CSC 533: Programming Languages

Spring 2017

Language evolution: C → C++ → Java

- C
  history, design goals, features, top-down design
- C++
  history, design goals, features, object-based design
- Java
  history, design goals, features, object-oriented design

C: early history

C was developed by Dennis Ritchie at Bell Labs in 1972

- designed as an in-house language for implementing UNIX
  - UNIX was originally implemented by Ken Thompson in PDP-7 assembly
  - when porting to PDP-11, Thompson decided to use a high-level language
  - considered the B language, a stripped-down version of BCPL, but it was untyped & didn't provide enough low-level machine access
  - decided to design a new language that provided high-level abstraction + low-level access
- the UNIX kernel was rewritten in C in 1973

became popular for systems-oriented applications, and also general problem solving (especially under UNIX)

- for many years, there was no official standard (other than Kernighan & Ritchie's 1978 book)
- finally standardized in 1989 (ANSI C or C89) and again in 1999 (C99)

C design goals

wanted the language to support

- systems programming (e.g., the implementation of UNIX)
- applications programming

as a result, the language had to

- support low-level operations but also high-level abstractions
- be close to the machine but also portable
- yield efficient machine code but also be expressive/readable

the dominant paradigm of the time was imperative programming

- a program is a collection of functions
- statements specify sequences of state changes
- supported top-down design

3

Program structure

a C program is a collection of functions

- libraries of useful functions can be placed in files and loaded using `#include`
- to be executable, a program must have a `main` method
- functions must call upward, or else place prototype above to warn the compiler

```
#include <stdio.h>
#include <string.h>

void oldMacVerse(char*, char*);

int main() {
    oldMacVerse("cow", "moo");
    return 0;
}

void oldMacVerse(char* animal, char* sound) {
    printf("Old MacDonald had a farm, E-I-E-I-O.\n");
    printf("And on that farm he had a %s, E-I-E-I-O.\n", animal);
    printf("With a %s-%s here, and a %s-%s there,\n",
           sound, sound, sound, sound);
    printf("  here a %s, there a %s, everywhere a %s-%s.\n",
           sound, sound, sound, sound);
    printf("Old MacDonald had a farm, E-I-E-I-O.\n");
}
```

4

Variables & bindings

primitive types

- `short`, `int`, `long`
  only guaranteed that 2 bytes ≤ sizeof(`short`) ≤ sizeof(`int`) ≤ sizeof(`long`)
  can specify integers in octal & hexadecimal, can declare to be unsigned
- `float`, `double`, `long double`
- `char` represents characters using ASCII codes (1 byte) – really another int type

types are bound statically

- all variable declarations must occur at the start of a block
- somewhat strongly typed, but loopholes exist

memory is bound…

- statically for global variables
- stack-dynamically for local variables
- heap-dynamically for malloc/free

5

Input & control

simple input via scanf

- must allocate space for the string
- can access chars using [] since an array

same control structures as C++/Java

- if/else, switch
- while, do-while, for
- break, continue

also has goto

- to support old-school programming

```
#include <stdio.h>
#include <string.h>

int isPalindrome(char*);

int main() {
    char input[20];
    printf("Enter a word: ");
    scanf("%s", &input);

    if (isPalindrome(input)) {
        printf("%s is a palindrome\n", input);
    }
    else {
        printf("%s is NOT a palindrome\n", input);
    }

    return 0;
}

int isPalindrome(char* word) {
    int len = strlen(word);

    int i;
    for (i = 0; i < len/2; i++) {
        if (word[i] != word[len-i-1]) {
            return 0;
        }
    }
    return 1;
}
```

6

#define

you can define constants using #define

- this is a preprocessor directive
- the first step in compilation is globally replacing each constant with its value

note: constants are NOT the same as constants in Java

```
#include <stdio.h>
#include <string.h>

#define MAX_LENGTH 20
#define BOOLEAN int
#define TRUE 1
#define FALSE 0

BOOLEAN isPalindrome(char*);

int main() {
    char input[MAX_LENGTH];
    printf("Enter a word: ");
    scanf("%s", &input);

    if (isPalindrome(input)) {
        printf("%s is a palindrome\n", input);
    }
    else {
        printf("%s is NOT a palindrome\n", input);
    }

    return 0;
}

BOOLEAN isPalindrome(char* word) {
    int len = strlen(word);

    int i;
    for (i = 0; i < len/2; i++) {
        if (word[i] != word[len-i-1]) {
            return FALSE;
        }
    }
    return TRUE;
}
```

