

CSC 533: Organization of Programming Languages

Spring 2007

Object-Oriented Programming (OOP)

- abstract data types
- classes & objects
 - encapsulation + data hiding
 - C++: class vs. struct, fields & member functions, separate compilation
 - Java: fields & methods, javadoc
- inheritance
 - derived classes, inheriting/overriding methods
 - dynamic (late) binding
- abstract classes & interfaces

1

Data abstraction

pre 80's: focus on process abstraction

recently: data abstraction increasingly important

Object-Oriented Programming (OOP) is an outgrowth of data abstraction in software development

an abstract data type (ADT) requires

1. encapsulation of data and operations
cleanly localizes modifications
2. information hiding (hide internal representation, access through operations)
makes programs independent of implementation, increases reliability

ADT's in C++

Simula 67: first to provide direct support for data abstraction

- class definition encapsulated data and operations
- no information hiding

C++ classes are based on Simula 67 classes, extend C struct types

- data known as *fields*, operations known as *member functions*
- each instance of a C++ class gets its own set of fields (unless declared `static`)
- all instances share a single set of member functions

data fields/member functions can be:

- *public* visible to all
- *private* invisible (except to class instances)
- *protected* invisible (except to class instances & derived class instances)

can override protections by declaring a class/function to be a friend

3

Separate compilation in C++

can split non-templated class definitions into:

- interface (.h), implementation (.cpp) files
- allows for separate (smart) compilation
- enables programmer to hide implementation details

```
#ifndef _DIE_H
#define _DIE_H

class Die
{
public:
    Die(int sides = 6);
    int roll();
    int getNumberOfSides();
    int getNumberOfRolls();
private:
    int numSides;
    int numRolls;
    static bool initialized;
};

#endif
```

```
#include <cstdlib>
#include <ctime>
#include "Die.h"

bool Die::initialized = false;

Die::Die(int sides) {
    numSides = sides;
    numRolls = 0;

    if (initialized == false) {
        srand((unsigned)time(NULL));
        initialized = true;
    }
}

int Die::roll() {
    numRolls++;
    return (rand() % numRolls) + 1;
}

int Die::getNumberOfSides() {
    return numSides;
}

int Die::getNumberOfRolls() {
    return numRolls;
}
```

4

ADTs in Java

Java classes look very similar to C++ classes

- member functions known as *methods*
- each field/method has its own visibility specifier
- must be defined in one file, can't split into header/implementation
- javadoc facility allows automatic generation of documentation
- extensive library of data structures and algorithms
 - List: ArrayList, LinkedList
 - Set: HashSet, TreeSet
 - Map: HashMap, TreeMap
 - Queue, Stack, ...
- load libraries using `import`

```
public class Die {
    private int numSides;
    private int numRolls;

    public Die() {
        numSides = 6;
        numRolls = 0;
    }

    public Die(int sides) {
        numSides = sides;
        numRolls = 0;
    }

    public int getNumberOfSides() {
        return numSides;
    }

    public int getNumberOfRolls() {
        return numRolls;
    }

    public int roll() {
        numRolls = numRolls + 1;
        return (int)(Math.random()*numRolls + 1);
    }
}
```

note: within methods, can prefix field/method access with "this.", e.g., `this.numSides`

5

Object-based programming

object-based programming (OBP):

- solve problems by modeling real-world objects (using ADTs)
- a program is a collection of interacting objects

when designing a program, first focus on the data objects involved, understand and model their interactions

advantages:

- natural approach
- modular, good for reuse
 - usually, functionality changes more often than the objects involved

OBP languages: must provide support for ADTs

e.g., C++, Java, JavaScript, Visual Basic, Object Pascal

6

Object-oriented programming

OOP extends OBP by providing for inheritance

- can derive new classes from existing classes
- derived classes inherit data & operations from parent class, can add additional data & operations

advantage: easier to reuse classes,
don't even need access to source for parent class

pure OOP languages: all computation is based on message passing (method calls)

e.g., Smalltalk, Eiffel, Java

hybrid OOP languages: provide for interacting objects, but also stand-alone functions

e.g., C++, JavaScript

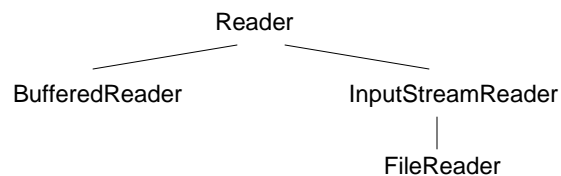
7

OOP big picture: inheritance + dynamic binding

necessary (but not sufficient) for OOP:

ADTs + inheritance + dynamic (late) binding

example: Java classes for reading character streams



BufferedReader & InputStreamReader are derived from (subclasses of) Reader

FileReader is derived from (subclass of) InputStreamReader

the derived class inherits all of the properties (data & methods) of the parent class

- an object of the derived class *IS_A* object of the parent class

8

OOP big picture: inheritance + dynamic binding

because an object of the derived class IS_A member of the parent class,
can pass it anywhere a parent object is expected.

```
public static void doSomething(Reader rdr) {  
    char ch = rdr.read();  
    . . .  
}
```

