Fundamental Concepts in OOP

- **Encapsulation**
  - Data Abstraction
  - Information hiding
  - The notion of class and object

- **Inheritance**
  - Code reusability
  - Is-a vs. has-a relationships

- **Polymorphism**
  - Dynamic method binding
Object Lifetime: Constructors

- **Constructors** are methods used to initialize the content of an object
  - They do not allocate space

- Most languages allow multiple constructors
  - They are distinguished using different names or different parameters (type and/or number)
  - Java and C++ overload the constructor name, so the appropriate methods is selected using the number and the type of the arguments
    - `Rectangle r;`:
    - Invokes the parameterless constructor
  - Smalltalk and Eiffel support different constructor names

Inheritance

- **Base Class** (or Parent Class or Superclass)
- **Derived Class** (or Child Class or Subclass)
- **Is-a relationship**
Java Example

- Dialog constructors
  - http://java.sun.com/j2se/1.4/docs/api/java/awt/Dialog.html\#constructor_summary
- FileDialog constructors

Constructors in Eiffel

```eiffel
class COMPLEX
creation
  new_cartesian, new_polar
feature {ANY}
  x, y : REAL;
  new_cartesian {x_val, y_val : REAL} as
    x := x_val; y := y_val;
  end;
  new_polar {r, theta : REAL} as
    x := r * cos (theta);
    y := r * sin (theta);
  end;
  -- other public methods
feature {NONE}
  -- private methods
end -- Class COMPLEX

... a, b : COMPLEX;
... !th.new_cartesian (0, 1);
... !a.new_polar (pi/2, 1);
```

Explicit Constructor Declaration
References and Values

- Some OO languages use the reference model
  - More elegant
  - Extra level of indirection on every access
  - E.g. Java, Simula, Smalltalk

- Other languages use the value model
  - More efficient
  - More difficult to control initialization
    » E.g. uninitialized objects, mutual references
  - E.g. C++, Ada 95

---

Constructors in C++

```cpp
foo b; // calls foo::foo ()
...
foo b (10, 'x'); // calls foo::foo (int, char)

foo a; // calls foo::foo ()
bar b; // calls bar::bar ()
...
foo c (a); // calls foo::foo (foo&)
foo d (b); // calls foo::foo (bar&)

foo c = a; // calls foo::foo (foo&)
foo d = b; // calls foo::foo (bar&)

foo a, c, d; // calls foo::foo () three times
bar b; // calls bar::bar ()

C = a; // calls foo::operator= (foo&)
d = b;  // calls foo::operator= (bar&)
```
Execution Order

• How is an object of class B derived from class A initialized?
• In C++ and Java, the constructor of A is invoked before the constructor of B
  – Why?
    » So the B constructor never sees uninitialized attributes
  – What are the arguments of the A constructor?
    » In C++, they are explicitly defined
      B::B (B::params) : A (A::args) { ... }
    » Furthermore, constructors for object arguments can also be initialized
      list_node() : prev(this), next(this), head_node(this), val(0) { }

Java Example

• See Java Language Specification

• Alternate constructor
• Superclass constructor
  – Unqualified superclass constructor
  – Qualified superclass constructor invocations
    » Inner classes
Object Lifetime: Destructors

- **Destructors** are methods used to finalize the content of an object
  - They do not deallocate space
  - Language implementations that support garbage collection greatly reduce the need for destructors
    » Most C++ compiler do not support GC

C++ Example

- In general, C++ destructors are used for manual storage reclamation

```cpp
class name_list_node : public gp_list_node {
  char *name;
public:
  name_list_node () { // pointer to the data in a node
    name = 0;       // empty string
  }
  name_list_node (char *n) {
    name = new char[strlen(n)];
    strcpy(name, n);    // copy argument into member
  } ~name_list_node () { // Destructor
    if (name != 0) {
      delete name;
    }
  }
};
```
Heap-based Allocation

- The **heap** is a region of storage in which subblock can be allocated and deallocated
  - This **not** the heap data structure

![Heap representation](image)

Garbage Collection

- Explicit reclamation of heap objects is problematic
  - The programmer may forget to deallocate some objects
    - Causing **memory leaks**
    - In the previous example, the programmer may forget to include the **delete** statement
  - References to deallocated objects may not be reset
    - Creating **dangling references**

![Garbage collection diagram](image)
Garbage Collection

• Explicit reclamation of heap objects is problematic
  – The programmer may forget to deallocate some objects
    » Causing memory leaks
    » In the previous example, the programmer may forget to include the delete statement
  – References to deallocated objects may not be reset
    » Creating dangling references

ptr2

Garbage Collection

• Automatic reclamation of the head space used by object that are no longer useful
  – Developed for functional languages
    » It is essential in this programming paradigm. Why?
  – Getting more and more popular in imperative languages
    » Java, C#, Python

• It is generally slower than manual reclamation, but it eliminates a very frequent programming error
  – Language without GC usually have memory profiling tools
Garbage Collection Techniques

- When is an object no longer useful?
- There are several garbage collection techniques that answer this question in a different manner
  - Reference counting
  - Mark-and-sweep collection

Reference Counting

- Each object has an associated reference counter

Stack

Heap

- The run-time system
  - keeps reference counters up to date, and
  - recursively deallocates objects when the counter is zero
Reference Counting

Problems

- Extra overhead of storing and updating reference counts
- Strong typing required
  - Impossible in a language like C
  - It cannot be used for variant records
- It does not work with circular data structures
  - This is a problem with this definition of useful object as an object with one or more references

Reference Counting

Circular Data Structures

- Each object has an associated reference counter

Stack

```
stooges := nil;
stooges
```

Heap

```
2 *larry*
1 *moe*
1 *curly*
```

Circular Structure

```
1 *larry*
1 *moe*
1 *curly*
```
Mark-and-Sweep Collection

- A better definition of *useless* object is one that cannot be reached by following a chain of valid pointers starting from *outside* the heap
- Mark-and-Sweep GC applies this definition
- Algorithm:
  - Mark every block in the heap as useless
  - Starting with all pointers outside the heap, recursively explore all linked data structures
  - Add every block that remain marked to the *free* list
- Run whenever the free space is low

Mark-and-Sweep Collection

- Block must begin with an indication of its size
  - Type descriptor that indicate their size makes this requirement unnecessary
- A stack of depth proportional to the longest reference chain is required
  - But we are already running are out of space!
  - Pointer reversal embeds the stack in the sequence of references in the heap
    - The GC reverses each pointer it traverses
Mark-and-Sweep Collection

Point Reversal

R is outside the heap

Store-and-Copy

- Use to reduce external fragmentation
  
  Heap
  
  ![Heap Diagram]
  
  Allocation request
  
  - S-C divides the available space in half, and allocates everything in that half until it is full
  - When that happens, copy each *useful* block to the other half, clean up the remaining block, and switch the roles of each half
Reading Assignment

- Scott
  - Read Sect. 10.3
  - Read Sect. 7.7.2 (dangling references)
  - Read Sect. 7.7.3 (garbage collection)
  - Garbage collection in Java JDK 1.2