch Mechanism

- When an exception is thrown, the **catch** block begins execution
 - The **catch** block has one parameter
 - The exception object thrown is plugged in for the **catch** block parameter
- The execution of the **catch** block is called *catching the exception*, or *handling the exception*
 - Whenever an exception is thrown, it should ultimately be handled (or caught) by some **catch** block

try-throw-catch Mechanism

```
catch( Exception e ) { . . . }
```

- The identifier **e** in the above **catch** block heading is called the **catch** block parameter
- The **catch** block parameter does two things:
 1. It specifies the type of thrown exception object that the **catch** block can catch (e.g., an **Exception** class object above)
 2. It provides a name (for the thrown object that is caught) on which it can operate in the **catch** block
 - Note: The identifier **e** is often used by convention, but any non-keyword identifier can be used

try-throw-catch Mechanism

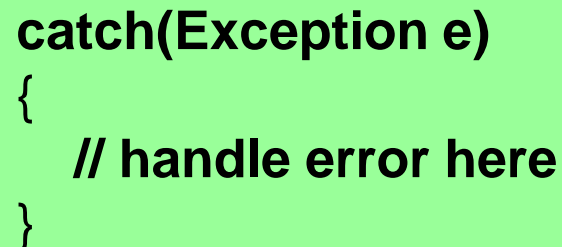
- When a **try** block is executed, two things can happen:
 1. No exception is thrown in the **try** block
 - The code in the **try** block is executed to the end of the block
 - The **catch** block(s) is (are) skipped
 - The execution continues with the code placed after the **catch** block(s)

try-catch Control Flow

```
try  
{  
    // code that might throw an Exception  
    // more code  
}
```



```
catch(Exception e)  
{  
    // handle error here  
}
```



```
// Method continues here
```



Case 1

The **try** block does **NOT** throw an Exception.

When the **try** block completes, the **catch** block is skipped

try-throw-catch Mechanism

2. An exception is thrown in the **try** block and caught in the **catch** block
 - The rest of the code in the **try** block is skipped
 - Control is transferred to a following **catch** block (in simple cases)
 - The thrown object is plugged in for the **catch** block parameter
 - The code in the **catch** block is executed
 - The code that follows that **catch** block is executed (if any)

try-catch Control Flow

```
try  
{  
    throw new Exception( "message");  
    // more code  
}
```

```
catch(Exception e)  
{  
    // handle error here  
}
```

```
// Method continues here
```

Case 2

The **try** block
throws an Exception.

The **try** block terminates,
the **catch** block executes,
code following the **catch**
block executes

Exception Classes

- The Java language defines a basic **Exception** class
 - There are more exception classes in the standard Java libraries
 - New exception classes can be defined like any other class
- All predefined exception classes have the following properties:
 - There is a constructor that takes a single argument of type **String**
 - The class has an accessor method **getMessage** that can recover the string given as an argument to the constructor when the exception object was created
- All programmer-defined classes should have the same properties

Exception Classes from Standard Packages

- The predefined exception class `Exception` is the root class for all exceptions
 - Every exception class is a descendent class of the class `Exception`
 - Although the `Exception` class can be used directly in a class or program, it is most often used to define a derived class
 - The class `Exception` is in the `java.lang` package, and so requires no `import` statement

Using the `getMessage` Method

```
. . . // method code
try
{
    . . .
    throw new Exception( StringArgument );
    . . .
}
catch(Exception e)
{
    String message = e.getMessage();
    System.out.println(message);
    System.exit(0);
} . . .
```


