

## **Swinburne University of Technology**

*School of Science, Computing and Engineering Technologies*

### **LABORATORY COVER SHEET**

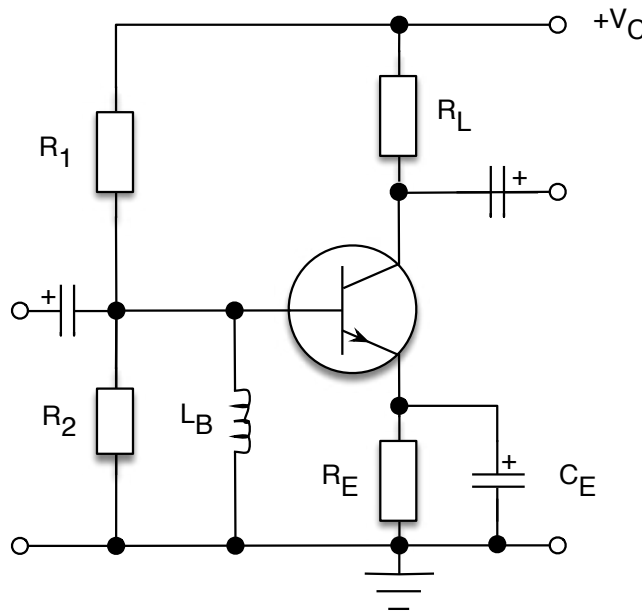
---

<b>Subject Code:</b>	COS30008
<b>Subject Title:</b>	Data Structures and Patterns
<b>Lab number and title:</b>	6, Inheritance and Method Overriding
<b>Lecturer:</b>	Dr. Markus Lumpe

---

***To invent, you need a good imagination and a pile of junk.***

**Thomas A. Edison**



**Figure 1: A Hypothetical Emitter Amplifier Circuit.**

## Lab 6: Inheritance and Method Overriding

Consider Figure 1, which shows a hypothetical stage amplifier circuit. Do not be alarmed, we are not going to analyze this circuit or start doing electronics in COS3008. It just serves as a motivation to experiment with object-oriented inheritance and method overriding, and to demonstrate that code reuse and incremental refinement are the key ingredients in object-oriented programming and data type construction.

The interesting features in the amplifier circuit with respect to object-oriented software design are the capacitor  $C_E$  and the inductor  $L_B$ . In this circuit, both serve as a frequency-dependent resistor to control the amplifier gain and frequency range. The specifics are not important and would rather distract from the task ahead. There is, however, an intriguing aspect: both *the capacitor and the inductor can be used in the place of a resistor* when we need a frequency-dependent resistor in a circuit. (Naturally, this description is very simplified.)

In software terms, this is a familiar scenario. It is called polymorphism: objects of different types can respond uniformly to a given operation. The type of polymorphism we are interested in is called “subtype polymorphism” that arises from object-oriented inheritance and method overriding. To model the required domain concepts (here a passive resistor, a capacitor, and an inductor), we create a corresponding class hierarchy and set up as root class an abstract class, called `ResistorBase`, which both encapsulates the common behavior and defines the variable features through the definition of pure virtual methods. In the solution, we then can create three subclasses that implement the respective variable behavior: a class `PassiveResistor` that models electric resistance of a resistor (frequency independent), a class `Capacitor` that implements *capacitive reactance* of a capacitor (decreases with frequency), and a class `Inductor` that captures the inductive reactance of an inductor (increases with frequency).

## Conditional Compilation

The test driver provided for this tutorial task makes use of conditional compilation via preprocessor directives. This allows you to focus only on the task you are working on.

