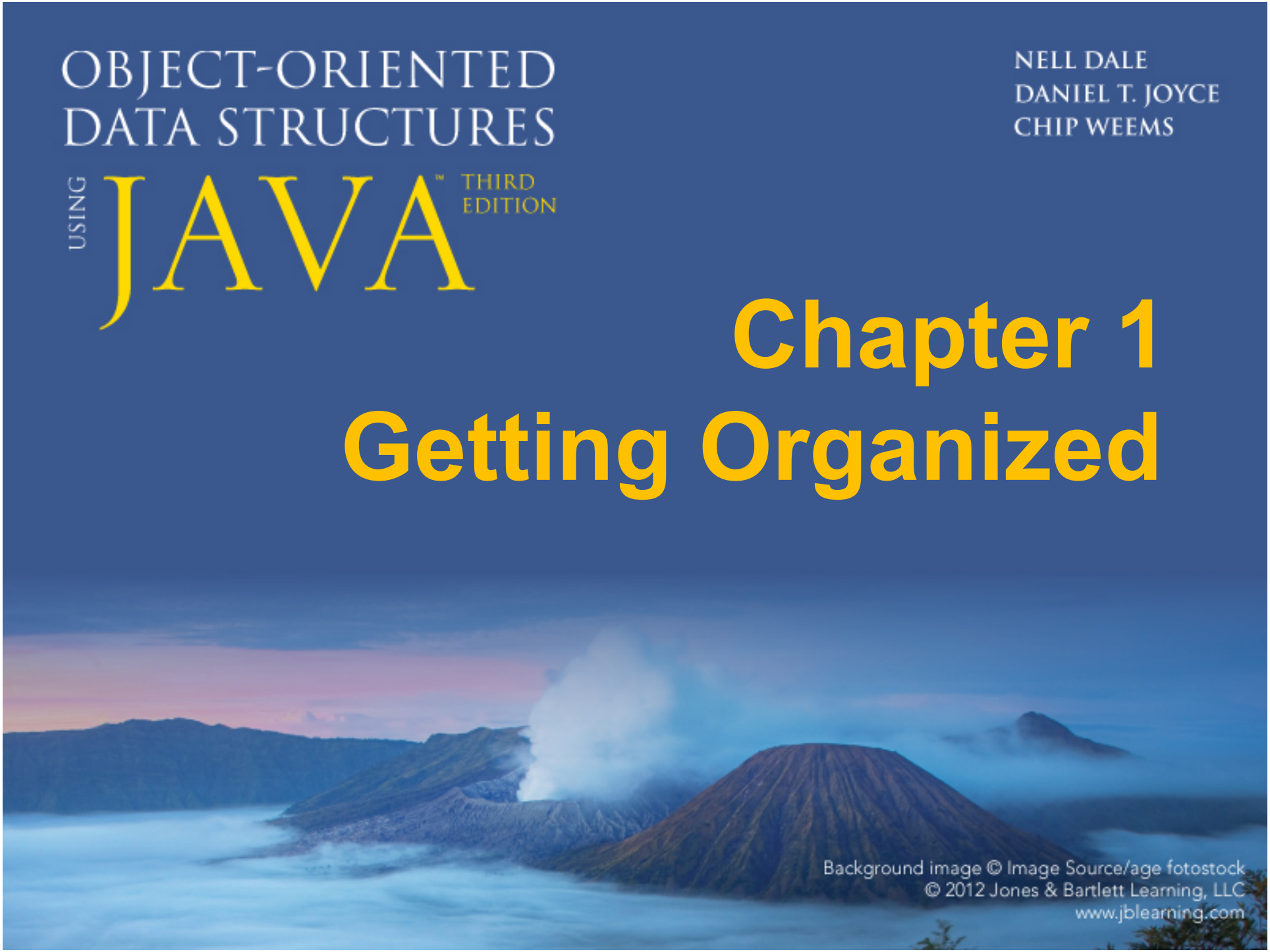

OBJECT-ORIENTED
DATA STRUCTURES
USING JAVA™ THIRD EDITION
NELL DALE
DANIEL T. JOYCE
CHIP WEEMS
Chapter 1
Getting Organized
Background image © Image Source/age fotostock
© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com

Chapter 1: Getting Organized

1.1 – Software Engineering
1.2 – Object Orientation
1.3 – Classes, Objects, and Applications
1.4 – Organizing Classes
1.5 – Data Structures
1.6 – Basic Structuring Mechanisms
1.7 – Comparing Algorithms: Big-O Analysis

