LECTURE 8

TESTING, EXCEPTIONS & PACKAGES

Slides by Donald W. Smith
TechNeTrain.com

Final Draft
10/30/2011

TESTING

Testing Terminology

- Error
- Fault
- Failure

Testing Terminology

- Error: doing something incorrectly
  - Types of errors
    - Error of omission
    - Error of commission

Testing Terminology

- Fault: the appearance/form/manifestation of an error
  - For example,
    - wrong requirements for a project
    - something wrong in a design diagram
    - something wrong in the code
  - Often called a ‘bug’

Testing Terminology

- Failure: inability of the system to do what is required
  - Note that one fault can lead to many failures
  - Also, no failures does not imply no faults; could be just poor testing

Test case

- A test case consists of
  - a set on input values
  - a set of preconditions that must hold
  - a set of expected outputs

Testing

- Ideally,
  - a test driver is created to run and check all test cases, and report any failures
  - the driver is executed after every change in the system, called regression testing
  - test cases are added whenever new test situations are discovered

Note that exhaustive testing is impossible, except for trivial programs, as too many cases are possible to test them all.

Approaches to Testing

- Human testing: people reading the documents/code
- Black-box testing: tests based on the specification only, ignores actual code
- White-box testing: tests based on the code used to implement the system
- Object-oriented testing: tests particularly suited to systems implemented by an object-oriented language

Bottom-up Testing

- Approach:
  - first test units that don't use any other units
  - repeatedly, test the integration of modules that only use modules already tested
  - continue until the whole system has been tested
- Advantages:
  - develop working and tested subsystems
  - thorough testing of low-level modules
  - don't need stubs (see below for definition)

Top-down Testing (cont.)

- Advantages:
  - Difficult task of integration done early, and often it is much easier as the interfaces of the modules to be used can be designed to work together smoothly.
  - One module is integrated into the working system at a time.
  - Early demonstrations of the whole system and its capabilities, although with incorrect output values.
  - May need fewer drivers, although the tester must be careful to not just test a module in the context of its anticipated use.

Top-down Testing (cont.)

- The testing approach usually matches the development approach.
- Often a hybrid approach is used:
  - bottom-up for well understood modules
  - top-down for nebulous modules.

Regression Testing

- As comprehensive as possible
- Rerun whenever a change is made
- Placement of tests
  - Separate class, eg, TestContainerOfBoxes
  - Special method, eg. main of BetterBox
  - Junit testing module, especially for testing modules that depend on other modules
- Easy to rerun and verify still correct
  - often a main method
  - data is often hard coded,
    - output must be short and easily checked by the tester
    - or, expected output stored in a file and actual output automatically compared to it
  - instead of hard coding the data, it can be read from a file

Black-box Testing

- Based on the specification of what the program should do (not how)
  - Need a precise specification of what the program should do
  - Specification is based on preconditions, postconditions, and contracts
- The tests should be written before the code.
- Techniques:
  - Outcome testing
  - Boundary-value testing

Outcome Testing

- Be sure that every routine, even the most trivial, is invoked at least once to ensure that it has the correct outcome
- If the unit might produce different results/outcomes that satisfy different constraints, then be sure to test that all of them can be obtained.
  - eg. a search can find the item or fail to find the item
- Test any special cases
- The same value or object supplied for more than one argument, sometimes called aliasing
- Be sure to test all error situations

Boundary-value Testing

- If an input is restricted to a certain range, then try
  - minimum value
  - maximum value
  - nominal value
  - minimum value + 1
  - maximum value – 1
  - 2 or more independent variables: only need to vary one at a time with others at the nominal value
  - 2 or more dependent variables; try all combinations of all of them
- If an argument can come in different sizes,

Boundary-value Testing

- If an input is restricted to a certain range, then try
  - minimum value
  - maximum value
  - nominal value
  - minimum value + 1
  - maximum value – 1
  - 2 or more independent variables: only need to vary one at a time with others at the nominal value
  - 2 or more dependent variables; try all combinations of all of them
- If something can be in different positions, then try the different positions

Boundary-value Testing (cont.)

- If an argument can come in different sizes, then try
  - min size
  - min size + 1
  - nominal size
  - max size
  - max size – 1
- Test boundaries of special cases
- Whatever boundaries can be found
- For robust testing, try:
  - minimum - 1
  - maximum + 1

EXCEPTION HANDLING

7.4 Exception Handling

- There are two aspects to dealing with run-time program errors:
  1) Detecting Errors