The test driver (i.e., `main.cpp`) uses `P1`, `P2`, `P3`, `P4`, and `P5` as variables to enable/disable the test associated with a corresponding problem. To enable a test just uncomment the respective `#define` line. For example, to test problem 2 only, enable `#define P2`:

```
// #define P1
#define P2
// #define P3
// #define P4
// #define P5
```

In Visual Studio, the code blocks enclosed in `#ifdef PX ... #endif` are grayed out, if the corresponding test is disabled. The preprocessor definition `#ifdef PX ... #endif` enables conditional compilation. XCode does not use this color coding scheme.

In addition, to facilitate the completion of this tutorial the implementations for the methods `convertBaseValueToUnitValue()` and `convertUnitValueToBaseValue()` defined in class `ResistorBase` are implemented in `ResistorBaseScaling.cpp` which is available on Canvas. Please analyze the implementation. These two methods provide the infrastructure to process, scale, and normalize the specification of resistor, capacitor, and inductor values.

## Problem 1

The abstract class `ResistorBase` is given as

```
#pragma once

#include <iostream>
#include <string>

class ResistorBase
{
private:
    // converts base value and unit value (object remains unchanged)
    // Example: the value of a register 56000.0 in unitless becomes 56.0kOhm
    void convertBaseValueToUnitValue( double& aNormalizedValue,
                                      std::string& aUnit ) const noexcept;

    // converts unit value to base value (updates object)
    // Example: the value of a register 56.0kOhm becomes 56000.0 in unitless form
    void convertUnitValueToBaseValue( const double& aRawValue,
                                      const std::string& aRawUnit );

    // base value of a resistor component
    double fBaseValue;

protected:
    // auxiliary methods (= 0 - pure virtual members)
    virtual bool mustScale( double aValue ) const noexcept = 0;
    virtual const double getMultiplier() const noexcept = 0;

    virtual const std::string getMajorUnit() const noexcept = 0;
    virtual const std::string getMinorUnits() const noexcept = 0;

    // setter for base value (used in method convertUnitValueToBaseValue())
    void setBaseValue( double aBaseValue ) noexcept;

public:
    // constructor with a default value
    ResistorBase( double aBaseValue = 0.0 ) noexcept;

    // required virtual destructor (implementation)
    virtual ~ResistorBase() {}

    // returns base value
    double getBaseValue() const noexcept;

    // returns (frequency-dependent) passive resistance value
    virtual double getReactance( double aFrequency = 0.0 ) const noexcept = 0;

    // returns (frequency-dependent) voltage drop
    double getPotentialAt( double aCurrent,
                          double aFrequency = 0.0 ) const noexcept;

    // returns (frequency-dependent) current flowing through a resistor
    double getCurrentAt( double aVoltage,
                        double aFrequency = 0.0 ) const noexcept;

    // resistor I/O
    friend std::istream& operator>>( std::istream& aIStream,
                                     ResistorBase& aObject );
    friend std::ostream& operator<<( std::ostream& aOStream,
                                     const ResistorBase& aObject );
};
```

The abstract class `ResistorBase` is quite complex. It defines private, protected, and public features. The private features address the required data conversion (scaling) of the base value and unit, so that we can effectively specify resistor, capacitor, and inductor values. Somebody has already implemented those features. You need to study them in order to use them in your solution. A great help is the C++ online reference: <http://www.cplusplus.com/reference>.

