Announcements

- HW8 due on Thursday
- HW9: Java Concurrency
  - Will post on Thursday
  - Due May 9th
  - Java infrastructure
    - Download the Eclipse IDE
- Rainbow grades
  - HW 1-6
  - Exam 1-2
  - Quiz 1-7
- Questions/concerns, contact us ASAP

Announcements

- HW9 is to be done in pairs
- Pair programming
- If you don’t have a partner, email me
  - Do so by Monday, May 2nd at noon. Requests received later than that will remain unanswered

Announcements

- A slight change in schedule
  - Was
    - Dynamic languages
    - OO languages
    - Concurrency
  - Now
    - OO languages
    - Concurrency
    - Dynamic languages

Last Class

- Control Abstraction
- Parameter Passing Mechanisms
  - Call by value
  - Call by reference
  - Call by value-result
  - Call by result
  - Call by name
  - Call by sharing

Lecture Outline

- Object-oriented programming
- Encapsulation and inheritance
- Initialization and finalization
- Subtyping and dynamic method binding
- Polymorphism
- Concurrency
Benefits of Object Orientation

- Abstraction
  - Classes bridge the gap between concepts in the application domain and software
  - E.g., domain concept of Customer maps to class Customer

- Encapsulation
  - Classes provide interface but hide data
  - Easier to understand and use
  - Can be changed internally with minimal impact

- Reuse
  - Inheritance and composition provide mechanisms for reuse

- Extensibility

Some Terminology

- Base class and derived class
  - E.g., in C++: class queue : public list
  - queue is the derived class and list is the base class
  - In Java we have class queue extends list

- Other terms are superclass and subclass
- Also, parent class and child class
- By deriving new classes programmer forms class hierarchies

Encapsulation and Inheritance

- Access control modifiers – public, private, and others
  - What portion of the class is visible to users?
  - Public, protected or private visibility
  - Java: Has package as default; protected is slightly different from C++
  - C++: Has friend classes and functions
  - Smalltalk and Python: all members are public

- With inheritance
  - What control does the superclass have over its fields and methods?
    - There are different choices
    - C++: a subclass can restrict visibility of superclass members
    - C#, Java: a subclass can neither increase nor restrict visibility of superclass members

Initialization and Finalization

- Reference model for variables used in Java, Smalltalk, Python
  - Every variable is a reference to an object
  - Explicit object creation: foo b = new foo();

- Value model for variables used in C++, Modula-3, Ada-95
  - A variable can have a value that is an object
  - Object creation can be implicit: e.g. foo b;
  - How are objects destroyed?

Question

- Consider the following code:

  A a;  // a is a local variable of type A
  a.m();  // We call method m on a

  What happens in C++?
  What happens in Java?

More on implicit creation in C++

- C++ requires that an appropriate constructor is called for every object implicitly created on the stack, e.g., A a;

  - What happens here: foo a;
    - Compiler calls zero-argument constructor foo::foo()
  - What happens here: foo a(10, ‘x’);
    - Calls foo::foo(int, char)
More on implicit creation in C++

- What happens here:
  
  ```c++
  foo a;
  foo c = a;
  ```

  - Calls `foo::foo()` at `foo a;` calls copy constructor `foo::foo(foo&)` at `foo c = a;`
  - = operator here stands for initialization, not assignment!

- Calls `foo::foo()` twice at `foo a, c;` calls assignment operator `foo::operator=(foo&)` at `c = a;`
  - = operator here stands for assignment!

Lecture Outline

- Object Oriented programming
- Encapsulation and inheritance
- Initialization and finalization
- Subtyping and dynamic method binding
- Polymorphism
- Concurrency

Subtyping and Dynamic Method Binding

- Subtyping and subtype polymorphism – the ability to use a subclass where a superclass is expected
  - Thus, dynamic method binding
  - Advantages?
  - Disadvantages?

