# 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 either:
  A. handled by the originator (detector) of the error, who doesn't have enough context to know what response is appropriate; OR
  B. managed through be some mechanism that allows the detector to pass information back to the recipient of the error, who will know how to deal with the error.

# Error Handling via Return Values

- In languages like C, 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 `printf( )` has a return value?
  - Checking all return values would result in unreadable code

# A Tradition-style Example

```
public class BankAccount {
   public static final in DEPOSIT_NEGATIVE = -1;
   public static final in DEPOSIT_OK = 0;

   private int balance = 0;

   public BankAccount( int minDeposit, int maxWithDraw ) {
       this.minDeposit = minDeposit;
       this.maxWithDraw = maxWithDraw;
   }

   public int deposit( int amt ) {
       if (amt < 0 ) return DEPOSIT_NEGATIVE;
       balance += amt;
       return DEPOSIT_OK;
   }

   // the rest of the BankAccount class follows
```

# Bank Account Deposit – cont'd

```
public class DepositExample {
   public static void main( String[ ] args ) {
        BankAccount myAccount = new BankAccount( 100, 10000 );
        … // Add code to read user's deposit request into "amt"
        int status = myAccount.deposit( amt );
        if (status == BankAccount.DEPOSIT_NEGATIVE)
        {
           // do something appropriate
        } else  // status == BankAccount.DEPOSIT_OK
           // do the good stuff
        }
    }
}
```

# Issues with Traditional Error Handling

- What if we also wanted to check values that were too large?
- What about new error types that we only run into later?
- What if we need to pass back complex information about the exact nature of the error?
- How to handle growing complexity and confusion of code?

# 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

# Better Error Handling

- **Separation of error detection from error handling (cont.)**
  - Language provides mechanism to automate facilitate error-handling, and enforces standardized communication and flow control between detector and handler
  - Reduces complexity of code
    - Code that works when nothing unusual happens is separated from the code that handles **exceptional situations**

# “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

- Programmer-defined code (or Java library software) utilizes some mechanism to signal 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

```
Scanner in = new Scanner( System.in);
System.out.println( "Enter Amt: ");

try {
   int amt = in.nextInt( );
   if (amt < 0)
        throw new Exception( "Negative value was input");
   System.out.println ("Thanks for your deposit");
   // addl. code that can assume the amt is not negative
}

catch( Exception e )
{
   // do something about the exception
}

System.out.println( "Have a nice day");
```

# The try & catch blocks

- Code which 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 is separate from, and immediately follows, the **`try`** block

# Non-local Throws

- The **`throw`** statement does not have to be lexically directly inside the body of a **`try`** block
  - If not in a **`try`** block, the **`throw`** propagates out of the method into the calling method
  - Can pop up out of multiple nested method invocations
  - Similar to a “non-local goto”

# throw Example

Code inside a class method detects an unrecoverable error and throws an exception object.

```
public class BankAccount {
    private int balance;
    public BankAccount( ) {
          balance = 0;
    }

    public int getBalance( ) { return balance; }

    // precondition – amount must be nonnegative
    // throws an exception if amount is negative
    // postcondition – balance updated
    public int deposit( int amt ) {
          if (amt < 0 )
                    throw new Exception("Deposit is Negative" );
          balance += deposit;
    }
```

# Original Bank Account Deposit Code

```
public class DepositExample {
    public static void main( String[ ] args ) {
        BankAccount myAccount = new BankAccount( 100, 10000 );

        … // Add code to read user's deposit request into "amt"

        int status = myAccount.deposit( amt );
        if (status == BankAccount.DEPOSIT_NEGATIVE) {
            // code that "handles" a negative deposit
            System.out.println( "Error: Deposit is negative" );
        }
        else { // status == BankAccount.DEPOSIT_OK
           // do the good stuff
            System.out.println( "New Balance = " +
                                    myAccount.getBalance());
        }
        System.out.println( "GoodBye" );
    }
}
```

