How does Java work?
Announcements

- The course webpage is now up and available at the correct address:

  http://www.cs.mcgill.ca/~cs303/

- Notes should be posted soon on the course webpage.

- Because of a new McGill policy I discovered, the date to the second midterm will probably change.

- A regular event from 2h30 to 4h00 has been added to my schedule every Tuesday/Thursday. That means I might be 5 minutes late to class sometimes. I apologize for the inconvenience.
• Decomposition and abstraction are techniques to construct large programs that are easy to understand, maintain and modify.

• Abstraction allows us to ignore details and treat different objects as thought they were the same.

• Parameterization generalizes to wider applicability

• Four kinds of abstraction:
  – Procedural abstraction
  – Data abstraction
  – Iteration abstraction
  – Type hierarchy
What is object-oriented programming?

Object-oriented programming (OOP) is a computer programming paradigm that emphasizes the following aspects:

- The use of *objects* - objects are used extensively to modularize and structure the computer program.
- *Abstraction* - combining multiple smaller operations into a single unit that can be referred to by name.
- *Encapsulation* - separating implementation from interfaces.
- *Polymorphism* - using the same name to invoke different operations on objects of different data types.
- *Inheritance* - defining objects data types as extensions and/or restrictions of other object data types.

(from Wikipedia.org)
Why use objects?

- The common answer to this is modularity and data hiding.
- Though this answer is correct, the true reasons to use objects are much more elaborate.
- Requirements, by definition, have a tendency to change. This is especially true on the work market.
- When building your design, you must not only account for current features, but you build for the future.
- This means you must be able to safely modify your code with each new feature.
- Object-oriented programming allows you to do this by shifting responsibility.
Why use objects? (cont.)

- Shifting responsibility is best explained with an example:
  You are asked to program an application that will simulated the behavior of students going to class. Before each class, the locations of each classroom is posted on a board.

- If you had to program this in a procedural programming language, you could build a switch statement that would check the schedule of each student to find their classroom.

- This solution be compared to having a hall monitor check the schedule (switch statement) and direct each student to their class.

- The more intuitive solution would be to ask each student to check the schedule themselves.

- This can easily be done with Object-Oriented programming.
Why use objects? (cont.)

- In a O.O. solution, each student could be represented by an object. Before class, each object would receive the schedule (or a reference to it) and figure out where it needs to go.

- The behavior of each student is encapsulated within the object.

- If the specification were changed:
  
  A new type of student must be added to the simulator: "the visiting student". This student has the same behavior as a typical student, but might be required (depending on classroom) to pick up an evaluation form before class.

- In our procedural solution, the program must modify the large switch to integrate this behavior.

- In the O.O. solution, the programmer can extend the student object and add this new behavior to the visiting student object.
Why use objects? (cont.)

- In our example, the responsibility of finding the class is shifted from the scheduler to the student.
- The behavior of our student is encapsulated into the object.
- The original student object remains untouched, which can represent an important saving in debugging time.
Building blocks for abstraction

- Procedural abstraction – Method declaration
- Data abstraction – Property/Class declaration
- Type hierarchy – Class declarations Inheritance
Abstract Classes

• Java provides a mechanism for defining prescriptive classes by placing the keyword abstract in front of a class.

• The abstract class cannot be instantiated. However, subclasses of abstract classes can be instantiated.

• The abstract class can define abstract methods which subclasses must implement to be concrete (can be instantiated).

• The abstract class can define methods and attributes which all subclasses inherit.
Abstract class example: Point

• The classic example of an object is the Point class.

• There are numerous ways to store position over a 2D plane.

• The abstract Point class allows to define the methods a class must have to inherit from Point.
  – x() and y() are abstract
  – only the method signature is specified
  – a concrete subclass must provide an implementation

• It also allows us to define functions that should be common among all Points.
  – distance(Point other) is concrete
  – the implementation defines the notion of Euclidean distance, independent of the actual implementation of x() and y()
public abstract class Point {

    public abstract float x();
    public abstract float y();

    public float distance(Point other) {
        // Effect: Returns the Euclidian distance between two points

        float dx = other.x() / this.x();
        float dy = other.y() / this.y();

        return sqrt(dx*dx + dy*dy);
    }
}

Code for abstract class Point
public class CartesianPoint extends Point {

    private float _x, _y;

    public CartesianPoint (float x, float y) {
        _x = x;
        _y = y;
    }

    public float x() { return _x; }
    public float y() { return _y; }
}
public class PolarPoint extends Point {

    private float _rho, _theta;

    public PolarPoint (float rho, float theta) {
        _rho = rho;
        _theta = theta;
    }

    public float x() { return _rho * cos(_theta); }

    public float y() { return _rho * sin(_theta); }
}

Abstraction in Point

- x() y() distance() provide abstraction by specification: all Points have these methods.
- distance() provides parameterization abstraction: implementation applies to all Points.

```java
Point c = new CartesianPoint(2.5, -3);
Point p = new PolarPoint(2, 3.14);

c.distance(p) ?
```
Abstract Classes

• Tell the implementor of sub classes which functionality should be realized

• Tell the user which functionality is supported by concrete sub classes

• Serve as a contract: defines the interface a user can count on

• Implement functionality which all sub classes have in common (concrete methods)

• Use abstract super class when subclasses only re-uses part of the implementation

• Use concrete super class when a subclass is a true extension: + fields + methods
Why extend a class?

