1. Because it can help you make better programming decisions when working with a singular language, or to help you bridge your understanding from one lang to another. Example: knowing how optimizers improve your code for you, so you know what good code looks like, as well as what the compiler will do for you.

2. fortran

3. cobol

4. c (and in-line asm) also Linux started adopting Rust in parts

5. You wind up with c++! Just joking... Having multiple ways to accomplish the same thing leads the programmer to debate best practice in their head, slowing them down. It also aids in readability to only have one way to do things, unifying codebases with multiple developers by force.

6. I'm assuming this is asking "what is an example of Java having multiple ways to accomplish the same thing", then interfaces and inheritance. If you have an interface with (I forget the exact syntax so I'm going to pseudocode it out)

interface dog:
ears, tail, nose

... and a class with the same features

class dog:
ears, tail, nose

then inheriting from either of these will provide the same results.

7. Ortogonality is a quasi-measure of how universal and independent lang features are. For example, an array in c can be of any type, from primitives to custom types to structs. Arrays are "orthogonal" because a ton of possibility opens up for data structures.

8. I think you could argue either way. I would say that orthogonal features are in general good and keep the language simple. In C, it makes sense to have the highly orthogonal struct as a cornerstone of the language. Some would call c

complex, but in my opinion it's one of the simplest languages -- it literally has the least number of features of most of the modern languages.

9. oh, cool, this dismisses my last answer so let's go for it.

1. pointer decay is annoying and represents a lack of orthogonality.

```
int* x()
{
   int z = 2;
   int y* = &z;
   return y;
}
```

when I dereference y, it will be an error or just a random memory address... however, in my opinion this makes perfect sense because y* points to nothing after that 2 goes out of scope. But let's just say that I don't believe that... there's my answer!

2. Structs and pointer decay

This is actually a really weak point of c. Building on my first answer:

```
Struct a
{
   struct* b structpointer;
};

struct b
{
   int x;
};
```

struct a contains one reference to a struct b. But see what happens when you try to reference b.

```
foo()
{
   struct a Alpha;
   struct b Beta;
```

```
    Alpha.structpointer = &Beta;
    return Alpha;
}

// then after you return Alpha, it's pointer decays to nothing but Alpha stays
and is passed as a copy.
// again, you're to use malloc here, but this gets pretty dicey pretty fast...
// now then, I argue that you should always store malloc'd pointers in a data
structure that you can easily loop through, calling free on each element, but
not everyone can be as cool as me I guess
```

3. I'm having a really hard time here. Void pointers are really not orthogonal. For example:

```
int main()
{
    void* x;
    int y;
    char* z = "hello world";
    char* compare = "goodbye, cruel world!";

    // make my void pointer point to my integer
    x = &y;

    // then if I go
    strcmp(compare, x);
    // I can't do that because my void pointer doesn't point to a string

    // likewise if my void pointer points to my string
    x = z;
    // then I can't do things that you can do to ints with them
    &x += 2;
}

// but again, I'd argue that void pointers are MUCH cleaner than the very
annoying type generics that you get in java. I'd argue that java's way is far
less orthogonal, but that's just my opinion I guess.
```

10. Type checking prevents what I just discussed in part 3 of the last question. You get more robust language server processing, syntax checking and

completion, etc. Without static typing, the compiler can't tell you before runtime if my strcmp(compare, x) is valid.

11. a, java

12. optimization

13. True

14. T

15. F, I think? In the class you're building overloaders for it probably leads to reduced readability, for the benefit of that class' implementation being far more readable. Don't want to be building a strcat() function in every file and such, haha. But also it can reduce readability if you have some weird class and have no idea what + or < might actually do. like:

```
class dog:
   ears, nose

foo:
   dog 1, dog 2;
   if dog 1 < dog 2:
      return true
```

How could dog 1 be less than dog 2? its weight? its height? its dollar value? how cute it is?? haha

16. It's hard to find a non-reliable programming language in the langs we typically write on a day to day basis (except for javascript, haha!). Reliability tells us how deterministic its outputs are. So, the most reliable languages have strong type systems and stability. Python programs can be extremely unpredictable on both of these fronts: they can run the same code in wildly different lengths of time, and the weird typeing can make it hard to predict why bugs occur. For example:

```
def pythonfunction():
   # if this fails, there's nothing preventing bugs from occuring
   x = someclass()
```

```
# also there's no knowing if this will work properly:
# the lsp can't predict this being a possible bug
x += 2
```

17. Exception handling is dealing with edge cases in your code. Java throws an exception when you, for example, divide by zero. So you set up a try catch block, and in the event of a divide by zero, you can have code that deals with that situation.

18. Generality is how universal its problem solving capabilities are. A programming language without the ability to perform addition would not be very general.

Portability means "write once, debug everywhere!". Just joking, portability is a measure of how reusable the same code is on multiple platforms. An example is c on ARM vs x86, with or without an OS, certain features of the language compile and run the exact same.

19. I have no idea, I think it'd be a pretty neat feature, except that most programmers can't use pointers cleanly! Though because java has method variables, pointers are less necessary. Since classes have access to their method variables and the ones they inherit... well, this basically acts like a pointer and solves the problem of having to copy memory all around the place. You get enough memory cleanliness this way.

20. To turn code into machine code.

21. To store variables and other identifiers.