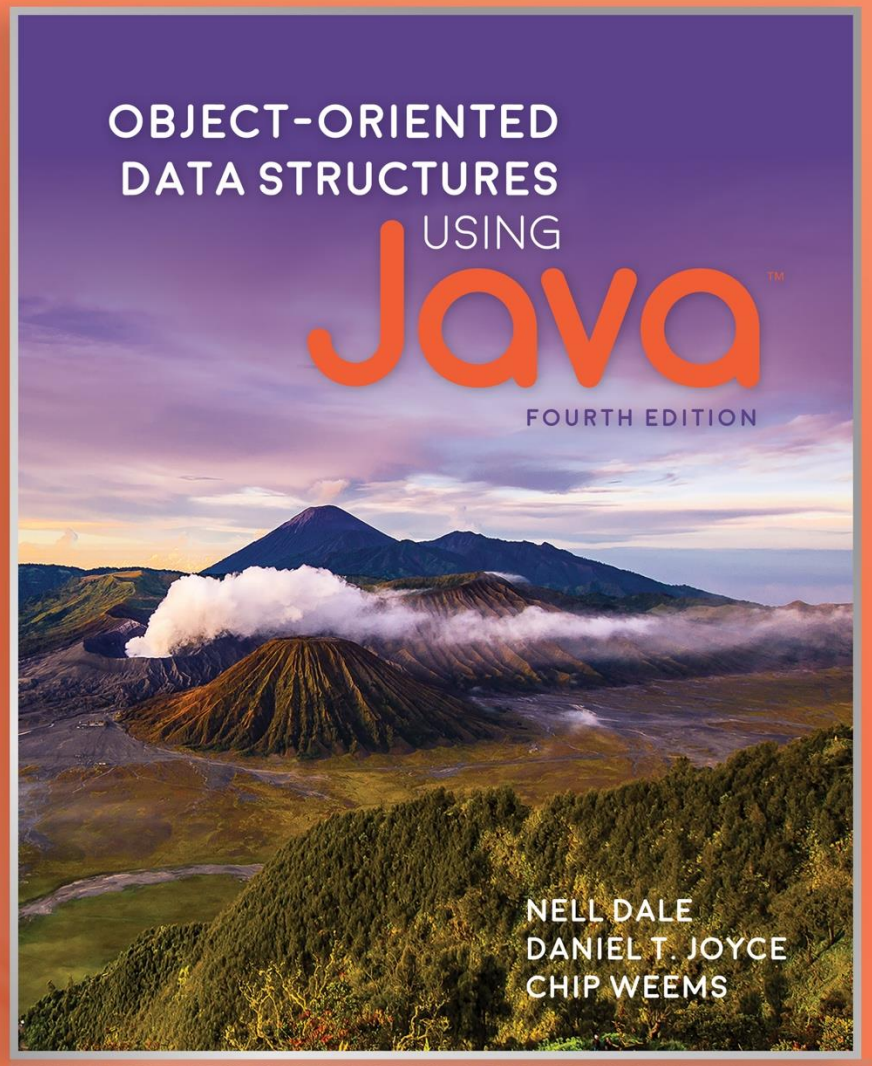
Chapter 1

Getting Organized



Chapter 1: Getting Organized

1.1 – Classes, Objects, and Applications
1.2 – Organizing Classes
1.3 – Exceptional Situations
1.4 – Data Structures
1.5 – Basic Structuring Mechanisms
1.6 – Comparing Algorithms: Order of Growth Analysis

1.1 Classes, Objects, and Applications

- 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, test for correctness, and reuse.

Classes and Objects

- 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)

```
package ch01.dates;
public class Date
{
  protected int year, month, 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.
  }

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

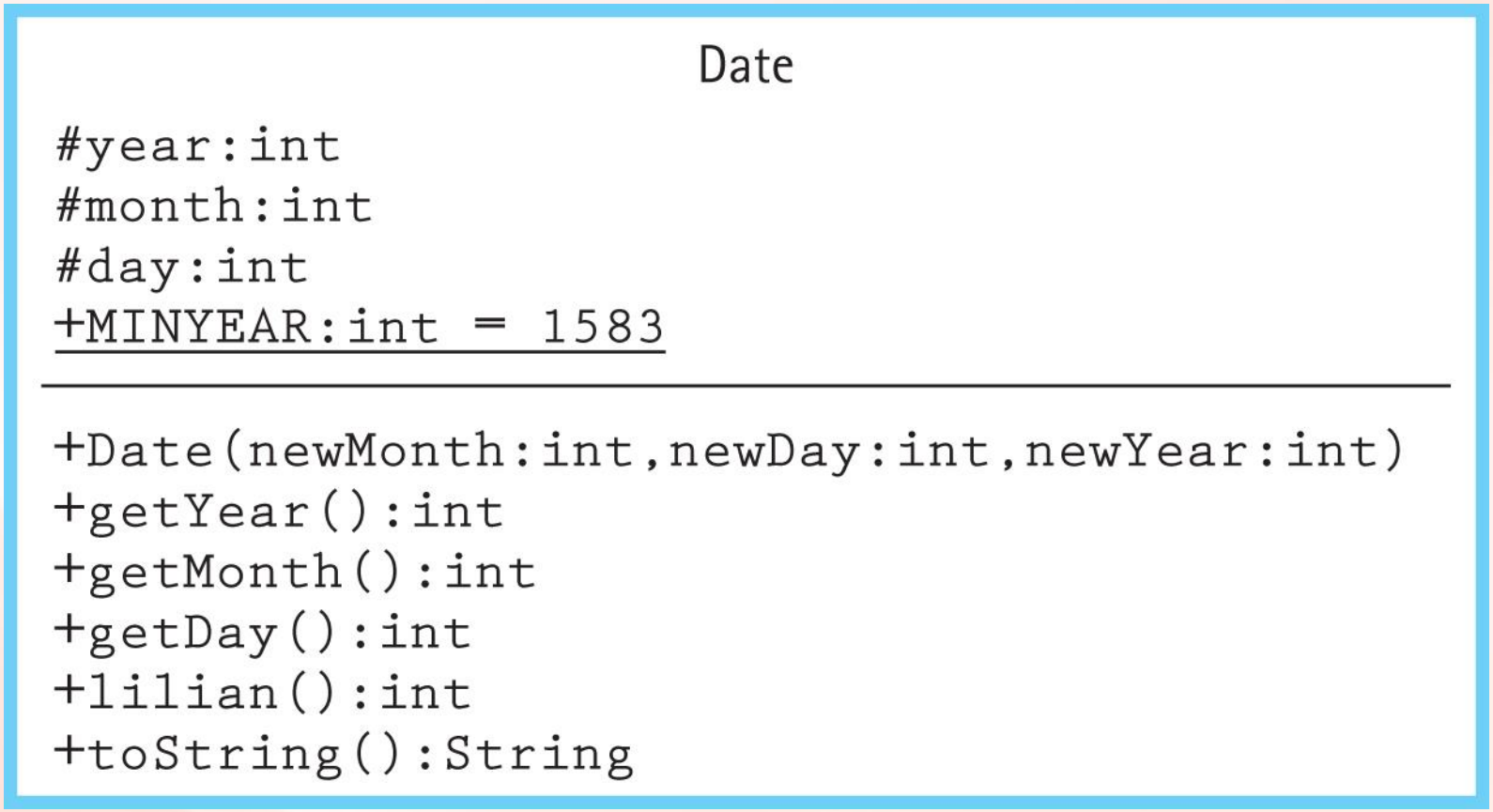
Java Access Control Modifiers

	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			

The Unified Method

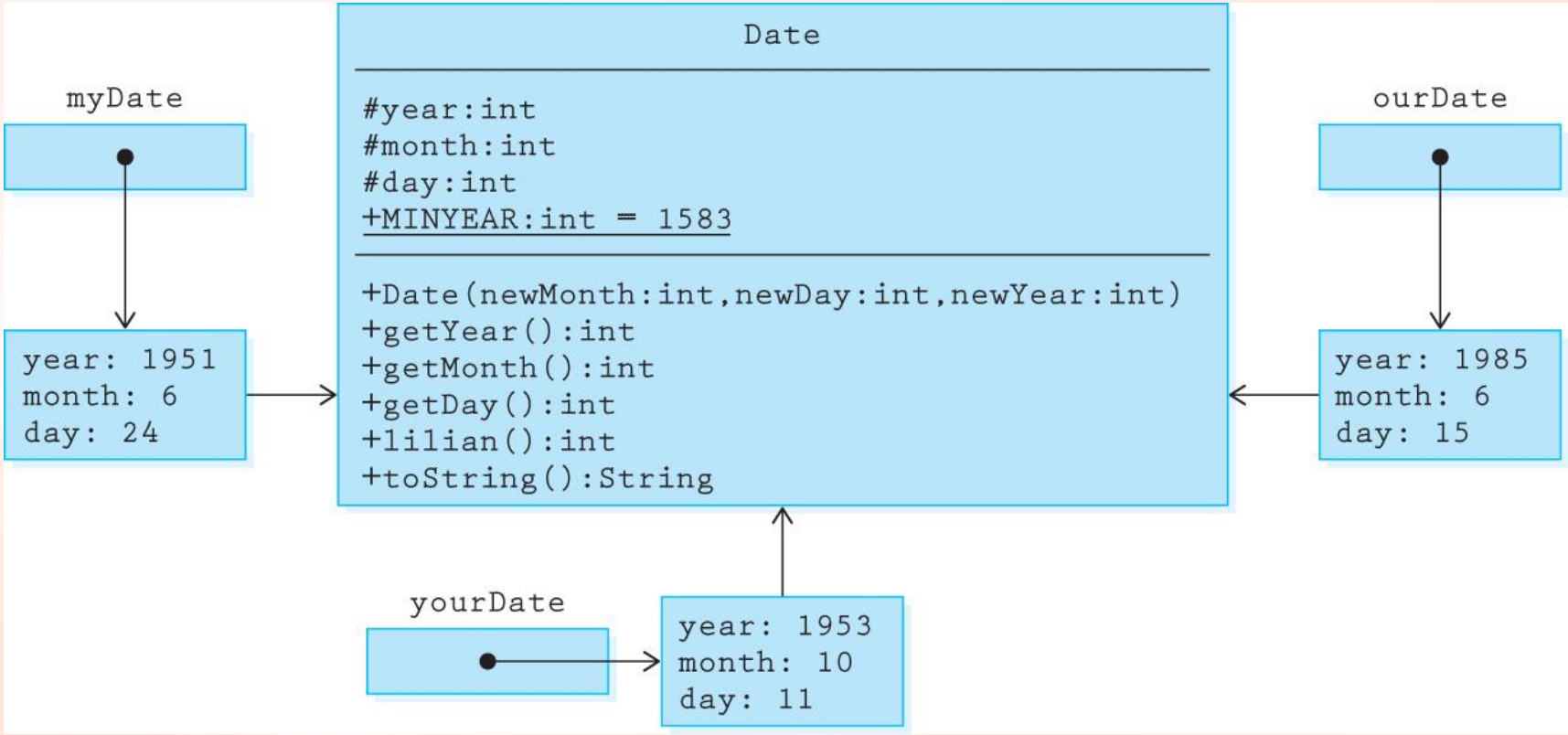
- Use-case driven: a description of a sequence of actions performed by a user within the system to accomplish some task
- Iterative and incremental: involves a series of development cycles
- Architecture-centric: diagrams indicate the overall structure of the system, the way in which its components interact

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.
//-------------------------------------------------------------------------
package ch01.apps;
import java.util.Scanner;
public class DaysBetween
{
  public static void main(String[] args)
  {
    Scanner scan = 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 = scan.nextInt();
    day = scan.nextInt();
    year = scan.nextInt();
    Date d1 = new Date(month, day, year);
```