     This is the easy part. You can 'throw' an exception

     Use the throw statement to signal an exception

     ```
     if (amount > balance)
     {
       // Now what?
     }
     ```

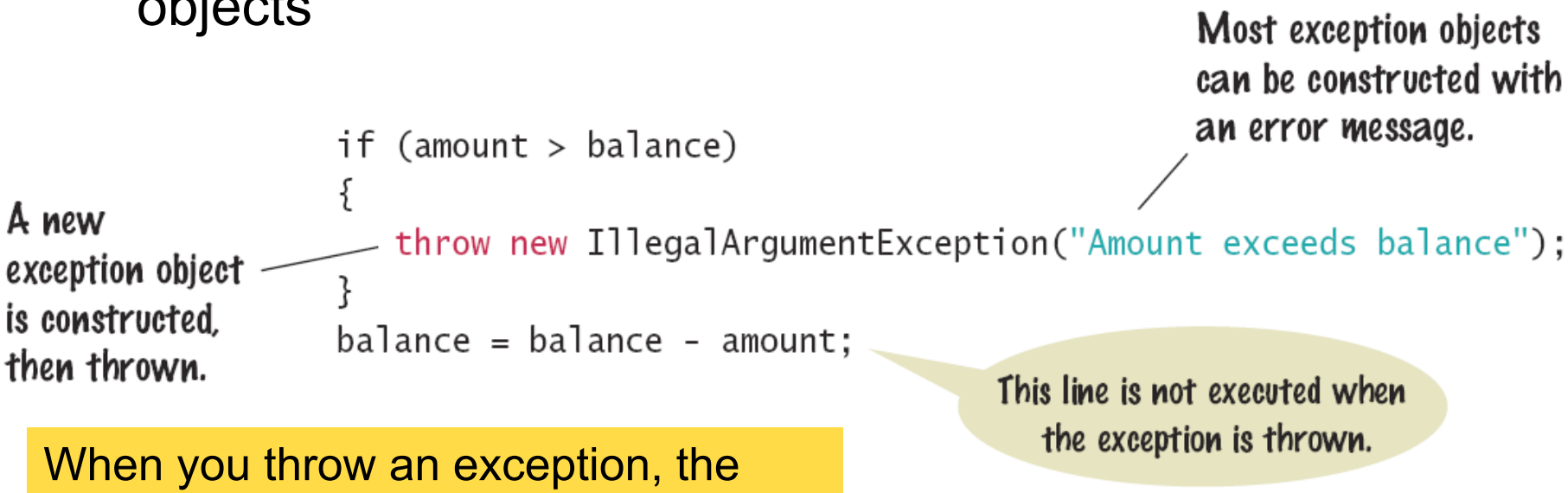
  2) Handling Errors

     This is more complex. You need to 'catch' each possible exception and react to it appropriately

- Handling recoverable errors can be done:
  - Simply: exit the program
  - User-friendly: As the user to correct the error

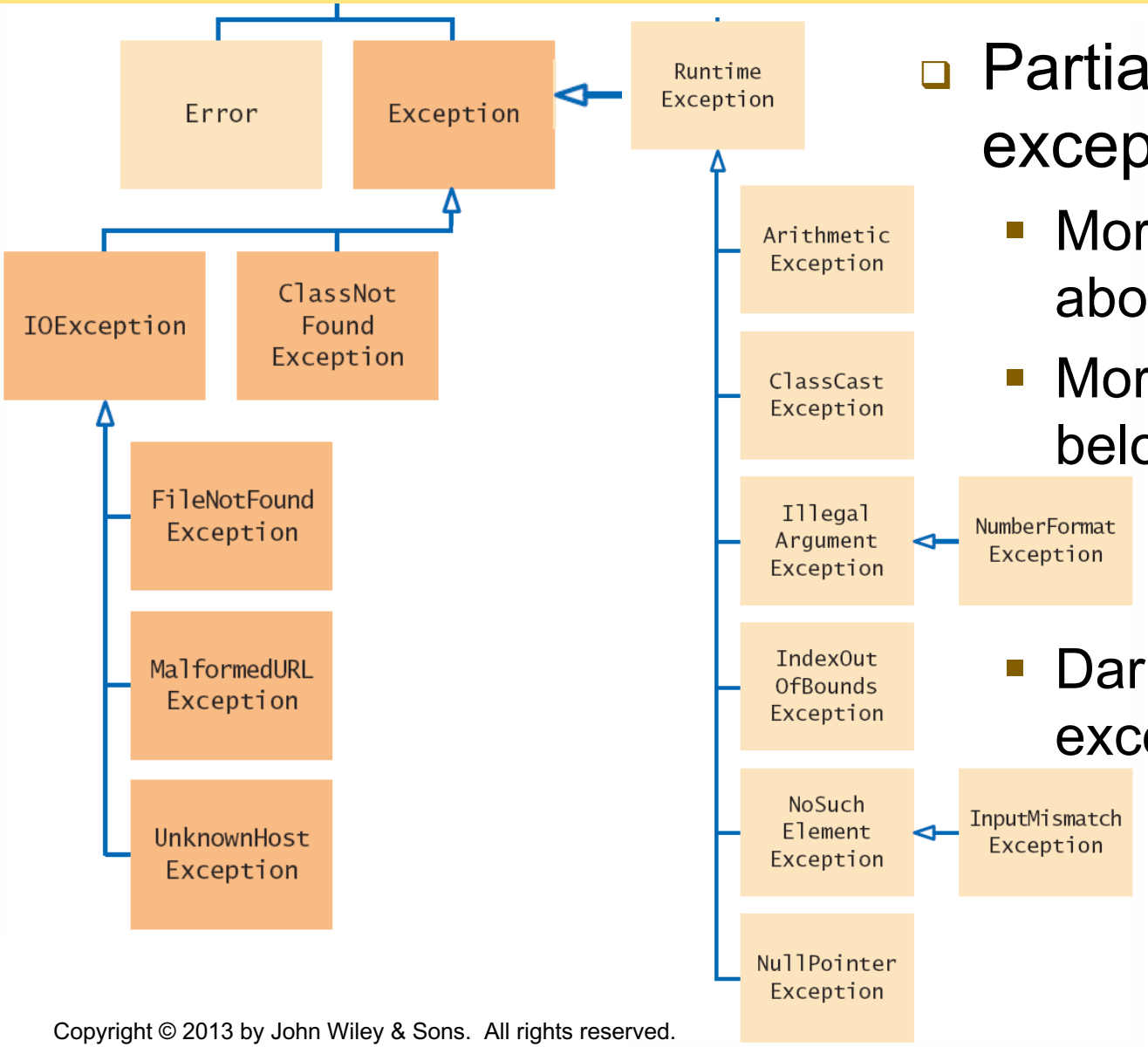
Syntax 7.1: Throwing an Exception

- When you throw an exception, you are throwing an object of an exception class
  - Choose wisely!
  - You can also pass a descriptive String to most exception objects



When you throw an exception, the normal control flow is terminated.

