CS 304 Lecture 1

Introduction to Data Structures

Xiwei Wang, Ph.D.

Assistant Professor

Department of Computer Science

Northeastern Illinois University

Chicago, Illinois, 60625

25 August 2016

Course Information

- CS 304 section 1 and 2
- Instructor: Xiwei ("She-Way") Wang
- Email: xwang9@neiu.edu
- Lecture:
  - Section 1: TR 1:40pm-2:55pm, CBM 158
  - Section 2: TR 5:40pm-6:55pm, CBM 114
- Office Hours: MW 4:30pm-5:30pm, T 3pm-5pm, LWH 3061
- Textbook: *Object-Oriented Data Structures Using Java*, Third Edition by Nell Dale, Daniel T. Joyce, and Chip Weems. Jones and Bartlett Publishers, Inc., 2012. ISBN: 978-1-4496-1354-9.

Grading

- There are five main components of your final grade:

(H) Homework Assignments	35%
(A) Attendance	8%
(V) Office Hour Visit	2%
(M) Midterm Exam	25%
(F) Final Exam	30%

- Your Weighted Average (WA) = 0.35(H) + 0.08(A) + 0.02(V) + .25(M) + .3(F)

Grading

- Course grade:
  - Your course grade is determined by the chart below.

Weighted Average	Course Grade
90% or higher	A
80% – 90%	B
70% – 80%	C
60% – 70%	D
0% – 60%	F

Homework assignments (35%)

- There will be bi-weekly homework assignments distributed throughout the semester. Each assignment may contain programming questions as well as short answers.
- All assignments are assigned on every other Monday and are due the second Thursday (by 11:59pm).
  - You will lose one point for every hour your assignment is late (starting at 12:00am which is one minute after the deadline). No submission is accepted after 11:59am Friday.

Homework assignments (35%)

- For example, the first HW assignment is assigned on Monday, August 29th and it is due Thursday, September 8th by 11:59pm. If you submit your assignment later than 11:59pm but before 11:59am on Friday, September 9th, your submission is still accepted but you will get late penalty according to the above rule. **Submissions later than 11:59am on Friday will NOT be accepted.**

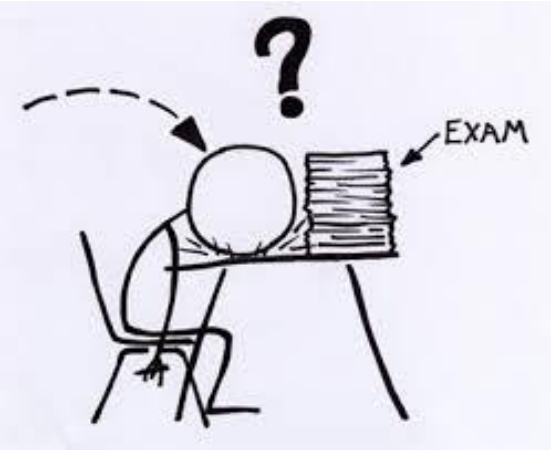
- Wrong submissions, including but not limited to, assignments belong to other courses, submissions with no answers but only homework specs, etc., will result in a ZERO for the corresponding assignments. **No resubmission is accepted.**

Attendance (10%)

- Attendance: 8% classroom + 2% office hour visit
  - Beginning the 2nd week, attendance will be taken during class.
  - You should visit me during my office hours at least once **before the midterm exam** to get the Office Hour Visit credit.

Examinations (25% + 30% = 55%)

- Midterm: 25%
  - Thursday, October 13th, during lecture.
- Final: 30%
  - Section 1: Tuesday, December 13th, 2:00-3:50pm, normal class location.
  - Section 2: Tuesday, December 13th, 6:00-7:50pm, normal class location.
  - Comprehensive, but focusing on new material.
- All exams are closed-book, closed-notes!



