Efficiency, Security and Portability

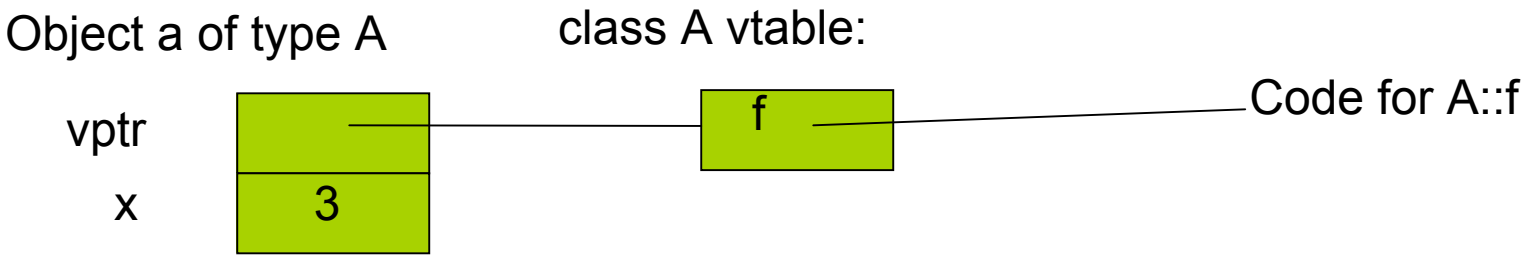
A comparative study of C++ and Java

Layout Of Class Objects

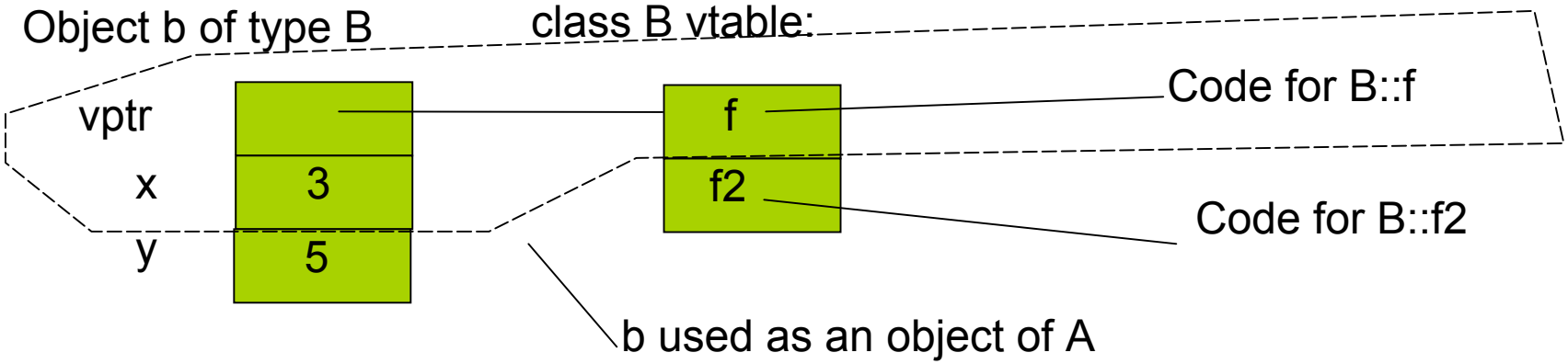
- OO abstractions are compound data types, so they need to be stored in memory
- Needs to extend layout of C structs with
  - Table to support dynamic lookup of methods
    - Collect all virtual methods into a single table
    - Offset of each method known at compile time
  - Common layout for base and derived classes
    - Members of base class is a subset of derived class members
    - Derived class layout contains a view of the base class layout
    - Dynamically change object views to support subtyping
- Problem: need to dynamically extend views of both methods and data members
- Solution: separately store the method table and data members; store base class members first

