Programming in Java

6. Object Oriented Programming

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” in the real world
  - 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

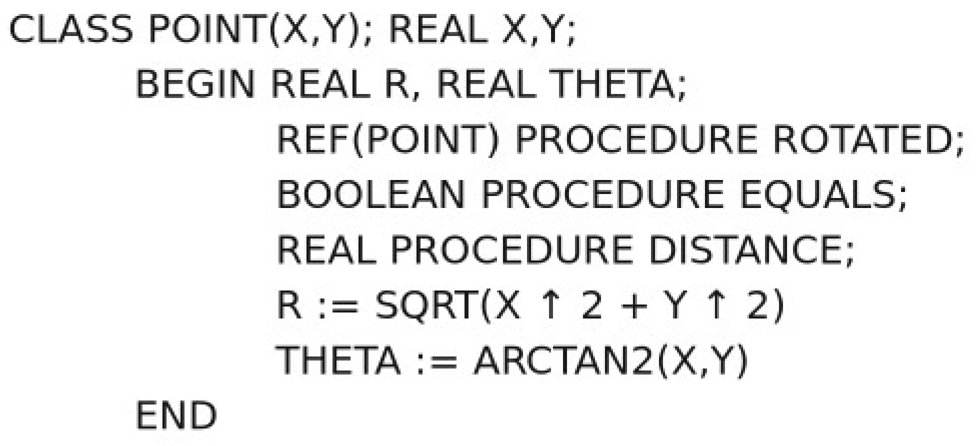
- First proposed by Professors Ole-Johan Dahl and Kristen Nygaard at the Nordic Centre for Computing at the University of Oslo in the late 1950s
- They posed the question
  - *“Why should people have to learn how to interact with a computer? Why can we not design a computer program that resembles what 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 was transpiled 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

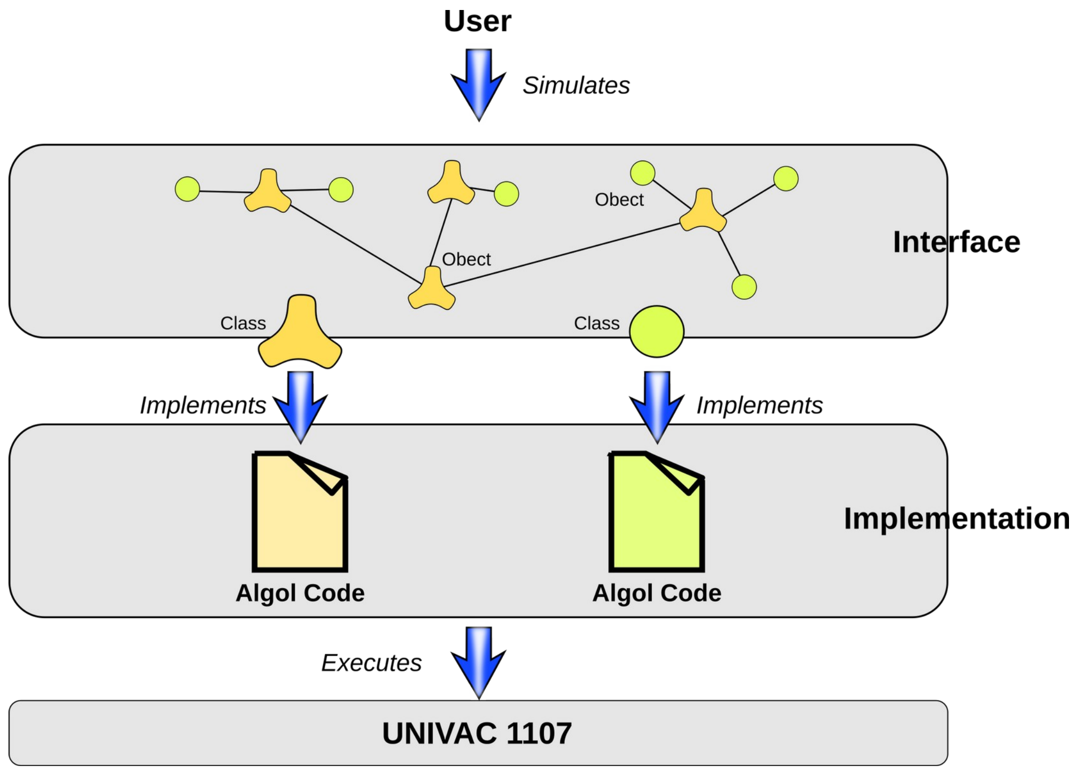
- The structure of a Java class definition is derived ultimately from the Simula definition
- This is an implementation of the object model
  - The class definition has a type, attributes and methods (called procedures)
  - We create objects by instantiating them from the class definition
- The code 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
```

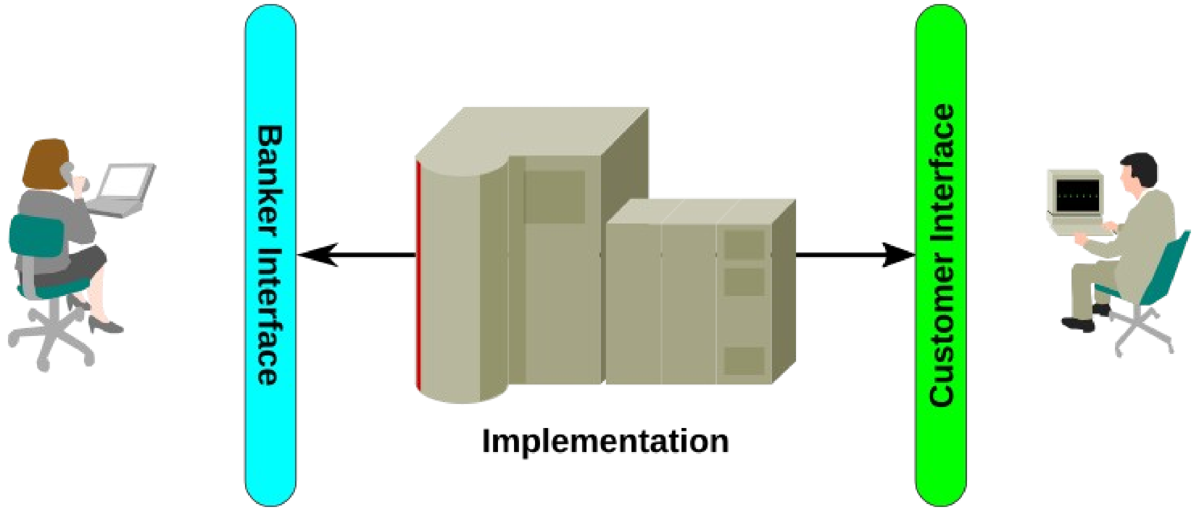
Interface and Implementation

- 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 physically 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)



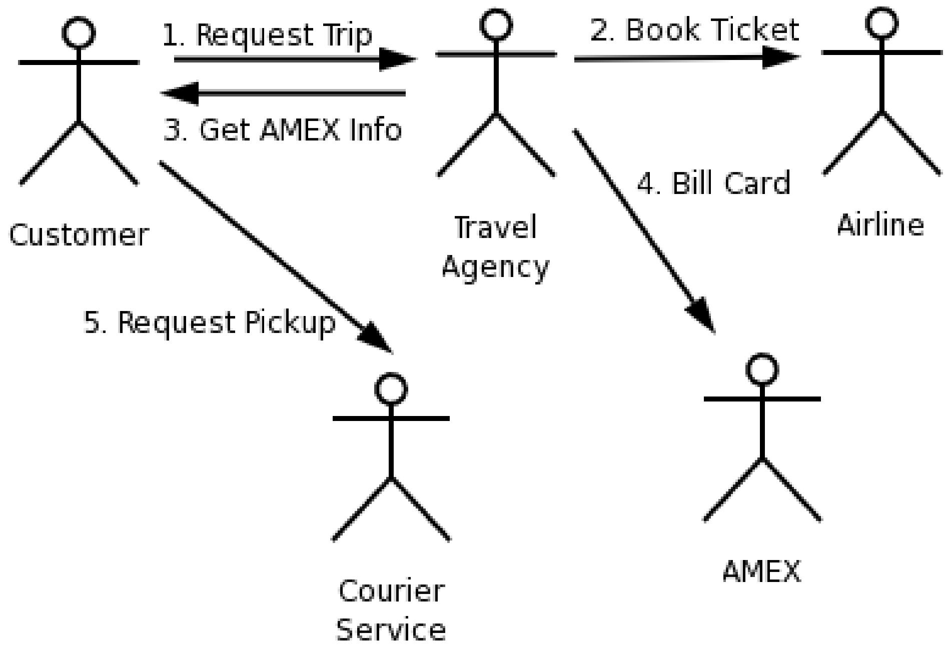
Interfaces as Views

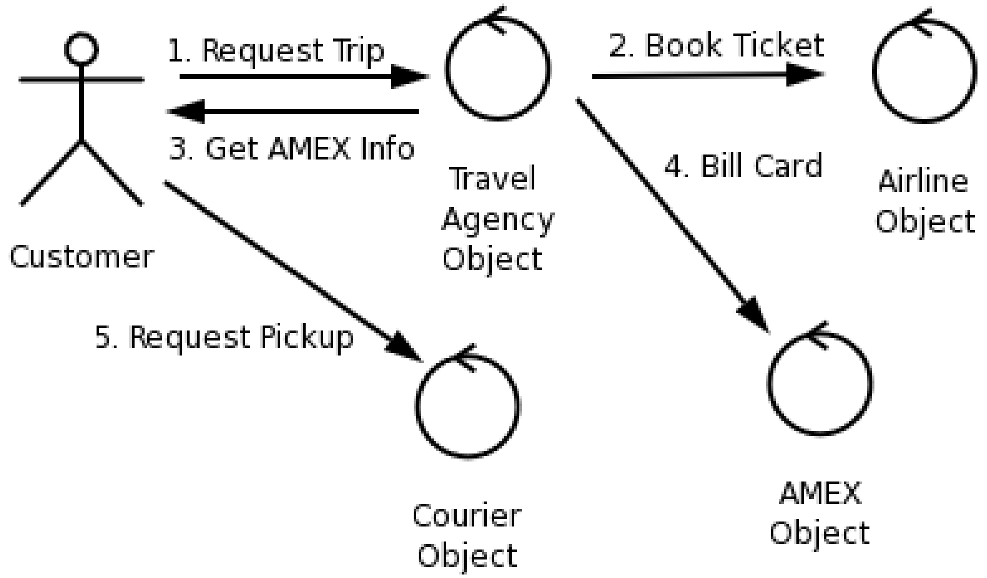
- One useful way to think of interfaces is that they are views into the underlying implementation
