```d
// You know what's coming...
module hello;

import std.stdio;

// args is optional
void main(string[] args) {
    writeln("Hello, World!");
}
```

If you're like me and spend way too much time on the internet, odds are you've heard about D. The D programming language is a modern, general-purpose, multi-paradigm language with support for everything from low-level features to expressive high-level abstractions.

D is actively developed by a large group of super-smart people and is spearheaded by Walter Bright and Andrei Alexandrescu. With all that out of the way, let's look at some examples!

```d
import std.stdio;

void main() {

    // Conditionals and loops work as expected.
    for(int i = 0; i < 10000; i++) {
        writeln(i);
    }

    // 'auto' can be used for inferring types.
    auto n = 1;

    // Numeric literals can use '_' as a digit separator for clarity.
    while(n < 10_000) {
        n += n;
    }

    do {
        n -= (n / 2);
    } while(n > 0);

    // For and while are nice, but in D-land we prefer 'foreach' loops.
    // The '..' creates a continuous range, including the first value
    // but excluding the last.
    foreach(n; 1..1_000_000) {
        if(n % 2 == 0)
            writeln(n);
    }

    // There's also 'foreach_reverse' when you want to loop backwards.
    foreach_reverse(n; 1..int.max) {
        if(n % 2 == 1) {
            writeln(n);
        } else {
            writeln("No!");
        }
    }
}
```

We can define new types with `struct`, `class`, `union`, and `enum`. Structs and unions are passed to functions by value (i.e. copied) and classes are passed by reference. Furthermore, we can use templates to parameterize all of these on both types and values!

```
// Here, 'T' is a type parameter. Think '<T>' from C++/C#/Java.
struct LinkedList(T) {
    T data = null;

    // Use '!' to instantiate a parameterized type. Again, think '<T>'.
    LinkedList!(T)* next;
}

class BinTree(T) {
    T data = null;

    // If there is only one template parameter, we can omit the parentheses.
    BinTree!T left;
    BinTree!T right;
}

enum Day {
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
}

// Use alias to create abbreviations for types.
alias IntList = LinkedList!int;
alias NumTree = BinTree!double;

// We can create function templates as well!
T max(T)(T a, T b) {
    if(a < b)
        return b;

    return a;
}

// Use the ref keyword to ensure pass by reference. That is, even if 'a' and 'b'
// are value types, they will always be passed by reference to 'swap()'.
void swap(T)(ref T a, ref T b) {
    auto temp = a;

    a = b;
    b = temp;
}

// With templates, we can also parameterize on values, not just types.
class Matrix(uint m, uint n, T = int) {
    T[m] rows;
    T[n] columns;
}

auto mat = new Matrix!(3, 3); // We've defaulted type 'T' to 'int'.
```

Speaking of classes, let's talk about properties for a second. A property is roughly a function that may act like an lvalue, so we can have the syntax of POD structures (`structure.x = 7`) with the semantics of getter and setter methods (`object.setX(7)`)!

```
// Consider a class parameterized on types 'T' & 'U'.
class MyClass(T, U) {
    T _data;
    U _other;
}

// And "getter" and "setter" methods like so:
class MyClass(T, U) {
    T _data;
    U _other;

    // Constructors are always named 'this'.
    this(T t, U u) {
        // This will call the setter methods below.
        data = t;
        other = u;
    }

    // getters
    @property T data() {
        return _data;
    }

    @property U other() {
        return _other;
    }

    // setters
    @property void data(T t) {
        _data = t;
    }

    @property void other(U u) {
        _other = u;
    }
}

// And we use them in this manner:
void main() {
    auto mc = new MyClass!(int, string)(7, "seven");

    // Import the 'stdio' module from the standard library for writing to
    // console (imports can be local to a scope).
    import std.stdio;
    // Call the getters to fetch the values.
    writefln("Earlier: data = %d, str = %s", mc.data, mc.other);

    // Call the setters to assign new values.
    mc.data = 8;
    mc.other = "eight";

    // Call the getters again to fetch the new values.
    writefln("Later: data = %d, str = %s", mc.data, mc.other);
}
```

With properties, we can add any amount of logic to our getter and setter methods, and keep the clean syntax of accessing members directly!

Other object-oriented goodies at our disposal include interfaces, abstract classes, and overriding methods. D does inheritance just like Java: Extend one class, implement as many interfaces as you please.

We've seen D's OOP facilities, but let's switch gears. D offers functional programming with first-class functions, `pure` functions, and immutable data. In addition, all of your favorite functional algorithms (map, filter, reduce and friends) can be found in the wonderful `std.algorithm` module!

```
import std.algorithm : map, filter, reduce;
import std.range : iota; // builds an end-exclusive range

void main() {
    // We want to print the sum of a list of squares of even ints
    // from 1 to 100. Easy!

    // Just pass lambda expressions as template parameters!
    // You can pass any function you like, but lambdas are convenient here.
    auto num = iota(1, 101).filter!(x => x % 2 == 0)
                           .map!(y => y ^^ 2)
                           .reduce!((a, b) => a + b);

    writeln(num);
}
```

Notice how we got to build a nice Haskellian pipeline to compute num? That's thanks to a D innovation know as Uniform Function Call Syntax (UFCS). With UFCS, we can choose whether to write a function call as a method or free function call! Walter wrote a nice article on this here. In short, you can call functions whose first parameter is of some type A on any expression of type A as a method.

I like parallelism. Anyone else like parallelism? Sure you do. Let's do some!

```
// Let's say we want to populate a large array with the square root of all
// consecutive integers starting from 1 (up until the size of the array), and we
// want to do this concurrently taking advantage of as many cores as we have
// available.

import std.stdio;
import std.parallelism : parallel;
import std.math : sqrt;

void main() {
    // Create your large array
```

```d
    auto arr = new double[1_000_000];

    // Use an index, access every array element by reference (because we're
    // going to change each element) and just call parallel on the array!
    foreach(i, ref elem; parallel(arr)) {
        elem = sqrt(i + 1.0);
    }
}
```