Notes

Programming Language

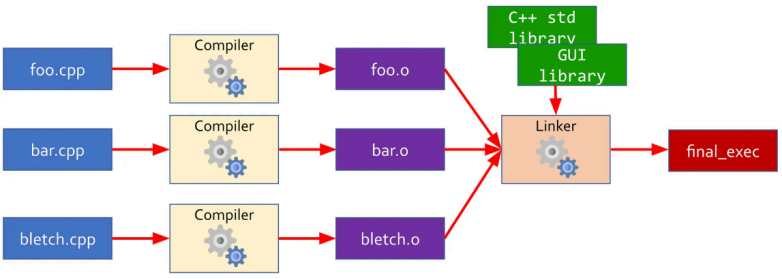
Paradigms and Dimensions

- The Major Language Paradigms
  - Imperative(명령) Language: made up of statements, loops, and mutable variables
  - OOP Languages: organized into classes and objects that send messages to each other
  - Functional Languages: no statements or iterations, only expressions, functions, constants, and recursion
  - Logic Languages: define a set of facts
    - ex. like(dogs, meat)
  - Many languages are Hybrids like C++, Scala, etc
- Language Dimensions
  - Type checking: static, dynamic
  - Parameter passing: by-value, by-reference, by-pointer, by-object reference, by-name
    - Pass by Value: copy of the original variable
    - Pass by Reference: alias for the variable
    - Pass by Pointer: contains the address of the original variable
  - Scoping: lexical, dynamic
  - Memory management: manual, automatic

Specification

- Syntax Spec: specifies sequences of tokens are grammatically allowed
  - use Extended Backus-Naur Form(EBNF) to define the language's syntax
- Semantic Spec: details areas such as type checking, the behavior of operators, how parameters are passed, etc

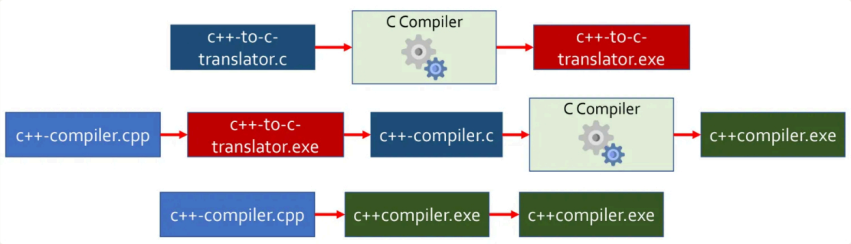
Compiler and Linker



- **Compiler**: translates program source into object modules (machine language or bytecode)
  - Lexical analyzer: breaks the source file into a stream of tokens (lexical units)
    - ex. while, (, i, <, 5,), ...
  - Parser: uses the language's grammar spec to validate that the tokens have a valid syntax
    - if the syntax is valid, the parser converts the tokens into an AST(Abstract Syntax Tree)

- Semantic Analyzer: checks the semantic validity of the tree and returns an annotated parse tree
  - ex. type checking, etc
  - updates the tree's node with details required for code generation
- Intermediate Representation Generator(IRG): produces an abstract representation of the program
  - totally independent of the original language
  - the representation could be a tree structure or CPU-agnostic instructions
    - ex. load i, load_constant 5, less_than, jump_if_zero L1, ...
- Code Generator: converts the IR to machine language or bytecode output
  - bytecode is a binary encoding like a machine language, but not for a real CPU. it is processed by an interpreter or just-in-time(JIT) compiler.

> C++ compilers are written in C++ through a process called "bootstrapping"



- **Linker**: combines multiple object modules and libraries into a single executable file or library

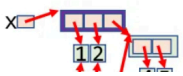
Interpreter

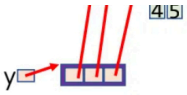
- directly executes program statements, without requiring them to first be compiled into machine language
- no machine code generated
- translated into lower-level representations at run time → overhead
- Process
  1. Load source file into RAM
  2. Initialize data structures needed by the interpreter
     - ex. create a variable to track what line of the program we're going to run next
     - ex. create a data structure to track the current value of each variable in the program
  3. Fetch the next statement to run
  4. Interpret the current statement and update the program's state
  5. Advanced to the next statement
  6. If the program has not finished running, go back to 3 and repeat

