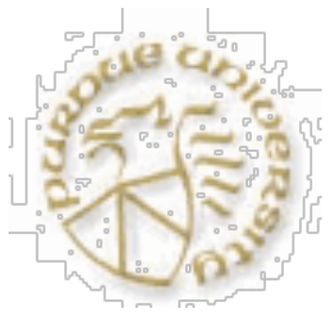
Exception Handling

CS 18000
Sunil Prabhakar
Department of Computer Science
Purdue University



When things go wrong

- Good programs should be robust -- i.e., they should be able to handle exceptional situations.
- What happens if we are trying to input an integer value and the user enters ten, or 3.45?
- A good program should tell the user to re-enter a valid integer.
- So far, this situation would result in the termination of our program when we execute Integer.parseInt() on this invalid string.
- How do we prevent this?

Handling errors

- One idea is to use if -then style tests whenever we expect that an error may arise.
- This is the style in C -- return values can signal the existence of an error.
- But this is clumsy, and inelegant.
- In Java, the exception handling mechanism is used instead.
- Erroneous (or unexpected) cases are handled by a special type of control flow.

Exceptions

- An *exception* is used to indicate that something unusual (that prevents regular processing) has occurred.
- When an exception occurs, or is *thrown*
  - an Exception object is created, and the normal sequence of flow is terminated, and
  - an exception handling mechanism is invoked which is responsible for handling or *catching* the thrown exception.

Uncaught Exceptions

- When a (runtime) exception is thrown, and the program does not specify how to handle it, it causes the program to terminate:

```
public class ReadInt {
    public static void main(String[] args) {
        String inputStr;
        int i;

        inputStr = JOptionPane.showInputDialog(null, "Enter an Integer");
        i = Integer.parseInt(inputStr);

        System.out.println("Read in " + i);
    }
}
```

catching An Exception

```
public class ReadInt2 {
    public static void main(String[] args) {
        String inputStr;
        int i;

        inputStr = JOptionPane.showInputDialog(null,
                                    "Enter Deposit Amount”);

        try {
            i = Integer.parseInt(inputStr);
        } catch (Exception e) {
            System.out.println("Invalid integer");
        }
    }
}
```

Does not crash due to invalid input.

Exception control-flow

Assuming that a Exception is thrown when executing this statement.

No exception

```
. . .
try{
   . . .
   stmt;
   . . .
} catch (Exception e){
   . . .
}
. . .
```

The **`catch`** block is not executed.

Exception thrown

```
. . .
try{
   . . .
   stmt;
   . . .
} catch (Exception e){
   . . .
}
. . .
```

The **`catch`** block is executed immediately after statement causing the exception. The rest of the **`try`** block is not executed.

Exception control-flow

Assuming that a Exception is thrown when executing this statement.

No exception

```
. . .
try{
   . . .
   stmt;
   . . .
} catch (Exception e){
   . . .
}
. . .
```

The **`catch`** block is not executed.

Exception thrown

```
. . .
try{
   . . .
   stmt;
   . . .
} catch (Exception e){
   . . .
}
. . .
```

The **`catch`** block is executed immediately after statement causing the exception. The rest of the **`try`** block is not executed.

Exception object

- An exception is thrown by creating an Exception object.
- The exception object is passed to the catch block as a parameter.
- It contains details about the actual exception that was thrown.

e is a catch block parameter corresponding to the exception object.