Exception Classes



- Partial hierarchy of exception classes
  - More general are above
  - More specific are below
  - Darker are Checked exceptions

Catching Exceptions

- Exceptions that are thrown must be 'caught' somewhere in your program

```
try
{
   String filename = . . .;
   Scanner in = new Scanner(new File(filename));
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
   System.out.println("Input was not a number");
}
```

Surround method calls that can throw exceptions with a 'try block'.

FileNotFoundException

NoSuchElementException

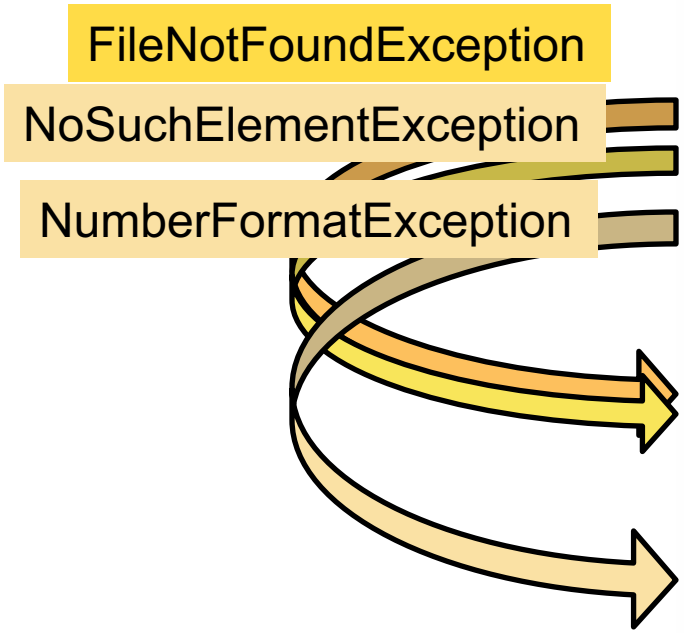
NumberFormatException

Write 'catch blocks' for each possible exception.

It is customary to name the exception parameter either 'e' or 'exception' in the catch block.

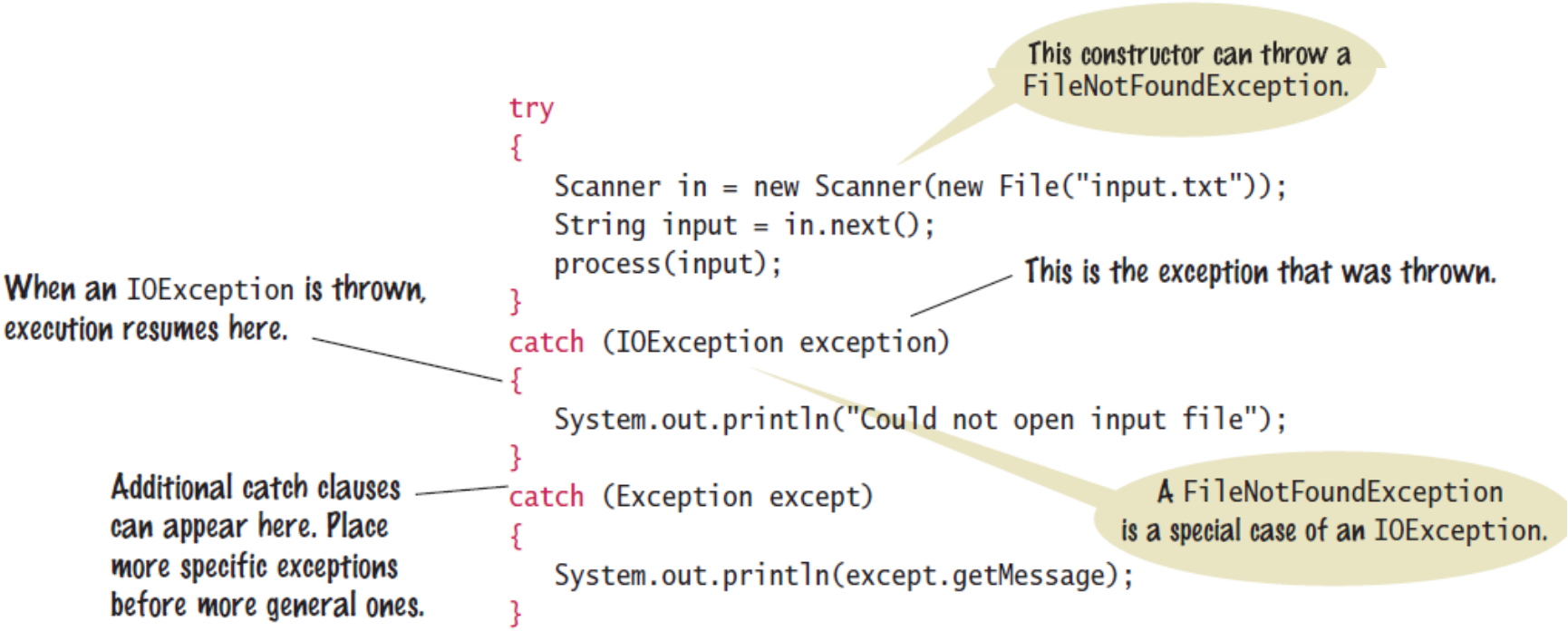
Catching Exceptions

- When an exception is detected, execution 'jumps' immediately to the first matching `catch` block
  - `IOException` matches both `FileNotFoundException` and `NoSuchElementException` is not caught




```
try
{
   String filename = . . .;
   Scanner in = new Scanner(new File(filename));
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
   System.out.println("Input was not a number");
}
```

