Object-Oriented Data Structures Using Java

# Chapter 2

## Data Design and Implementation

# Data

- The representation of information in a manner suitable for communication or analysis by humans or machines
- Data are the nouns of the programming world:
  - The objects that are manipulated
  - The information that is processed

# Data Type

- A category of data characterized by the supported elements of the category and the supported operations on those elements
- Simple data type examples: integers, real numbers, characters

# Definitions

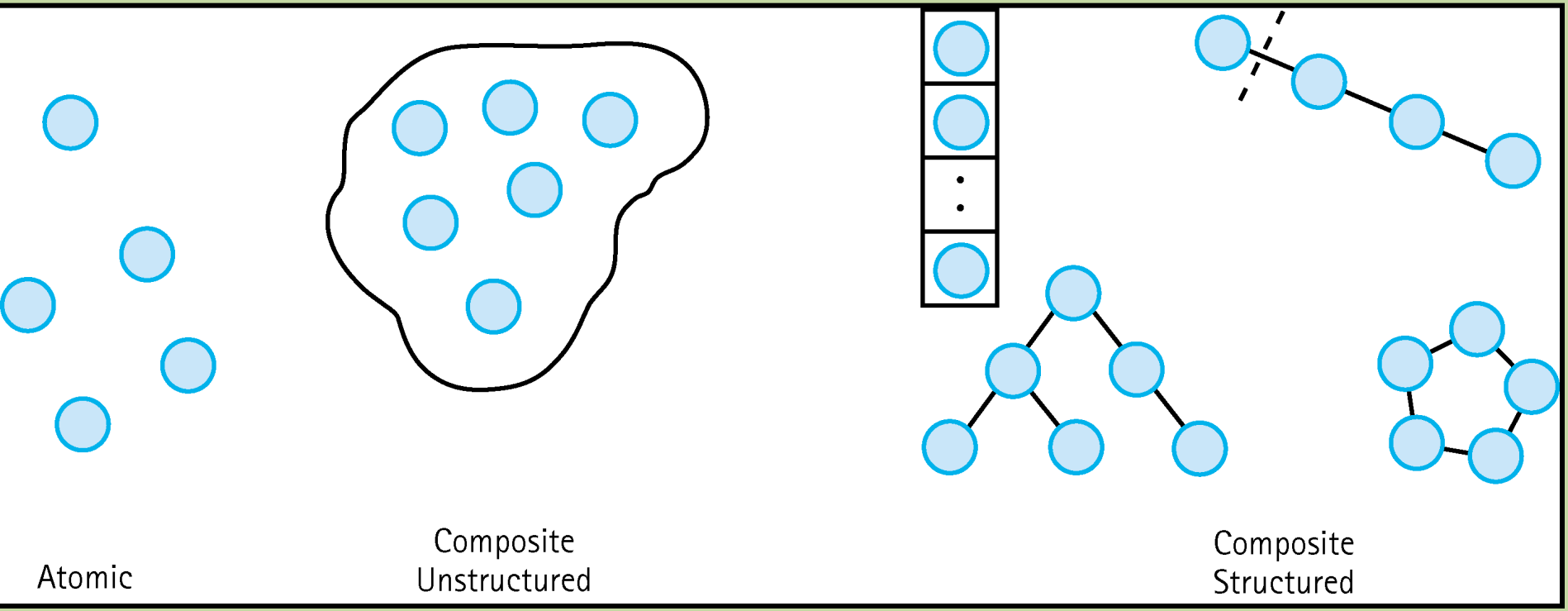
**Atomic or primitive type** A data type whose elements are single, nondecomposable data items

**Composite type** A data type whose elements are composed of multiple data items

# Definitions *(cont'd)*

**Structured composite type** An organized collection of components in which the organization determines the means of accessing individual data components or subsets of the collection