# Better Bank Account Deposit Code

```
public class DepositExample {
    public static void main( String[ ] args ) {
        BankAccount myAccount = new BankAccount( 100, 10000 );

        … // Add code to read user's deposit request into "amt"

        try {
            myAccount.deposit( amt );
            System.out.println( "New Balance = " +
                                        myAccount.getBalance());
        }
        catch (Exception e)
        {
            // code that "handles" a negative deposit
            System.out.println( e.getMessage( ));
        }
        System.out.println( "GoodBye" );
}
```

# `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
  - If there is no surrounding `try` block, abort the method, and look for a `try` block in the caller
- 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

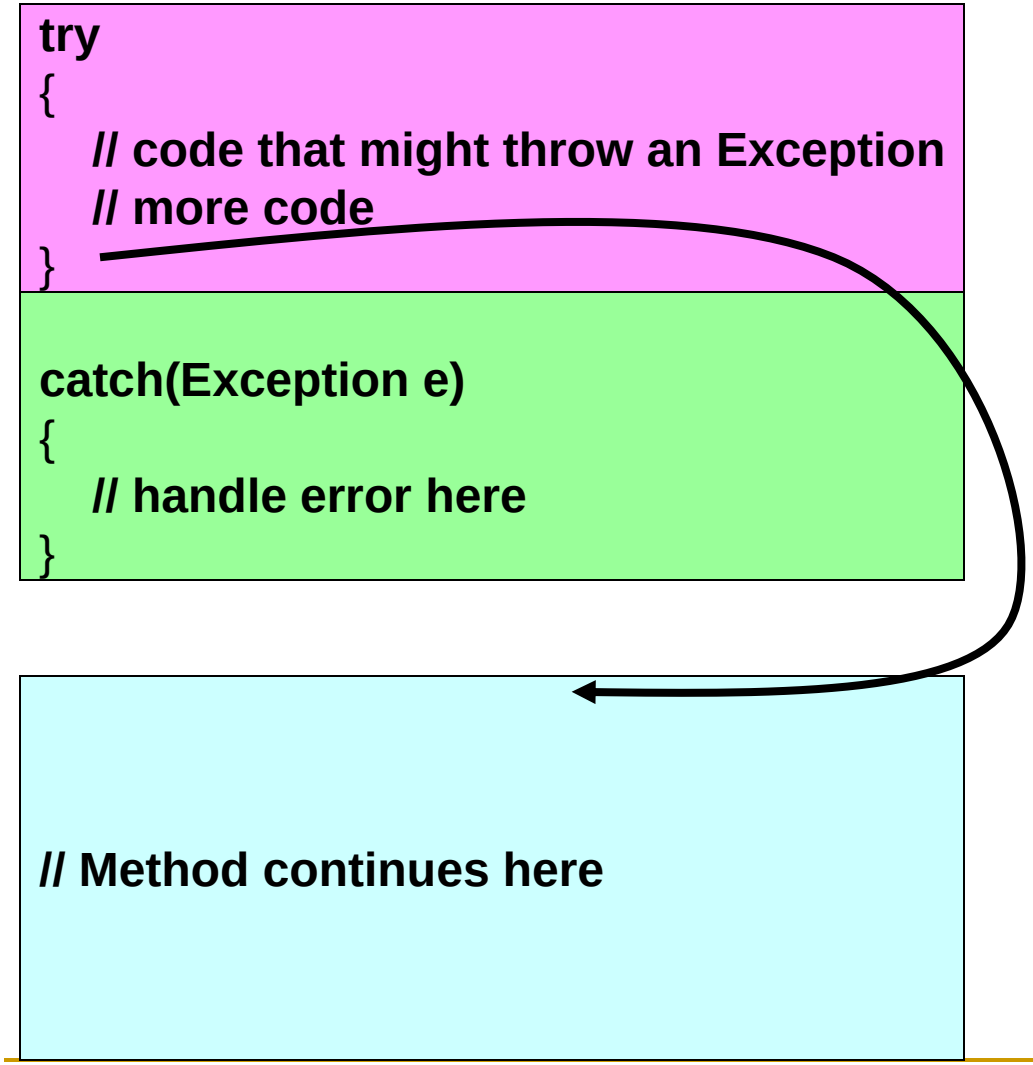
```
catch( Exception e ) { . . . }
```

- 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
- So, it is like an embedded method definition

