Java Programming Unit 12

Collections

Orange Coast College
Computer Science 170
Spring 2004

The Plan For Unit 12

- **Reading**: Ch 8 (342-354, 362-373), Ch 9 (399-426)
- **Arrays**: how to create and process primitive arrays
- **Objects & Vectors**: how to work with object collections
- **Syntax Errors**: working with the Java compiler
- **Runtime Errors**: errors that occur as your program runs
- **Exceptions**: the Throwable classes and try-catch
- **Lab**: OrderPlease shows you an exceptional time
- **Homework**: Modify OrderPlease

Lesson A: Arrays

- How do you store information in your program?
  - In Java, you store information in **variables**
  - A variable stores a **particular kind** of information
  - Ineffective when many related variables are needed

```java
public class CS170
{
   // Attributes
   Student cs170100 = new Student();
   Student cs170101 = new Student();
   Student cs170102 = new Student();
}
```
What Are Arrays?

- An array is like a collection of values
  - Holds multiple values instead of a single value
- An array can hold values of any type
  - Both objects and primitive values
  - Each object or value must be of the same type
- The entire collection shares a single name
  - Individual objects (values) are called elements
  - Elements are accessed by means of their position
  - You'll use a subscript or index to specify position

Array Variables

- In Java, array variables are references
  - A little like object variables
  - Array variables refer to a collection of values

Creating Arrays

- Two step process similar to creating an object
  - To create an object you first declare a variable
  - Then, create an object and assign to the variable

```java
Frog kermit;
public void init()
{
    kermit = new Frog();
}
```

- You create arrays in an almost identical manner
To declare an array variable you specify:

- The **type of elements** you'll store in the array
  - This can be any type, object or primitive
- A **name** for the entire collection
  - Following standard naming rules
- A **set of empty brackets** following either name or type
  - Java programmers tend to favor brackets after type
  - Associates brackets with type rather than with variable

**Step 1: Declare the Variable**

**Declaration Examples**

- Here are some examples
  - `char[] gradeAR;`  
    - Creates an array variable that refers to a group of chars  
    - In this example, the brackets appears after the type
  - `int pieces[];`  
    - Creates an array variable that refers to a group of ints  
    - Note brackets can appear after the variable name
  - `Elephant[] herd;`  
    - Creates an array variable that refers to a herd of Elephants

**Step 2: Create the Array**

- Array is created with `new` just like other objects
  - Use an **array constructor**, not an object constructor
- **Object constructors**:
  - Use the name of the class, followed by parentheses
  - Are actually methods defined inside the class definition
  - May be overloaded [like the `Label` constructors]
  - Are not used to create primitive values like `int`

```java
Button b;
b = new Button("Press Me");
```
Array Constructors
- Use name of class followed by brackets
- Inside the brackets, put the number of elements
- Examples:
  ```java
  gradeAR = new char[5];
  pieces = new int[nTiles * nTiles];
  herd = new Elephant[100];
  ```
- Number can be a constant, variable or expression
- Object arrays hold references, not objects

Array Initialization
- It is common to define variable and create array all in one statement like this:
  ```java
  double[] scores = new double[5];
  ```
- When you create an array the values are set to:
  - 0 if the array holds numeric or char values
  - false if the array holds boolean values
  - null if the array holds object references
- You can create an array by simply providing a set of alternate values using an initializer list

Initializer Lists
- An initializer list is a set of values, separated by commas, enclosed in braces
  ```java
  char[] allGrades = {'A', 'B', 'C', 'D', 'F', 'W', 'I'};
  ```
- When using an initializer list, you:
  - Do not provide a size for the array
  - Java counts the number of elements
  - You do not use the new operator
Accessing Array Elements

- To access an individual array element you use the element's **subscripted name**
  - The array name along with a bracketed subscript
- Store [or retrieve] a value in an array like this:
  
  ```java
  gradeAR[0] = 'A';
  ```

- Subscript can be constant or variable
  
  ```java
  int i = 0;
  char current = gradeAR[ i ];
  ```

Elemental Errors I

- There are two kinds of errors you can make when it comes to storing or retrieving values in an array:
- 1. Store the wrong **kind of thing** in an element
  
  ```java
  gradeAR[0] = 3.2;
  ```

  - This is a **compile-time** error
  - You can't store a **double** in a **char** array