Object Layout and Single Inheritance

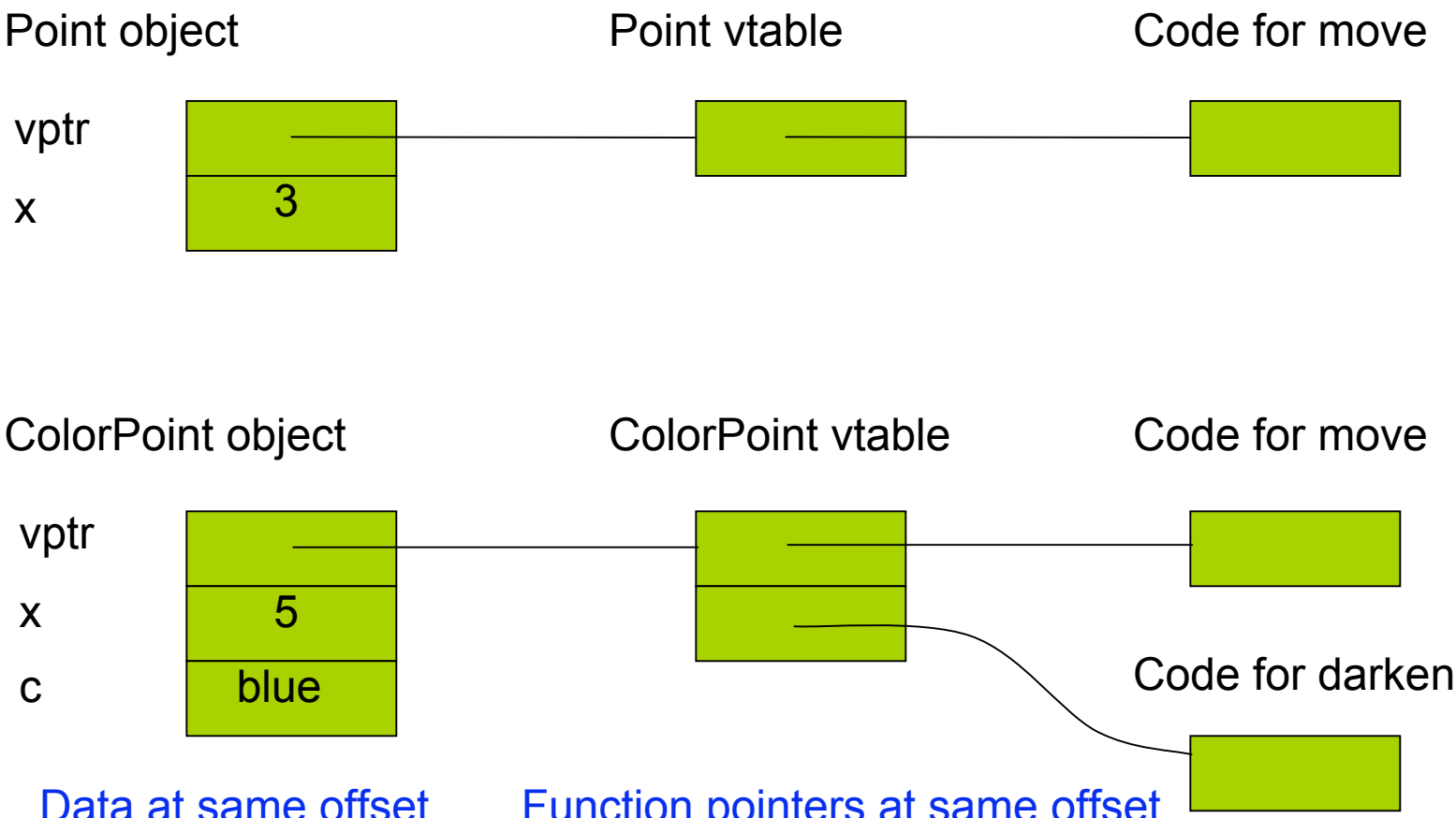
```
class A { int x;     public:  virtual int f() { return x;}  };
```




```
class B : public A  { int y;     public: virtual int f() { return y; }
                                         virtual void f2() { … }    };
```



