CISC 370: Inheritance, Abstract Classes, Exceptions

June 15, 2006

Review

- Quizzes
  - Grades on CPM
- Conventions
  - Class names are capitalized
  - Object names/variables are lower case
    - String.doStuff();
    - string.doStuff();
- Encapsulation
- Inheritance
- Packages

Encapsulation

- Hide implementation details
  - can change the implementation details without programmer's relying on it
- Protect against accidental or willful stupidity
  - may have interdependent fields
  - can't modify one field without updating the others
    - to keep in a consistent state --> method can do all necessary steps

Encapsulation

- If fields can be manipulated directly, the number of possibilities you have to test becomes unmanageable
  - What could the user do?
  - Can use methods to perform error checking
    - What if user set chicken to have a negative height?
- Internal fields and methods visible outside the class clutter up API
  - makes class tidy and easier to use and understand

Inheritance

- Build new classes based on existing classes
  - Allows you to reuse code
- Use `extends` keyword to make a subclass
- The is a relationship
  - Classic mark of inheritance
- Constructor chaining

Inheritance

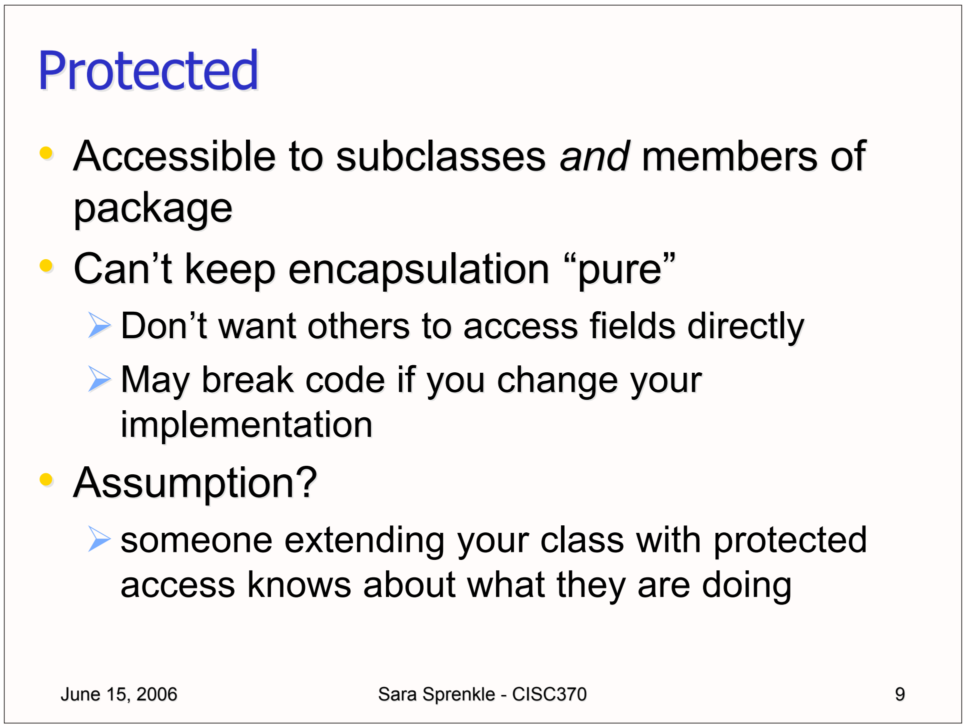
- Access modifiers in subclasses
  - can make access to subclass more restrictive but not less restrictive
- Class fields and methods are *not* inherited
- Constructors are not inherited
  - We had to define Rooster(String name, int height, double weight) event though similar constructor in Chicken

Inheritance

- If you're uncertain which to use (protected, package, or private), use the most restrictive
  - Changing to less restrictive is easy
  - Changing to more restrictive may break the code that uses your classes.