- could be called with a FileReader or a BufferedReader

note: parameter type cannot be determined at compile time – must be bound
dynamically

- at run-time, can determine which type of object is being passed in
- select the appropriate method

9

Java example: Person class

```
public class Person {  
    private String name;  
    private String SSN;  
    private char gender;  
    private int age;  
  
    public Person(string name, string SSN, char gender, int age) {  
        this.name = name;  
        this.SSN = SSN;  
        this.gender = gender;  
        this.age = age;  
    }  
  
    public void birthday() {  
        this.age++;  
    }  
  
    public String toString() {  
        return "Name: " + this.name + "\nSSN: " + this.SSN +  
            "\nGender: " + this.gender + "\nAge: " + this.age;  
    }  
}
```

```
Person somePerson = new Person("Bill", "123-45-6789", 'M', 19);  
somePerson.birthday();  
System.out.println(somePerson);
```

<i>data:</i> name social security number gender age ... <i>operations:</i> create a person have a birthday view person info ...

10

Extending a class

now suppose we want to represent information about students

Student = Person + additional attributes/capabilities

(1) could copy the Person class definition, rename as Student, add new features

advantages? disadvantages?

(2) could define the Student class so that it has a Person object embedded inside it

```
public class Student {  
    private Person self;  
    // ADDITIONAL DATA FIELDS  
  
    // METHODS DEFINING STUDENT BEHAVIOR  
}
```

advantages? disadvantages?

11

Extending via inheritance

(3) better solution – use inheritance

- define a Student class that is *derived* from Person
- a derived class inherits all data & functionality from its parent class
- in effect, a Student IS_A Person (with extra)

example: extending person

- data: *inherited person data*
school
level (1-12 high school, 13-16 college, 17- grad school)
...
- operations: *inherited person member functions*
constructor (with data values)
advance a level
...

12

Student class

```
public class Student extends Person {
    private String school;
    private int level;

    public Student(String name, String SSN, char gender,
        int age, String school, int level) {
        super(name, SSN, gender, age);
        this.school = school;
        this.level = level;
    }

    void advance() {
        this.level++;
    }
}
```

"extends" specifies that Student extends (is derived from) the Person class

Student constructor calls the Person constructor via "super" to initialize inherited fields, then initializes its own fields

note: only new data fields and member functions are listed, all data/functions from Person are automatically inherited

```
Student someStudent = new Student("Bill", "123-45-6789", 'M', 19, "Creighton", 13);
someStudent.birthday();
someStudent.advance();
System.out.println(someStudent);
```

13

private vs. protected

recall, data/functions in a class can be

- **private**: accessible only to methods of the class
- **public**: accessible to any program that includes the class

with inheritance, may want a level of protection in between

- **protected**: accessible to methods of the class
AND methods of derived classes

in our example, Person data fields were declared private

- Student class cannot directly access those data fields
- must instead go through Person methods (just like any other class/program)

serious drawback: when you design/implement a class, have to plan for inheritance

- future extensions to a class are not always obvious

14

Overriding methods

when a derived class adds data, existing functionality may need updating

- can override existing methods with new versions
e.g., Student class has additional data fields
→ toString method must be overridden to include these

```
public class Student extends Person {
    private String school;
    private int level;

    public Student(String name, String SSN, char gender,
        int age, String school, int level) {
        super(name, SSN, gender, age);
        this.school = school;
        this.level = level;
    }

    void advance() {
        this.level++;
    }

    public String toString() {
        return super.toString() + "\nSchool: " + this.school + "\nLevel: " + this.level;
    }
}
```

uses "super" to call toString method of parent class.

Note: must do this since Person data is private (instead of protected)

15

Polymorphism

different classes can have methods with the same names

- since methods *belong to* instances of the class, the compiler does not have any trouble determining which code to execute

```
Person somePerson = new Person("Chris", "111-11-1111", 'F', 20);
somePerson.birthday(); // calls Person's birthday
System.out.println(somePerson); // calls Person's toString

Student someStudent = new Student("Terry", "222-22-2222", 'M', 20, "Creighton", 14);
someStudent.birthday(); // calls Student's birthday (inherited from Person)
someStudent.advance(); // calls Student's advance
System.out.println(someStudent); // calls Student's toString (overriding Person)
```

16

IS_A relationship

important feature of inheritance:

an instance of a derived class is considered to be an instance of the parent class

```
a Student IS_A Person
a FileReader IS_A Reader
```

thus, a reference to a parent object can refer to a derived object

```
Person p = new Student("Terry", "222-22-2222", 'M', 20, "Creighton", 14);
```

the IS_A relationship is central to the utility of inheritance

- can define generic methods that work for a family of objects

```
public void foo(Person p) {
    . . .
    p.birthday();
    . . .
    System.out.println(p);
    . . .
}
```

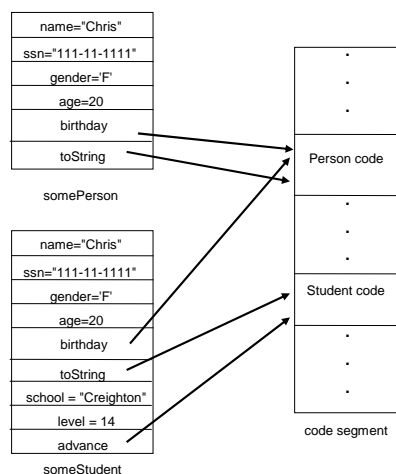
p.birthday()	no problem, since Person & Student share the same method
p.toString()	which method is called, Person's or Student's?