Looking Up Methods

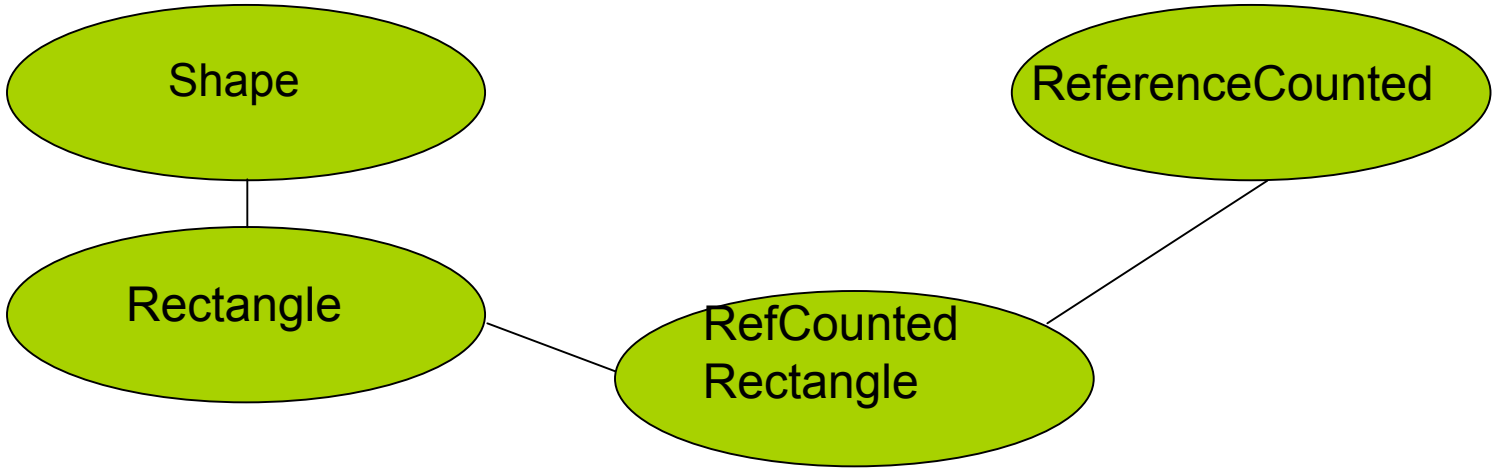



```
Point p = new Pt(3);
p->move(2);          // (*(p->vptr[0]))(p,2)
```