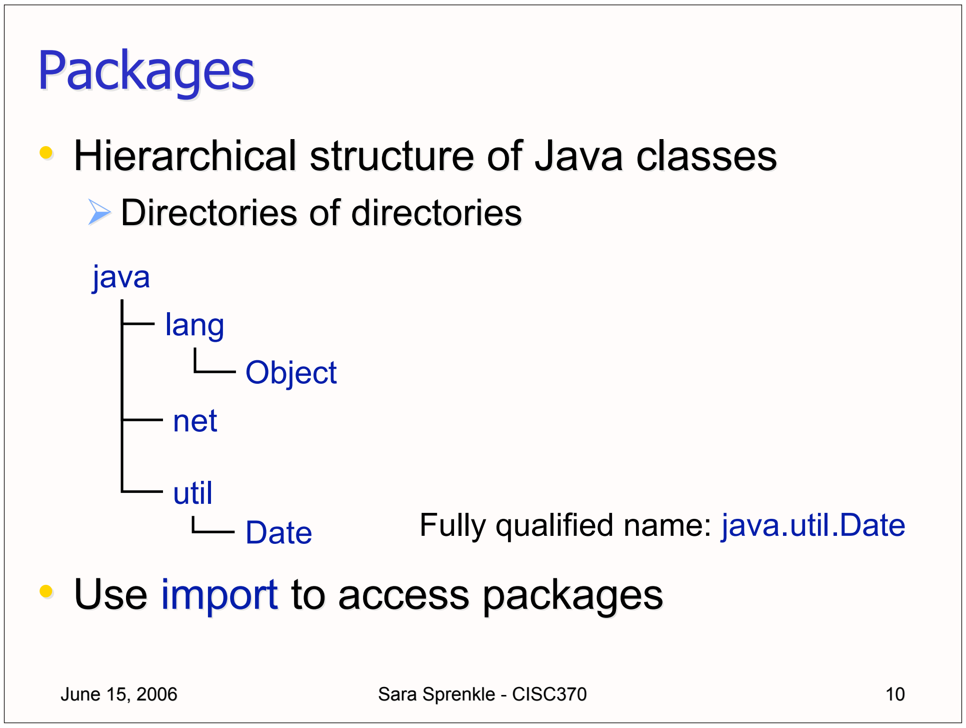
Member Visibility

Accessible to	Member Visibility			
	Public	Protected	Package	Private
Defining Class	Yes	Yes	Yes	Yes
Class in same package	Yes	Yes	Yes	No
Subclass in different package	Yes	Yes	No	No
Non-subclass different package	Yes	No	No	No



Protected

- Accessible to subclasses *and* members of package
- Can’t keep encapsulation “pure”
  - Don’t want others to access fields directly
  - May break code if you change your implementation
- Assumption?
  - someone extending your class with protected access knows about what they are doing



Packages

- Hierarchical structure of Java classes
  - Directories of directories

```
java
 ├── lang
 │     └── Object
 ├── net
 └── util
       └── Date
```

