Three useful categories

Learning a programming language involves:

Syntax: The grammar rules defining a program fragment.

Semantics: The meaning of various programming fragments.

Pragmatics: How to *effectively* use language features, libraries, IDEs, ...

All three of these are important in how easy it is to easily write high-quality software.

For all categories, consider the principle of least surprise.

Some vocabulary

Do *not* confuse the following four!

- *value*:

- *variable*:

- *type*:

- *expression*:

Some vocabulary

Do *not* confuse the following four!

- *value*: a datum – the fundamental piece of information that can be represented in the program

  E.g. `37` or `"hi"`. Values can be passed to functions, returned, stored in variables.
- *variable*:

- *type*:

- *expression*:

Some vocabulary

Do *not* confuse the following four!

- *value*: a datum – the fundamental piece of information that can be represented in the program

  E.g. `37` or `"hi"`. Values can be passed to functions, returned, stored in variables.
- *variable*: an identifer which, at run time, evaluates to some particular value.
- *type*:

- *expression*:

Some vocabulary

Do *not* confuse the following four!

- *value*: a datum – the fundamental piece of information that can be represented in the program

  E.g. `37` or `"hi"`. Values can be passed to functions, returned, stored in variables.
- *variable*: an identifer which, at run time, evaluates to some particular value.
- *type*: a set of values

  E.g. Java's `short` = {-32768,..., -1,0,+1,+2, ..., +32767}.
- *expression*:

Some vocabulary

Do *not* confuse the following four!

- *value*: a datum – the fundamental piece of information that can be represented in the program

  E.g. `37` or `"hi"`. Values can be passed to functions, returned, stored in variables.

- *variable*: an identifer which, at run time, evaluates to some particular value.

- *type*: a set of values

  E.g. Java's `short` = {-32768,..., -1,0,+1,+2, ..., +32767}.

- *expression*: a piece of syntax which evaluates to some particular value.

  E.g. `3+4*5` or `sqrt(16)`.

Some vocabulary (cont.)

- *literal:* a piece of syntax which evaluates immediately to a particular value.

  E.g. Java `37` or `045` are both literals representing the value 37, which is of *type* `int`. And `37.`, `37d`, `37e0` are each literal `double` s.

  (We *will* often conflate a literal with the value it represents, and only say "literal" when we're emphasizing that we're dealing with syntax.)

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3).equals("hay")
```

Morever: string-literals with + are computed at compile-time.

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3).equals("hay") // true
```

Morever: string-literals with `+` are computed at compile-time.

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3).equals("hay") // true
"Cathay".substring(3) == "hay"
```

Morever: string-literals with `+` are computed at compile-time.

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3).equals("hay") // true
"Cathay".substring(3) == "hay"      // false
```

Morever: string-literals with + are computed at compile-time.

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3).equals("hay") // true
"Cathay".substring(3) == "hay"      // false
"Cathay" == "Cathay")
```

Morever: string-literals with + are computed at compile-time.

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3).equals("hay") // true
"Cathay".substring(3) == "hay"      // false
"Cathay" == "Cathay")               // true (!)
```

Morever: string-literals with + are computed at compile-time.

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3).equals("hay") // true
"Cathay".substring(3) == "hay"      // false
"Cathay" == "Cathay")               // true (!)
```

Morever: string-literals with + are computed at compile-time.

```
"Cat" + "hay" == "Cathay"
```

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3).equals("hay") // true
"Cathay".substring(3) == "hay"      // false
"Cathay" == "Cathay")               // true (!)
```

Morever: string-literals with + are computed at compile-time.

```
"Cat" + "hay" == "Cathay"       // true (!)
```

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3).equals("hay") // true
"Cathay".substring(3) == "hay"      // false
"Cathay" == "Cathay")               // true (!)
```

Morever: string-literals with + are computed at compile-time.

```
"Cat" + "hay" == "Cathay"        // true (!)
"Cat".concat("hay") == "Cathay"
```

(*) This optimization is only safe because Java strings are immutable.

trivia: Interning Java string-literals

Literals occur in the source-code text, and can be processed at compile-time. In Java, string literals are "interned": If the same string-literal occurs twice, the the compiler is smart enough to only make one object(*), and use the same reference in both places.

```
"Cathay".substring(3).equals("hay") // true
"Cathay".substring(3) == "hay"      // false
"Cathay" == "Cathay")               // true (!)
```

Morever: string-literals with + are computed at compile-time.

```
"Cat" + "hay" == "Cathay"        // true (!)
"Cat".concat("hay") == "Cathay" // false
```

(*) This optimization is only safe because Java strings are immutable.

typing: when?

statically-typed: At compile-time, the types of all declared names are known.

Can be provided by programmer and checked by type-system, or inferred by the language (ML, Haskell).

(C# allows simple `var n = 5;` and infers `n` ∈ int).

dynamically-typed: Language knows the type of every value.

But a variable might hold values of different types, over its lifetime. php, javascript, racket. Each value may include some extra bits, indicating its type.

typing: other approaches

duck typing: Care about an object having a field/method, not any inheritance.

E.g. javascript

untyped:

E.g. assembly

type-safe: Any type error is caught (either dynamically or statically).

Note that C is not type-safe, due to casting. Java's casting *is* type-safe (since a bad cast will fail at run-time).

typing: strong/weak/non

These terms are often used in different ways:

strongly typed: no/few implicit type conversions,
or statically typed

weakly typed: many implicit type conversions,
or dynamically typed

Consider Java `Math.sqrt(16)`, and Java vs php `"50" + 60`.

Compiling vs Interpreting

- A compiler is a function

  $compile : source\text{-}code \rightarrow machine\text{-}code$

  The resulting machine-code, when executed, runs the program which produces a resulting value.

- An interpreter is a function

  $eval : expr \rightarrow value$

  which evaluates an expression, producing a resulting value.

Compiling vs Interpreting (cont.)

- Running interpreted code, you are running the interpreter, which is looking at the source-expression as if it were data.
- Running compiled code, you are running the program directly.
- Compiled code: faster, but platform-specific.

The distinction is practical, but not fundamental: You can even claim that CPUs are simply interpreters: they read compiled-code as data (from memory), and update their internal state accordingly.

- A compromise: compile to *byte code* (or, javascript), and run an interpreter for that byte code. A speed/platform-dependence trade-off.