17

Dynamic (late) binding

in Java, method calls are bound dynamically

- in effect, the object type is determined at run-time & the appropriate method called
- this is implemented by storing within the object a reference for each method



the reference stores the address of the corresponding code for that class

when a method is called, the corresponding reference is used to find the correct version of the code

- `System.out.println(p)` will use the Student version of `toString` if the object is really a Student

note: in C++, member functions calls are bound statically by default

- would always call parent version
- if want dynamic binding, declare "virtual"

18

Abstract classes

there are times when you want to define a class hierarchy, but the parent class is incomplete (more of a placeholder)

- e.g., the Statement class from HW3
- want to be able to talk about a hierarchy of statements (including Assignment, Output, If), but there is no "Statement"

an *abstract class* is a class in which some methods are specified but not implemented

- can provide some concrete fields & methods
- the keyword "abstract" identifies methods that must be implemented by a derived class
- you can't create an object of an abstract class, but it does provide a framework for inheritance

19

Statement class

```
public abstract class Statement {
    public abstract void read(SourceCode program);
    public abstract void execute(VariableTable variables);
    public abstract Statement.Type getType();
    public abstract String toString();

    public static enum Type { OUTPUT, IF, ASSIGNMENT }

    public static Statement getStatement(SourceCode program) {
        Statement stmt;

        Token lookAhead = program.peek();
        if (lookAhead.toString().equals("output")) {
            stmt = new Output();
        }
        else if (lookAhead.toString().equals("if")) {
            stmt = new If();
        }
        else if (lookAhead.getType() == Token.Type.IDENTIFIER) {
            stmt = new Assignment();
        }
        else {
            System.out.println("SYNTAX ERROR: Unknown statement type");
            System.exit(0);
            stmt = new Output(); // KLUDGE
        }

        stmt.read(program); // POLYMORPHISM!
        return stmt; // POLYMORPHISM!
    }
}
```

Statement class provides framework for derived classes

- static enum Type and getStatement method are provided as part of the abstract class
- the other methods MUST be implemented exactly in a derived class

20

Derived statement classes

derived classes define specific statements (assignment, output, if)

- each will have its own private data fields
- each will implement the methods appropriately
- as each new statement class is added, must update the Type enum and the getStatement code

for HW4, will extend & add new types of statements

- counter-driven repeat loop
- input statement
- subroutines w/ parameters & local variables

```
public class Assignment extends Statement {
    private Token vbl;
    private Expression expr;

    public void read(SourceCode program) { ... }
    public void execute(VariableTable vbIs) { ... }
    public Statement.Type getType() { ... }
    public String toString() { ... }
}
```

```
public class Output extends Statement {
    private Expression expr;

    public void read(SourceCode program) { ... }
    public void execute(VariableTable vbIs) { ... }
    public Statement.Type getType() { ... }
    public String toString() { ... }
}
```

```
public class If extends Statement {
    private Expression expr;
    private ArrayList<Statement> stmts;

    public void read(SourceCode program) { ... }
    public void execute(VariableTable vbIs) { ... }
    public Statement.Type getType() { ... }
    public String toString() { ... }
}
```

21

Interfaces

an abstract class combines concrete fields/methods with abstract methods

- it is possible to have no fields or methods implemented, only abstract methods
- in fact this is a useful device for software engineering
define the behavior of an object without constraining implementation

Java provides a special notation for this useful device: an *interface*

- an interface simply defines the methods that must be implemented by a class
- a derived class is said to "implement" the interface if it meets those specs

```
public interface List<E> {
    boolean add(E obj);
    void add(index i, E obj);
    void clear();
    boolean contains (E obj);
    E get(index i);
    int indexOf(E obj);
    E set(index i, E obj);
    int size();
    ...
}
```

an interface is equivalent to an abstract class with only abstract methods

note: can't specify any fields, nor any private methods

22

List interface

interfaces are useful for grouping generic classes

- can have more than one implementation, with different characteristics

```
public class ArrayList<T> implements List<T> {
    private T[] items;
    . . .
}

public class LinkedList<T> implements List<T> {
    private T front;
    private T back;
    . . .
}
```

- using the interface, can write generic code that works on any implementation

```
public numOccur(List<String> words, String desired) {
    int count = 0;
    for (int i = 0; i < words.size(); i++) {
        if (desired.equals(words.get(i))) {
            count++;
        }
    }
}
```

in Java, a class can implement more than one interface

- e.g., `ArrayList<E>` implements `List<E>`, `Collection<E>`, `Iterable<E>`, ...

but can extend *at most* one parent class **WHY?**

23