EMBEDDED SYSTEMS PROGRAMMING 2016-17

OO Basics

CLASS, METHOD, OBJECT...

- **Class**: abstract description of a "concept"
- **Object**: concrete realization of a "concept".
  An object is an **instance** of a class

Members

- **Method**: piece of executable code
- **Field**: piece of memory containing data.
  Fields store the results of the computation

CLASSES: DECLARATION VS. IMPLEMENTATION

- **Java**: declaration always coincides with implementation

- **C++**: declaration can be separate from implementation

EXPORTING DECLARATIONS

- Header files
  - Java: no, declarations extracted automatically from implementations
  - C++: yes

- Declarations can be read by many source files
  - (Java: no header files)
  - C++: "`#include`" directive

ACCESS MODIFIERS

In both Java and C++, methods and fields can be

- **`public`**
- **`private`**: accessible only by elements of the same class
- **`protected`**: accessible only by elements in its class, and classes in the same package (Java) or friends of the class (C++)

ACCESS MODIFIERS: DEFAULT

- **Java**: members are visible only within their own package ("package private")

- **C++**: members are public

CONSTRUCTOR AND DESTRUCTOR (1/2)

- **Constructor**: special method called (often automatically) at the instantiation of an object. It may accept parameters to initialize fields

- **Destructor**: special method called (often automatically) when an object is destroyed

- If present, constructors/destructors are invoked automatically. Multiple constructors can be defined with different parameters

CONSTRUCTOR AND DESTRUCTOR (2/2)

- **Java**: the constructor must be named as the class. The destructor must be called **`finalize()`**

- **C++**: the constructor must have the same name as the class. The destructor has the same name as the class, but with a tilde (“**~**”) in front of it

THE POINT CLASS: JAVA

```
public class Point
{
    private double x;
    private double y;

    // Default constructor
    public Point()
    {
        x = 0.0;
        y = 0.0;
    }

    // Standard constructor
    public Point(double cx, double cy)
    {
        x = cx;
        y = cy;
    }

    // Accessor methods

    // Methods to set the coordinates to new values
    public void SetX(double cx) { x=cx; }
    public void SetY(double cy) { y=cy; }

    // Returns the distance from the origin
    public double Distance()
    {
        return java.lang.Math.sqrt(x*x+y*y);
    }
}
```

THE POINT CLASS: C++ (1/2)

```
#include <cmath>        // new-style C++ header

class Point
{
private:
    double x;
    double y;

public:
    // Default constructor
    Point()
    {
        x = 0.0;
        y = 0.0;
    }

    // Standard constructor
    Point(double cx, double cy)
    {
        x = cx;
        y = cy;
    }

    // Accessor methods

    // Methods to set the coordinates to new values
    void SetX(double cx) { x=cx; }
    void SetY(double cy) { y=cy; }

    // Method that returns the distance from the origin
    double Distance()
    {
        return sqrt(x*x+y*y);
    }
};
```

THE POINT CLASS: C++ (2/2)

- Method declaration distinct from method definition

```
#include <cmath>

class Point
{
private:
    double x;
    double y;

public:
    Point();
    Point(double cx, double cy);
    void SetX(double cx);
    void SetY(double cy);
    double Distance();
};


// Default constructor
Point::Point()
{
    x = 0.0;
    y = 0.0;
}

...
```

ACCESSING VARIABLES AND METHODS (1/2)

- **Java**: the following example shows how to
  1. access a variable
  2. call a method
  3. call a constructor from another

all within the same class

```
public Point()    // Default constructor
{
     // Invoke the standard constructor
   this(0.0, 0.0);
}

public Point(double cx, double cy) // Standard constructor
{
     x = cx;      // Access to a variable
     SetY(cy);    // Call to a method defined in the class
}
```

ACCESSING VARIABLES AND METHODS (2/2)

- **C++**: the following example shows how to
  1. access a variable
  2. call a method

within the same class

- Calling a constructor from another: no way

```
Point(double cx, double cy)
{
    x = cx;      // Access to a variable
    SetY(cy);    // Call to a method defined in the class
}
```