Dynamic Lookup Of Methods In C++

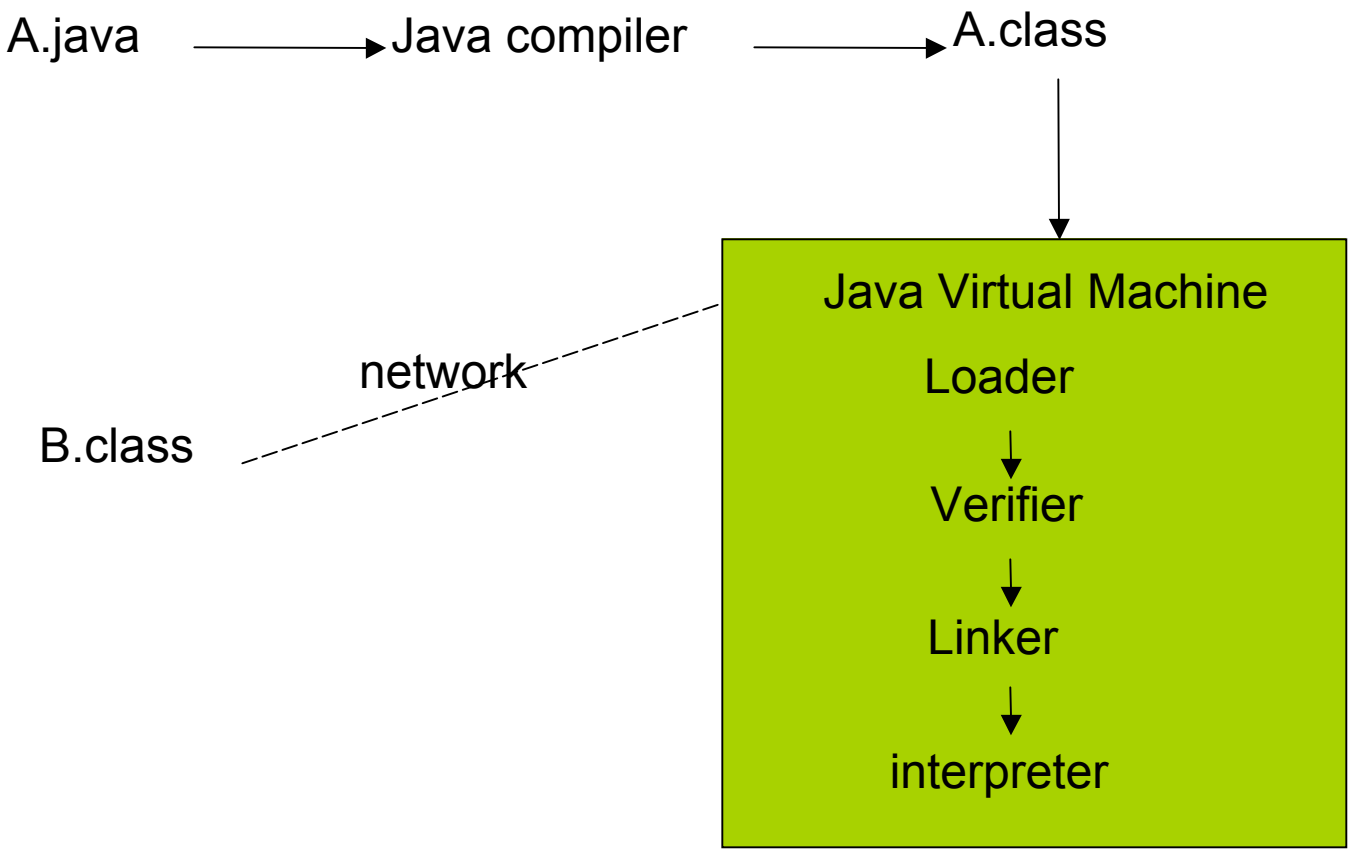
- C++ compiler knows all the base classes
  - Offset of data and function pointer are same in subclass and base class
  - Offset of data and function pointer known at compile time
  - Code p->move(x) compiles to equivalent of (*(p->vptr[move_offset]))(p,x)

Multiple Inheritance



- Inherit independent functionality from independent classes
- Members from different base classes are lined up one after another
  - Views of all base classes followed by members of derived class
  - Type casting may result in change of object start address
  - Each virtual method impl must remember starting address of its class
- C++: support multiple inheritance
- Java: single inheritance only, but can support multiple interfaces
  - Interfaces do not have data members