`ResistorBaseScaling.cpp`:

```
#include "ResistorBase.h"
#include <stdexcept>

using namespace std;

void ResistorBase::convertBaseValueToUnitValue( double& aNormalizedValue,
                                                std::string& aUnit ) const noexcept
{
    aNormalizedValue = getBaseValue();
    std::string lPrefixes = getMinorUnits();

    for ( size_t i = 0; i < lPrefixes.size(); i++ )
    {
        // stop scaling at maximum unit
        if ( mustScale( aNormalizedValue ) && ( i < lPrefixes.size() - 1 ) )
        {
            aNormalizedValue *= getMultiplier();
        }
        else
        {
            if ( i > 0 )
            {
                aUnit += lPrefixes[i];
            }
            aUnit += getMajorUnit();
            break;
        }
    }
}

void ResistorBase::convertUnitValueToBaseValue( const double& aRawValue,
                                                const std::string& aRawUnit )
{
    std::string lMajorUnit = getMajorUnit();
    std::string lMinorUnits = getMinorUnits();

    // error handling
    // test basic features (raw unit too long and not containing major unit)
    if ( (aRawUnit.size() > lMajorUnit.size() + 1) ||
        (aRawUnit.find( getMajorUnit() ) == std::string::npos) )
    {
        throw std::invalid_argument( "Invalid unit specification" );
    }

    // test scale features, aRawUnit[0] must be contained in minor units
    if ( (aRawUnit.size() == lMajorUnit.size() + 1) &&
        lMinorUnits.find( aRawUnit[0] ) == std::string::npos )
    {
        throw std::invalid_argument( "Invalid unit scale specification" );
    }

    // adjust base value
    double lMultiplier = 1.0;

    size_t i = lMinorUnits.find( aRawUnit[0] );
    double lRawValue = aRawValue;

    // scale raw value first
    for ( ; i > 0 ; i-- )
    {
        if ( mustScale( lRawValue ) )
```

```

        lRawValue /= getMultiplier();
    else
        break;
}

// adjust multiplier
for ( ; i > 0; i-- )
{
    lMultiplier *= 1.0/getMultiplier();
}

setBaseValue( lRawValue * lMultiplier );
}

```

These two private member functions perform data pre- and post-processing for the I/O operators. The method `convertUnitValueToBaseValue()` allows input data to be properly converted to an internal base value. For example, the input "56kOhm" yields 56000.0 as base value. Similarly, the method `convertBaseValueToUnitValue()` converts a base value to an external representation for the output. For example, the base value 0.00047 yields "470uF" as output for a capacitor. The I/O operators have to locally declare and use two variables of type `double` and `string`, respectively, in order to interact with the methods `convertUnitValueToBaseValue()` and `convertBaseValueToUnitValue()`. In addition, the behavior of these methods is controlled by protected methods `mustScale()`, `getMultiplier()`, `getMajorUnit()`, and `getMinorUnits()`, which yield results specific to the underlying domain concept (i.e., resistor or capacitor). Please note that both, `convertBaseValueToUnitValue()` and `convertUnitValueToBaseValue()` use reference parameters. We use *call-by-reference*. For `flatten`, we use *in* parameters, whereas `normalize` has *out* parameters.

To implement abstract class `ResistorBase`, recall Ohm's law:  $R = \frac{V}{I}$ , the potential difference (voltage) across an ideal conductor is proportional to the current flowing through it. We can rewrite this formula to obtain the voltage drop at a resistor for a given current:  $V = R * I$ , and the current flowing through the resistor for a given voltage:  $I = \frac{V}{R}$ . The value  $R$  can be obtained by calling method `getReactance()` that takes a frequency as argument.

You cannot test the abstract class `ResistorBase` at this stage. We need a class that implement all the pure virtual members first. We start with class `PassiveResistor`.

The class `PassiveResistor` is given as

```
#pragma once

#include "ResistorBase.h"

class PassiveResistor : public ResistorBase
{
protected:
    // auxiliary methods

    // aValue >= 1000.0
    bool mustScale( double aValue ) const noexcept override;
    // 1.0/1000.0
    const double getMultiplier() const noexcept override;
    // "Ohm"
    const std::string getMajorUnit() const noexcept override;
    // "OkM", the first letter means "no minor unit"
    const std::string getMinorUnits() const noexcept override;

public:

    // constructor with a default value
    PassiveResistor( double aBaseValue = 0.0 ) noexcept;

    // returns (frequency-independent) passive resistance value
    double getReactance( double aFrequency = 0.0 ) const noexcept override;
};
```

The class `PassiveResistor` does not define any new instance variables. It just overrides the inherited pure virtual methods to adjust the base variable reporting and conversion. (Remember, public inheritance means that class `PassiveResistor` inherits all public methods of class `ResistorBase`.) Please note that we make no claim that the class `PassiveResistor` faithfully implements the behavior of a passive resistor (e.g., we do not model the conversion from electrical to thermal energy). We just model electric resistance in this tutorial.

