Object Oriented Programming

1. Java 8 Interface Changes
2. OO Application Development
3. Assertions
4. Java Errors and Exceptions

Java 8 Interface Changes

- Prior to Java 8, interfaces can only declare methods (i.e., they can provide only abstract methods).
- To support lambda functions, Java 8 has introduced a big change to interfaces: you can now define default methods and static methods inside interfaces.

Java 8 Interfaces. Default methods

- Make it easy to evolve interfaces
- Before Java 8, if you add a new method in an existing interface, such an addition would break the classes implementing the interface since they will not have defined that method
- Default methods are to **add external functionality** to existing classes without changing their state.

Java 8 Interfaces. Default methods

- Example from http://javarevisited.blogspot.ro

```
interface Multiplication{
    int multiply(int a, int b);

    default int square(int a){
        return multiply(a, a);
    }
}
```

Java 8 Interfaces. Default methods

- Example from http://javarevisited.blogspot.ro

```
interface Multiplication{
    int multiply(int a, int b);

    default int square(int a){
        return multiply(a, a);
    }
}
```

- Any concrete classes of **`Multiplication`** interface only have to implement the abstract method **`multiply()`**
- The default method **`square()`** method can be used directly.

Java 8 Interfaces. Default methods

- Example from http://javarevisited.blogspot.ro

```
Multiplication product = new Multiplication(){
      @Override
      public int multiply(int x, int y){
      return x*y;
      }
};
int square = product.square(2);
int multiplication = product.multiply(2, 3);
```

- You can reduce a lot of boiler plate code by using lambda expression, which is also introduced on Java 8 (more later)

Java 8 Interfaces. Functional interfaces

- There are numerous interfaces in Java library that declare a single abstract method
- A functional interface specifies only one abstract method.
  - Since functional interfaces specify only one abstract method,they are sometimes known as Single Abstract Method (SAM) type or interface.
- It may have any number of default or static methods defined in it
- You can tag functional interface with @FunctionalInterface annotation

About annotations

- In its simplest form, an annotation looks like the following:
  - @Entity
- Annotations can be applied to declarations: declarations of classes, fields, methods, and other program elements.
  - When used on a declaration, each annotation often appears, by convention, on its own line

Annotations

- Java annotations are typically used for the following purposes:
  - Compiler instructions
  - Build-time instructions
  - Runtime instructions
- You can place Java annotations above classes, interfaces, methods, method parameters, fields and local variables

Annotations

- Three built-in annotations which are used to give the Java compiler instructions:
  - @Deprecated: used to mark a class, method or field as deprecated, meaning it should no longer be used
  - @Override: used above methods that override methods in a superclass
  - @SuppressWarnings makes the compiler suppress warnings for a given method
- More: https://docs.oracle.com/javase/tutorial/java/annotations

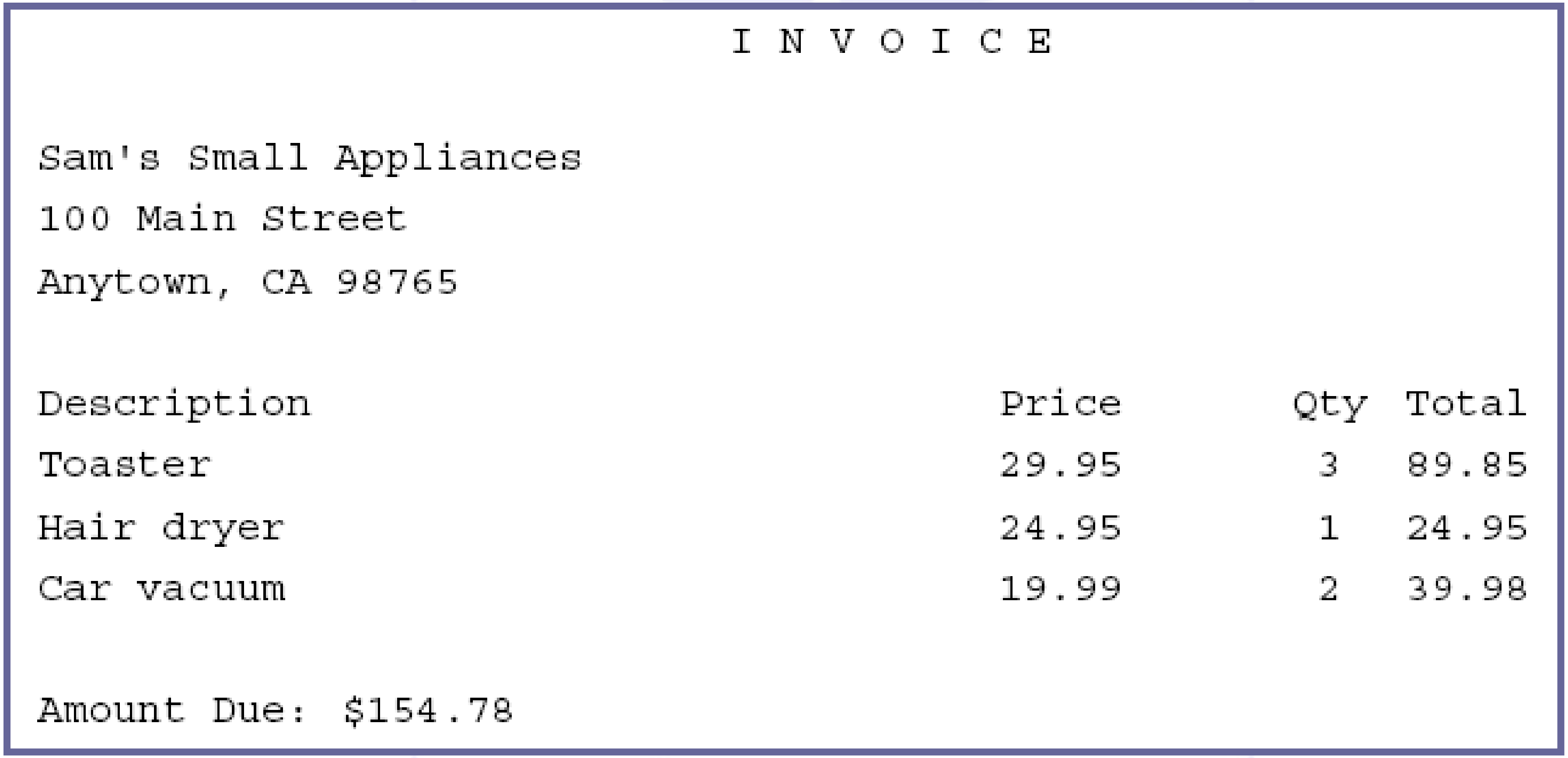
Five-Part Development Process

- Gather requirements
- Use CRC cards to find classes, responsibilities, and collaborators
- Use UML diagrams to record class relationships
- Use `javadoc` to document method behavior
- Implement your program

Analysis class rules of thumb

- About three to five responsibilities per class
- No class stands alone
- Beware of many very small classes
- Beware of few but very large classes
- Beware of “functoids” – a functoid is a really a normal procedural function disguised as a class.
- Beware of omnipotent classes
  - Look for classes with “system” or “controller” in their name!
- Avoid deep inheritance trees