Fully qualified name: java.util.Date

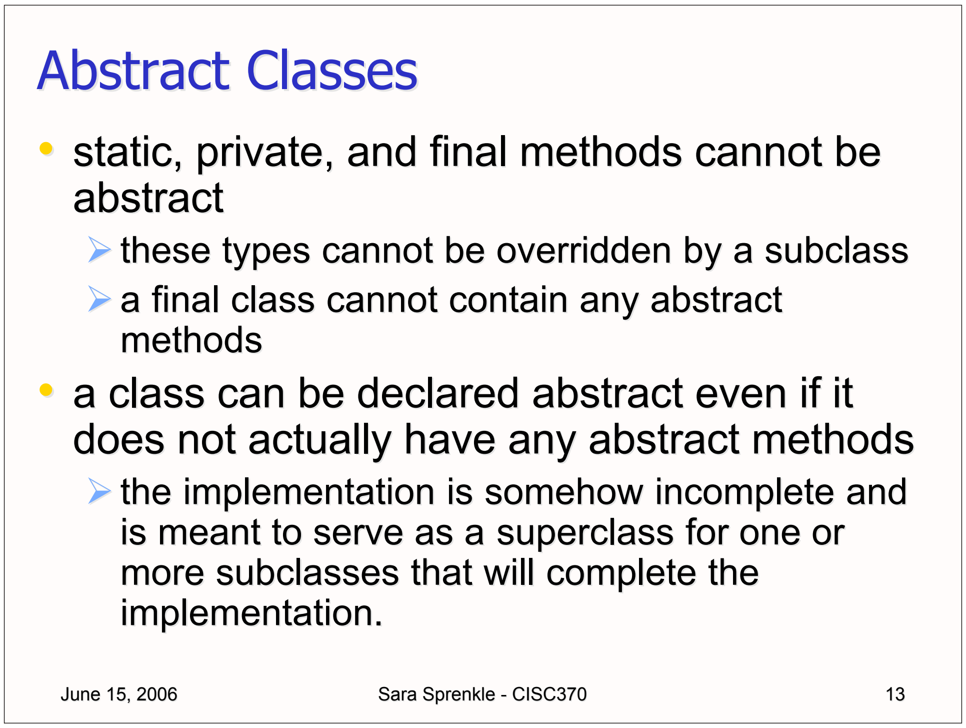
- Use import to access packages

Abstract Classes

- Some methods defined, others not defined
- Classes in which not all methods are implemented are *abstract classes*.
  - `public abstract class ZooAnimal`
- Blank methods are labeled with the *abstract* keyword also
  - `public abstract void exercise();`

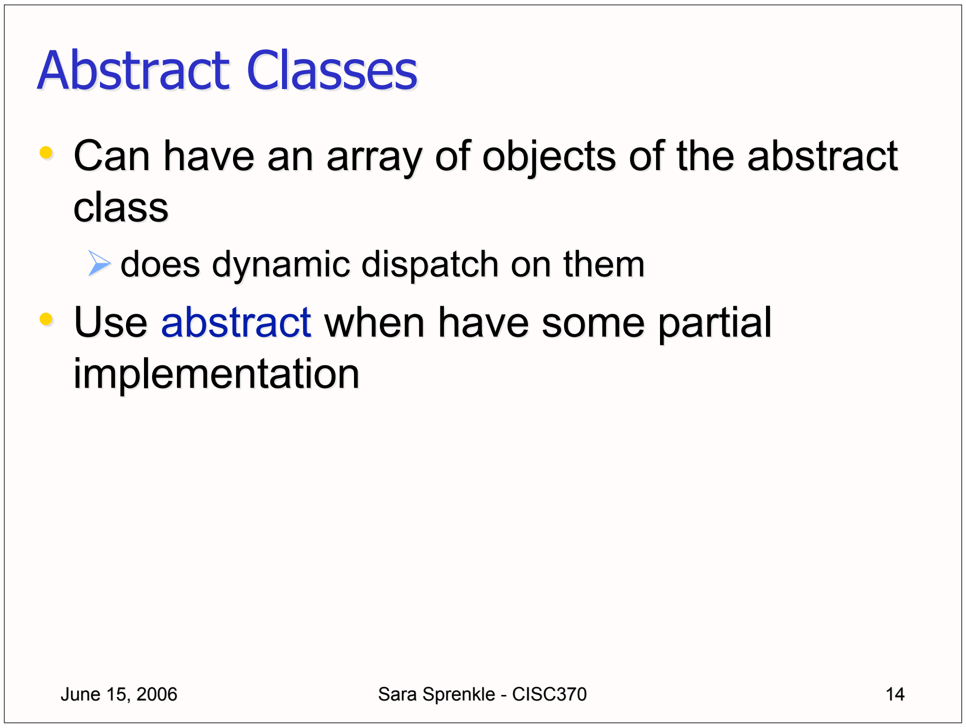
Abstract Classes

- An abstract class cannot be instantiated
- Subclass of an abstract class can only be instantiated if it overrides **each** of the **abstract methods** of its superclass and provides **implementation**
  - If subclass does not override all abstract methods, it is also abstract



Abstract Classes

- static, private, and final methods cannot be abstract
  - these types cannot be overridden by a subclass
  - a final class cannot contain any abstract methods
- a class can be declared abstract even if it does not actually have any abstract methods
  - the implementation is somehow incomplete and is meant to serve as a superclass for one or more subclasses that will complete the implementation.



Abstract Classes

- Can have an array of objects of the abstract class
  - does dynamic dispatch on them
- Use abstract when have some partial implementation

Examples of abstract classes

- Define the abstract methods
  - Example 1:
    - java.net.Socket
    - java.net.SSLSocket (abstract)
  - Example 2:
    - java.util.Calendar (abstract)
    - java.util.GregorianCalendar

Interfaces

- Like abstract classes with only abstract methods
  - A set of requirements for classes to conform to
- Pure specification, no implementation
- Classes implement one or more interfaces.