  - A view shows only what is of interest to the viewer
  - A view may translate something in the implementation into a representation unique to that view
  - A record in a bank database is presented as “My Account” to a customer
- 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 that only implements 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

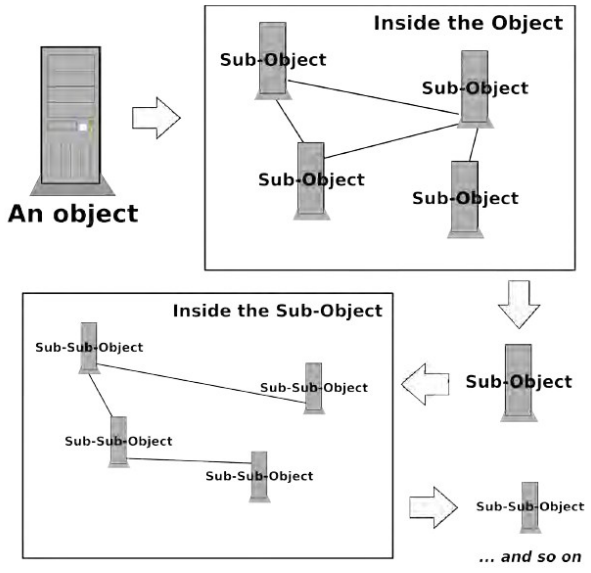




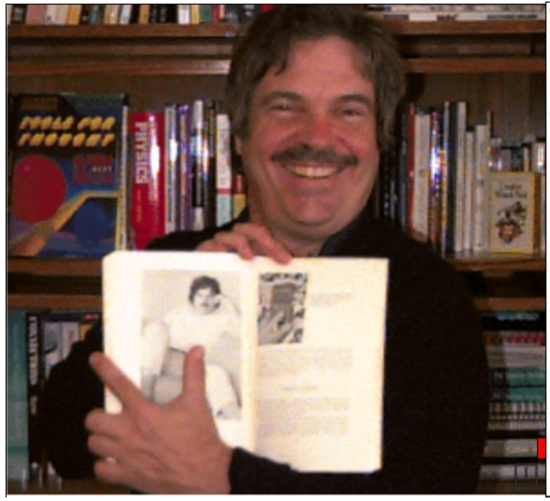
The manual system of people talking to each other – travel agents, airline representatives, courier dispatchers

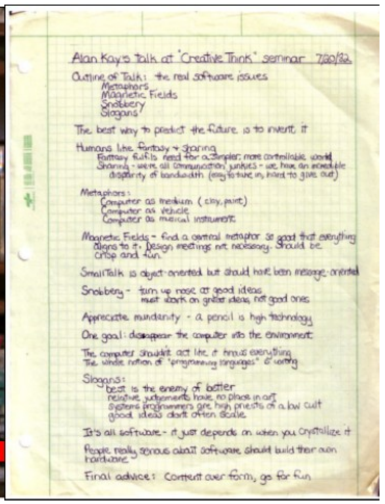
The system we want to build where the entities communicate directly instead of through human interfaces

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





Alan Kay's talk at "Creative Think" seminar 7/20/82

Outline of Talk: the real software issues
Metaphors
Magnetic Fields
Snobbery
Slogans

The best way to predict the future is to invent it

Humans like fantasy + sharing
Fantasy fulfils need for a simpler, more controllable world
Sharing - we're all communication junkies - we have an incredible disparity of bandwidth (easy to take in, hard to give out)

Metaphors:
Computer as medium (clay, paint)
Computer as vehicle
Computer as musical instrument

Magnetic Fields - find a central metaphor so good that everything aligns to it. Design meetings not necessary. Should be crisp and fun.