Example: Simplified Invoice



```
                              I N V O I C E

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Description                            Price        Qty  Total
Toaster                                29.95          3  89.85
Hair dryer                             24.95          1  24.95
Car vacuum                             19.99          2  39.98

Amount Due: $154.78
```

Example: Simplified Invoice

- Classes that come to mind: `Invoice`, `LineItem`, and `Customer`
- Good idea to keep a list of candidate classes
- Brainstorm, simply put all ideas for classes onto the list
- You can cross not useful ones later

Finding Classes

- Keep the following points in mind:
  - Class represents set of objects with the same behavior
    - Entities with multiple occurrences in problem description are good candidates for objects
    - Find out what they have in common
    - Design classes to capture commonalities
  - Represent some entities as objects, others as primitive types
    - Should we make a class Address or use a String?
  - Not all classes can be discovered in analysis phase
  - Some classes may already exist

Printing an Invoice – Requirements

- Task: print out an invoice
- Invoice: describes the charges for a set of products in certain quantities
- Omit complexities
  - Dates, taxes, and invoice and customer numbers
- Print invoice
  - Billing address, all line items, amount due
- Line item
  - Description, unit price, quantity ordered, total price
- For simplicity, do not provide a user interface
- Test program: adds line items to the invoice and then prints it

Printing an Invoice – CRC Cards

- Discover classes
- Nouns are possible classes

Invoice
Address
LineItem
Product
Description
Price
Quantity
Total
Amount Due

Printing an Invoice – CRC Cards

- Analyze classes

```
Invoice
Address
LineItem    // Records the product and the quantity
Product
Description // Field of the Product class
Price       // Field of the Product class
Quantity    // Not an attribute of a Product
Total       // Computed-not stored anywhere
Amount Due  // Computed-not stored anywhere
```

- Classes after a process of elimination

Invoice
Address
LineItem
Product

Very GOOD Examples

- Address Book
  - http://www.math-cs.gordon.edu/courses/cs211/AddressBookExample/
- Automatic Teller Machine
  - http://www.math-cs.gordon.edu/courses/cs211/ATMExample/

Reasons for rejecting a candidate class

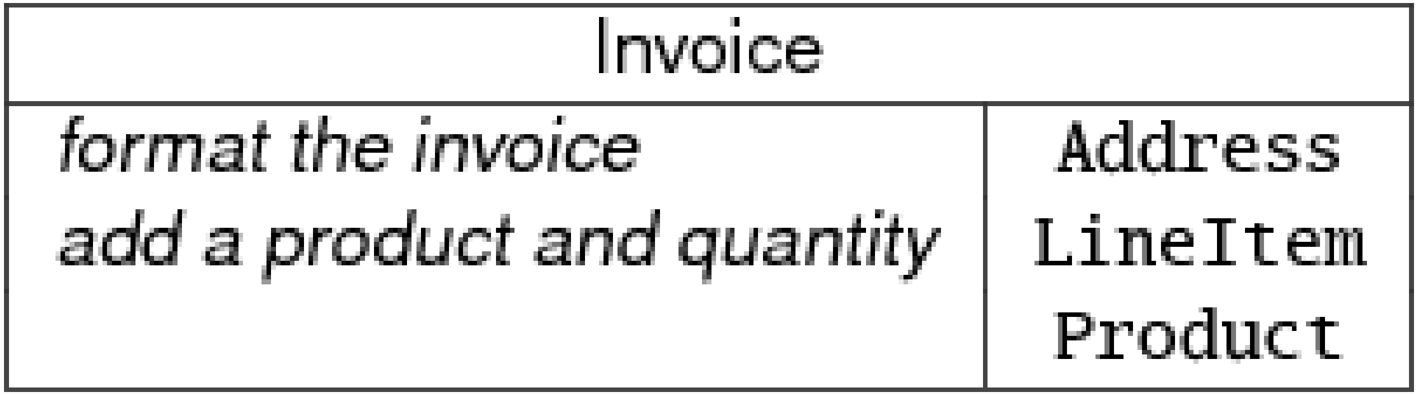
Sign	Reason for suspicion
Class with verbal name (***infinitive or imperative***)	May be a simple subroutine, not a class
Fully effective class with only one method	May be a simple subroutine, not a class
Class described as "performing" something	May not be a proper data abstraction
Class with no methods	May be an opaque piece of information, not an ADT. Or may be an ADT, the routines having just been missed
Class introducing no or very few features (***but inherits features from parents***)	May be a case of "taxomania"
Class covering several abstractions	Should be split into several classes, one per abstraction

CRC Cards for Printing Invoice

- Both `Invoice` and `Address` must be able to format themselves – responsibilities:
  - `Invoice` *format the invoice* and
  - `Address` *format the address*
- Add collaborators to invoice card: `Address` and `LineItem`
- For `Product` card – responsibilities: *get description, get unit price*
- For `LineItem` CRC card – responsibilities:*format the item, get the total price*