Example of an Interface

- We have seen before how to make an array of Chicken object variables.
- We can call the Arrays.sort() method, a method of the Arrays class
- Arrays.sort() has the ability to sort arrays of any object class.
  - Need a way to decide if one object is less than, greater than, or equal to another object.
  - Class of objects must be comparable.
- Comparable is an interface…

java.lang.Comparable

```
public interface Comparable
{
        int compareTo(Object other);
}
```

- Any object that is Comparable must have a method named compareTo(), which takes an Object as a parameter and returns an integer
  - < 0 for less than
  - 0 for equals
  - > 0 for greater than

Implementing an Interface

- To make a class implement an interface
  - In the class definition, you need to specify that the class will implement the specific interface.
  - You provide a definition for all of the methods specified in the interface.
- An interface is very similar to an abstract (or virtual) class in C++…
  - a set of requirements that any implementing class must have

How to determine Chicken order?

- What if made the Chicken class Comparable?

Comparable Chickens

```
class Chicken implements Comparable
{
  . . .
  public int compareTo(Object otherObject)
  {
      Chicken other = (Chicken)otherObject;
      if (height < other.getHeight() )  return -1;
      if (height > other.getHeight())   return  1;
      return  0;
  }
}
```

One way: order by height

What if otherObject is not a Chicken?

Comparable Chickens

```
class Chicken implements Comparable
{
  . . .
  public int compareTo(Object otherObject)
  {
      Chicken other = (Chicken)otherObject;
      if (height < other.getHeight() )  return -1;
      if (height > other.getHeight())   return  1;
      if( weight < other.getWeight())   return -1;
      if (weight  > other.getWeight())  return 1;
      return  0;
  }
}
```

Order by height, then weight
Could have more conditions for “breaking ties” --> comparing names

Comparable Interface in Java Docs

- API documentation says what the compareTo() method should do:
  - Return a –1 if the first object is less than the second object (passed as a parameter)
  - Return a 1 if the second object (passed as a parameter) is less than the first object
  - Return a 0 if the two objects are equal
- Can see what Java library classes implement Comparable

Interfaces

- only object (not class) methods
- all are public methods
  - implied if not explicit
  - error to have protected or private (Why?)
- fields are constants that are static and final
- Can implement multiple interfaces
  - separated by commas in definition

Testing for Interfaces

- We can also use the `instanceof` operator to see if an object implements a particular interface
  - e.g., to determine if an object can be compared to another object using the Comparable interface.

```
if (obj instanceof Comparable) {
      // runs if whatever class obj is an instance of
      // implements the Comparable interface
}
else {
      // runs if it does not implement the interface
}
```

Interface Object Variables

- We can use an object variable to refer to an object of any class that implements an interface
- Using this object variable, we can only access methods that are present in the interface.
- For example…

```
Object obj;
…
if (obj instanceof Comparable)
{
      Comparable comp = (Comparable)obj;
      boolean res = comp.compareTo(obj2);
}
```

Interface Definitions

```
public interface Comparable
{
        int compareTo(Object other);
}
```

- We do not need to specify the methods as public
  - All interface methods are public by default

Interface Definitions and Inheritance

- We can also extend interfaces
  - allows a chain of interfaces that go from general to more specific with each step
- For example, let's define an interface for a object which is capable of moving:

```
public interface Movable
{
        void move(double x, double y);
}
```

Interface Definitions and Inheritance

- A powered vehicle is also Movable
  - it must also have a MPG() method, which will return its gas mileage

```
public interface Powered extends Movable
{
        double miles_per_gallon();
}
```

June 15, 2006 Sara Sprenkle - CISC370 29

Constants in an Interface

- If a variable is specified in an interface, it is automatically a constant
  - `public static final` variable

```
public interface Powered extends Movable
{
        double miles_per_gallon();
        double SPEED_LIMIT = 95;
}
```

- An object that implements the Powered interface has a constant SPEED_LIMIT defined

June 15, 2006 Sara Sprenkle - CISC370 30

Interface Definitions and Inheritance

- Powered interface extends the Movable interface.
- Any object that implements the Powered interface must satisfy all the requirements of that interface as well as its superinterface.
  - A Powered object must have a miles_per_gallon() and move() method

Multiple Interfaces

- A class can implement multiple interfaces
  - An interface is a promise to implement given methods
  - Can have more than one interface and fulfill the requirements of each one.
- But, NOT possible with inheritance
  - a class can only extend (or inherit from) one class.

```
public final class String implements
     Serializable, Comparable, CharSequence { …
```

Using Interfaces

- Common use:
  - define constants for multiple classes/package
  - Something like global constants
- Marker Interface
  - Interface that is empty
  - Use to identify an object that has a certain property

Using interface or abstract class?