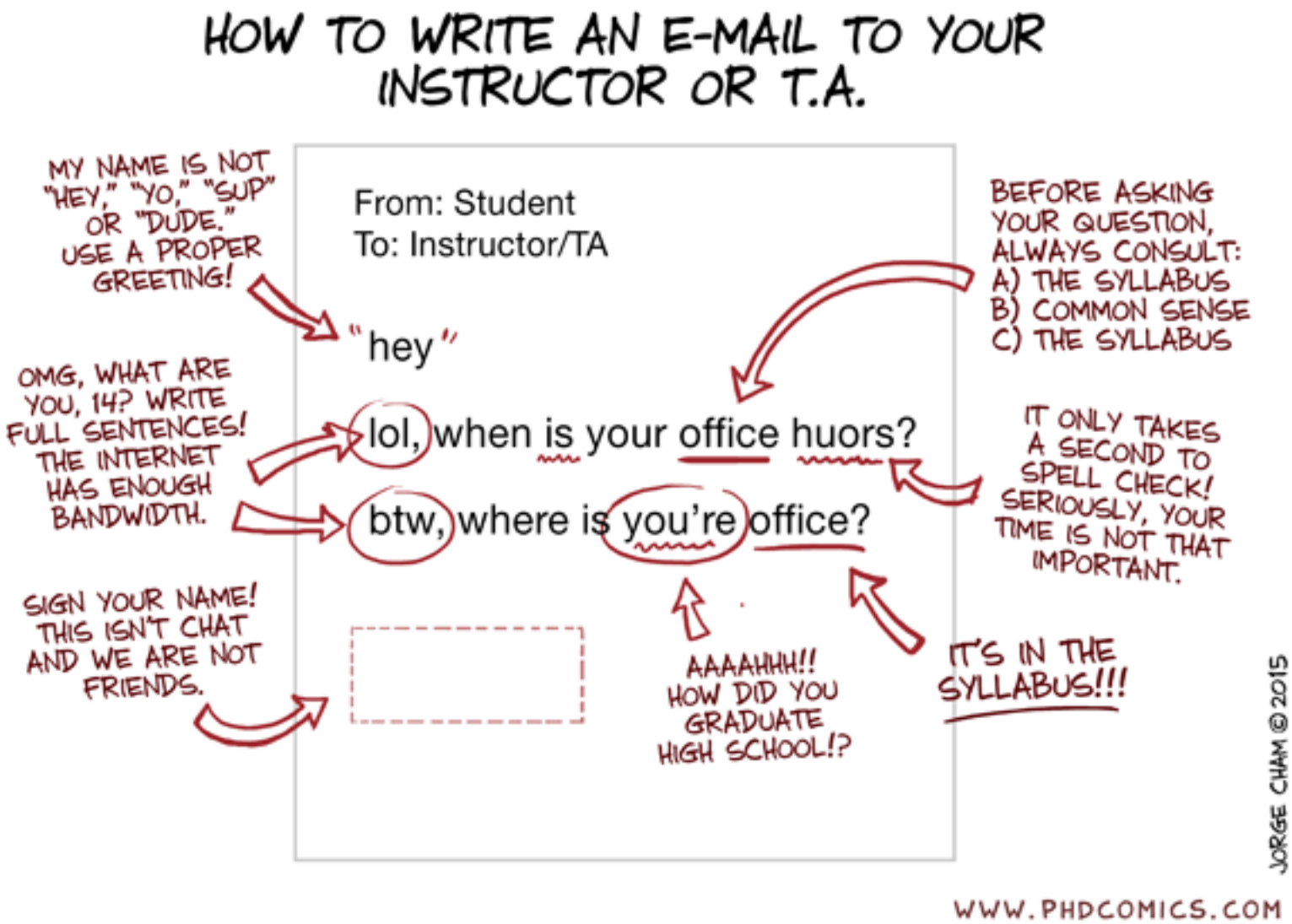
Cheating and plagiarism

- **All** assignments are individual work.
- Do not share work or code with other students.
- Do not show your work or code to other students.
- Do not use code you found online.
- In general: no code from anyone but you, and instructor.
- No signing in for others.
- No notes on exams!
- Minimum penalty: zero on assignment.
- Subsequent offenses (in any class!) may be punished more severely.
- If you have a question about whether it's okay to use something, talk to me.

Other rules

- You are expected to attend and pay attention to all classes.
- Be alert and engaged, do not distract others.
- Be respectful: harassment and discrimination will not be tolerated.
- NO cell phones, audible pagers, or any electronic devices are allowed during lecture or tests.
- There will be NO extra credit projects to improve your grade. Do NOT ask me for extra work to improve your grade.
- If you have a documented disability that requires accommodation, contact me as soon as possible. You will need a letter from the Student Disability Services on campus.
- Do not send me your code in an email. Questions regarding code should be discussed during office hours.
- Per university policy, I do not discuss grades via email.

Writing emails



Course goals

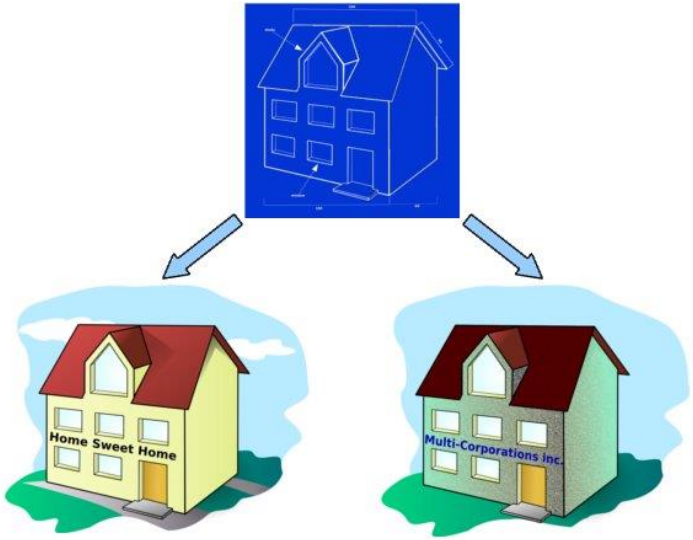
- Some of the goals of this course (full list in syllabus):
  - Classes
  - Interfaces
  - Abstract data types:
    - stacks, queues, lists, trees, heaps, and graphs.
  - Searching and sorting
  - Recursion

Object orientation

- What is an object?
  - Objects can represent "real-world" entities such as bank accounts, cars, persons, etc.
- What does an object represent?
  - Information: we say the objects have attributes.
  - Behavior: we say the objects have responsibilities.
- Objects are easy to implement, modify, and test for correctness.
- Object-oriented classes, when designed properly, are very easy to reuse.

Classes, objects, and applications

- An object is an instantiation of a class.
- A class defines the structure of its objects.
- A class definition includes variables (data) and methods (actions) that determine the behavior of an object.
- Example: the `Date` class.