Syntax 7.2: Catching Exceptions

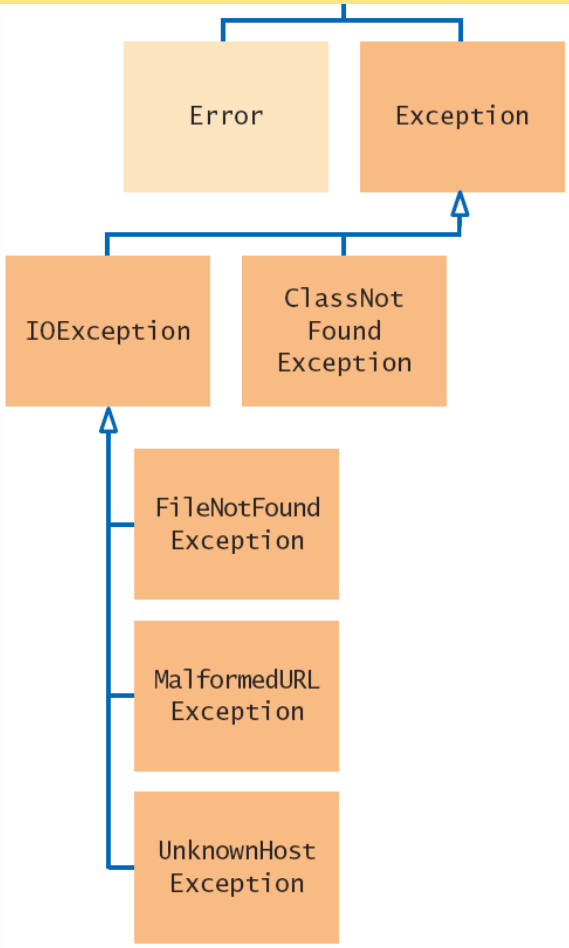


- Some exception handling options:
  - Simply inform the user what is wrong
  - Give the user another chance to correct an input error
  - Print a 'stack trace' showing the list of methods called

```
exception.printStackTrace();
```

Checked Exceptions

- Throw/catch applies to three types of exceptions:
  - **Error:** Internal Errors
    - not considered here
  - **Unchecked:** RunTime Exceptions
    - Caused by the programmer
    - Compiler **does not check** how you handle them
  - **Checked:** All other exceptions
    - Not the programmer's fault
    - Compiler **checks** to make sure you handle these
    - Shown darker in Exception Classes



Checked exceptions are due to circumstances that the programmer cannot prevent.

Syntax 7.3: The `throws` Clause

- Methods that use other methods that may `throw` exceptions must be declared as such
  - Declare all **checked** exceptions a method throws
  - You may also list **unchecked** exceptions

```
public static String readData(String filename)
      throws FileNotFoundException, NumberFormatException
```

You must specify all checked exceptions that this method may throw.

You may also list unchecked exceptions.

The `throws` Clause (continued)

- If a method handles a checked exception internally, it will no longer `throw` the exception.
  - The method does not need to declare it in the `throws` clause
- Declaring exceptions in the `throws` clause ‘passes the buck’ to the calling method to handle it or pass it along.