7

Function parameters

all parameter passing is by-value

- but can achieve by-reference passing through pointers

- get the address of the variable using &
- pass the address to the function as parameter
- then dereference the address using *

```
#include <stdio.h>

void getValues(int*, int*);
void process(int*, int*);
void display(int, int);

int main() {
    int x, y;
    GetValues(&x, &y);
    Process(&x, &y);
    Display(x, y);

    return 0;
}

void getValues(int* a, int* b) {
    printf("Enter two numbers: ");
    scanf("%d%d", a, b);
}

void process(int* a, int* b) {
    if (*a > *b) {
        int temp = *a;
        *a = *b;
        *b = temp;
    }
}

void display(int a, int b) {
    printf("%d + %d = %d\n", a, b, (a+b));
}
```

8

C arrays

by default, C array allocation is:

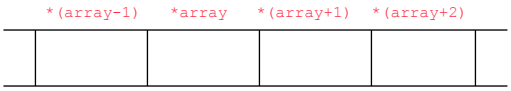
- static (allocated on the stack at compile time), if a global variable
- fixed stack-dynamic (size is fixed at compile time, memory is allocated on the stack during run time), if a local variable
- unlike Java, there is no length field associated with an array and no bounds checking – you must keep track!

arrays and pointers are linked in C

- can think of an array as a pointer to the first element

```
int counts[NUM_LETTERS];              // counts ≡ &counts[0]
```

- array indexing is implemented via pointer arithmetic: `array[k] ≡ *(array+k)`



9

Array example

because of the size limit, storing an arbitrary number of items is ugly

- must set a max size
- as you read in items, must keep count and make sure don't exceed the limit
- must then pass the size around with the array in order to process

- note: could have written

```
int* nums
```

instead of

```
int nums[]
```

```
#include <stdio.h>
#define MAX_SIZE 20

void getNums(int[], int*);
int smallest(int[], int);

int main() {
    int numbers[MAX_SIZE];
    int count = 0;

    getNums(numbers, &count);
    printf("The smallest number is %d\n",
           smallest(numbers, count));

    return 0;
}

void getNums(int nums[], int* cnt) {
    int nextNum;

    printf("Enter numbers (end with -1): ");
    scanf("%d", &nextNum);
    while (nextNum != -1 && *cnt < MAX_SIZE) {
        nums[*cnt] = nextNum;
        (*cnt)++;
        scanf("%d", &nextNum);
    }
}

int smallest(int nums[], int cnt) {
    int small = nums[0];
    int i;
    for (i = 1; i < cnt; i++) {
        if (nums[i] < small) {
            small = nums[i];
        }
    }
    return small;
}
```

10

Data structures

can define new, composite data types using `struct`

- `struct { … }` defines a new structure
- `typedef … NAME;` attaches a type name to the struct

note: a struct is NOT a class

- there is no information hiding (i.e., no `private`)
- there are no methods

```
#include <stdio.h>
#include <math.h>

typedef struct {
    int x;
    int y;
} Point;

double distance(Point, Point);

int main() {
    Point pt1;
    Point pt2;

    pt1.x = 0;
    pt1.y = 0;

    pt2.x = 3;
    pt2.y = 4;

    printf("Distance = %f\n",
            distance(pt1, pt2));

    return 0;
}

double distance(Point p1, Point p2) {
    return sqrt(pow(p1.x - p2.x, 2.0) +
                pow(p1.y - p2.y, 2.0));
}
```

11

Dynamic memory

by default, data is allocated on the stack

- to allocate dynamic memory (from the heap), must explicitly call `malloc`
- `malloc` returns a `(void*)`
- must cast to the appropriate pointer type
- when done, must explicitly deallocate memory using `free`

```
#include <stdio.h>
#include <stdlib.h>

void getNums(int[], int);
int smallest(int[], int);

int main() {
    int* numbers;
    int count = 0;

    printf("How many numbers? ");
    scanf("%d", &count);
    numbers = (int *)malloc(count * sizeof(int));

    getNums(numbers, count);
    printf("The smallest number is %d\n",
           smallest(numbers, count));

    free(numbers);

    return 0;
}

void getNums(int nums[], int cnt) {
    int i;
    printf("Enter the numbers: ");
    for (i = 0; i < cnt; i++) {
        scanf("%d", &nums[i]);
    }
}

int smallest(int nums[], int cnt) {
    int i, small = nums[0];
    for (i = 1; i < cnt; i++) {
        if (nums[i] < small) {
            small = nums[i];
        }
    }
    return small;
}
```