  - C++: static binding is default, dynamic binding is specified with keyword `virtual`
  - Java: dynamic binding is default, static binding is specified with `final`

Example

- Application draws shapes on screen
- Possible solution in C

```c
enum ShapeType { circle, square };  
struct Shape { ShapeType t };  
struct Circle  
{ ShapeType t; double radius; Point center; };  
struct Square  
{ ShapeType t; double side; Point topleft; };  
```

```c
void DrawAll(struct Shape *list[], int n) {  
  int i;  
  for (i = 0; i < n; i++) {  
    struct Shape *s = list[i];  
    switch (s->t) {  
      case square: DrawSquare(s); break;  
      case circle: DrawCircle(s); break;  
    }  
  }  
}
```

What problems do you see here?
Example

- **OO Solution in Java**

  ```java
  abstract class Shape { public void draw(); }
  class Circle extends Shape { … }
  class Square extends Shape { … }
  public void DrawAll(Shape[] list) {
    for (int i=0; i < list.length; i++) {
      Shape s = list[i];
      s.draw();
    }
  }
  ```

Benefits of Subtype Polymorphism

- Enables extensibility and reuse
  - In our example, we can extend the `Shape` hierarchy with no modification to the client of hierarchy, `DrawAll`
  - We can reuse `Shape` and `DrawAll`
- Subtype polymorphism enables the Open/closed principle
  - Software entities (classes, modules) should be open for extension but closed for modification
  - Credited to Bertrand Meyer

Benefits of Subtype Polymorphism

```java
abstract class Shape { public void draw(); }
class Circle extends Shape { … }
class Square extends Shape { … }
class Triangle extends Shape { … }
```

Extending the Java code requires no changes in `DrawAll`!
Thus, it is closed for modification.

Extending the C code triggers modifications in `DrawAll` (and throughout the code)!

Benefits of Subtype Polymorphism

- “Science” of software design teaches Design Patterns
- Design patterns promote design for extensibility and reuse
- Nearly all design patterns make use of subtype polymorphism!

Design Patterns

- A design pattern is a solution to a design problem that occurs over and over again
- Gang of Four (GoF) book: “Design Patterns: Elements of Reusable Object-Oriented Software”, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (the Gang of Four), Addison Wesley 1995
- Documents 23 still widely used design patterns
- Each pattern is described in terms of **intent**, **motivation**, **applicability**, **structure**, etc.

The Adapter Design Pattern

- From GoF Book
- **Intent**: Convert the interface of a class into another interface that the clients expect
- **Motivation**: A graphical editor that lets draw and arrange different shapes
  - All shapes need to conform to an interface
  - Line, Poly shapes are easy to manipulate
  - The Text shape is hard to manipulate, but there is a TextView toolkit (with a different interface)
Motivation

UML notation:
- Boxes are classes (Shape, TextView, Line, Text).
- T denotes composition (e.g., class Text has field text of type TextView).

Applicability

- To use an existing class whose interface does not match the one we need
  - TextView toolkit implements what we need, but its interface does not match ours
  - Adapter Text converts TextView’s GetExtent() into BoundingBox(), as required in Shape interface
- To upgrade to a new text manipulation toolkit, which has a different interface, we only need to change adapter Text
  - Editor remains closed to modification!

Structure: Object Adapter

Polymorphism, again

- Subtype polymorphism
  - What we just discussed… Code can use a subclass B where a superclass A is expected
  - Standard in object-oriented languages
- Parametric polymorphism
  - Code takes a type as parameter
    - Explicit parametric polymorphism
    - Implicit parametric polymorphism
  - Standard in functional programming languages!

Explicit Parametric Polymorphism

- Occurs in Ada, Clu, C++, Java, Haskell (type classes)
- There is an explicit type parameter
- Explicit parametric polymorphism is also known as genericity
- e.g. in C++:
  ```cpp
template<class V>
  class list_node {
    list_node<V>* prev;
    V* header;
  }

template<class V>
  class list { /*list_node<V> header; prev;*/
  }
```

Explicit Parametric Polymorphism

- Usually (but not always!) implemented by creating multiple copies of the generic code, one for each concrete type
  ```cpp
typedef list_node<int> int_list_node;
typedef list<int> int_list;
```
- Object-oriented languages usually provide both subtype polymorphism and explicit parametric polymorphism, which is referred to as generics
Explicit Parametric Polymorphism

- Generics are tricky...
- Consider this C++ code (uses the STL):
  ```cpp
  list<int> l;
  sort(l.begin(), l.end());
  ```
  Compiler produces around 2K of text of error messages, referring to code in the STL

  - The problem here is that the STL's `sort` requires a RandomAccessIterator, while the `list` container provides only a Bidirectional Iterator

In Java Bounded Types restrict instantiations by client

- Generic code can perform operations permitted by the bound
  ```java
  class MyList1<E extends Object> {
    void m(E arg) {
      arg.intValue(); //compile-time error: Object does not have intValue()
    }
  }
  ```
  ```java
  class MyList2<E extends Number> {
    void m(E arg) {
      arg.intValue(); //OK. Number has intValue()
    }
  }
  ```

Implicit Parametric Polymorphism

- Occurs in Scheme, Python and others
- There is no explicit type parameter, yet the code works on many different types!

