Programming in Java

1. Introducing Java

Evolution of Software Development

- How we write code has gone through a number of eras
  - Each era is characterized by a specific approach to “How We Write Code”
- These eras are the result of several factors:
  - **Business drivers** – The types of tasks that the consumers (who pay for software) want it to do for them
  - **Available Technology** – The limitations and capabilities of existing hardware and infrastructure (like networks)
  - **Tools and techniques** – The theoretical and practical engineering methodologies and toolsets
- Main Eras
  - **1940s to 1950s** – *Hardware Rules* – Machine code and instruction sets
  - **1950s to 1970s** – *Code Cowboys* – Coding by instinct “Real programmers only use assembler”
  - **1970s to 1990s** – *Software Engineering* – Adaptation of engineering practices and high level language
  - **1990s to 2010s** – *Human Cyber Systems* – Massively networked systems, Internet and on-line users
  - **2010s to Present** – *Cyber Physical Systems* – Internet of Things, big data, massive scale architectures, AI

Programming Paradigms

- A programming paradigm is a set of:
  - Assumptions about what the components of a program should be
  - Techniques and principles for building programs
  - Assumptions about what sort of problems are solved best by that paradigm
  - Best practices for software design and code style
- There are multiple programming paradigms
  - Most have been around since the start of high level programming in the 1960s
- A paradigm becomes “main-stream” when a set of problems arise that the paradigm is better suited to solve than the paradigm currently in use
- The main-stream paradigms in use today – and supported in Java are:
  - Structured or procedural programming
  - Object Oriented Programming
  - Functional Programming

Structured Programming

- Code is structured into reusable modules
  - Called subroutines or functions or procedures
  - Standard libraries of procedure are part of the programming language
- Users can define their own procedures and libraries
  - Procedures are the highest level of organization
  - Implementation of DRY
- Many Java class libraries organized like this.
- Image is a FORTAN subroutine

```
C ***************************************************************
C
      SUBROUTINE TRISOL(A,B,C,D,H,N)
C
C ************** TRI-DIAGONAL MATRIX SOLVER ************************
C
C *** THIS TRIDIAGONAL MATRIX SOLVER USES THE THOMAS ALGORITHM ********
C
      dimension A(250),B(250),C(250),D(250),H(250),W(250),R(250),G(250)
      W(1)=A(1)
      G(1)=D(1)/W(1)
      do 100 I=2,N
      I1=I-1
      R(I1)=B(I1)/W(I1)
      W(I)=A(I)-C(I)*R(I1)
      G(I)=(D(I)-C(I)*G(I1))/W(I)
  100 continue
      H(N)=G(N)
      N1=N-1
      do 200 I=1,N1
      II=N-I
      H(II)=G(II)-R(II)*H(II+1)
  200 continue
      return
      end
```

OO Programming

- Procedures are methods within class definitions
  - There are no stand-alone procedures
  - Java methods tend to be where the imperative style of coding is mostly found
- Some Java classes allow for static methods
  - The class is used to define an API that works just like a library in a structured programming language
- OO programming is about where we write and store our reusable code
  - Specifically, in class definitions
  - Image is Simula67 code from the mid 1960s
  - From the official Simula reference material

```
Begin
   Class Glyph;
      Virtual: Procedure print Is Procedure print;;
   Begin
   End;

   Glyph Class Char (c);
      Character c;
   Begin
      Procedure print;
        OutChar(c);
   End;

   Glyph Class Line (elements);
      Ref (Glyph) Array elements;
   Begin
      Procedure print;
      Begin
         Integer i;
         For i:= 1 Step 1 Until UpperBound (elements, 1) Do
            elements (i).print;
         OutImage;
      End;
   End;

   Ref (Glyph) rg;
   Ref (Glyph) Array rgs (1 : 4);

   ! Main program;
   rgs (1):- New Char ('A');
   rgs (2):- New Char ('b');
   rgs (3):- New Char ('b');
   rgs (4):- New Char ('a');
   rg:- New Line (rgs);
   rg.print;
End;
```

Functional Programming

- Procedures are defined like mathematical functions
  - Functions act like data
  - Said to be first class citizens
- Programs are treated like function composition in math
  - LISP introduced functional programming in the 1950s – before FORTRAN
  - Top Listing shown is 1960s APL code for computing a matrix determinant
  - The code below assigns a function to a variable Avg that averages a vector of numbers
  - You can try APL interactively at
    - *https://tryapl.org/*

```
      ∇DET[⎕]∇
     ∇ Z←DET A;B;P;I
[1]    I←⎕IO
[2]    Z←1
[3]   L:P←(|A[;I])⍳⌈/|A[;I]
[4]    →(P=I)/LL
[5]    A[I,P;]←A[P,I;]
[6]    Z←-Z
[7]   LL:Z←Z×B←A[I;I]
[8]    →(0 1 ∨.=Z,1↑⍴A)/0
[9]    A←1 1 ↓A-(A[;I]÷B)∘.×A[I;]
[10]   →L
[11]  ⍝EVALUATES A DETERMINANT
     ∇
```

```
      Avg←{(+⌿⍵)÷≢⍵}
      Avg 1 6 3 4
3.5
      |
```

Modern Programming Languages

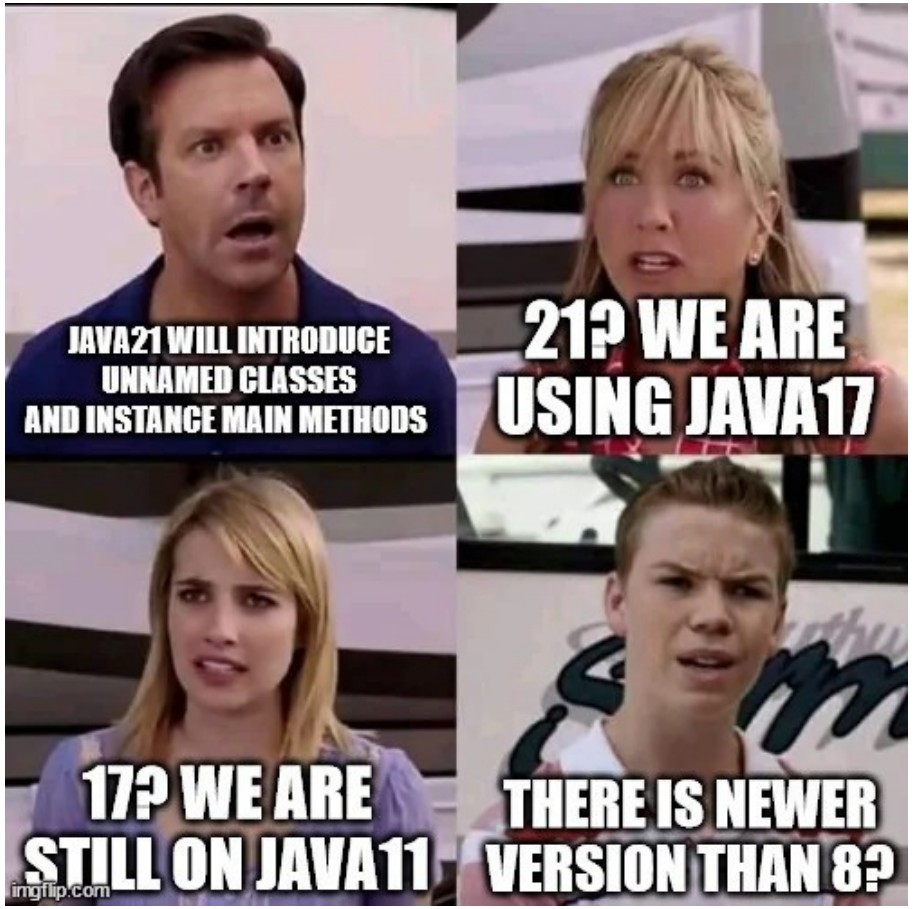
- Most modern programming languages support more than one paradigm
  - Modern languages like Rust, Go and Julia are designed to support multiple paradigms
- Legacy Languages are often revised to add support for a paradigm
  - COBOL added object oriented support
  - Java added support for functional programming in Java 8
- Why paradigms go mainstream
  - Most of the different paradigms have existed for over 50 years
  - They are designed to solve a particular class of problems
  - The types of problems industry deals with change over time
  - Changes often result from changes in technology and user requirements
  - Existing paradigms may not be able to solve these new problems
  - A different paradigm is main-streamed that can solve the problems

