Exceptions

or Handling Things when things go wrong

Before we begin

- Are there any questions on inheritance

Today's class

- Exceptions
  - A means for handling "exceptional" situations.

Back to our payroll app

- Recall from the Payroll class:

```
public void addPerformer (Performer P)
{
  if (nPerf == MAXPERF)
    System.err.println ("Payroll is full.");
  else {
     performer[nPerf] = P;
     nPerf++;
  }
}
```

- Suppose the caller of addPerformer wants to do something else besides just printing an error message?

Exceptions

- Exceptions allow a method to tell the caller when an error has occurred
  - Many times it is the calling function that knows what to do when an error occurs.
  - Exceptions allow the caller to respond to the error rather than the method itself.
  - Different callers may wish to respond to particular errors differently.

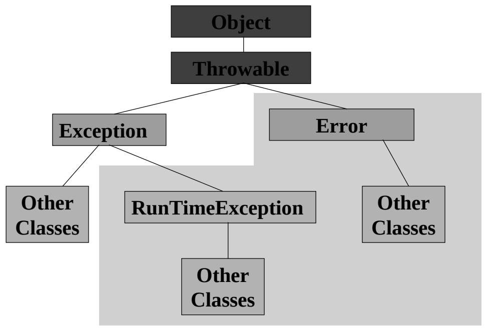
Exceptions

- When an error occurs, an exception will be *thrown*.
- When an exception is thrown, the exception gets passed to the calling function.
- This function may:
  - *Catch* the exception, then perform whatever error handling is appropriate or
  - Pass the exception up the call stack to the function that called it.
- If an exception reaches the main method and is not caught and handled, the program will terminate.

Exceptions in Java

- In Java, an *exception object*, holding information about the error, is created and thrown.
- This object contains:
  - A snapshot of the program when the error occurred
  - An optional error message

Exceptions in Java



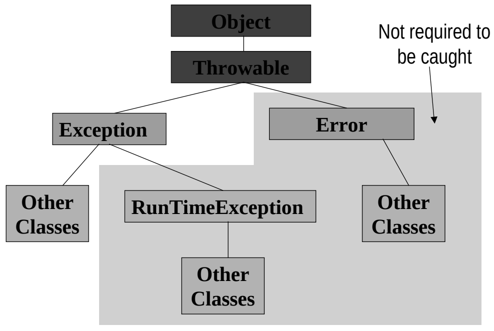
Exceptions in Java

- `Throwable`
  - top class in hierarchy
- `Error`
  - Thrown when a very serious condition occurs
  - Not expected to be caught or dealt with
- `Exception`
  - Errors that can and should be caught

Exceptions in Java

- `RunTimeException`
  - Exceptions that are not required to be caught.
    - ArithmeticException
    - IndexOutOfBoundsException
    - NullPointerException
- All `Exception`s that are not dervived from `RunTimeException` must either be caught or “declared”

Exceptions in Java



Throwing Exceptions

- A method can throw an Exception by using the throw clause.
- If a method throws a non-runtime exception, the definition of the function must “declare” that an exception can be thrown.
- When a method throws an exception, execution in that method ceases.

Throwing Exceptions

- Payroll app
  - Let’s throw an exception when the Payroll is overfilled instead of issuing an error message

```
public void addPerformer (Performer P)
{
  if (nPerf == MAXPERF)
    System.err.println ("Payroll is full.”);
  else {
     performer[nPerf] = P;
     nPerf++;
  }
}
```

Throwing Exceptions

```
public class PayrollFullException extends Exception
{
    public PayrollFullException (String msg)
   {super (msg);}
}


public class Payroll {...
public void addPerformer (Performer P)
   throws PayrollFullException
{
  if (nPerf == MAXPERF)
    throw new PayrollFullException(“I am full”);

  performer[nPerf] = P;
  nPerf++;
}
}
```

Throwing Exceptions

```
/**
  * Adds a performer to the payroll.  Will throw
  * an exception if the payroll is currently full.
  *
  * @param P the performer to be added
  * @exception PayrollFullException if payroll is full
  */
public void addPerformer (Performer P)
     throws PayrollFullException
{
  if (nPerf == MAXPERF)
    throw new PayrollFullException();

  performer[nPerf] = P;
  nPerf++;
}
```

Throwing Exceptions

```
/**
  * Adds a performer to the payroll.  Will throw
  * an exception if the payroll is currently full.
  *
  * @param P the performer to be added
  * @throws PayrollFullException if payroll is full
  */
public void addPerformer (Performer P)
     throws PayrollFullException
{
  if (nPerf == MAXPERF)
    throw new PayrollFullException();

  performer[nPerf] = P;
  nPerf++;
}
```

Catching Exceptions

- You catch and handle exceptions by using the `try/catch/finally` statement

`try {`

statement(s) that can throw exceptions

`}`

`catch (`*ExceptionClass* `E) {`

statements that handles exception

`}`

`finally {`

cleanup code

`}`

Catching Exceptions

```
public void doUpdate (Performer P)
{
    try {
       addPerformer (P);
    }
    catch (PayrollFullException E)
    {
       // print out the error message
       System.out.println (E.getMessage());

       // do whatever else must be done…
     }
}
```

Catching Exceptions

- The last example will only catch Exceptions of class `PayrollFullException`
- One can catch Exceptions of multiple classes, each with a different handler by using the general form of the `try/catch/finally` statement.