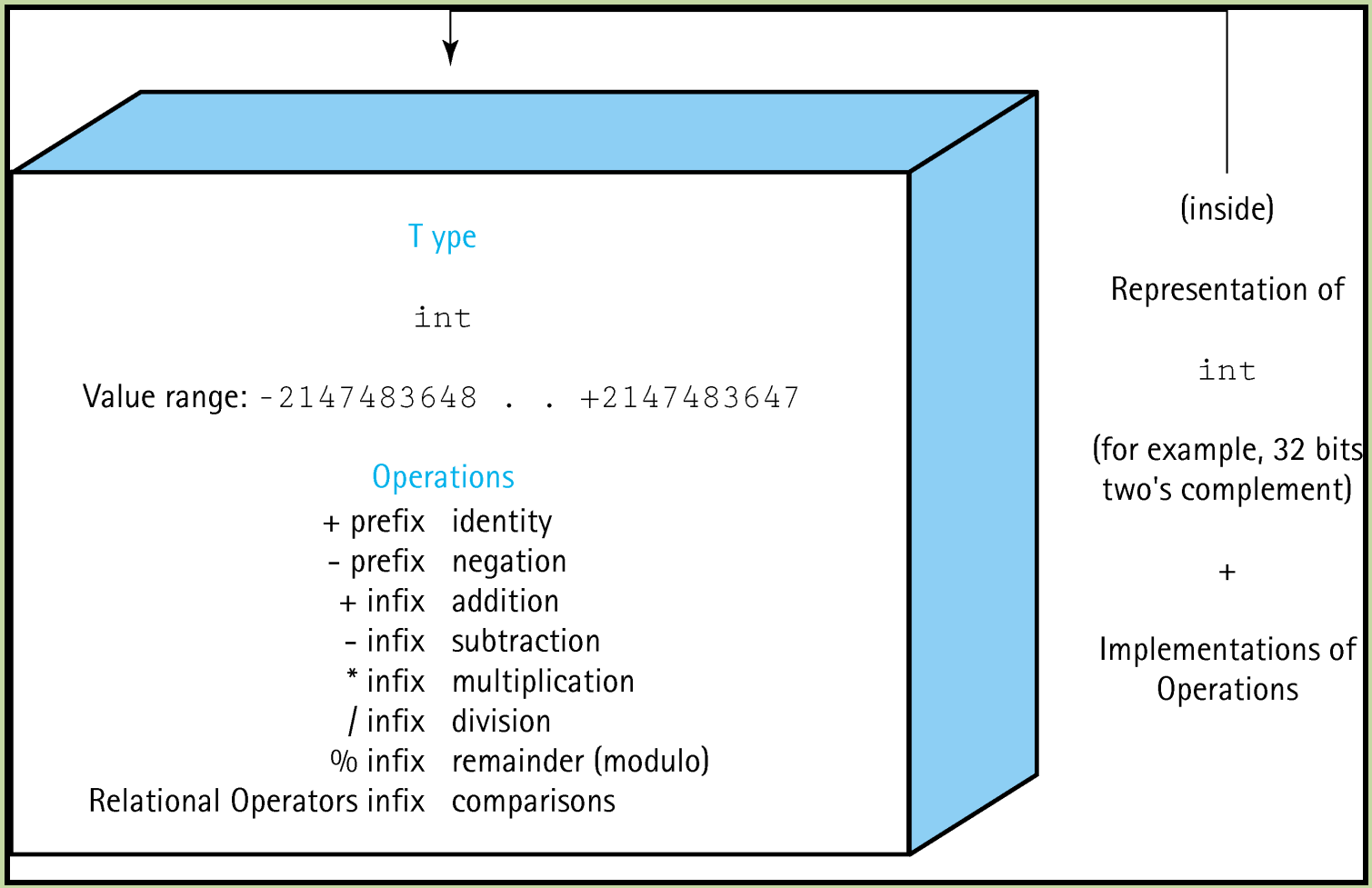
# Atomic (Simple) and Composite Data Types



# Definitions

**Data abstraction** The separation of a data type's logical properties from its implementation

**Data encapsulation** The separation of the representation of data from the applications that use the data at a logical level; a programming language feature that enforces information hiding


Type
int
Value range: -2147483648 . . +2147483647
Operations
+ prefix identity
- prefix negation
+ infix addition
- infix subtraction
* infix multiplication
/ infix division
% infix remainder (modulo)
Relational Operators infix comparisons
(inside)
Representation of
int
(for example, 32 bits two's complement)
+
Implementations of Operations

# Abstract Data Type (ADT)

**Abstract data type (ADT)** A data type whose properties (domain and operations) are specified independently of any particular implementation

- In effect, all Java built-in types are ADTs.
- In addition to the built-in ADTs, Java programmers can use the Java *class* mechanism to build their own ADTs.

# Data Structure

- A collection of data elements whose logical organization reflects a relationship among the elements
- It is best to design data structures with ADTs.

# Basic Operations on Encapsulated Data

- A constructor is an operation that creates a new instance (object) of the data type.
- Transformers (sometimes called *mutators*) are operations that change the state of one or more of the data values.

# Basic Operations on Encapsulated Data

- An observer is an operation that allows us to observe the state of one or more of the data values with out changing them.
- An iterator is an operation that allows us to process all the components in a data structure sequentially.

