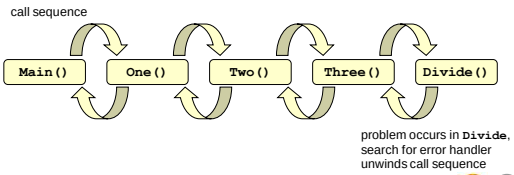
Exceptions

developmentor

Exception pattern

- **Exceptions are a notification mechanism**
- **Pattern:**
  - exception generated when condition detected
  - propagated back through the call chain
  - caught and handled at higher level



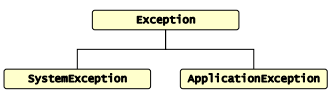
Predefined exceptions

- **.NET Framework defines many exception types**
  - for common conditions
  - implemented as library classes
  - names end in **`Exception`**

```
class ArithmeticException       ... { ... }
class FileNotFoundException     ... { ... }
class IndexOutOfRangeException  ... { ... }
class InvalidCastException      ... { ... }
class NullReferenceException    ... { ... }
class OutOfMemoryException      ... { ... }
```


Exception types

- **Exceptions organized into inheritance hierarchy**
  - all are derived from **`Exception`**
  - system exceptions from **`SystemException`**
  - user defined exceptions from **`ApplicationException`**



Exception generation

- **Exception generated with keyword `throw`**
  - must create and throw exception object
  - can be generated by user code, CLR, or .NET Framework

user code →
```
int Divide(int n, int d)
{
  if (d == 0)
    throw new DivideByZeroException("divide by zero");

  return n / d;
}
```

CLR →
```
int[] data;
...
data[index] = 17;
```

Framework →
```
FileStream f = new FileStream("MyFile.dat", FileMode.Open);
```


Catching exceptions

- **User code can handle exception**
  - regular code placed in **`try`** block
  - handler code put inside **`catch`** clause

error →
code skipped →
execute handler →
```
void Process()
{
  try
  {
    int[] data = new int[10];

    data[10] = 17;
    ...
  }
  catch (IndexOutOfRangeException e)
  {
    ...
  }
}
```

Multiple catch

- A `try` block can have multiple `catch` clauses
  - tested in order
  - only matching handler executed

handle failure of `new` →
handle invalid index →

```
void Process(int index)
{
  try
  {
    int[] data = new int[10];
    data[index] = 17;
    ...
  }
  catch (OutOfMemoryException e)
  {
    ...
  }
  catch (IndexOutOfRangeException e)
  {
    ...
  }
}
```


Locating a handler

- Appropriate handler chosen by type thrown
  - unwind stack until caught
  - non-matching handlers ignored
  - thread terminates if exception not caught

```
void One()
{
  try
  {
    Two();
  }
  catch (DivideByZeroException e)
  {
    ...
  }
}
```

```
void Two()
{
  try
  {
    Three();
  }
  catch (IOException e)
  {
    ...
  }
}
```

```
void Three()
{
  int q;

  q = Divide(3, 0);

  ...
}
```

will be executed

throws exception

Exception objects

- Exception objects contain information about error
  - `Message` and `StackTrace` defined in `Exception` class
  - available in all exception types
  - specific exception types may add more information

access exception info →

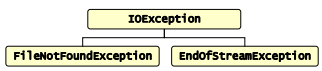
```
void Process()
{
  try
  {
    ...
  }
  catch (IndexOutOfRangeException e)
  {
    string msg   = e.Message;

    string trace = e.StackTrace;
  }
}
```


Catching base class

- Can catch base class
  - also catches all derived class

catch `IOException` and all derived classes →

```
void Process()
{
  try
  {
    ...
  }
  catch (IOException e)
  {
    ...
  }
}
```

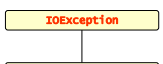


Catch clause ordering

- Most specific `catch` must be listed first
  - derived class before base
  - otherwise base clause would catch all
  - derived clause would never be reached

derived →
base →

```
void Process()
{
  try
  {
    ...
  }
  catch (FileNotFoundException e)
  {
    ...
  }
  catch (IOException e)
  {
    ...
  }
}
```



Catching Exception

- Catching `Exception` catches all derived types of `Exception`
  - base class for all exception types

catch any exception →

```
void Process()
{
  try
  {
    ...
  }
  catch (Exception e)
  {
    ...
  }
  ...
}
```

General catch clause

- **`catch` clause without specification catches all exceptions**
  - called *general catch clause*

catch any exception →

```
void Process()
{
  try
  {
    ...
  }
  catch
  {
    ...
  }
  ...
}
```


Throw without argument

- **`throw` with no argument re-throws current exception**
  - useful when combined with general catch

catch any exception →
do local cleanup →
propagate exception →

```
void Process()
{
  try
  {
    ...
  }
  catch
  {
    ...

    throw;
  }
  ...
}
```


Resource release

- **Difficult to release resources in presence of exceptions**
  - exception may cause cleanup code to be skipped

may throw exception →
skipped when exception thrown →

```
void Process()
{
  FileStream data = new FileStream("File.dat", FileMode.Create);

  data.WriteByte(0x10);

  ...

  data.Close();
}
```


Finally

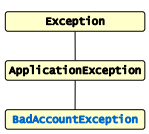
- **`try` can have optional `finally` block**
  - always executed when control leaves associated try block
  - catch block(s) allowed but not required
  - useful for cleaning up resources

always executed →

```
void Process()
{
  FileStream data = null;

  try
  {
    data = new FileStream("File.dat", FileMode.Create);
    data.WriteByte(0x10);
    ...
  }
  finally
  {
    data.Close();
  }
}
```


Custom exception

- **Can create custom exception types**
  - should derive from **`ApplicationException`**
  - should choose name ending in **`Exception`**



Implementing custom exception

- **Custom exception typically:**
  - stores additional data describing error
  - provides constructor that takes string error message
  - chains to base constructor to store string

custom data →
base constructor →

```
class BadAccountException : ApplicationException
{
  public int id;

  public BadAccountException(string msg, int id)
    :base(msg)
  {
    this.id = id;
  }
}
```

Throwing custom exception

- **Can create and throw object of custom exception type**
  - just like system exceptions

check for valid account number →
create and throw →

```
class Bank
{
  public void Withdraw(int id, double amount)
  {
    if (!accounts.Contains(id))
    {
      throw new BadAccountException("Invalid account", id);
    }
    ...
  }
  ...
}
```


Catching custom exception

- **Custom exceptions caught by type**
  - just like system exceptions

catch custom exception →
custom data from `Exception` →

```
void Save(Bank b)
{
  try
  {
    b.Withdraw(1234, 1000);
  }
  catch (BadAccountException e)
  {
    int    i = e.id;
    string s = e.Message;
  }
}
```


Benefits

- **Benefits of exception handling**
  - decouples detection from handling
  - separates handler code
  - facilitates grouping and differentiation of conditions