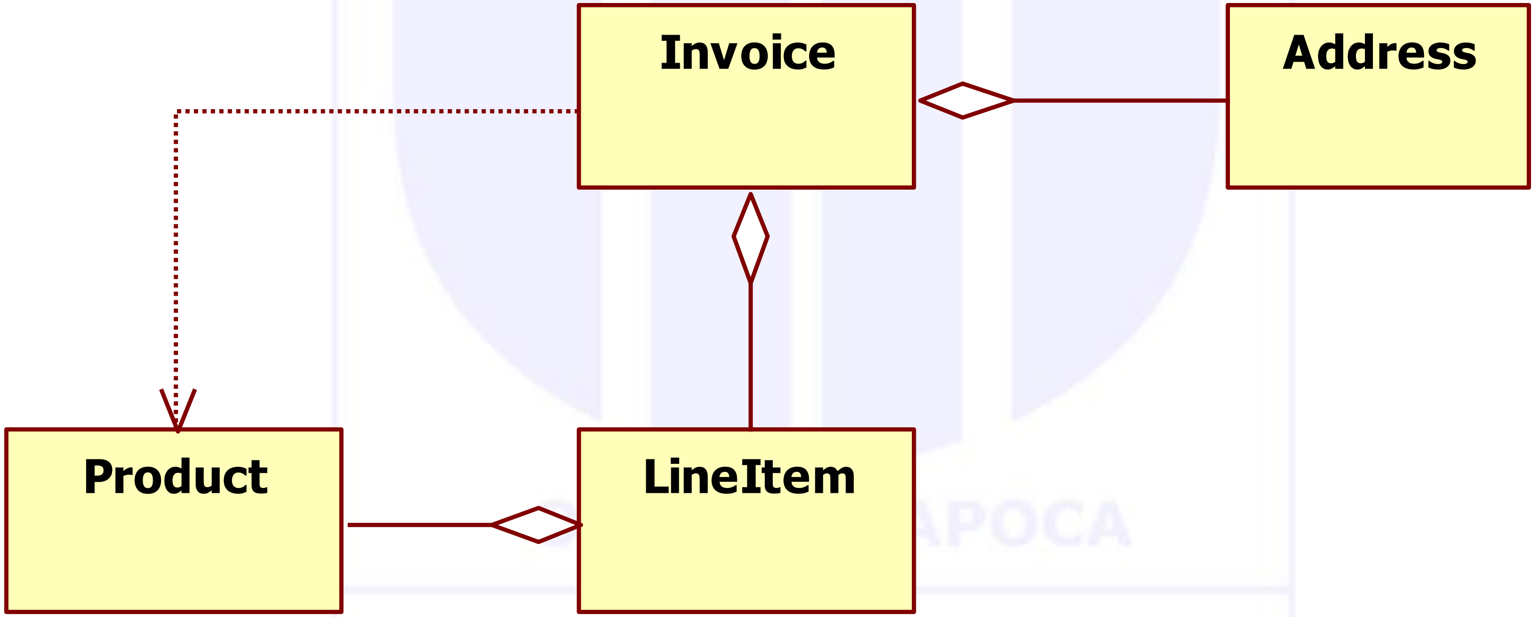
CRC Cards for Printing Invoice

- **`Invoice`** must be populated with products and quantities:



Invoice	
format the invoice	`Address`
add a product and quantity	`LineItem`
	`Product`

Printing an Invoice – UML Diagram



Tools for UML Diagraming

- WhiteStarUML
  - https://sourceforge.net/projects/whitestaruml/
  - open source project
- UML designer (integrates in Eclipse IDE)
  - http://www.umldesigner.org/download/
- Modelio UML
  - https://sourceforge.net/projects/modeliouml/?source=typ_redirect
- List of UML Tools
  - http://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

Types of Specifications

- Class Diagrams
- Object Diagrams
- Activity Diagrams (control flow diagrams)
- Assertions (preconditions, postconditions, invariants)
  - Others
- Note that first three are incomplete specifications

Class Specification

- A software specification indicates the task (or some aspect of the task) that is supposed to be performed when software executes
- A class specification defines the semantics (behavior) of a class by way of:
  - a class invariant to describe what is always true of the class's objects.
  - specifications for each of the classes methods.
- Each *method specification* consists of
  - a precondition (optional),
  - a modifies clause (optional), and
  - a postcondition.

Method Specification

- A *precondition* states the conditions that are necessary for the method to properly execute
- A *modifies* clause is a list of objects that might be altered by executing the method.
- A *postcondition* states what is true when the method completes execution

Assertion

- An *assertion* is a statement of fact that is presumed true relative to a code location(s). Example

```
// assert: str is a String  and  str.length > 2
char firstChar, secondChar, bigChar;
firstChar = str.charAt(0);
secondChar = str.charAt(1);
if (firstChar > secondChar)   {
  bigChar = firstChar;
} else {
  bigChar = secondChar;
}
/* assert:
  str.length > 2
  and (str.charAt(0) > str.charAt(1)
            implies bigChar == str.charAt(0))
  and (str.charAt(0) ≤ str.charAt(1)
            implies bigChar == str.charAt(1)) */
```

Assertion Notation

- Assertions are based on logic and certain program notations (i.e., variable references and possibly non-void method calls).
- Assertions should NOT contain action verbs
- Logical Operators

  not *SubAssertion1* - The subassertion must be false.

  SubAssertion1 ***and*** *SubAssertion2* - Both subassertions must be true.

  SubAssertion1 ***or*** *SubAssertion2* - One or both subassertion is true.

  SubAssertion1 ***implies*** *SubAssertion2* - When the first subassertion is true, the second must also be true

Assertion Notation

- Another logical notation, known as quantification, permits expressing assertions about data structures.
- Universal quantification
  - ***forAll*** (*type var : boundaryCondition* | *SubAssertion*)
  - Example:

forAll (Integer j : 0≤j≤2 | arr1[j] > 0)

meaning: arr1[0] > 0 ***and*** arr1[1] > 0 ***and*** arr1[2] > 0

Assertion Notation

- Existential quantification
  - ***exists*** *(type var : boundaryCondition | SubAssertion)*
  - Example:

exists (Integer j : 0≤j≤2 | arr1[j] == 5)

meaning: arr1[0] ==5 ***or*** arr1[1] == 5 ***or*** arr1[2] == 5

Quantification Examples

- Assume two arrays of double: **`a1`** and **`a2`** and **`a1.length == a2.length == 4`**

```
forAll (Integer r : 0 ≤ r < 3 | a1[r] < a1[r+1] )
forAll (Integer w : 0 ≤ w ≤ 3 | a1[w] == a2[w] )
exists (Integer k : 0 ≤ k ≤ 3
  | a1[k] == 22 and a2[k] == 22 )
exists (Integer k : 0 ≤ k ≤ 3
  | ( a1[k] < 0
     and forAll (Integer j : k < j ≤ 3 | a2[k] == a1[j]) ) )
forAll (j,k : 0 ≤ j,k ≤ 3 and j != k | a1[j] != a2[k] )
```

Where to Place Assertions

- Possible places
  - Class invariant
  - Method postcondition
  - Method precondition
  - Loop invariant

Assertions Example

```
/** class invariant
  distanceInMiles > 0  and  timeInSeconds > 0 */
public class LapTime   {
    private double distanceInMiles, timeInSeconds;

    /** pre:  d > 0  and  t > 0
        post: distanceInMiles == d and timeInSeconds == t  */
    public LapTime(double d, double t)   {
        distanceInMiles = d;
        timeInSeconds = t;
    }

    /** post:  distanceInMiles == 60
          and   timeInSeconds == 3600 */
    public void setTo60MPH()  {
       distanceInMiles = 60;
       timeInSeconds = 3600;
    }
}
```

Special Postcondition Notations

- Return value (result)

```
// Within LapTime class
/** post: result == distanceInMiles / (timeInSeconds*3600)
 */
public double milesPerHour()   {
    double velocity;
    velocity = distanceInMiles/(timeInSeconds*60*60);
    return velocity
}
```

- Previous value (@pre)

```
// Within LapTime class
/** post:  distanceInMiles == distanceInMiles@pre * 2 */
public void doubleTheMileage()   {
    distanceInMiles = distanceInMiles * 2;
}
```

Design by Contract

- Method caller guarantees…
  - precondition & class invariant (at time of method call)
- Method is required to ensure…
  - postcondition & class invariant (at time of method return)
- Addendum: A modifies clause can stipulate what alterations are permitted

Problems During Execution

- A program often encounters problems as it executes.
  - It may have trouble reading data,
  - there might be illegal characters in the data, or
  - an array index might go out of bounds.
- Java Errors and Exceptions enable the programmer deal with such problems.
  - You can write a program that recovers from errors and keeps on running.
  - *A program should not crash when the user makes an error*!
- Input and output is especially error prone.
- Exception handling is essential for I/O programming

Exception Example

- The program:

```
import java.util.Scanner;
public class InputMismatchExceptionDemo {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter one integer:");
        int inputNumber = keyboard.nextInt();
        System.out.println("The square of  " + inputNumber + " is "
  + inputNumber * inputNumber);
    }
}
```

