What’s the most striking difference?

High-level languages have rich facilities for naming “things.”

Java Interface definition

x86 Assembly
Names

Required for abstraction.
- Assembly only has values & addresses & registers.
  - Machine dependence!

Names enable abstraction.
- Can refer to something without knowing the details (e.g., exact address, exact memory layout).
  - Let the compiler worry about the details.
- Can refer to things that do not yet exist!
  - E.g., during development, we can (and often do) write code for (Java) interfaces that have not yet been implemented.

Abstraction

Control Abstraction vs. Data Abstraction

Control Abstraction

- Can hide arbitrary complex code behind a simple name.
  - For example, addition can be simple (int) or “difficult” (vector).
Abstraction

Control Abstraction vs. Data Abstraction

Data Abstraction
Abstract Data Types (ADTs)
Reason about concepts instead of impl. details.
Programmer doesn’t know memory layout, address, whether other interfaces are implemented, what invariants need to be ensured, etc.

Binding
Associating a name with some entity.
(or “object,” but not the Java notion of an object)

Binding vs. Abstraction.
- Introducing a name creates an abstraction.
- Binding a name to an entity resolves an abstraction.

Binding time:
- When is a name resolved?

Binding Time

Increasing Flexibility
Increasing Efficiency
Binding Time

Keywords, e.g. "if"
Tool vendor chooses predefined symbols and std. library impl.
Programmer chooses names, data structures, and algorithms.
Compiler determines exact memory layout, code order, etc.
Resolve names in static library, e.g., "printf" in the C library.
OS loads dynamic libraries, e.g., Windows loads DLLs, Unix SOs.
Name resolved based on input.
E.g., plugins, interfaces.

Increasing Flexibility:

Increasing Efficiency:

Called static or early binding.

Called dynamic or late binding.
Object Lifetime vs. Binding Lifetime

Lifetime
- Entity: “alive” if memory is allocated (and initialized).

Name initially unbound: no binding exists.

B is allocated and bound to Name
Object Lifetime vs. Binding Lifetime

Lifetime
- Entity: “alive” if memory is allocated (and initialized).

Entity A

Entity B

Name

New binding: Name refers to B again.

A is allocated and bound to Name. It still exists!

A continues to exist for some time until it is deallocated.
Object Lifetime vs. Binding Lifetime

- Entity: “alive” if memory is allocated (and initialized).

Lifetime
- Entity: “alive” if memory is allocated (and initialized).

The binding from Name to B is shadowed by a new binding to A.

Object Lifetimes

- **Static** objects, which have a fixed address and are not deallocated before program termination.
- **Stack** objects, which are allocated and deallocated in a Last-In First-Out (LIFO) order.
- **Heap** objects, which are allocated and deallocated at arbitrary times.

Static Allocation

Some memory is required throughout program execution.
- **Multi-byte constants.**
- Strings (“hello world”).
- Lookup tables.
- **Global variables.**
- The program code.

Must be allocated before program execution starts.
- Requirements specified in program file.
- Allocated by OS as part of program loading.
- The size of static allocation is constant between runs.

Caution: this is not the same as Java’s static.
Static Allocation

Some memory is required throughout program execution.

- Multi-byte constants
- Strings ("hello world")
- Lookup tables
- Global variables
- The program code

Must be allocated before program execution starts.
- Requirements specified in program file.
- Allocated by OS as part of program loading.
- The size of static allocation is constant between runs.

Some memory is required throughout program execution.

- Multi-byte constants
- Strings ("hello world")
- Lookup tables
- Global variables
- The program code

Must be allocated before program execution starts.
- Requirements specified in program file.
- Allocated by OS as part of program loading.
- The size of static allocation is constant between runs.

Some memory is required throughout program execution.

- Multi-byte constants
- Strings ("hello world")
- Lookup tables
- Global variables
- The program code

Must be allocated before program execution starts.
- Requirements specified in program file.
- Allocated by OS as part of program loading.
- The size of static allocation is constant between runs.
Static Allocation

Some memory is required throughout program execution.
- **Compile-time constants.**
  Value must be known at compile time.
  - Elaboration-time constants.
    Value computed at runtime; compiler disallows subsequent updates.