ALLOCATING OBJECTS (1/2)

- **Instantiation** = creation of an object from a class (i.e., an instance of the class)

- **Java**: use the **`new`** keyword. `new` returns a reference (not a pointer!) to the newly allocated object

```
// Step 1: definition of a reference variable
// for the appropriate class
Point ImaginaryUnit;

// Step 2: creation of the object (instantiation)
ImaginaryUnit = new Point(0.0, 1.0);
```

ALLOCATING OBJECTS (2/2)

- **Instantiation** = creation of an object from a class (i.e., an instance of the class)

- **C++**: simply define the object as if it were a variable. As an alternative, the **`new`** keyword can be used to dynamically allocate the object on the heap

```
// Solution 1: just define the object
Point RealUnit(1.0, 0.0);

// Solution 2: define a pointer, then allocate an object with "new"
Point * ImaginaryUnit;
ImaginaryUnit = new Point(0.0, 1.0);
```

INVOKING OBJECT METHODS

- Java:

```
ImaginaryUnit.SetX(0.0);
```

- C++:

```
RealUnit.SetX(0.0);           // For objects

ImaginaryUnit->SetX(0.0);     // For pointers
```

INHERITANCE

- **Inheritance**: creation of new classes
  that extend the behavior of previously-defined classes
  while retaining the original behavior for some aspects

- Java: **`extends`** keyword
- C++: colon "`:`" operator

INHERITANCE: EXAMPLES (1/3)

- Java:

```
public class Pixel extends Point
{
    public byte color[];     // New: color in RGB format

    public Pixel()           // Redefinition of default constructor
    {
        super();             // Invoking the default constructor of Point

        color = new byte[3];
        color[0] = color[1] = color[2] = 0;
    }

    // Further new fields and methods can be placed here
}
```

- Redefinition of a method is called **overriding**

INHERITANCE: EXAMPLES (2/3)

- Java (wrong code):

```
public class Pixel extends Point
{
    public byte color[];      // New: color in RGB format

    public Pixel()            // Redefinition of default constructor
    {
    x = 0.0;
    y = 0.0;
    color = new byte[3];
    color[0] = color[1] = color[2] = 0;
    }

   // Further new fields and methods can be placed here
}
```

- Does not work because `x` and `y` are private in `point`, hence inaccessible to subclasses.
  It must not work, otherwise it would break **encapsulation**

ENCAPSULATION

- **Encapsulation**: the internal status of a class/object is kept hidden to the maximum possible extent. When necessary, portion of the status can only be accessed via approved methods

- **Encapsulation increases robustness**
  Hiding the internals of an object keeps it consistent by preventing developers from manipulating it in unexpected ways

- **Encapsulation helps in managing complexity**
  Enforcing a strict discipline for object manipulation limits nasty inter-dependencies between objects

INHERITANCE: EXAMPLES (3/3)

- C++:

```
class Pixel: public Point
{
public:
   unsigned char *color;     // New: color in RGB format

   Pixel()
   {
       color = new unsigned char [3];
       color[0] = color[1] = color[2] = 0;
   }

   // Further new fields and methods can be placed here
};
```

- The base class constructor is called automatically
- Again, trying to access `x` and `y` results in a compile-time error

ON THE USE OF NEW

- In C++ there is no garbage collector: memory allocated with `new()` must be deallocated explicitly! This is mandatory to avoid **memory leaks**

- In C++, memory is released with **`delete`** (in the destructor, for instance)

```
~Pixel()                        // Destructor: memory is deallocated here
{
    delete[] color;
}
```

POLYMORPHISM

- From the Merriam-Webster dictionary:
  "**the quality or state of existing in, or assuming, different forms**"

- In OO languages: an object instantiated from a derived class is polymorphic because it behaves **both** as an object of the subclass **and** as an object of the superclass

THE "STATIC" KEYWORD

- Fields and methods can be associated with either
  - a class (**static** field/method)
  - an object (**instance** field/method)

- If a field/method is marked with the `static` keyword, only one copy of it exists

STATIC FIELDS (1/2)