Elemental Errors II

- 2. Use an invalid subscript to access an element
  
  ```java
  gradeAR[5] = 'C';
  ```

  - **gradeAR** has only 5 elements
    - The subscripts go from 0 to 4, not 0 to 5
  - This creates a **runtime** error
    - Your program prints an error message
Array Bounds

- Every array has a public field, length, that stores the number of elements in the array
- When using the length field, remember:
  - Strings have a length() method, arrays have a field
  - String s = "Abara";
  - int lenS = s.length();
  - int lenA = gradeAR.length;

- The array bounds is 0 to length-1

Arrays and Loops I

- Real advantage of arrays?
  - Apply repetitive processing to a group of variables.
  - Need to use a loop
- The for loop is the loop of choice
  - The length field provides a natural count bound
- Common basic array processing loops:
  - Initializing an array
  - Summing the elements in an array
  - Counting the elements in an array
  - Finding the largest or smallest element

Arrays and Loops II

- Example 1: Use a loop to initialize an array
  - Fills an array with random values between 0 and len
  - Assumes randAR is an array of doubles

```java
int len = randAR.length;
for (int idx = 0; idx < len; idx++)
    randAR[idx] = Math.random() * len;
```
Arrays and Loops III

- Example 2: Counting elements in a char array
  ```java
  int As=0, Bs=0, Cs=0, Ds=0, Others=0;
  int len = grades.length;
  for (int i = 0;  i < len;  i++)
      switch (grades[i])
      {
          case 'A': As++; break;
          case 'B': Bs++; break;
          case 'C': Cs++; break;
          case 'D': Ds++; break;
          default: Others++;
      }
  ```

Arrays and Loops IV

- Example 3: Summing the elements in an array
  ```java
  double balance = 0, checks = 0, deposits = 0;
  int len = trans.length; // array of transactions
  for (int i = 0; i < len; i++)
  {
      balance += trans[i];
      if (trans[i] > 0)
          deposits += trans[i];
      else
          checks += trans[i] * -1;
  }
  ```

Passing Arrays to Methods

- Array variables can be passed to methods
  - Declare the formal argument as type[]
  ```java
double findSmallest( double[] ar )
{
    double smallest = ar[0];
    for(int i = 1; i < ar.length; i++)
      if ( ar[ i ] < smallest ) smallest = ar[i];
    return smallest;
}
```
Passing Arrays to Methods

What happens when you pass an array variable to a method? [Assume an array variable named nums]

When passed, the reference to the array is sent to the formal argument (ar) and the actual argument (nums) both refer to the same array. This means arrays MAY be modified inside methods.

void doubleIt(double ar[]) {
    for (int i = 0; i < ar.length; i++)
        ar[i] *= 2;
}

Modifying Array Elements

When passed, the reference to the array is sent to the formal argument (ar) and the actual argument (nums) both refer to the same array. This means arrays MAY be modified inside methods.

Array Copies I

Sometimes this behavior causes problems. In those cases, you need to make a copy of your array. How do you make a copy of an array?

Method 1: The wrong way

```java
int [] copy = ar;
```
Array Copies II

Method 2: Use `new` to create an array of the same size

Then, use a loop and copy each element like this:

```
int [] copy = new int[ar.length];
for (int i = 0; i < ar.length; i++)
    copy[i] = ar[i];
```

This works correctly; only downside is it is fairly slow.

Array Copies II

Method 3: Use the `System.arraycopy()` method

- Optimized native code for your platform
- Takes five arguments:

```
int copy = new int[ ar.length ];
System.arraycopy(ar, 0, copy, 0, ar.length); // Copy from/many elements
```

Returning an Array

In Java, methods can also return arrays, just like objects

- The return type of the method is `type[]`
- Create a local array, process and then return it like this:

```
public double[] makeRandom(int howMany)
{
    double[] ar = new double[ howMany ];
    for(int i = 0; i < ar.length; i++)
        ar[i] = Math.random();
    return ar;
}
```

Introducing Exercise 12A
You create object arrays using the same syntax as that you used for arrays of primitive types:

```java
Button[] toolBar;
toolBar = new Button[5];
```

- Each `toolBar` element contains an object variable.
- Each object variable is set to `null`.
- See illustration on the next slide.