Using the `getMessage` Method

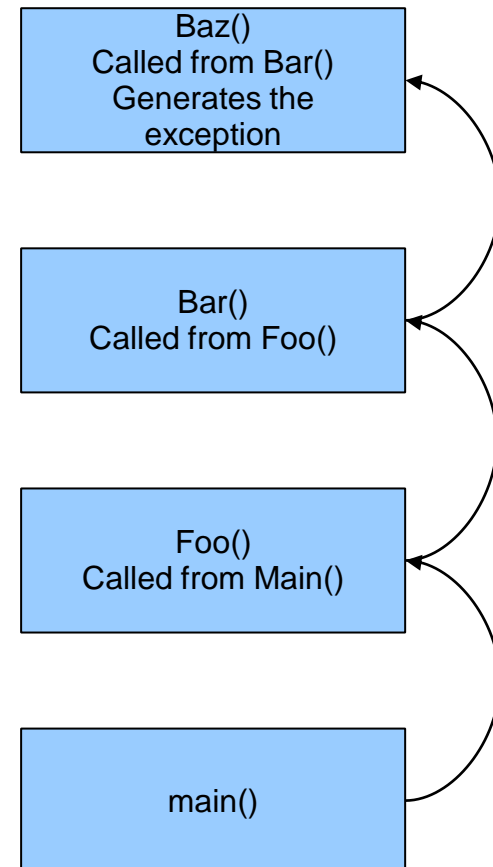
- Every exception has a `String` instance variable that contains some message
 - This string typically identifies the reason for the exception
- In the previous example, `StringArgument` is an argument to the `Exception` constructor
- This is the string used for the value of the `String` instance variable of exception `e`
 - Therefore, the method call `e.getMessage()` returns this string

Where does it start?

Regular modular programming will cause the call stack to grow.

Each frame of the stack will execute a series of statements that will eventually lead to the program to terminate successfully.

Any time an Exception is generated the series of operations that would normally occur in sequential ordering is halted at that statement.



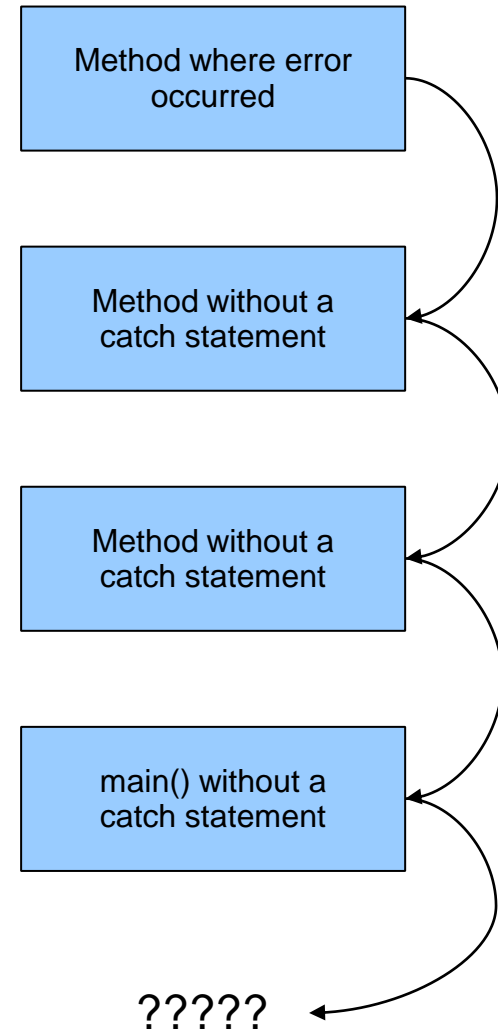
Why does my program crash?

The exception will be generated in a method.

Throw will throw the Exception Object down the stack to a catch block.

When no catch block is present in the previous stack frame, it thrown down until it is caught.

If nothing catches that exception, your program will crash.



Why does my program crash?

If every method up to and including the main method simply includes a **throw** an exception, that exception may be thrown but never caught

In a GUI program (i.e., a program with a windowing interface), nothing happens - but the user may be left in an unexplained situation, and the program may be no longer be reliable

In non-GUI programs, this causes the program to terminate with an error message giving the name of the exception class

Every well-written program should eventually catch every exception by a **catch** block in some method