# Notes on the Complex library

## Use

Download the jar. To use it in a project, include the jar in the project classpath. Use appropriate imports, such as

```
import com.waltermilner.complex.Complex;
```

That's it

## Complex variables

The class Complex models those - simply wrapping double values for real and imaginary parts.
One constructor simply has two double parameters for real and imaginary parts. There is a toString method so we can sop a complex value:

```
     Complex c1=new Complex(1,0.12345);
     System.out.println(c1); // 1 + 0.123i
```

The toString formats with 3 decimal places.

More examples:

```
        c1=new Complex(0,1);
        System.out.println(c1); //0 + 1i
        c1=new Complex(1,0);
        System.out.println(c1); // 1 + 0i
        c1=new Complex(1,-5.2);
        System.out.println(c1); // 1 - 5.2i
        c1=new Complex();
        System.out.println(c1); //0 + 0i
```

There is also a copy constructor:

```
        c1=new Complex(-0.1, Math.PI);
```

```
        Complex c2 = new Complex( c1); // copy constructor
        System.out.println(c2); // -0.1 + 3.142i
```

There is a static method which produces a new number in polar form, supplying a modulus and argument:

```
        c2=Complex.polar(1, Math.PI);
        System.out.println(c2); // -1 + 0i
```

This can't be a constructor, because we already have one with two double arguments.

There are methods to form the complex conjugate, modulus, real and imaginary parts:

```
        c1=new Complex(1,-1);
        System.out.println(c1.conjugate()); // conjugate = 1 + 1i
        System.out.println(c1.mod()); // mod = root 2 = 1.4142135623730951
        System.out.println(c1.re()); // real part 1.0
        System.out.println(c1.im()); // imaginary part -1.0
```

Square root and integral power:

```
        // square root and power
        System.out.println(c1.pow(2).sqrt()); // back to 1,-1
```

Find the argument:

```
        System.out.println(new Complex(0,1).arg()); // arg = pi/2 = 1.5707963267948966
```

and arithmetic:

```
        // arithmetic
        c1=new Complex(1,1);
        c2=new Complex(-1,1);
        System.out.println(c1.add(c2)); // 0 + 2i
        System.out.println(c1.sub(c2)); // 2 + 0i
        System.out.println(c1.mul(c2)); // -2 + 0i
        System.out.println(c1.div(c2).mul(c2)); // c1/c2*c2 = c1 = 1 + 1i
        System.out.println(c1.div(2)); // division by scalar = 0.5 + 0.5i
        System.out.println(c1.pow(1)); // exponentiation to integer power = 1 + 1i
        System.out.println(c1.pow(2)); // 0 + 2i
        System.out.println(c1.pow(2).div(c1)); // c1^2 / c1 = c1 = 1 + 1i
```

Note that something like c1.add(c2) does not change c1. It returns a new complex value, equal to c1+c2. The other methods work in a similar way.

## Complex functions

There is a base class named ComplexFunction, and you need to extend this to define your own functions. ComplexFunction is abstract.

For example, to set up $f(z)=z^2$:

```
public class F1 extends ComplexFunction {

    public Complex eval(Complex z) {
        return z.mul(z);
    }

    public Complex eval(double x, double y) {
        return new Complex(x * x - y * y, 2 * x * y);
    }

    public String toString() {
        return ("z*z");
    }
}
```

So you need to override eval(Complex z),  eval(double x, double y) and toString().  Basically you just need to say how to calculate your function.

One option was to use Java 8's ability to define function objects. This was not used, so that the library would work with older JRE versions.

Once you have a function, you can evaluate it:

```
        ComplexFunction f1=new F1(); // z^2
        System.out.println(f1.eval(new Complex(1,1))); // 0 + 2i
        System.out.println(f1.eval(1,1)); //0 + 2i
```

You could attempt to evaluate it at a point where it is not defined - which produces an exception.

You can find df/dz at some z value:

```
        System.out.println(f1.deriv(new Complex(0,1))); // 2z = 0+2i
```

ComplexFunction has a method named cr, which uses a numerical approximation to check the Cauchy-Riemann equations to check the derivative exists. So derive starts off

```
        if (!cr(z)) {
```

```
            throw new ComplexException(this + " is not differentiable at " + z);
        }
```

This can be checked directly:

```
        System.out.println(f1.cr(new Complex(0,0))); // true
        System.out.println(f1.cr(new Complex(0,2))); // true
        System.out.println(f1.cr(new Complex(2,0))); // true
```

since $z^2$ is differentiable everywhere.

This is the code of cr:

```
    public boolean cr(Complex z0) {
        double dux = (eval(z0.add(dx)).sub((eval(z0)))).re();
        double dudx = dux; // really its dux/h - but this is the same
        // for the other differentials, which we just compare, so leave them all
        // multiplied by h
        double dvy = (eval(z0.add(dy)).sub((eval(z0)))).im();
        double dvdy = dvy;
        double duy = (eval(z0.add(dy)).sub((eval(z0)))).re();
        double dudy = duy;
        double dvx = (eval(z0.add(dx)).sub((eval(z0)))).im();
        double dvdx = dvx;
        return (close(dudx, dvdy) && close(dudy, -dvdx));
    }
```

so this uses numerical approximations to the derivatives to check whether partial du/dx = dv/dy and du/dy = -dv/dx. This is just an *approximation*, but it shows the idea.

Similarly we can check whether it is continuous:

```
        System.out.println(f1.isContinuous(new Complex(2,3))); // true
```

f(z)=1/z is more interesting:

```
        ComplexFunction f1=new F2(); // 1/z
        System.out.println(f1.eval(new Complex(1,1))); // 0.5 - 0.5i
        System.out.println(f1.eval(1,1)); // 0.5 - 0.5i
        System.out.println(f1.deriv(new Complex(0,1))); // 1+0i
        // System.out.println(f1.cr(new Complex(0,0)));  undefined exception
        System.out.println(f1.cr(new Complex(0,2))); // true
```

```
        System.out.println(f1.cr(new Complex(2,0))); // true
        System.out.println(f1.isContinuous(new Complex(0,0))); // false
        System.out.println(f1.isContinuous(new Complex(1,0))); // true
        System.out.println(f1.isContinuous(new Complex(0,1))); // true
```

## Line integrals

For example:

```
        f1=new F1(); // z*z
        Complex val = f1.lineIntegral(new Complex(0,0), new Complex(2,1));
        System.out.println(val); // 0.667 + 3.667i: should be 0.66667 + 3.666667 i
```

This is working out

$$\int_{z=0+0i}^{z=2+i} z^2\,dz$$

taking the integral over the straight line path.

Or we can construct a path from a sequence of straight lines:

```
        ArrayList<Complex> path = new ArrayList<Complex>();
        path.add(new Complex(0,0));
        path.add(new Complex(2,0));
        path.add(new Complex(2,1));
        path.add(new Complex(0,0)); // closed path
        System.out.println(f1.integralOnPath(path)); // 0.002 + 0.007i : should be 0 + 0i
```

## Integral on circle

circleIntegral calculates an integral on a circle with given centre and radius. For example, if f1 is $z^2$,

```
System.out.println(f1.circleIntegral(new Complex(),1)); // 0 + 0i
```

calculates

$$\int_C z^2\,dz$$

where C is the unit circle centered on the origin. This is zero due to the Cauchy-Gorsat theorem.

On the other hand

```
        f1=new F2(); // 1/z
        System.out.println(f1.circleIntegral(new Complex(),1)); // 0 + 6.283i
```

since due to the Residue Theorem

$$\int_C \frac{1}{z}dz = 2\pi i$$