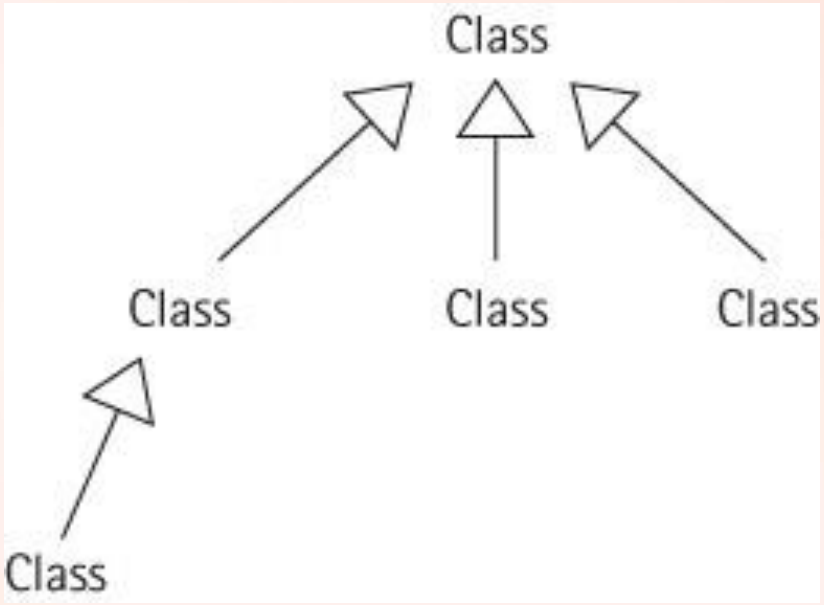
```
    System.out.println("Enter the second date:");
    month = scan.nextInt();
    day = scan.nextInt();
    year = scan.nextInt();
    Date d2 = new Date(month, day, year);

    if ((d1.getYear() <= Date.MINYEAR)
        ||
        (d2.getYear() <= Date.MINYEAR))
      System.out.println("You entered a 'pre-modern' date.");
    else
    {
      System.out.println("The number of days between");
      System.out.print(d1);
      System.out.print(" and ");
      System.out.print(d2);
      System.out.print(" is ");
      System.out.println(Math.abs(d1.lilian() - d2.lilian()));
    }
  }
}
```

1.2 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.



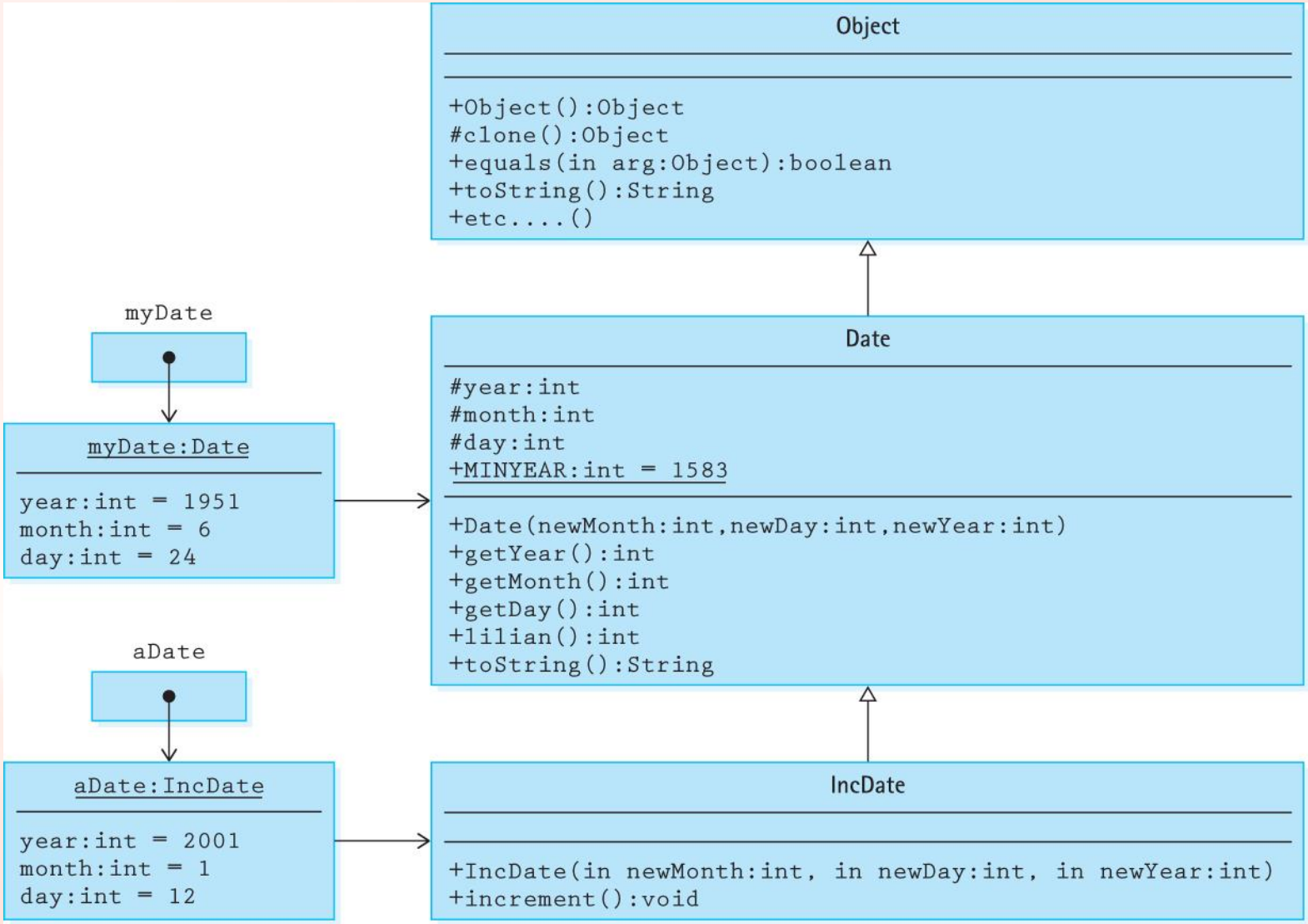
Example of Inheritance

```
package ch01.dates;

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.



Java's Inheritance Tree

- Java supports single inheritance only.
- Method calls are resolved by searching *up* the inheritance tree.
- All Java classes can be traced up to the Object class.
- We use the `@Override` notation to indicate the redefinition of an inherited method.

Inheritance-Based Polymorphism

- Polymorphism – an object variable can reference objects of *different* classes at *different* times.
- A variable declared to be of "type" `T` can hold a reference to an object of type `T` or of any descendant of `T`.
- See example, next slide.