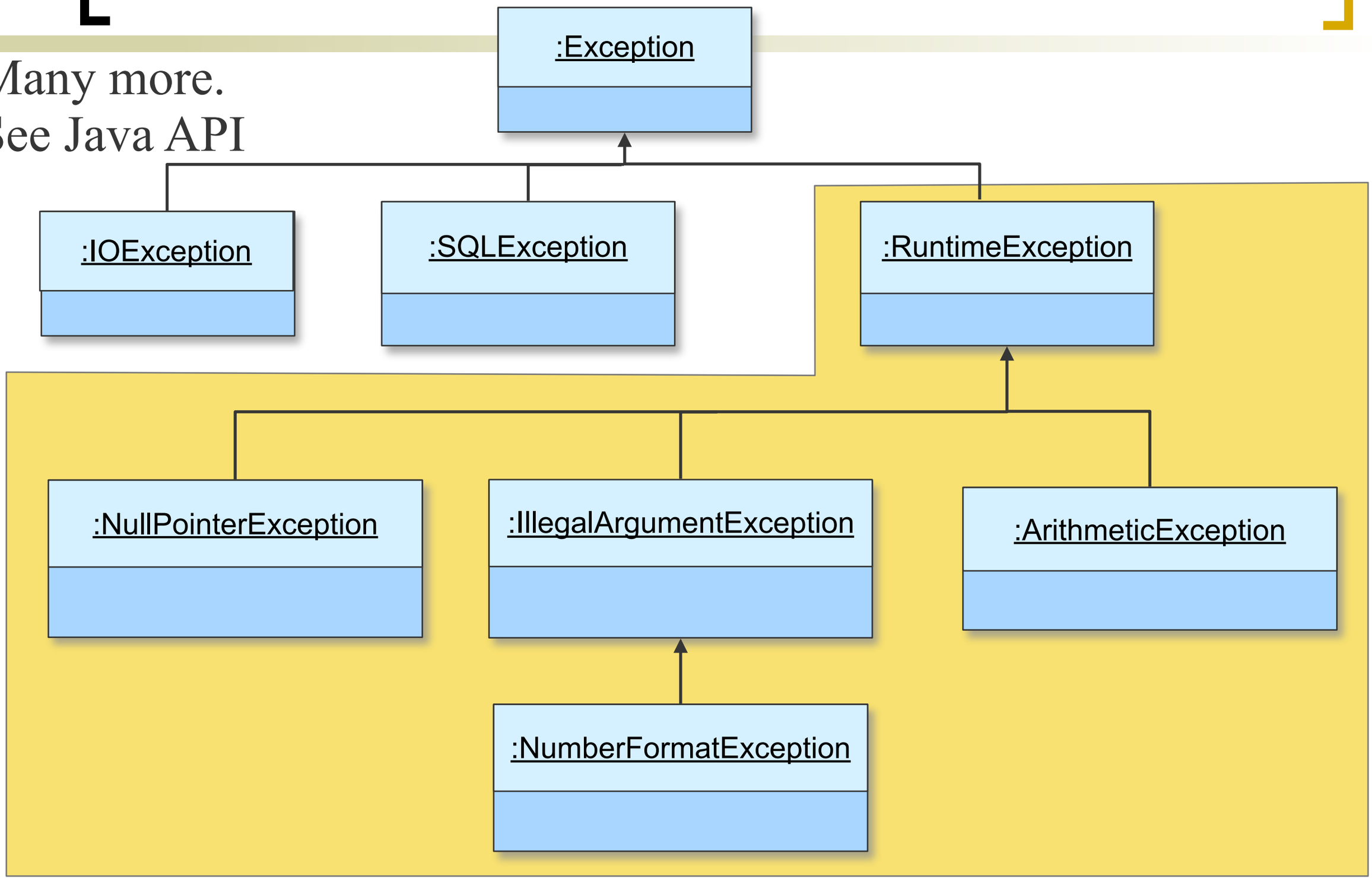
```
try {
   . . .
} catch (Exception e){
   . . .
}
```

Exception object

- The exception object contains details about the exception.
  - The `getMessage()` method simply returns a string of text that describes the exception.
  - The `printStackTrace()` method gives us the order (and line numbers) in which methods had been called when the exception took place.
    - In reverse order of the calls
    - The last method call is listed first, main is last.

The Exception Class Family

Many more.
See Java API



Multiple `catch` Blocks

- If more than one type of exception can take place, we may want to handle each one differently.
- A single `try-catch` statement can include multiple catch blocks, one for each type of exception.
- Only the first matching `catch` block is executed.
- Matching is based on the class of the exception.
- *Make sure to list classes lower in the hierarchy before listing classes higher up.*

Multiple `catch` Blocks

```
try {
    i = Integer.parseInt(inputStr);
    i = 5 / i;


} catch (NumberFormatException e) {
    // code to handle NumberFormatExceptions.
    System.out.println("Number Format Error -- not a valid integer”);

} catch (NullPointerException e) {
    // code to handle NullPointerExceptions.
    System.out.println("Null Pointer Exception”);

} catch (Exception e) {
    // code to handle all other exceptions.
    System.out.println("Some other exception took place:" + e.getMessage());

    e.printStackTrace();
}
```

Terminating a program

- It is possible to terminate a program at any point in its execution (maybe because a very serious error has occurred).
- This is achieved by calling

```
System.exit(0);
```

- This call takes any integer value as a parameter.
- The program is immediately terminated.
- Should be a last resort — try to end gracefully.