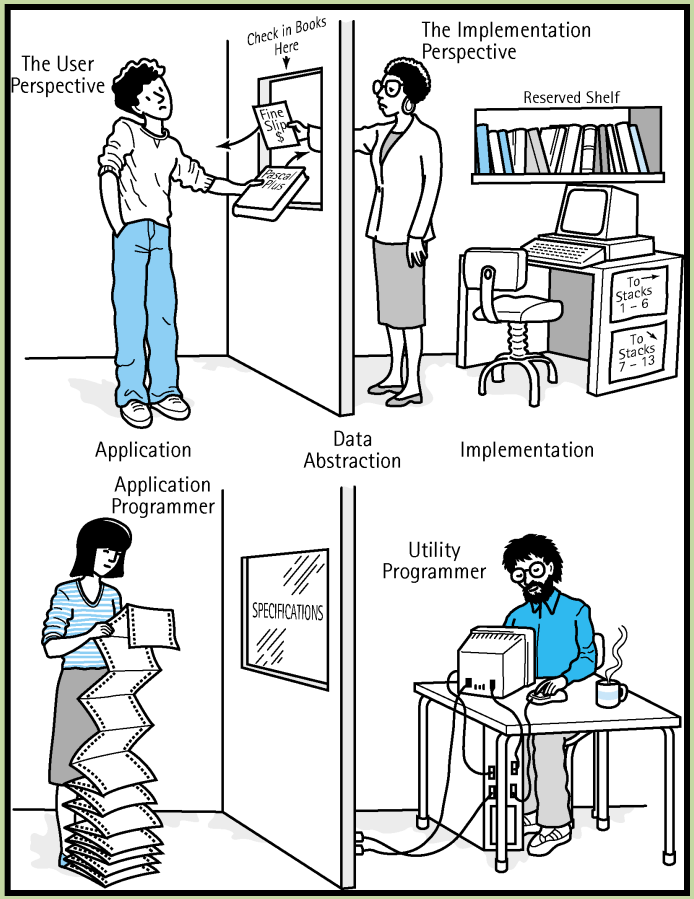
# Data Levels of an ADT

- *Logical (or abstract) level*: Abstract view of the data values (the domain) and the set of operations to manipulate them.
- *Application (or user) level*: Here the application programmer uses the ADT to solve a problem.

# Data Levels of an ADT *(Cont'd)*

- *Implementation level*: A specific representation of the structure to hold the data items, and the coding of the operations in a programming language.

# Communication between the Application Level and Implementation Level



# Java's Built-In Types

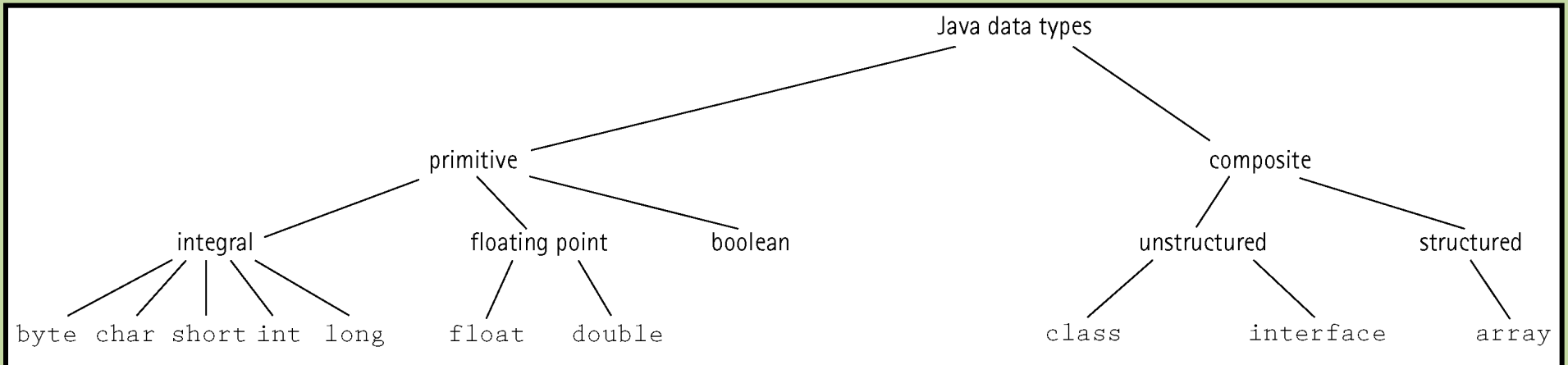



Figure 2.5 *Java data types*

# The Java *Class* Construct

- Can be a mechanism for creating composite data types.
- Is composed of named data fields (class and instance variables) and methods.
- Is unstructured because the meaning is not dependent on the ordering of the members.