SmallTalk is object-oriented but should have been message-oriented

Snobbery - turn up nose at good ideas
must work on great ideas, not good ones

Appreciate mundanity - a pencil is high technology

One goal: disappear the computer into the environment

The computer shouldn't act like it knows everything
The whole notion of "programming languages" is wrong

Slogans:
best is the enemy of better
relative judgements have no place in art
systems programmers are high priests of a low cult
good ideas don't often scale

It's all software - it just depends on when you crystallize it

People really serious about software should build their own hardware

Final advice: Content over form, go for fun

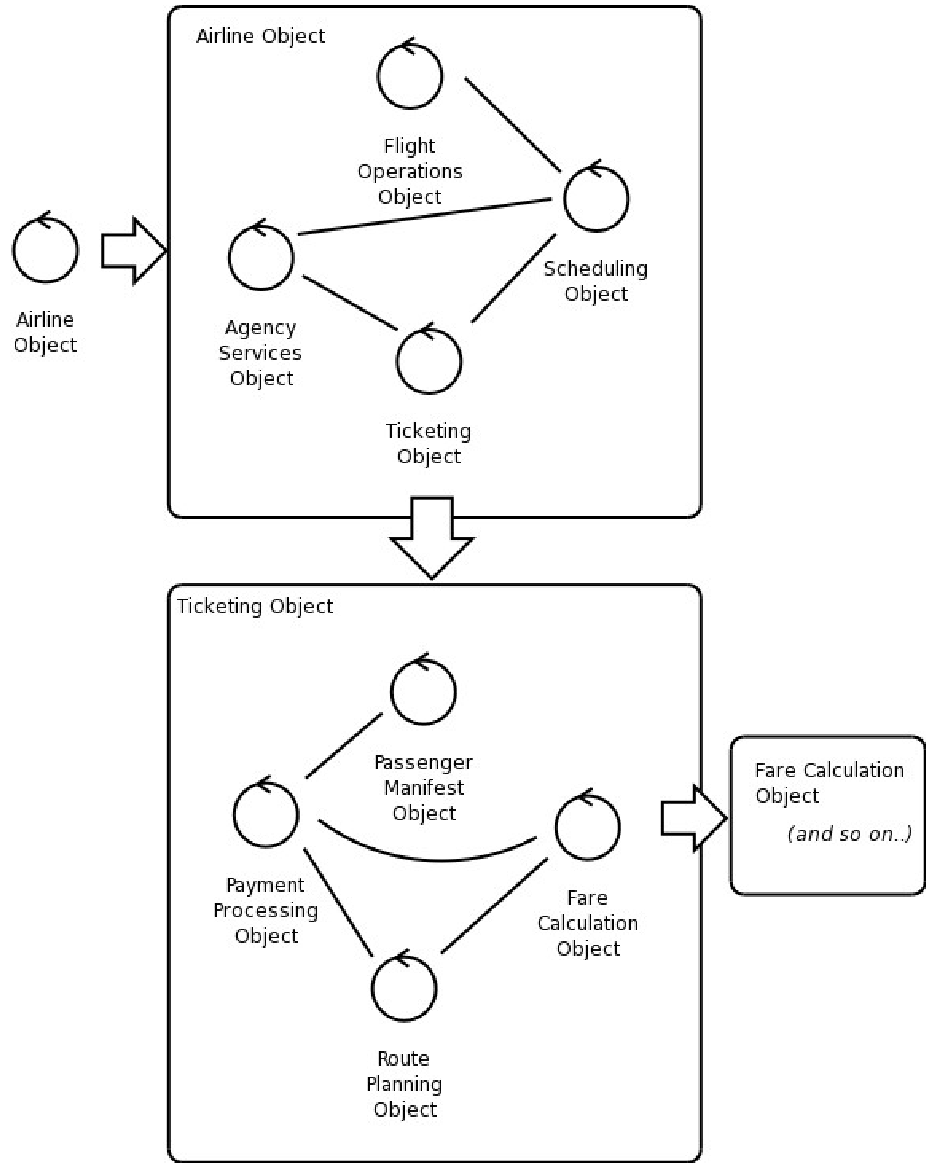
I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages (so messaging came at the very beginning – it took a while to see how to do messaging in a programming language efficiently enough to be useful)

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

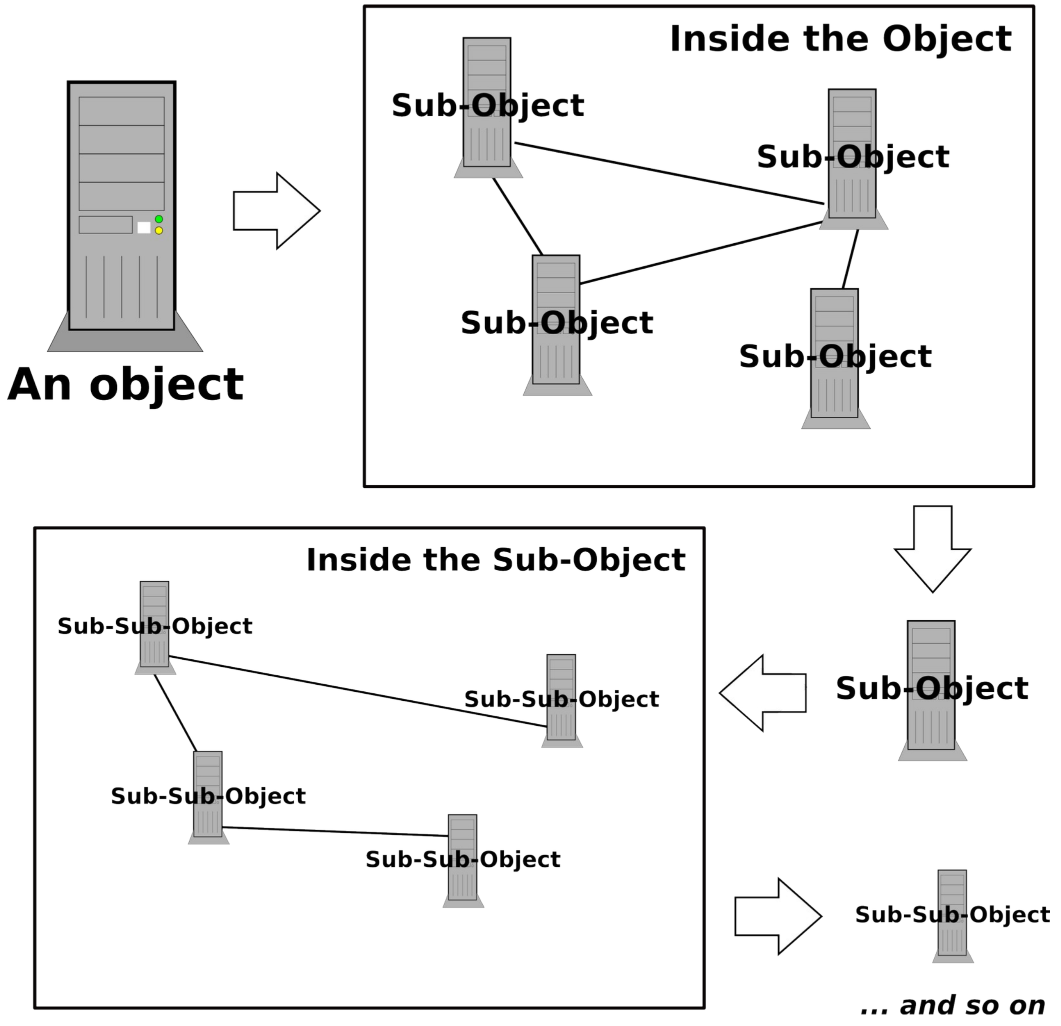
Recursive Design

- 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

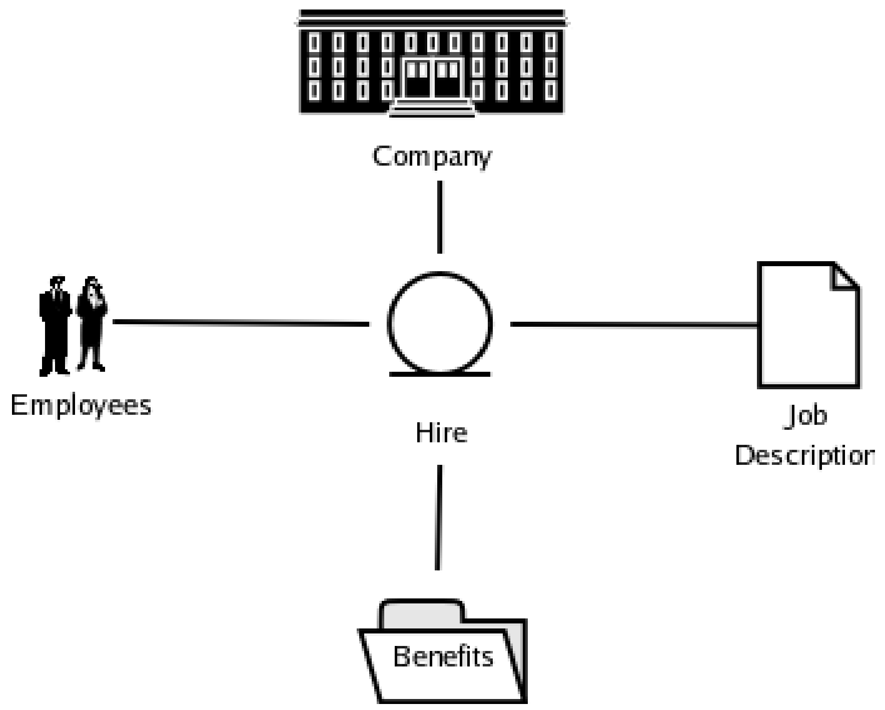