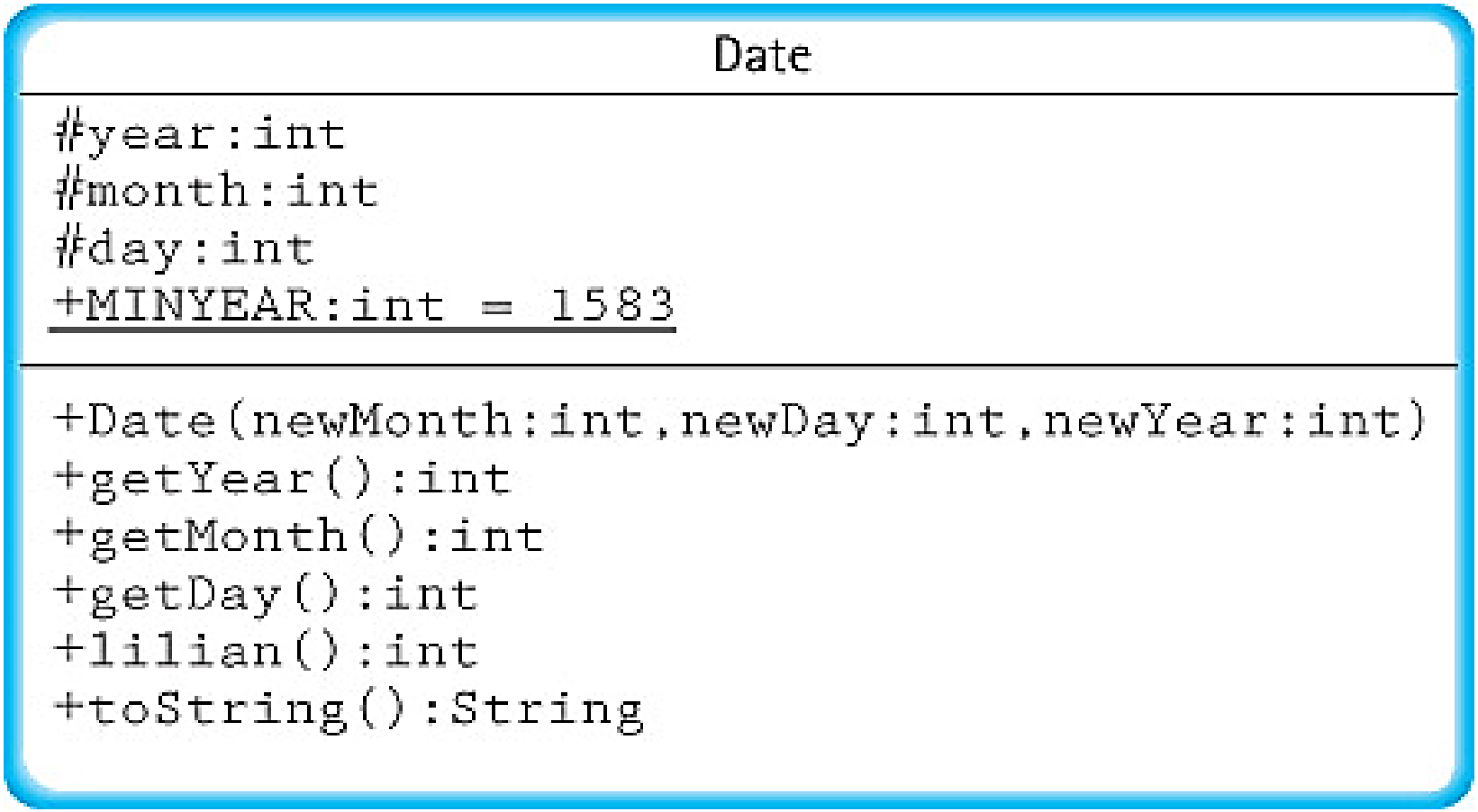
Classes, objects, and applications

```
public class Date
{
  protected int year;
  protected int month;
  protected int day;
  public static final
      int MINYEAR = 1583;

  // Constructor
  public Date(int newMonth,
              int newDay,
             int newYear)
  {
    month = newMonth;
    day = newDay;
    year = newYear;
   }

  // Observers
  public int getYear()
  {
    return year;
  }
  public int getMonth()
  {
    return month;
  }

  public int getDay()
  {
    return day;
  }

  public int lilian()
  {
    // Returns the Lilian Day
    // Number of this date.
  }

  public String toString()
  // Returns this date as a
  // String.
  {
    return(month + "/" + day
                 + "/" + year);
  }
}
```

Java access control modifiers

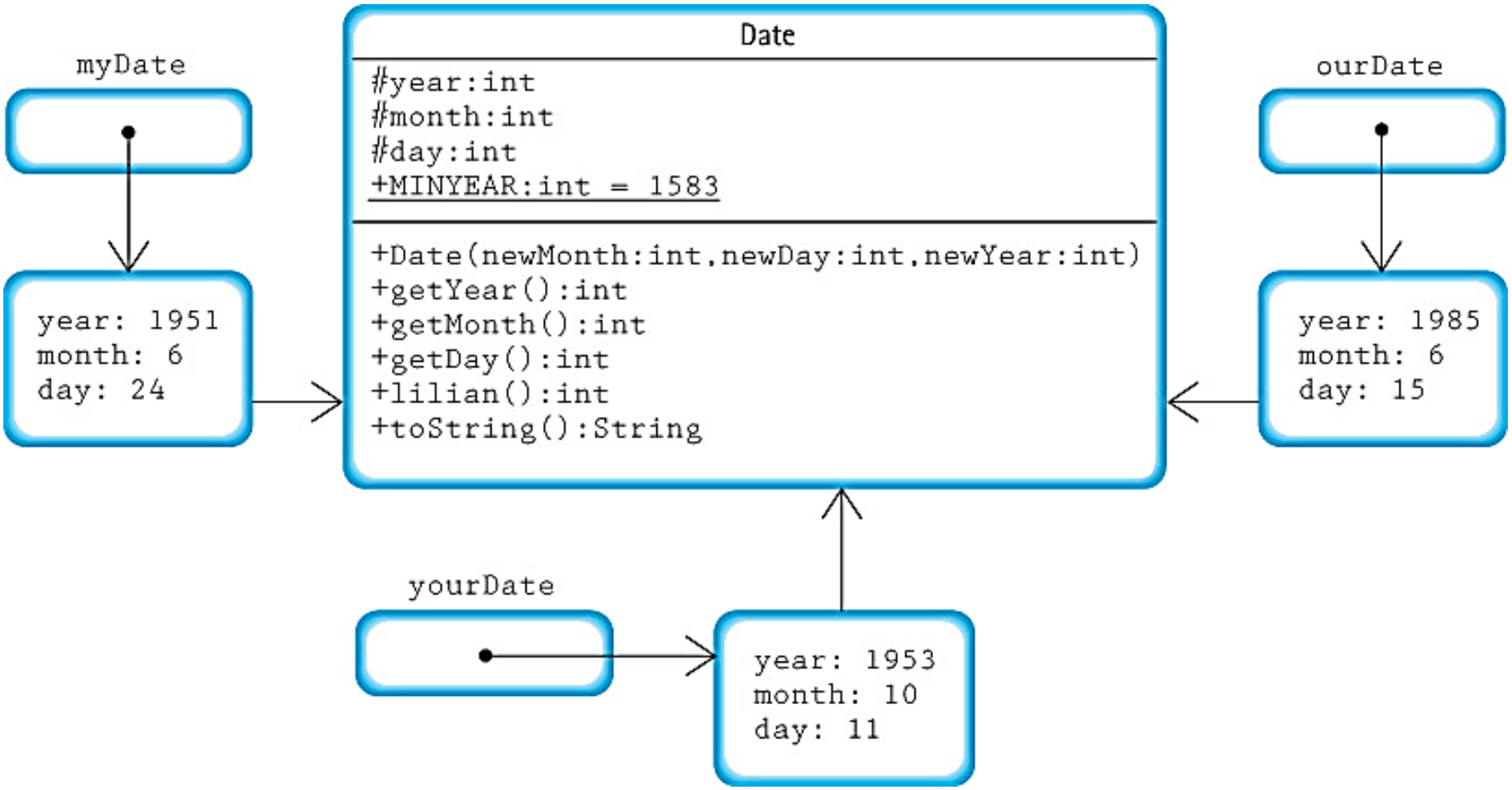
	Within the Class	Within Subclasses in the Same Package	Within Subclasses in Other Packages	Everywhere
public	X	X	X	X
protected	X	X	X	
package	X	X		
private	X			