Before you can make use of the `toolBar` array, you must create the actual object elements separately:

```java
int len = toolBar.length;
for (int i = 0; i < len; i++)
{
    toolBar[i] = new Button("" + (i + 1));
}
```

- See illustration on the next slide.
Object Arrays IV

```java
for(i=0; i < 5; i++)
    toolBar[i] = new Button("**+(i+1))
```

TheToolBar Example I

- Longer example using object arrays
- The `toolbar` field is an array of 5 Buttons
- Examine the applet: `ToolBar.java`
- The `init()` method creates the `toolbar` buttons
  - The `Panel p` is used to organize the buttons
  - The `String` array `captions` contains the titles
  - A loop goes through each element of `toolbar`
    - A `Button` constructor is passed a caption
    - An `ActionListener` is attached to the button
    - The button is added to the panel

TheToolBar Example II

- The `actionPerformed()` method
  - How do you tell which button was clicked?
  - In the GuessOMatic we used an if-else-if-else
  - Here, we can use a loop
- General steps:
  1. Which object was clicked?
  2. Loop through each element, save index
  3. Use switch to process the index
Introducing Vectors

- The biggest problem with arrays is the fixed size
  - Means you always have to plan for the worst case
- The Vector class is a “growable” array
- A few limitations
  - Can only hold objects, not primitives
  - Can be less efficient than arrays
  - Must use functions to retrieve and store objects

Creating A Vector

- Vectors are part of the java.util package
- Three basic constructors

  ```java
  Vector v1 = new Vector();
  Vector v2 = new Vector(100);
  Vector v3 = new Vector(100, 25);
  ```

- The capacity of a vector is the elements that it can hold
  - Default Vector() has a capacity of 10
  - The Vector(int) constructor lets you specify size

Vector Resizing

- A Vector will grow as new elements are added
  - The default strategy is to double the capacity when the current size is exceeded.
  - Vector v1 will increase from 10 to 20, then to 40
- You can specify an alternative growth size
  - Vector v3 has an initial capacity of 100 like v2
  - Vector v2 grows from 100 to 200 to 400
  - Vector v3 grows from 100 to 125 to 150
Adding Vector Elements

- There are two methods to add elements

- Button one = new Button("One");
  Button zero = new Button("Zero");
  v1.addElement(one);
  v2.insertElementAt(zero, 0);

- The addElement() method
  - Adds the object to the end of the Vector
  - The Vector grows if necessary
- The insertElementAt() method
  - Inserts the element at the specified position
  - Other elements are moved "up"

Storing Vector Elements

- You can’t use simple assignment like arrays
- If you want to store the Button object named button, then instead of saying:

  toolBar[ idx ] = button;

- you say:

  v1.setElementAt(button, idx);

Retrieving Vector Elements

- Similarly, to retrieve the third element you don’t say:

  Button b = toolBar[ 2 ];

- Instead, you say:

  Button b = (Button) v1.elementAt( 2 );

- Notice that you have to cast the returned value
  - Vector methods return type Object
  - It’s up to you to remember what is really stored there
  - This also means you can’t store primitives in Vector
Other Vector Methods

- Although it might not look like it from the previous slide, Vectors can make your code simpler
- You don't have to explicitly size them
- Class has many methods that require "work" with an array
- `contains()` to see if an element exists in an Vector
- `indexOf()` to retrieve the position of an element
- `removeElementAt()` to remove an element and "close up" the gap left by its removal

- Example: `ToolBarV.java`
- Introducing Exercise 12B

Lesson C: Compile-Time Errors

- Programs can have different "kinds" of errors
- What are compile-time errors?
  - Errors that violate Java's "rules of grammar"
  - Can be detected when you compile your program
- What are other kinds of errors?
  - Run-time errors can cause your program to crash
  - Logic errors which produce the wrong output
- Prefer compile-time to run-time errors
  - Allows you to catch errors before shipping product

Error1.java

```java
import java.applet.*;
import java.awt.*;
public class Error1 extends Applet
{
    int num;
    num = 32;
    public static void main(String args[])
    {
        int var;
        System.out.println("var = " + var);
        System.out.println("Now " + num);
        return true;
    }
}
```

Step 1: Examination
Can you find the errors?
Step 2: Compile the Code

What do these error messages mean?