Paradigm Focus

- Structured programming
  - Developed to respond to the need to automate the business processes in the 1960s
  - COBOL = Common Business Oriented Language
  - A data set was read in, algorithms used to process it, the results written out
  - Typically done in batch mode on a mainframe
  - There is still a massive installed based of COBOL and other structured code running business operations in the public and private sector
- Object Oriented programming
  - Structured programming doesn’t do distributed computing and networks well
  - The rise of the Internet in the 1990s made this a requirements
  - OO languages, led by Java, had the right paradigm to do this sort of computing
- Functional programming
  - Ideal tool to handle streaming data at scale which became a need in the mid 2010s with the rise of big data
  - OO and structured programming don’t do this well

Re-inventing Java

- As new paradigms become mainstream Java incorporates features from those paradigms
  - What Java code looks like has changed over time as the language changes
- Java programmers often have to both
  - Write new Java code for modern architectures like microservices
  - Support legacy Java code written in earlier versions of Java

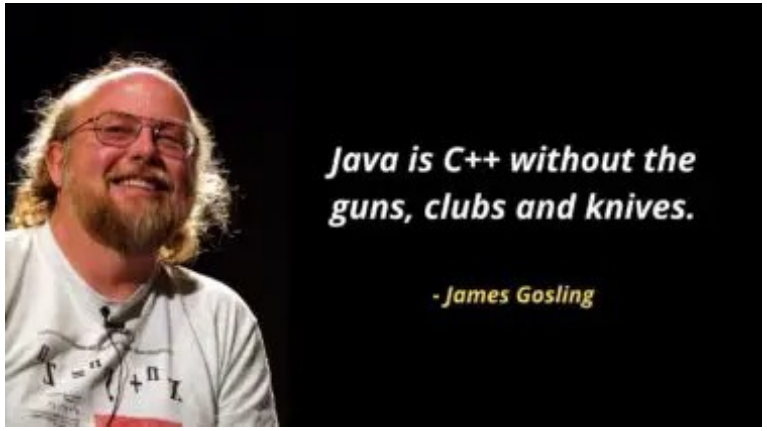


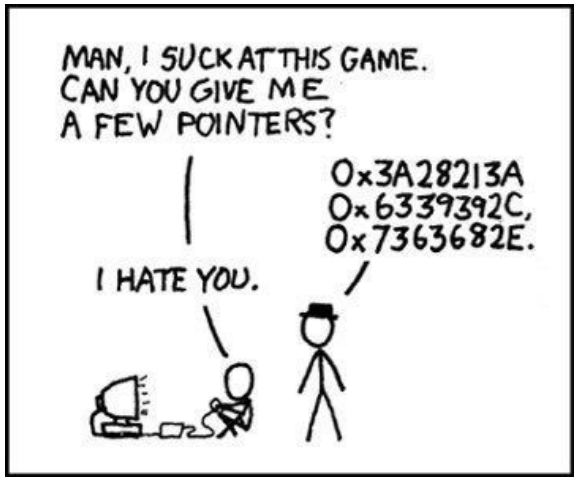
The Java Story

- Java was a project developed at SUN Microsystems to support interactive TV
  - Hardware wasn't powerful enough to do processing
  - Repurposed to do client side computing in browsers on the web
  - Up until that time, all computing took place on the servers
  - This created bandwidth issues and performance bottlenecks
- Java would run code called "Applets" in a special browser plugin or sandbox
  - Much like JavaScript does now
  - Principle designer was Canadian software engineer James Gosling

The Language Design

- 1990s – C++ and OO were very popular
- Gosling intended Java to be a better C++
  - Easy for C++ programmers to learn and adopt
- Threw away the non-OO features of C++
- “Fixed” the two most problematic features of C++
  - Direct access to memory pointers
  - Memory leaks
- Wanted portability
  - Write once, run anywhere
  - Designed around a fan-out VM architecture that had proven to be effective in other languages
- More rigorous error checking
  - For example, array bounds checking





Java Social Engineering

- Very concerned about keeping a Java Standard
  - *My impression is that a really, really high-order concern for the whole development community is interoperability and consistency. James Gosling*
  - Did not want to open source the language but still make it free
  - Did not want to be in the business of “building” Java compilers
- Published two specifications
  - The Java Language Specification that describes how Java works,
    - *Like other standard language specifications (C++ ISO standard for example)*
  - The Java Virtual Machine Specification that describes how the runtime environment (JRE) works
    - *JRE included the Java Virtual Machine and local native libraries*
- SUN protected their IP using very restrictive trademarking
  - Anyone could write and distribute their own Java development tools (JDK) and JRE
  - Provided it passed the Java Compliance Suit tests
  - If your product didn’t, you were not allowed to call it Java. Period. Or SUN would come after you

Fool Around and Find Out

- Microsoft developed a version of Java called J++ that was not compliant to the Java spec
  - SUN sued to enforce their trademark
  - SUN won and Microsoft had to drop J++
  - Although J++ later became the basis for .Net and C#

Sun, Microsoft settle Java suit

Sun Microsystems and Microsoft have settled their long-running lawsuit over Microsoft's use of Sun's Java software.

Stephen Shankland

March 15, 2002 5:10 a.m. PT

5 min read

Sun Microsystems and Microsoft have settled their long-running lawsuit over Microsoft's use of Sun's Java software.
Under the settlement, Microsoft will pay Sun $20 million and is permanently prohibited from using "Java compatible" trademarks on its products, according to Sun. Sun also gets to terminate the licensing agreement it signed with Microsoft.