The `finally` Block

- There are situations where we need to take certain actions regardless of whether an exception is thrown or not.
- We place statements that must be executed regardless of exceptions, in the `finally` block.
- Commonly used to perform cleanup (e.g., disconnecting from a database, or closing a network connection)

Exception control-flow

Assuming that a Exception is thrown when executing this statement.

No exception

```
. . .
try{
   . . .
   stmt;
   . . .
} catch (Exception e){
   . . .
}
. . .
```

The **`catch`** block is not executed.

Exception thrown

```
. . .
try{
   . . .
   stmt;
   . . .
} catch (Exception e){
   . . .
}
. . .
```

The **`catch`** block is executed immediately after statement causing the exception. The rest of the **`try`** block is not executed.

Exception control-flow

Assuming that a Exception is thrown when executing this statement.

No exception

```
. . .
try{
   . . .
   stmt;
   . . .
} catch (Exception e){
   . . .
}
finally {
   . . .
}
. . .
```

Exception thrown

```
. . .
try{
   . . .
   stmt;
   . . .
} catch (Exception e){
   . . .
}
finally {
   . . .
}
. . .
```

`finally` block is always executed.

Salient points

- If multiple catch blocks are defined they are tested in order -- only the first that matches the thrown exception gets executed.
  - List them from more specific to general.
  - CAUTION: if A is a subclass of B, then an exception of class A is also an exception of class B!
- Even if there is a `return` from the `try` or `catch` blocks, the `finally` block **is** executed before returning!
- Even if no matching catch block is found for an exception, the `finally` block gets executed

Caution: order of catch blocks

```
try {
        . . .
        i = Integer.parseInt(inputStr);
        . . .
    } catch (Exception e){
        . . . // code to handle general exceptions.
    } catch (NullPointerException e){
        . . . // code to handle NullPointerExceptions.
    } catch (NumberFormatException e){
        . . . // code to handle NumberFormatExceptions.
    }
}
```

Will never get executed!

Passing the Buck: Propagating Exceptions

Propagating exceptions

- If an exception occurs and there is no matching catch block, then the exception is **propagated**.
  - control passes to the calling method (like a return)
  - if the caller has no matching catch block, the same happens
  - eventually, if the main method does not handle the exception, the runtime system handles it.

Exception handling

```
public static void main(String[] args){
  ...
  a.methodA();
  ...
}
```

```
public void methodA(){
  try{
    methodB();
  } catch (NumberFormatException e){
    . . .
  }
  . . .
}
```

NumberFormatException is thrown when executing this statement.

methodA has a matching catch block: Exception his **handled**

methodB has no matching catch block: Exception his **propagated**

```
public void methodB(){
  stmt;
}
```

Exception handling

```
public static void main(String[] args){
  ...
  a.methodA();
  ...
}
```

main() also has no matching catch block: Program is **Terminated**

```
public void methodA(){
  try{
    methodB();
  } catch (NumberFormatException e){
    . . .
  }
  . . .
}
```

NullPointerException is thrown when executing this statement.

methodA also has no matching catch block: Exception is **propagated**

methodB has no matching catch block: Exception is **propagated**

```
public void methodB(){
  stmt;
}
```

Types of exceptions

- There are types of exceptions
  - Checked exceptions
  - Unchecked exceptions
- Unchecked exceptions are those that can be thrown during the normal operation of the Java Virtual Machine
  - RuntimeException and its descendants.
  - NullPointerException, ArithmeticException, IndexOutOfBoundsException, … (see API)

Types of exceptions (cont.)

- Unchecked exceptions need not be explicitly handled (as we have done so far)
  - If unhandled, will lead to program termination.
- Checked exceptions must be explicitly handled by the program.
  - Any method that could result in a checked exception being thrown must either:
    - Handle it with a **`try-catch`** block, OR
    - Propagate and explicitly declare this possibility.