Class diagram for the `Date` class



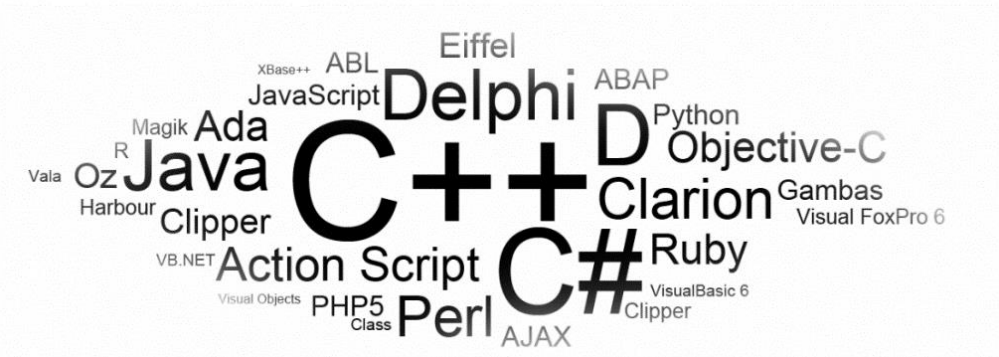
Objects

```
Date myDate = new Date(6, 24, 1951);
Date yourDate = new Date(10, 11, 1953);
Date ourDate = new Date(6, 15, 1985);
```



Applications

- An object-oriented application is a set of objects working together, by sending each other messages, to solve a problem.
- In object-oriented programming a key step is identifying classes that can be used to help solve a problem.
- An example – using our **`Date`** class to solve the problem of calculating the number of days between two dates.



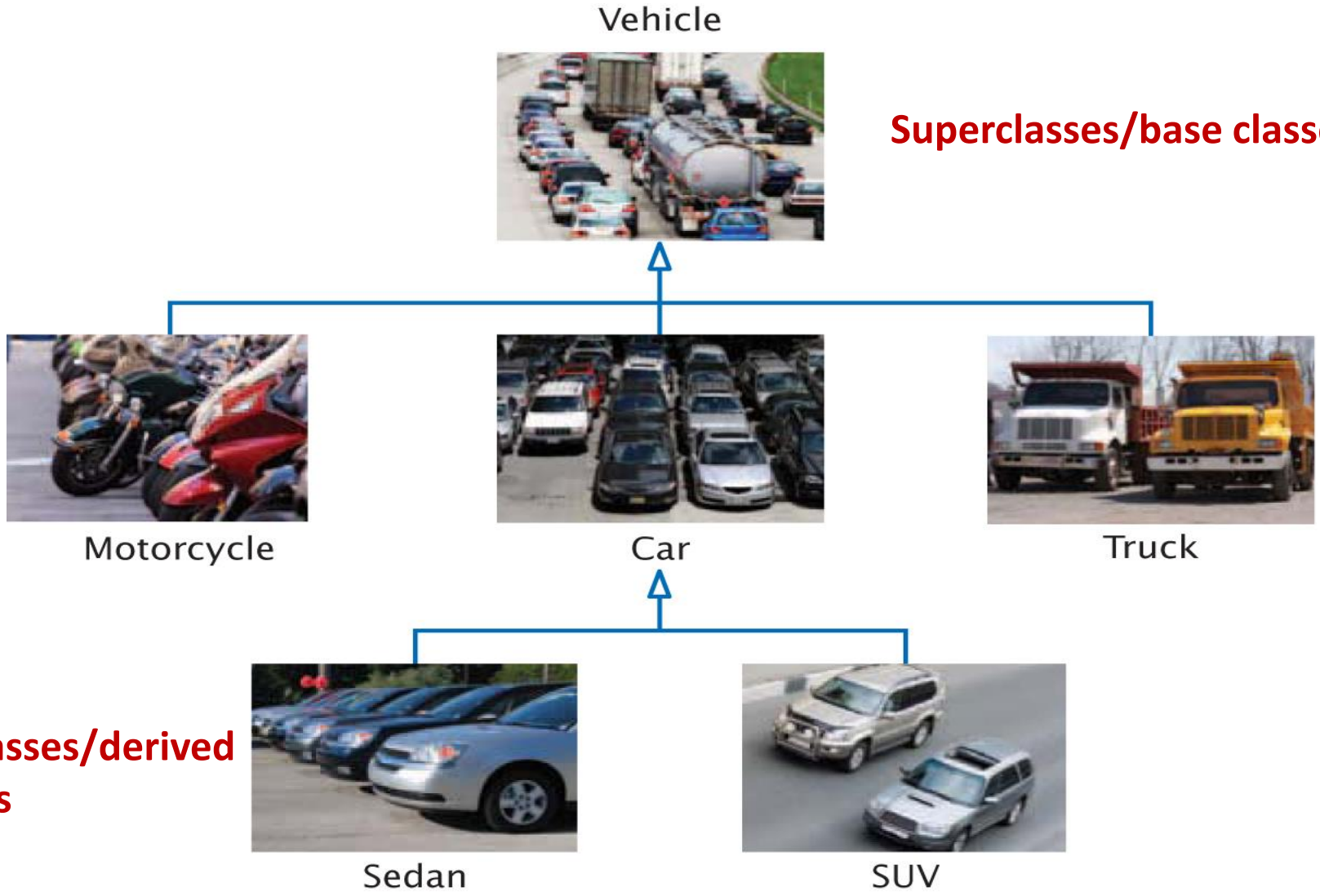
DaysBetween design

```
display instructions
prompt for and read in info about the first date
create the date1 object
prompt for and read in info about the second date
create the date2 object
if dates entered are too early
    print an error message
else
    use the date.lilian method to obtain the
       Lilian Day Numbers
    compute and print the number of days
       between the dates
```

Organizing classes

- During object-oriented development, hundreds of classes can be generated or reused to help build a system.
- Two of the most important ways of organizing Java classes are
  - Inheritance: classes are organized in an "is-a" hierarchy.
  - Packages: let us group related classes together into a single named unit.