Two reasons:

• interface of sub class includes interface of super class
  – subclass looks like super class from the outside
  – used anywhere the super class is used

• implementation of sub class is similar to super class
  – re-use implementation code
  – inherit fields + method implementations

What if you don’t re-use implementation?
• Interfaces are like abstract classes, but
  – without any concrete methods (all methods are public and abstract)
  – without any fields other than static final fields (constants)
  – use keyword implements instead of extends

• A class can extend only one class, but can implement multiple interfaces
Point as Interface

```java
public interface Point {
    public float x();
    public float y();
    public float distance(Point other);
}

public class CartesianPoint implements Point { ...
public class PolarPoint implements Point { ...

    • No longer any default implementation for distance(Point other).
    • However, unrelated classes can provide Point functionality.
```
public interface WeightedObject {
    public float weight();
}

public interface ColoredObject {
    public float red();
    public float green();
    public float blue();
}

public class CartesianPoint implements Point, WeightedObject, ColoredObject {...

public class PolarPoint implements Point, WeightedObject, ColoredObject {...
Interfaces vs. Abstract Classes

- Abstract classes define a hierarchical relationship:
  - class B is a special type of class A
  - B has all of A’s fields and methods
- Interfaces define a form of behavior:
  - Class B behaves as specified by interface A
Why avoid multiple inheritance?

- Diamond problem:
  - A inherits from B and C
  - B and C both inherit from D
  - Does an object of type A have one or two versions of the field defined in D?

- Method dispatch ambiguity:
  - A inherits from B and C
  - B and C both define method foo()
  - Which method is executed on A.foo()?

- Any proposed solution has its problems
  - huge objects in C++
  - obfuscated control flow
Method Overriding

• New definition for method in a subclass: same parameter numbers, type and order.

• Appropriate method is called by dynamic binding: determine run time class of receiver object.

• Can not be determined at compile time:

    A and B inherit from C;
    C someObject;
    if (mouseClicked) {
        someObject = new A();
    } else {
        someObject = new B();
    }
    someObject.foo()
Method Overloading

- Methods with same name but different parameters:
  - different number
  - different type
  - different order

- Appropriate method is invoked based on the declared type of parameters.

- This can be determined at compile time.

- Method Overloading is considered *syntactic sugar*.

  *Syntactic sugar* is a term coined by Peter J. Landin for additions to the syntax of a language that do not affect its expressiveness but make it *sweeter* for humans to use.

  (Wikipedia.org)
class A {
    void foo (A a) {System.out.println("Class A Function A");}
    void foo (B b) {System.out.println("Class A Function B");}
}
class B extends A {
    void foo (A a) {System.out.println("Class B Function A");}
    void foo (B b) {System.out.println("Class B Function B");}
}
public class madness {
    public static void main (String args[]) {
        A aa = new A();
        A ab = new B();
        B bb = new B();
        aa.foo(aa); aa.foo(ab); aa.foo(bb);
        ab.foo(aa); ab.foo(ab); ab.foo(bb);
        bb.foo(aa); bb.foo(ab); bb.foo(bb);
    }
}

Quiz time: What is printed?
A aa = new A;
A ab = new B;
B bb = new B;

aa.foo(aa); aa.foo(ab); aa.foo(bb);
ab.foo(aa); ab.foo(ab); ab.foo(bb);
bb.foo(aa); bb.foo(ab); bb.foo(bb);

• For parameters, only consider the declared type.

• For receiver, care only about the instantiated type.

• To avoid confusion, don’t mix overloading with overriding.
Example of responsible overloading

```java
public abstract class Document {
    public print () {
        this.print(defaultPrinter)
    }
    // print this document
    public abstract print (Printer p);
}

public class pdfFile extends Document {
    public print (Printer p) {}  
}

public class psFile extends Document {
    public print (Printer p) {}  
}
```
An other example of abstraction

Building a drawing program, the object-oriented way.
Design Patterns

• By using these basic components, we can construct software architectures with re-usable components.

• Design patterns are
  – collections of data abstractions
  – programming tricks
  – common usage of object-oriented constructs
  – similar to architectural standards (living room)
  – a common language for software architectures

• Iteration abstraction is a design pattern.
Summary

- O.O. Programming allows programmers to shift responsibility.
- Java has a rich set of abstraction building blocks:
  - Abstract classes (concrete)
  - Interfaces
  - Overloading
  - Overriding
- Design patterns are built from basic constructs.
Tool of the day

- When I have the opportunity, I want to introduce new tools related to Java.
- It could be a JVM, a compiler, an editor, or any tool related to (or built in) Java.
- If you ever have a suggestion for the tool of the day, please feel free to send me an email.
jEdit

- jEdit is an open source programmer’s text editor.
- It’s built in Java, so you can use it on any platform: Windows, Unix, Mac-OS, Os/2, etc
- It’s a mature project (over 5 years old) and it is built to be extended.
  (The following list was taken from the program webpage).
  - Built-in macro language; extensible plugin architecture. Dozens of macros and plugins available.
  - Plugins can be downloaded and installed from within jEdit using the ”plugin manager” feature.
  - Auto indent, and syntax highlighting for more than 80 languages.
- You can download jEdit at:
  
  http://www.jedit.org/