Java Virtual Machine



- Compiler and virtual machine
  - Compiler produces bytecode

Java Class Loader & Verifier

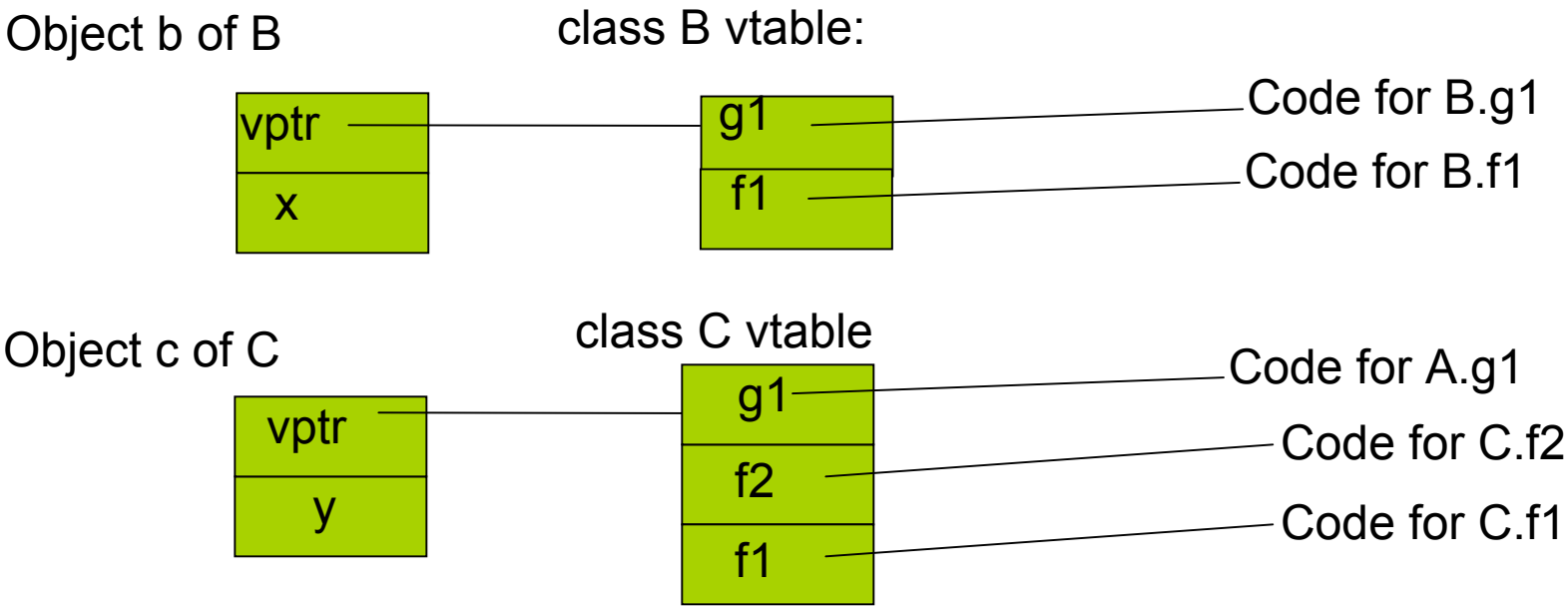
- Runtime system loads classes as needed
  - When class is referenced, loader searches for file of compiled bytecode instructions
- Default loading mechanism can be replaced
  - Can extend the default ClassLoader class
  - Can obtain bytecode from alternate source
  - Bytecode may not come from standard compiler
    - Evil hacker may write dangerous bytecode
- Verifier checks correctness of bytecode
  - Every instruction must have a valid operation code
  - Every branch instruction must branch to the start of some other instruction, not middle of instruction
  - Every method must have a structurally correct signature
  - Every instruction obeys the Java type discipline

JVM Dynamic Linking

- Java programs are compiled into bytecode
  - Each class has a table containing all dynamic methods
  - Every bytecode file has a **constant pool** containing information for all symbolic names
- Dynamic linking: add compiled class or interface
  - Create and initialize static fields
  - Checks symbolic names and replaces with direct references as they are used in instructions
    - Instruction includes index into constant pool
    - Constant pool stores symbolic names
    - Store once, instead of each instruction, to save space
  - First execution of instruction
    - Use symbolic name to find field or method in constant pool
    - Rewrite bytecode to remember method location
  - Second execution
    - Use modified "quick" instruction to simplify search

Java Object Layout and Interface

```
interface I { int f1(); }
interface J { int f2(); }
class A { public int g1() {…} }
class B  implements I extends A  { int x; int g1() {…} int f1() {…} }
class C  implements J implements I extends A { int y; int f2() {…} int f1() {…} }
```



When b and c are used as objects of I, f1 occupies different vtable entries
When b and c are used as objects of A, g1 always occupies the same vtable entry

Method Lookup in Java

- Dynamic method invocation
  - Look at constant pool for specification of methods
  - Find the real class of the object operand
    - must implement the interface or inherit from the base class
  - Find the class method table
    - Which maps methods to their offsets in vtable
  - Find the location of method in class's method table
    - Find the method with the given name and signature
    - Dynamic linking => may not be the same at compilation
  - Rewrite bytecode to remember method location
    - If object has class type, location is same for all objects
    - If object has interface type, location is unknown
      - Cache both the location and class table, check before proceed
- Call the method with new activation record, etc.