```
// cutoff is random int between 1 and 100
Object obj;
if (cutoff <= 50)
    obj = new String("Hello");
else
    obj = new Date(1,1,2015);
System.out.println(obj.toString());
```

What does the above code print?

We cannot predict. The binding of the `obj` variable to a class (`String` or `Date`) occurs dynamically, at run time.

`obj` is a polymorphic object.

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 ch02.stacks.*;
```

1.3 Exceptional Situations

- **Exceptional situation** Associated with an unusual, sometimes unpredictable event, detectable by software or hardware, which requires special processing. The event may or may not be erroneous.
- For example:
  - a user enters an input value of the wrong type
  - while reading information from a file, the end of the file is reached
  - an impossible operation is requested of an object, such as an attempt to access information that is not yet available

Exceptions with Java

- The Java exception mechanism has three major parts:
  - Defining the exception – usually as a subclass of Java's `Exception` class
  - Generating (raising) the exception – by recognizing the exceptional situation and then using Java's `throw` statement to "announce" that the exception has occurred
  - Handling the exception – using Java's `try` – `catch` statement to discover that an exception has been thrown and then take the appropriate action

Exceptions and ADTs
An Example

- We create a new date related class called `SafeDate` that includes a constructor which throws an exception if it is passed an illegal date
- First, we create our own exception class:

```
public class DateOutOfBoundsException extends Exception
{
  public DateOutOfBoundsException()
  {
    super();
  }

  public DateOutOfBoundsException(String message)
  {
    super(message);
  }
}
```

Here is an example of a constructor that throws the exception:

```
public SafeDate(int newMonth, int newDay, int newYear)
           throws DateOutOfBoundsException
{
  if ((newMonth <= 0) || (newMonth > 12))
    throw new DateOutOfBoundsException("month " + newMonth + "out of range");
  else
    month = newMonth;

  day = newDay;

  if (newYear < MINYEAR)
    throw new DateOutOfBoundsException("year " + newYear +
                                       " is too early");
  else
    year = newYear;
 }
```

Example of a program that throws the exception out to interpreter:

```
public class UseSafeDate
{
  public static void main(String[] args)
                          throws DateOutOfBoundsException
  {
    SafeDate theDate;
    // Program prompts user for a date
    // M is set equal to user's month
    // D is set equal to user's day
    // Y is set equal to user's year
    theDate = new SafeDate(M, D, Y);

    // Program continues ...
  }
}
```

The interpreter will stop the program and print an "exception" message, for example

```
Exception in thread "main" DateOutOfBoundsException: year 1051 is too early
        at SafeDate.<init>(SafeDate.java:18)
        at UseSafeDate.main(UseSafeDate.java:57)
```

An example of a program that catches and handles the exception:

```
public class UseSafeDate
{
  public static void main(String[] args)
  {
    SafeDate theDate;
    boolean DateOK = false;

    while (!DateOK)
    {
      // Program prompts user for a date
      // M is set equal to user's month,
      // D is set equal to user's day
      // Y is set equal to user's year
      try
      {
        theDate = new SafeDate(M, D, Y);
        DateOK = true;
      }
      catch(DateOutOfBoundsException DateOBExcept)
      {
        output.println(DateOBExcept.getMessage());
      }
    }

    // Program continues ...
  }
}
```

General guidelines for using exceptions

- An exception may be handled any place in the software hierarchy—from the place in the program module where it is first detected through the top level of the program.
- Unhandled built-in exceptions carry the penalty of program termination.
- Where in an application an exception is handled is a design decision; however, exceptions should always be handled at a level that knows what the exception means.
- An exception need not be fatal.
- For non-fatal exceptions, the thread of execution can continue from various points in the program, but execution should continue from the lowest level that can recover from the exception.

Java `RunTimeException` class

- Exceptions of this class are thrown when a standard run-time program error occurs.
- Examples of run-time errors are division-by-zero and array-index-out-of-bounds.
- These exceptions can happen in virtually any method or segment of code, so we are not required to explicitly handle these exceptions.
- These exceptions are classified as **unchecked exceptions**.

Error Situations and ADTs

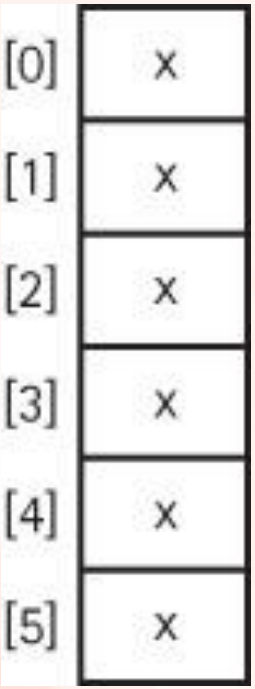
- When dealing with error situations within our ADT methods, we have several options:
  - Detect and handle the error within the method itself. Best approach if the error can be handled internally.
  - Detect the error within the method, throw an exception related to the error and force the calling method to deal with the exception. If not clear how to handle an error situation, this approach might be best … throw it out to a level where it can be handled.
  - Ignore the error situation. With this approach, if the preconditions of a method are not met, the method is not responsible for the consequences.

1.4 Data Structures

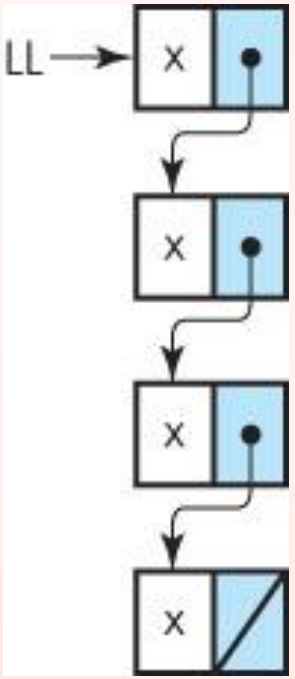
- The way you view and structure the data that your programs manipulate greatly influences your success.
- A language's set of primitive types (Java's are byte, char, short, int, long, float, double, and boolean) are not sufficient, by themselves, for dealing with data that have many parts and complex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