- With input: `Enter one integer:h1`
- Results in:

```
java.util.InputMismatchException
  at java.util.Scanner.throwFor(Scanner.java:819)
  at java.util.Scanner.next(Scanner.java:1431)
  at java.util.Scanner.nextInt(Scanner.java:2040)
  at java.util.Scanner.nextInt(Scanner.java:2000)
  at
  InputMismatchExceptionDemo.main(InputMismatchExceptionDemo.java:11
  )
```

Example Discussion

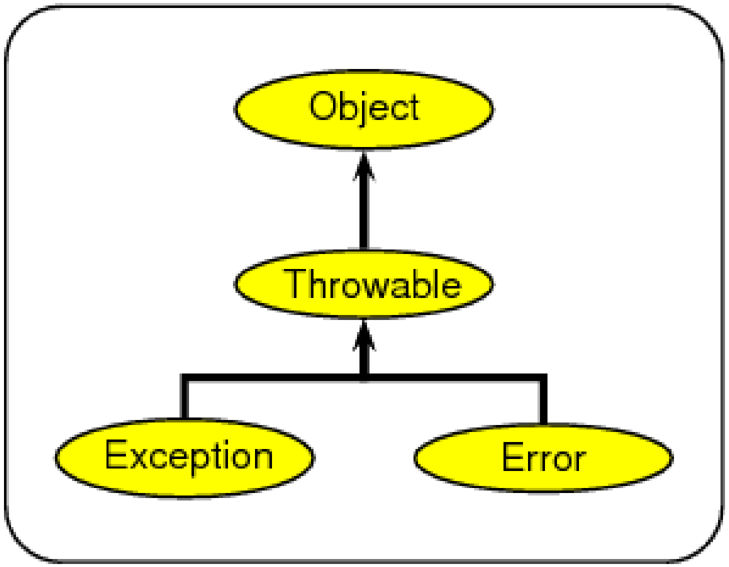
- Nothing is wrong with the program.
  - The problem is that `nextInt` cannot convert "**`h1`**" into an **`int`**.
  - When `nextInt` found the problem it **threw** a `InputMismatchException`.
  - The Java run-time system caught the exception, halted the program, and printed the error messages

Exceptions and Errors

- An **exception:** a problem that occurs when a program is running.
  - When an exception occurs, the JVM creates an object of class **`Exception`** which holds information about the problem.
  - A Java program itself may **catch** an exception. It can then use the Exception object to recover from the problem.
- An **error**, also, is a problem that occurs when a program is running.
- An error is represented by an object of class **`Error`**.
  - But an error is too severe for a program to handle. The program must *stop running*.

Throwable Hierarchy

- Class **`Exception`** and class **`Error`** both descend from **`Throwable`**.
  - A Java method can "throw" an object of class **`Throwable`**.
  - E.g. Integer.parseInt("zzz") throws an exception when it tries to convert "zzz" into an integer.
- Exceptions != Errors: programs can be written to recover from Exceptions, but programs can't be written to recover from Errors



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 Mechanism

- The basic way of handling exceptions in Java consists of the ***try-throw-catch*** trio
- 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
  - It can also contain code that throws an exception if something unusual happens

```
try {
   CodeThatMayThrowAnException
}
```

try-throw-catch Mechanism

```
throw new
   ExceptionClassName(PossiblySomeArguments);
```

- 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

- A **`throw`** statement is similar to a method call:

```
throw new ExceptionClassName(SomeString);
```

  - In the above example, the object of class ***`ExceptionClassName`*** is created using a string as its argument
  - This object, which is an argument to the **`throw`** operator, is the exception object thrown
- Instead of calling a method, a **`throw`** statement calls a **`catch`** block

`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) {
   ExceptionHandlingCode
}
```

- A `catch` block looks like a method definition that has a parameter of type `Exception` class
  - It is not really a method definition, however
- A `catch` block is a separate piece of code that is executed when a program encounters and executes a `throw` statement in the preceding `try` block
  - A `catch` block is often referred to as an *exception handler*
  - It can have at most one parameter

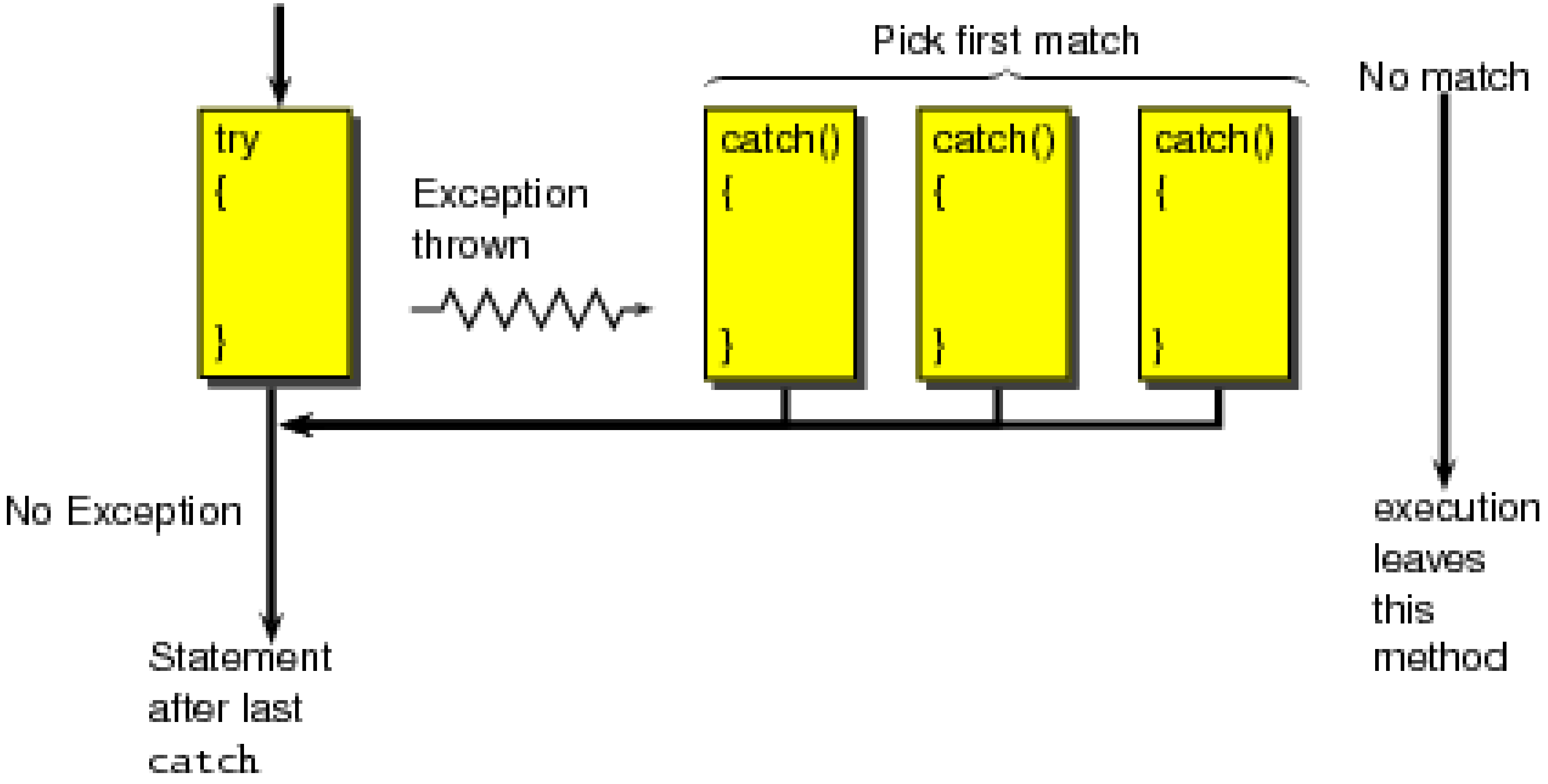
`try-throw-catch` Mechanism

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

- The identifier `e` in the above `catch` block heading is called the `catch` block parameter
- The `catch` block parameter:

1. Specifies the *type of thrown exception* object that the `catch` block can catch (e.g., an `Exception` class object above)
2. 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



An Example with Two Exceptions