The `finally` clause

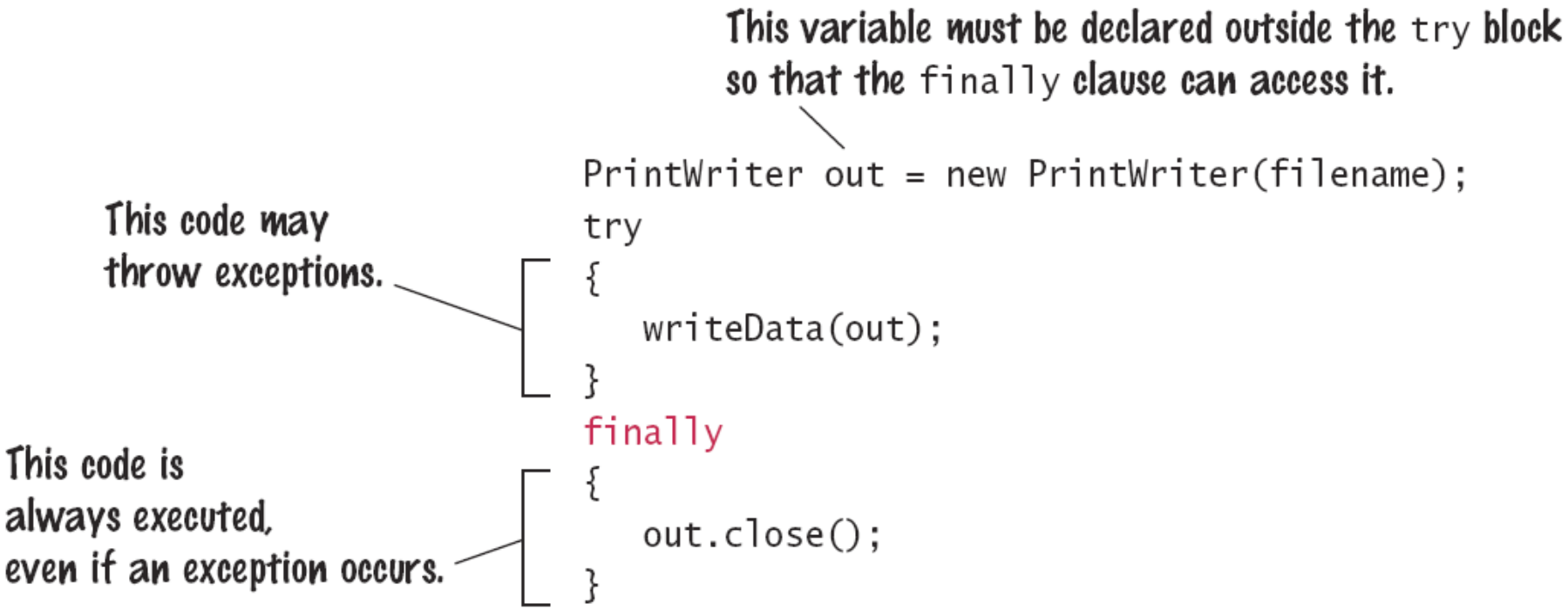
- `finally` is an optional clause in a `try/catch` block
  - Used when you need to take some action in a method whether an exception is thrown or not.
    - The finally block is executed in both cases
  - Example: Close a file in a method in all cases

```
public void printOutput(String filename) throws IOException
{
   PrintWriter out = new PrintWriter(filename);
   try
   {
      writeData(out);   // Method may throw an I/O Exception
   }
   finally
   {
      out.close();
   }
}
```

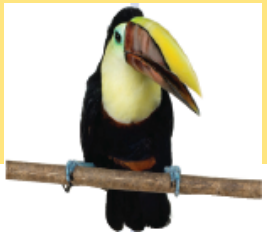
Once a `try` block is entered, the statements in a `finally` clause are guaranteed to be executed, whether or not an exception is thrown.

Syntax 7.4: The `finally` Clause

- Code in the `finally` block is always executed once the `try` block has been entered

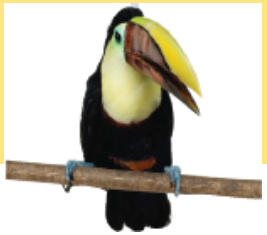


Programming Tip 7.1



- Throw Early
  - When a method detects a problem that it cannot solve, it is better to throw an exception rather than try to come up with an imperfect fix.
- Catch Late
  - Conversely, a method should only catch an exception if it can really remedy the situation.
  - Otherwise, the best remedy is simply to have the exception propagate to its caller, allowing it to be caught by a competent handler.

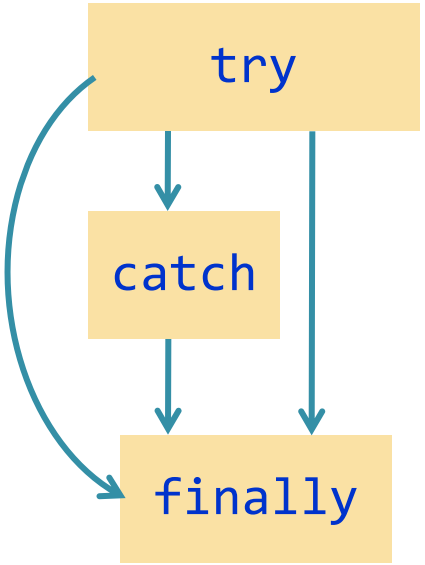
Programming Tip 7.2



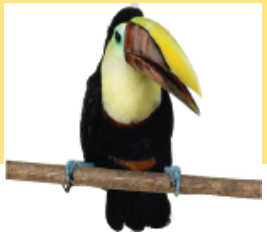
- Do Not Squelch Exceptions
  - When you call a method that throws a checked exception and you haven't specified a handler, the compiler complains.
  - It is tempting to write a 'do-nothing' catch block to 'squelch' the compiler and come back to the code later. **Bad Idea!**
    - Exceptions were designed to transmit problem reports to a competent handler.
    - Installing an incompetent handler simply hides an error condition that could be serious..

Programming Tip 7.3

- Do not use `catch` and `finally` in the same `try` block
  - The `finally` clause is executed whenever the try block is exited in any of three ways:
    1. After completing the last statement of the `try` block
    2. After completing the last statement of a `catch` clause, if this try block caught an exception
    3. When an exception was thrown in the `try` block and not caught

