Physical reality and quantum reality are rather... different. While physical reality; the world we can see and touch, can be easily quantified, measured and examined, quantum reality; the world of fundamental subatomic particles, atoms, photons and energy in it's most basic units, is another story. However, understanding at least the basic foundations of quantum theory is important to understanding the evolving field of quantum computers.

Quantum particles, these atoms and their sub-parts such as electrons, protons, neutrons and even smaller, the quarks and neutrinos that make them up are very difficult to quantify. This is because the very act of measuring a quantum particle can change it's behavior, its orientation and even the probability of where it will end up as it is moving.

There was an experiment called the double slit experiment. Researchers set up a filter with two slits in it and shot electrons at it. They observed the resulting pattern on the other side, expecting to see the electrons register exactly opposite of the slits. What they saw however was an alternating pattern of hits and misses as if the electrons were not separate particles, but rather like they were the ripples in a pond that a rock was tossed into. When they decided to measure the path the electrons took to figure out how they created this result, they were greeted with what they were originally expecting: two lines of electrons. The measurement changed them to act like they were separate physical objects, bound to the same laws of physics as everything else you can see and touch.

This is an example of a quantum effect called interference: how a particle is measured can change its outcome.

Every particle, including an electron has a spin: It can either be up or it can be down in either the x, y or z direction (imagine a 3D graph). For a long time scientists knew that electrons can bond together around their parent atom and they were attracted to their opposite spin. However, what researchers didn't know until a couple of years ago was that they can be attracted to each other over vast distances, and that any change in any property of an electron (or any particle) will change the other, even if separated over thousands of miles!

This is what is known as quantum entanglement: when two or more particles affect each other as if they are linked somehow, even over vast distances.

Think of a particle as a bit, where the spin up (1) or spin down (0) can be a 1 or a 0. Taken by itself, this particle can be either, or even both, until its spin is measured. This is what is called a superposition. The probability of either it being a 1 or a 0 is significant.

This is where the concept of a Qubit comes in. Because of the superposition state and the fact that the possibility of a qubit being either a 1 or a 0 is significant, qubits can carry 2^N amount of information where N is the number of qubits used. It is even possible to encode different messages in the same group of qubits, at which point the only real obstacle is decoding the correct message.

What is a qubit? Physically they are just particles, most probably electrons, living in the core of quantum processors that affect their quantum properties.

A quantum computer is an interesting machine:
Unlike traditional computers, it uses very little power (25kW). Most of that power is used by the cooling system. This cooling system goes through a series of layers till it reaches the sealed core which is cooled to as close to zero Kelvin (also known as Absolute Zero; the temperature at which all

molecular movement stops) to slow the atoms inside the quantum processor. This allows them to be controlled and measured in a methodical way, using the concepts of quantum interference and quantum entanglement. These atoms are the Qubits.

A qubit's spin, like the bit values in every computer, can be represented by a 1 or a 0. Unlike normal bits, a qubit can be considered as a superposition of itself where the lines between 1 and 0 become “foggy” and it is considered both and neither at the same time. Here the probability that the qubit is a 1 or a 0 becomes significant and probability coefficients are revealed by the mathematics behind quantum mechanics. I won't touch on that much here.

Just know that mathematics wise, qubits and the operations performed on them are treated as vectors and matricies. A qubit is shown as a vector or a matrix of 2 rows and 1 column, whose members are a 1 and a 0. Quantum Logic Gates change their probabilities through matrix operations and can influence the outcome of quantum measurements, to help encode and decode information contained in a series of qubits. These qubits can be passed to gates singularly or as entangled pairs or entangled groups, although not every gate can take any number of qubits. There are single and double qubit gates as well as gates that can accept any number of qubits. The physical gates in the quantum processor can also be used in both directions, unlike the logic gates in traditional CPUs.

Any programming language can be used to create or model quantum algorithms. Python, R and Scala are all easily capable.

However, Microsoft's Q# language provides a friendly framework that is optimized for quantum algorithms and abstracts much of the math away from the developer so that anyone with the basic understanding of quantum mechanics can use the language.

Q# is a functional language that executes top-down and is, for the most part an immutable language (meaning that values can not be changed once they are assigned).

When you create a Q# program you are given two files. A .qs file for your quantum code and a .cs file for the C# driver code needed to initialize and interact with the quantum simulator included in the Micosoft Quantum SDK.

This can be found here: https://www.microsoft.com/en-us/quantum

Operations form the basis of quantum computation in Q#.
Method signature as follows:

```
namespace YourNamespace {

        open Library1; //for now, Microsoft.Quantum.Primitive
        open Library2; //for now, Microsoft.Quantum.Canon
        ...
        open LibraryN;

        operation Name_of_Op (Para1: Type,  Para2: Type,  ...ParaN : Type ) : ( ReturnType1,
        ReturnType2 … ReturnTypeN)
                {
                        body{
```

```
                    // Enter your quantum code here
                    //This is where operations using quantum gates and qubits is performed
                }
            }
        }
```

Functions are used to perform classic non-quantum computation and their method signature is similar to the above, but is declared using the term function.