Implementation Dependent Structures

Array

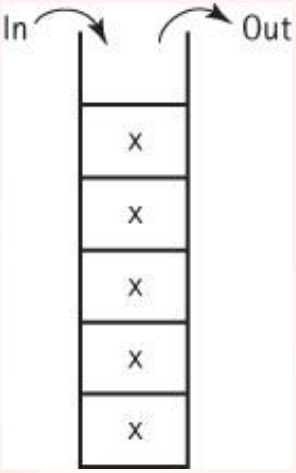


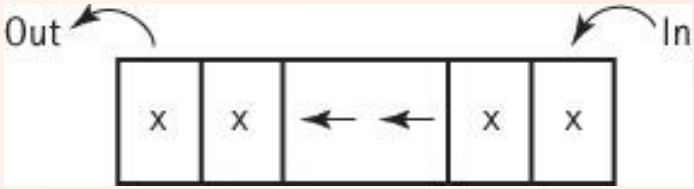
Linked List



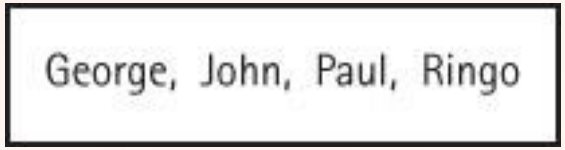
Implementation Independent Structures

Stack

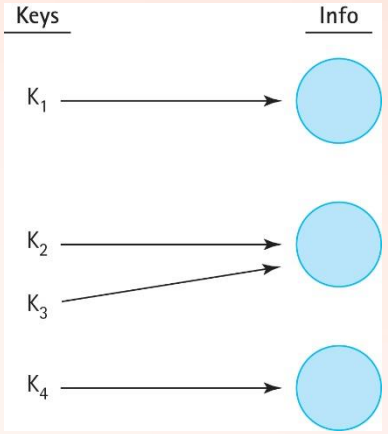




Queue

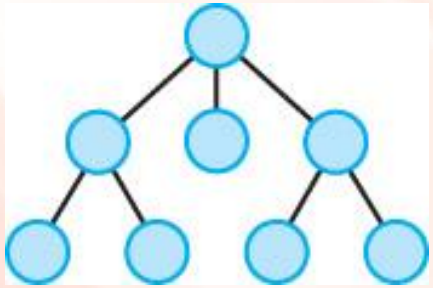


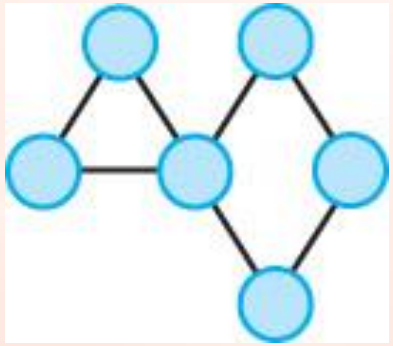
Sorted List



Map

Tree

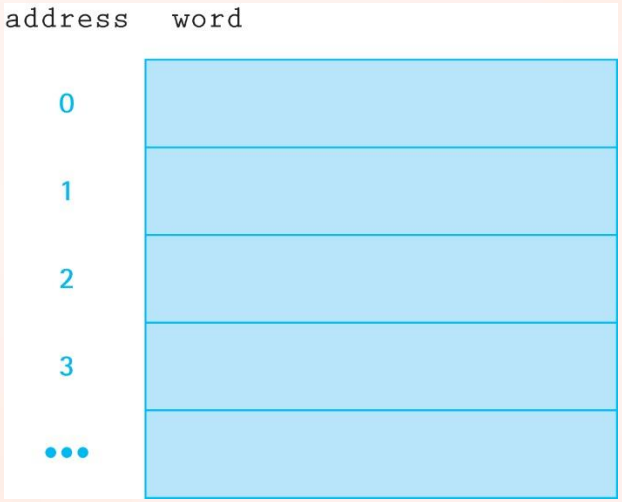




Graph

1.5 Basic Structuring Mechanisms

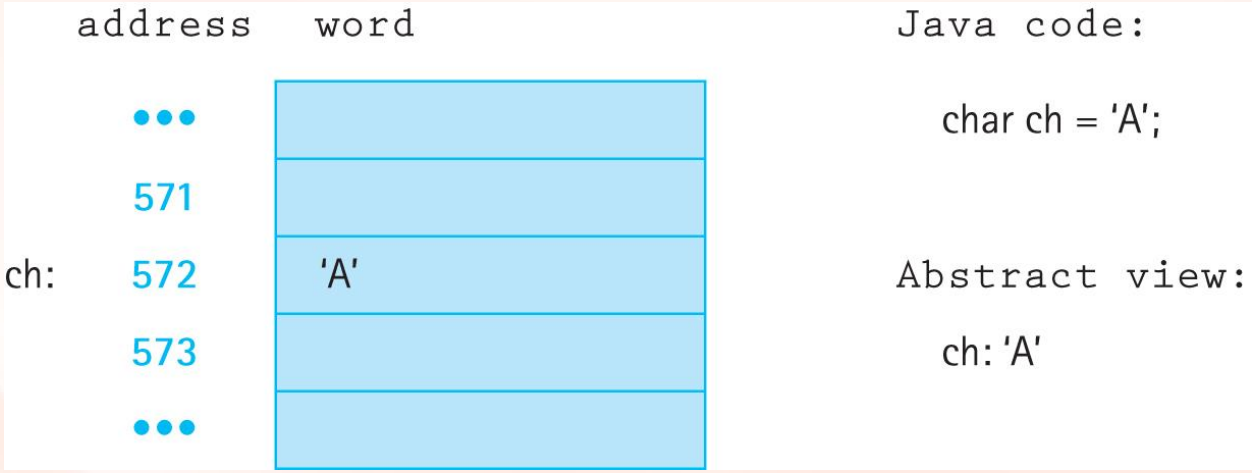
- All programs and data are held in memory.
- Memory consists of a contiguous sequence of addressable words:



- A variable in our program corresponds to a memory location.

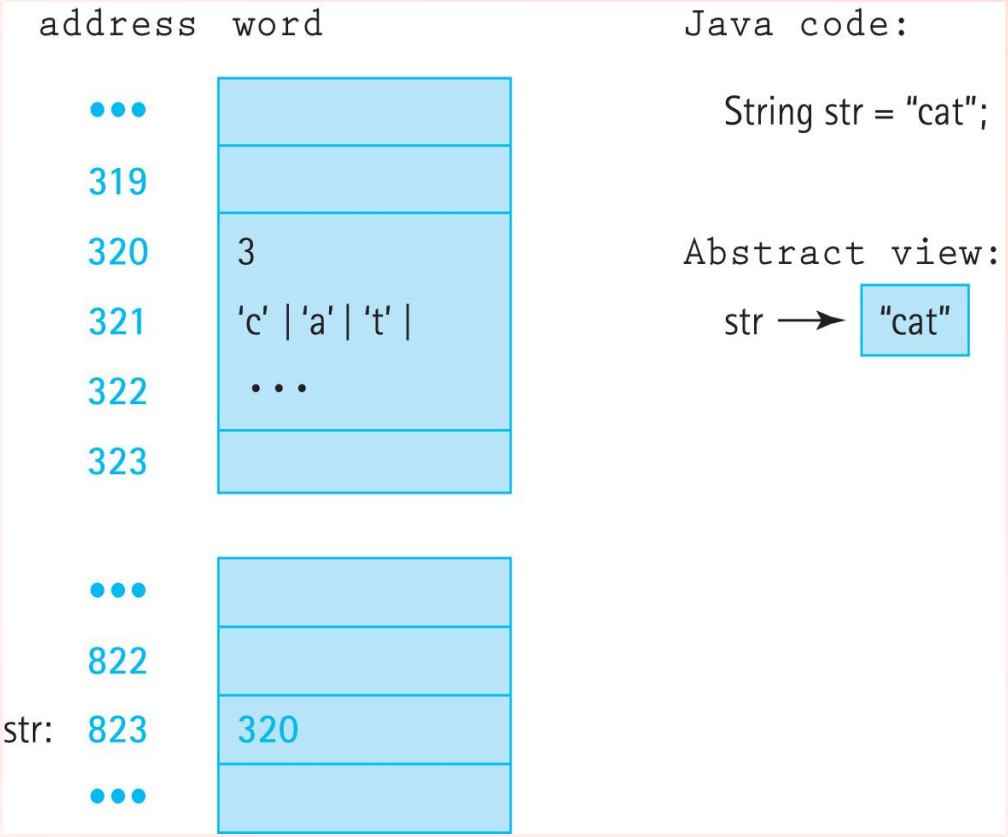
Direct Addressing …

- … is when the memory location associated with the variable holds the value of the variable.
- This corresponds to how primitive variables are used in Java:



Indirect Addressing …

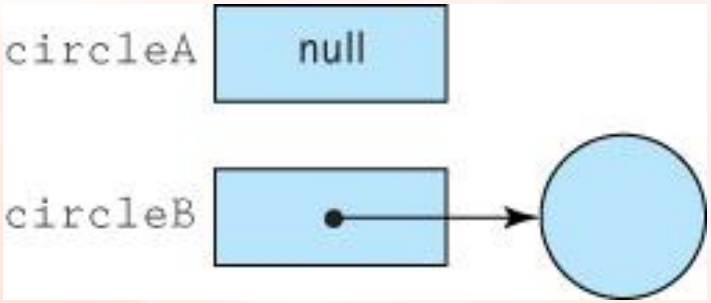
- … is when the memory location associated with the variable holds the address of the location that holds the value of the variable.
- This corresponds to how reference variables (objects) are used in Java



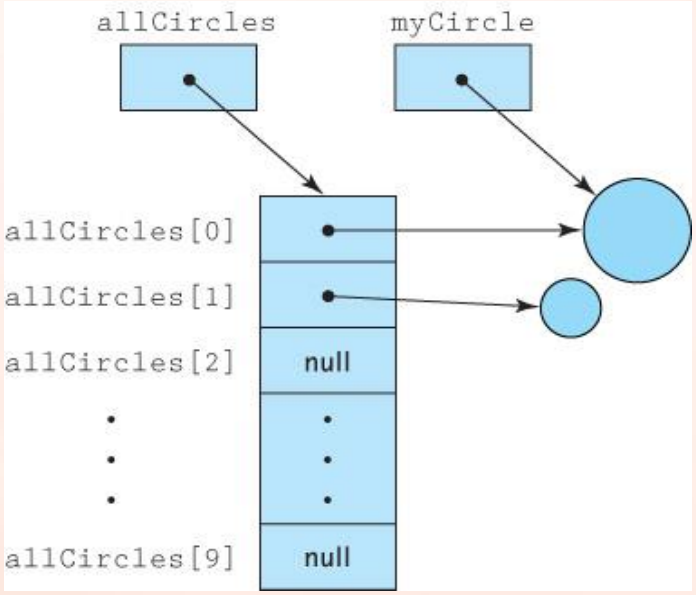
The Two Basic Structuring Mechanisms

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

References



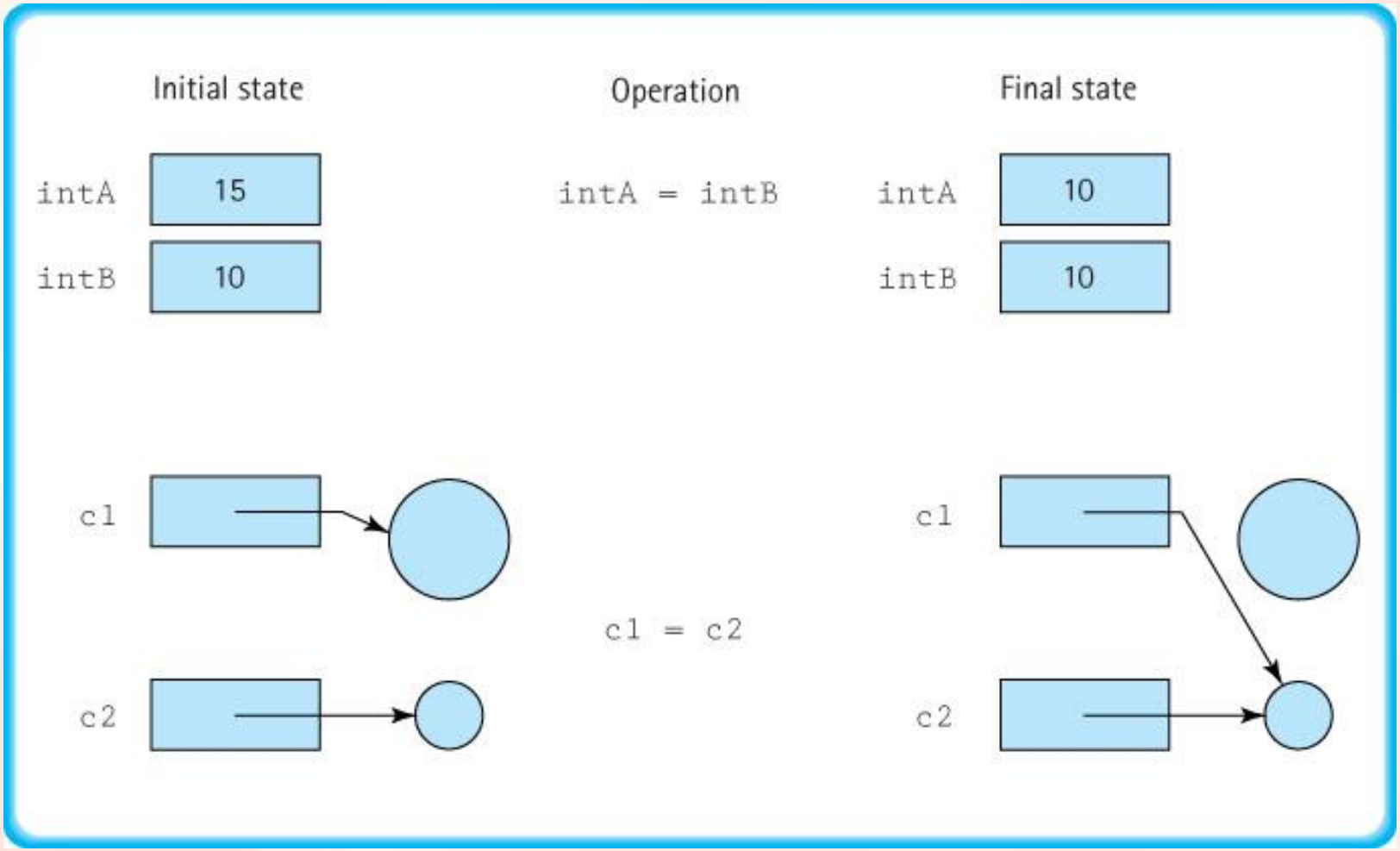
Arrays



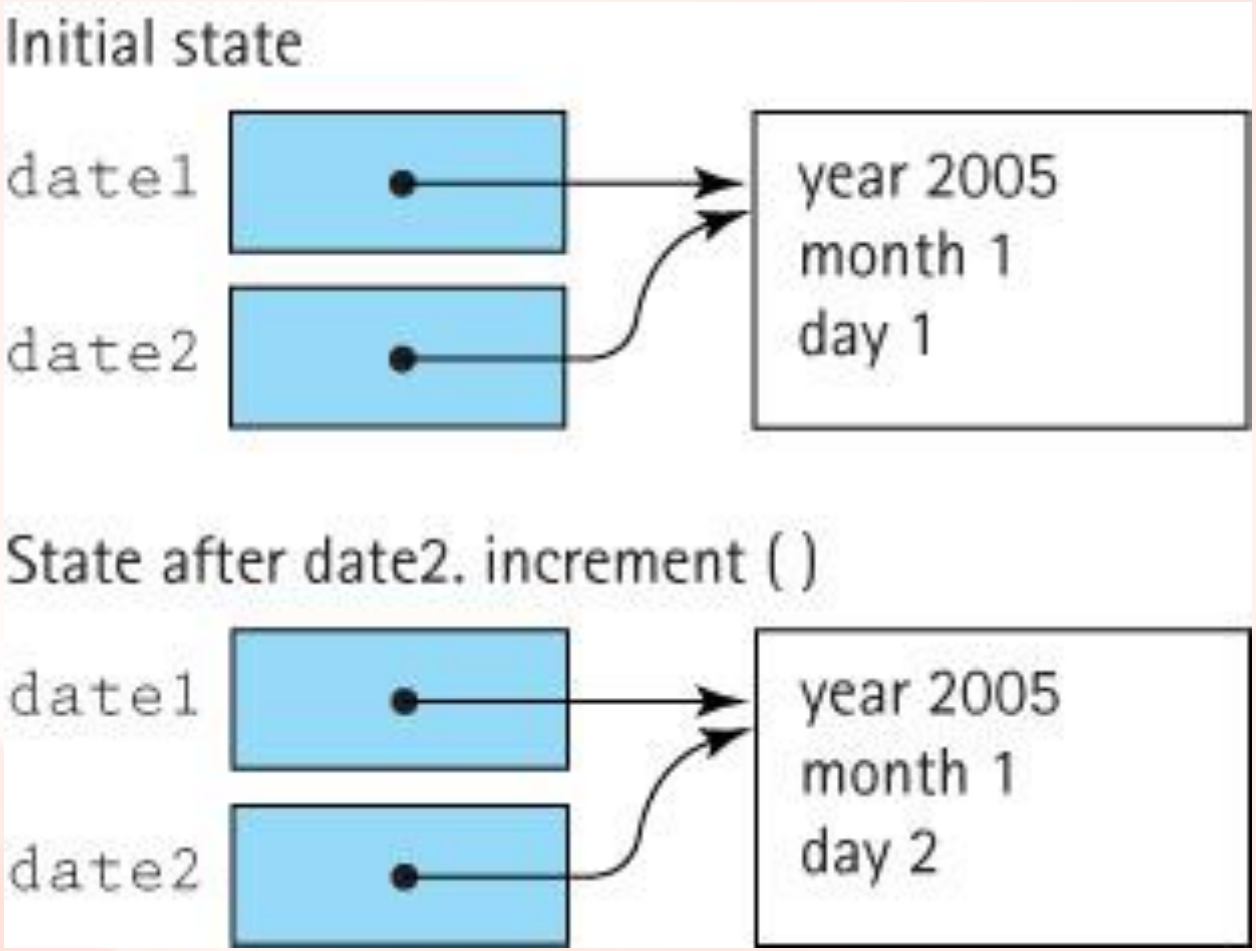
References

- Are memory addresses (use indirect addressing)
- 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 …

