What is an “object”?

*Objects* are units of data with the following properties:

- **typed and self-contained**
  
  Each object is an instance of a *type* that defines a set of *methods* (signatures) that can be invoked to operate on the object.

- **encapsulated**
  
  The only way to operate on an object is through its methods; the internal representation/implementation is hidden from view.

- **dynamically allocated/destroyed**
  
  Objects are created as needed and destroyed when no longer needed, i.e., they exist outside of any program scope.

- **uniquely referenced**
  
  Each object is uniquely identified during its existence by a name/OID/reference/pointer that can be held/passed/stored/shared.
Why are objects useful for systems?

The properties of objects make them useful as a basis for defining persistence, protection, and distribution.

- **Objects are self-contained and independent.**
  
  Objects are a useful granularity for persistence, caching, location, replication, and/or access control.

- **Objects are self-describing.**
  
  Object methods are dynamically bound, so programs can import and operate on objects found in shared or persistent storage.

- **Objects are abstract and encapsulated.**
  
  It is easy to control object access by verifying that all clients invoke the object’s methods through a legal reference.

  Invocation is syntactically and semantically independent of an object’s location or implementation.
Tricks With Objects (I)

1. Extend the object name space outside of a process and across a distributed system.
   - Linked data structures can be partitioned across the nodes and traversed with *location-independent invocation*.
     Emerald, Guide

2. Extend the object name space across secondary storage.
   - Objects (and their references) may live longer than processes; fault objects into memory as they are referenced.
     POMS and other persistent object stores and OODBs
   - Eliminate “impedance mismatch” between memory/disk.
     type-checked secondary storage with type evolution
Tricks With Objects (II)

3. Define RPC services as objects.
   - Allows persistent, location-independent name space with dynamic binding and/or dynamic activation.
     Argus, Eden, Clouds, Arjuna
   - Encapsulate with a clean object wrapper for external access.

4. Make object references unforgeable and reject invocation attempts with invalid references.
   - An unforgeable object reference is called a *capability*.
     Cambridge CAP, IBM System/38 and AS/400, Intel 432
     CMU Hydra and Mach, Stanford V, Amoeba, Eden
   - Use as a basis for protected sharing/interaction/extension.
Some Other Aspects of Object Models

1. Objects may be active or passive.
   An active object contains its own thread(s); typically incoming invocations are queued and serviced by these threads.
   Passive objects sit there and wait to be invoked; the invoking thread enters the object for the duration of the call.

2. An object’s mapping to the underlying OS or machine features is often expressed in terms of granularity.
   A coarse-grained object is equivalent to a process or address space invoked with messages or cross-domain calls.
   A medium-grained object lives with others within a process and is protected by its addressing wrapper.
   A fine-grained object is a heap-allocated block of memory.
The Trouble with Objects

Why were these OO systems seen to have failed by the U.S. systems research community?

• Many sacrificed performance for elegance.
  “Performance is paramount” is (was?) an accepted axiom.
• Many depended on (slow and/or obscure) OO languages at a time when C was dominant in systems.
  OO concepts had not yet penetrated the culture.
• Those that were not integrated with OO languages could not benefit fully from the elegance of the model.
  nonuniform view of “system objects” and “language objects”
• Few adherents were able to communicate the relevance of OO systems to real application needs.
Emerald

Emerald is a classic and influential distributed object system.

- Distribution is fully integrated into the language, its implementation, and even its type model.
  This is a strength and a weakness: combines language issues and system issues that should be separated.
- Objects can be freely moved around the network
  Programmers see a uniform view of local and remote objects.
  Moving objects “take their code and threads with them”.
- Local invocation is fast; remote invocation is transparent.
  supports pass-by-reference for RPC
Understanding Emerald

1. Emerald was marketed to OS researchers as a lightweight alternative to *process migration* (a hot topic at the time).

   Process migration was accepted as a means to balance load, handle failures, or initiate a remote activity.

2. Emerald eliminated key problems with process migration.

   OS-dependent state associated with migrating processes
   
   high cost of interaction among colocated processes

3. Emerald was seen as a sort of lightweight “operating system” as well as a language.

   The “kernel” is a runtime library in a Unix process (one per node) within which all Emerald programs run.