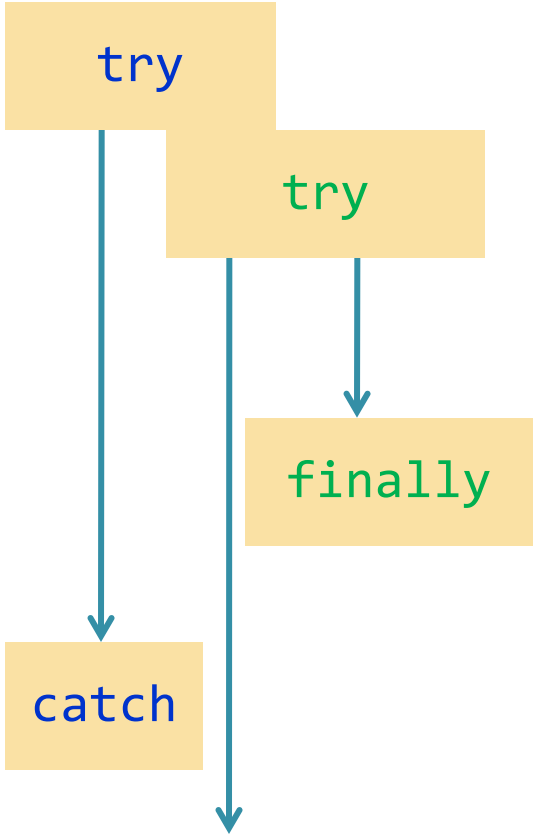


Programming Tip 7.3



- It is better to use two (nested) `try` clauses to control the flow

```
try
{
   PrintWriter out = new PrintWriter(filename);
   try
   {       // Write output    }
   finally
   {  out.close(); }  // Close resources
}
catch (IOException exception)
{
   // Handle exception
}
```



7.5 Handling Input Errors

- File Reading Application Example
  - Goal: Read a file of data values
    - First line is the count of values
    - Remaining lines have values
  - Risks:
    - The file may not exist
      - `Scanner` constructor will throw an exception
      - `FileNotFoundException`
    - The file may have data in the wrong format
      - Doesn't start with a count
        - » `NoSuchElementException`
      - Too many items (count is too low)
        - » `IOException`

```
3
1.45
-2.1
0.05
```

Handling Input Errors: `main`

- Outline for method with all exception handling

```
boolean done = false;
while (!done)
{
  try
  {
    // Prompt user for file name
    double[] data = readFile(filename);   // May throw exceptions
    // Process data
    done = true;
  }
  catch (FileNotFoundException exception)
  {      System.out.println("File not found.");  }
  catch (NoSuchElementException exception)
  {      System.out.println("File contents invalid.");  }
  catch (IOException exception)
  {      exception.printStackTrace();  }
}
```

Handling Input Errors: readFile

- Calls the Scanner constructor
- No exception handling (no catch clauses)
- finally clause closes file in all cases (exception or not)
- throws IOException (back to main)

```
public static double[] readFile(String filename) throws IOException
{
   File inFile = new File(filename);
   Scanner in = new Scanner(inFile);
   try
   {
      return readData(in);   // May throw exceptions
   }
   finally
   {
      in.close();
   }
}
```

Handling Input Errors: readData

- No exception handling (no `try` or `catch` clauses)
- `throw` creates an `IOException` object and exits
- unchecked `NoSuchElementException` can occur

```
public static double[] readData(Scanner in) throws IOException
{
   int numberOfValues = in.nextInt();   // NoSuchElementException
   double[] data = new double[numberOfValues];
   for (int i = 0; i < numberOfValues; i++)
   {
      data[i] = in.nextDouble();         // NoSuchElementException
   }
   if (in.hasNext())
   {
      throw new IOException("End of file expected");
   }
   return data;
}
```

Summary: Input/Output

- Use the `Scanner` class for reading text files.
- When writing text files, use the `PrintWriter` class and the `print/println/printf` methods.
- Close all files when you are done processing them.
- Programs that start from the command line receive command line arguments in the main method.

Summary: Processing Text Files

- The `next` method reads a string that is delimited by white space.
- The `Character` class has methods for classifying characters.
- The `nextLine` method reads an entire line.
- If a string contains the digits of a number, you use the `Integer.parseInt` or `Double.parseDouble` method to obtain the number value.
- Programs that start from the command line receive the command line arguments in the `main` method.

Summary: Exceptions (1)

- To signal an exceptional condition, use the `throw` statement to throw an exception object.
- When you `throw` an exception, processing continues in an exception handler.
- Place statements that can cause an exception inside a `try` block, and the handler inside a `catch` clause.
- Checked exceptions are due to external circumstances that the programmer cannot prevent.
  - The compiler checks that your program handles these exceptions.

Summary: Exceptions (2)

- Add a `throws` clause to a method that can `throw` a checked exception.
- Once a `try` block is entered, the statements in a `finally` clause are guaranteed to be executed, whether or not an exception is thrown.
- `Throw` an exception as soon as a problem is detected.
- `Catch` it only when the problem can be handled.
- When designing a program, ask yourself what kinds of exceptions can occur.
- For each exception, you need to decide which part of your program can competently handle it.

PACKAGES & ACCESS

Packages

- Organizing classes into directories
  - Split up directories that have too many classes
  - Keep related classes together
  - In a Java project,
    - the directory structure must be reflected in the code
    - directories are called packages
    - package names are relative to the ‘base’ directory, where the classes will be compiled and the system executed

Example

```
examples/
        firstProgram/
                FirstProgram.java
              ….
        packages/
            TopLevel.java
            packA/
                A.java
                packA1/
                                A1.java
            packB/
                B.java
```

firstProgram and packages will be base directories.