- Interfaces
  - Any class can use (can implement multiple interfaces)
  - no implementation
  - Implementing lots of methods multiple times can be annoying
  - Adding a method will break classes that implement
- Abstract class
  - Can contain partial implementation
  - Can’t extend/subclass multiple classes
  - Can add non-abstract methods without breaking subclasses

One option: Use Both!

- Define interface, e.g., MyInterface
- Define abstract class, e.g., AbstractMyInterface
  - implements interface
  - provides implementation for some methods

Exceptions

Errors

- Programs encounter errors when they run.
  - Users may enter data in the wrong form
  - files that should exist sometimes do not
  - printers run out of paper in the middle of printing
  - program code always has bugs.
- Errors are bad. When one happens, your program should do one of two things:
  - Revert to a stable state and continue.
  - Allow the user to save data and then exit the program gracefully.

Error Codes – Why They Don't Always Work

- The traditional method of indicating an error in a method (function) call is to return a specific sentinel value.
  - the read() function in C returns a –1 if the read was unsuccessful
- What is the general problem with sentinels?
- What does a function that returns an integer return in the case of an error?
  - It is not always possible to return an error code, when an error has occurred in a method.

Methods: An Alternate Ending

- Java allows a method to take an alternate exit path if it is unable to complete its task in the normal, correct way.
- A method can opt to not return a value.
  - Instead it throws an object that encapsulates the error information.
- Exception: the object that is thrown
- A method can return its specified return type (the normal/correct case) or it can throw an exception (the error case).

Methods: An Alternate Ending

- If a method throws an exception
  - it does not return anything
  - execution does not resume immediately following the method call (as it would if the method returns a normal value)
- JVM's **exception-handling mechanism** searches for an **exception handler**
  - Exception handler: error recovery code
    - runs to deal with a particular error condition.

Exception Classification

- All exceptions indirectly derive from a class **Throwable**.
  - Subclasses: **Error** and **Exception**
- Important Throwable methods
  - getMessage
    - Detailed message about error
  - printStackTrace
    - Prints out where problem occurred and path to reach that point
    - Also getStackTrace to get the stack in non-text format

Exception Classification: Error

- **Error** is an internal error
  - JVM-generated in the case of resource exhaustion or an internal problem
    - Out of Memory error (When can that happen?)
  - Program's code should not and can not throw an object of this type.
  - Unchecked exception

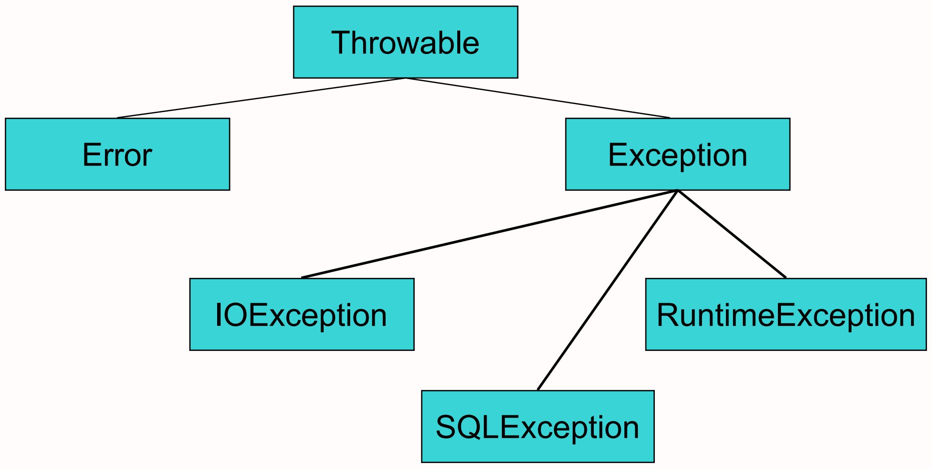
Exception Classification

- An **Exception** is the kind of **Throwable** objects programs deal with.
  - **RuntimeException** something that happens due to a programming error you made
    - Unchecked exception
    - ArrayOutOfBoundsException ← Seen before
    - NullPointerException
    - ClassCastException
  - Lots of checked exceptions
    - e.g., **IOException, SQLException**