Reading Error Messages

- How do you read a compiler error message?
  - File containing error and line number
  - Description of error message
  - Exact place on line where error occurred

- Two problems with error messages
  - One error can produce other "cascading" errors
    - Wrong java.Applet.*; import produces second
  - One error can hide other errors
    - Invalid return hidden because of parentheses

Understanding Compiler Errors

- There are four common categories of errors
  - Categorizing your error is the first step in locating, understanding, and fixing it

- The four basic compile-time error categories:
  - Structural errors
  - Typographical errors
  - "Real" syntax errors
  - Type errors
Structural Errors I

- Java compiler does not process file sequentially
- Doesn't work like a wood-chipper or juicer
- Compiling your code is a multi-step process
- First step is to organize your code
- Compiler breaks your source-code into different kinds of "chunks"
- Organization dependent on matching delimiters

Error2.java

```java
import java.applet.*;
import java.awt.*;
public class Error2 extends Applet {
    int num;
    num = 32;
    public static void main(String args[])
    {
        integer var;
        System.out.println("var = + var");
        System.out.println("Now ' + num);
        return TRUE;
    }
}
```

Step 1: Examination

How many errors?

Examining Error2.java

- Here are the errors you should have found
  - ink and integer used instead of int
  - C++ pointer-to-member operator used
  - Keywords public and static are capitalized
  - A void function returns TRUE [What's TRUE?]
  - An assignment is performed outside a method
  - Parentheses, not braces, delimit class and method
  - String constants not correctly delimited
  - How many errors can the compiler find?
Examining Error2.java

The Result?
Fewer errors than Error1

Structural Errors II

- Not really only 4 errors
  - Because Java can’t tell where your class begins and ends, it can’t examine the individual elements, such as misspelled type names, contained inside the class
  - Once you fix the structural errors, the other errors will appear
- Fixing structural errors is the first step
  - Match all parentheses, braces, and quotes

Structural Errors III

- Three steps for avoiding structural errors
  1) Know what goes where
     - Import and package statement must appear first
     - Field definitions must be inside class, outside method
     - Method definitions must be inside a class
     - Program statements must appear inside a method
     - Formal arguments go inside parentheses
  2) Write structural statements in one piece
     - Add opening and closing braces at same time
  3) Use consistent and easily checked indent style
"Real" Syntax Errors
- Once Java has "chunked" your code, it makes sure each chunk is in the right place: these are "real" syntax errors
- Some examples of real syntax errors
  - Using keywords in an inappropriate place
    ```java
    int double a;
    int a = 10, int b = 20;
    ```
  - Omitting a keyword when it is necessary
    ```java
    void myMethod(int a, b) { }
    ```
  - Only solution: memorization, attention to detail

Typographical Errors
- Typing your code incorrectly [Misspelling]
- Java usually pinpoints these fairly accurately
- Three most common typographical errors:
  - Missing semicolon
    - Small, hard to see, and easily confused with :
  - Using assignment [ ] instead of equality [ ]
  - Using the wrong case
    - Keywords must be in lower case
    - You must spell your identifiers consistently

Type Errors
- Caused when you attempt to store a value of one type into a variable designed for another type
- Three rules to remember to avoid type errors
  - You can store little values in big buckets
    - Store int in long variable, but not vice-versa
  - You can store less-precise values in more-precise buckets
    - Store float in double variable, but not vice-versa
  - Store similar things in similar buckets
    - Numeric variables can't store booleans or objects
Finding the Errors I

- Where do you start when you get 25 errors?
  - Often, first errors scroll off the screen (2K/XP improved)
  - You need a strategy to systematically attack errors
- Start JCreator, then download Error3.java
- Step 1: Fix structural errors first
  - "Comment out" entire class body using /* */
  - If errors still appear then:
    - Made an error in the basic structure of your class
    - Made an error in one of your import statements
    - Made an error in your class header

Finding the Errors II

- Step 2: Reveal the attributes
  - Move the opening comment character so that the compiler can see the attributes
  - This assumes you have your attributes at the beginning
  - Recompile and fix errors
  - Errors occur because of mistakes in declaring variables
- Step 3: Divide and Conquer
  - Reveal each method, one by one
  - Fix errors in each method as they occur
- Introducing Exercise 12C

Lesson D: Runtime Errors

- Runtime errors occur when your program runs
  - Cannot be caught by the Java compiler
- Three categories of runtime errors
  - Errors that cause the JVM to print an error message
    - These are called runtime exceptions
    - May cause your program to "crash"
  - Errors that cause your program to hang
  - Errors that cause your program to behave incorrectly
    - These are called logical errors
Runtime Exceptions I

- JVM notices a problem and notifies you
  - Provides opportunity intercept and correct
- Example: `RuntimeError.java` [Run the applet]
  - Open your Java Console window so you can see
  - Press the "Nobody's Home" button
  - JVM Message: `NullPointerException`
- What is a `NullPointerException`?
  - Simply means an "uninitialized object"
- Solution: Fix your code