- Alan Kay suggested that an object oriented approach is the best way to model and to design and develop these systems
  - Developed his principles of OOP
  - Java and other OO languages are designed to support these principles even though Kay thought they weren't OO enough
- The principles of OOP address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



LearnQuest

OOP #1

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

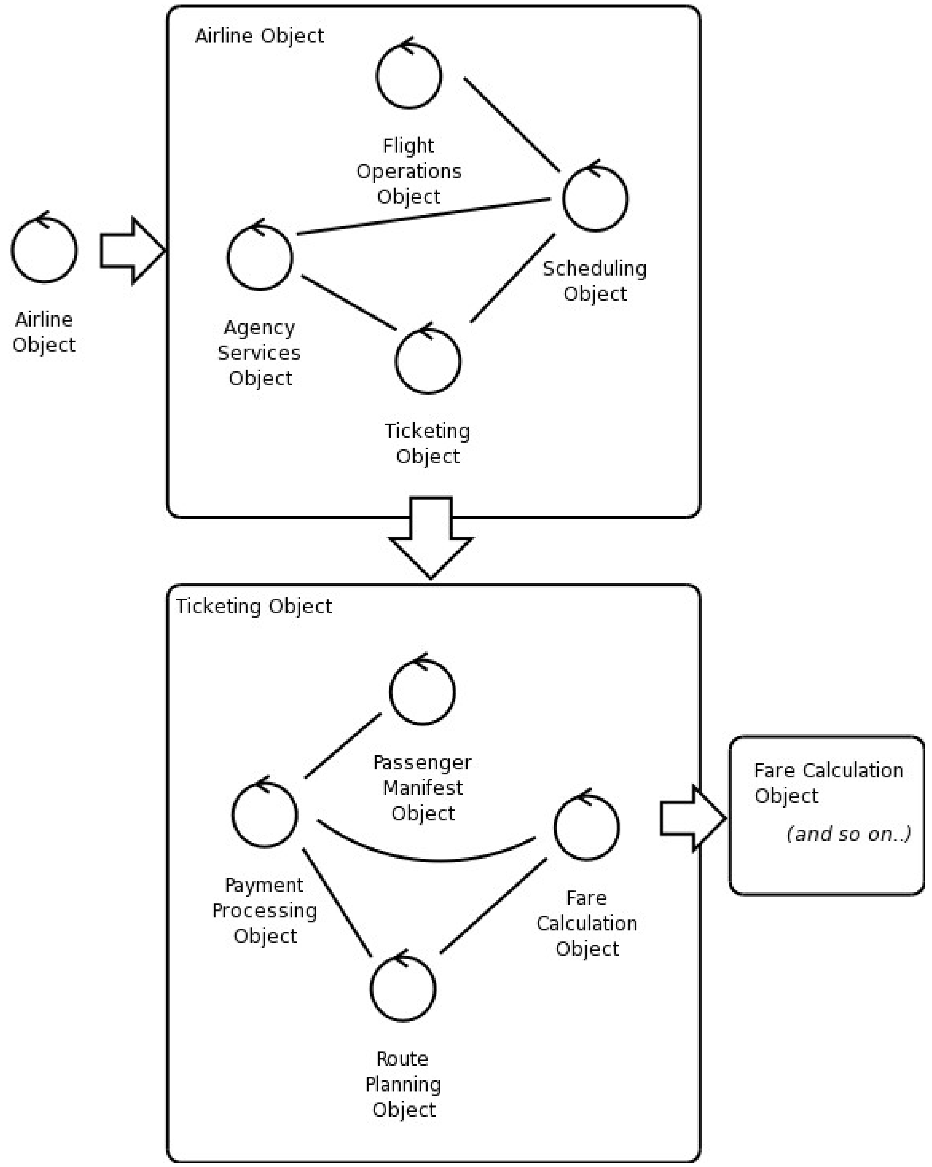


OOP #2

- **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

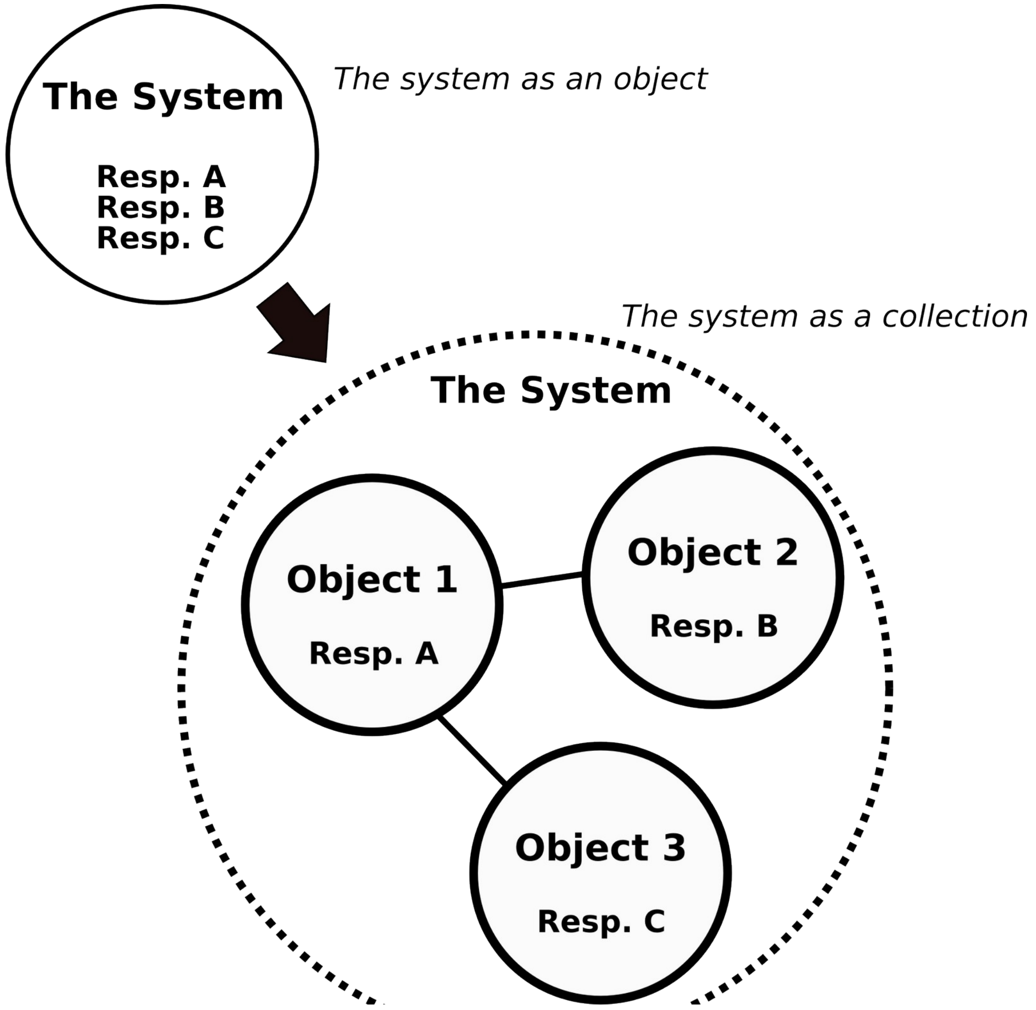
OOP #3

- **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.



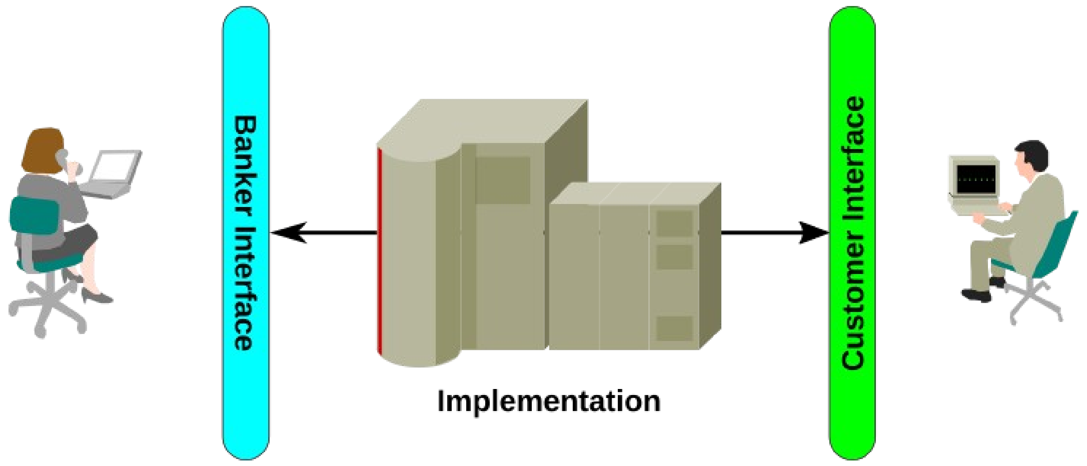
OOP #4

- **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



The Object Model

- The heart of Java programming is the object model
- All code in Java is written in class definitions and associated structures
- In the rest of this module we will examine
  - The structure of a class including attribute and methods
  - How objects are instantiated from class definitions
  - The design of Java methods
  - The different use cases for instance methods and variables as opposed to static methods and use cases

The Object Model

- There is no formal definition of the object model
- However in OO it is generally agreed that
  - Objects have a type and unique identity
  - Object contain a set of data relevant to their type which are called attributes or instance variables
  - Objects contain a set of instance methods that execute the messages other objects send them
  - The attributes and methods an object has are defined by its type
- A class definition consists of:
  - A set of instance variables representing the attributes of the object
  - An optional set of static variables representing attributes of all the objects of that type collectively
    - For example, the number of objects in existence of a type is a property of the collection or class, not any individual object in that class
  - A set of executable functions called instance methods that define the behaviour of objects of that type
  - An optional set of static methods that refer to some functionality of the class as a whole
    - For example, incrementing the number of objects or a type in existence

Designing Classes

- Writing classes is not the same as writing a sort method
- Remember that OOP is *iconic* which means that our classes look like what they automate.