Packages

- Important packages in the Java library:

Package	Purpose	Sample Class
`java.lang`	Language support	`Math`
`java.util`	Utilities	`Random`
`java.io`	Input and output	`PrintStream`
`java.awt`	Abstract Windowing Toolkit	`Color`
`java.applet`	Applets	`Applet`
`java.net`	Networking	`Socket`
`java.sql`	Database Access	`ResultSet`
`javax.swing`	Swing user interface	`JButton`
`omg.w3c.dom`	Document Object Model for XML documents	`Document`

Organizing Related Classes into Packages (1)

- To put a class in a package, you must place

```
package packageName;
```

  as the first statement in its source

- Package name consists of one or more identifiers separated by periods

Organizing Related Classes into Packages (2)

- For example, to put the `BankAccount` class into a package named `com.horstmann`, the `BankAccount.java` file must start as follows:

```
package com.horstmann;

public class BankAccount
{
   . . .
}
```

- **Default package** has no name, no `package` statement

Importing Packages

- Can always use class without importing:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

- Tedious to use fully qualified name
- Import lets you use shorter class name:

```
import java.util.Scanner;
...
Scanner in = new Scanner(System.in);
```

- Can import all classes in a package:

```
import java.util.*;
```

- Never need to import classes in package `java.lang`
- Don’t need to import other classes in the same package

Package Names

- Use packages to avoid name clashes

  `java.util.Timer`

  vs.

  `javax.swing.Timer`

- Package names should be unambiguous
- Recommendation: start with reversed domain name:

  `com.horstmann`

- `edu.sjsu.cs.walters`: for Britney Walters' classes (`walters@cs.sjsu.edu`)

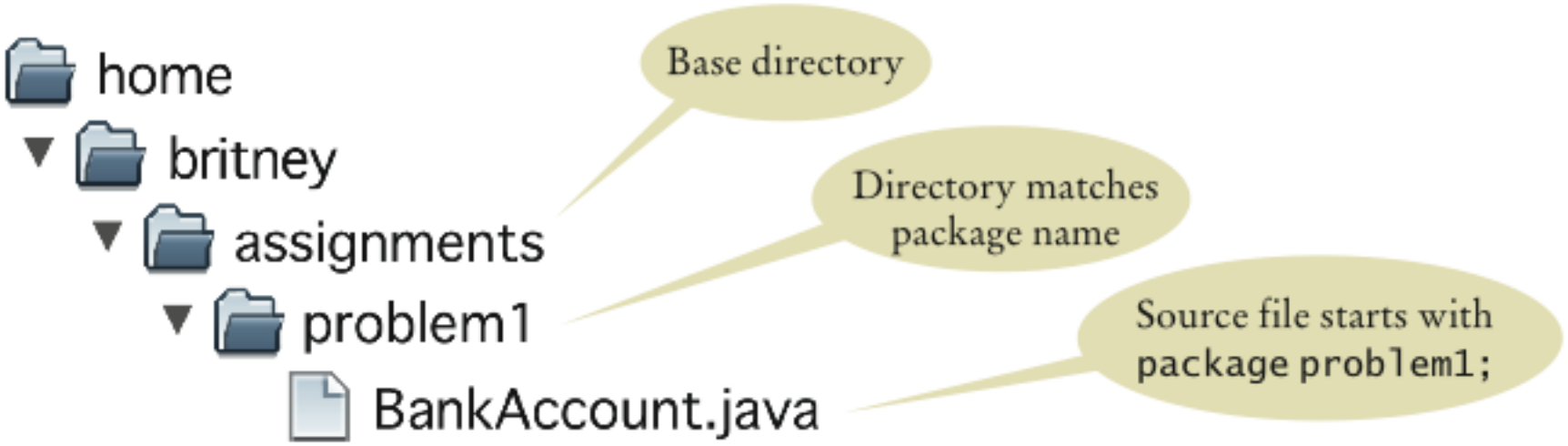
How Classes Are Located

- **Base directory:** holds your program's source files
- Path of a class source file, relative to base directory, must match its package name
- Example: if base directory is

  `/home/britney/assignments`

  place source files for classes in package `problem1` in directory

  `/homehome/britney/assignments/problem1`



Compilation

- A class can refer to any public class of the same package/directory as itself. However, to refer to a class in another package, it needs either an import clause, or the full name of the class (with packages specified), to access any of its members.

```
C:\courses\270\examples\packages>javac TopLevel.java

C:\courses\270\examples\packages>java  TopLevel
Constructing an instance of B
Constructing an instance of A1
Constructing an instance of A1
Constructing an instance of B
Constructing an instance of A1
```

Compilation

- Compilation and execution must occur in the directory that contains the start of the package structure.
  - To compile and execute class A1, you must be in the packages directory, not the packA1 directory.

```
C:\courses\270\examples\packages>javac   packA\packA1\A1.java

C:\courses\270\examples\packages>java   packA.packA1.A1
Constructing an instance of B
Constructing an instance of A1
Constructing an instance of A1
```

Compilation

- Note the following:
  - For **javac**, the argument is uses the **directory notation** of the operating system
  - For **java**, the argument uses the **package notation** of Java

```
C:\courses\270\examples\packages>javac   packA.packA1.A1.java
error: cannot read: packA.packA1.A1.java
1 error

C:\courses\270\examples\packages>java   packA\packA1\A1
Exception in thread "main" java.lang.NoClassDefFoundError:
    packA\packA1\A1
(wrong name packA\packA1\A1)
                at java.lang. …
                at java.lang. …
                …
```

Compilation

- Accessing a class in a package not contained in the base directory.
  - Place the path to the directory containing the package in the CLASSPATH environment/system variable (see Horstmann).

    or
  - Place . (for the current directory) and the path to the directory containing the package in an argument to javac and/or java. For both, use directory notation with ';' as separator in Windows, and ':' in unix.