Runtime Exceptions II

- Press the "Problems with Math" button
  - JVM reports an `ArithmeticException`
  - Caused by divide by zero
  - Solution? Don't do that!!! [Fix your code]
- Press the "How Much is a Cat" button
  - Different than the other two exceptions
  - Not designed to protect integrity of JVM or universe
  - `NumberFormatException` : means "I don't know how"
  - Solution? Use try-catch to recover

Hanging Programs

- Programs that stop responding
  - Obvious that something is wrong
  - No helpful message from the JVM
- Two general reasons for hanging programs
  - You have written an endless loop
  - You have a "system" error
Endless Loops

- There are two common causes of endless loops
  - The classic reason: forgetting to update the condition inside the loop body.
  - The unavoidable reason: you've written a boolean condition that can never be false.

```java
while (age > 21 || age < 65) {}
```

- Usually means you've confused AND and OR.

System Errors

- Errors that cause the O/S to stop responding.
  - Not usually caused by your program.
- Two ways to cause a system error:
  - Deplete a necessary O/S resource
    - In Windows, for instance, you can run out of Graphics resources by creating Color objects inside a loop or by forgetting to dispose of Graphics objects obtained by calling the getGraphics() method.
  - Cause a deadlock
    - Occurs when two processes are waiting for the operating system to deliver same resource.

Regaining Control

- Some increasingly desperate suggestions for recovering from endless loops and system errors
  - Try the application's "Stop", "Exit" or "Quit" button
  - Try to load a different page into your browser
  - Click the "close" button in applet viewer
  - Select a command window and press CTRL+C
  - Exit your browser
  - Use CTRL+ALT+DELETE and cancel the task
  - Turn off the power
Logical Errors

- Logical errors are the most difficult to fix
- Cause your program to run incorrectly
- No error messages are printed
- No obvious malfunction such as hanging
- Incorrect results only produced sporadically

Solution for logical problems?
- Call it a "feature" [ancient computer joke]
- Rigorous testing to find logical errors
- Debugging strategy to remove them
- Introducing Exercise 12D

Lesson E: Exceptional Situations

- Sometimes, runtime errors are caused by unusual situations, not programming errors
- For example: writing data to a file
  - Most of the time things go uneventfully, but...
    - The disk may be full
    - There may be a hardware error
    - The file may have been changed to read-only
- Fragile code ignores the possibility of problems
- Robust code anticipates such problems

The Traditional Method I

- Tradition method is to use "completion codes"

```
GET A FILENAME
OPEN THE FILE
IF THERE IS NO ERROR OPENING THE FILE
READ SOME DATA
IF THERE IS NO ERROR READING THE DATA
PROCESS THE DATA
WRITE THE DATA
IF THERE IS NO ERROR WRITING THE DATA
CLOSE THE FILE
IF THERE IS NO ERROR CLOSING FILE
RETURN
```
The Traditional Method II

- Things to notice with completion codes
  - Almost every step may fail, except "PROCESS DATA"
  - Program just be prepared to deal with failure
  - Very difficult to determine "normal" actions
    - Program so concerned with things that can go wrong that it is hard to tell if you are doing the right things in the right order
  - Difficult to use if library function contains such code
    - Function must return several error codes so that user can tell what went wrong

The Exceptional Method

- Using exceptions the same code looks like this:

```java
try {
    get a filename
    open the file
    read some data
    process the data
    write the data
    close the file
    return
} catch (Exception e) {
    if error opening the file then ...
    if error reading the data then ...
    if error writing the data then ...
    if error closing the file then ...
}
```

The Advantage of Exceptions

- Compare the two methods of handling "bad things"
- Code that uses the exception strategy:
  - Is shorter and easier to read
  - Makes the normal logic the main focus of the method
    - In the completion code version, the error-handling code obscures the normal actions
  - Allows you to decide whether to handle the problem or defer handling the error on a case-by-case basis
    - Ideal for library-based code
Exception and Error Objects

- In Java, when a runtime error occurs:
  - The JVM notices the problem and creates one of two types of Throwable objects
    - An "Error" object or an "Exception" object
  - The Throwable object is thrown back up the call stack until a method is found that wants to catch it
  - If it is not "caught" then the JVM runtime system prints an error message
- See illustration on the next slide