Python

Object References

- Every Python object(variable) is allocated on the heap (similar to pointers in C++)
- Assignment of object references(eg. `c2 = c`) does NOT make a copy of the object
- `isinstance(obj, class)`: return `True` if the object is an instance of the class or instance of a subclass
- Copy method
  - `import copy`
  - Deep copy: `copy.deepcopy(x)`
    - make a copy of the top-level object and every object referred to directly or indirectly by the top-level object
  - Shallow copy: `copy.copy(x)`
    - a copy of the top-level object only





- Immutable type: numbers, string, tuple, complex, frozenset
  - `arr += ['bean']`: in place replacement; same as `arr.append('bean')` and `arr[len(arr):] = ['bean']`
  - however, `arr = arr + ['bean']` is different, it creates a new list!

Default Parameters

Python does not reset the default value back to its initial value.

```
def puzzle(w, words = []):
  words.append(w)
  print(words)

puzzle("This") # ["This"]
puzzle("is") # ["This", "is"]
puzzle("confusing!") # ["This", "is", "confusing"]
```

This happens with any default value that's a mutable object like arrays, dict, and sets.

The way to fix:

```
def puzzle(w, words = None):
  if words is None:
    words = []
  words.append(w)
  print(words)
```

Class

- if there is no instance variable `joke` in the self instance, then `self.joke` will reference the class variable `joke` on the class that self is an instance of (in this case, Joker)
- However, if an instance variable `self.joke` exists, it will shadow the class variable of the same name, overriding and hiding it.

Inheritance

- The subclass constructor does not automatically call the superclass constructor
  - must EXPLICITLY call the super class's constructor to run it!
  - ex. `super().__init__()`
- Even though an overridden function call is coming from the base class, Python calls the subclass method

```
class Pet:
  def __init__(self):
    self.var = 1
    print("Pet")
  def play(self):
    self.do_playful_thing()
  def do_playful_thing(self):
    print("Pet play")
  def print_var():
    print(self.var)

class Dog(Pet):
  def __init__(self):
    self.var = 2
    print("Dog")
  def do_playful_thing(self):
    print("Dog play")
  def run(self):
    super().play();
```

```
x = Dog() # does not print "Pet" since the constructor is not calling super().__init__()
x.play() # prints "Dog play", not "Pet play" !!!
x.run() # this also prints "Dog play"
x.print_var() # prints 2 !! if self.var is not initialized in Dog class nor superclass is initialized, it wil
```

Handling Errors and Exceptions

- Look Before You Leap (LBYL)
  - prevent errors or exceptional situations from happening
  - conditional statements
  - common in C, Java
- Easier to Ask Forgiveness than Permission (EAFP)
  - handle errors or exceptional situations after they happen
  - exception-handling; try-catch statement
- Comparison criteria
  - Number of checks
    - EAFP can avoid unnecessary duplicated checks
  - Readability and clarity
    - EAFP puts the most common case on the front
  - Race condition risk
    - In a multi-threaded environment, the LBYL approach can risk introducing a race condition between “the looking” and “the leaping”
  - Code performance
    - exception handling is fast and efficient in Python
    - if your code faces only a few errors, then EAFP is probably the most efficient strategy
- EAFP gotchas
  - you must not run code with side effects
  - use concrete exceptions as much as possible
  - use LBYL when code run into a ton of errors and exceptional situations
- Checking for Objects’ Type and Attributes
  - Checking for an object’s type should be avoided as much as possible
  - Python typically relies on an object’s behavior rather than on its type
  - interacts with objects by directly calling their methods and accessing their attributes without checking the object’s type beforehand

```
def add_user(username, users):
    if isinstance(users, list): # LBYL
        users.append(username)
    try: # EAFP
        users.append(username)
    except AttributeError:
        pass
```

```
from pathlib import Path

file_path = Path("/path/to/file.txt")

if file_path.exists():
try:
    with file_path.open() as file:
        print(file.read())
else:
```

```
except IOError as e:
    print("file not found")
```