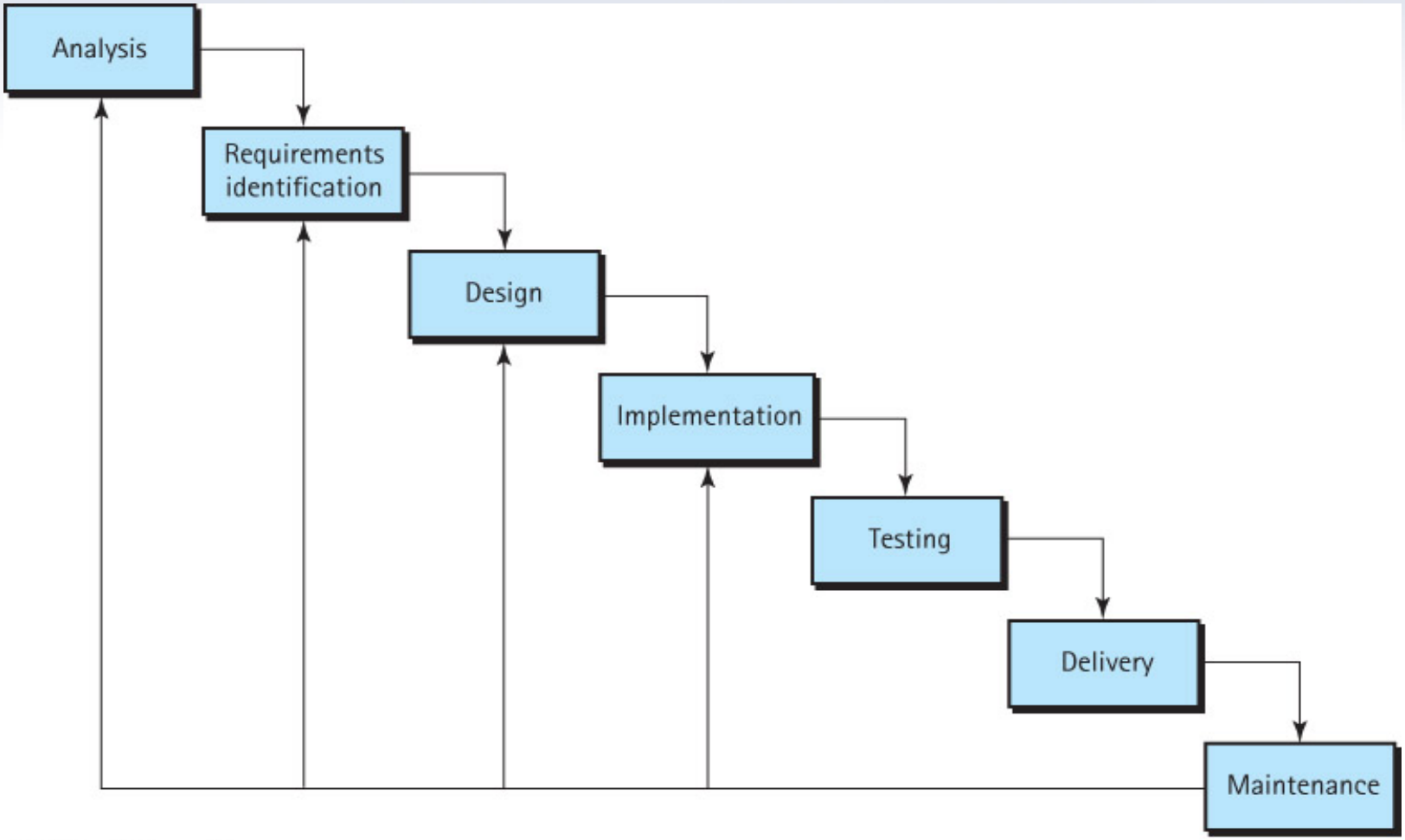
1.1 Software Engineering

- The field devoted to the specification, design, production, and maintenance of non-trivial software products.
- Includes supporting activities such as
  - cost estimation
  - documentation
  - team organization
  - use of tools

Software Life Cycle Activities

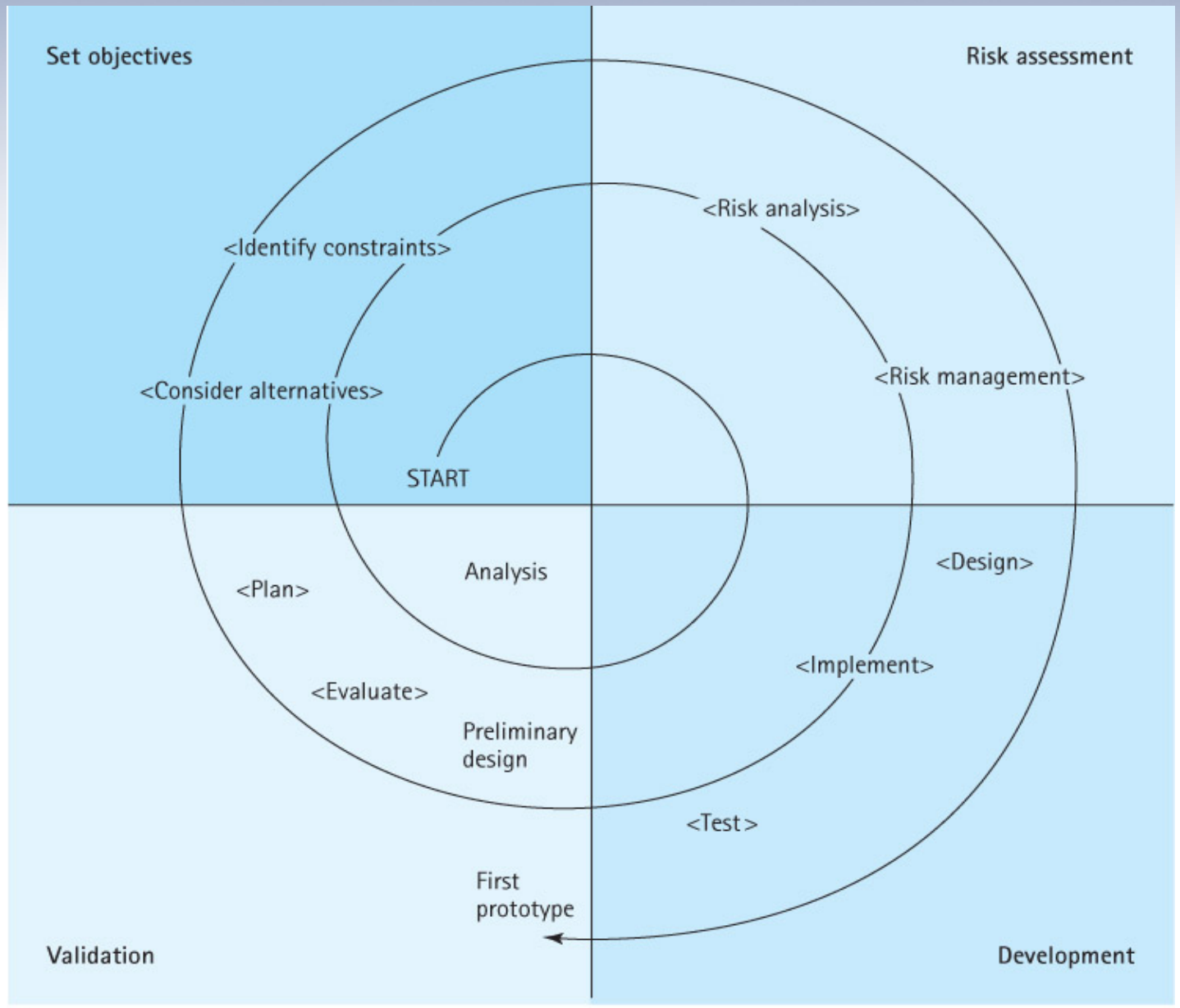
- Problem analysis
- Requirements
  - elicitation
  - specification
- Design
  - high-level
  - low-level
- Testing
- Verification
- Delivery
- Operation
- Maintenance

Waterfall Life-Cycle Model



(a) Waterfall model

Spiral Life-Cycle Model



(b) Spiral model

Agile Methods

- Customer involvement across life cycle
- Incremental development and delivery
- Embrace change
  - Customer satisfaction is primary goal
- Pair Programming
- and more

Some Goals of Quality Software
(no matter what approach you use)

- It works.
- It can be modified without excessive time and effort.
- It is reusable.
- It is completed on time and within budget.