Following Exceptions

Error and Exception Classes
The Error Classes
- Notice the different kinds of Error classes
  - InternalError
  - OutOfMemoryError
  - UnknownError
- Fatal situations that you cannot normally deal with
  - How do you fix an “Unknown” error, for instance?
- Normally, ignore the Error classes
- Simply allow to percolate up to runtime

The Exception Classes
- Situations you may want to handle in your code
- Two general categories
  - Unchecked exceptions [RuntimeException]
    - You are not required to specifically deal with these
    - Most of the time you, treat them like Error object
    - NumberFormatException is one you probably will want to handle in your own code
  - Checked exceptions
    - Exceptions that will commonly occur
    - You must explicitly handle all checked exceptions

Using try-catch
- In Java, you handle exceptions with try-catch
  - Put the code that may “fail” into a “try” block.
  - Provide one or more ‘catch’ blocks immediately following the “try” block
  - You should provide catch blocks for all errors you want to handle. Do this by choosing to catch particular classes.
  - If you don’t provide a catch block for a particular exception, then the exception will be propagated
The try-catch Syntax

The try block I

- The try block consists of:
  - The keyword try
  - Followed by a brace-delimited block
  - Braces are required: part of the syntax
  - This is unlike loops and if statements
- Inside the try block:
  - Place any number of statements that may throw an exception
  - If a method throws a checked exception, it must be placed in a try block [or be thrown]

The try Block II

- A try-block example
  - Methods throwing checked exceptions (IOException and InterruptedException) must be placed in a try
  - Unchecked exceptions may be placed in a try

```java
int 1 = 3, b = 0, n = 0, x = 0;
try {
    n = System.in.read(); // IOException
    Thread.sleep(100); // InterruptedException
    x = a / b; // ArithmeticException
}
```
**The catch Block I**

- Place one or more immediately following try
- Syntax for catch block is a little tricky

```java
    catch (SomeException1 e1)
    {
        // Handle exception 1 here
    }
    catch (SomeOtherException e2)
    {
        // Handle exception 2 here
    }
```

- Exception var names must be unique to try-catch block

**The catch Block II**

- You must provide a catch block for every checked exception that may be thrown inside your try block.
- This example fails to deal with `InterruptedException`
- Can also catch the `Exception` superclass

```java
    int 1 = 3, b = 0, n = 0, x = 0;
    try
    {
        n = System.in.read(); // IOException
        Thread.sleep(100); // InterruptedException
        x = a / b; // ArithmeticException
    }
    catch (IOException ioe) {/*System.in.read failure */}
```

**Catching Exceptions**

- Use appropriate catch blocks
- Fix uninitialized objects
- Fix arithmetic errors
- Fix array errors
- Catch generic exceptions
- Catch specific exceptions
- Handle all exceptions
An Alternative to try-catch

- You don't have to use `try...catch`
- Instead, you can "throw" the exception
- Add `throws Exception` to method definition
- You can throw any kind of exception
- Means that caller must use `try-catch` or throw the exception as well, but your code doesn't have to
  - If caller doesn't handle, will eventually be handled by the JVM, just like errors and unchecked exceptions
- See illustration on next slide

Using the throws Heading

```java
public int readInt() throws IOException {
    int result = 0;
    String s = "";
    while ( ch != ' ' & ch != '\r' ) {
        s += (char) ch;
        ch = System.in.read();
    }
    result = Integer.parseInt(s);
    return result;
}
```

- When an exception is thrown
  - Control jumps to the `catch` block that handles it
  - Code that manages resources may be skipped

- The `try-catch` has an optional `finally` part
  - This will always be called
  - Use to handle resource depletion

Now, Finally, finally
Using finally

A "resource" is allocated in the try block

```java
try {
    Resource r = new Resource();
    doSomethingWith(r);
} catch (SomeException e) {
    // cope with tragic failure
} finally {
    r.dispose();
}
```

Exception Strategies

- When to catch and when to throw?
  - If you know how to handle a situation, use try-catch
  - If you don't know how to handle an error, or,
  - If you think there may be situations where users of your code will want to handle an error differently, then use throw.
- Use exceptions for exceptional situations only
  - Example: Don't use exceptions as loop control
  - Introducing Exercise 12E

Finish Up

- Assignments due on May 11
- Quiz 12, Chapter 9 Lab [OrderPlease]
- Homework 11 [ColorTicketI]
- Homework 12 [OrderPlease II] due May 18
  - Make use add a valid zip code and credit-card date
  - Add exception class for each occurrence
  - Use try-catch to handle errors
  - Tell users how to fix their errors