  - Usually, there is a single copy of the code, and all type checking is delayed until runtime
  - If the arguments are of type as expected by the code, code works
  - If not, code issues a type error at runtime

Intro to Concurrency and Concurrency in Java

Read: Scott, Chapter 12.1-12.2
Concurrency

- **Concurrent program**
  - Any program is concurrent if it may have more than one active execution context --- more than one “thread of control”
- Concurrency is everywhere
  - A multithreaded web browser
  - An IDE which compiles while we edit
- What are the reasons for the (renewed) interest in concurrency in programming languages?

Concurrency and Parallelism

- **Concurrent** characterizes a system in which two or more tasks may be underway (at any point of their execution) at the same time
- A concurrent system is **parallel** if more than one task can be physically active at once
  - This requires more than one processor

Parallelism in Software Systems

- Arises at different granularity
  - From simple and small tasks, to large and complex tasks
- Instruction-level parallelism (ILP)
- Vector parallelism
  - Essentially, like Scheme’s map
- **Thread-level parallelism**
  - Tasks are now arbitrarily complex; concurrency is no longer hidden from programmer

Multiprocessor Machines

- Two broad categories of parallel architectures
  - Shared-memory machines
    - Those in which processors share common memory
  - Non-shared-memory machines
    - Those in which processors must communicate with messages

What Exactly is a Processor?

- For 30+ years, it used to be the single chip with a CPU, cache and other components
- Now, it can mean a single “device” with multiple chips; each chip can have multiple cores; each core can have multiple hardware threads. Also, subsets of the cores can share different levels of cache

What Exactly is a Processor?

- OS and programming language abstract away hardware complexity
- For us, programmers, “processor” means a thread/task of computation
  - Or the hardware that runs the thread of computation
- But as we saw many times in this class, abstraction (i.e., improved programmability) comes at a price!
Fundamentals of Concurrent Programming

- Two programming models for concurrency
  - Shared memory
    - Some program variables are accessible to multiple threads — threads have shared state
    - Threads communicate (interact) through shared state
  - Message passing
    - Threads have no shared state
    - One thread performs explicit send to transmit data to another

Communication
- More formally, refers to any mechanism that allows one thread to obtain information produced by another thread
- Explicit in message passing models
- Implicit in shared memory models

Synchronization
- Refers to any mechanism that allows the programmer to control the relative order of operations performed by different threads
- Implicit in message passing models
- Explicit and necessary in shared memory models

Shared Memory Model

- Programming language support for the shared memory model
  - Explicit support for concurrency
    - E.g., Java, C#: explicit threads, locks, synchronization, etc.
  - Libraries
    - C/C++: The POSIX include <pthreads.h>
    - Many types, macros and routines for threads, locks, other synchronization mechanisms
- We will take a closer look at Java

Threads
- Java has explicit support for multiple threads
- Two ways to create new threads:
  - Extend java.lang.Thread
    - Override "run()" method
  - Implement Runnable interface
    - Include a "run()" method in your class
- Starting a thread
  - new MyThread().start();
  - new Thread(runnable).start();
- Thread creation is abstracted away by Executor framework

Terminology
- Concurrent programming with shared memory is about managing shared mutable state
  - Shared state — memory locations that can be accessed by multiple threads
  - Mutable state — the value of a location could change during its lifetime
- Atomic action — action that executes on the machine as a single indivisible operation
  - E.g., read the value of variable \( i \) is atomic
  - E.g., write the value of variable \( i \) is atomic
  - E.g., \( i++ \) is not atomic

Next Class
- Concurrency in Java
  - Threads
  - What can go wrong with threads?
  - Synchronized blocks
  - The Executor framework
- Java Programming Assignment