1.2 Object Orientation

- Objects represent
  - information: we say the objects have *attributes*
  - behavior: we say the objects have *responsibilities*
- Objects can represent "real-world" entities such as bank accounts
- Objects are self-contained and therefore easy to implement, modify, and test for correctness
- Object-oriented classes, when designed properly, are very easy to reuse

1.3 Classes, Objects, and Applications

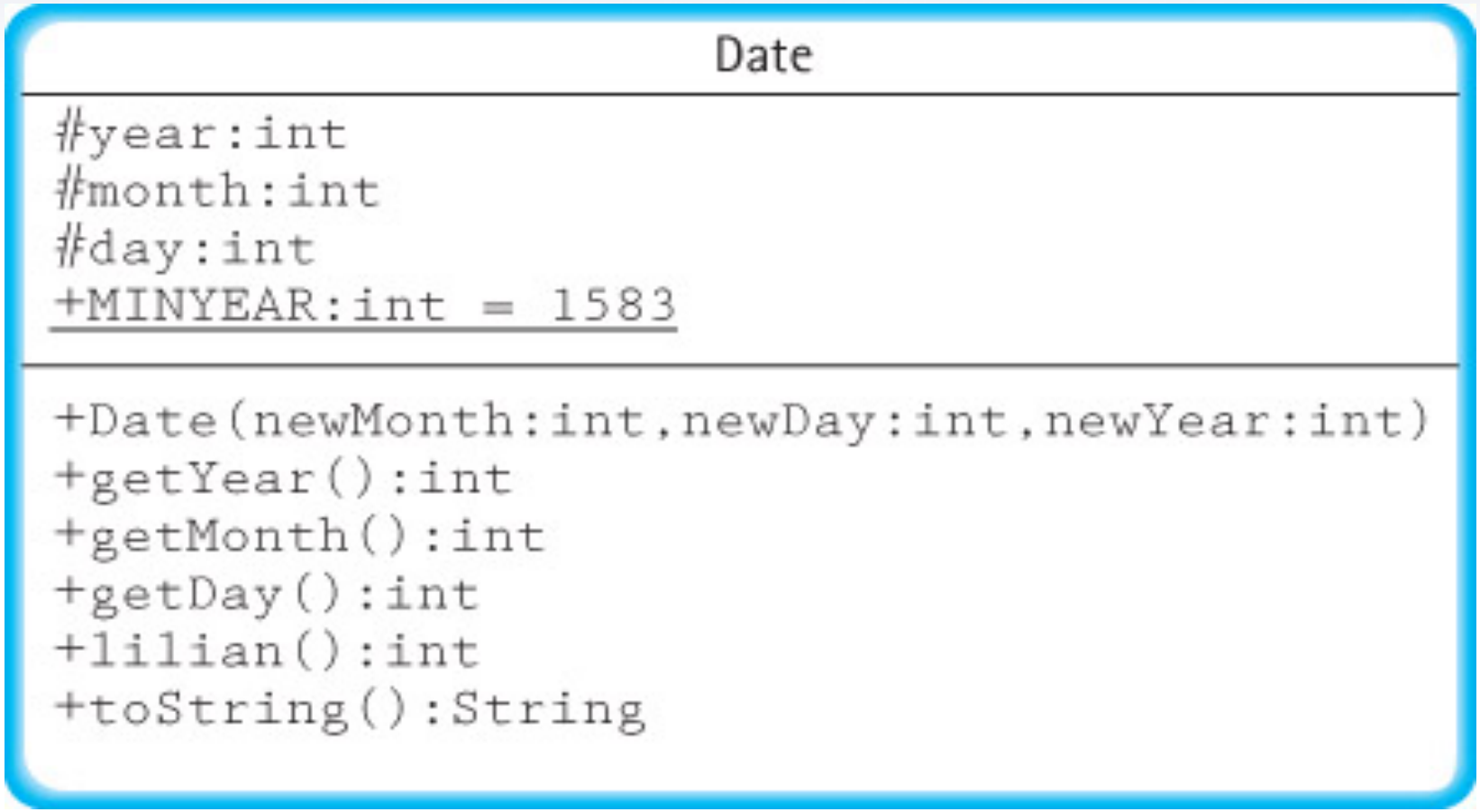
- An object is an instantiation of a class
- Alternately, a class defines the structure of its objects.
- A class definition includes variables (data) and methods (actions) that determine the behavior of an object.
- Example: the **`Date`** class (next slide)

```
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.
    // Algorithm goes here.
  }

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

Java Access Control Modifiers

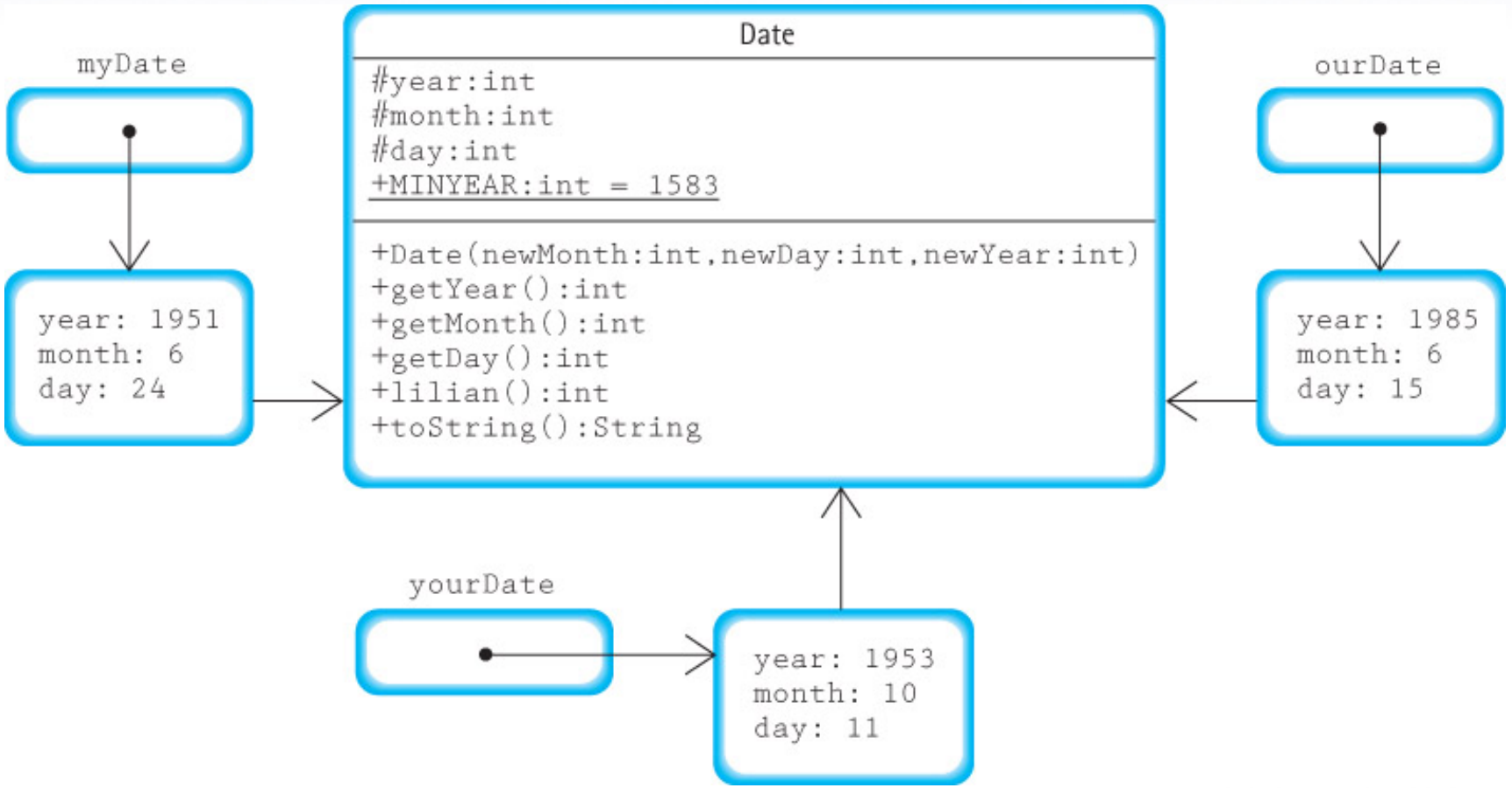
	Within the Class	Within Subclasses in the Same Package	Within Subclasses in Other Packages	Everywhere
public	X	X	X	X
protected	X	X	X	
package	X	X		
private	X			

Class Diagram for 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 (next 3 slides)

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
```

```
//-------------------------------------------------------------------------
// DaysBetween.java          by Dale/Joyce/Weems                Chapter 1
//
// Asks the user to enter two "modern" dates and then reports
// the number of days between the two dates.
//-------------------------------------------------------------------------
import java.util.Scanner;
public class DaysBetween
{
  public static void main(String[] args)
  {
    Scanner conIn = new Scanner(System.in);
    int day, month, year;

    System.out.println("Enter two 'modern' dates: month day year");
    System.out.println("For example January 12, 1954 would be: 1 12 1954");
    System.out.println();
    System.out.println("Modern dates occur after " + Date.MINYEAR + ".");
    System.out.println();

    System.out.println("Enter the first date:");
    month = conIn.nextInt();
    day = conIn.nextInt();
    year = conIn.nextInt();
    Date date1 = new Date(month, day, year);
```