- Example: Java

```
class Customer
{
    static int MaxCustomerID = 0; // unique to class
    int CustomerID;               // different in each instance

    /* ... */

    public Customer()             // constructor
    {
        ++MaxCustomerID;
        CustomerID = MaxCustomerID;
    }

    /* ... */
}
```

STATIC FIELDS (2/2)

- Example: C++

```
class Customer
{
    static int MaxCustomerID;    // initialize OUTSIDE THE CLASS
    int CustomerID;              // different in each instance

    /* ... */

  public:
    Customer()                   // constructor
    {
        ++MaxCustomerID;
        CustomerID = MaxCustomerID;
    }

    /* ... */
};
```

STATIC METHODS (1/2)

- Example: Java

```
public class MathClass
{
    ...   // The constructor goes here

    // Accessor methods

    // The arctangent of a number can be calculated
    // even if no object of type MathClass has been
    // allocated
    public static double arctan(double x)
    {
       ...
    }

    ...   // Additional methods go here
}
```

STATIC METHODS (2/2)

- Example: C++

```
class MathClass
{
public:

    ...   // The constructor goes here

    // Accessor methods

    static double arctan(double x)
    {
        ...
    }

    ...   // Additional methods go here

};
```

EXCEPTIONS

- An **exception** is an event (usually due to an error condition) that occurs at **run time** and **alters** the normal flow of execution

- Exceptions can be raised by library code or by the programmer itself

- Exceptions **must** be managed!
  Unmanaged exceptions lead to program termination

EXCEPTIONS: JAVA (1/2)

- An exception is an object
- Raise an exception: **`throw`** keyword
- Exceptions thrown by a method must be declared in the method's header

```
class DivideByZeroException extends Exception { }


public class Point
{

    // Divides point coordinates by a given factor
    public void ScaleByAFactor(double f) throws DivideByZeroException
    {
        if(f==0.0) throw new DivideByZeroException();
        else
        {
            x = x / f;
            y = y / f;
        }
    }
}
```

EXCEPTIONS: JAVA (2/2)

- Handle an exception:
  `try...catch()...finally`

```
try                                     // code that could throw an exception
{
    ImaginaryUnit.ScaleByAFactor(sf);
}


catch(DivideByZeroException e)  // code that handles the exception;
{                                       // executed only if an exception happens
    // Do something
    System.err.println("Division by zero!");
}


finally                                 // code finishing up the operation;
{                                       // executed in any case
    file.close();
}
```

- Multiple `catch` blocks can be present

EXCEPTIONS: C++ (1/2)

- An exception is not necessarily an object
- Raise an exception: **`throw`** keyword
- Thrown exceptions cannot be declared

```
public class Point
{
    //...

    // Divides point coordinates by a given factor
    void ScaleByAFactor(double f)
    {
        if(f==0.0) throw 123;    // Throws an integer
        else
        {
            x = x / f;
            y = y / f;
        }
    }
};
```

EXCEPTIONS: C++ (2/2)

- Handle an exception: **`try...catch()`**

```
try                                     // code that could throw an exception
{
    ImaginaryUnit.ScaleByAFactor(sf);
}

catch(int e)                            // code that handles the exception;
{                                       // executed only if an exception happens
    // Do something
    cerr << "Division by zero!";
}
```

- Multiple `catch` blocks can be present.
  `catch(...)` (with the 3 dots) catches all exceptions

- No `finally` available

ASSERTIONS

- An assertion is a statement to **test an assumption** about the program that the programmer thinks must be true at a specific place.
  If the assertion is not true, an error is generated

- The test is performed at **run-time**,
  hence the program is slowed down a tiny bit

- Java: **`assert`** keyword, raises exceptions

- C++: macro to simulate assertions

ASSERTIONS: EXAMPLE

- Java:

```
/* Remove an user from a data structure */
/* ... */
assert (NumberOfUsers >= 0);
```

- C++:

```
#include <cassert>

/* Remove an user from a data structure */
/* ... */
assert (NumberOfUsers >= 0);
```

LAST MODIFIED: MARCH 3, 2017