Inheritance



Superclasses/base classes

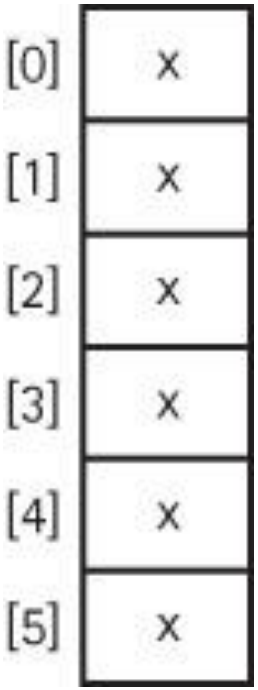
Subclasses/derived classes

Data structures

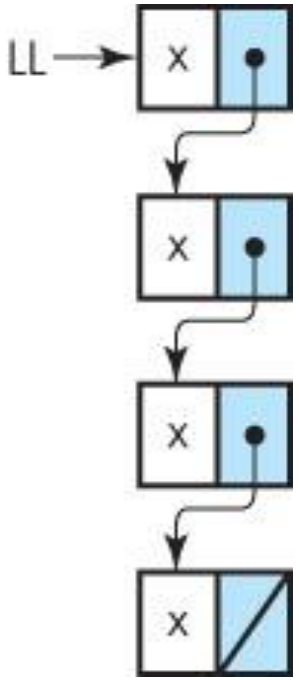
- The way you view and structure the data that your programs manipulate greatly influences your success.
- A language's set of primitive types (Java's are **`byte`**, **`char`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, and **`boolean`**) are not sufficient, by themselves, for dealing with data that have many parts and complex interrelationships among those parts.
- Data structures provide this ability.

Implementation dependent structures

Array

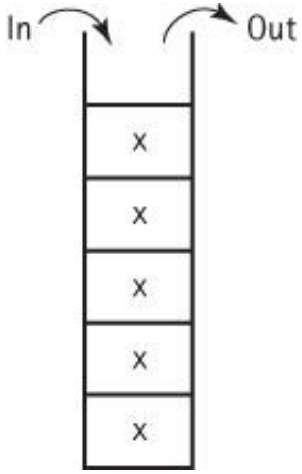


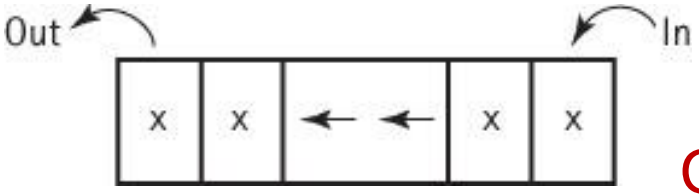
Linked List



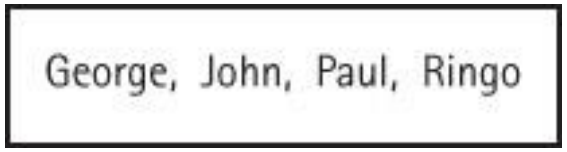
Implementation independent structures

Stack



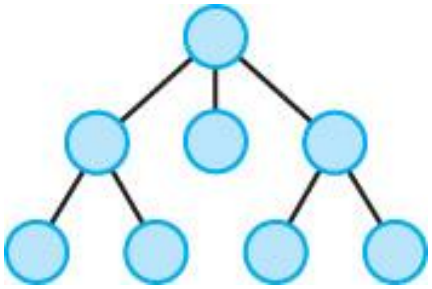


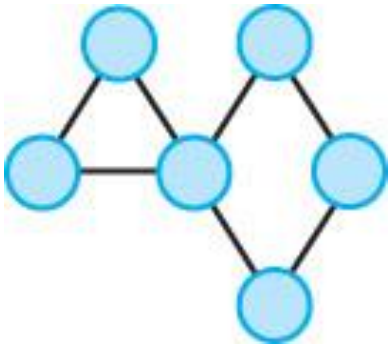
Queue



Sorted List

Tree



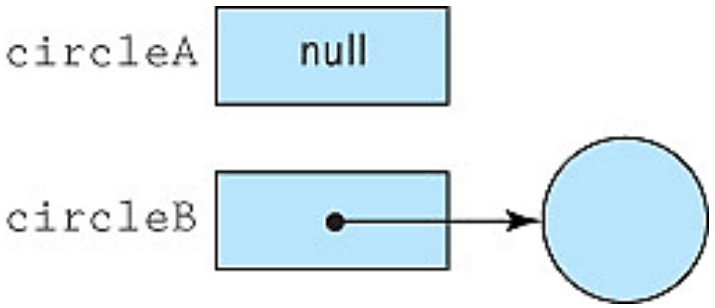


Graph

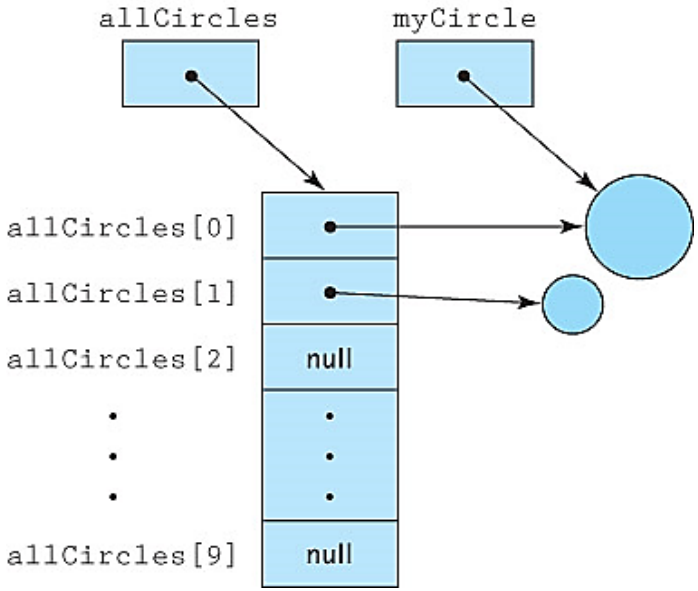
Basic structuring mechanisms

- There are two basic structuring mechanisms provided in Java (and many other high level languages):