# Records

**Record** When a class is used strictly to hold data and does not hide the data.

- A record is a composite data type made up of a finite collection of not necessarily homogeneous elements called fields.
- Accessing is done directly through a set of named field selectors.

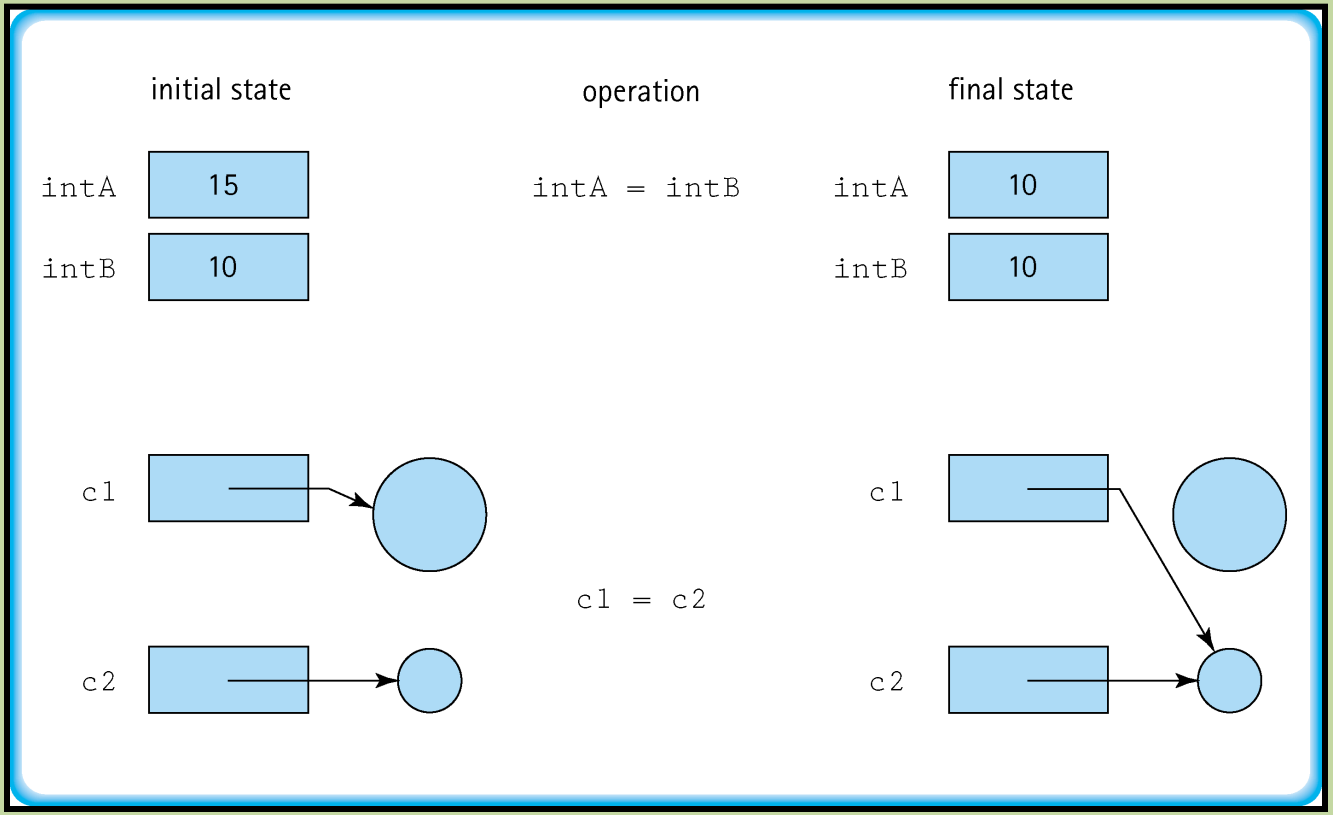
Defines a Circle Record (object)

```
public class TestCircle
{
  static class Circle
  {
    int xValue;       // Horizontal position of center
    int yValue;       // Vertical position of center
    float radius;
    boolean solid;    // True means circle filled
  }

  public static void main(String[] args)

  {
    Circle c1 = new Circle();
    c1.xValue = 5;
    c1.yValue = 3;
    c1.radius = 3.5f;
    c1.solid = true;

    System.out.println("c1:   " + c1);
    System.out.println("c1 x: " + c1.xValue);
  }
}
```

Creates an instance of a circle record (object)