Reference Implementations

- SUN also provided “reference implementations”
  - These were a JDK and JRE that demonstrated compliance to the specs and compliance test suite
  - These were not intended to be commercial products
- Many versions of JRE were developed for different platforms
  - IBM maintains versions of JRE to run on AIX, IBM Linux, z/OS and IBM i
- Java was originally free but proprietary
  - Eventually migrated into open source reference implementation
  - Java and the JVM are now open source
  - These are available at https://jdk.java.net/
  - These are supported by the openJDK project and Oracle
  - Also motivated by the fear that when Oracle bought SUN, they might implement expensive licensing for using Java
- Oracle has opted more for a commercial support and redistribution plan for enterprises

The JCP – Java Community Process

- SUN created the Java Community Process (JCP)
  - Open committee that was the only authority to issue Java specifications
  - Would decide the direction Java would take based on input from the user community
- It resulted re-inventing Java as an enterprise application delivery platform instead of remaining a JavaScript like browser automation engine
  - Resulted in J2EE (EE) which introduced server side Java, servlets, Java Server Pages and a lot more
- Responsible for introducing new features like generics, functional programming and other innovations
  - Done in response to the requests to the JCP
  - Also responsible for deprecating Java features that are no longer useful
- From their website
    - *The JCP is the mechanism for developing standard technical specifications for Java technology. Anyone can register for the site and participate in reviewing and providing feedback for the Java Specification Requests (JSRs), and anyone can sign up to become a JCP Member and then participate on the Expert Group of a JSR or even submit their own JSR Proposals.*

Demo

Introduction to IDEs and Hello World

Lab 1-1

Hello World and Lab Setup

Object Oriented Fundamentals

- OO is based on three principles that have been evolving since the 1950s
- Iconicity
  - The idea that our programs are automating something “out there”
  - Our programs should look like what they are automating
  - The units of automation should make sense to a domain expert
- Recursive Design
  - The systems that we automate tend to be structured in layers or hierarchies
  - Our design process should be scale independent
  - As we can design the sub-systems of a system in the same way we design the system itself
- The Object Model
  - Our experience tells us that we perceive that systems are made up of objects in the real world
  - Then the most effective way to design an object is to have a similar sort of construct in our code

The Principle of Iconicity

- First proposed by Professors Ole-Johan Dahl and Kristen Nygaard at the Nordic Centre for Computing at the University of Oslo in the late 1950s
- The posed the question
  - *“Why should people have to learn how to interact with a computer? Why can we not design a computer program that resembles what it automates so that people can interact with it based only on what they already know?”*
- And developed the principle of iconicity
  - *Systems should look like what the automate so domain experts can use the system without training*
- This led to several corollaries
  - The separation of interface and implementation is essential
  - Designs should be based on domain modeling

Simula

- Dahl and Nygaard implemented these ideas in the Simula programming language in 1961-5
- The Simula code tanspiled to Algol which was then compiled
- The code ran on a UNIVAC 1107 which had
  - 256K of memory
  - Tape drive storage
  - 6MB of disk space
  - Accessed via punch cards and a line printer

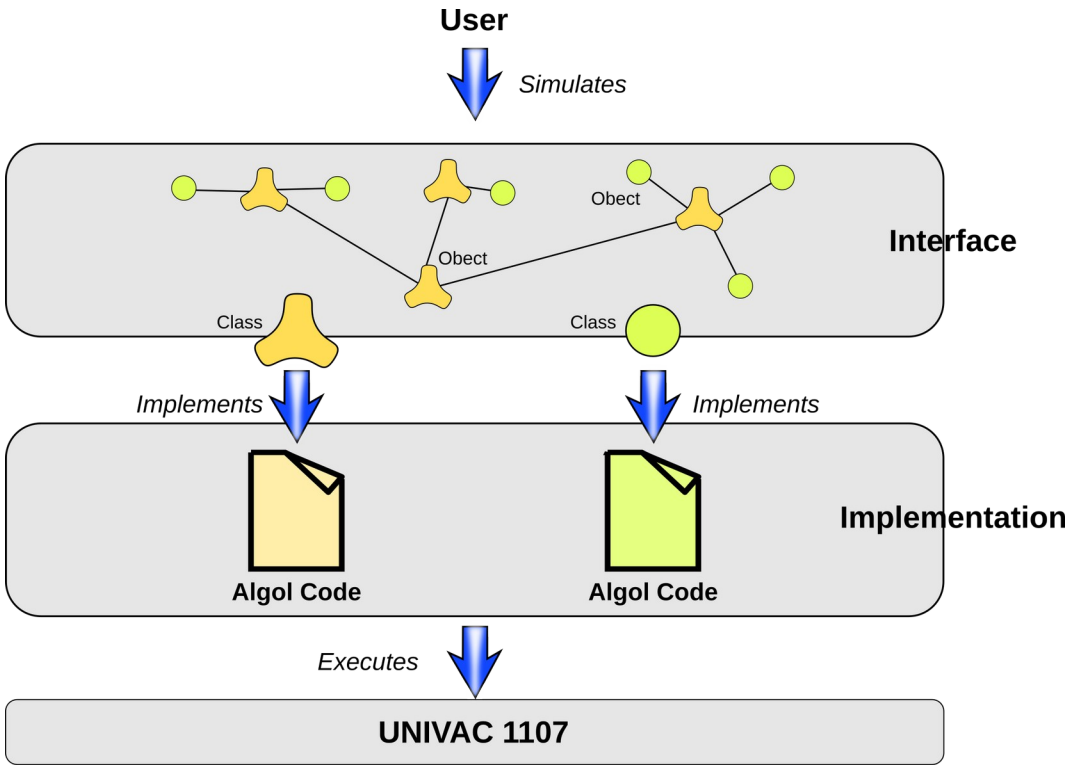
Simula Code

- The structure of a Java class definition is derived ultimately from the Simula definition
- This is an implementation of the object model
  - The class definition has a type, attributes and methods (called procedures)
  - We create objects by instantiating them from the class definition
- The code shows a class representing a geometrical point located at (X,Y)
  - ROTATED(P,N) Rotates this point about the point P by N degrees,
  - EQUALS(P) checks if this point and P are the same geometrical point,
  - DISTANCE(P) return the distance between this point and the point P.
  - Each time a point is created, R and THETA are calculated from the actual parameter values of X and Y.

```
CLASS POINT(X,Y); REAL X,Y;
    BEGIN REAL R, REAL THETA;
        REF(POINT) PROCEDURE ROTATED;
        BOOLEAN PROCEDURE EQUALS;
        REAL PROCEDURE DISTANCE;
        R := SQRT(X ↑ 2 + Y ↑ 2)
        THETA := ARCTAN2(X,Y)
    END
```

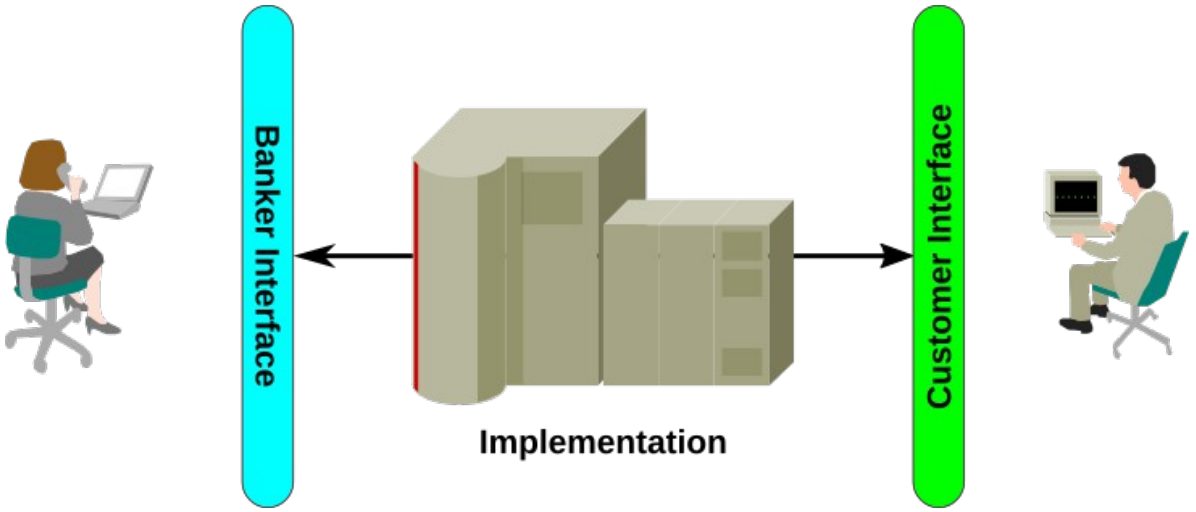
Interface and Implementation

- The class constructs in Simula provided an iconic interface that users could interact with
  - But the code could not be executed
- It had to be turned into executable code
  - This is the implementation
  - Implementation code is not iconic
- Keeping these two layers separate
  - Allows for different implementations
  - FORTRAN on and IBM/360 for example
  - The implementation is never iconic
  - It's a black box except to the developer
- This is called the decoupling of interface and implementation