Java Type Safety And Security

- Run-time type checking
  - All casts are checked to make sure type safe
  - All array references are checked for out-of-bound access
  - References are tested for null-pointers
- Type safety
  - Automatic garbage collection
  - No pointer arithmetic
  - If program accesses memory, that memory is allocated to the program and declared with correct type
- Security Manager: keep track of privileges of classes
  - Separate class loaders for different web sites
  - Different name spaces for classes from different loaders
  - Throws securityException if security is violated

Design objectives

- C++: focus on efficiency
  - Add OO features to C without compromising efficiency
    - C Philosophy: give programmers every control
    - Backward compatible with C
  - Design principle: if you do not use a feature, you should not pay for it
- Java: Portability, Simplicity, and safety
  - Programs transmitted over the internet
    - Flexibility: dynamic linking, concurrent execution
    - Independent of native machines
  - Internet users must be protected from program errors and malicious programming
    - Bytecode interpreted instead of compiled
    - Type safety through runtime verification

Comparing Java with C++

- Interpreted + Portability + Safety - Efficiency
  - Compiled to byte code: a binary form with type information
- Dynamically linked + Portability + Flexibility - Efficiency
- Pure object-oriented + Simplicity - Efficiency
  - Almost everything is an object, does not allow global functions
- Objects accessed by ptr: + Simplicity - Efficiency
  - No problems with direct manipulation of objects
- Type safe + Safety +Simplicity - Efficiency
  - Arrays are bounds checked; no pointer arithmetics; no unchecked type casts
  - Garbage collected
- Built-in concurrency support + Portability
  - Used for concurrent garbage collection
  - Part of network support: download data while executing

Encapsulation

- Access control
  - Private: internal data representation
  - Protected: representations shared by derived classes
  - Public: interface to the outside
- C++ friend classes and functions
  - If A is a friend of class B, A can access all members of B
    - Non-symmetric: A is a friend of B ≠ B is a friend of A
    - Example: everything in B is private, but A is B's friend so B is part of A's internal representation
  - Circumvent access control based on OO inheritance
    - A class must know all of its friends
- Java packages
  - Another level of encapsulation
  - Members without access modifier have package visibility
  - Separate local classes from remote classes from the internet
    - A class does not need to know who will be in the same package

15

Method Binding

- Static methods
  - OO global functions in a name space
  - Supported by both C++ and Java
- Dynamic methods
  - Methods that are dynamically looked up at runtime
  - C++: virtual functions
  - Java: all non-static member functions
- C++ non-virtual methods
  - Can be treated as global functions with an extra parameter
  - Implementation more efficient than dynamic methods
- Java final methods
  - Cannot be redefined in derived classes
  - Implementation can be optimized
- Dynamic vs. non-dynamic methods
  - Flexibility vs. efficiency

Polymorphism

- Ad-hoc polymorphism: resolved at compile-time
  - Supported by both C++ and Java
    - C++: allow overloading both operators and functions
    - Java: disallow overloading of operators
- Subtype polymorphism
  - Multiple inheritance: supported in C++, not Java
  - C++: allow static casting from basetype to subtype
  - Java: runtime check required from basetype to subtype
- Parametric polymorphism
  - C++ templates: type-checked at link-time
  - Java generics: based on dynamic casting
- Inheritance of implementation only
  - Supported in C++, not Java

C++ and Java Design Decisions

- Encapsulation: data and function abstractions
  - C++: private, public, protected, friend
  - Java: private, public, protected, package
- Dynamic binding of functions
  - C++: virtual, non-virtual and static functions
  - Java: final, non-final and static functions
- polymorphism
  - C++: operator overload, subtype inheritance, templates
  - Java: operator overload, interface and public inheritance, generics
- Inheritance and mutation
  - C++: public and protected/private inheritance. Multiple inheritance
  - Java: public inheritance only. Multiple inheritance for interface only
- Memory Management
  - C++: objects on stack or in heap; free memory with destructors
  - Java: objects in heap only. Garbage collection

History Of C++ And Java

- C++
  - Designed by Bjarne Stroustrup at Bell Labs for research on simulation
  - Object-oriented extension of C based primarily on Simula
  - Popularity increased in late 1980's and early 1990's
  - Features were added incrementally
    - Classes, templates, exceptions, multiple inheritance, ...
- Java
  - Designed by James Gosling et. al at Sun, 1990–95 for "set-top box", small networked device with television display
    - Graphics
    - Communication between local program and remote site
    - Developers don't have to deal with crashes, etc.
  - Internet application
    - Simple language for programs transmitted over network