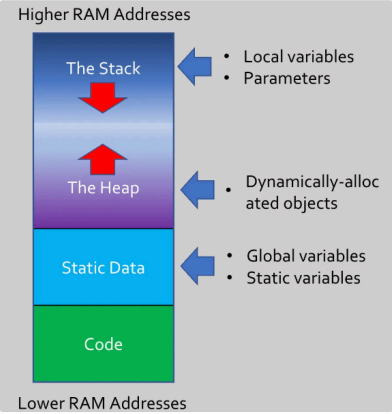
Numpy

- 💡 Why Numpy dot product is significantly faster than the Python list operation?
  - Numpy implementation heavily relies on pre-compiled, optimized C code
    - in Python, the interpreter will convert the bytecode into machine-executable instructions on each invocation of the dot product
    - this process has already been done ahead of time for the C code
  - Numpy provides support for contiguous arrays where each element is next to each other in memory
    - in Python, everything is an object, which means that the matrix variable contains a bunch of pointers → objects scattered through RAM (poor spatial locality)

Types

Variable and Value

- Variable has a name, type, value, binding, storage, lifetime, scope, and mutability
- Value does not have name, binding, and scope



Types

- Type defines a range of values, size and encoding, what operations we can perform, where it can be used, and how it's converted/cast to other types
- Types of Types
  - Primitives: int, float, char, bool, enum, pointer
  - Composites: string, struct(record), class, union, tuple, container(array, set, map, ..)
  - Others: generic type, function type, boxed type(an object whose only data member is a primitive type)
- Supertype and Subtype in inheritance
- **Value Type**: can be used to instantiate objects/values and define pointers/obj refs/references
  - ex. `Dog d("Kuma")`, `Dog *d`
- **Reference Type**: can only be used to define pointers/object references/references, but not instantiate objects/values
  - ex. abstract class cannot instantiate objects ⇒ ~~`Shape s(Blue)`~~ `Shape *s`
- Type Equivalence: the criteria by which a programming language determines whether two values or variables are of equivalent types
  - Name Equivalence: only if their type names are identical
    - most of statically typed languages

```
// type S and T are structurally identical
struct S { string a; int b; };
struct T { string a; int b; };
```

```
S s1, s2;
T t1, t2;
s1 = s2;
s1 = t1; // type mismatch error
```

- Structural Equivalence: if their structures are identical, regardless of their type names
  - most of dynamically typed languages

```
// type S and T are structurally identical
struct S { string a; int b; };
struct T { string a; int b; };

let s1, s2 : S;
let t1, t2 : T;
s1 = s2;
s1 = t1;
```

- Type Checking Approaches

	Static	Dynamic
Strong	C#, Go, Haskell, Java, Scala	Javascript, Python, Perl, PHP
Weak	Assembly, C, C++	n/a

Compile-time (Static) vs Run-time (Dynamic)

- **Static typing**
  - a type checker checks that all operations are consistent with the types of the operands being operated on prior to the program's execution
  - must have a fixed type bound to each variable at its time of definition
  - if the type checker can't assign distinct types to all variables, functions, and expressions and verify type compatibility, then it generates a compiler error
  - Type Inference: infer types without explicitly annotating types
    - ex. `auto` in C++, `:=` in Go
  - Downcast: type checking done at runtime
    - ex. `dynamic_cast` in C++
  - static type checking is conservative

```
// following code generates error: no member named 'bite' in 'Mammal'
void handlePet(Mammal& m) {
  if (m.name() == "Spot") m.bite() // fail because Mammal does not have bite
  if (m.name() == "Meowmer") m.scratch() // fail because Mammal does not have scratch
}
int main() {
  Dog d("Spot");
  Cat c("Meowmer");
  handlePet(d);
  handlePet(c);
}
```

  - + faster code, detect bugs earlier, no need to write custom code to check types
  - - conservative and may error out on perfectly valid code, type checking phase before execution
- **Dynamic Typing**
  - the safety of operations on variables/values is checked as the program runs
  - if the code attempts an illegal operation on a value, an exception is generated or the program crashes
  - types are associated with values and not variables

- the compiler/interpreter stores type information (type tag) along with every value/object
- Type Annotation: annotate type but does not prevent changing to a different type of value
- Duck Typing: checks an object's suitability for an operation based on the presence of required methods rather than the object's type
- + flexibility, operate on different data types, simpler code, faster prototyping
- - detect errors much later, slow due to run-time type checking, more testing for the same level of assurance, no way to guarantee safety across all possible executions

- **Gradual Typing**
  - hybrid approach; can choose whether to specify a type for variables/parameters
  - type checking occurs partly before execution and partly during runtime
  - if a variable is untyped, then type errors for that variable are detected at runtime
  - if specify a type, then some type errors can be detected at compile time
  - may pass an untyped variable or expression to a typed variable. it will compile and check for errors at runtime