References



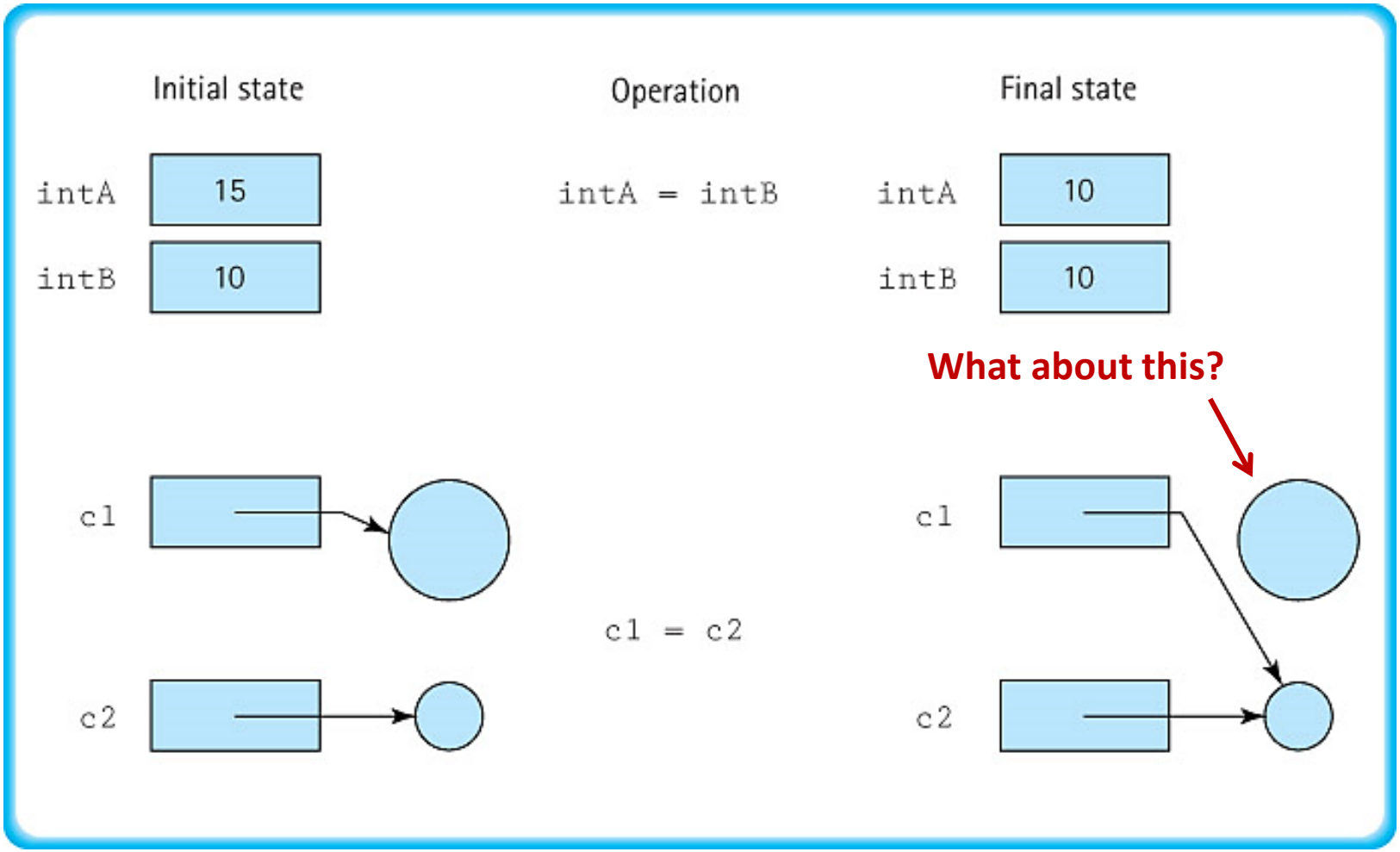
Arrays



References

- Java includes two types of variables:
  - Primitive variables - a primitive type refers to a primitive data type
  - Reference variables – a reference type refers to a class or an array.
- References are memory addresses.
- Other names for references: links, addresses, or pointers.
- Java uses the reserved word `null` to indicate an "absence of reference" (like a `NULL` pointer in C++).
- A variable of a reference (non-primitive) type holds the address of the memory location that holds the value of the variable, rather than the value itself.

Assignment statements



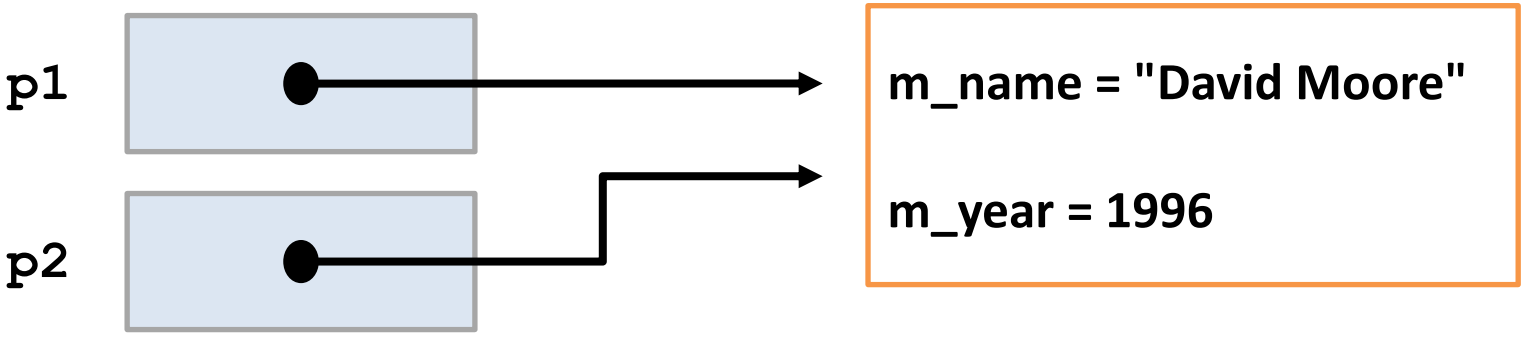
Assignment statements - aliases

Person
- m_name: String - m_year: int
+ Person(name: String, year: int) + setName(name: String): void + setYear(year: int): void

