What is a Program?

- A *recipe* for doing something
- A precise set of instructions
  - Generally from a limited set of available instructions
- Like the rules for a game, or how to build something, or directions to your house, or a recipe for macaroni and cheese

Tic-Tac-Toe

- Draw a big #
- First player draws an X in one square
- Second player draws an O in some square.
- Continue until someone has three letters in a row or diagonal, or all the squares are filled

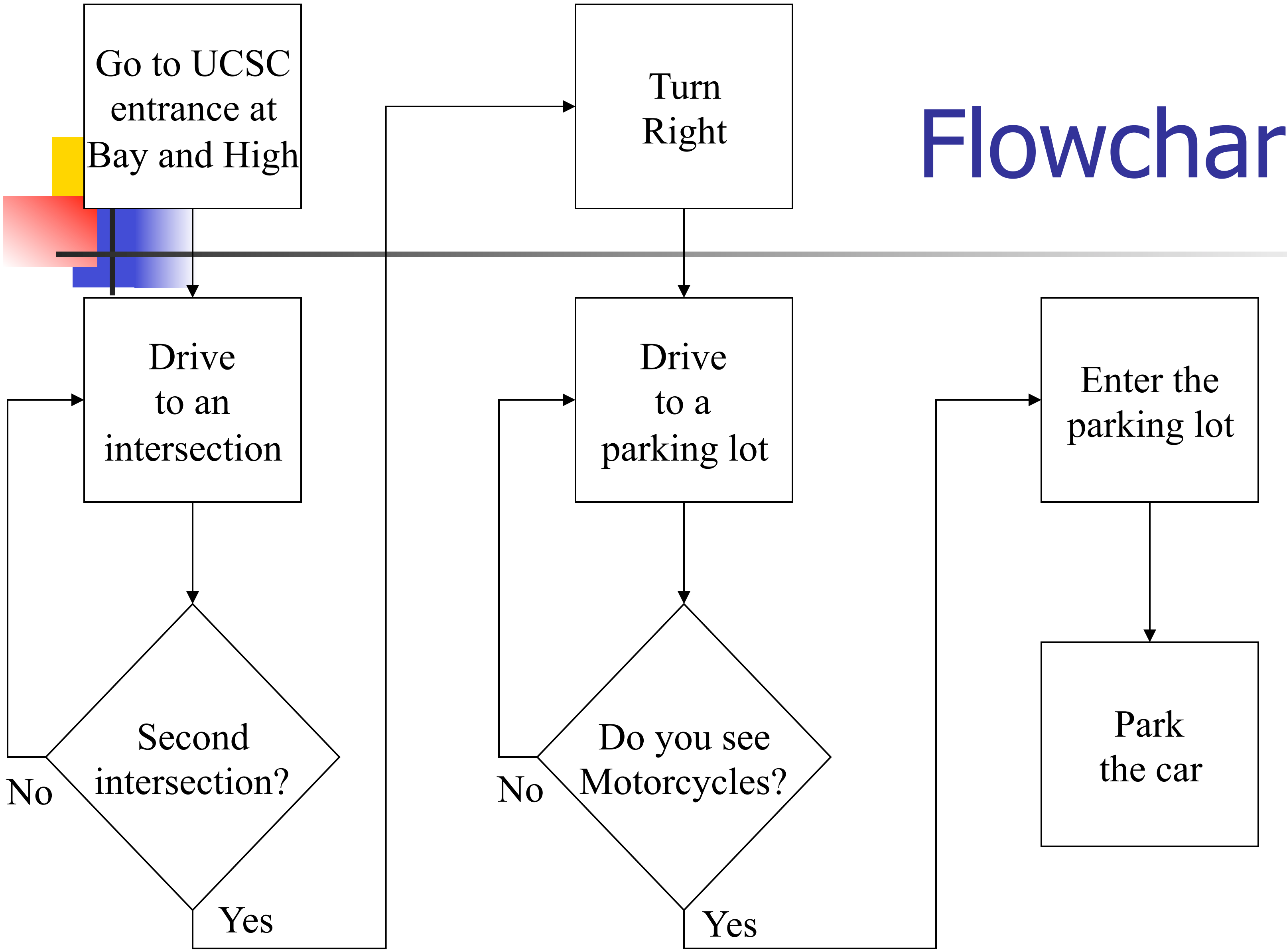
Macaroni and Cheese

- Boil some water
- Open the box
- Remove the cheese packet
- Put the macaroni in the boiling water
- Boil for 7 minutes
- Drain water
- Add butter and cheese powder (from packet)
- Stir