Propagating Checked Exceptions

- A method that propagates a checked exception must declare this possibility:
  - the method header must include the reserved word **`throws`** followed by a list of the classes of exceptions that may be propagated
  - optional for runtime (unchecked) exceptions

```
public int accessDB( ) throws SQLException {
   . . .
   // code that may result in a SQLException
   . . .
}
```

Handling Unchecked Exceptions

`parseInt` throws NumberFormatException (see API).

Catcher

```
void methodA( ){
   try {
      int i = Integer.parseInt(s);
   } catch (NumberFormatException e) {
     . . .
   }
}
```

Propagators

```
void methodB( ) {
   int i = Integer.parseInt(s);
}
```

```
void methodB( ) throws  NumberFormatException {
   int i = Integer.parseInt(s);
}
```

Optional to declare this propagation

Handling Checked Exceptions

`Scanner(File )` throws FileNotFoundException (see API).

Catcher

```
void methodA( ){
   File f = new File (“Data.txt");
   try {
      scanner = new Scanner(f);
   } catch (FileNotFoundException e) {
     . . .
   }
}
```

Propagator

```
void methodB( ) throws  FileNotFoundException {
   File f = new File (“Data.txt");
   scanner = new Scanner(f);
}
```

Must declare this propagation

Creating Custom Exceptions

Throwing Exceptions

- We can throw an exception at any point in our code.
- To do this, we create an exception object and **`throw`** it.
- If this is a checked exception, we must declare that we throw this exception (unless we catch the exception).

```
public float squareRoot(float value) throws Exception {
    . . .
    if (value < 0)
     throw new Exception ("Imaginary numbers not yet supported");
    . . .
}
```

Defining Custom Exceptions

- Should only need to do this if we want to
  - capture extra information, or
  - handle this class in a special `catch` block.
- In order to define a new exception class, we **must**:
  - Extend an exception class. Good idea to extend the Exception class.
  - Define a default constructor.
  - Call the parent's constructor as the first call in the constructor for the new exception: `super(msg);`

Problem

- In order to avoid cluttering our code with try-catch blocks whenever we need to get an integer value from the using using the showInputDialog() method, we will create a helper method

- A static method called `getInt()` in a class called `SafeInputHelper`

```
int i;
while (true) {
  i = SafeInputHelper.getNextInt("Enter an Integer");
  System.out.println("User input is:" + i);
}
```

SafeInputHelper

```
import javax.swing.*;

public class SafeInputHelper {

 static int getNextInt(String msg) {
    String str;
    String errorMsg = msg + “\n Invalid integer format, please re-enter";
    int i;

    do{
      str = JOptionPane.showInputDialog(null, msg);
      try{
        i = Integer.parseInt(str);
        return i;
      } catch (NumberFormatException e) {
            msg = errorMsg;
      }
    } while (true);
  }
}
```

Ascending Input Helper

- Let us assume that our application often needs to input several streams of integers in ascending order with a minimum jump between values.
  - each stream has its own starting point and minimum jump.
- Create a helper class to input such values: AscendingInputHelper
- This class throws a new type of exception that signals that the ascending rule was violated: AscendingException.

AscendingException

```
public class AscendingException extends Exception {
    private int lastEntry;         // the last valid entry
    private int errorEntry;        // the violating entry
    private int minimumIncrement; // the required minimum increment
    private static final String DEFAULT_ERROR_MSG = "Invalid Ascending Sequence";

    public AscendingException(int badEntry, int last, int inc) {
        this(DEFAULT_ERROR_MSG, badEntry, last, inc);
    }

    public AscendingException(String msg, int badEntry, int last, int inc) {
        super(msg);
        errorEntry = badEntry;
        lastEntry = last;
        minimumIncrement = inc;
    }

    public int getLastEntry() { return lastEntry; }

    public int getErrorEntry() { return errorEntry; }

    public int getMinimumIncrement() { return minimumIncrement; }
}
```

AscendingInputHelper

```
public class AscendingInputHelper {
    private int lastValue;           // the previous input for this sequence
    private int minimumIncrement;    // the minimum increment required

    public AscendingInputHelper(int start, int minInc) {
        lastValue = start;
        minimumIncrement = minInc;
    }

    public int getNextInt() throws AscendingException {
        int i;

        //Get the next integer from the user
        i = SafeInputHelper.getNextInt("Enter Next Integer");

        //if invalid ascent, throw exception with appropriate data
        if (i < lastValue + minimumIncrement)
            throw new AscendingException(i, lastValue, minimumIncrement);
        lastValue = i;
        return i;
    }
}
```