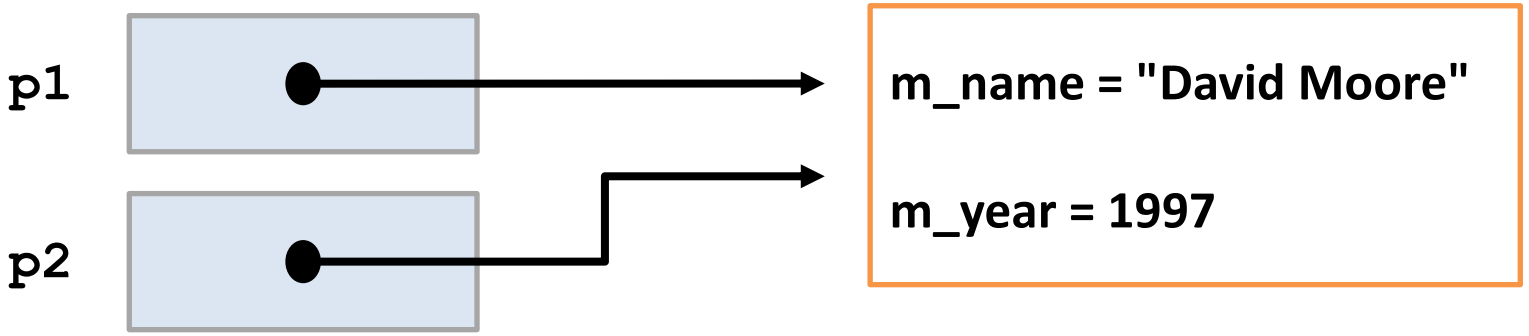
Assignment statements - aliases

```
Person p1 = new Person("David Moore", 1996);
Person p2 = p1;
```