```
public class DoubleMistake {
    public static void main(String[] args)  {
        int num = 5, denom = 0, result;
        int[] arr = {7, 21, 31};
        try
        {
            result = num / denom;
            result = arr[num];
        }
        catch (ArithmeticException ex)    {
            System.out.println("Arithmetic error");
        }
        catch (IndexOutOfBoundsException ex)  {
            System.out.println("Index error");
        }
    }
}
```

Note. The second exception will never get thrown. Why?

try-throw-catch Mechanism

- When an exception is thrown by a statement in the try{} block, the catch{} blocks are examined one-by-one starting with the first.
- Only *one* catch{} block is picked.
- If no catch{} block matches the exception, none is picked, and execution leaves this method (just as if there were no catch{} block.)
- The first catch{} block to match the type of the exception gets control.
- The most specific exception types should appear first in the structure, followed by the more general exception types.
- The statements in the chosen catch{} block execute sequentially. After the last statement executes, control goes to the first statement that follows the try/catch structure.
- Control does *not* return to the try block.

User Friendly Example

```
import java.lang.* ;
import java.io.* ;

public class SquareUser
{
  public static void main ( String[] a ) throws
     IOException
  {
    BufferedReader stdin =
        new BufferedReader ( new
     InputStreamReader( System.in ) );
    String  inData = null;
    int      num = 0;
    boolean inputOK = false;
    while ( !inputOK )
    {
      System.out.print("Enter an integer:");
      inData = stdin.readLine();
```

```
try
{
   num       = Integer.parseInt( inData
);
   inputOK = true;
 }
 catch (NumberFormatException ex )
 {
   System.out.println("You entered
invalid data." );
   System.out.println("Please try
again.\n" );
 }
}
System.out.println("The square of " +
 inData + " is " + num*num );
}
}
```

The `finally` clause

- Exception terminates current method
- Danger: Can skip over essential code
- Example:

```
reader = new FileReader(filename);
Scanner in = new Scanner(reader);
readData(in);
reader.close();
// May never get here
```

- Must execute `reader.close()` even if exception happens
- Use `finally` clause for code that must be executed "no matter what"

The `finally` clause

- Executed when `try` block is exited in any of three ways:
  - After last statement of `try` block
  - After last statement of `catch` clause, if this `try` block caught an exception
  - When an exception was thrown in `try` block and not caught
- Cay Horstmann recommendation: don't mix `catch` and `finally` clauses in same `try` block

The `finally` clause

- BlueJ example `(ExceptFinallyEx)`

Multiple `catch` clauses and `finally`

- **If you have any `catch` clauses associated with the `try` block, you must put the `finally` clause after all the `catch` clauses. Example:**

```
try {
    // Block of code with multiple exit points
}
catch (OneException e) {
    System.out.println("Caught one!");
}
catch (OtherException e) {
    System.out.println("Caught other!");
}
catch (AnotherException e) {
    System.out.println("Caught another!");
}
finally {
    // Block of code that is always executed when the try block is exited,
    // no matter how the try block is exited.
    System.out.println("Finally is always executed");
}
```

Exception Classes

- There are more exception classes
  - In the standard Java libraries
  - New exception classes can be defined
- 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

- Numerous *predefined* exception classes are included in the standard packages that come with Java
  - For example:

    ```
    IOException
    NoSuchMethodException
    FileNotFoundException
    ```

  - Many exception classes must be imported in order to use them

    ```
    import java.io.IOException;
    ```

- The predefined exception class **`Exception`** is the root class for all exceptions
  - Every exception class is a descendent class of the class **`Exception`**
  - Used directly, or, most often, to define a derived class
  - It is in the **`java.lang`** package, 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);
}  . . .
```

- Every exception has a `String` instance variable that contains some message
  - This string typically identifies the reason for the exception
- `StringArgument` 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

- 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

A Programmer-Defined Exception Class

```
public class DivisionByZeroException extends Exception
{
   public DivisionByZeroException()
   {
        super("Division by zero.");
   }
   public DivisionByZeroException(String message)
   {
        super(message);
   }
}
```

More can be done in a exception constructor, but this form is common

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

Exception Object Characteristics

- The two most important things about an exception object are its *type* (i.e., 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

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

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

Multiple `catch` Blocks

- Each `catch` block can only catch values of the exception class type given in the `catch` block heading
- Different types of exceptions can be caught by placing more than one `catch` block after a `try` block
  - Any number of `catch` blocks can be included, but they must be placed in the correct order
- A `try` block can potentially throw any number of exception values, and they can be of differing types
  - In any one execution of a `try` block, at most one exception can be thrown (since a `throw` statement ends the execution of the `try` block)
  - However, different types of exception values can be thrown on different executions of the `try` block

Pitfall: Catch the More Specific Exception First

- When catching multiple exceptions, the order of the `catch` blocks is important
  - **When an exception is thrown in a `try` block, the `catch` blocks are examined in order**
  - **The first one that matches the type of the exception thrown is the one that is executed**

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

- Because a `NegativeNumberException` is a type of `Exception`, all `NegativeNumberExceptions` will be caught by the first `catch` block before ever reaching the second block
  - **The catch block for `NegativeNumberException` will never be used!**
- For the correct ordering, simply reverse the two blocks

Throwing an Exception in a Method

- Sometimes it makes sense to throw an exception in a method, but not catch it in the same method
  - Some programs that use a method should just end if an exception is thrown, and other programs should do something else
  - In such cases, the program using the method should enclose the method invocation in a **`try`** block, and catch the exception in a **`catch`** block that follows
- In this case, the method itself would *not include* **`try`** and **`catch`** blocks
  - However, it would have to include a ***`throws`*** *clause*

Declaring Exceptions in a `throws` Clause

- If a method can throw an exception but does not catch it, it must provide a *warning*
  - This warning is called a ***`throws`*** *clause*
  - The process of including an exception class in a throws clause is called *declaring the exception*