# `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



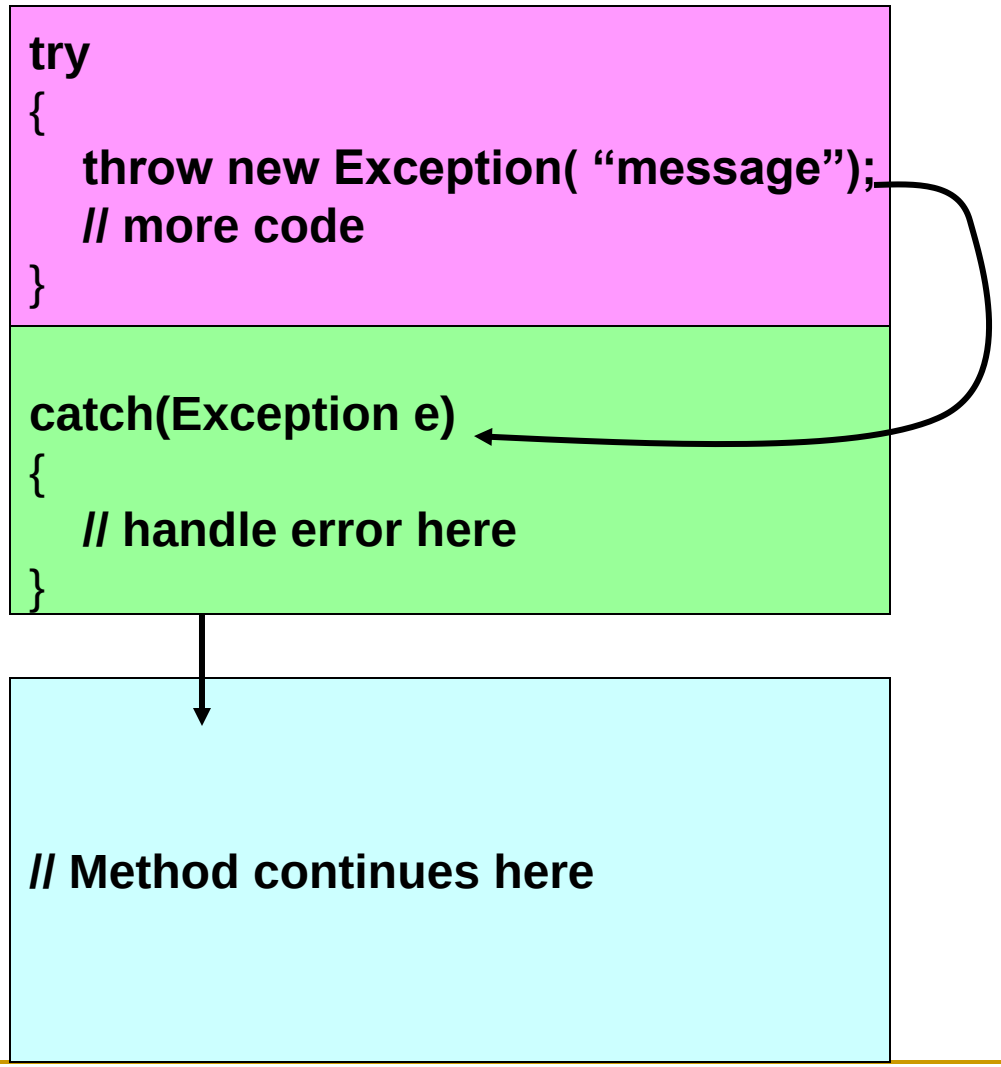

**Case 1**
The **try** block does **NOT throw** an Exception.

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




**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

# Defining Exception Classes

- A **`throw`** statement can throw an exception object of any exception class
- Instead of using a predefined class, exception classes can be programmer-defined
- Two main motivations for creating your own **`Exception`** subclass:
  - A different type of exception can be defined to identify each different exceptional situation
  - These can be tailored to carry the precise kinds of information needed in the **`catch`** block