   The Emerald “kernel” had its own support for “processes”, which we would now call “threads”, and execution...protection...persistence.
Issues for Emerald

1. How to implement object references so that they are location-independent?
   How to ensure uniqueness of object IDs?
   How to locate remote objects, e.g., if they have moved?

2. What is the “hook” for transparent location-independent invocation?
   How to make it fast if the invoked object is local?

3. How to migrate and dynamically import code and threads?

4. What are the semantics of argument passing?

5. Who’s going to implement distributed garbage collection?
Uniform Mobility: an Example

**Step 1:** a thread invokes a purple object on node A, which recursively invokes a blue object on the same node.

**Step 2:** the blue object moves to node B concurrently with the invocation.

How to preserve inter-object pointers across migration?
How to keep threads “sticky” with migrating objects?
How to maintain references in stack activation records?
How to maintain linkages among activation records?
What about virtual addresses in CPU registers?
Object References in Emerald

Emerald represents inter-object references as pointers into an *object descriptor* in an *object table* hashed by a unique *object identifier* (OID).

The object table has a descriptor for every resident object, and for every remote object referenced by a resident object, and then some.

When an object moves, its containing references must be found (using its template) and updated to point to descriptors on the destination node.

References to the moving object need not be updated because they indirect through the object table.
Uniform Mobility Example, Continued

Step 3: the purple object moves to node C before the invocation returns.

What to do with the thread’s activation record for the purple object?
- cost of context switch

How to find the purple object to return into its activation record?

How to keep forwarding pointers up to date? (eager vs. lazy)
- iterative lookup
  - piggyback on passed references and remote returns

Why are timestamps needed on the forwarding pointers?
Are serial numbers a sufficient form of timestamp?
The Relevance of Emerald

Emerald defines a conceptual basis for understanding today’s distributed object systems.

CORBA, RMI, EJB, DCOM

Emerald showed what is possible from a distributed object environment in its purest form.

1. Uniform view of local/remote objects: orthogonality of location.
   referencing, invocation/return
   garbage collection

2. Uniform object model is compatible with (local) performance.
   extended features impose a cost only when used

3. Location of mobile objects by reference hints and forwarding.
Distributed Objects in the Real World (I)

The purity of Emerald flows from a common language, architecture, and security domain.

1. Can we use distributed objects as a basis for interoperability among software modules written in different languages?
   
   IDL converts distributed objects into a packaging/integration technology. 
   
   What about type checking? Garbage collection?

2. Can objects interact across systems with different data formats?

   *IOP and C/XDR define standard wire formats for transmitted data.

3. Can objects interact securely across mutually distrusting nodes and/or object infrastructures by different vendors?

   How are object references stored, transmitted, and validated?
Distributed Objects in the Real World (II)

Emerald has no provision for handling failures of any kind.

How can we find objects in the presence of node failures?

What should we do about activities that were pending in failed nodes/objects?

How can we recover object state after failures? How can we ensure that the recovered state is consistent?

Can we safely execute object invocations from nodes with intermittent connectivity?

What about long-term storage of objects, and invocation of stored objects that are not currently active?

persistence/uniqueness/stability of object IDs
Distributed Objects in the Marketplace

1. Remote Method Invocation (RMI)
   API and architecture for distributed Java objects

2. Microsoft *Component Object Model* (COM/DCOM)
   binary standard for distributed objects for Windows platforms
   e.g., clients generated with Visual Basic, servers in C++
extends OSF DCE standard for RPC

3. CORBA (Common Object Request Broker Architecture)
   OMG consortium formed in 1989
   multi-vendor, multi-language, multi-platform standard

4. Enterprise Java Beans (EJB) [1998]
   CORBA-compliant distributed objects for Java, built using RMI
EJBs represent a convergence of CORBA and Java/RMI.

• The OMG alliance and CORBA (1990) predate Java.
  
  Before Java, many (e.g., Sun and IBM) saw CORBA as the “last best hope” for cross-platform computing in a Windows world.

• RMI was designed for Java (1996) before complete CORBA standards emerged.
  
  RMI leverages the Java language to be “closer to Emerald”, with almost uniform objects and distributed garbage collection.

• CORBA and RMI serve similar goals, but both are needed.
  
  CORBA trades RMI’s purity for compatibility across languages.
  
  RMI is now billed as an API compatible with non-Java clients and servers “using any distributed object standard”.