Strictness - Strong vs Weak

- **Strong Type Checking**
  - guarantees that all operations are only invoked on objects/values of appropriate types
  - no possibility of an unchecked runtime type error
  - Type-safety: prevent invalid operation
    - Validates all of the operands used in the expression have compatible types
    - All conversions/casts between different types are checked and if the types are incompatible, then an exception will be generated
  - Memory safety: prevent inappropriate memory access
    - All pointers are initialized to null or assigned to point at a valid object at creation
    - Forces all variables to be initialized before use or generates a runtime error if an initialized variable is accessed
    - Access to arrays is bounds-checked
    - Ensures objects can't be used after they are destroyed
    - if a language is not memory-safe, you might access a value using the wrong type
  - Checked Casts: a type-cast that results in an exception/error if the cast is illegal
  - + dramatically reduced software vulnerabilities, earlier detection and fixing of bugs/errors
  - - lower performance and legacy
- **Weak Type Checking**
  - does not guarantee that all operations are invoked on objects/values of appropriate types
  - not type-safe, not memory-safe
  - can have undefined behavior at runtime

> 💡 Untagged Unions
>
> - generally only provided in untyped languages or in a type-unsafe language (as in C)
> - do not keep track of the currently "active" data type
> - allow interpreting data in multiple ways without enforcing type correctness → weak typing
> - can lead to undefined or platform-dependent behavior

▼ Practice

```
sub ComputeSum {
  $sum = 0;
  foreach $item (@_) {
    $sum += $item;
  }
  print("Sum of inputs: $sum\n");
}
```

```
ComputeSum(10, "90", "cat");
# we've run this code a million times, and each time it prints "Sum of inputs: 100"
# => since this did not have undefined behavior or unchecked type error, it is a strong type checking !!
# => the language converts string to int, and a string without digits is treated as zero.
```

```
fun process(x: Any) {
  when (x) {
    is Int -> print(x)
    is String -> print(x.length)
    is IntArray -> print(x.sum())
    else -> print((x as Dog).bark())
  }
}
fun main() {
  var x = Person("Carey", "Nachenflopper")
  process(x);
}
// this generates a runtime error
// => it is strongly-typed because it is preventing invalid casting at runtime
// => it is statically typed because it is explicitely casting x to Dog
```

Conversions and Casts

- **Type Conversion**
  - generates a whole new value
  - typically used to convert between primitives
    - ex. `(int)pi`
- **Type Cast**
  - takes a value and views it as if it were another type of value; no new value is created
  - typically used with objects
    - ex. `Person &p = (Person&)mary`
- Categories
  - Explicit vs Implicit
  - Widening vs Narrowing
  - Checked vs Unchecked

Explicit vs Implicit

- Explicit: use explicit syntax to force the conversion/cast
  - Explicit Conversion
    - `(int)pi` in Python, `static_cast<int>(pi)` in C++, `as` in Rust, `parseInt(pi)` in Javascript
  - Explicit Cast: `(Mammal *) animal`
    - `dynamic_cast<Student*>(p)` (downcast) in C++, `as` in Kotlin, `(Student)` in Java
  - telling the compiler to change what would be a compile time error into a runtime check
  - in a strongly typed language, the program will still perform a runtime check before allowing the cast operation
- Implicit: happens without explicit syntax
  - Implicit Conversion(Coercions)
    - most languages have a prioritized set of rules that govern implicit conversions that are allowed
      - ex. C++ implicit conversion rules: long double > double > float > unsigned long int > long int/unsigned int
    - **Type Promotion**: implicit widening conversion; coercion that converts a narrow type into a wider type
  - Implicit Cast
    - most implicit casts are 'upcast' from a subclass to a superclass
    - static type language may not allow implicit cast

Widening vs Narrowing

- Widening Conversion: converts a narrower type to a wider type
  - ex. short → int, int → long, float → double, short(2bytes) → double(8bytes)
  - Value-preserving: the converted value is always the same
- Narrowing Conversion: converts from wider type to a narrower type or between two unrelated types
  - ex. long → int, float → int, int(4bytes) → float(4bytes)
  - NOT value-preserving: the converted/casted value might be different than the original
- Widening Cast (Upcast): casts a subtype variable as its supertype
  - ex. Student → Person
  - upcasts are always safe because that every subtype object is guaranteed to have all of the properties of the supertype
- Narrowing Cast (Downcast): casts a supertype variable as one of its subtypes
  - ex. Person → Professor
  - downcast may fail if the actual object is not compatible with the downcasted type
  - downcast must be explicit for static language