Directions to my house

- Go to the UCSC campus entrance at Bay and High Street
- Enter the campus
- Turn right at the second opportunity
- Stay to the left as you follow the road
- Turn right into the parking lot with the motorcycles
- Park in an empty spot

Flowchart



Go to UCSC entrance at Bay and High

Drive to an intersection

Second intersection?

No

Yes

Turn Right

Drive to a parking lot

Do you see Motorcycles?

No

Yes

Enter the parking lot

Park the car

What can computers understand?

- A computer probably couldn’t follow the instructions we just gave for playing tic-tac-toe or getting to someone’s house
- Different computers have different basic operations they can perform, like addition, subtraction, draw a line, etc.
    - Generally lower level than the abstractions we usually use
- A Compiler converts our programs into language a computer can understand

Algorithm

- A sequence of instructions
- The sequence of instructions must terminate
- The instructions are precise
    - Unambiguous and uniquely interpreted
- The instructions are effective
    - Doable, and in a finite amount of time
- There are inputs and outputs

Example: Making change in dime/ penny land

- Assumptions
  - Only dimes and pennies
  - Cost < $1
- Inputs and Outputs
  - Input: price of item
  - Output: number of dimes and pennies to return from $1 payment

Change-making continued

- Algorithm
  - Subtract *price* from $1 and store the result in *change*
  - Divide *change* by 10, and store the integer result in *dimes*
  - Divide *change* by 10 and store the remainder in *pennies*
  - Print out the values in *dimes* and *pennies*
  - Halt

Software Life Cycle

Problem Analysis and Specification

- What needs to be done

Design

- How it should be done
- Creation of a solution (algorithm)

Implementation

- Implement the algorithm as a program

Verification

- Does the program do what it is supposed to do
- Does the program not do what it is not supposed to do

Maintainance

- Change what the program does
- Includes both bug fixes and modifications

Problem Analysis and Specification

- What it does:
  - Clearly defines problem - what is/is not being solved
  - Refines imprecise problem to one that is solvable given existing constraints
  - Constitutes an agreement on what is to be done
  - May discover problems
    - Inconsistency, vagueness, impossibility
  - Leads the way to the solution
  - May contain desirable and optional items

Problem Analysis and Specification (cont.)

- What it does (cont.):
  - Should be specific enough to be testable, so you know if/when the problem has been solved
  - Often done inadequately
- What it doesn't do:
  - Specify how to solve the problem
- Important parts of a problem specification
  - A list of inputs
  - A list of constants
  - A list of outputs

Problem Analysis and Specification (cont.)

- Examples:
  - Yes: Should calculate change in dime/penny land
  - No: Should be fast
  - Yes: Should run in less than 10 seconds
  - No: Should use quicksort
- For the programming assignments, I will provide a problem specification
  - This specification may be incomplete
  - It is your job to analyze and understand the specification and refine it as necessary

Design

- What it does:
    - Clearly specifies how the problem will be solved
    - Allows developers to determine what resources will be needed to solve the problem
    - Hopefully solves all problems that could arise in the development of the software component
    - Is used as a recipe for doing the actual coding
- What it doesn't do:
    - A design is not code and does not contain any code.
        - May contain *pseudocode*
    - A design is not specific to any language, although it usually is specific to a type of language

Design (cont.)

A software design typically has 3 parts:

1. Identification of the data objects that are required to solve the problem
2. Identification of the operations that must be applied to the data objects in order to solve the problem
3. Construction of a detailed sequence of steps (an algorithm) that specifies how the operations can be applied to the data objects to solve the problem

Implementation (cont.)