  - Or at least what they are motivating influences our choice of classes
- Classes have two *layers*
  - An *interface* through which other classes interact with our class
    - The interface is made up of the public methods for that class (more on that in a bit)
  - An *implementation* which is where our code and data exist
    - The implementation cannot be accessed by anything outside the class
    - As a result, we are free to re-architect, re-design and rewrite our implementation code
    - As long as we keep the interface stable, changing the implementation will not break anything
- The actual classes and their public interfaces are defined during the design process
  - Usually comes out of a high level architecture
  - Defines the roles and responsibilities of the different classes that need to be written
  - As well as the public interfaces the classes will need to expose

Stack Versus Heap Memory

- This is standard architecture
- Usable memory for applications is divided into
  - The stack: Under the control of the OS
  - The heap: Under the control of the user
- The heap starts at the highest available memory address and grows down
- The stack starts at the lowest memory address and grows up
  - Up tp the limit of the allocated stack memory
- The white space in the middle is available memory for the heap
- If it goes to zero then
  - When the heap tries to allocate memory an out of memory error is generated
  - This usually causes a program to terminate

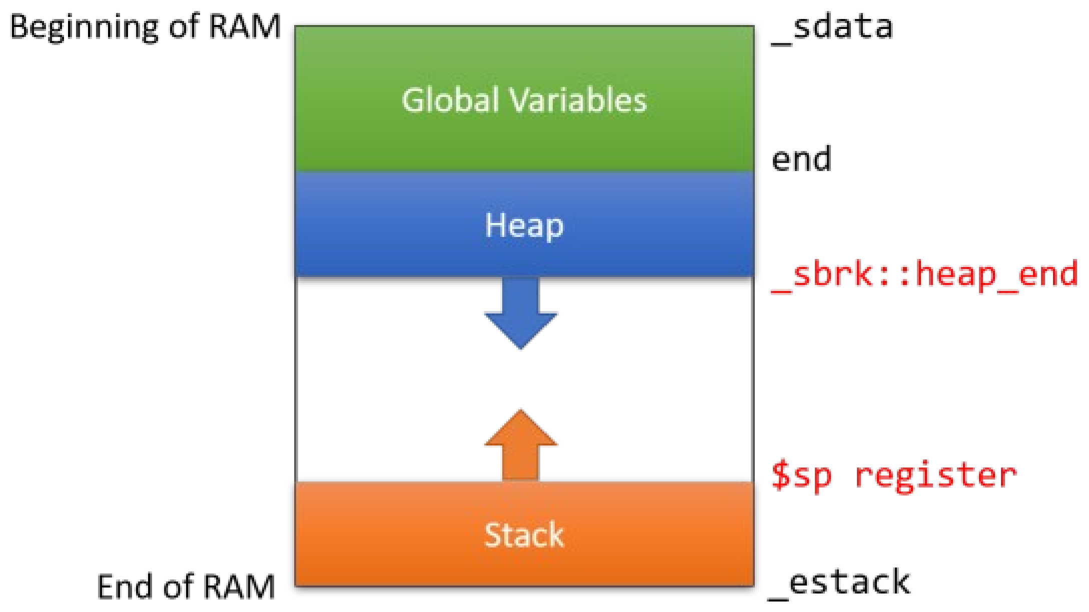
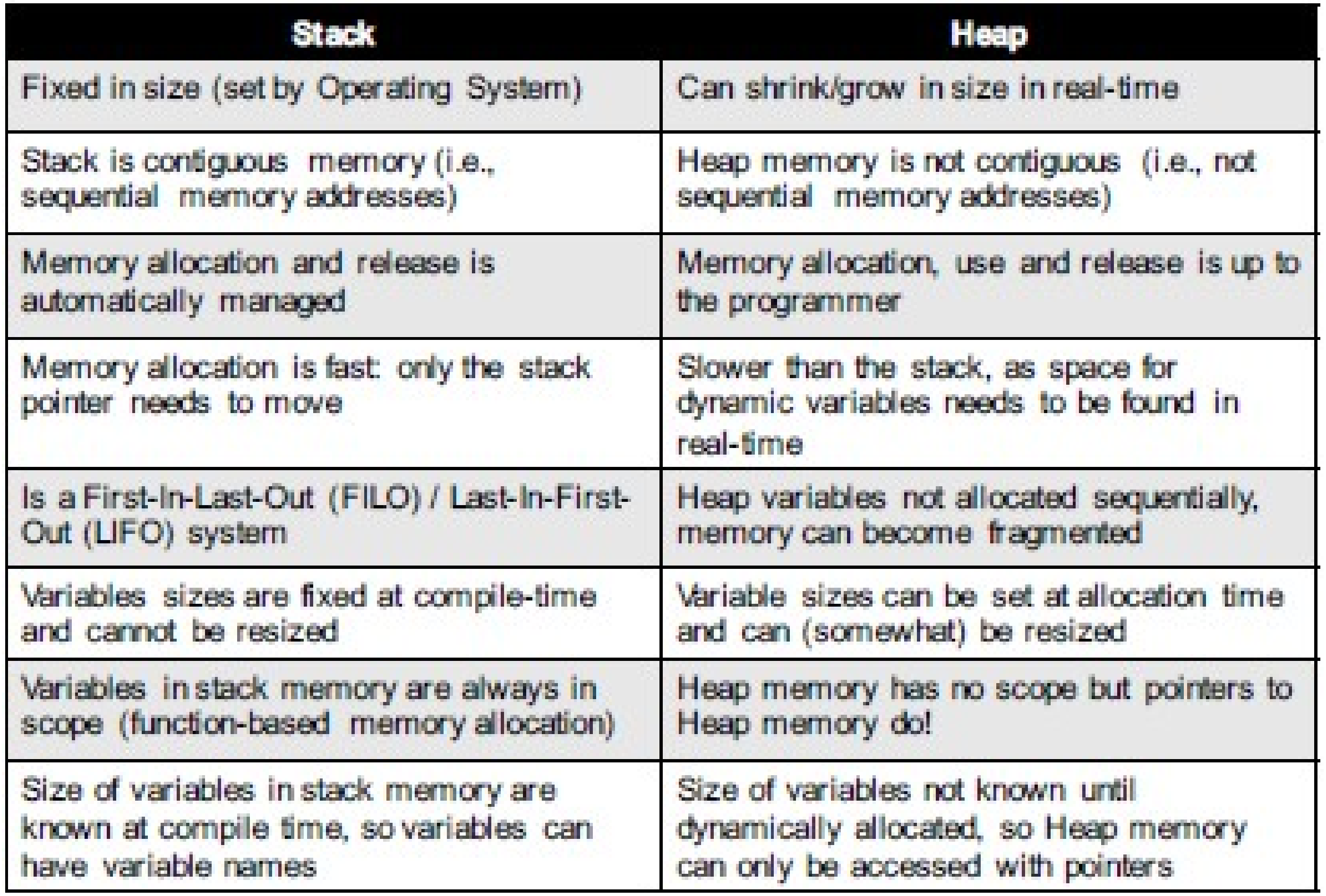


Image Credit: https://visualgdb.com/documentation/embedded/stackheap/

Stack Versus Heap Memory

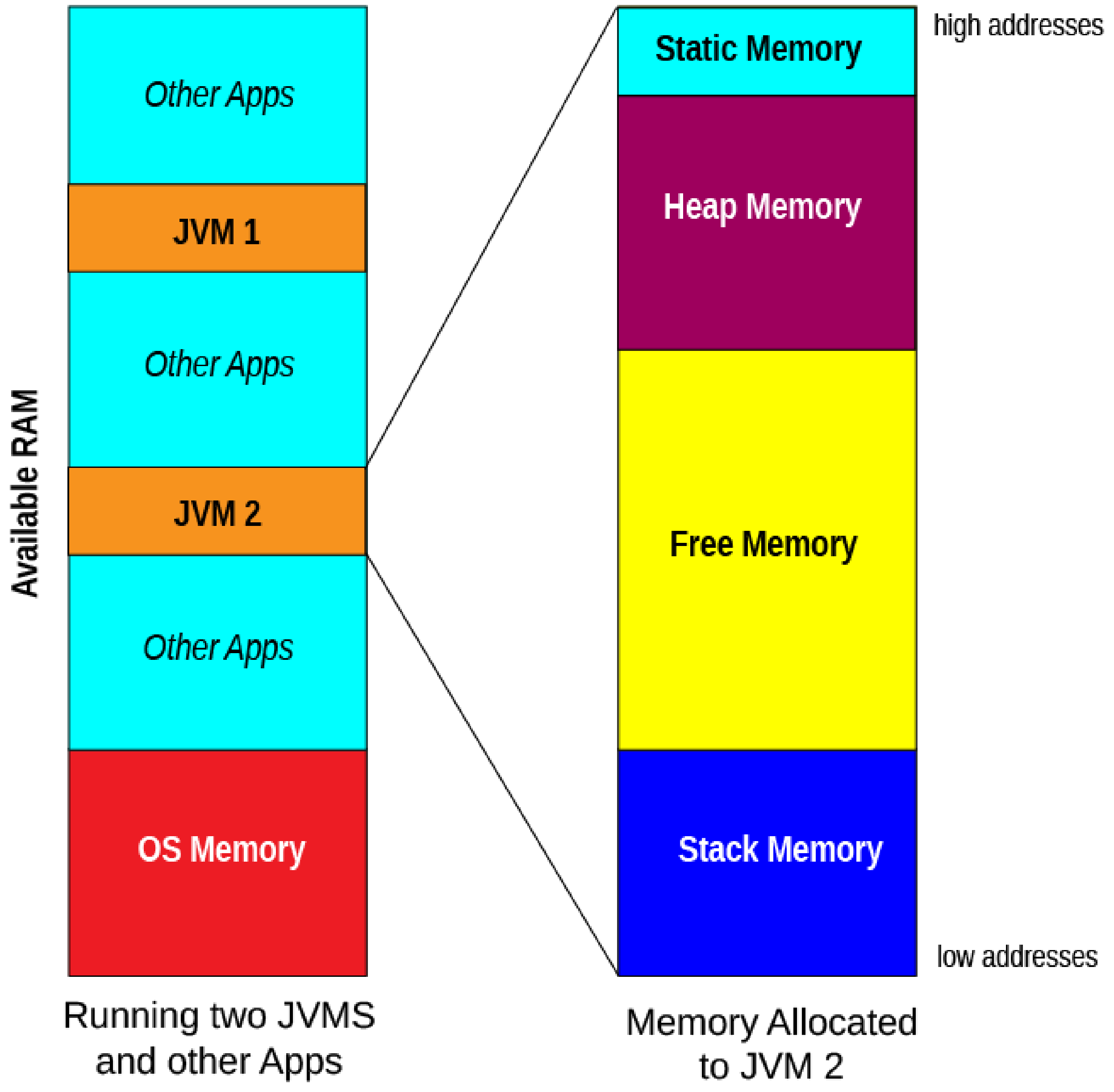


Stack	Heap
Fixed in size (set by Operating System)	Can shrink/grow in size in real-time
Stack is contiguous memory (i.e., sequential memory addresses)	Heap memory is not contiguous (i.e., not sequential memory addresses)
Memory allocation and release is automatically managed	Memory allocation, use and release is up to the programmer
Memory allocation is fast: only the stack pointer needs to move	Slower than the stack, as space for dynamic variables needs to be found in real-time
Is a First-In-Last-Out (FILO) / Last-In-First-Out (LIFO) system	Heap variables not allocated sequentially, memory can become fragmented
Variables sizes are fixed at compile-time and cannot be resized	Variable sizes can be set at allocation time and can (somewhat) be resized
Variables in stack memory are always in scope (function-based memory allocation)	Heap memory has no scope but pointers to Heap memory do!
Size of variables in stack memory are known at compile time, so variables can have variable names	Size of variables not known until dynamically allocated, so Heap memory can only be accessed with pointers

Image Credit: hhttps://cppbetterexplained.com

Java Memory Management

- Memory in the JVM is handled exactly like an OS handles physical memory
- The size of the stack is fixed when the JVM starts up
  - This can be adjusted by tweaking the JVM parameters
- The static memory is where any data that remains in memory for the duration of the time the JVM is running is located



Memory and Data

- There are three kinds of storage types in Java
- Static data