Interface and Implementation

- This idea is fundamental to modern Java design
- Consider a bank application
  - There are two domains that conceptualize a bank account, and they are not the same
  - A banker's view of a bank account is not the same as the customer (bankers love debits and credits)
  - Neither view is the implementation
  - The bank account is a record in an SQL database
- There may be multiple interfaces
  - We can change interfaces without changing the implementation
  - We can change the implementation without changing the interfaces
  - The actual implementation is a black box to the banker and the customer (but not the DBA)



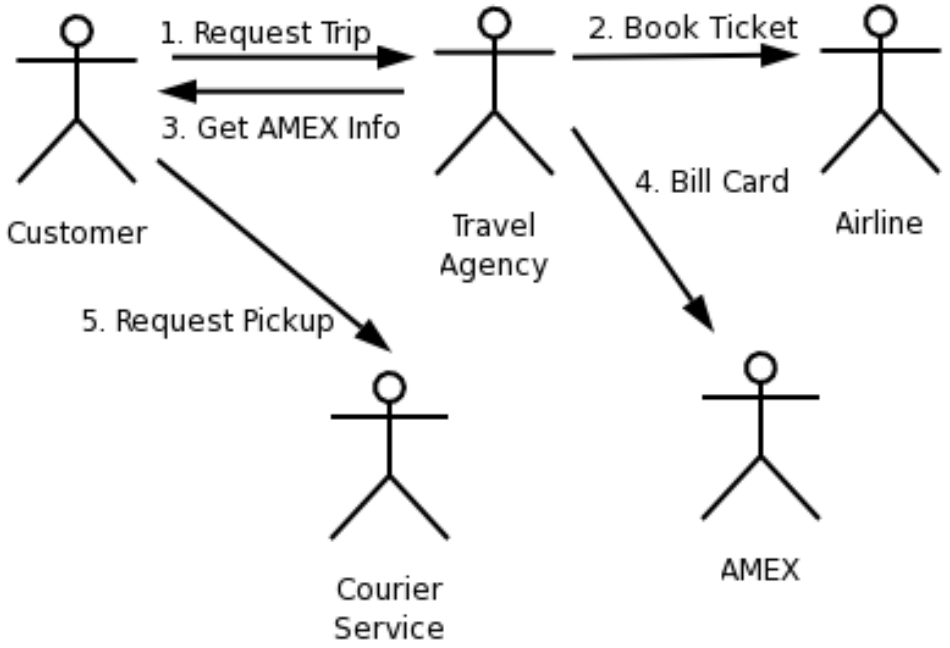
Interfaces as Views

- One useful way to think of interfaces is that they are views into the underlying implementation
  - A view shows only what is of interest to the viewer
  - A view may translate something in the implementation into a representation unique to that view
- Multiple interfaces provide different views into the same underlying implementation
  - In standard database development, these interfaces are called.. wait for it... views
- The challenge of developing interfaces that are both useful and realistic will be a topic we will touch on later in the course
    - *“We don't see things as they are, we see them as we are.” Anias Nin*
    - *“What you see is not reality but perspective, what you hear is not truth but opinion.” Anonymous*
- Fundamental rule of programming:
  - Don’t write code without a thorough analysis of the problem domain, or you will be writing code only to your view of the problem, which might different than the user’s views
    - *“First solve the problem, then write the code.” John Johnson*

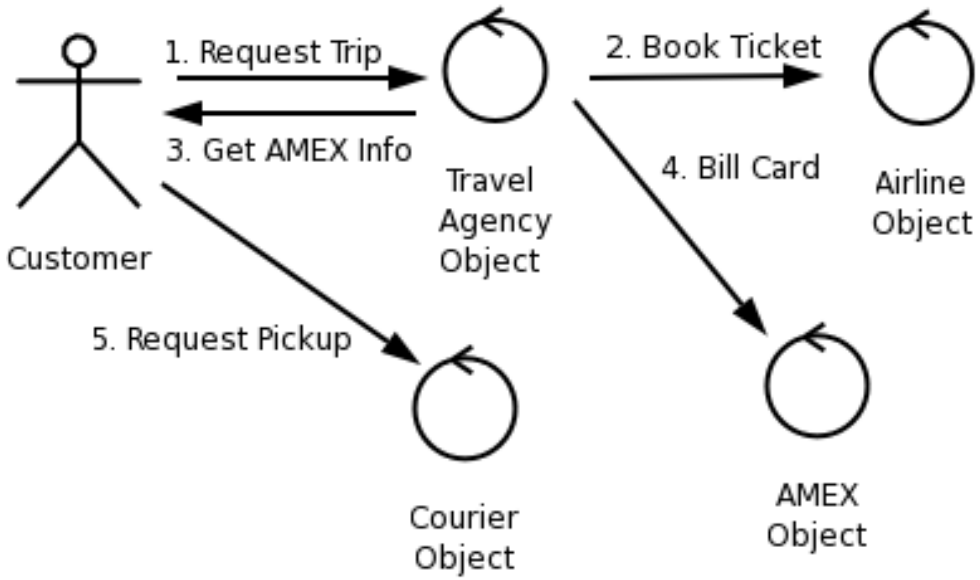
The Network Problem

- Structured programming does not have an easy solution to the problem of distributed agent systems which are characterized by specific properties:
  - They are distributed
    - *There is no central CPU or processing node*
    - *Each agent has its own processing capability.*
  - They are concurrent
    - *The processing done by each agent proceeds independently of the processing in any other agent.*
  - They collaborate
    - *Agents work together by collaborating to accomplish tasks*
    - *This collaboration takes place by the sending of messages back and forth.*
  - They are heterogeneous
  - The agents that make up a system do not all have to be of the same kind. As long as they can send and receive the appropriate messages, they can be of any type
- These sorts of systems describe many types of networks we may want to model
  - Human work teams, financial transactions between companies, biological processes, computer networks etc

The Network Problem

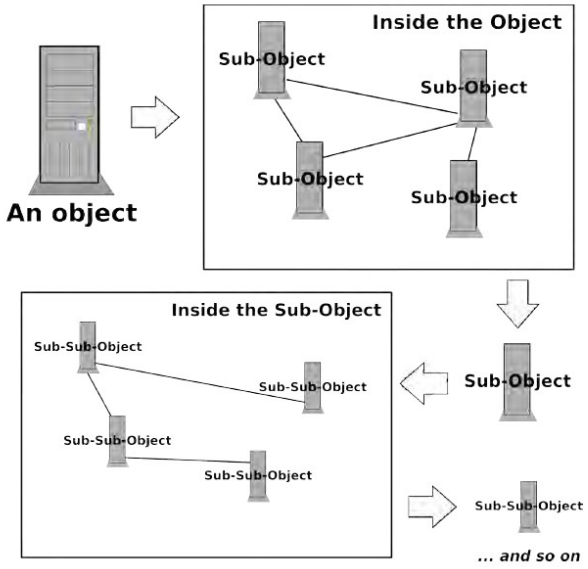


The manual system of people talking to each other – travel agents, airline representatives, courier dispatchers