You may notice that we do not need to define explicit I/O operators for class `PassiveResistor`. We can reuse the existing ones from class `ResistorBase`. It works due to *dynamic method invocation*. In the operators, we call the service functions `flatten` and `normalize`, which in turn rely on the class-specific variants of `mustScale()`, `getMultiplier()`, `getMajorUnit()`, and `getMinorUnits()`. Hence, when use input a `PassiveResistor` object, the input operator will format it according to the principles of a passive resistor. That is, it will use "Ohm" as unit and scale by 0.001. For example, if the value of the resistor is 5600.00Ohm, then the output operator needs to produce "5.6kOhm".

Defining the class `PassiveResistor` in this way allows us to write

```
void runP1()
{
    PassiveResistor lR;

    cout << "Enter resistor value: ";
    cin >> lR;
    cout << "Passive resistor value: " << lR << endl;
    cout << "Current at 9.0V: " << lR.getCurrentAt( 9.0 ) << "A" << endl;
    cout << "Voltage drop at 200mA: " << lR.getPotentialAt( 0.2 ) << "V" << endl;
}
```

which should produce the following output

```
Enter resistor value: 560hm  
Passive resistor value: 560hm  
Current at 9.0V: 0.160714A  
Voltage drop at 200mA: 11.2V
```



## Problem 2

Using the abstract class `ResistorBase`, we can construct a class `Capacitor` to model "capacitive reactance." The capacitive reactance is the frequency-dependent resistance value of a capacitor given by the formula  $X_C = \frac{1}{2\pi fC}$ . Both, the value of the capacitor and the signal frequency significantly contribute to the capacitive reactance. In general, the capacitive reactance is inversely related to the signal frequency.

To model a capacitor, we define class `Capacitor` as a public subclass of class `ResistorBase`:

```
#pragma once

#include "ResistorBase.h"

class Capacitor : public ResistorBase
{
protected:
    // auxiliary methods

    // aValue < 1.0
    bool mustScale( double aValue ) const noexcept override;
    // 1000.0
    const double getMultiplier() const noexcept override;
    // "F"
    const std::string getMajorUnit() const noexcept override;
    // minor units: "Fmupn", the first letter means "no major unit"
    const std::string getMinorUnits() const noexcept override;

public:

    // constructor with a default value
    Capacitor( double aBaseValue = 0.0 ) noexcept;

    // returns (frequency-dependent) passive resistance value
    double getReactance( double aFrequency = 0.0 ) const noexcept override;
};
```

The class `Capacitor` does not define any new instance variables. It just overrides the inherited pure virtual methods to adjust the base variable reporting and conversion. (Remember, public inheritance means that class `Capacitor` inherits all public methods of class `ResistorBase`.) Please note that we make no claim that the class `Capacitor` faithfully implements the behavior of a capacitor. We just model capacitive reactance in this tutorial.

The new implementation for the inherited virtual method `getReactance()` has to model the capacitive reactance, which requires the value of  $\pi$ . You can bring the value  $\pi$  of in two ways into your compilation unit. First, we can use the following expression for this purpose

```
double PI = 4.0 * atan(1.0);
```

The function `atan()` is defined in `cmath`.

Second, add before any `#include` statement the line

```
#define _USE_MATH_DEFINES
```

and include `cmath`. This will allow you to use the constant `M_PI` of type `double` in your program.