# Defining Exception Classes

- Every exception class to be defined must be a derived class of some already defined exception class
  - It can be a derived class of any exception class in the standard Java libraries, or of any programmer defined exception class
- Constructors are the most important members to define in an exception class
  - They must behave appropriately with respect to the variables and methods inherited from the base class
  - Often, there are no other members, except those inherited from the base class
- The following exception class performs these basic tasks only

# Exception Object Characteristics

- The two most important things about an exception object are its type (exception class) and the message it carries
  - The message is sent along with the exception object as an instance variable
  - This message can be recovered with the accessor method `getMessage`, so that the catch block can use the message

# Preserve `getMessage`

- For all predefined exception classes, `getMessage` returns the string that is passed to its constructor as an argument
  - Or it will return a default string if no argument is used with the constructor
- This behavior must be preserved in all programmer-defined exception class
  - A constructor must be included having a string parameter whose body begins with a call to `super`
  - The call to `super` must use the parameter as its argument
  - A no-argument constructor must also be included whose body begins with a call to `super`
  - This call to `super` must use a default string as its argument

# Programmer-Defined Exception Class Guidelines

- Exception classes may be programmer-defined, but every such class must be a derived class of an already existing exception class
- The class **`Exception`** can be used as the base class, unless another exception class would be more suitable
- At least two constructors should be defined, sometimes more
- The exception class should allow for the fact that the method **`getMessage`** is inherited

# A Programmer-Defined Exception Class

**Display 9.3 A Programmer-Defined Exception Class**

```
1  public class DivisionByZeroException extends Exception
2  {
3      public DivisionByZeroException()
4      {
5          super("Division by Zero!");
6      }

7      public DivisionByZeroException(String message)
8      {
9          super(message);
10     }
11 }
```

*You can do more in an exception constructor, but this form is common.*

*super is an invocation of the constructor for the base class Exception.*

# Tip: An Exception Class Can Carry a Message of Any Type: int Message

- An exception class constructor can be defined that takes an argument of another type
  - It would stores its value in an instance variable
  - It would need to define accessor methods for this instance variable
- A programmer defined exception class may include any information that might be helpful to the recipient

# An Exception Class with an `int` Message

**Display 9.5 An Exception Class with an `int` Message**

```
1  public class BadNumberException extends Exception
2  {
3      private int badNumber;

4      public BadNumberException(int number)
5      {
6          super("BadNumberException");
7          badNumber = number;
8      }

9      public BadNumberException()
10     {
11         super("BadNumberException");
12     }

13     public BadNumberException(String message)
14     {
15         super(message);
16     }

17     public int getBadNumber()
18     {
19         return badNumber;
20     }
21 }
```

# Constructors and Exceptions

Up until now we've had no way to recover if a bad parameter was passed to a constructor. We usually just exited the program with System.exit(). A better way is to throw an exception

```
public class BankAccount {
   private int balance;
   public BankAccount( int startingBalance ) {
       if (startingBalance < 0)
          throw new BadNumberException( startingBalance );
       balance = startingBalance;
   }
}
```

# trying constructors

```
public class BankAccountDemo {
   public static void main( String[ ] args ) {
       BankAccount myAccount; // outside the try block???
        try {
            Scanner in = new Scanner( System.in );
            System.out.print( "Input starting balance: " );
            int startBalance = in.nextInt( );
            myAccount = new BankAccount(  startBalance );
            // more of the good stuff here
        }
        catch (BadNumberException bne )
        {
            // handle the bad input
            System.out.println("Deposits must be postive");
            Sysetem.out.println("You entered " +
                                              bne.getBadNumber());
        }
        System.out.println( "good bye ");
   }
}
```