Exceptions

Traditional Error Handling

- Calling method checks a return value
- Might be a numeric or true/false code
  - OpenFileMethod(); // returns T/F
  - ReadFile(); // will fail if return code not checked

Traditional Error Handling

```
public bool CreateDatabase()
{
  if (CreatePhysicalDatabase())
    if (CreateTables())
      if (CreateIndexes())
        return true;
      else
        handle error;
        return false;
    else
      handle error; return false;
  else
    handle error; return false;
}
```

Exceptions

- Using exceptions is the way to handle error conditions in your code.
  - Console.WriteLine("Enter amount to convert:");
  - string s = Console.ReadLine();
  - int num = Convert.ToInt32(s); ← may throw System.FormatException
- And this will abort your program if the error is not caught and handled somehow.

Exceptions

```
try
{
  string s = Console.ReadLine();
  int num = Convert.ToInt32(s);
}
catch (FormatException fe)
{
  Console.WriteLine("Bad Input " + fe.Message);
}
```

Exceptions

- Compare to previous traditional example

```
try
{
  GenerateDatabase()
}
catch (Exception e)
{
  handle error – can construct e so that it tells us
   who failed
}
```

Exceptions

- Where GenerateDatabase consists of calls to:
- CreatePhysicalDatabase()
- CreateTables()
- CreateIndexes()

- If any cause an error, will be caught by our error handler.

Exceptions

- Unlike error codes, exceptions are impossible to ignore.
- Another advantage – Constructing objects

System.Exception

- Base class for all exceptions
- If need to cause an exception, simplest form is:

```
Exception e = new Exception();
throw e;
```

Exception block structure

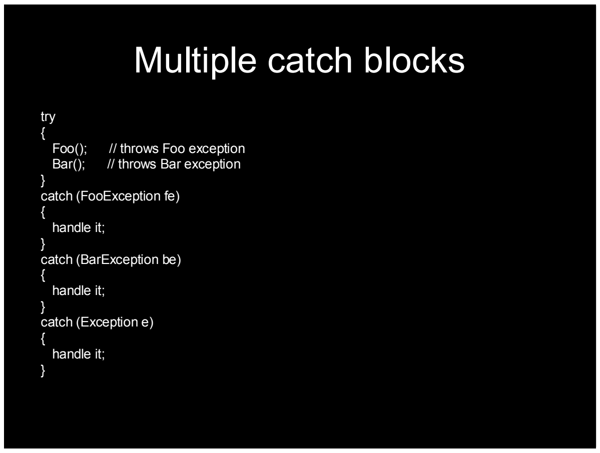
```
try
{
   code that you hope will succeed but might fail
   you can even call other methods here that may
    cause exceptions
}
catch (SomeException exceptionObjectName)
{
   code to deal with the problem
}
```

Rethrowing an exception

```
try
{
   Foo();
}
catch (Exception e)
{
   do some processing with e;
   throw;  // rethrows exception e
}
```

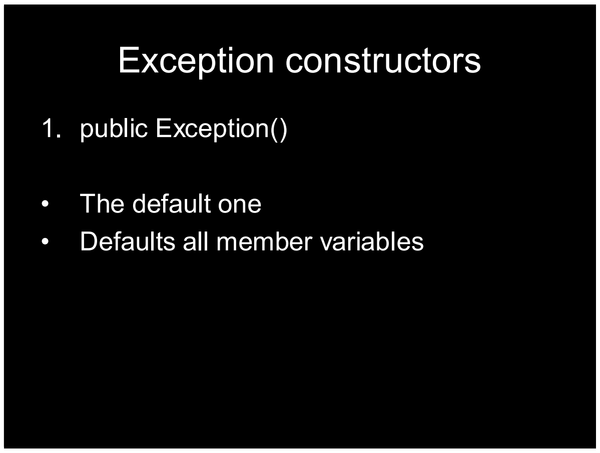
try..catch..finally

```
acquire file handle
try
{
   do something with file;
}
catch (Exception e)
{
   handle exception;
}
finally
{
   release file resource
}
```



Multiple catch blocks

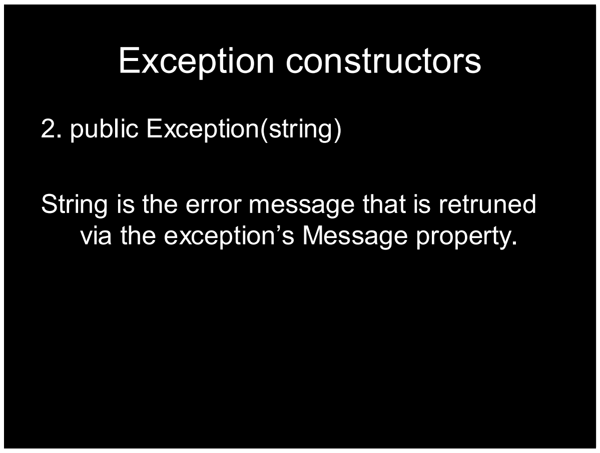
```
try
{
   Foo();     // throws Foo exception
   Bar();     // throws Bar exception
}
catch (FooException fe)
{
   handle it;
}
catch (BarException be)
{
   handle it;
}
catch (Exception e)
{
   handle it;
}
```



Exception constructors

1. public Exception()