12

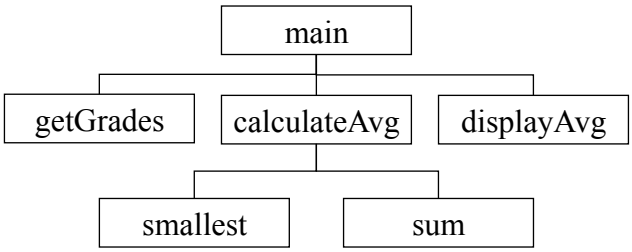
Top-down design

the dominant approach to program design in the 70's was top-down design

- also known as *iterative refinement*

general idea:

- focus on the sequence of tasks that must be performed to solve the problem
- design a function for each task
- if the task is too complex to be implemented as a single function, break it into subtasks and repeat as necessary



13

C++ design

C++ was developed by Bjarne Stroustrup at Bell Labs in 1984

- C++ is a superset of C, with language features added to support OOP

design goals:

1. support object-oriented programming (i.e., classes & inheritance)
2. retain the high performance of C
3. provide a smooth transition into OOP for procedural programmers

backward compatibility with C was key to the initial success of C++

- could continue to use existing C code; learn and add new features incrementally

however, backward compatibility had far-reaching ramifications

- C++ did add many features to improve reliability & support OOP
- but, couldn't remove undesirable features
  ➔ it is a large, complex, and sometimes redundant language

14

Added reliability features: pass by-reference

in C, all parameter passing was by-value

```
void reset(int num) {
    num = 0;
}
```

```
int x = 9;
reset(x);
printf("x = %d", x);
```

- but, could get the effect of by-reference via pointers

```
void reset(int* num) {
    *num = 0;
}
```

```
int x = 9;
reset(&x);
printf("x = %d", x);
```

C++ introduced cleaner by-reference passing (in addition to default by-value)

```
void reset(int & num) {
    num = 0;
}
```

```
int x = 9;
reset(x);
cout << "x = " << x;
```

15

Added reliability features: constants

in C, constants had to be defined as preprocessor directives

- weakened type checking, made debugging more difficult

```
#define MAX_SIZE 100
```

C++ introduced the `const` keyword

- can be applied to constant variables (similar to `final` in Java)
  the compiler will catch any attempt to reassign

```
const int MAX_SIZE = 100;
```

- can also be applied to by-reference parameters to ensure no changes
  safe since const; efficient since by-reference

```
void process(const ReallyBigObject & obj) {
    . . .
}
```

16

Other reliability features

in C, there was no boolean type – had to rely on user-defined constants

- C++ `bool` type still implemented as an int, but provided some level of abstraction

```
#define FALSE 0
#define TRUE 1

int flag = TRUE;
```

```
bool flag = true;
```

in C, there was no string type – had to use char arrays & library functions

- C++ `string` type encapsulated basic operations inside a class

```
char* word = "foo";

printf("%d", strlen(word));
```

```
string word = "foo";

cout << word.length();
```

17

Other reliability features

in C, memory was allocated & deallocated using low-level system calls

- C++ introduced typesafe operators for allocating & deallocating memory

```
int* a = (int*)malloc(20*sizeof(int));
…
free(a);
```

```
int* a = new int[20];
…
delete[] a;
```

in C, all variable declarations had to be at the beginning of a block

- C++ declarations can appear anywhere, can combine with initialization

```
if (inputOK) {
  int num1, num2;
  displayInstructions();
  num1 = getValue();
  num2 = getValue();
  …
}
```

```
if (inputOK) {
  displayInstructions();
  int num1 = getValue();
  int num2 = getValue();
  …
}
```

18

ADT's in C++

in order to allow for new *abstract data types*, a language must provide:

1. encapsulation of data + operations (to cleanly localize modifications)
2. information hiding (to hide internal details, lead to implementation-independence)

Simula 67 was first language to provide direct support for data abstraction

- class definition encapsulated data and operations; but 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`

19

C++ classes

C++classes followed the structure of structs (i.e., records)