If return type of either an operation or a function is void, then the parenthesis after the parameters is left empty, but is still included in the declaration. If there is only one return type the return parenthesis can be omitted but the parameter parenthesis must be included if there is only one parameter because the type and name must be included.

```
//in namespace declaration

function Name_of_Func (Para1: Type,  Para2: Type,  ...ParaN : Type ) : ( ReturnType1,
        ReturnType2 … ReturnTypeN){

            //Computation regarding known measured values is done here. The results from a
            //measured series of qubits can be passed to functions as parameters, however qubits
            //will not be initialized and flipped around here

}
```

A Functor is a factory that defines a new operation or function from an existing one. The standard types of functors are: Adjoint and Controlled
Adjoint operations are applied to the reverse value of a qubit.
Controlled operations are applied using a measured qubit as a control value.

This can be declared at the end of the operation body:

```
    {
        //code above
        Adjoint auto
        Controlled auto
        Adjoint Controlled auto
    }
```

This basically means do this operation automatically to the other possibilities not explicitly specified in the operation, using a controlled qubit automatically sectioned off and assigned.

Like C#, Q# statements end in semicolons;

Also like C# Int, Double, Bool, String are valid types and correspond to the same rules regarding them.

The basic type in Q# that differentiates it from other languages is the Qubit. As stated above it refers to some specific spin value of a physical qubit in a quantum processor and once measured, it will be a 1 or a 0 and the possibilities of it being either value can be used to encode and decode information as well.

Qubits are initialized in a using statement just under the body statement:
using (qubits = Qubit[num_of_qubits])

The type Result is the result type of a quantum measurement. It is either One or Zero
It must be spelled out like that: 1 != One and 0!=Zero

However Q# also includes PauliI, PauliX, PauliY, PauliZ which correspond to operations flipping a qubit around its X,Y,Z axis. These "spin flips" may change the probability of a qubit being a 1 or 0.

This can have important implications in the use of quantum measurements as encryption algorithms, because you can have information hidden through the use of random quantum key generation.

Range expressions, like the ones contained in for loops are declared a little differently:
(start_value .. //*optional_mid_value..* end_value)

Q# includes for(thing in val..end_val)
{//*code*} loops
and if(condition)
{//*code*}
elif
{//*code*} for control flow.

Mutable variables whose values can be changed are declared using the mutable keyword.
Immutable values can be declared using a let statement.

Variables can be assigned to using a lowercase set statement: set varName = value;

Qubits can be assigned to using an uppercase Set statement: Set(initial, qubits[0]), usually not far underneath the using statement.

Qubits can be used temporarily in a block of code using a borrowing statement.

Measurement is done using the M operation: M(qubit);
This is a 1 way operation: Until the qubit is Reset at the end of the operation block, it will have whatever value the measurement gives. Measurements usually include both a value and the probability coefficients that a qubit is either a 1 or a 0.

Qubits must be Reset to Zero before they are de-allocated at the end of the code block. This is usually done using a Reset statement or a ResetAll statement. ResetAll (qubits = Zero);

Gate Operations:
These are applied as Gate_Name(Qubit(s) affected);
The most common gate operation you will come across is the H or Hadramard gate and the CNOT or Controlled Not gate. These two gates are needed to simulate quantum entanglement. With proper entanglement you can theoretically interact and affect qubits in other quantum computers far away from the one you are using.

The H gate creates the superposition state on the qubit it interacts with. It is used simply by: H(qubit);

The CNOT gate takes two qubits and checks to see if the first one is a 1. If it is, it flips the other, otherwise it does nothing. This links them together. You can chain qubits together by linking new ones to the entangled pair's qubits. It is used by: CNOT(firstQubit, secondQubit);

Pauli operators correspond to matrix multiplications and take a single Qubit.

X also known as a NOT gate applies the matrix {[1,0][0,1]}
Y applies the matrix {[0, -i][i,0]}
Z applies the matrix {[1,0][0,-1]}
the PauliI operator I just checks to see if the qubit's inverse is the same.
Rx, Ry and Rz perform rotations around the Pauli axis listed above.
R1 is applied to the qubit's eigenstate (which is just a fancy way of saying probability of being a 1 or 0) which is represented by the equation prob0 + or – prob1 = 1.

The H gate as mentioned above creates the superposition state by applying a 1/Sqrt(2) to a matrix whose values are all 1s to the qubit. This symbolizes the state that it is equally likely a qubit will be measured as a 1 or a 0.

The T gate performs a rotation and is often chained with the H gate.

CCNOT performs a controlled Not operation like mentioned above but using a separate qubit as a control. Meaning this gate accepts 3 qubits. 2 to entangle, 1 to measure by.

The SWAP gate swaps the quantum state of two entangled qubits.