Exception Classification

- So, if something is programmer's fault
  - **RuntimeException**.
  - otherwise, an **Error** or another **Exception**.
- Common: **IOException**
  - trying to read past the end of a file
  - trying to open a bad URL
  - File not found
  - etc…

Exception Classification



Checked and Unchecked

- Unchecked: any exception that derives from **Error** or **RuntimeException**
- Checked: any other exception, e.g., from **IOException**
- Programmer need to create and handle checked exceptions
  - not unchecked exceptions (except to try to make sure that they don't occur in the first place!)

Unchecked Exceptions

- Two types of unchecked exceptions:
  - Derived from the class **Error**:
    - Any line of code in a Java program can generate this because it is internal
    - You don't need to worry about what to do if this happens.
  - Derived from the class **RuntimeException**
    - Indicates a bug in the program
    - Don't worry about what to do if it happens
      - fix the bug!

Checked Exceptions

- Need to be handled in your program
- Advertise the exceptions that a particular method throws
  - For each method, tell the compiler:
    - what the method returns
    - what could possibly go wrong
- As an example, java.io.BufferedReader

The BufferedReader Class

- contains a method, `readLine()`, which reads a line from a stream, such as a file or network connection
- Its header looks like:

```
public String readLine() throws
    IOException
```

- readLine can
  - return a String (if everything went right)
  - throw an IOException (if something went wrong)

Programmer-Defined Methods

- Advertise only the checked methods that your method can throw
  - Your method calls a method that throws a checked exception
  - Your method detects an error in its processing and decides to throw an exception

Passing an Exception Up

- So, if we were to write a method which calls the readLine() method of a BufferedReader:

```
String readData(BufferedReader in)
   throws IOException
{
     String str1;
     str1 = in.readLine();
     return str1;
}
```

Passing an Exception Up

```
String readData(BufferedReader in)
   throws IOException
{
     String str1;
     str1 = in.readLine();
     return str1;
}
```

Throws the IOException

- Our readData() method calls a method that can throw an IOException
- readLine() will throw this exception to us
  - Assuming we don't want to deal with exceptions, we simply throw the exception as well
    - whoever called readData will handle exception

Throwing Our Own Exception

- If we have a program which is reading a file byte-by-byte. We know in advance how big this file is supposed to be.
- What do we do if we reach an EOF byte while we should still have data to read in?
- We need to generate our own exception.

Throwing Our Own Exception

- For example…

```
String readBytes(BufferedReader in, int num_bytes)
     throws EOFException
{
      while (. . .)
      {
        if (char_in == EOF)
        {
            if (number_read < num_bytes)
                  throw new EOFException();
        }
        . . .
      }
      . . .
}
```

Fibanacci.java

Throwing Our Own Exception

```
if (num_read < num_bytes)
        throw new EOFException();
```

- If we encounter an EOF, we make a new object of class **EOFException**
  - class derived from **IOException**
- After making exception object, we throw it
  - The method ends at this point
  - The calling program needs to deal with our exception, which tells it that we encountered an EOF before we should have.

A More Descriptive Exception

- There are actually two constructors for **all** Exception classes
  - default (no parameters)
  - one that takes a String
    - describe the condition that generated this exception more fully

```
if (num_read < num_bytes)
{
        String gripe = "I read " + num_read +
                " when I should have read " + num_bytes;
        throw new EOFException(gripe);
}
```

Creating Our Own Exception Class

- The **EOFException** class described the error our method encountered well.
  - not always the case.
  - Many exceptions derived from **IOException**, but plenty more conditions.
- What do you do when you cannot find a predefined exception class the describes your condition?
  - Make a new exception class!

Creating Our Own Exception Class

```
public class FileFormatException extends IOException
{
        public FileFormatException()
        { }

        public FileFormatException(String gripe)
        {
                super(gripe);
        }
}
```

- Now, we are ready to throw exceptions of type **FileFormatException**

Catching Exceptions

- After we throw an exception, some part of our program needs to *catch* it
  - some part of our program
    - knows how to deal with the situation that caused the exception
    - receives it
    - handles the problem
      - Hopefully gracefully, without exiting

The try/catch Block

- The simplest way to catch an exception is to use a *try/catch* block
- Simplest form of this block looks like:

```
try {
        code;
        more code;
}
catch (ExceptionType e)
{
        error code for ExceptionType
}
```

Try/Catch Block

- The code in the try block runs first
  - If it completes without an exception, the catch block(s) are skipped
  - If the try code generates an exception, a catch block runs
    - remaining code in the try block is skipped.