- for backward compatiblity, structs remained
- but only difference: in a struct, fields/functions are `public` by default
  in a class, fields/functions are `private` by default

```
struct Point {
    int x;
    int y;
};
```

```
struct Point pt;
pt.x = 3;
pt.y = 4;
```

```
class Point {
  public:
    Point(int xCoord, int yCoord) {
      x = xCoord;  y = yCoord;
    }

    int getX() const { return x; }

    int getY() const { return y; }

  private:
    int x;
    int y;
};
```

```
Point pt(3, 4);
```

20

Memory management

as in C, local variables in C++ are bound to memory stack-dynamically

- allocated when declaration is reached, stored on the stack
- this includes instances of classes as well as primitives

can use `new` & `delete` to create heap-dynamic memory

- requires diligence on the part of the programmer

- must explicitly delete any heap-dynamic memory, or else garbage references persist (there is no automatic garbage collection)
- in order to copy a class instance with heap-dynamic fields, must define a special copy constructor
- in order to reclaim heap-dynamic fields, must define a special destructor

21

Example: card game

can separate class definition into 2 files

- allows for separate (smart) compilation

```
// Card.h
//////////////////////////////////////

#ifndef _CARD_H
#define _CARD_H

using namespace std;

const string SUITS = "SHDC";
const string RANKS = "23456789TJQKA";

class Card {
  public:
    Card(char r = '?', char s = '?');
    char GetSuit() const;
    char GetRank() const;
    int GetValue() const;
  private:
    char rank;
    char suit;
};

#endif
```

```
// Card.cpp
//////////////////////////////////////////////

#include <iostream>
#include <string>
#include "Card.h"
using namespace std;

Card::Card(char r, char s) {
    rank = r;
    suit = s;
}

char Card::GetRank() const {
    return rank;
}

char Card::GetSuit() const {
    return suit;
}

int Card::GetValue() const {
    for (int i = 0; i < RANKS.length(); i++) {
        if (rank == RANKS.at(i)) {
            return i+2;
        }
    }
    return -1;
}
```

22

Example (cont.)

classes/functions can be templated

- idea later adopted by Java generics
- here, vector class is similar to Java ArrayList

```
// DeckOfCards.h
//////////////////////////

#ifndef _DECKOFCARDS_H
#define _DECKOFCARDS_H

#include <vector>
#include "Card.h"
using namespace std;

class DeckOfCards {
  public:
    DeckOfCards();
    void Shuffle();
    Card DrawFromTop();
    bool IsEmpty() const;
  private:
    vector<Card> cards;
};

#endif
```

```
// DeckOfCards.cpp
//////////////////////////////////////////////////////////

#include <string>
#include "Die.h"
#include "Card.h"
#include "DeckOfCards.h"
using namespace std;

DeckOfCards::DeckOfCards() {
  for (int suitNum = 0; suitNum < SUITS.length(); suitNum++) {
    for (int rankNum = 0; rankNum < RANKS.length(); rankNum++) {
      Card card(RANKS.at(rankNum), SUITS.at(suitNum));
      cards.push_back(card);
    }
  }
}

void DeckOfCards::Shuffle() {
  Die shuffleDie(cards.size());

  for (int i = 0; i < cards.size(); i++) {
    int randPos = shuffleDie.Roll()-1;
    Card temp = cards[i];
    cards[i] = cards[randPos];
    cards[randPos] = temp;
  }
}

Card DeckOfCards::DrawFromTop() {
  Card top = cards.back();
  cards.pop_back();
  return top;
}

bool DeckOfCards::IsEmpty() const {
  return (cards.size() == 0);
}
```

23

Example (cont.)

following the convention from C:

`main` is a stand-alone function, automatically called if present in the file

```
#include <iostream>
#include <string>
#include "Card.h"
#include "DeckOfCards.h"
using namespace std;

int main() {
    DeckOfCards deck1, deck2;

    deck1.Shuffle();
    deck2.Shuffle();

    int player1 = 0, player2 = 0;
    while (!deck1.IsEmpty()) {
        Card card1 = deck1.DrawFromTop();
        Card card2 = deck2.DrawFromTop();

        cout << card1.GetRank() << card1.GetSuit() << " vs. "
             << card2.GetRank() << card2.GetSuit();

        if (card1.GetValue() > card2.GetValue()) {
            cout << ":  Player 1 wins" << endl;
            player1++;
        }
        else if (card2.GetValue() > card1.GetValue()) {
            cout << ":  Player 2 wins" << endl;
            player2++;
        }
        else {
            cout << ": Nobody wins" << endl;
        }
    }
    cout << endl <<"Player 1: " << player1
         << "  Player2: " << player2 << endl;

    return 0;
}
```