Debugging is performed using Assert and AssertProb operations. Their signatures are as follows.

operation Assert (bases : Pauli[], qubits : Qubit[], result : Result, msg : String) : ()

operation AssertProb (bases : Pauli[], qubits : Qubit[], result : Result, prob : Double, msg : String, tol : Double) : ()

If either assert fails, it ends with calling the fail block which prints the message to the console.

For more information check out the Documentation here: https://docs.microsoft.com/en-us/quantum/?view=qsharp-preview

The C# code reference for the driver to run the simulator can be found here: https://docs.microsoft.com/en-us/dotnet/api/microsoft.quantum.simulation.simulators.qctracesimulators?view=quantumsdk-dotnet-0.1.1712.901-preview

I wont go much into it here, but if you use new QuantumSimulator(); it runs the local simulator included in the SDK. If you use new QCTraceSimulator(); it runs the Azure simulator that can run

simulations on operations requiring greater than 40 qubits.
More information on it can be found in the Managing Quantum Machines and Drivers subsection of the documentation.

Now lets look at some code. (Please see Next Page for the Code.)
The code is an image to preserve formatting. Comments were added to aid in understanding.
Also there are a couple of interesting factoids.
This is Microsoft's Teleportation example, which was replaced in the Documentation with creating a Bell State. I think the Teleportation example is easier to understand as a beginner.

References Copied from Slides:

SDK: https://www.microsoft.com/en-us/quantum/development-kit

Documentation: https://docs.microsoft.com/en-us/quantum/quantum-qr-intro?view=qsharp-preview

IBM 5Qubit Graphical Simulator: https://quantumexperience.ng.bluemix.net/

Quora: What is a Qubit? https://www.quora.com/What-is-a-Qubit-1

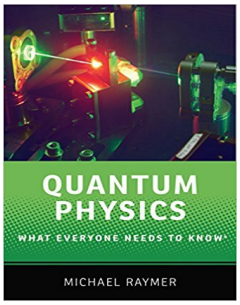
Video of the inside of a Quantum Computer: https://www.youtube.com/watch?v=60OkanvToFI&t=600s

How Quantum Computers Work: https://computer.howstuffworks.com/quantum-computer.htm

Gizmodo's Article on Quantum Computers: https://gizmodo.com/what-the-hell-is-a-quantum-computer-and-how-excited-sho-1819296509

Quantiki: https://www.quantiki.org/

A Series of Lecture Slides: https://www.cl.cam.ac.uk/teaching/0910/QuantComp/notes.pdf



Also Book: Quantum-Physics-Everyone-Needs-Know

```
namespace Quantum.Teleportation
{

        open Microsoft.Quantum.Primitive;
        open Microsoft.Quantum.Canon;

    operation EPR(Qubit: Q1, Qubit: Q2):(){
        body {
            //This operation causes the entanglement of two qubits and will form the basis of future computation.
            H(Q1);
            CNOT(Q1, Q2);
            // EPR stands for Einstein–Podolsky–Rosen paradox. It is a reference to the idea that the concept of quantum entanglement
            //(specifcally that the wave function did not take into account full description of reality)
            //violated the laws of known physics at the point in time when it was first being discussed.
            //It is named for the three physicists involved.
        }
        }

    operation Teleport(Qubit: MessageQ, Qubit: Here, Qubit: There):(){
        //Remember Qubits are 3Dimensional objects, with an X, Y and Z rotation.
        //They are often pictured as spheres on a grid (Bloch Sphere)

        body
        {
            //Entangles Here to There
            EPR(Here, There);
            //Entangles MessageQ to Here
            CNOT(MessageQ, Here);
            //Creates an equal probability of MessageQ to be a 0 or 1, creating the superposition state
            H(MessageQ);
            //Sets a variable to the measurement of Here
            let M_Here = M(Here);

            if (M_Here == One)
            {
                X(There);//Flip the X direction of There
            }
            //Sets a variable to the measurement of MessageQ
            let M_Message = M(MessageQ);

            if (M_Message == One)
            {
                Z(There);//Flip the Z direction of There
            }
        }

    }

    operation TeleportTest(Result: Msg):(Result){
        body
        {
            //Remember mutable is the only way to create a variable that can be changed
            mutable ResultVal = Zero;

            using (qubits = Qubit[3])//Initialize 3 qubits
            {
            let MsgQ = qubits[0];
            //Set MsgQ to the Result Message State
            Set(Msg, MsgQ);

            Teleport(MsgQ, qubits[1], qubits[2]);

            //ResultVal is set to the Measurement of the third Qubit.
            //When passed through Teleport, the state of the first qubit should carry over to the state of the third due to entanglement.
            //In future quantum computers, this can occur over several miles.
            set ResultVal = M(qubits[2]);

            }
            return ResultVal;
            }

        }
        //This experiment is usually depicted as performed by two fictional physicists named Alice and Bob
        //working in two separate laboratories, using prepared electrons or photons.
        //Entanglement was confirmed to exist beyond a shadow of doubt, in 2015.
}
```