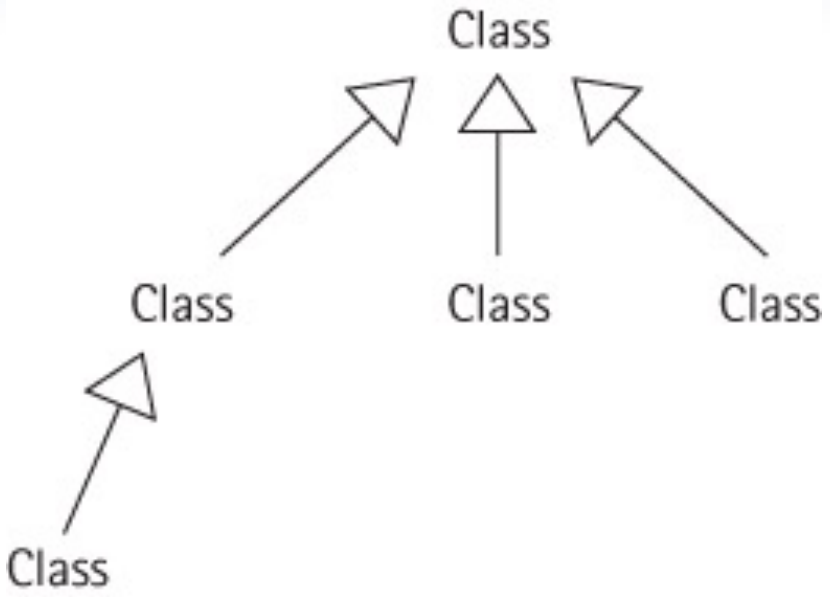
```
    System.out.println("Enter the second date:");
    month = conIn.nextInt();
    day = conIn.nextInt();
    year = conIn.nextInt();
    Date date2 = new Date(month, day, year);

    if ((date1.getYear() <= Date.MINYEAR)
        ||
        (date2.getYear() <= Date.MINYEAR))
      System.out.println("You entered a 'pre-modern' date.");
    else
    {
      System.out.println("The number of days between");
      System.out.print(date1);
      System.out.print(" and ");
      System.out.print(date2);
      System.out.print(" is ");
      System.out.println(Math.abs(date1.lilian() - date2.lilian()));
    }
  }
}
```

1.4 Organizing Classes

- During object-oriented development hundreds of classes can be generated or reused to help build a system.
- The task of keeping track of these classes would be impossible without organizational structure.
- 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

- Allows programmers to create a new class that is a specialization of an existing class.
- We say that the new class is a subclass of the existing class, which in turn is the superclass of the new class.



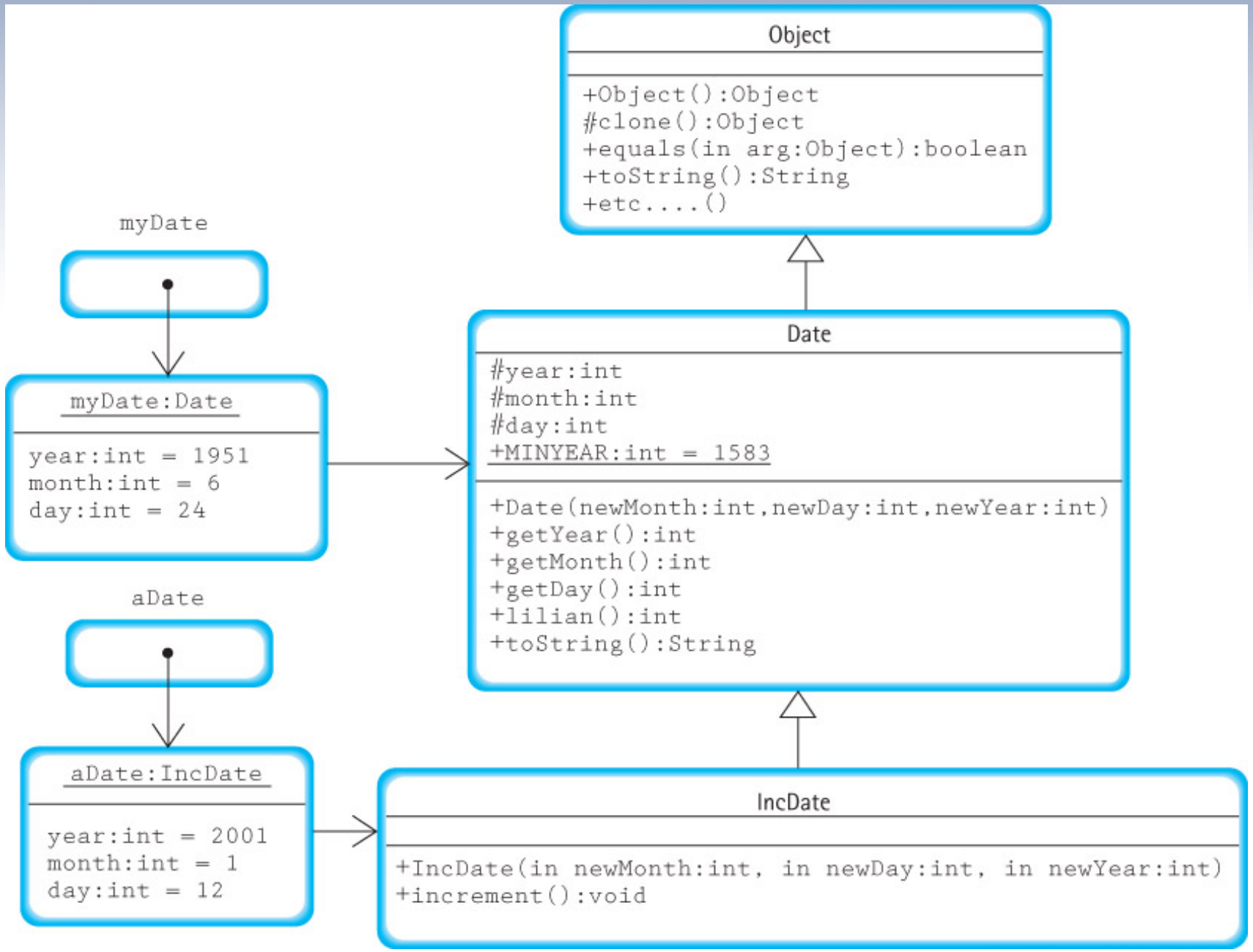
Example of Inheritance

```
public class IncDate extends Date
{
  public IncDate(int newMonth, int newDay, int newYear)
  {
    super(newMonth, newDay, newYear);
  }

  public void increment()
  // Increments this IncDate to represent the next day.
  // For example if this = 6/30/2005 then this becomes 7/1/2005.
  {
    // increment algorithm goes here
  }
}
```