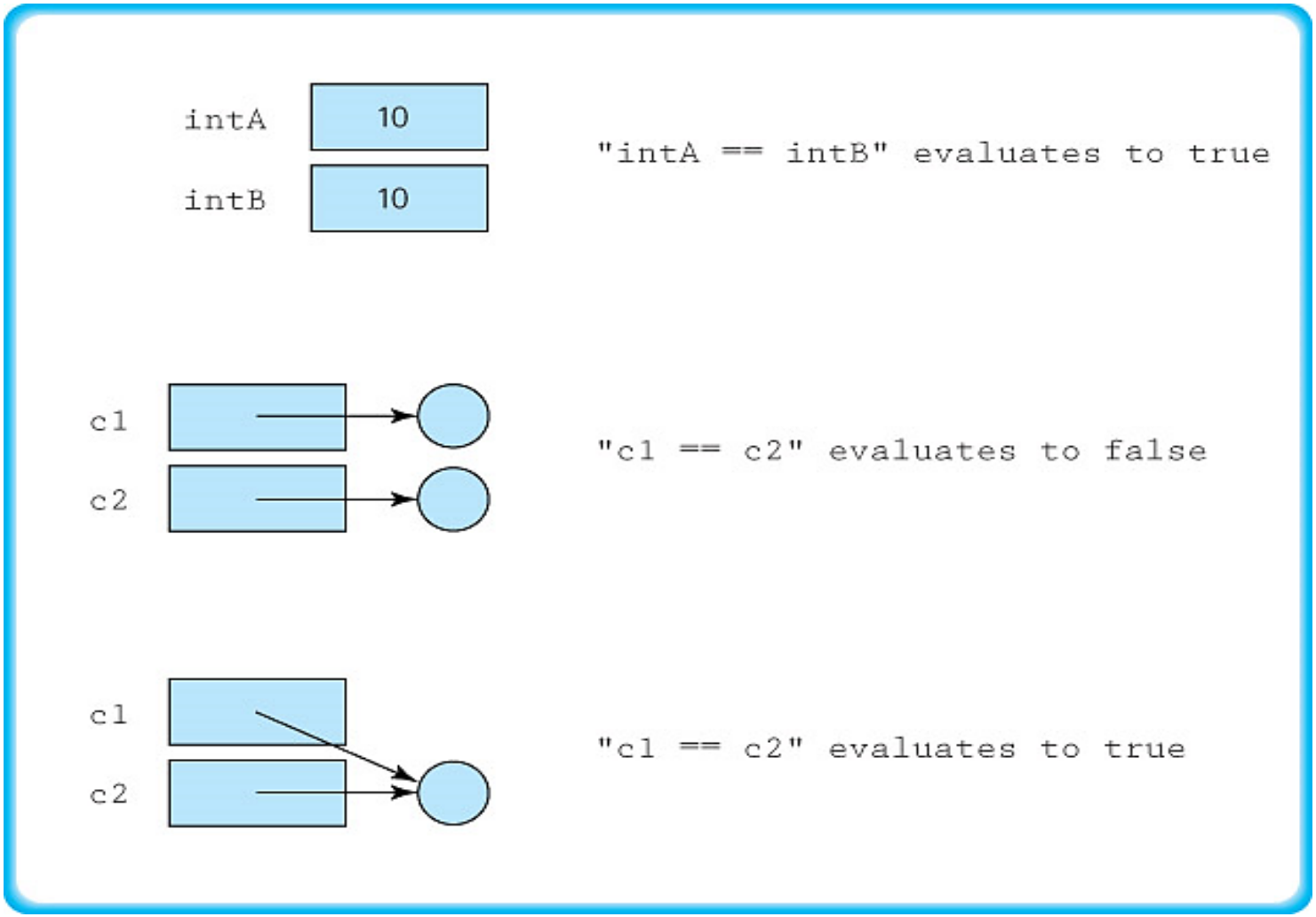
Initial state



State after `p2.setYear(1997)`



Comparison statements

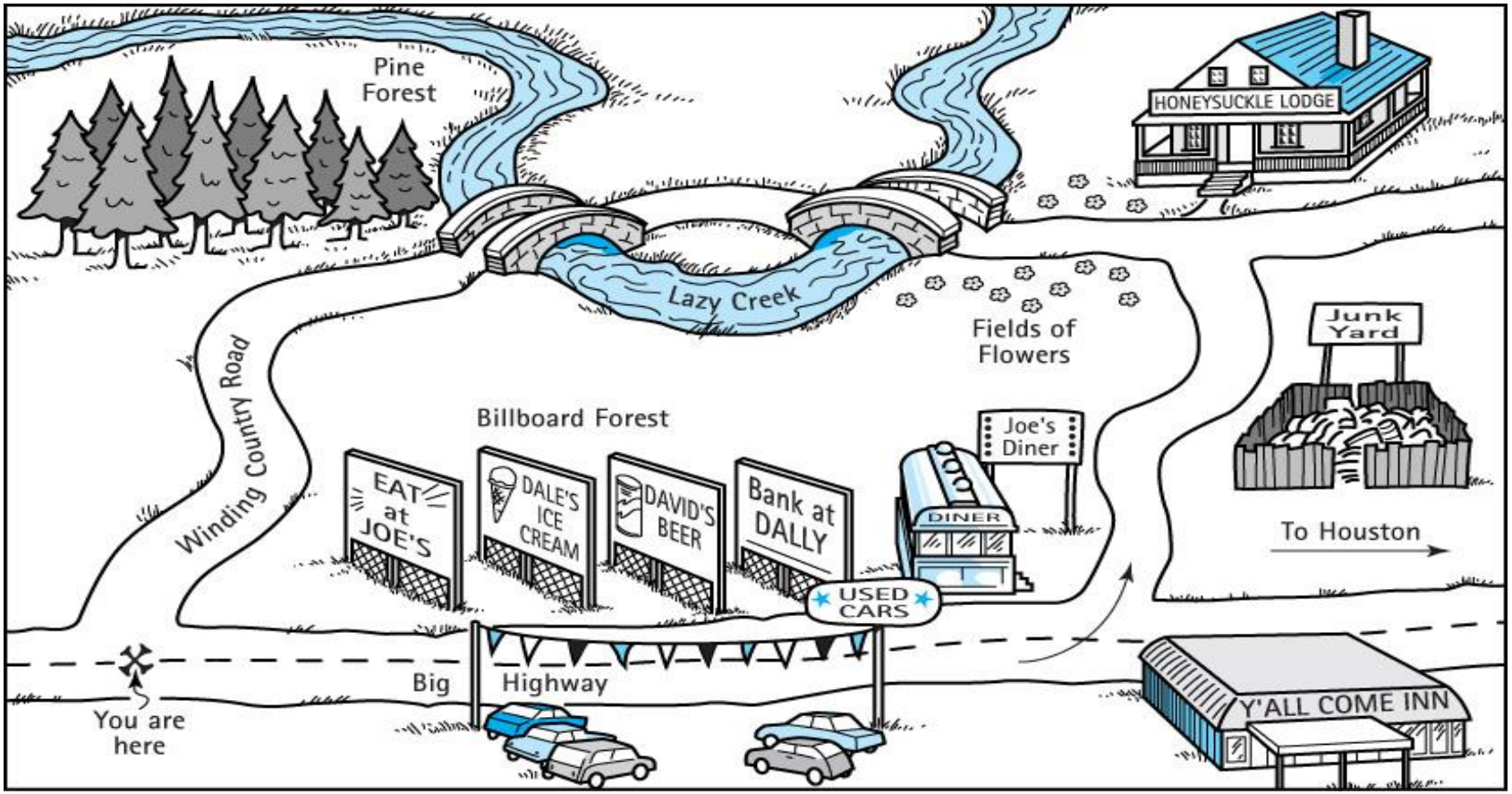


Garbage management

- **Garbage** - The set of currently unreachable objects.
- **Garbage collection** - The process of finding all unreachable objects and deallocating their storage space.
- **Deallocate** - To return the storage space for an object to the pool of free memory so that it can be reallocated to new objects.
- **Dynamic memory management** - The allocation and deallocation of storage space as needed while an application is executing.

Comparing algorithms: Big-O analysis

- There can be more than one algorithm to solve a problem.



Counting operations

- To measure the complexity of an algorithm we attempt to count the number of basic operations required to complete the algorithm.
- To make our count generally usable we express it as a function of the size of the problem.
- Rather than count all operations, select a fundamental operation, an operation that is performed "the most", and count it.
  - For example, only count comparison operations when comparing sorting algorithms.

Big-O notation

- How does the performance change with the size of the input?
  - Suppose it takes $2N^2 + 5N - 3$ operations to sort a list of size N.
  - What happens if we double the size of the input?
    - $8N^2 + 10N - 3$: almost four times as long
    - As N gets bigger and bigger, the $2N^2$ term becomes more important, and the $5N$ term becomes less important.
    - If we care mostly about big problems, we can ignore those lower-order terms.

Big-O notation

- The **order of complexity** of an algorithm is a measure of how its performance changes as the problem instance gets bigger.
- Written in "big O notation": O(N^2), O(N).
  - Write the expression for number of operations: $2N^2 + 5N - 3$
  - Take the highest-order (fastest-growing) term: $2N^2$
  - Drop the constant: N^2
  - So we say the algorithm has complexity O(N^2).
  - "Order N squared" or "Big O of N squared"

Big-O notation

- An expression like O(N^2) represents a **complexity class**:
  - All the formulas with the highest-order term n^2.
  - $N^2 + 3N - 1$, $N^2/10 + N$, $100N^2 - 50$, …
- Computer scientists have names for some of the most common complexity classes:

worse

- O(1): **constant**. Doesn't depend on the size of the input.
- O($\log_2 N$): **logarithmic**. Doubling the input adds an amount of time.
- O(N): **linear**. Doubling the input doubles the time.
- O($N \log_2 N$): N **log** N. Good sorting algorithms, e.g., Mergesort
- O(N^2): **quadratic**. Doubling the input quadruples the time.
- O(2^N): **exponential**. Adding one to the input size doubles the time.

Comparison of growth rates

N	$\log_2 N$	$N \log_2 N$	N^2	N^3	2^N
1	0	1	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
16	4	64	256	4,096	65,536
64	6	384	4,096	262,144	**requires 20 digits**
128	7	896	16,384	2,097,152	**requires 39 digits**
256	8	2,048	65,536	16,777,216	**requires 78 digits**

Three complexity cases

- Best case complexity - Related to the minimum number of steps required by an algorithm, given an ideal set of input values in terms of efficiency.
- Average case complexity - Related to the average number of steps required by an algorithm, calculated across all possible sets of input values.
- Worst case complexity - Related to the maximum number of steps required by an algorithm, given the worst possible set of input values in terms of efficiency.
- To simplify analysis yet still provide a useful approach, we usually use worst case complexity, count a fundamental operation, and use Big-O complexity.

Examples

- ```
  sum = 0;
  for (count = 1; count <= n; count++)
      sum = sum + count;
  ```

  O(N)

- ```
  sum = ((n + 1) * n) / 2;
  ```

  O(1)

Action items

- Read book chapter 1.