Access modifiers (public, protected, private)

- **public** : anybody can access
- no modifier : anybody in the same package
- **private** : only access is within the instance or other instances of the same class
- **protected** : anybody in the same package, and descendants in another package in their own implementation
  - (not quite correct; the target of the access must have the type of the descendant)

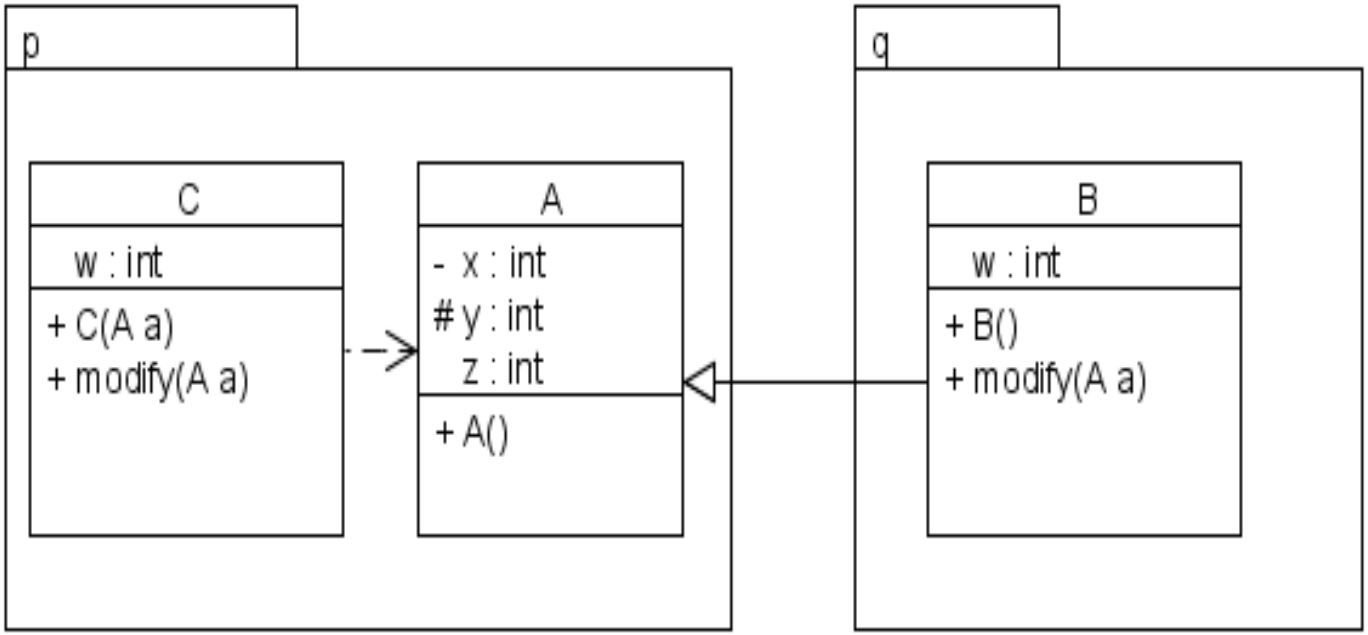
- Classes and constructors are normally public
- Fields are normally **private** or **protected**
- Methods are normally **public** or **private**

```
package p;
public class A
{
        private int x;
        protected int y;
        int z;

        public A()
        {
        }
}
```

```
package p;
public class C
{
        int w;
        public C(A a)
        {
                w = a.x;        // valid?
                a.y = 6;        // valid?
                a.z = 7;        // valid?
                A newA = new A();// valid?
        }
         public void modify(A a)
        {
                a.y = 6; // valid?
        }
}
```

```
package p;
public class A
{
        private int x;
        protected int y;
        int z;

        public A()
        {
        }
}
```

```
package p;
public class C
{
        int w;
        public C(A a)
        {
                w = a.x;        // INVALID
                a.y = 6;        // VALID
                a.z = 7;        // VALID
                A newA = new A();// VALID
        }
         public void modify(A a)
        {
                a.y = 6; // VALID
        }
}
```


p
q
C
w : int
+ C(A a)
+ modify(A a)
A
- x : int
y : int
z : int
+ A()
B
w : int
+ B()
+ modify(A a)

```
package p;
public class A
{
        private int x;
        protected int y;
        int z;

        public A()
        {
        }
}
```

```
package q;
import p.A;
public class B extends A
{
        int w;
        public B()
        {
                w = x;    // valid?
                w = y;    // valid?
                w = z;    // valid?
                B b = new B();  // valid?
                w = b.x;    // valid?
                w = b.y;    // valid?
                A a = new A();  // valid?
                w = a.y;   // valid?
        }
        public void modify(A a)
        {
                w = a.y; // valid?
        }
}
```

```
package p;
public class A
{
        private int x;
        protected int y;
        int z;

        public A()
        {
        }
}
loop
```

```
package q;
import p.A;
public class B extends A
{
        int w;
        public B()
        {
                w = x;    // INVALID
                w = y;    // VALID
                w = z;    // INVALID
                B b = new B();  // Valid, but ∞

                w = b.x;    // INVALID
                w = b.y;    // VALID
                A a = new A();  // VALID
                w = a.y;   // INVALID
        }
        public void modify(A a)
        {
                w = a.y; // INVALID
// (a must have type B, not type A, for otherwise it can refer to another descendant of A)
        }
}
```