    ```
    throws AnException  //throws clause
    ```

  - The following states that an invocation of **`aMethod`** could throw **`AnException`**

    ```
    public void aMethod() throws AnException
    ```

- Note that **`main( )`** is also a method that may have an exception specification:

```
public static void main(String[] args) throws Exception
```

Declaring Exceptions in a `throws` Clause

- If a method can throw more than one type of exception, then separate the exception types by commas

```
public void aMethod() throws
   AnException, AnotherException
```

- If a method throws an exception and does not catch it, then the method invocation ends immediately

The Catch or Declare Rule

- Most ordinary exceptions that might be thrown within a method must be accounted for in one of two ways:

1. The code that can throw an exception is placed within a `try` block, and the possible exception is caught in a `catch` block within the same method
2. The possible exception can be declared at the start of the method definition by placing the exception class name in a `throws` clause

The Catch or Declare Rule

- The first technique handles an exception in a `catch` block
- The second technique is a way to shift the exception handling responsibility to the method that invoked the exception throwing method
- The invoking method must handle the exception, unless it too uses the same technique to "pass the buck"
- Ultimately, every exception that is thrown should eventually be caught by a `catch` block in some method that does not just declare the exception class in a `throws` clause

The Catch or Declare Rule

- In any one method, both techniques can be mixed
  - Some exceptions may be caught, and others may be declared in a `throws` clause
- However, these techniques must be used *consistently* with a given exception
  - If an exception is *not declared*, then it must be *handled* within the method
  - If an exception is *declared*, then the *responsibility* for handling it is *shifted* to some other calling method
  - Note that if a method definition encloses an invocation of a second method, and the second method can throw an exception and does not catch it, then the first method must catch or declare it

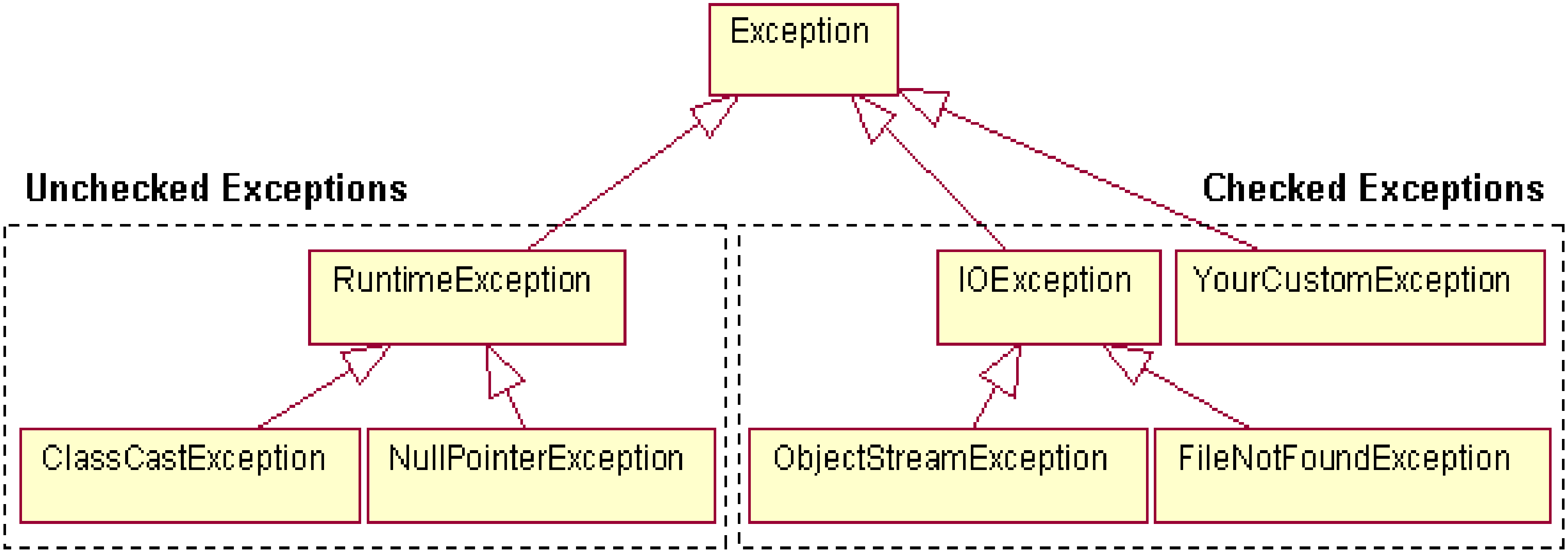
Checked and Unchecked Exceptions

- Exceptions that are subject to the catch or declare rule are called *checked* exceptions
  - The compiler checks to see if they are accounted for with either a `catch` block or a `throws` clause
  - The classes `Throwable`, `Exception`, and all descendants of the class `Exception` are checked exceptions
- All other exceptions are *unchecked* exceptions
- The class `Error` and all its descendant classes are called *error classes*
  - Error classes are *not* subject to the Catch or Declare Rule