  - Also called permanent data
  - This is data that is initialized when the JVM starts up
  - It remains in memory until the JVM shuts down
  - This includes constants and interned data (we will define that later)
- Automatic data
  - This is data that is managed by the JVM on the stack
  - This data is primarily of local variables created during execution of a method
  - The stack removes these variables from memory when they go out of scope
- Managed data
  - This is data that is created on the heap in the code, usually with the “new” operator
  - It remains on the heap until it can no longer be accessed from the code
  - Inaccessible data is deleted from the heap when the garbage collector runs

Managed Variables

- Managed variables are create by our code
  - All user defined types are managed
  - Strings and arrays are also managed, but will deal with those later
  - More typically, managed variables are called *objects*
- Objects are created on the heap
  - By using the *new* operator
  - *New* returns the memory location of the newly created object
  - This is called a reference to the object and is essentially an integer that we cannot modify
  - Objects live until they go out of scope
- The scope of an object
  - An object is in scope as long as there is at least one variable that refers to (contains the address) of the object
  - Once the object can not longer be referenced, is marked for deletion so its memory can be reclaimed
  - The JVM keeps track of how many variables refer to the object (the reference count)
  - The object's memory is reclaimed during something called garbage collection

Lab 6-1

Working with Objects

Visibility

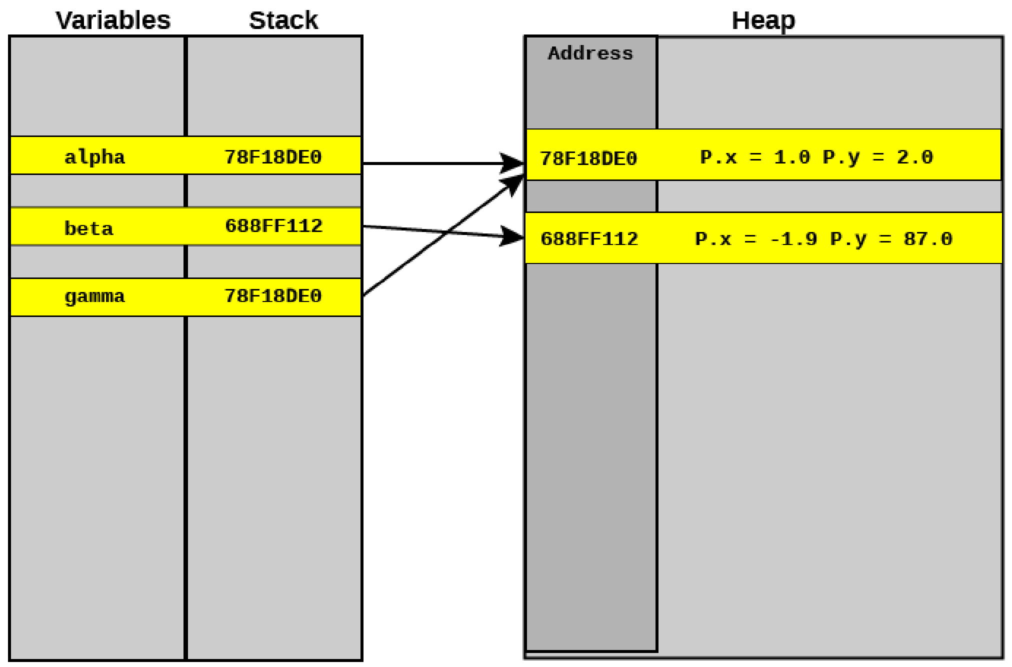
- There are three visibility modifiers used for both variables and methods
  - **Public** – visible to any object that can see the containing object
  - **Package** – visible to any object *in the same package* as the containing object
  - **Private** – visible only to methods in the containing object (data hiding)
- That means that there are actually two interfaces to each class
  - The *public* interface which are all the methods marked *public*
  - The *package* interface which are all the methods without a *public* or *private* modifier
  - Note that the *public* interface is a subset of the *package* interface

Instance Variables

- Classes can contain instance variables
  - These can be of any data type
  - The extent and scope of instance variables is the same as the object they belong to
  - Each object has its own copy of the instance variables
  - Below, 78718DE0 has a reference count of 2, while 688FF112 has a reference count of 1

```
lass P {
 float x;
 float y;
 P(int a, int b);


 alpha = new P(1.0,2.0);
 beta = new P(-1.9, 87.0);
 gamma  = alpha;
```



LearnQuest

Lab 6-2

Visibility

Methods

- Methods in Java have two parts
  - The return value
  - The method signature which is made up of the method name plus the argument list
  - For example, all of the following can be used in a class definition because they have different signatures
    - String convert(int k)
    - String convert(int k, int u)
    - String covert(boolean b)
  - However, this would be an error because it is the same signature as a prior method, the return value does not make up part of the signature