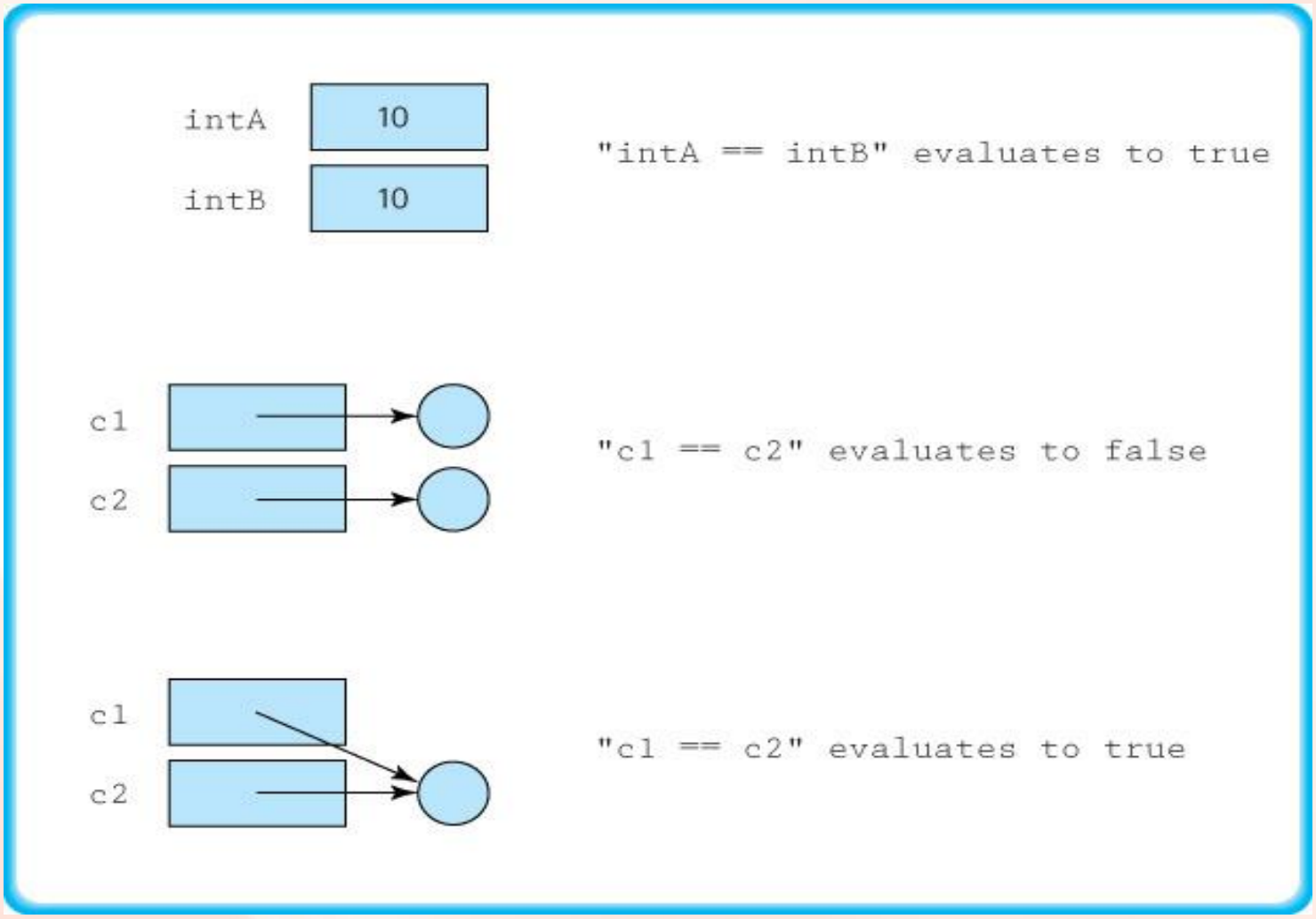
Assignment Statements



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 38 to 43 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

1.6 Comparing Algorithms: Order of Growth Analysis

- Alice: “I’m thinking of a number between 1 and 1,000.”
- Bob: “Is it 1?”
- Alice: “No . . . it’s higher.”
- Bob: “Is it 2?”
- Alice: “No . . . it’s higher.”
- Bob: “Is it 3?”
- Alice: rolls her eyes . . .

Sequential Search Algorithm

Algorithms

- A sequence of unambiguous instructions that solve a problem, within a finite amount of time, given a set of valid input
- Analysis of algorithms is important area of computer science
- The efficiency of algorithms and the code that implements them can be studied in terms of both time (how fast it runs) and space (the amount of memory required).

time and space can be inter-related

Counting Operations

- To measure the complexity of an algorithm we attempt to count the number of basic operations required to complete the algorithm
- Rather than count all operations, select a fundamental operation, an operation that is performed “the most”, and count it.

Counting Operations Example

Problem: guess secret number between 1 and 1,000 (the Hi-Lo game)

```
Hi-Lo Sequential Search:
  Set guess to 0
  do
    Increment guess by 1
    Announce guess
  while (guess is not
         correct)
```

- Number of operations:
  - Depends on how "lucky" you are

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**

Counting Operations Example

Problem: guess secret number between 1 and 1,000 (the Hi-Lo game)

```
Hi-Lo Sequential Search:
  Set guess to 0
  do
    Increment guess by 1
    Announce guess
  while (guess is not
         correct)
```

- Number of operations worst case is:
  - 1 to set guess
  - 1000 increments
  - 1000 announces
  - 1000 tests for correctness
  - Total: 3001 ….. but
- This is too dependent on
  - counting approach
  - language
  - language level
  - compiler
- and is difficult for more complex algorithms

Isolate a fundamental operation

- Rather than count all operations, select a fundamental operation, an operation that is performed “the most”, and count it.
- For the “guess the number” problem a basic operation might be “announce guess” …
  - count is 1,000

- But what if problem size changes:
  - “I’m thinking of a number between 1 and 1 million!”