Exceptions to the Catch or Declare Rule

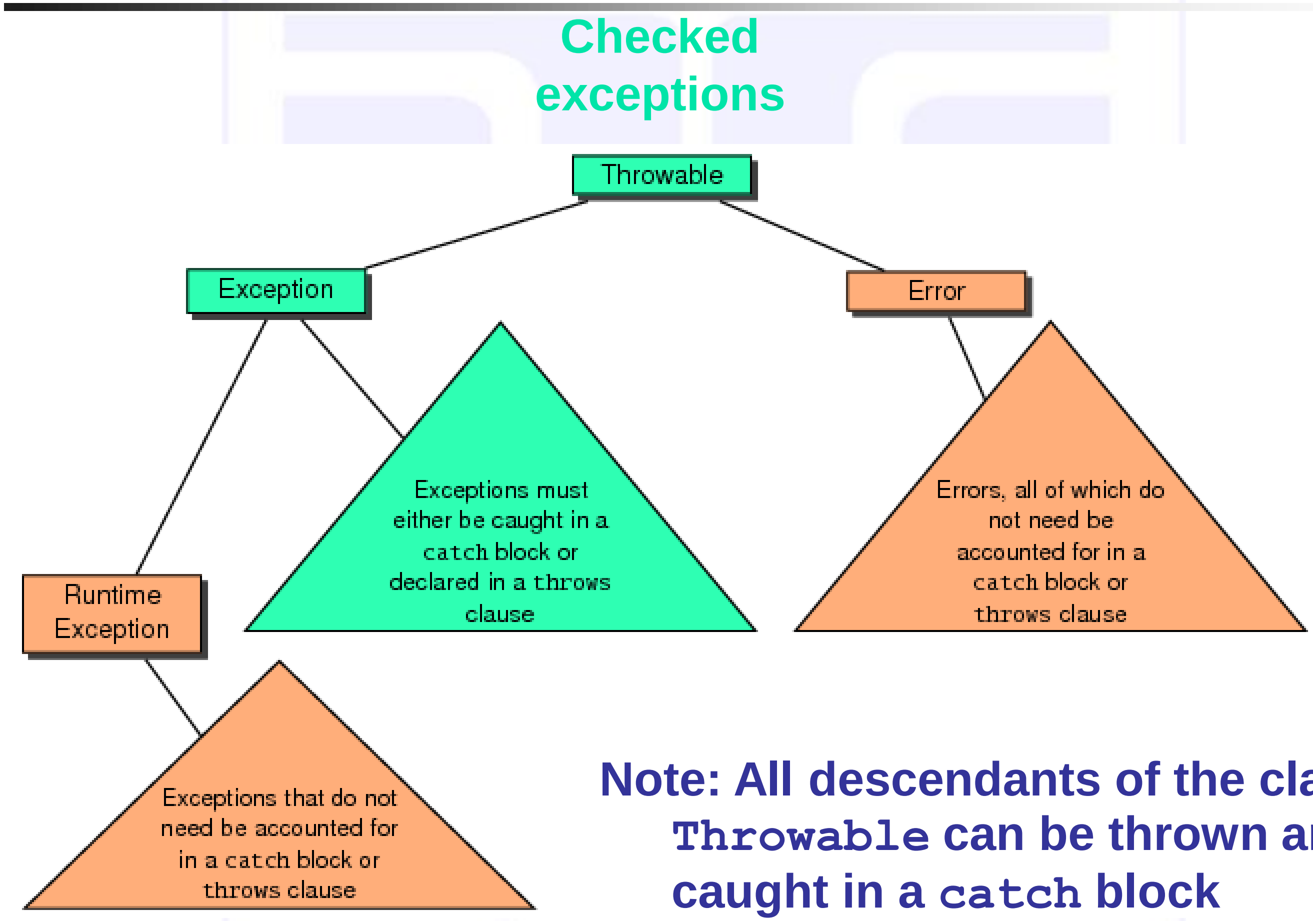
- *Checked* exceptions must follow the Catch or Declare Rule
  - Programs in which these exceptions can be thrown will not compile until they are handled properly
- *Unchecked* exceptions are exempt from the Catch or Declare Rule
  - Programs in which these exceptions are thrown simply need to be corrected, as they result from some sort of error

Checked and Unchecked Exceptions



Note. This is a part of the hierarchy

Hierarchy of Throwable Objects


Checked exceptions
Throwable
Exception
Error
Runtime Exception
Exceptions must either be caught in a catch block or declared in a throws clause
Errors, all of which do not need be accounted for in a catch block or throws clause
Exceptions that do not need be accounted for in a catch block or throws clause

Note: All descendants of the class `Throwable` can be thrown and caught in a `catch` block

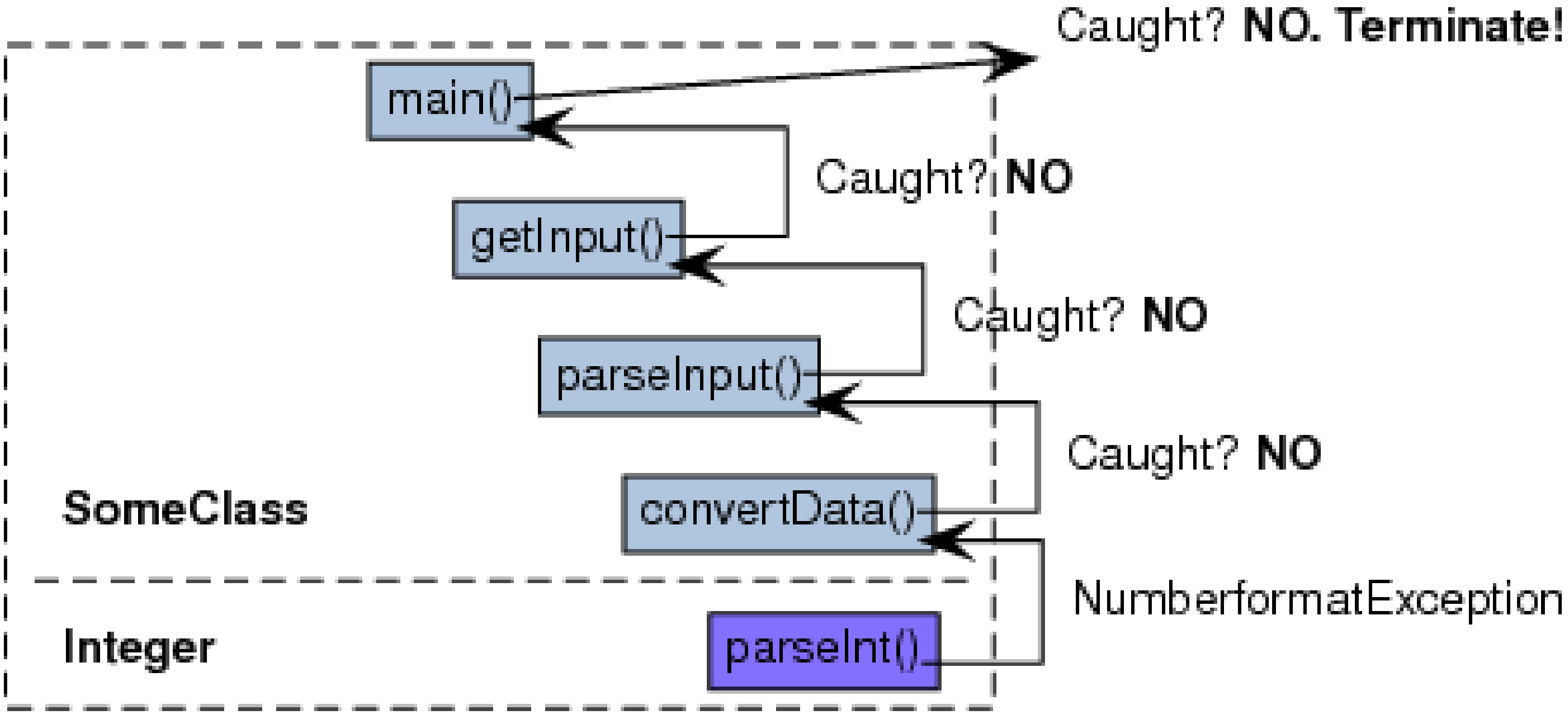
The `throws` Clause in Derived Classes

- When a method in a *derived* class is *overridden*, it should have the *same exception classes* listed in its `throws` clause that it had in the base class
  - Or it should have *a subset* of them
- A derived class *may not add* any exceptions to the `throws` clause
  - But it can delete some

What Happens If an Exception is Never Caught?