24

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

25

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

26

C++ example: Person class

data: name
social security number
gender
age
...

operations: create a person
have a birthday
display person info
...

```
class Person {
  public:
    Person(string nm, string id, char sex, int yrs) {
        name = nm;  ssn = id;  gender = sex;  age = yrs;
    }

    void Birthday() {
        age++;
    }

    void Display() {
        cout << "Name: " << name << endl << "SSN : << ssn << endl
             << "Gender: " << gender << endl << "Age: " << age << endl;
    }

  private:
    string name, ssn;
    char gender;
    int age;
};
```

```
Person somePerson("Bjarne", "123-45-6789", 'M', 19);
somePerson.Birthday();
somePerson.Display();
```


Student class: extending Person

```
class Student : public Person {
  public:
    Student(string nm, string id, char sex, int yrs,
            string sch, int lvl) : Person(nm, id, sex, yrs) {
        school = sch;  grade = lvl;
    }

    void Advance() {
        grade++;
    }

    void Display() {
        Person::Display();
        cout << "School: " << school << endl
             << "Grade: " << grade << endl;
    }

  private:
    string school;
    int grade;
};
```

specifies that Student is derived from Person, public fields stay public

Student constructor initializes its own data fields, but must call the Person constructor to initialize inherited data

can override a function from the parent class, but still access using the scope resolution operator ::

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

```
Student someStudent("Bjarne", "123-45-6789", 'M', 19, "Creighton", 13);
someStudent.Birthday();
someStudent.Advance();
```

note: `private` data fields are hidden even from derived classes (e.g., name cannot be accessed in Person)

- can access if defined to be `protected` instead

IS_A relationship

important relationship that makes inheritance work:

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

  a `Student` IS_A `Person`
  an `ifstream` IS_A `istream` IS_A `iostream`

- thus, a pointer to a parent object can point to a derived object

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

- since by-reference parameters are really just pointers to objects, this means you can write generic functions that work for a family of objects

```
void Foo(Person & p) {              // can call with a Person or Student
    . . .
    p.Birthday();                   // calls Person::Birthday on either
    . . .
}
```

29

IS_A relationship & dynamic binding

BUT… for the IS_A relationship to work in general, member functions must be bound *dynamically*

- if the parent & derived classes both have member functions with the same name, how can the function know which type of object until it is passed in?

```
void Foo(Person & p)
{
    . . .
    p.Birthday();        // calls Person::Birthday
    . . .
    p.Display();         // calls ???
}

--------------------------

Foo(somePerson);         // would like for Foo to call Person::Display
Foo(someStudent);        // would like for Foo to call Student::Display
```

UNFORTUNATELY… *by default* C++ binds member functions statically

- compiler looks at the parameter type in the function, `Person`, and decides that `p.Display()` must be calling `Person::Display`

30

Dynamic binding & virtual

to specify dynamic binding, individual member functions in the parent class must be declared "virtual"

```
class Person
{
  public:
    . . .
    virtual void Display() {
        cout << "Name: " << name << endl << "SSN : << ssn << endl
             << "Gender: " << gender << endl << "Age: " << age << endl;
    }
  private:
    . . .
};
```

```
Foo(somePerson);          // calls Person::Display in Foo

Foo(someStudent);         // calls Student::Display in Foo
```

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

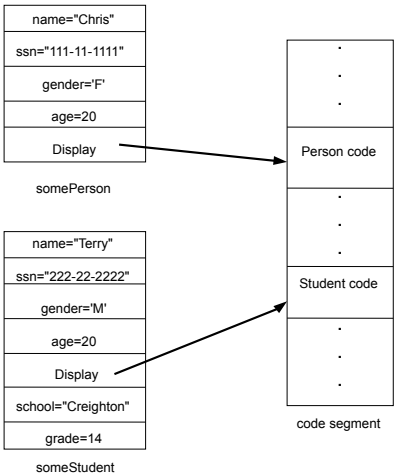
note: Java performs dynamic binding automatically

31

Implementing virtual member functions

with static binding, the address of the corresponding code is substituted for the call

with dynamic binding, an extra pointer field must be allocated within the object



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

when a virtual member function is called, the corresponding pointer in that object is dereferenced to find the correct version of the code