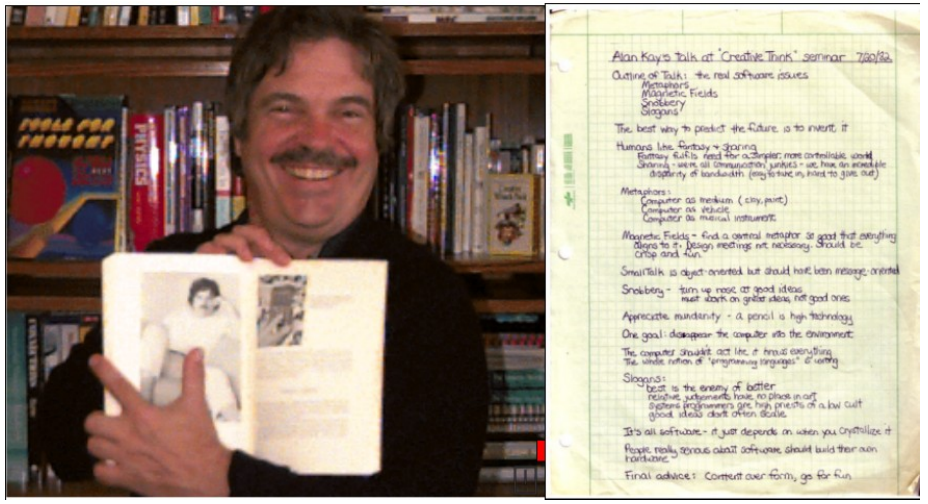


The system we want to build where the entities communicate directly instead of through human interfaces

Recursive Design



1. A system is built up in layers
2. Each layer is itself an object
3. Each layer is made up of a collection of peer objects which provide the functionality of that layer
4. There is no restriction to the types of objects that exist within a particular layer



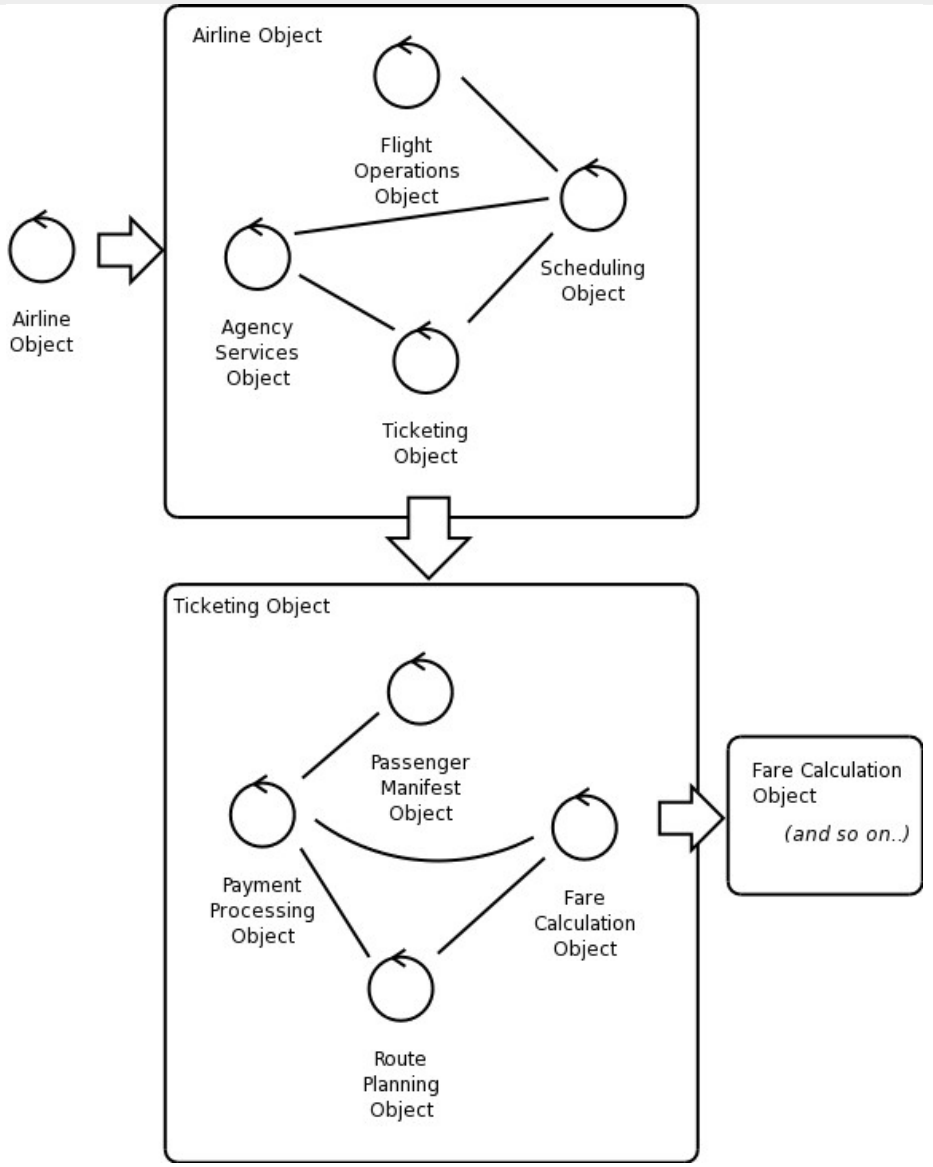
I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning – it took a while to see how to do messaging in a programming language efficiently enough to be useful)

Instead of dividing a system (or computer) up into functional subsystems, we think of it as a being made up of a collection of little processing engines, called objects. Each object will have similar computational power to the whole and processing happens when the objects work together. However, and this is the recursive part, each object can then be thought of in turn as a collection of sub-objects, each with similar computational power to the object, and so on.