- If every method up to and including the main method simply includes a `throws` clause for 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

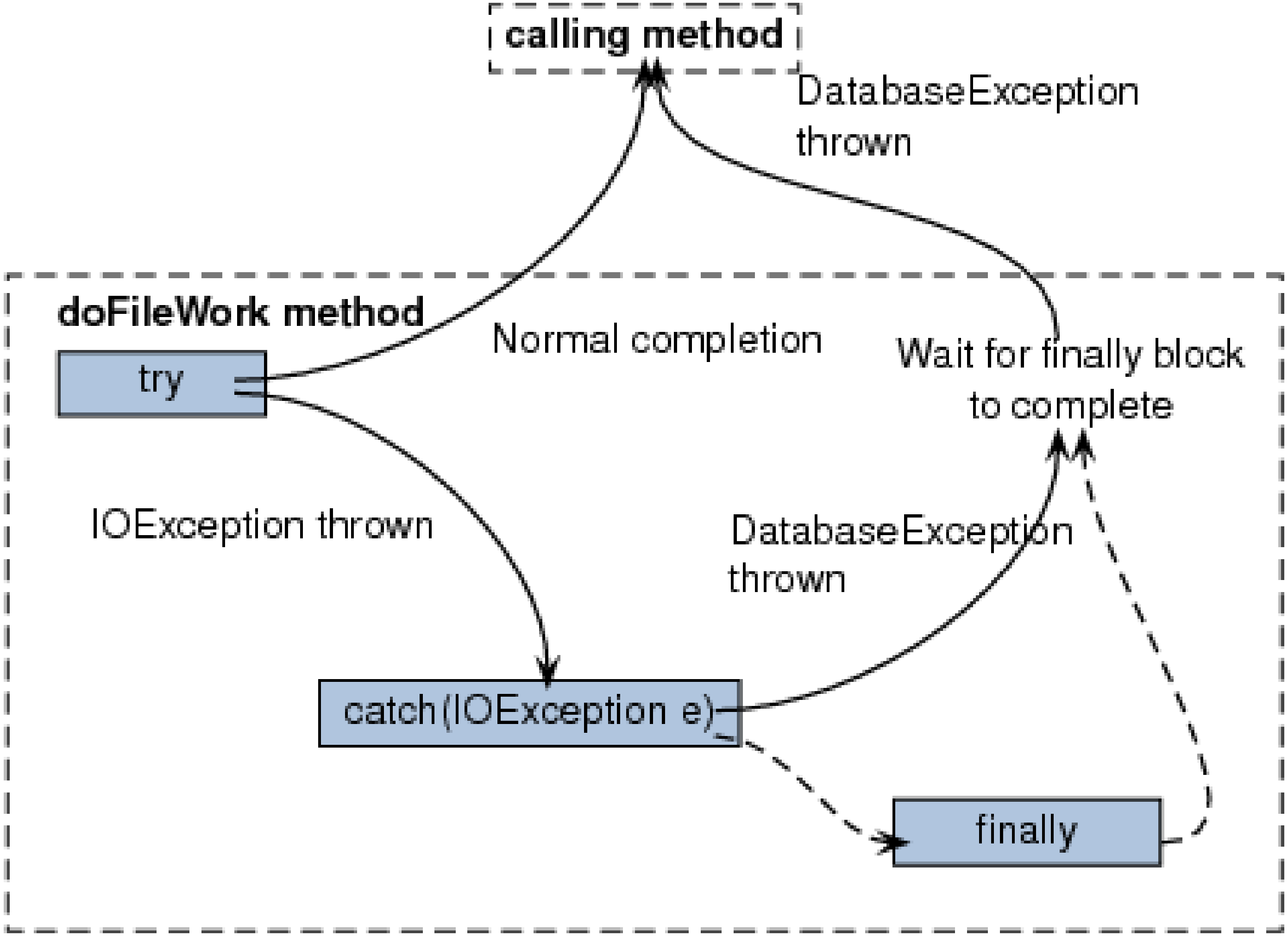
Exception propagation



Another Example

```
public void doFileWork(String filename)
   throws DatabaseException{
 FileOutputStream fos = null;
 ObjectOutputStream oos = null;
 try{
  fos = new
   FileOutputStream(filename);
  oos = new ObjectOutputStream(fos);
  oos.writeObject(obj);
 }
 catch(IOException e){
  throw new DatabaseException(
   "Problem while working with
  "+filename+": "
   +e.getMessage());
 }
finally{
  try{
   if(oos!=null){
     oos.close();
   }
   if(fos!=null){
     fos.close();
   }
  }
  catch(IOException e){
   throw new DatabaseException(
     "Problem while working with
  "+filename+": "
     +e.getMessage());
  }
 }
}
```

Example Discussion



When to Use Exceptions

- Exceptions should be reserved for situations where a method encounters *an unusual or unexpected case that cannot be handled* ***easily*** *in some other way*
- When exception handling must be used, here are some basic guidelines:
  - Include `throw` statements and list the exception classes in a `throws` clause within a method definition
  - Place the `try` and `catch` blocks in a different method

When to Use Exceptions

- Here is an example of a method from which the exception originates:

```
public void
   someMethod()
                throws
   SomeException
{
   . . .
   throw new

   SomeException(SomeArg
   ument);
   . . .
}
```

- When `someMethod` is used by an `otherMethod`, the `otherMethod` must then deal with the exception:

```
public void otherMethod()
{
   try {
     someMethod();
     . . .
   }
   catch (SomeException e)
   {
     CodeToHandleException
   }
   . . .
}
```

Exception Guidelines

- If your method encounters an abnormal condition that it can't handle, it should throw an exception.
- Avoid using exceptions to indicate conditions that can reasonably be expected as part of the normal functioning of the method.
- If your method discovers that the client has breached its contractual obligations (for example, by passing in bad input data), throw an unchecked exception.

Exception Guidelines

- If your method is unable to fulfill its contract, throw either a checked or unchecked exception.
- If you are throwing an exception for an abnormal condition that you feel client programmers should consciously decide how to handle, throw a checked exception.
- Define or choose an already existing exception class for each kind of abnormal condition that may cause your method to throw an exception.

Rethrowing Exceptions

- After an exception is caught, it can be rethrown if is appropriate.
- When rethrowing an exception you can choose the location from where the stack trace says the object was thrown.
  - You can make the rethrown exception appear to have been thrown from the location of the original exception throw, or
  - from the location of the current rethrow.
- To rethrow an exception and have the stack trace indicate the original location, simply rethrow the exception:

```
try {
    cap(0);
} catch(ArithmeticException e) {
    throw e;
}
```

Rethrowing Exceptions

- For the stack trace to show the actual location from which the exception is being rethrown: call the exception's fillInStackTrace() method.
  - This method sets the stack trace information in the exception based on the current execution context. Example:

```
try {
    cap(0);
}
catch(ArithmeticException e) {
    throw (ArithmeticException)e.fillInStackTrace();
}
```

- Call `fillInStackTrace()` on the same line as the throw statement – thus the line number specified in the stack trace matches the line on which the throw statement appears.
  - The `fillInStackTrace()` method returns a reference to the `Throwable` class, so you need to cast the reference to the actual type of the exception.

- BlueJ example `(DataSetReader)`

Reading

- Eckel: chapter 13
- Barnes: chapters 6, 8, 9
- Deitel: chapters 11, 12, 13

Summary

- OO Application development
  - Using CRC cards example
- Assertions
- Exceptions and Errors.
  - Checked vs. unchecked exceptions
  - try and catch statements
  - finally clause
  - Catch or declare rule
  - throws clause
  - throw statement