Checked vs Unchecked

- Checked
  - in a strongly typed language, every conversion/cast with the potential for an issue is checked for validity at runtime
- Unchecked
  - in a weakly typed language, some invalid conversions/cast may not be checked and lead to undefined behavior

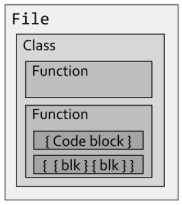
Scoping

Scope and Lifetime

- Scope: covers the visibility of variables and functions within a program
- In-scope (Active binding): a variable is in-scope if it can be accessed by its name in a particular part of a program
- Lexical Environment: the set of in-scope variables and functions at a particular point in a program
  - scope changes as a program runs and the environment changes as variables come in or go out of scope
- Lifetime (Extent): lifetime from its creation to destruction
  - a variable's lifetime may include times when the variable is in scope, and times when it is not in scope (but still exists and can be accessed indirectly)
  - some languages like Python allow explicitly controlling a variable's lifetime like `del`
  - values may have a lifetime that extends indefinitely and they are often independent of variables
    - ex. `d = Berry(); return d;` : d lifetime ends but the value it refers to is still alive out of the function

Lexical(Static) Scoping

- all programs are comprised of a series of nested contexts



- the most dominant scoping approach
- LEGB Rule: Local → Enclosing → Global (top-level var/func) → Built-in
- Expressions, Blocks, Functions, Classes/Structs, Namespaces, Global

Dynamic Scoping

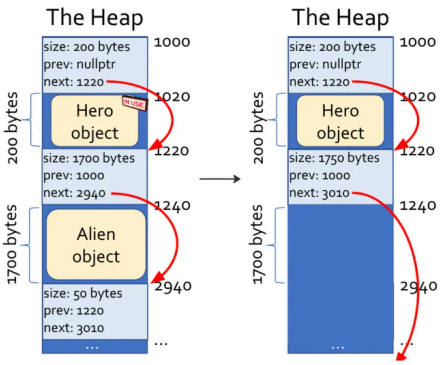
- If a variable can't be found, the program then searches the calling function for the variable.

Memory Management

❗ even languages with automated memory management can sometimes run out of memory

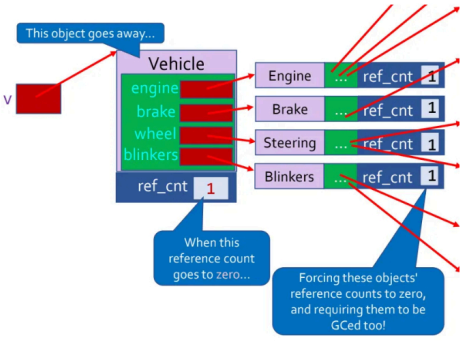
Garbage Collection

- Automatic reclamation of memory which was allocated by a program, but which is no longer referenced
  - the programmer does not explicitly control object destruction
  - ex. C#, Go, Java, Javascript, Python, Haskell, ...
- Pros
  - + eliminates memory leaks
  - + eliminates dangling pointers and use of dead objects
  - + eliminates double-free bugs
  - + eliminates manual memory management
- **Mark and Sweep**
  - ex. Go, Java, Javascript
  - Mark phase: identifies all objects that are still referred to and thus considered to be in use (similar to bfs)
    1. identifies all root objects and adds their object references to a stack or queue for investigation (global variables and local variables across all stack frames, and parameters on the call stack)
    2. uses the stack or queue to search from the root objects and mark all reachable objects as "in use"
  - Sweep phase: scans all heap memory from start to finish, and frees all blocks not marked as being in use
    - traverse all memory blocks in the heap (each block holds a single object/value/array) and examine each object's in-use flag
    - all memory blocks in the heap are linked together top-to-bottom in a linked list
    - if an object in the block is in use, remove the mark; otherwise, free the block
    - adjacent free blocks can then be coalesced into a single large block