Must be allocated before program execution:
- Requirements specified in program file.
- Allocated by OS as part of program load.
- The size of static allocation is constant between runs.

**Caution:** this is not the same as Java's `static`.

Advantages & Disadvantages

**Advantages.**
- No allocation/deallocation runtime overhead.
- Static addresses.
- Compiler can optimize accesses.

**Limitations.**
- Allocation **size fixed**; cannot depend on input.
- Wasteful; memory is always allocated.
- Global variables are error-prone.

Advice: avoid global variables.

Runtime Stack

Hardware-supported allocation area.
- Essential for subprogram calls.
- Grows **top-down** in many architectures.
- **Size limit:** stack overflow if available space is exhausted.
- Max. size of stack can be **adjusted at runtime**.
- OS is often involved in stack management.
Subroutines & Static Allocation

Calling a function/method/subroutine requires memory.

- Arguments.
- Local variables.
- Return address (where to continue execution after subroutine completes).
- Some bookkeeping information.
  - E.g., to support exceptions and call stack traces.

Where should this memory be allocated?

Static Allocation of Subroutine Memory

One approach: statically allocate memory for each subroutine.
(e.g. early versions of Fortran)

Problem: Waste
Most of the allocations will be unused most of the time.
One approach: statically allocate memory for each subroutine.
(e.g. early versions of Fortran)

**Problem: Waste**
Most of the allocations will be unused most of the time.

**Problem: No Recursion**
Subroutines may not be called while their memory is already in use.

Limited recursion depth can be allowed by allocating memory for multiple subroutine instances. But this increases waste...

Subroutine memory is allocated on-demand from the runtime stack.
Stack Frames

On a subroutine call:
- New stack frame pushed onto stack.
- Stack frames differ in size (depending on local variables).
- Recursion only limited by total size of stack.
- Reduced waste: unused subroutines only consume memory for code, not variables.
- Stack frame popped from stack on return.

Calling Sequence

Compilers generate code to manage the runtime stack.
- Setup, before call to subroutine.
- Prologue, before subroutine body executes.
- Epilogue, after subroutine body completes (the return).
- Cleanup, right after subroutine call.

```java
private void checkForKleeneclosure(NFA fo) throws IOException {
    if (readNextChar() == ' ')
        throw new IOException();
    if (readNextChar() == '')
        throw new IOException();
    else
        unreadLastChar();
}
```

PL implementation: caller does setup and teardown, callee does prologue and epilogue

PL implementation: caller does setup and teardown, callee does prologue and epilogue
Calling Sequence Example

```c
int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
}
```

```c
int main(int argc, char** argv)
{
    int sum = add(10, 20);
    printf("sum = %d\n", sum);
    return 0;
}
```

```
C program
```

```
Call of add() from main().
```

```
C program, x86-64 assembly
```

```
Call of add() from main().
```

```
Call of add() from main().
```

```
Call of add() from main().
```

```
Call of add() from main().
```

```
Call of add() from main().
```
Calling Sequence

C program, x86-64 assembly

Prologue
push stack & load arguments

int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
}

int main(int argc, char** argv)
{
    int sum = add(10, 20);
    printf("sum = %d\n", sum);
    return 0;
}

Epilogue
setup return value & restore stack

C program, x86-64 assembly

Prologue
push stack & load arguments

int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
}

int main(int argc, char** argv)
{
    int sum = add(10, 20);
    printf("sum = %d\n", sum);
    return 0;
}

Epilogue
setup return value & restore stack

C program, x86-64 assembly

Call Setup
prepare arguments

int add(int a, int b)
{
    int c;
    c = a + b;
    return c;
}

int main(int argc, char** argv)
{
    int sum = add(10, 20);
    printf("sum = %d\n", sum);
    return 0;
}

Epilogue
setup return value & restore stack

C program, x86-64 assembly

Epilogue
setup return value & restore stack

C program, x86-64 assembly
**Calling Sequence**

- **Prologue**
  - push stack & load arguments

- **Call Setup**
  - prepare arguments

- **Cleanup**
  - save return value

- **Epilogue**
  - setup return value & restore stack

---

**Stack Trace**

Exception in thread "main" NotYetImplementedException:
at NFA.<init>(NFA.java:25)
at NFA.<init>(NFA.java:18)
at Show.showNFA(Show.java:68)
at Show.main(Show.java:23)

---

**Advantages & Disadvantages**

**Advantages.**
- Negligible (in most cases) runtime overhead.
- Efficient use of space.
- Recursion possible.
- Offset of local variable within frame usually constant.