Declaring and Using Date and IncDate Objects

```
Date myDate = new Date(6, 24, 1951);
IncDate aDate = new IncDate(1, 11, 2001);

System.out.println("mydate day is:   " +
myDate.getDay());
System.out.println("aDate day is:    " +
aDate.getDay());

aDate.increment();

System.out.println("the day after is: " +
aDate.getDay());
```

See Extended Class Diagram next slide.



Packages

- Java lets us group related classes together into a unit called a package. Packages provide several advantages. They
  - let us organize our files.
  - can be compiled separately and imported into our programs.
  - make it easier for programs to use common class files.
  - help us avoid naming conflicts (two classes can have the same name if they are in different packages).

Using Packages

- A Java compilation unit can consist of a file with
  - the keyword **`package`** followed by an identifier indicating the name of the package:

    ```
    package someName;
    ```
  - import declarations, to make the contents of other packages available:

    ```
    import java.util.Scanner;
    ```
  - one or more declarations of classes; exactly one of these classes must be public
- The classes defined in the file are members of the package.
- The imported classes are not members of the package.
- The name of the file containing the compilation unit must match the name of the public class within the unit.

Using Packages

- Each Java compilation unit is stored in its own file.
- The Java system identifies the file using a combination of the package name and the name of the public class in the compilation unit.
- Java restricts us to having a single public class in a file so that it can use file names to locate all public classes.
- Thus, a package with multiple public classes must be implemented with multiple compilation units, each in a separate file.

Using Packages

- In order to access the contents of a package from within a program, you must import it into your program:

```
import packagename.*;
import packagename.Classname;
```

- The Java package rules are defined to work seamlessly with hierarchical file systems:

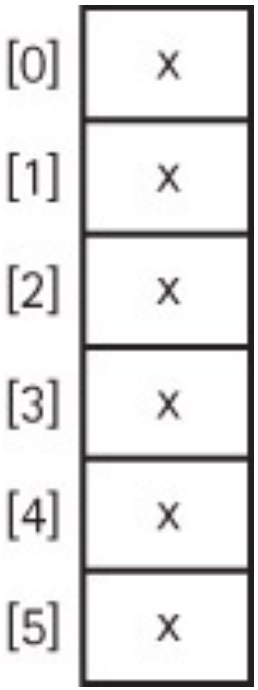
```
import ch03.stacks.*;
```

1.5 Data Structures

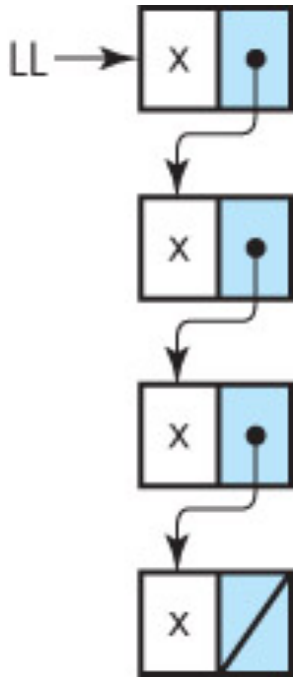
- The way you view and structure the data that your programs manipulate greatly influences your success.
- A language's set of primitive types (Java's are byte, char, short, int, long, float, double, and boolean) are not sufficient, by themselves, for dealing with data that have many parts and complex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