  - Memory fragmentation: the heap becomes peppered with small, unused memory blocks where previously freed objects used to be
    - becomes slow to find free chunks of memory big enough to accommodate new object allocations
- **Mark and Compact**
  - ex. C#, Haskell
  - Mark phase: same as Mark and Sweep
  - Sweep phase: compact all marked/in-use objects to a new contiguous block of memory, then adjust all pointers to the proper relocated addresses
    - discover all active objects and move them into a new block of memory
    - the original block of memory is just treated as if it's empty and can be reused as a whole
    - - needs to reserve half the memory for compaction

- **Reference Counting**
  - ex. Python, Swift, Perl
  - each object keeps a count of the number of active object references that point at it
  - when an object is destroyed, all objects transitively referenced by that object must also have their reference counts decreased



  - removing a single reference can potentially lead to a cascade of objects freed at once ⇒ SLOW
  - So instead of destroying an object as soon as its count becomes zero, add it to a list of pending objects, and then reclaim memory regularly over time

> 💡 Choosing Garbage Collection
>
> - many objects of diverse sizes with frequent allocations and deletion: Mark and Compact
> - lots of objects with cyclical references to each other: avoid Reference Counting
> - low-RAM device: Mark and Sweep
> - a program for real-time devices: Reference Counting

Unpredictability

- Bulk garbage collection (M.S. & M.C.) occurs when free memory runs low (memory pressure) and the program's execution is frozen temporarily while this happens
- impossible to predict when a given object will actually be freed by the collector
  - what if each object creates a large temporary file?

Ownership Model

- every object is 'owned' by one or more variables in the program
- when the last owner variable's lifetime ends, the object it owns is freed automatically
- In Rust, every object is owned by a single variable in the program
  - ownership can be transferred to a new variable via assignment or parameter passing, invalidating the old variable

```
fn foo(s3: String) {
  println!("{}", s3);
} // s3's lifetime ends, string object freed
fn main() {
  let s1 = String::from("I owned!");
  let s2 = s1 // ownership transferred to s2
  println!("{}", s1); // compiler error
  foo(s2);
  println!("{}", s2); // compiler error
} // nothing left to free
```

  - Borrowing: a variable may refer to an object without taking ownership
    - the borrower may request exclusive read/write access (for thread safety) or non-exclusive read-only access

```rust
fn foo(s3: &String) {
  println!("{}", s3);
}
fn main() {
  let s1 = String::from("I owned!");
  let s2 = s1 // ownership transferred to s2
  foo(&s2);
  println!("{}", s2);
} // s2 goes out of scope, string object freed
```

- In C++, a smart pointer works like a traditional pointer but also provides automatic memory management
  - Unique Pointer: exclusively owns the responsibility for freeing a heap-allocated object
    - ex. `std::unique_ptr<Nerd> p = std::make_unique<Nerd>("Carey", 100);`
    - no duplicating or passing to other functions
      - ex. ~~`std::unique_ptr<Nerd> p2 = p;`~~, ~~`some_func(p);`~~
  - Shared Pointer: shares the responsibility for freeing a heap-allocated object
    - when the last shared pointer goes away, it frees the object
  - Weak Pointer

Object Destruction

- many objects hold resources (eg. network connections, temp files, other objects) which need to be released when their lifetime ends
- **Destructor**: a cleanup method that's automatically called by the language the instant an object's lifetime ends
  - used in Non-GC languages like C++, Rust
  - GUARANTEED to run the instant that the object's lifetime ends
  - deterministic rules that govern when destructors are run
- **Finalizer**: a cleanup method that is called JUST BEFORE the garbage collector reclaims an object's memory
  - to free all NON-MEMORY resources held by an object
  - used to release unmanaged resources like file handles or network connections which aren't garbage collected
  - called by garbage collector → finalizer may not run at a predictable time or at all
  - ex. `finalize()` in Java, `__del__()` in Python
- **Manual Disposal Methods**: like a destructor, but the programmer must update their program to call it when an object is no longer needed explicitly
  - to ensure objects free their resources in GC languages
  - a function that the programmer must manually call to free non-memory resources; not called automatically by GC
  - ex. `Dispose()` in C#

Mutability / Immutability

Immutability Approaches

- Class Immutability
  - can designate that all objects of a class are immutable after construction
  - ex. ~~`s[0] = 'J'`~~
- Object Immutability
  - can designate some objects of a particular class as immutable
  - ex. `const`
- Assignability Immutability
  - can designate that a variable may not be re-assigned to a new value
  - but mutation can be made to the original referred-to object
  - ex. `final`