Size of Input

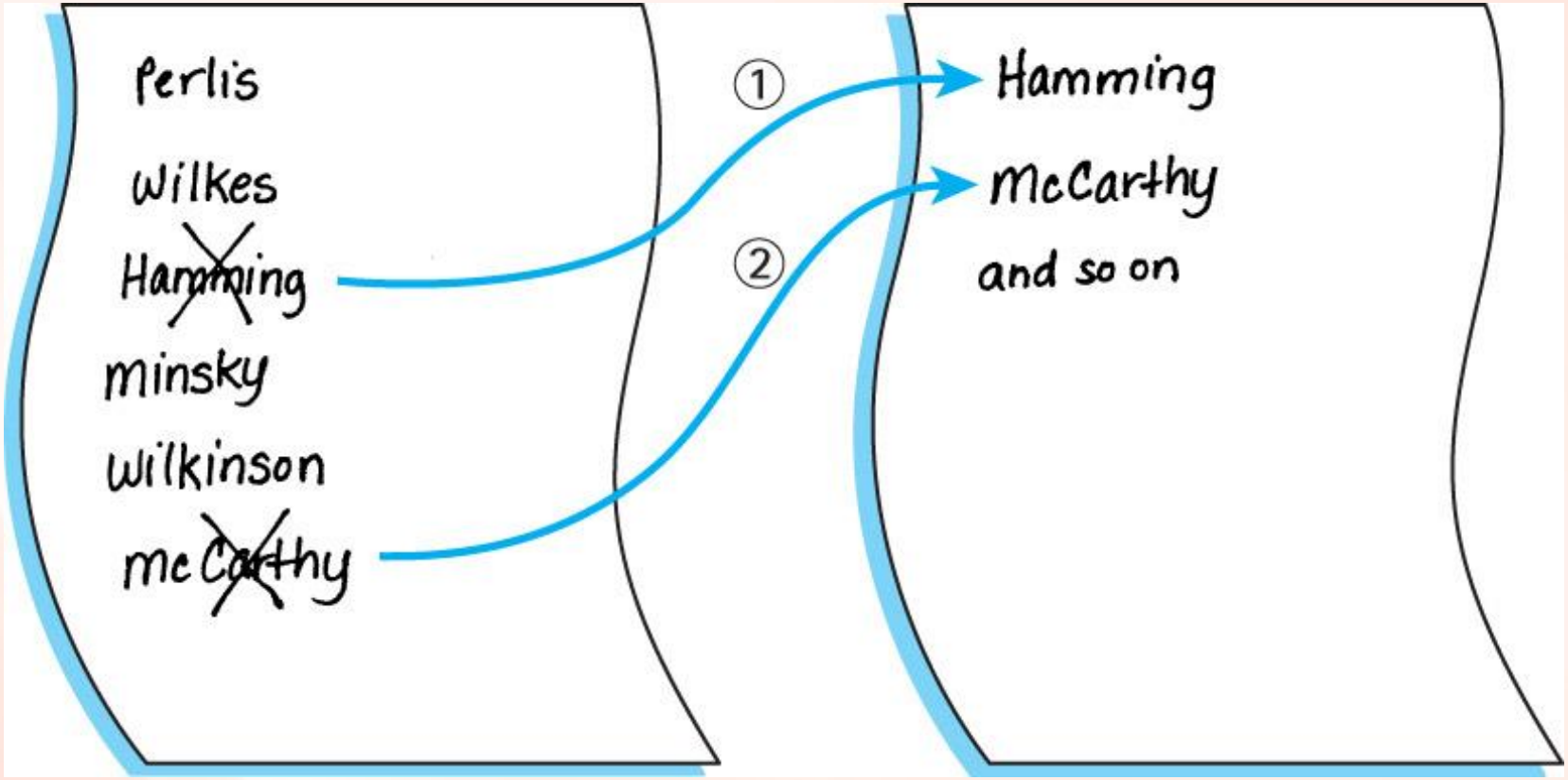
- To make our count generally useable we express it as a function of the size of the problem
- “I’m thinking of number between 1 and N”
  - Sequential Search: N “announces”
  - Binary Search (cut search area in half at each step): $\log_2 N$ “announces”

A further simplification: Order of Growth 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.
- Order of growth 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.

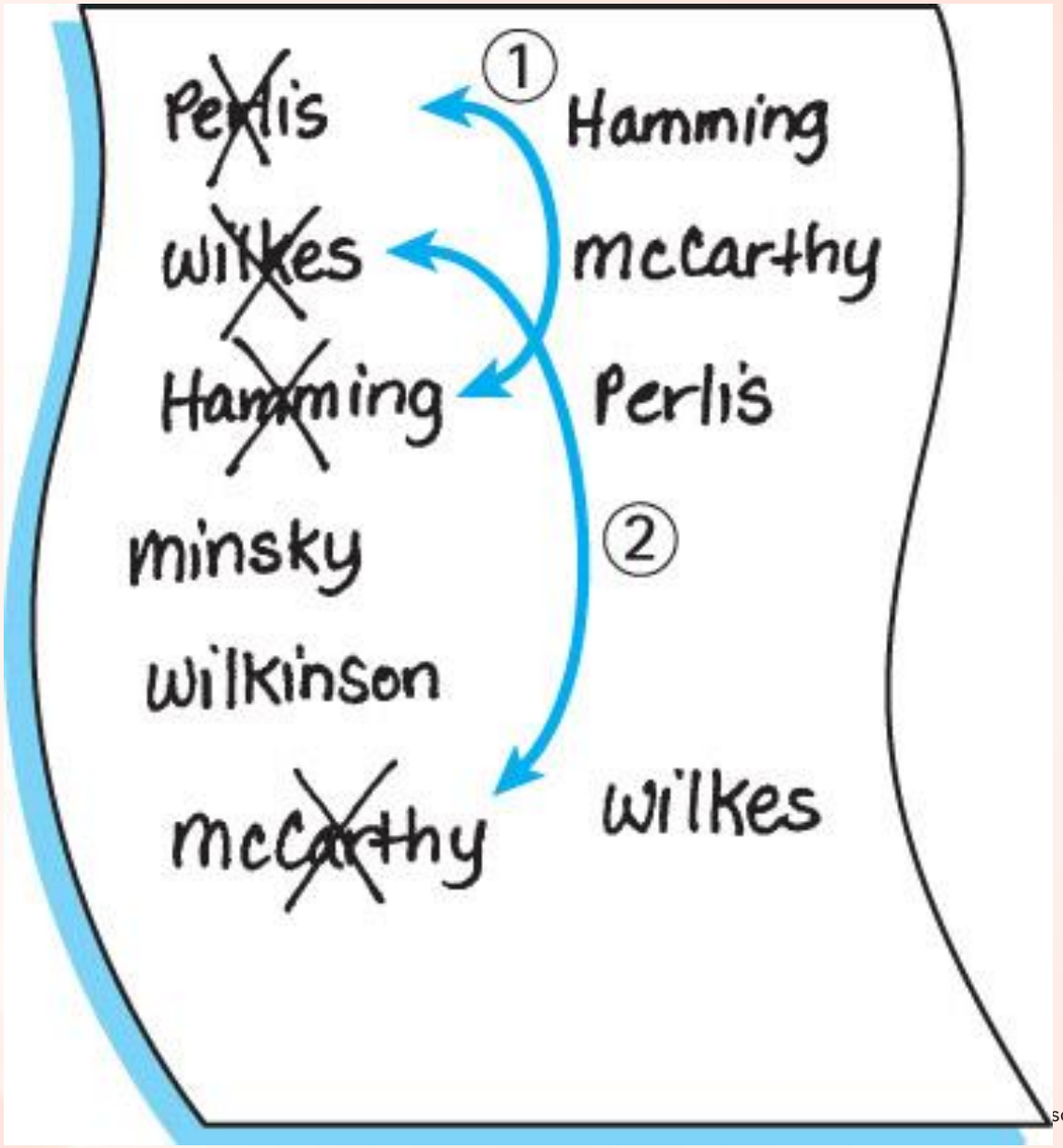
Selection Sort

- If we were handed a list of names on a sheet of paper and asked to put them in alphabetical order, we might use this general approach:
  - *Select* the name that comes first in alphabetical order, and write it on a second sheet of paper.
  - Cross the name out on the original sheet.
  - Repeat steps 1 and 2 for the second name, the third name, and so on until all the names on the original sheet have been crossed out and written onto the second sheet, at which point the list on the second sheet is sorted.