Alan Kay

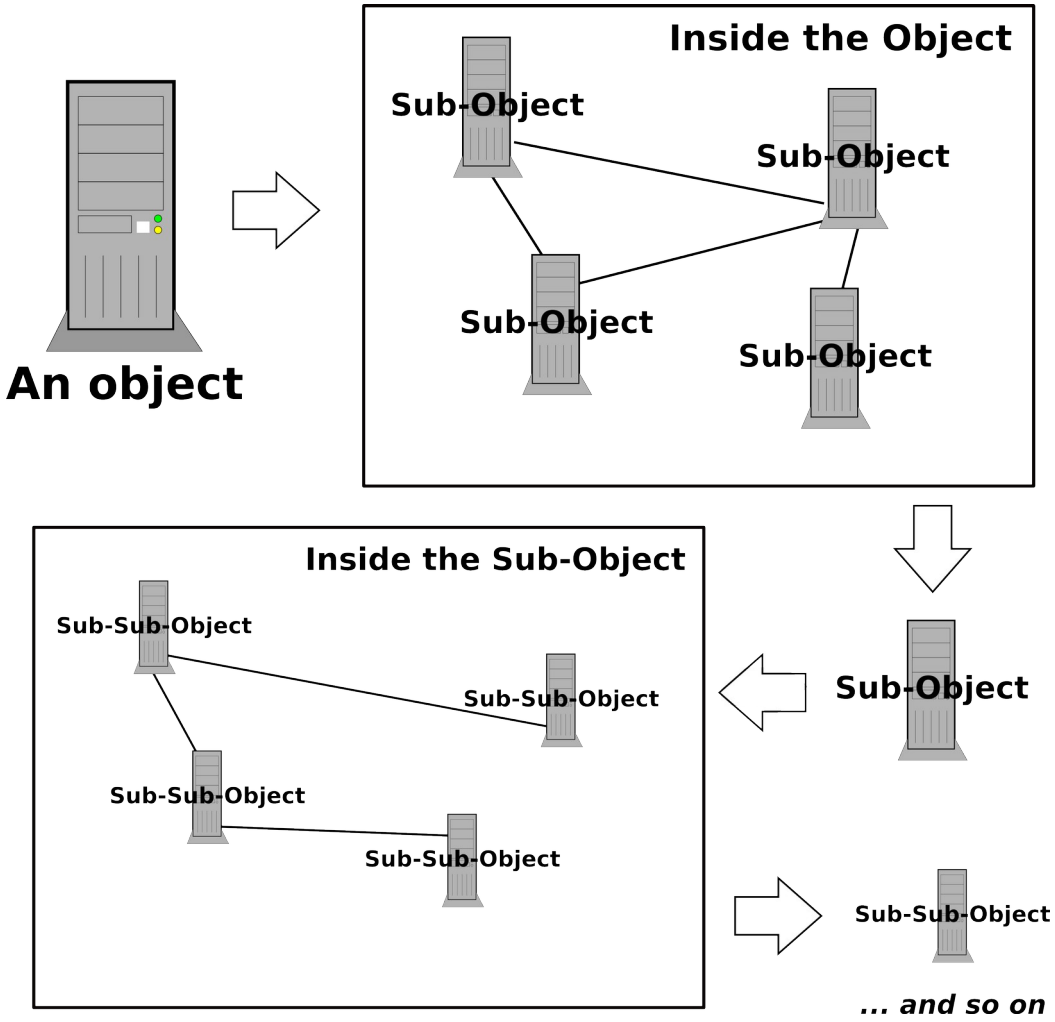
Recursive Design

- Consider the airline object from a previous slide
  - It actually is made up of a hierarchy of networks of smaller objects
- This is a complex system made up of layers of sub-systems of agents…
- This is characteristic of
  - Companies and organizations
  - Biological systems
  - Mechanical systems etc



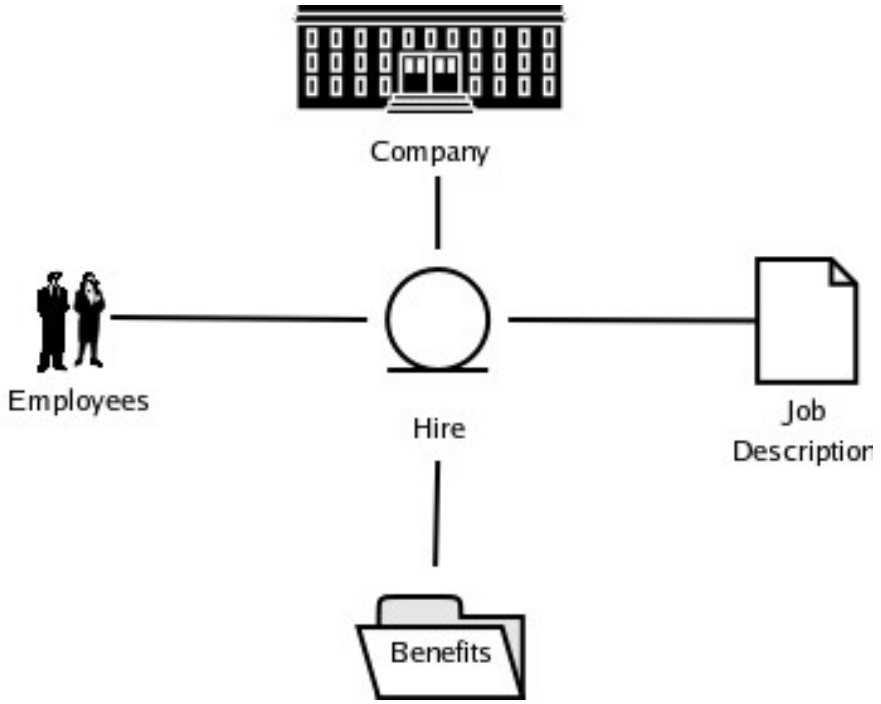
Recursive Design

- Alan Kay suggested that an object oriented approach is the best way to model and to design and develop these systems
  - Developed his principles of OOP
  - Java and other OO languages are designed to support these principles even though Kay thought they weren't OO enough
- The principles of OOP address the recursive nature of these systems



OOP #1

- **Everything is an object**
  - Everything can be represented as an object
- Actions and transactions are objects
  - Being born is a birth
  - Buying something is a purchase
  - Selling something is a sale
- Relationships between objects can be objects
  - Two married people is a marriage
  - A work relationship may be a job
- What this means is that the only basic construct we need in a OO language is a way to create objects