Note: each call to a virtual function implies one level of indirection

→ static binding more efficient

32

Java

Java was developed at Sun Microsystems, 1995

- originally designed for small, embedded systems in electronic appliances
- initial attempts used C++, but frustration at limitations/pitfalls

recall: C++ = C + OOP features

the desire for backward compatibility led to the retention of many bad features

desired features (from the Java white paper):

simple	object-oriented	network-savvy
interpreted	robust	secure
architecture-neutral	portable	high-performance
multi-threaded	dynamic	

note: these are desirable features for any modern language (+ FREE)

→ Java has become very popular, especially when Internet related

33

ADTs in Java

recall: 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
- recall: objects are heap-dynamic

```
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;
  }
}
```

34

Inheritance in Java

achieve inheritance by "extending" a class

- can add new methods or override existing methods
- can even remove methods (but generally not considered good design – WHY?)

```
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;
  }
}
```

recall: Java uses "super" to call a constructor or method from the parent class

here, call the super constructor to initialize the private fields

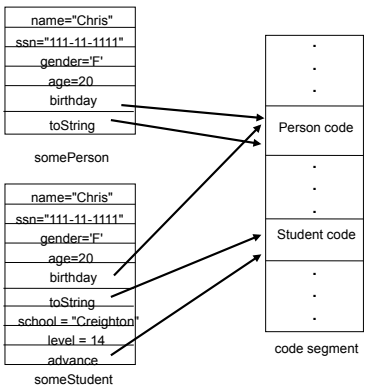
also, call the super.toString to print the private fields

35

Dynamic (late) binding

in Java, all method calls are bound dynamically

- this was not the default in C++, required declaring methods to be "virtual"
- the implementation of dynamic binding is the same as in C++



since dynamic binding is used, each method call will refer to the most specific version

```
public void foo(Person p) {

    . . .
    p.birthday();
    . . .
    System.out.println(p);
    . . .
}
```

i.e., `foo(somePerson)` will call the `Person` version, while `foo(someStudent)` will call the `Student` version

36

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 HW2
- 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

note: you can define abstract classes in C++, but in a very kludgy way

37

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
  - can implement in different ways (e.g., ArrayList, LinkedList) but still write methods that work on all implementations (e.g., Collections.sort)

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

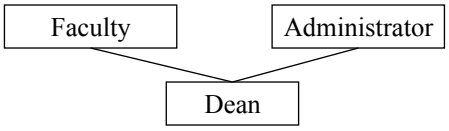
38

Multiple interfaces

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?



suppose a Dean class is defined that implements two interfaces

- the Dean class must implement the union of the listed methods – OK!

but if inheritance were used, conflicts could occur

- what if both parent classes had fields or methods with the same names?
- e.g., would `super.getRaise()` call the `Faculty` or the `Adminstrator` version?

C++ allows for multiple inheritance but user must disambiguate using ::

- Java simply disallows it as being too tricky & not worth it

39

Non-OO programming in Java

despite its claims as a pure OOP language, you can write non-OO code same as C++

- static methods can call other static methods

for large projects, good OO design leads to more reliable & more easily maintainable code

```
/**
 *  Simple program that prints a table of temperatures
 *
 *  @author     Dave Reed
 *  @version    3/10/17
 */
public class FahrToCelsius {
  private static double FahrToCelsius(double temp) {
    return 5.0*(temp-32.0)/9.0;
  }

  public static void main(String[] args) {
    double lower = 0.0, upper = 100.0, step = 5.0;

    System.out.println("Fahr\t\tCelsius");
    System.out.println("----\t\t-------");

    for (double fahr = lower; fahr <= upper; fahr += step) {
      double celsius = FahrToCelsius(fahr);
      System.out.println(fahr + "\t\t" + celsius);
    }
  }
}
```

40

Functional programming in Java 8

Java 8 was released in 2014, introduced function programming constructs

- lambda expressions (unnamed functions)
- first-class functions
- streams with filter, map, reduce, …

```
int sumAbove = 0;
for (int n : nums) {
    if (n > 50) {
        sumAbove += n;
    }
}

-------------------------------------------------------------------------

int sumAbove = nums.stream().filter(n -> n > 50).reduce(0, Integer::sum);
```

let's go to the source for functional programming: LISP/Scheme

41