Using AscendingInputHelper

```
public class TestAscendingInput {
    public static void main(String args[]) {
        int newValue, total = 0;

        // Create an ascendingInputHelper object
        AscendingInputHelper ascInput = new AscendingInputHelper(0, 3);

        while (true) {
            try {
                newValue = ascInput.getNextInt();
                total += newValue;
            } catch (AscendingException e) {
                JOptionPane.showMessageDialog(null,
                        "Error with order of input\n" + e.getMessage() +
                                "\nEntered value: " + e.getErrorEntry() +
                                "\n Previous value: " + e.getLastEntry() +
                                "\n Minimum Increment required: " +
                                e.getMinimumIncrement());
                System.out.print("Total of valid inputs: " + total);
                System.exit(0);
            }
        }
    }
}
```

Assertions

Assertions

- Exceptions handle unexpected behavior during execution.
- Sometimes programs fail due to logical errors in the code.
- Assertions are a mechanism available to detect logical errors.
- An assertion is essentially a sanity check regarding the state of data at a given point in the program.

Assertions

- The syntax for the **assert** statement is

```
assert <boolean expression>;
```

where `<boolean expression>` represents the condition that must be true if the code is working correctly.

- If the expression results in **false**, an **AssertionError** (a subclass of **Error**) is thrown.
- Error is sibling of Exception (children of Throwable)

Sample Use #1

```
public double deposit(double amount) {
   double oldBalance = balance;
   . . . //some complex processing to change balance
   assert balance >= oldBalance;
}

public double withdraw(double amount) {
   double oldBalance = balance;
   . . . //some complex processing to change balance
   assert balance < oldBalance;
}
```

Second Form

- The assert statement may also take the form:

```
assert <boolean expression>: <expression>;
```

where `<expression>` represents the value passed as an argument to the constructor of the **AssertionError** class. The value serves as the detailed message of a thrown exception.

Sample Use #2

```
public double deposit(double amount) {

   double oldBalance = balance;

   . . . //some complex processing to change balance
    assert balance > oldBalance :
      "Serious Error - balance did not " +
      " increase after deposit";
}
```

AscendingInputAssert

```
class AssendingInputAssert {
    public static void main(String args[]) {
        int minIncrement = 3;
        int lastValue = 0, newValue, total = 0;

        //Keep asking for ascending values until error
        while (true) {
            try {
                newValue = SafeInputHelper.getNextInt(
                                "Enter next value in sequence");

                //assert the requirement
                assert (newValue - lastValue) >= minIncrement;

                total += newValue;
                lastValue = newValue;
            } catch (AssertionError e) { //catch assert errors
                JOptionPane.showMessageDialog(null,
                     Invalid increment: terminating program");
                System.out.print("Total of valid inputs: " + total);
                System.exit(0);
            }
        }
    }
}
```

Compiling Programs with Assertions

- Before Java 2 SDK 1.4, the word **assert** is a valid non-reserved identifier. In version 1.4 and after, the word **assert** is treated as a regular identifier to ensure compatibility.

- To enable the assertion mechanism, compile the source file using (automatic in IntelliJ with Java version >= 1.4)

```
javac -source 1.4 <source file>
```

Running Programs with Assertions

- To run the program with assertions enabled, use

```
java –ea <main class>
```

- If the `–ea` option is not provided, the program is executed without checking assertions.
- With IntelliJ add this as a VM option under configurations.

Use of Assertions

- Do not use assertions for:
  - argument checking
  - while executing a required part of your code
- Use them to ensure assumptions that you are making. Examples:
  - if you require your int values to be non-negative at some point in the code
  - if all allowed options of a switch statement have been covered, and there should be no other value, add an assertion to the default case

Assertion Examples

```
int i;
...
//expecting i to be positive
assert (i >= 0)
```

```
char grade;
...
switch(grade) {
    case 'A':
        //handle case A ...
        break;
    case 'B':
        //handle case B...
        break;
    ...
    default:
        //not other options
        assert false: grade;
}
```

```
void foo() {
    for (...) {
        if (...)
        return;
    }
    // Execution should never reach this point!
    assert false;
}
```

From Java Tutorial. The compiler may not allow the assert if it can be sure that execution never reaches this line.