Catching Multiple Exceptions

```
try {statement(s)}
catch (ExceptionClass1 name1) {…}
catch (ExceptionClass2 name2) {…}
catch (ExceptionClass3 name3) {…}
...
catch (ExceptionClassn namen) {…}
finally {cleanup code }
```

Catching Multiple Exceptions

- Of course, since all exceptions are subclasses of the Exception class, you can catch all exceptions:

```
try {statement(s)}
catch (Exception E) {…}
finally {…}
```

In this case, exceptions of all types will be handled the same

Catching Multiple Exceptions

- When catching multiple exceptions in a single try/catch statement
  - The more specific Exceptions must be listed first.

Catching Multiple Exceptions

```
public FooException extends Exception {…}
----------------------------------------------
try {
  // call a function that throws a Foo Exception
}
catch (Exception E) { // do something}
catch (FooException F) { // do something else}
```

Will cause a compile error

Polymorphism and Exceptions

- Polymorphism does indeed work when declaring what exceptions are thrown by a function

```
public void doit (int a) throws
Exception
 {
   if (a > 0) throw new FooException();
     else throw new OofException();
  }
```

Polymorphism and Exceptions

- However, functions calling doit must catch exceptions declared by doit.

```
try {
  doit(10);
}
catch (FooException E) { // do something}
catch (OofException F) { // do something else}
```

Would cause a compile error

Polymorphism and Exceptions

- This is okay

```
try {
  doit(10);
}
catch (FooException F) { // do something}
catch (OofException O) { // do something else}
catch (Exception E) { // do even something else}
```

Catching exceptions

- How do you know what exceptions need to be caught?
  - Check javadocs for objects whose methods you are calling
  - Let the compiler do the checking.

Passing on exceptions

- A method M that calls a method P that throws an exception may choose not to catch the exception.
  - The exception will get passed to the caller of M.
  - If P throws an exception that is not a `RunTimeException`, M must declare that it too can throw an exception

Passing on Exceptions

```
public void doUpdate (Performer P)
{
    addPerformer (P);
}
```

Would cause a compiler error since `addPerformer` throws a `PayrollFullException` and `doUpdate` doesn't catch it.

Passing on Exceptions

Instead, if `doUpdate` wants to pass on this exception it must declare that it can throw a `PayrollFullException`

```
public void doUpdate (Performer P)
  throws PayrollFullException
{
    addPerformer (P);
}
```

Back to Payroll

- Note that we could have left out the error check altogether.

```
public void addPerformer (Performer P)
{
  performer[nPerf] = P;
  nPerf++;
}
```

- What would happen when the payroll gets overfilled?

Finally `finally`

- The `finally` clause is optional, and is not frequently used
- It allows for cleanup of actions that occurred in the `try` block but may remain undone if an exception is caught
- Code in the finally block will get called regardless of whether an exception is caught or not
- Most useful when there is more than 1 exit from a function

Finally `finally`

```
try {
   // some code that opens a window
   openWindow();
}
catch (MildException M)
{
   // do some handling, but okay to continue after
   // handling error
}
catch (BadException B)
{
   // do some exception handling..but leave function
   // since error is to severe to carry on
   ...
   return;
}

// close the window opened in the try block
closeWindow();
```

Finally `finally`

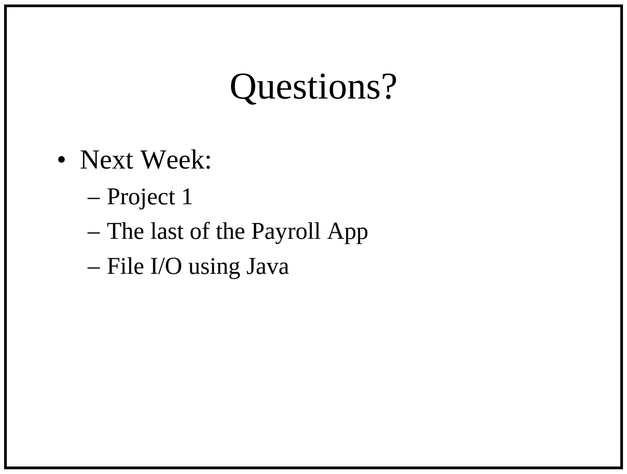
```
 try {
   // some code that opens a window
   openWindow();
}
catch (MildException M)
{
   // do some handling, but okay to continue after
   // handling error
}
catch (BadException B)
{
   // do some exception handling..but leave function
   // since error is to severe to carry on
   ...
   // must assure window gets closed
   closeWindow();
   return;
}
// close the window opened in the try block
closeWindow();
```

Finally `finally`

```
try {
     // some code that opens a window
     openWindow();
  }
  catch (MildException M)
  {
     // do some handling, but okay to continue after
     // handling error
  }
  catch (BadException B)
  {
     // do some exception handling..but leave function
     // since error is to severe to carry on
     ...
     return;
  }
 finally {
     // close the window opened in the try block
     // will get called no matter if an exception
     // is caught or not
     closeWindow();
  }
```

Summary

- Exceptions
- `Throwable` hierarchy
- Throwing exceptions
- Catching exceptions
  - `try / catch / finally`
- Passing exceptions on



Questions?

- Next Week:
  - Project 1
  - The last of the Payroll App
  - File I/O using Java