    - float convert(int k)
  - This is called method overloading or method polymorphism
  - The idea was that similar operations should have similar names for readability
  - Called method polymorphism because the same method has different forms depending on its parameter list

Method Invocation

- When we send a message to an object,
  - We use the notation obj.method() and expect some sort of return value
  - When we call a method in the same class, we typically use the form this.method()
    - The *this* is optional but it is considered good form so that the code is more understandable
  - The same is true for data
    - If we referred to the Student.name instance variable from method in the Student class
    - The form this.name would be used
    - The this is optional but once again, it is considered good form to use it
- All data should be private
  - It is accessed through special methods called *getters* and *setters*
  - Not providing a *setter* for an instance variable makes it read only
  - Not providing a *getter* restricts access to the data to just methods in the class definition
  - The *getter* methods are typically *public*
  - *Setters* also function as validators to check to see if a value being set is legal

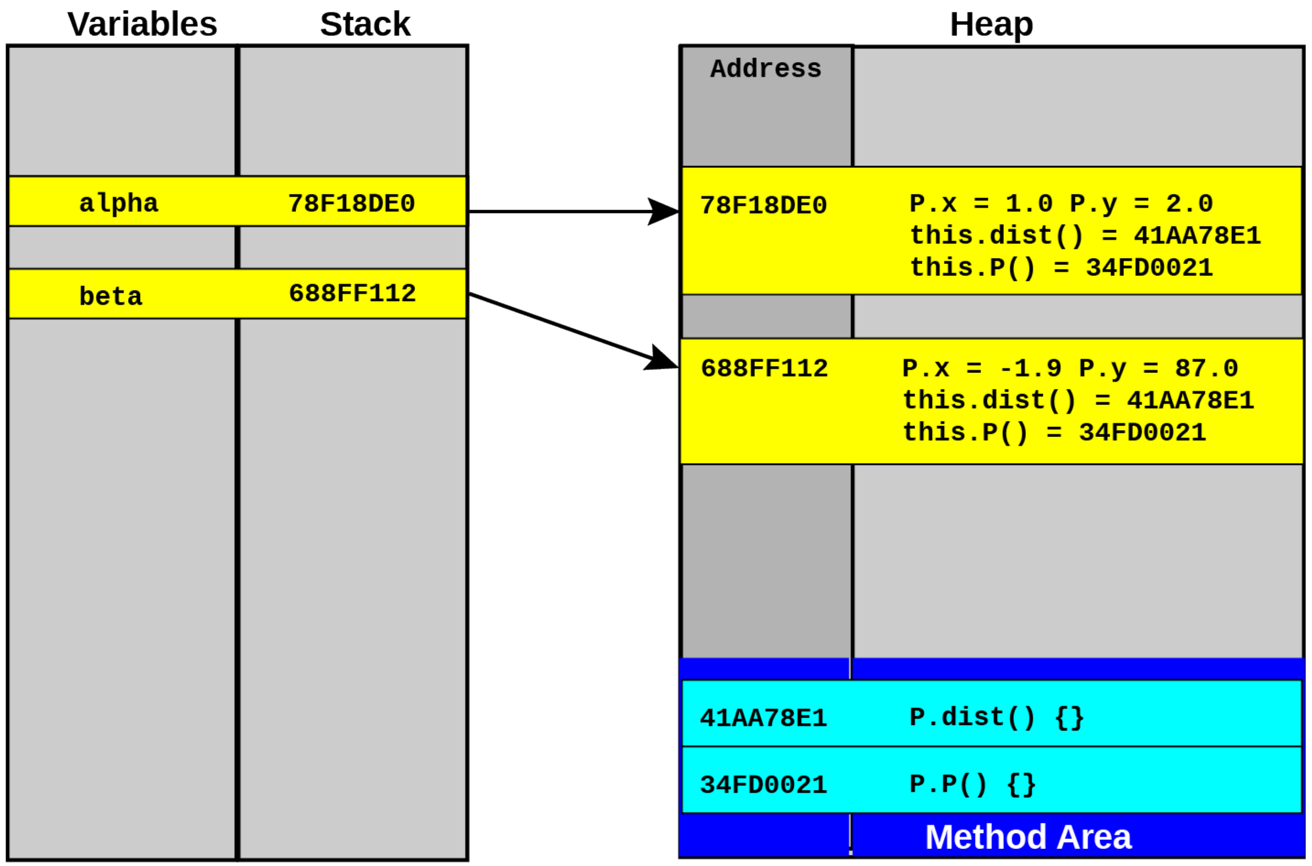
Memory Location of Methods

- All methods go into a special constant area in the heap
  - This is done when the class is loaded into the JVM
- When an object is created
  - Only heap memory for the instance variables is allocated
  - The object’s methods are pointers to the methods loaded into the method are
  - All objects share the same set of methods

Method Locations

```
class P {
  float x;
  float y;
  P(int a, int b);
  int dist();
}

P alpha = new P(1.0,2.0);
P beta = new P(-1.9, 87.0);
P gamma  = alpha;
```



Lab 6-3

Methods

Static Members

- Static Variables and data are syntactically the same as instance variables
  - They are just prefixed by the keyword *static*
- Rules for static methods
  - They can only refer to static data or other static methods
  - They may not refer to any non-static data or non-static methods
    - Remember that static mean initialized when the JVM starts
    - Instance variables and methods cannot be referenced until an object is created
- Constants are define using *public static final* variables
  - By convention, constants are always in upper case

Lab 6-4

Static Members

Constructor

- A constructor is a special method in the class that initializes the instance variables of an object when it is created
- The object creation process is:
  - First allocate memory for the object on the heap
  - Execute the constructor to initialize the object
- There can be multiple constructors, each with a different parameter list
- If you don't supply a constructor, Java supplies a default one
  - But as soon as you supply any constructor the default one is no longer present
- Constructors are like other instance methods in that they can access instance variables
  - But they are like static methods in the sense they are stored like static methods
- It is considered good Java form to always initialize instance variables using constructors

Lab 6-5

Constructors

Java™