**Limitations.**
- Stack space is a limited resource.
- Stack frame size fixed (in many languages).
- Some offset computations required at runtime.
- Object lifetime limited to one subroutine invocation.

Advice: use stack allocation when possible. (except for large buffers)
The Heap

Arbitrary object lifetimes.
- Allocation and deallocation at any point in time.
- Can persists after subroutine completion.
- Very flexible: required for dynamic allocations.
- Most expensive to manage.

Memory Management

Allocation.
- Often explicit.
  - C++: new
  - Compiler can generate implicit calls to allocator.
    - E.g., Prolog.

Deallocation.
- Often explicit.
  - C++: delete
  - Sometimes done automatically.

Increasing Virtual Addresses
Memory Management

Allocation:
- Often explicit.
  - C++: `new`
  - Compiler can generate implicit calls to `allocator`.
  - E.g., Prolog

Deallocation:
- Often explicit.
  - C++: `delete`
  - Sometimes done automatically.

Allocation Problem:
Given a size $S$, find a contiguous region of unallocated memory of length at least $S$.
(and be very, very quick about it)

Common Techniques

Allocator implementation.
- Variable size.
  - List traversals, slow coalescing when deallocated.

- Fixed-size blocks.
  - $2^n$ or Fibonacci sequence.
  - "Buddy allocator," "slab allocator"
    - Memory blocks are split until desired block size is reached.
    - Quick coalescing: on free block is merged with its "buddy."

In practice.
- Most modern OSs use fixed-size blocks.
- Allocator performance crucial to many workloads.
- Allocators for multicore systems are still being researched.

Internal Fragmentation

Negative impact of fixed-size blocks.
- Block-size usually too large.
- Some memory is wasted.
Internal Fragmentation

Negative impact of fixed-size blocks.
• Block-size usually too large.
• Some memory is wasted.

Allocated...

...but partially unused.

External Fragmentation

Non-contiguous free memory.
• In total, there is sufficient available space...
• ...but there is none of the free blocks is large enough by itself.

new

new
External Fragmentation

Non-contiguous free memory.
- In total, there is **sufficient available space**...
- ...but there is none of the free blocks is large enough by itself.

Compacting the Heap

Merge free space.
- Copy existing allocations & update all references.
**“Dangling” References**

Binding / object lifetime mismatch.
- Binding exists longer than object.
- Object de-allocated too early; access now illegal.
- "Use-after-free bug" (free is the C deallocation routine)
- "Dangling" pointer or reference.

**Memory “Leaks”**

Omitted deallocation.
- Objects that "live forever."
- Even if no longer required.
  - Possibly no longer referenced.
- Waste memory; can bring system down.
- A problem in virtually every non-trivial project.
Memory "Leaks"

- Omitted deallocation.
- Objects that "live forever" and waste space until program termination.
- Even if no longer required.
- Possibly no longer referenced.
- Waste memory; can bring system down.

Garbage Collection

- Manual deallocation is error-prone.
  - "Dangling references."
  - "Use after free."
  - Possibly unnecessarily conservative.
  - "Memory leaks."
- Garbage collection.
  - Automatically deallocates objects when it is safe to do so.
  - Automated heap management; programmer can focus on solving real problem.
  - We will focus on garbage collection techniques later in the semester.

Summary & Advise

- Static Allocation
  - Not dynamically sizable; lifetime spans virtually whole program execution; use only sparingly.
- Stack Allocation
  - Lifetime restricted to subroutine invocation; allocation and deallocation is cheap; use whenever possible.
- Heap Allocation
  - Arbitrary lifetimes; use garbage collection whenever possible; use for large buffers and long-living objects.