今天我们将学习变量与可变性。声明变量是使用let关键字。而且在默认形况下变量是不可变的，一旦一个变量被绑定到一个值之后，这个变量就不能被重新赋值了，如果重新赋值的话，那么在编译的时候就会报错。声明变量时，在变量前面加上mut，就可以使变量可变。

下面我们来看一个例子。我们在命令行中输入’cargo new variables’，然后用vscode打开’main.rs’文件，首先我们定义一个变量x， 使得x等于10，之后我们用一个println的宏，宏就相当于一个系统写好的函数可以直接用，然后我们用大括号，这里相当于一个占位符。占位符可以在打印时替换为逗号后面的内容，这里是x。现在看来x是不可变的，因为在声明x的时候，前面没有加上‘mut’这个关键字。如果这个时候我们重新给x绑定一个值，比如20，这时候编译器就会报错。我们运行一下看一看是什么报错。。。’cannot assign twice to imutable variable’。如果要想让这段代码编译通过，那么我们需要让x是一个可变的变量，也就是在x的前面加上一个’mut’。。。现在编译器不报错了。我们运行一下，看到现在可以运行。这里会有一个warning，说value assigned to ‘x’is never read. 因此我们要再次print这个x。这样就没有warning了。warning不是error，这是一个可能的警告，不是必须要进行修改。

下面我们来看变量与常量。常量在绑定以后也是不可改变的，但是它与不可变的变量有很多区别。第一点区别就是常量不可以使用mut关键字，因为常量永远都是不可变的，其次，声明常量需要使用const关键字，而不是let，而且类型必须被标注清楚。第三点，常量可以在任何作用域内进行声明，包括全局作用域。第四点，常量只可以绑定到常量表达式，无法绑定到函数的调用结果也无法绑定到只能在运行时才能算出来的值。而且在程序的运行期间，常量在其声明的作用域内一直有效。

下面说一个概念叫做隐藏，在Rust中可以使用相同的名字声明新的变量。新的变量会隐藏之前声明的同名变量。我们看一下例子。。。定义x = 10，然后我直接给x赋值为20，这是不可以的，以为你前面说过如果不指定mut的话，x是不可变的，但是如果我们用let，这里重新制定了x为20，这就是隐藏，但是之前的x也就“死了”，被隐藏了。隐藏其实还可以改变绑定的类型，可以绑定数字也可以绑定一句话等。但是如果用mut赋值的话，我们之后并不能改变x的类型了。mut只是说x里装的数据可以变，但是数据类型不能变。关于数据类型，我们下节课会讲。

Today we will learn about variables and variability. Declaring variables is to use the let keyword. And in the default state, the variable is immutable. Once a variable is bound to a value, the variable cannot be reassigned. If it is reassigned, an error will be reported during compilation. When declaring a variable, add mut in front of the variable to make the variable that can be changed.Let's look at an example.

We enter'cargo new variables' in the command line, and then use vscode to open the'main.rs' file. First, we define a variable x so that x is equal to 10. Then we use a println macro, which is equivalent to a system function can be used directly, and then we use braces, which is equivalent to a placeholder here. The placeholder can be replaced with the content after the comma when printing, here is x. Now it seems that x is immutable, because when x is declared, the keyword ‘mut’ is not added in front. If we bind a value to x again, such as 20, the compiler will report an error at this time. Let's run it to see what the error is. . .

’Cannot assign twice to imutable variable’. If we want this code to compile and pass, then we need to make x a variable that can be modified. that is, add a ‘mut’ in front of x. . .

Now the compiler does not report an error. Let's run it and see that it can run now. There will be a warning here, saying that value assigned to ‘x’ is never read. So we have to print the x again. Then there is no warning.

Let's look at variables and constants.Constants are also immutable after binding, but there are many differences between them and immutable variables. The first difference is that constants cannot use the mut keyword, because constants are always immutable. Secondly, the const keyword is required to declare a constant instead of let, and the type must be clearly marked. The third is that constants can be declared in any scope, including the global scope.The fourth is that constants can only be bound to constant expressions. They cannot be bound to the result of a function call, and they cannot be bound to a value that can only be calculated at runtime. And during the running of the program, the constant is always valid in the scope of its declaration.

Let's say that a concept is called shadowing. You can declare new variables with the same name in Rust. The new variable will hide the previously declared variable with the same name. Let's look at an example. . .

Define x = 10, and then I directly assign the value of x to 20. This is impossible. Because that we said earlier that if you don’t specify mut, x is immutable, but if we use let, we re-set x to 20. This is shadowing. the previous x is also "dead" and is been “shadowing”. In fact, shadowing can also change the type of binding, you can bind a number or a String and so on, but if we use mut assignment, we can't change the type of x later. Mut just means that the data contained in x can be changed, but the data type cannot be changed. Regarding data types, we will talk about it in the next class.