Accesses a field value

# Assignment Statement: Primitive Types versus Objects



# Ramifications of Using References

- Aliases—we have two names for the same object.

```
output.println(date1);
date2.increment();
output.println(date1);
```

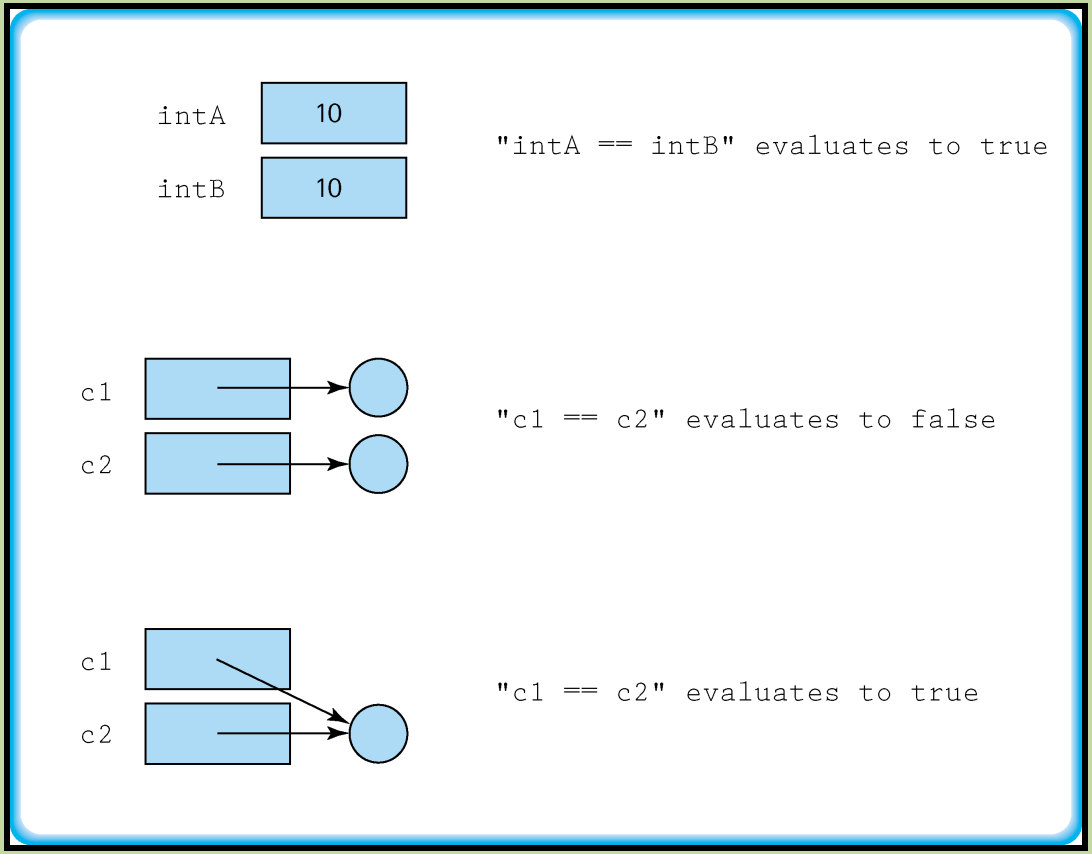
# Ramifications of Using References

- **Garbage** Memory space that has been allocated to a program but can no longer be accessed by a program.

```
Circle c1;
for (n = 1; n <= 100; n++)
{
  Circle c1 = new Circle();
  // code to initialize and use c1 goes here
}
```

- **Garbage collection** The process of finding all unreachable objects and deallocating their storage space

# Ramifications of Using References



# Ramifications of Using References

## Parameter Passing

- All Java arguments are passed by value.
- If the variable is of a primitive type, the actual value (`int`, `double`, and so on) is passed to the method.
- If it is a reference type, then the reference that it contains is passed to the method.

# Interfaces

1. All variables declared in an interface must be `final`, `static` variables.
2. All methods declared in an interface must be abstract.
3. **Abstract method** A method declared in a class or an interface without a method body.

# An Example of an Interface

```
public interface FigureGeometry
{
  public static final float Pi = 3.14;

  public abstract float perimeter();
  // Returns perimeter of current object

  public abstract float area();
  // Returns area of current object

  public abstract int weight(int scale);
  // Returns weight of current object
}
```

Interfaces can be used in the following ways:

- *As a contract*—Capture an abstract view in an interface.
- *To share constants*—Define constants in an interface and have each class implement the interface.

- *To replace multiple inheritance*—A class can extend one superclass, but it can implement many interfaces.
- *To provide a generic type mechanism*—In Chapter 3 you learn how to use the Java interface construct to provide generic structures.