Implementation Dependent Structures

Array

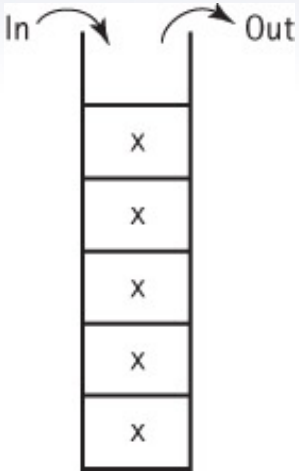


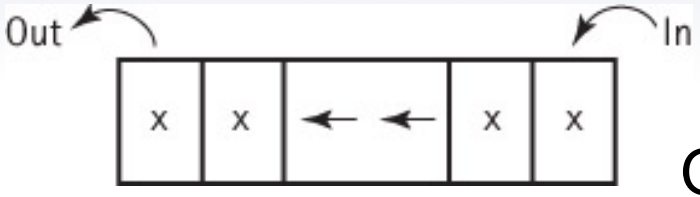
Linked List



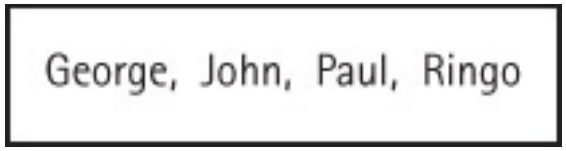
Implementation Independent Structures

Stack



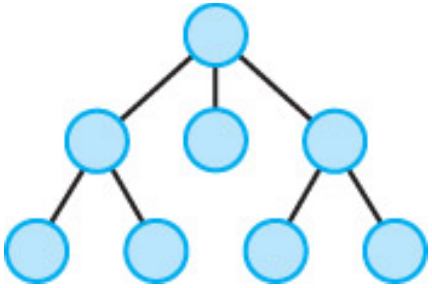


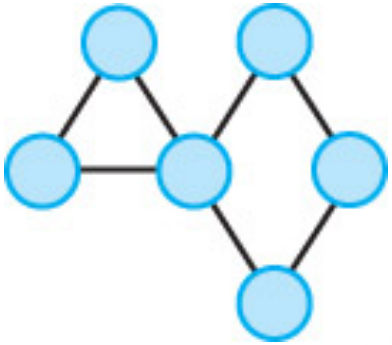
Queue



Sorted List

Tree



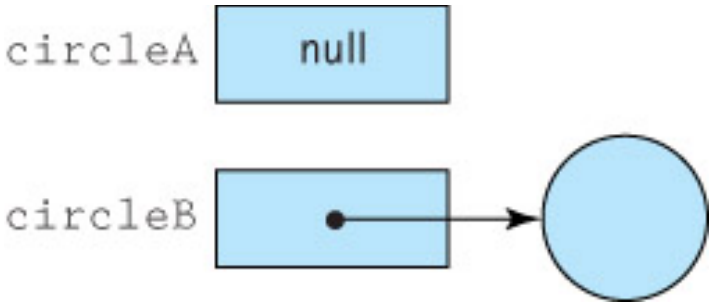


Graph

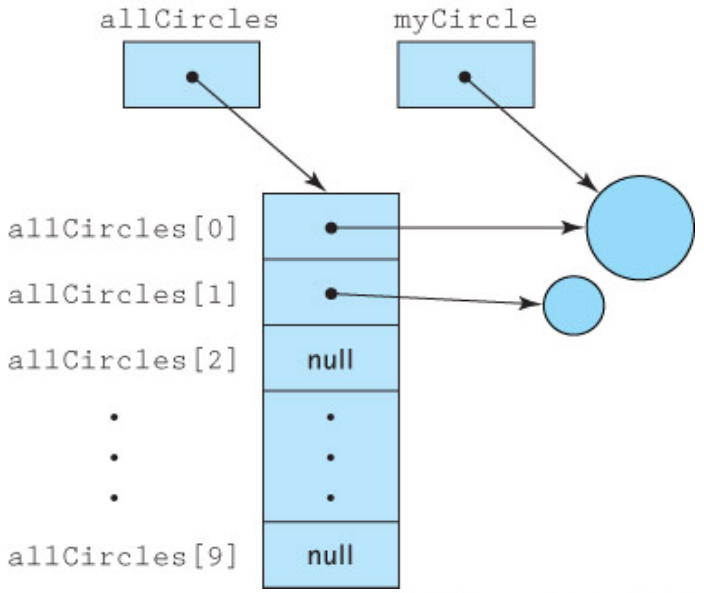