The try/catch Block

```
try {
        code;
        more code;
}
catch (ExceptionType e)
{
        error code for
        ExceptionType
}
```

- If the code inside the try {} block does *not* throw an exception of ExceptionType, the catch {} block is skipped.
- If an exception of a type other than ExceptionType is thrown inside the try {} block, the method exits immediately and the program dies.

The try/catch Block

```
try {
        code;
        more code;
}
catch (ExceptionType1 e)
{
        error code
        for ExceptionType
}
catch (ExceptionType2 e)
{
        error code
        for ExceptionType
}
```

- You can have more than one catch {} block.
  - lets you handle more than one type of exception that can be thrown inside your try {} block.
- If ExceptionType1 does not catch the exception, it falls to ExceptionType2, and so forth
  - run the first matching catch {} block.

Can catch any type with **Exception e**, but won't have customized messages

try/catch … an Example

```
public void read(BufferedReader in)
{
        try {
                boolean done = false;
                while (!done)
                {
                        String line=in.readLine();
                        // this could throw IOException!
                        if (line == null)
                                done = true;
                }
        }
        catch (IOException exp) {
                exp.printStackTrace();
        }
}
```

Prints out stack trace to method call that caused the error

try/catch … an Example

```
public void read(BufferedReader in)
{
        boolean done = false;
        while (!done)
        {
                try {
                        String line=in.readLine();
                        // this could throw IOException!
                        if (line == null)
                                done = true;
                }
                catch (IOException exp) {
                        exp.printStackTrace();
                }
        }
}
```

More precise try/catch may help pinpoint error
But could result in messier code

The finally Block

- Can add a **finally** block after all possible catch blocks
  - Code in finally block **always** runs, after the code in the try and/or catch blocks
    - after the try block finishes, or if an exception occurs, after the catch block finishes.
- Allows you to clean up or do maintenance before the method ends (one way or the other)
  - E.g., closing database connections

The try/catch/finally Blocks

```
try {
        statement1;
        statement2;
}
catch (EOFException e)
{
        statement3;
        statement4;
}
finally
{
        statement5;
        statement6;
}
```

- Which statements run if:
  - Neither statement 1 nor statement 2 throws an exception
  - Statement 1 throws an EOFException
  - Statement 2 throws an EOFException
  - Statement 1 throws an IOException

What to do with a Caught Exception?

- We dump the stack after the exception occurs
  - What else can we do?
- Often, the best answer is to do nothing but report the problem
- If an exception occurs in the readLine() method, our read() method should probably pass up to whoever called it
- Instead of catching this exception, simply advertise that the read() method can throw an IOException. Let whoever calls the read() method catch and handle the exception.

A Further Example

```
public void read(BufferedReader in)
        throws IOException
{
       boolean done = false;
       while (!done)
       {
              String line=in.readLine();
              // this could throw IOException!
              if (line == null)
                     done = true;
       }
}
```

Checked Exceptions

- Why are these called checked exceptions?
  - the compiler *checks* to make sure you deal with such an exception.
- If you call a method that could generate a checked exception, you can either
  - catch and handle it, or
  - have your method throw the exception up to whoever called it by advertising the exception
  - You MUST do one of these two things

Methods Throwing Exceptions

- The online API documentation will tell you if a method can throw an exception.
  - If so, you must handle it
- If your method could possibly throw an exception (by generating it or by calling another method that could), advertise it!
  - If you can't handle all sorts of errors, that's OK…let whoever is calling you worry about it. However, they can only do that if you advertise any exceptions you can't deal with.

Programming with Exceptions

- Exception handling is slow
- Use one big *try* block instead of nesting try-catch blocks too deep
- Don't ignore exceptions
  - it's better to pass them along to higher calls

Benefits of exceptions?

Benefits of Exceptions

- Force error checking/handling
  - Otherwise, won't compile
  - Does not guarantee "good" exception handling
- Ease debugging
  - Stack trace