An improvement

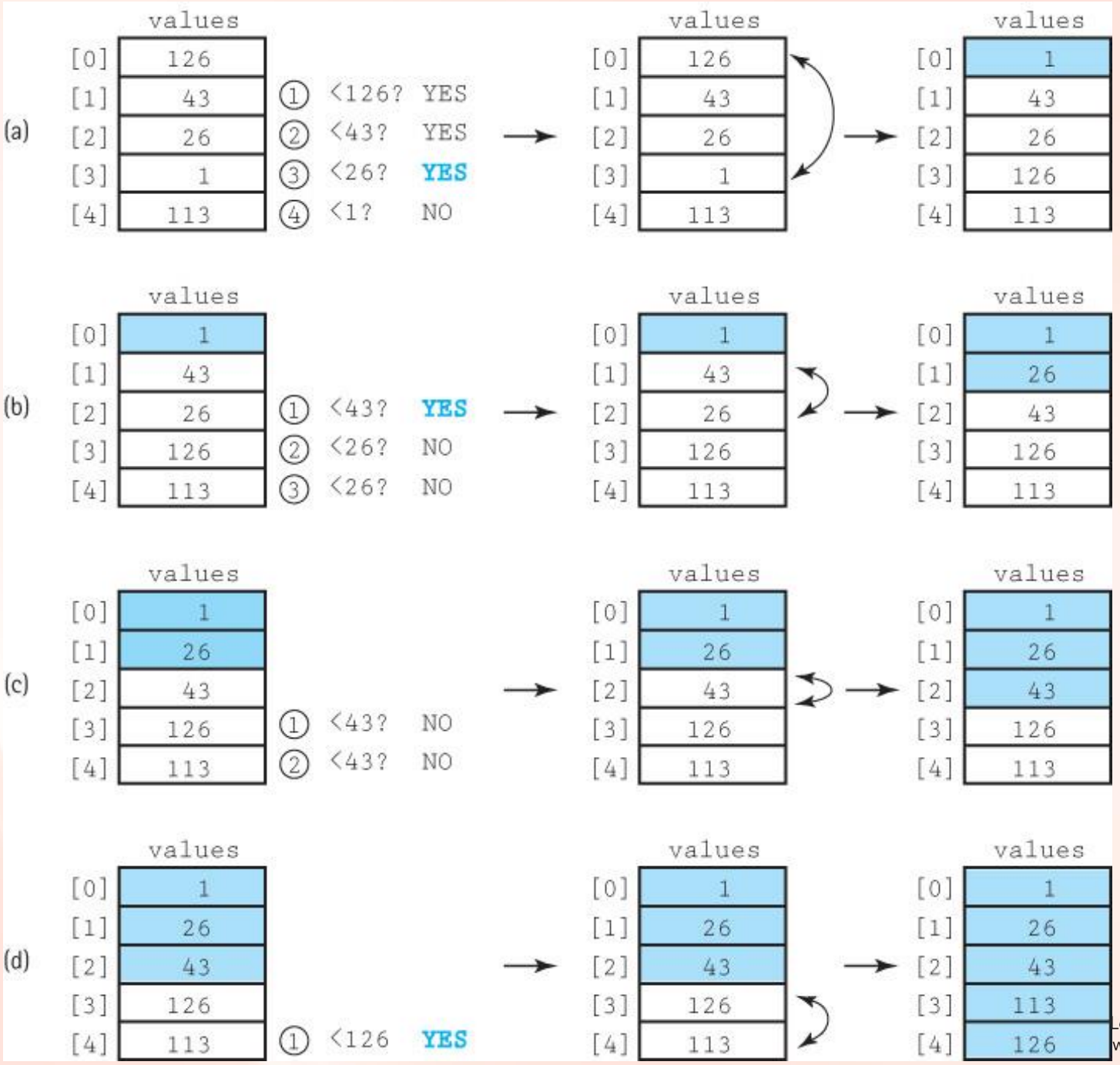
- Our algorithm is simple but it has one drawback: It requires space to store two complete lists.
- Instead of writing the “first” name onto a separate sheet of paper, exchange it with the name in the first location on the original sheet. And so on.



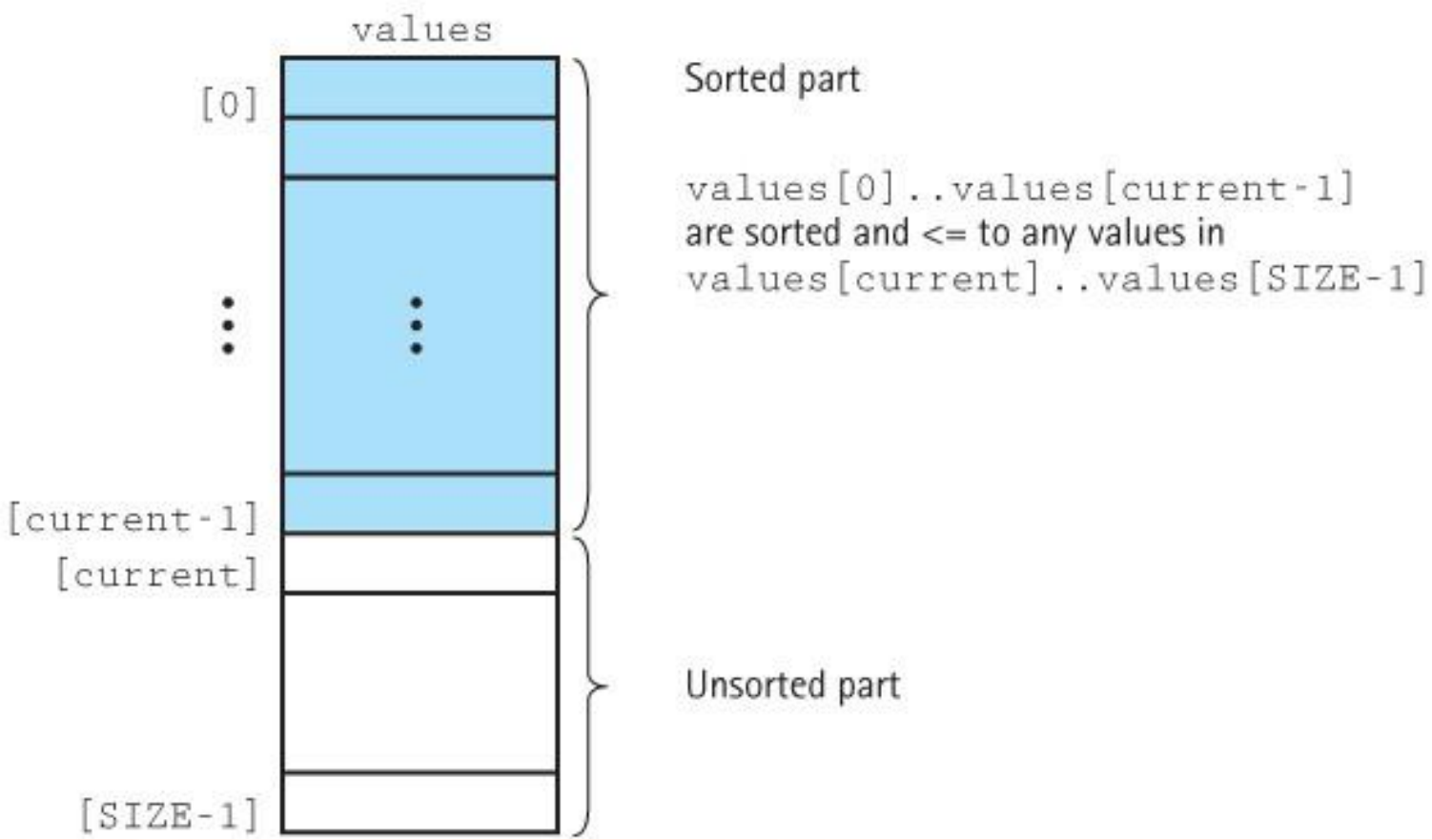
Selection Sort Algorithm

SelectionSort
for current going from 0 to SIZE - 2
    Find the index in the array of the smallest unsorted element
    Swap the current element with the smallest unsorted one

An example is depicted on the following slide …



Selection Sort Snapshot



Selection Sort Code

```
static int minIndex(int startIndex, int endIndex)
// Returns the index of the smallest value in
// values[startIndex]..values[endIndex].
{
  int indexOfMin = startIndex;
  for (int index = startIndex + 1; index <= endIndex; index++)
    if (values[index] < values[indexOfMin])
      indexOfMin = index;
  return indexOfMin;
}

static void selectionSort()
// Sorts the values array using the selection sort algorithm.
{
  int endIndex = SIZE - 1;
  for (int current = 0; current < endIndex; current++)
    swap(current, minIndex(current, endIndex));
}
```

Selection Sort Analysis

- We describe the number of comparisons as a function of the number of elements in the array, i.e., `SIZE`. To be concise, in this discussion we refer to SIZE as *N*
- The `minIndex` method is called *N* - 1 times
- Within `minIndex`, the number of comparisons varies:
  - in the first call there are *N* - 1 comparisons
  - in the next call there are *N* - 2 comparisons
  - and so on, until in the last call, when there is only 1 comparison
- The total number of comparisons is

  $(N - 1) + (N - 2) + (N - 3) + \ldots + 1$

  $= N(N - 1)/2 = 1/2N^2 - 1/2N$
- The Selection Sort algorithm is $O(N^2)$

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 11, have N log N complexity.
- $O(N^2)$ is called quadratic time. Some simple sorting algorithms such as Selection Sort 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**

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 Order of Growth complexity
  - especially when interested in “large” problems