# Arrays

An array differs from a class in the following three ways:

1. An array is a homogenous structure, whereas classes are heterogeneous structures.
2. A component of an array is accessed by its position in the structure, whereas a component of a class is accessed by an identifier (the name).
3. Because array components are accessed by position, an array is a structured composite type.

# Ways of Combining our Built-In Types and Classes into Versatile Hierarchies

1. **Aggregate objects**—The instance variables of our objects can themselves be references to objects.
2. **Arrays of objects**—Simply define an array whose components are objects.

# Ways of Combining our Built-In Types and Classes into Versatile Hierarchies *(Cont'd)*

3. **Two-dimensional arrays**—To represent items in a table with rows and columns.
4. **Multilevel Hierarchies**—Create whatever sort of structure best matches our data.

Examples…

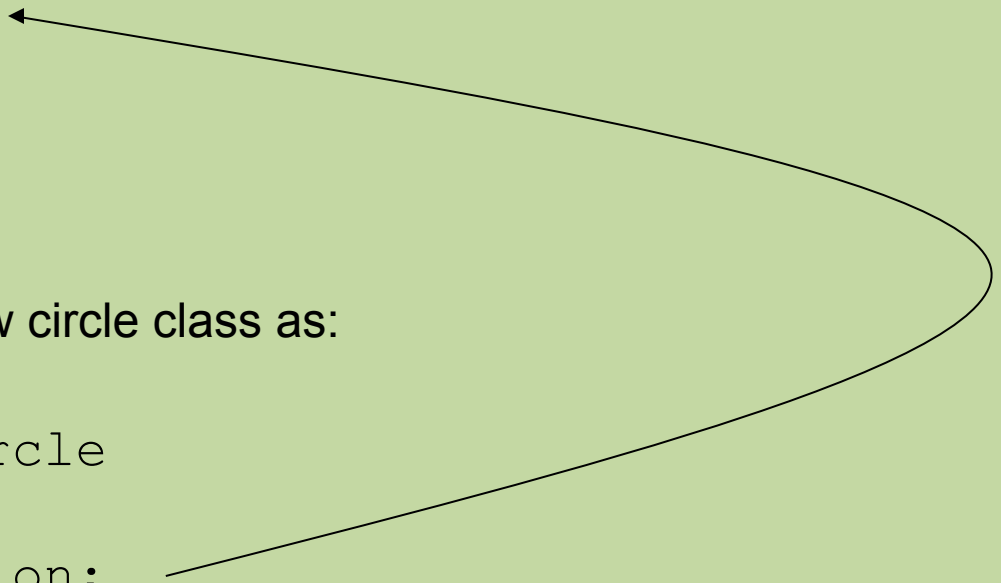
# Example of an Aggregate Object

Suppose we define:

```
public class Point
{
public int xValue
public int yValue
```

Then, we could define a new circle class as:

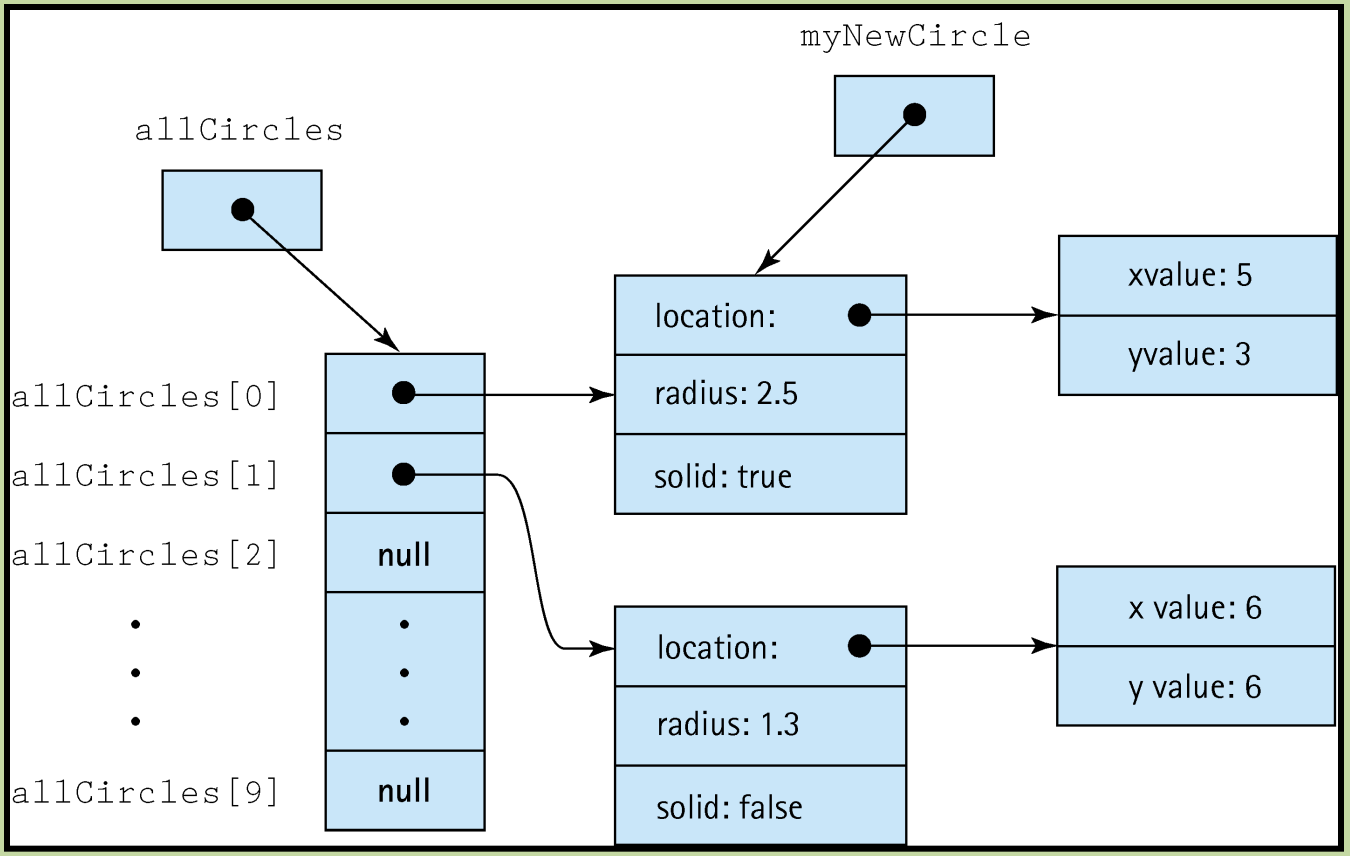
```
Public class NewCircle
{
public Point location;
public float radius;
public boolean solid;
}
```



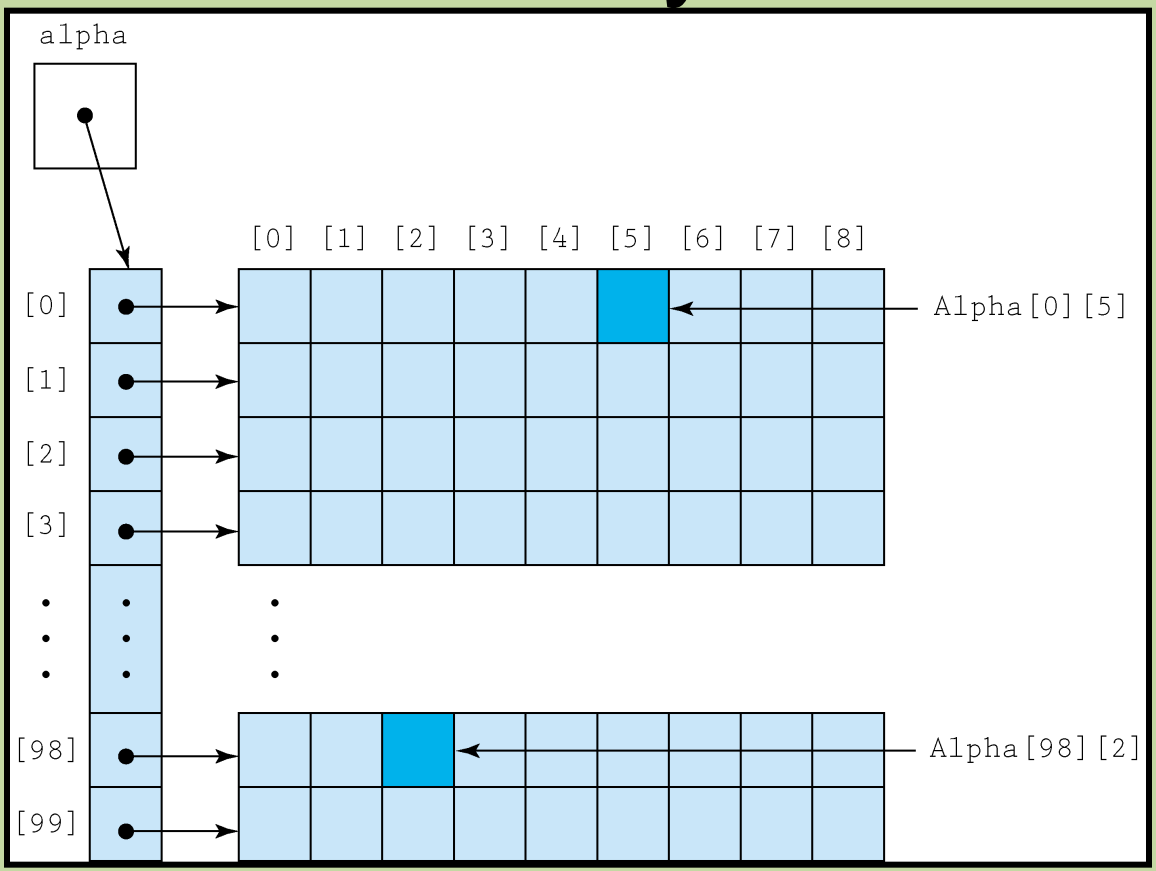
# Example of an Array of Objects

```
NewCircle[] allCircles = new NewCircle[10];
allCircles[0] = new NewCircle();
allCircles[0] = myNewCircle;
allCircles[1] = new NewCircle();
allCircles[1].location = new Point();
allCircles[1].location.xValue = 6;
allCircles[1].location.yValue = 6;
allCircles[1].radius = 1.3f;
allCircles[1].solid = false;
```