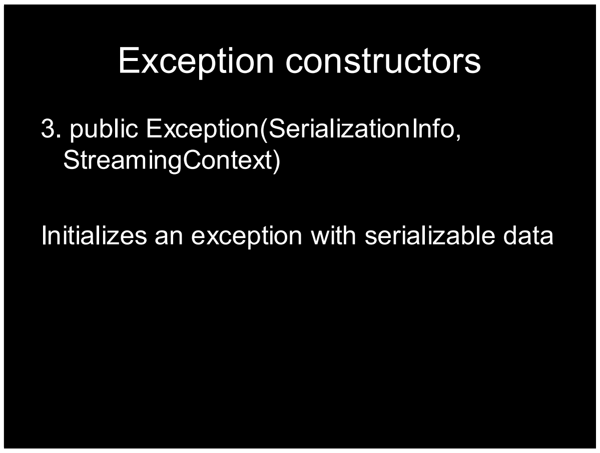
- The default one
- Defaults all member variables



Exception constructors

2. public Exception(string)

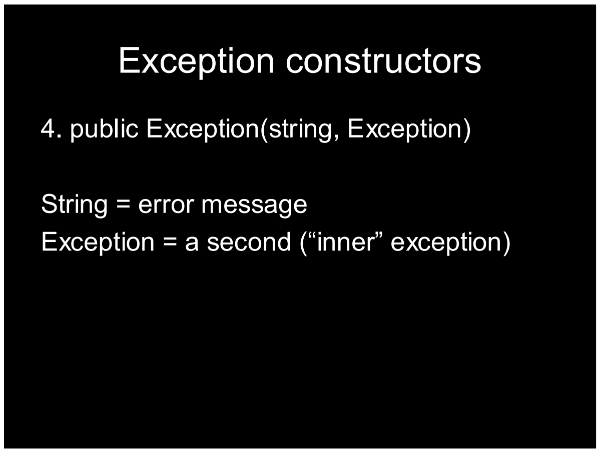
String is the error message that is retruned via the exception's Message property.



Exception constructors

3. public Exception(SerializationInfo, StreamingContext)

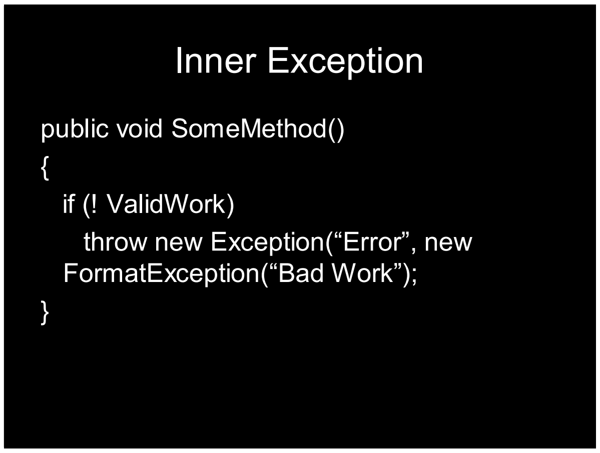
Initializes an exception with serializable data



Exception constructors

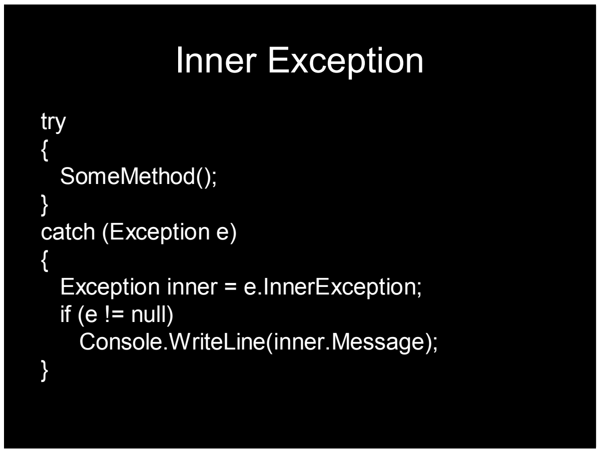
4. public Exception(string, Exception)

String = error message
Exception = a second ("inner" exception)



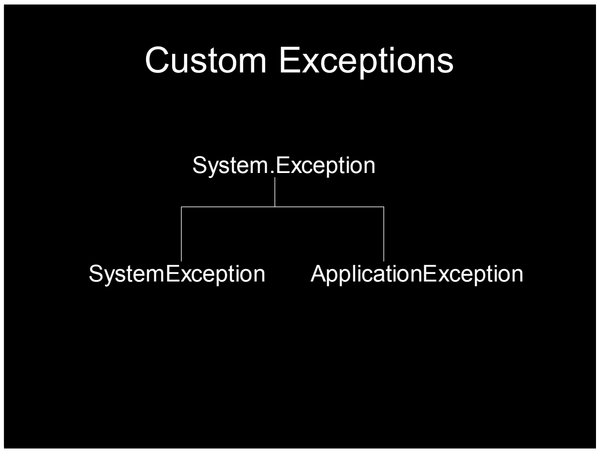
Inner Exception

```
public void SomeMethod()
{
   if (! ValidWork)
      throw new Exception("Error", new
   FormatException("Bad Work");
}
```



Inner Exception

```
try
{
  SomeMethod();
}
catch (Exception e)
{
  Exception inner = e.InnerException;
  if (e != null)
    Console.WriteLine(inner.Message);
}
```



Custom Exceptions