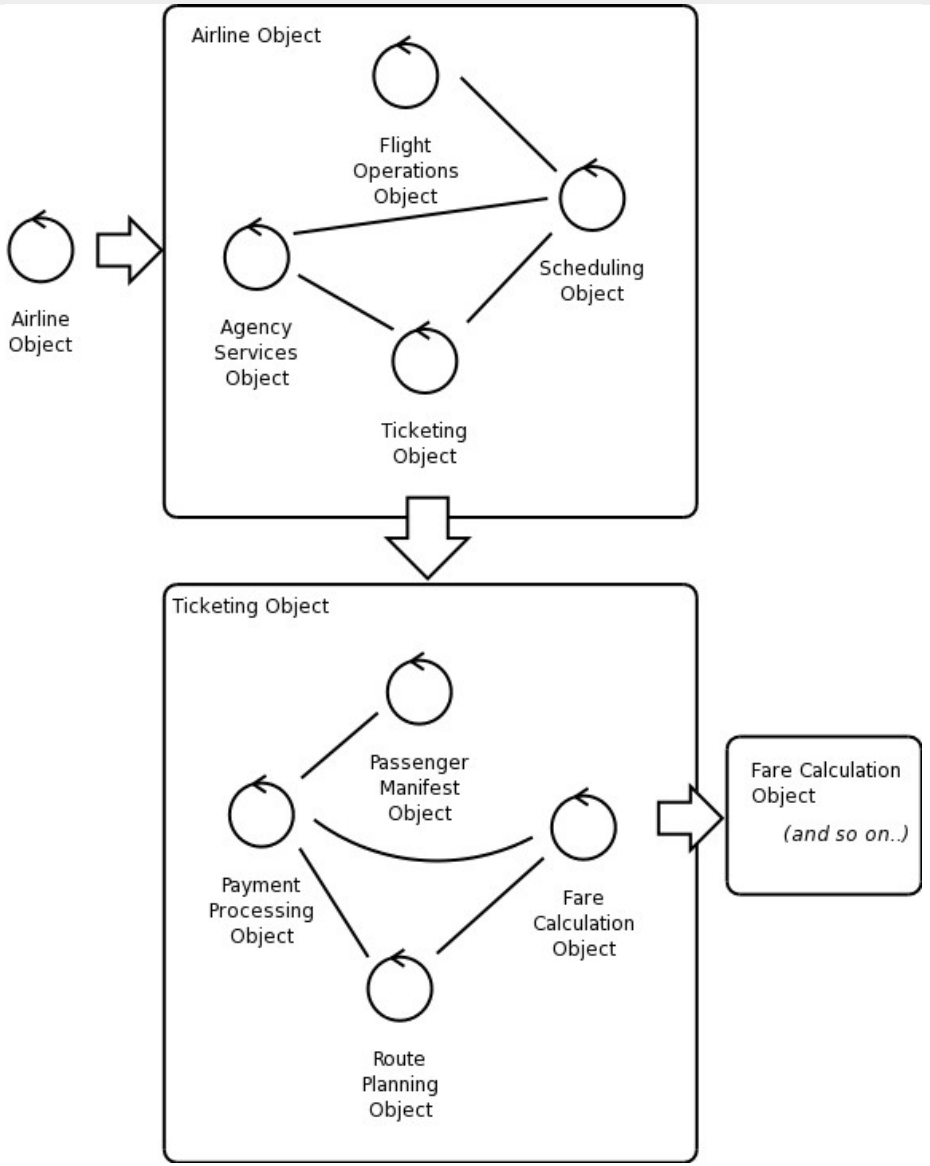


OOP #2

- **Systems are collections of objects collaborating for a purpose and coordinating their activities by sending messages to each other**
  - Systems tend to be gestalts – the system is more than the sum of the objects that make it up
    - *The “more” is the layer of organization that coordinates the objects*
    - *We often see a phenomenon called emergent behaviour in systems when the system displays behaviours that are not present in the individual objects that make up the system*
- However, by OOP#1, this system can be treated as an object
  - A programmer is an object
  - A group of programmers that work together on a project are a team object
  - A group of teams that work together on projects is an R&D department object

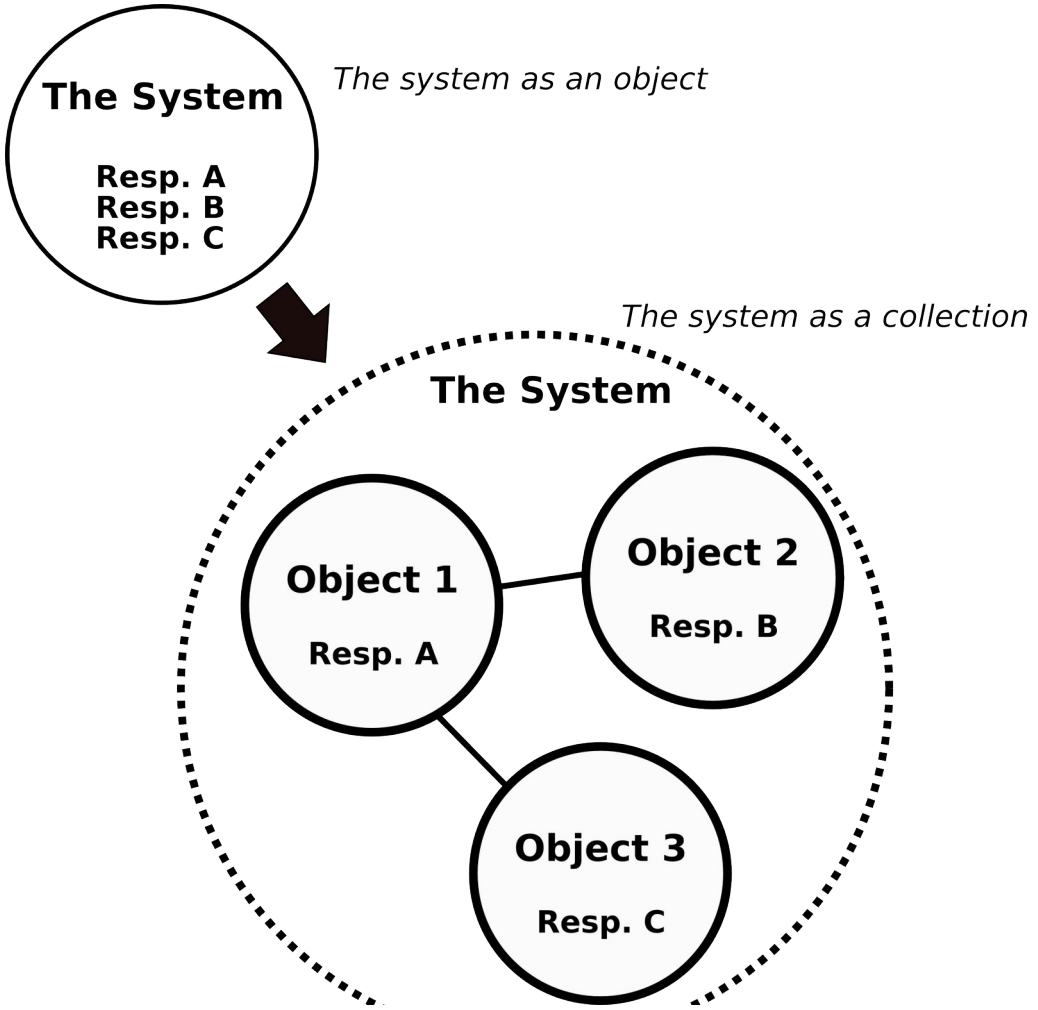
OOP #3

- **An object can have an internal structure composed of hierarchies of sub-objects.**
  - An object is a system (OOP #2)
  - The functionality of the object is delegated to sub-systems
  - The sub-systems themselves are objects (OOP #1)
- The first three axioms provide a framework for designing both an OO language and specific application designs that are exhibit this recursive property.



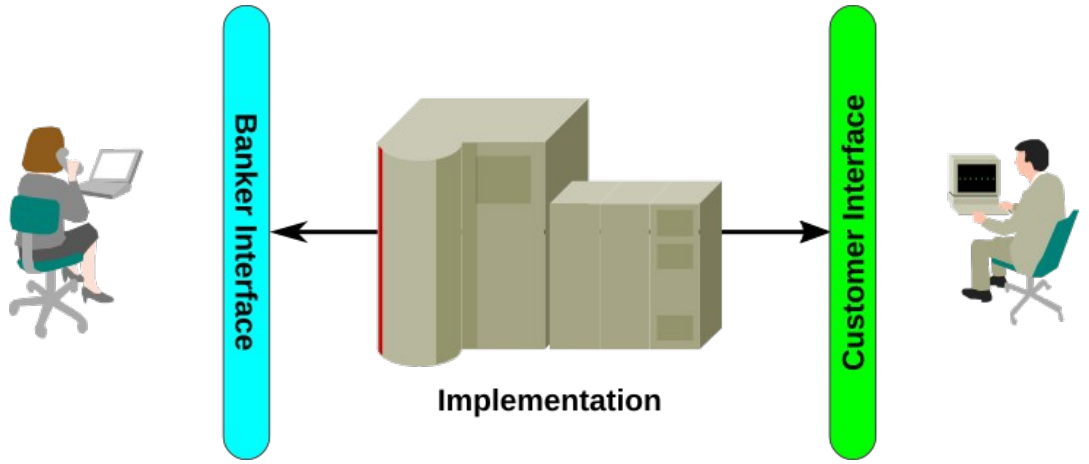
OOP #4

- **Objects within a system have individual responsibilities**
  - The responsibility of a system as a whole is distributed across the objects that make up the system by delegating specific responsibilities to individual objects
- This axiom can be summarized as
  - Each object has one responsibility or specialization
- This an OO version of several well known design principles in both software and engineering
- When a responsibility can be decomposed into sub-responsibilities
  - Each sub-responsibility can be assigned to a sub-object that specializes in that sub-responsibility
  - This is often called a functional decomposition