1.6 Basic Structuring Mechanisms

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

References



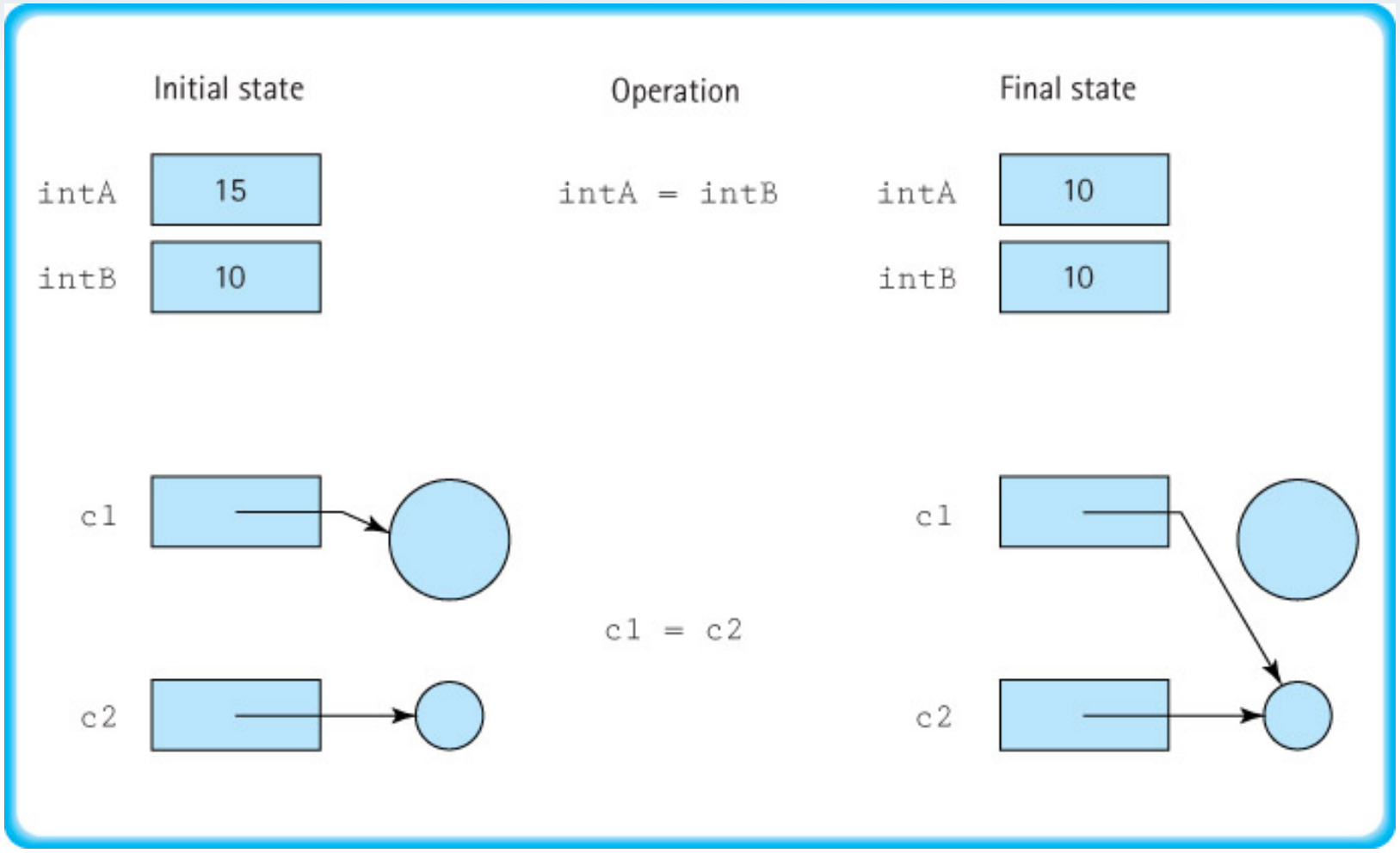
Arrays



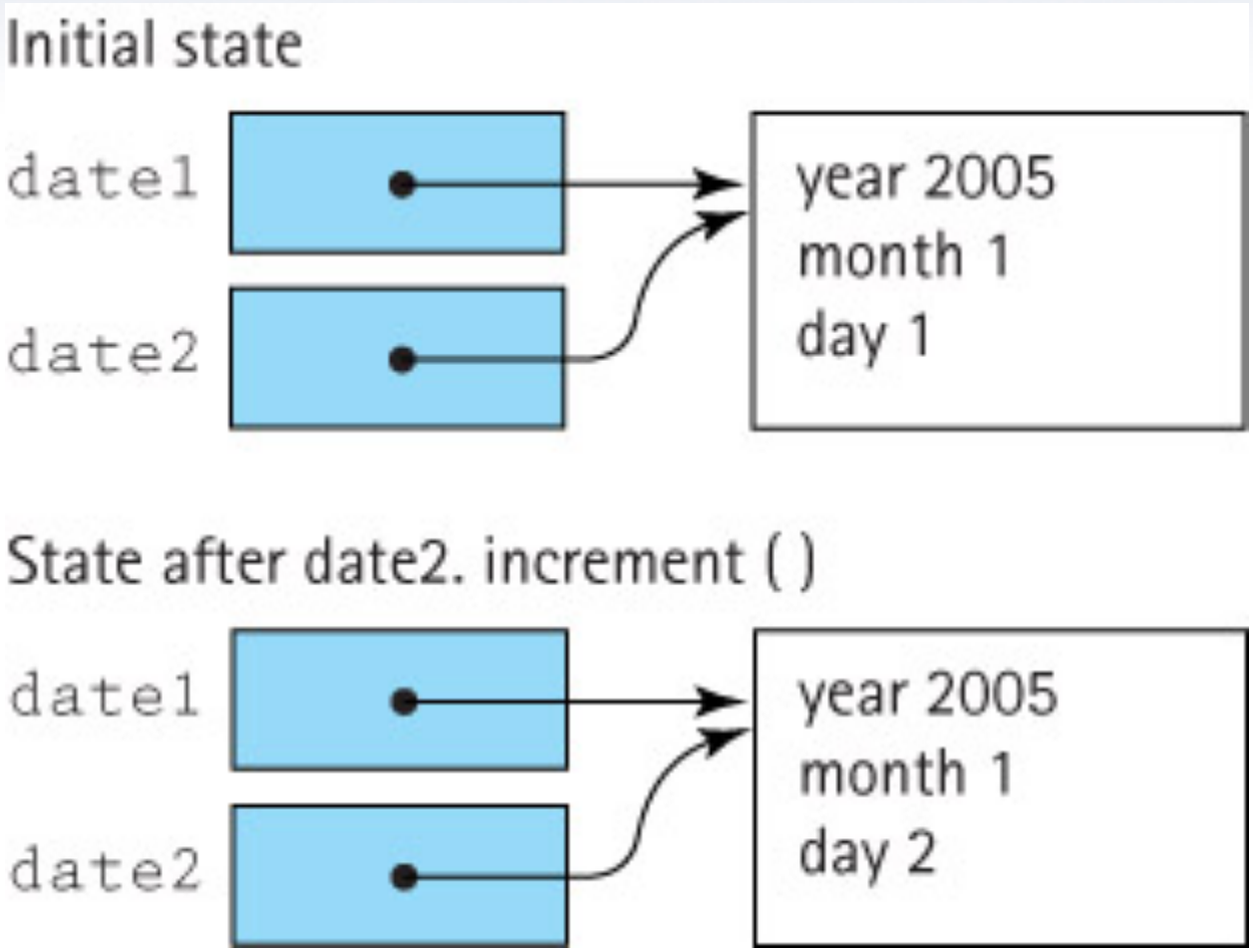
References

- Are memory addresses
- Sometimes referred to as *links*, *addresses*, or *pointers*
- Java uses the reserved word **`null`** to indicate an "absence of reference"
- 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.
- This has several ramifications …

