OO Programming Languages Syntax
Exceptions Handling

Elena Punktaya, op205@cam.ac.uk
Abstract class defines shared implementation

Methods declared Abstract MUST be implemented by subclasses

Interface can only define method signatures (names, parameters, output) and no implementation

Classes can Extend other classes and Implement interfaces

By default, all class methods can be overridden (extended/implemented) in the subclasses

```java
//abstract class declaration
public abstract class Account
{
    private int mAccountNumber;
    //constructor
    public Account(int accountNumber)
    {
        mAccountNumber = accountNumber;
    }
    // abstract method declaration
    public abstract void credit(Amount amount);
}

//interface declaration
public interface IVerifiable
{
    //declaring the interface method
    public boolean isVerified();
}

//PayPal derives from Account and realises IVerifiable
public class PayPalAccount extends Account implements IVerifiable
{
    //constructor, calls the superclass
    public PayPalAccount(int accountNumber)
    {
        super(accountNumber);
    }
    //implementation of the abstract method
    public void credit(Amount amount)
    {
        //send money to PayPal
    }
    //implementation of the interface method
    public boolean isVerified()
    {
        //do check and return result
    }
}
```
**OO in C#**

- Very similar to Java syntax and concepts, but also some differences

- Any non-abstract class methods have to be explicitly declared *virtual* so can be overridden (extended/implemented) in the subclasses

- C# vs Java comparison:

```csharp
//abstract class declaration
public abstract class Account
{
    private int mAccountNumber;
    //constructor
    public Account(int accountNumber)
    {
        mAccountNumber = accountNumber;
    }
    // abstract method declaration
    public abstract void credit(Amount amount);
}

//interface declaration
public interface IVerifiable
{
    //declaring the interface method
    public bool isVerified();
}

//PayPal derives from Account and realises IVerifiable
public class PayPalAccount extends :Account, implements IVerifiable
{
    //constructor, calls the superclass
    public PayPalAccount(int accountNumber) : base(accountNumber)
    {
        super(accountNumber);
    }
    //implementation of the abstract method
    public void credit(Amount amount)
    {
        //send money to PayPal
    }
    //implementation of the interface method
    public bool isVerified()
    {
        //do check and return result
    }
}
```
• Methods need to be declared \textit{virtual} to be extended (as in C#)

• \textit{Pure virtual methods} (ending declarations with “=0”) are equivalent to abstract methods in Java

• No dedicated concept of \textit{Interfaces}, but same effect is achieved by defining a class that contains \textit{pure virtual methods} only

• C++ and Java differences

```c++
//class declaration
class Account {
    //declaring all publicly accessible methods/attributes
    public:
    //constructor
    Account(int accountNumber) {
        mAccountNumber = accountNumber;
    }
    // abstract method declaration
    virtual void credit(Amount amount) = 0;

private:
    int mAccountNumber;
};

//interface (equivalent) declaration
class IVerifiable {
    //pure virtual (abstract) method declaration
    public:
    virtual bool isVerified() = 0;
};

//PayPal derives from Account and IVerifiable
public class PayPalAccount : public Account, public IVerifiable {
    public:
    //constructor, calls the superclass
    PayPalAccount(int accountNumber):Account(accountNumber) {
    }
    //declaring the implementation
    virtual void credit(Amount amount);
    //declaring the implementation
    virtual bool isVerified();
};
```
Error Handling

• All software encounters error conditions during operations

• Good software will manage error situations gracefully and robustly

• Error handling has to be implemented in the code

• A standard option from procedural languages – Error Codes

• Main idea:
  - use a code to indicate some specific error
  - make the function to return a code
  - check the return code if it is OK or error

```java
// add error reporting to code that can fail
public int doSomethingRisky()
{
    <..>
    if (problemNumber1Happened)
        return 1;
    else if (problemNumber2Happened)
        return 2;
    else
        return 0; //all good
}

//using this method, need to build in checks
int result = riskyAction.doSomethingRisky();
if (result == 0)
{
    //All good, can proceed
}
else if(result == 1)
{
    //handle Problem1
}
else if(result == 2)
{
    //handle Problem2
}
```
Exceptions

• Errors such as the above represent exceptions to the normal program flow.

• Handling exceptions via return codes has a number of disadvantages:
  - Extra code needs to be inserted in each function to pass the errors back.
  - If one function fails to check for errors and pass them back, the errors will not get handled.
  - The extra error checking obscures the main function of the code, making it difficult to understand.
  - Error recovery code becomes intertwined with the normal operation code.
  - Functions cannot use return values for normal purposes.

• There is another way...

• Exceptions!

```java
// add error reporting to code that can fail
public void doSomethingRisky() {
    <...>
    if (problemNumber1Happened)
        // throw forces us to exit the current method and returns an exception object
        throw problemNumber1Exception;
    else if (problemNumber2Happened)
        throw problemNumber2Exception;
}

// using this method, wrap it in try/catch block
// to catch returned exceptions
try {
    riskyAction.doSomethingRisky();
} catch (ProblemNumber1Exception error) {
    // handle Problem1
} catch (ProblemNumber2Exception error) {
    // handle Problem2
}
```
Exceptions

• C++ example

• Using exceptions, the code is easier to follow because the error handling parts are clearly separated from the regular program flow

• An exception handler can throw the exception again allowing some errors to be trapped and repaired and others to be propagated, e.g. from `TE_GetPrice()` to `MyTrader()` to the main method, where it is handled

• Exceptions should REALLY be exceptional and not be a part of normal program flow

```cpp
//C++ class to define Exceptions
class TradingErr {
  TradingErr (ErrType ee, Time tt) {e=ee; t=tt;}
    ErrType e;
    Time t;
};

//main program method
int main() {
  try {
   ..<>
    //MyTrader() can re-throw an exception
    MyTrader();
   ..<>
    //Exception handling
  } catch (TradingError x) {
    ReportError(x.e, x.t);
  }
}

//------------------------------------------------
void MyTrader() {
 ..<>
  //code that can re-throw an exception
  float price = TE_GetPrice(day);
 ..<>
}

//------------------------------------------------
float TE_GetPrice(int day) {
 ..<>
  //code that can throw an exception
  if (!Valid(day))
    throw TradingErr(BAD_DAY,TimeNow());
 ..<>
}
No Code Wars! – Tools for the Job

• Debating comparative advantages and disadvantages of programming languages makes a good (if often heated) conversation, but in reality the choice of the language is often dictated by the application!

• For example, in mobile application development:
  - Android, Blackberry, J2ME: Java
  - iPhone: Objective-C
  - Windows Phone: C#
  - Symbian (RIP): C++

• Being proficient in a range of languages helps

Source: http://www.rackspace.com/cloud/blog/2011/05/17/infographic-evolution-of-computer-languages/