You may notice that we do not need to define explicit I/O operators for class `Capacitor`. We can reuse the existing ones from class `ResistorBase`. It works due to *dynamic method invocation*. In the operators, we call the service functions `convertUnitValueToBaseValue()`

and `convertBaseValueToUnitValue()`, which in turn rely on the class-specific variants of `mustScale()`, `getMultiplier()`, `getMajorUnit()`, and `getMinorUnits()`. Hence, when use input a `Capacitor` object, the input operator will format it according to the principles of a capacitor. That is, it will use "F" (Farad) as unit and scale by 1000.0. For example, if the value of the capacitor is 0.0000047F, then the output operator needs to produce "4.7uF" (we use the letter 'u' instead of the Greek symbol  $\mu$ ).

Defining the class `Capacitor` in this way allows us to write

```
void runP2()
{
    Capacitor lC;

    cout << "Enter capacitor value: ";
    cin >> lC;
    cout << "Capacitor value: " << lC << endl;

    // create a temporary passive resistor object to properly format value
    cout << "XC at 60Hz: " << PassiveResistor( lC.getReactance( 60.0 ) ) << endl;
    cout << "Current at 9V and 60Hz: " << lC.getCurrentAt( 9.0, 60.0 ) << "A"
        << endl;
    cout << "Voltage drop at 2mA and 60Hz: " << lC.getPotentialAt( 0.002, 60.0 )
        << "V" << endl;
}
```

which should produce the following output

```
Enter capacitor value: 470uF
Capacitor value: 470uF
XC at 60Hz: 5.64379 Ohm
Current at 9V and 60Hz: 1.59467A
Voltage drop at 2mA and 60Hz: 0.0112876V
```

Note, we use `PassiveResistor( lC.getValue( 60.0 ) )` to convert the capacitive reactance into a plain resistor value in order to obtain the desired formatted output.

### Problem 3

Using the classes `PassiveResistor` and `Capacitor`, we can construct a time series to determine the signal frequency at which a  $470\mu\text{F}$  (0.00047) capacitor reaches a capacitive reactance of less than 10 $\Omega$ . A do-while loop that increments the signal frequency by 50Hz in each step works best for this purpose.

```
void runP3()
{
    Capacitor lC( 0.00047 );

    cout << "Capacitor value: " << lC << endl;

    double lXC = 0.0;
    double lFrequency = 50.0;

    do
    {
        lXC = lC.getReactance( lFrequency );
        cout << "XC at " << setw( 5 ) << lFrequency << "Hz:\t"
             << PassiveResistor( lXC ) << endl;
        lFrequency += 50.0;
    } while (lXC > 1.0);
}
```

We should reach a capacitive reactance of less than 10 $\Omega$  in seven steps:

XC at	50Hz:	6.772550 $\Omega$
XC at	100Hz:	3.386280 $\Omega$
XC at	150Hz:	2.257520 $\Omega$
XC at	200Hz:	1.693140 $\Omega$
XC at	250Hz:	1.354510 $\Omega$
XC at	300Hz:	1.128760 $\Omega$
XC at	350Hz:	0.9675070 $\Omega$

## Problem 4

Using the abstract class `ResistorBase`, we can construct a class `Inductor` to model "inductive reactance." The inductive reactance is the frequency-dependent resistance value of an inductor given by the formula  $X_C = 2\pi fL$ . Both, the value of the inductor and the signal frequency significantly contribute to the inductive reactance. In general, the inductive reactance changes proportionally with the signal frequency.

To model a capacitor, we define class `Inductor` as a public subclass of class `ResistorBase`:

```
#pragma once

#include "ResistorBase.h"

class Inductor : public ResistorBase
{
protected:
    // auxiliary methods

    // aValue < 1.0
    bool mustScale( double aValue ) const noexcept override;
    // 1000.0
    const double getMultiplier() const noexcept override;
    // "H"
    const std::string getMajorUnit() const noexcept override;
    // minor units: "Hmupn", the first letter means "no major unit"
    const std::string getMinorUnits() const noexcept override;

public:

    // constructor with a default value
    Inductor( double aBaseValue = 0.0 ) noexcept;

    // returns (frequency-dependent) passive resistance value
    double getReactance( double aFrequency = 0.0 ) const noexcept override;
};
```

The class `Inductor` does not define any new instance variables. It just overrides the inherited pure virtual methods to adjust the base variable reporting and conversion. (Remember, public inheritance means that class `Inductor` inherits all public methods of class `ResistorBase`.) Please note that we make no claim that the class `Inductor` faithfully implements the behavior of an inductor. We just model inductive reactance in this tutorial.