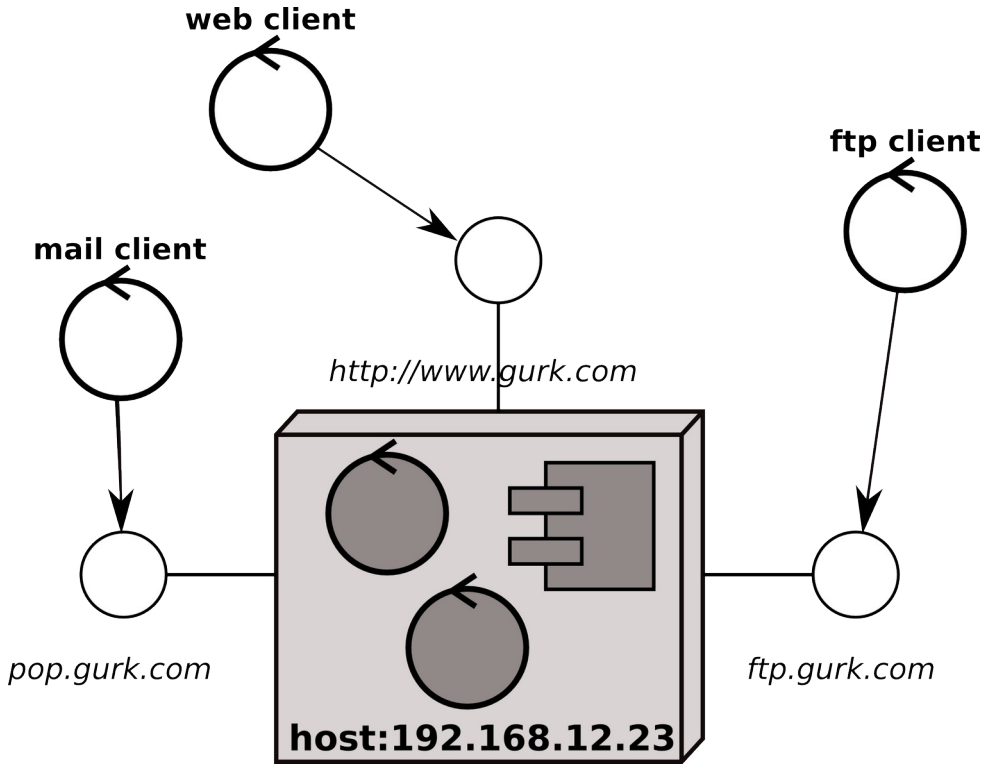
OOP #5

- **An object presents an interface that specifies which requests can be made of it and what the expected results of those requests are**
  - The interface is independent of the actual internal workings of the object
  - The interface presented by the object is often referred to as the object's "contract"
- An interface is what is visible to the other objects
  - It can be thought of as a list of the messages that the object understands
  - And a description of how the object will respond as a result of receiving a particular message



OOP #6

- **An object is of one or more types**
  - A type includes a role the object plays, a set of responsibilities and an interface
  - Each type may be derived from a hierarchy of types.
- This ties together the ideas of roles, interfaces and responsibilities under the concept of “type”
  - This axiom is unique because it does not talk about structure
- We don’t try to define what “type” means
  - Rather we note that it is a way to the ideas of the role objects play in a system, the interface associated with that role and the responsibilities of that role.
- An object that has more than one type is said to be *polymorphic*



Java Packages

- Java packages combine two concepts
  - A directory like structure for organizing Java classes
  - Namespaces to avoid naming conflicts between classes in different package
- Packages as structure
  - A package can be thought of as a directory in a file system
  - It has a name
  - It can contain various Java constructs like class definitions
  - But it can also contain other packages which enables a recursive organization
- Package are implemented as directories
  - Java is intended to be portable across file systems
  - Packages are an abstraction that is implemented into the local files system by the JRE
  - Dots are used as a directory structure
    - *com.accounting.payroll → com/accounting/payroll or com\accounting\payroll*

The package Statement

- There may be many files in a single package
  - There is no analogue to a directory table in a package
  - This is very OS dependent
  - Trying to have the package index or keep track of its contents is not technically feasible
- Instead, each file remembers which package it should be in
  - This is the package statement which is the first line in every file
  - There is a default unnamed package where files without package declarations go
  - This is only used for small quick and dirty code
  - Using it consistently is considered poor Java style

```
package com.mycorp;

public class Boot {

    public static void main(String[] args) {
        System.out.println("Welcome to MyCorp");

    }

}
```

The Bootstrap Problem

- All programs have to start somewhere, often called the bootstrap code
  - Most compiled programming languages have a main() function that represents the entry point to the program execution
- The problem is that Java only allows methods (functions) inside classes
  - That means that we have to stick a main() method is some class
  - Java doesn’t care where it is, as long as it can find it
- This is problematic for several reasons
  - Putting the main() method in an arbitrary location makes it hard to find
  - Whatever class it is in now has an additional responsibility – starting the application
- Best practice
  - Create a special class with only one responsibility – to run the main() method
  - And put the class in a specific location, the topmost package for example, so that it does not pollute the rest of the code

Demo

Working with Java Packages

Package as Namespace

- For this discussion we are going to only be referring to class definitions
  - This is just to illustrate the concept of visibility
  - We will refine this later when we work with classes in more detail
- Java uses *package* visibility by default
  - This means that every class in a package can refer to every other class in the same package
- Some classes can be declared to be *public*
  - This means that classes outside the package can also refer to it
- Java has a couple of special rules for public classes
  - There can only be one public class defined per file
  - The file has to have the same name as the public class
  - This makes it easier for Java to manage

Fully Qualified Class Names

- When code in one package refers to a public class in another package
  - Java has to find that definition.. somewhere
  - There is no index so just using the class name alone is pointless
- One way to help Java out is to prefix the class name with a full qualified name which is done by adding the package name as a prefix to the class name
  - This is like using a fully qualified path name in a file system

```
2  package com.mycorp;
3
4  public class Boot {
5
6      public static void main(String[] args) {
7          // can't find class
8          Coder kent = new Coder();
9          // Fully qualified class name
10         com.mycorp.dev.Coder anish = new com.mycorp.dev.Coder();
11
12     }
13
14 }
15
```

The import Statement

- Since it's a pain to use full qualified names, using the import statement makes easier
  - The full qualified class name is placed after the package statement in an import statement
  - This directs Java where to look to find the definition of the imported class
  - The import statement doesn't move anything, just allows Java to use the class name as a alias for the imported fully qualified name
- If there are a number of classes to be imported from a package the wildcard can be used instead of listing all the classes to be imported.

```
package com.mycorp;
import com.mycorp.dev.Coder;
// or import com.mycorp.dev.*;

public class Boot {

    public static void main(String[] args) {
        Coder kent = new Coder();
    }

}
```

Naming Conflicts

- It may may happen that an imported class name may conflict with another class
  - There may already be a class with that name in the package
  - There maybe two different classes with the same name being imported from two different packages
  - This is a name space collision
- In this case, to avoid ambiguity, one of the imported classes will have to use its full qualified name
- Notice in the example, the first import had to be removed
  - Java is only concerned with import statements when looking for collisions

```
package com.mycorp;
import com.mycorp.dev.Coder;
import com.mycorp.dev.backend.Coder;

public class Boot {

    public static void main(String[] args) {
        Coder kent = new Coder(); // ??? Which coder??
        Coder bjarne = new Coder(); // again.. which"
    }
}
```

```
package com.mycorp;
import com.mycorp.dev.backend.Coder;

public class Boot {

    public static void main(String[] args) {
        com.mycorp.dev.Coder kent = new com.mycorp.dev.Coder();
        Coder bjarne = new Coder(); // again.. which"
    }
}
```

Demo

Namespaces and Importing


Java™