- Once the design is complete, coding can begin
  - Given a good design, this should be very straightforward
  - All hard problems should have been worked out in the design stage
  - New hard problems should send the project (temporarily) back to the design stage

Implementation (cont.)

- Good code should be
  - Correct
  - Readable and Understandable
  - Modifiable
  - Ideally: Reusable

Implementation

- Good programs should be:
  - Well structured
    - Break programs into meaningful parts
    - Strive for simplicity and clarity
  - Well documented (commented))
    - Good comments before each program and/or function
    - Good comments before each important part of a program/ function
    - Use meaningful identifiers (function and variable names)
  - Aesthetically pleasing
    - Space things out and use blank lines between logical blocks
    - Use alignment and indentation to emphasize relationships

Verification

- Each program and subprogram should be tested against it's requirements
  - To see that it does what it is supposed to do
  - To make sure that it does not do what it is not supposed to do
- Tests should include correct and incorrect inputs
  - Even nonsense inputs
- Regression tests
  - Make sure that new changes don't break old functionality

Maintenance

- Bugs are found that need to be fixed
- Requirements change
- Components are reused
- Enhancements are made
- Generally accomplished by repeating the first four steps
- Most software development effort is maintenance

Example: Problem Specification/ Analysis

Problem: Write a program that, given diameter of a circle, computes the area and circumference

Description: Compute and output the area and circumference of a circle given the diameter.

Input: Diameter of the circle

Outputs: Area and Circumference of the circle

Constants: Pi, and maybe formulas for area and circumference of circles

Example: Design

- **Data objects**:
  - **Variables**:
    - Real: *diameter, circumference, area, radius*
  - **Constants**:
    - Real: **pi**
  - **Operations**:
    - *radius* = *diameter* / 2
    - *circumference* = 2 * **pi** * *radius*
    - *area* = **pi** * $\textit{radius}^{2}$

Example: Design (cont.)

Algorithm:

1. Get diameter from user
2. Calculate radius
3. Calculate circumference
4. Calculate area
5. Print out values of circumference and area

Example: Implementation

```
// Calculate area and circumference
import tio.*;                    // use the package tio
class CalcAC {
  Public static void main () {
    int diameter;
    double radius, circumference, area, pi;
    Scanner in = new Scanner(System.in);
    pi = 3.14159625;
    system.out.println(“Diameter: “);
    diameter = in.nextInt();
}
```

Example: Implementation (cont.)

```
    radius = diameter / 2;
    circumference = 2 * pi * radius;
    area = pi * radius * radius;

    system.out.println(“Circumference = “);
    system.out.println(circumference);
    system.out.println(“Area = “);
    system.out.println(area);
  }
}
```

Example: Verification

- Run the above program with:
  - No input (shouldn't be allowed)
  - Negative input (shouldn't be allowed)
  - Positive input (should work)
  - Fractional input (should work)
  - Very large input (should work)
  - What else?

What’s So Special About Java?

- Java is (relatively) new
  - Based heavily on preexisting languages: C, C++, Pascal, Ada, …
  - Integrates good ideas from all of these, plus some new ones
  - Integrated with new technologies (especially the web)
- Java is well structured
  - Everything is contained in classes
  - One class defines one object
  - One class definition per file
  - File names match class names

What’s So Special About Java?

- Java is Object Oriented
  - Everything is an object
- Java is (relatively) easy to use
  - Uniform, simple, well structured
- Java is web aware
  - Integrated with the web
  - Good communication primitives
  - Integrated into web browsers

What’ s So Special About Java?

- Applets!
  - Can be executed automatically by the browser within a web page
    - My code is automatically downloaded and executed on your machine
- Java has integrated GUI functionality
  - Good graphical interface functionality fully integrated into the language.
- Java is platform independent
  - Runs on Windows, Unix, Mac, and just about everything else

Reminder

- Read Chapter 1
- Read the summary and review questions
  - If anything is confusing, reread that part of the chapter
- Work some or all of the exercises
- Note: Labs start next week