The new implementation for the inherited virtual method `getReactance()` has to model the inductive reactance, which requires the value of  $\pi$ . You can bring the value  $\pi$  of in two ways into your compilation unit. First, we can use the following expression for this purpose

```
double PI = 4.0 * atan(1.0);
```

The function `atan()` is defined in `cmath`.

Second, add before any `#include` statement the line

```
#define _USE_MATH_DEFINES
```

and include `cmath`. This will allow you to use the constant `M_PI` of type `double` in your program.

You may notice that we do not need to define explicit I/O operators for class `Inductor`. We can reuse the existing ones from class `ResistorBase`. It works due to *dynamic method invocation*. In the operators, we call the service functions `convertUnitValueToBaseValue()` and `convertBaseValueToUnitValue()`, which in turn rely on the class-specific variants of

`mustScale()`, `getMultiplier()`, `getMajorUnit()`, and `getMinorUnits()`. Hence, when use input an `Inductor` object, the input operator will format it according to the principles of an inductor. That is, it will use "H" (Henry) as unit and scale by 1000.0. For example, if the value of the inductor is `0.2H`, then the output operator needs to produce "200mH".

Defining the class `Inductor` in this way allows us to write

```
void runP4()
{
    Inductor lL;

    cout << "Enter inductor value: ";
    cin >> lL;
    cout << "Inductor value: " << lL << endl;

    // create a temporary passive resistor object to properly format value
    cout << "XC at 10kHz: " << PassiveResistor( lL.getReactance( 10000.0 ) )
        << endl;
    cout << "Current at 9V and 10kHz: " << lL.getCurrentAt( 9.0, 10000.0 )
        << "A" << endl;
    cout << "Voltage drop at 2mA and 10kHz: "
        << lL.getPotentialAt( 0.002, 10000.0 ) << "V" << endl;
}
```

which should produce the following output

```
Enter inductor value: 0.2H
Inductor value: 200mH
XC at 10kHz: 12.5664kOhm
Current at 9V and 10kHz: 0.000716197A
Voltage drop at 2mA and 10kHz: 25.1327V
```

Note, we use `PassiveResistor( lC.getValue( 10000.0 ) )` to convert the capacitive reactance into a plain resistor value in order to obtain the desired formatted output.

## Problem 5

Using the classes `PassiveResistor` and `Inductor`, we can construct a time series to determine the signal frequency at which a 520mH (0.52) inductor reaches a capacitive reactance of more than 1kOhm. A do-while loop that increments the signal frequency by 50Hz in each step works best for this purpose.

```
void runP5()
{
    Inductor lL( 0.52 );

    cout << "Inductor value: " << lL << endl;

    double lXC = 0.0;
    double lFrequency = 50.0;

    do
    {
        lXC = lL.getReactance( lFrequency );
        cout << "XC at " << setw( 5 ) << lFrequency << "Hz:\t"
             << PassiveResistor( lXC ) << endl;
        lFrequency += 50.0;
    } while (lXC < 1000.0);
}
```

We should reach an inductive reactance of more than 1kOhm in seven steps:

XC at	50Hz:	163.3630hm
XC at	100Hz:	326.7260hm
XC at	150Hz:	490.0880hm
XC at	200Hz:	653.4510hm
XC at	250Hz:	816.8140hm
XC at	300Hz:	980.1770hm
XC at	350Hz:	1.14354kOhm