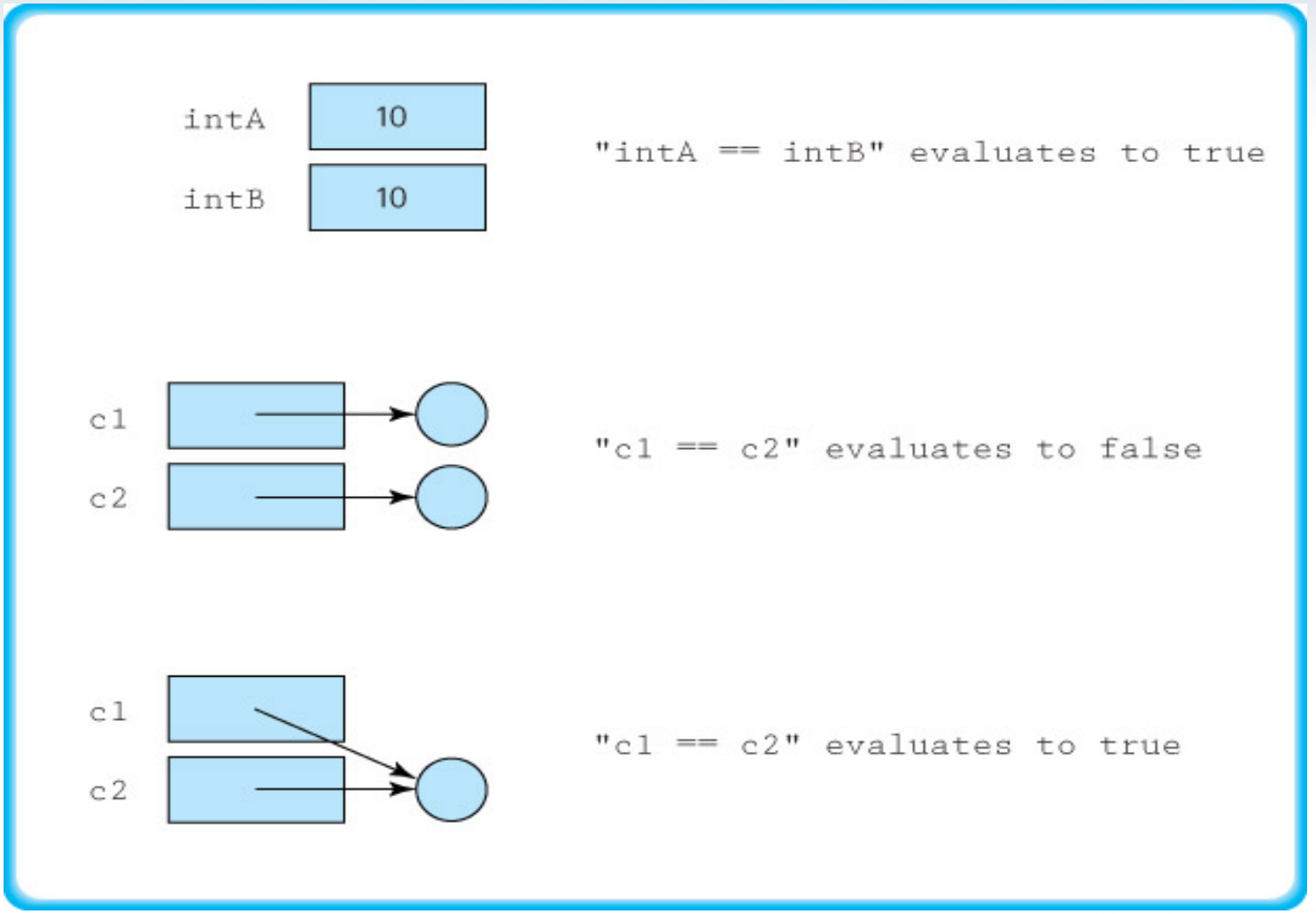
Assignment Statements



Be aware of aliases



Comparison Statements



Garbage Management

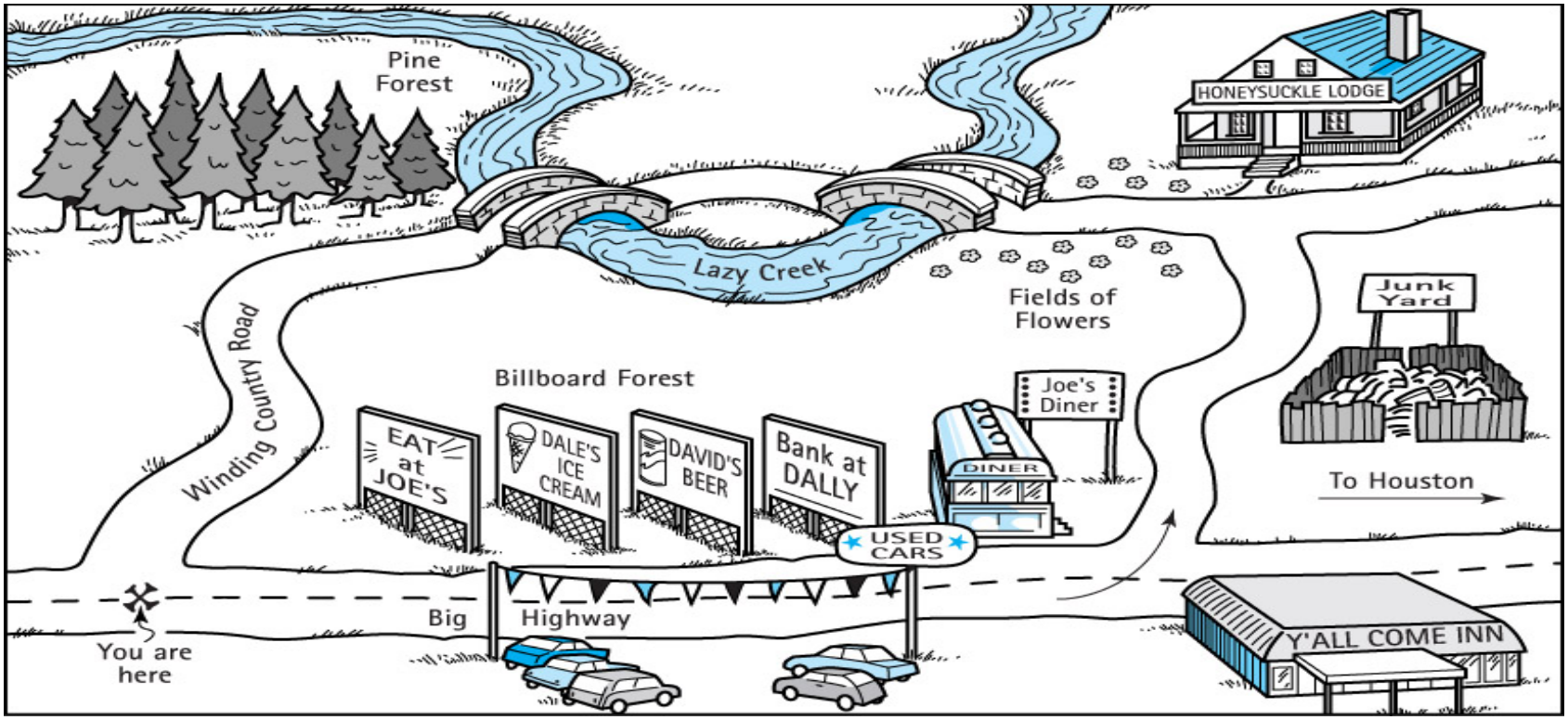
- **Garbage** The set of currently unreachable objects
- **Garbage collection** The process of finding all unreachable objects and deallocating their storage space
- **Deallocate** To return the storage space for an object to the pool of free memory so that it can be reallocated to new objects
- **Dynamic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

Arrays

- We assume students are already familiar with arrays. The subsection on pages 37 to 41 reviews some of the subtle aspects of using arrays in Java:
  - they are handled “by reference”
  - they must be instantiated
  - initialization lists are supported
  - you can use arrays of objects
  - you can use multi-dimensional arrays

1.7 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.

Counting Operations Example

problem: return true if sum of numbers in array is > 0, false otherwise

```
Set sum to zero

while more integers
  Set sum to
       sum + next int

if sum > 0
  Return true
else
  Return false
```

- if N = size of array the number of operations required is N + 3
- But
  - too dependent on programming language and counting approach
  - difficult if problem/ algorithm is more complicated

Isolate a fundamental operation

- Rather than count all operations, select a fundamental operation, an operation that is performed “the most”, and count it.
- For example, if the problem is to sort the elements of an array, count the number of times one element is compared to another element, i.e., only count comparison operations when comparing sorting algorithms.

A further simplification: Big-O Notation

- We measure the complexity of an algorithm as the number of times a fundamental operation is performed, represented as a function of the size of the problem.
- For example, an algorithm performed on an N element array may require $2N^2 + 4N + 3$ comparisons.
- Big-O notation expresses computing time (complexity) as the term in the function that increases most rapidly relative to the size of a problem.
- In our example, rather than saying the complexity is $2N^2 + 4N + 3$ we say it is $O(N^2)$.
- This works just as well for most purposes and simplifies the analysis and use of the complexity measure.

Common Orders of Magnitude

- O(1) is called bounded time. The amount of work is not dependent on the size of the problem.
- $O(\log_2 N)$ is called logarithmic time. Algorithms that successively cut the amount of data to be processed in half at each step typically fall into this category.
- O(N) is called linear time. Adding together the elements of an array is O(N).
- $O(N \log_2 N)$ is called N log N time. Good sorting algorithms, such as Quicksort, Heapsort, and Mergesort presented in Chapter 10, have N log N complexity.
- $O(N^2)$ is called quadratic time. Some simple sorting algorithms are $O(N^2)$ algorithms.
- $O(2^N)$ is called exponential time. These algorithms are extremely costly.

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**

Ways to simplify analysis of algorithms

- Consider worst case only
  - but average case can also be important
- Count a fundamental operation
  - careful; make sure it is the most used operation within the algorithm
- Use Big-O complexity
  - especially when interested in “large” problems

Sum of Consecutive Integers

Algorithm Sum1

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

Sum1 is O(N)

Algorithm Sum2

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

Sum2 is O(1)

Finding a Number in a Phone Book

Algorithm Lookup1
Check first name
While (unsuccessful)
  Check the next name

Lookup1 is O(N)

Algorithm Lookup2
Search area = entire book
Check middle name in search area
While (unsuccessful)
  If middle name > target name
    Search area = first half of search area
  Else
    Search area = second half of search area
  Check middle name in search area

Lookup2 is O($\log_2 N$)