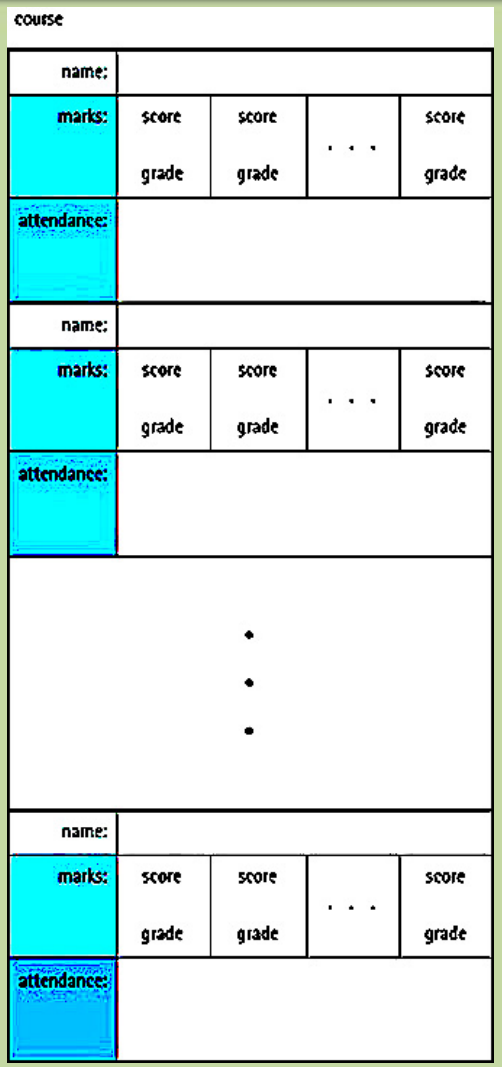
# The `allCircles` Array



# Example of a Two-Dimensional Array



# Example of a Multilevel Hierarchy



# Sources for Classes

- *The Java Class Library*—A class library that includes hundreds of useful classes.
- *Build your own*—You create the needed class, possibly using pre-existing classes in the process.
- *Off the shelf*—Software components, such as classes or packages of classes, which are obtained from third party sources.

# Some Important Library Packages

| Some Important Library Packages | |
|---|---|
| `java.awt` | (Abstract Windowing Toolkit) Contains tools for creating user interfaces, graphics, and images. |
| `java.awt.event` | Provides interfaces and classes for handling the different types of events created by AWT components. |
| `java.io` | System input and output through data streams, serialization, and the file system. |
| `java.lang` | Provides basic classes for use in creating Java programs. |
| `java.math` | Provides classes for performing mathematical operations. |
| `java.text` | Provides classes and interfaces for handling text, dates, numbers, and messages. |
| `java.util` | Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array). |
| `java.util.jar` | Provides classes for reading and writing the JAR (Java ARchive) file format, which is based on the standard ZIP file format. |

# Some Useful General Library Classes

- *The System Class*—To obtain current system properties
- *The Random Class*—to generate a list of random numbers
- *The DecimalFormat Class*—to format numbers for output

# Some Useful General Library Classes

- *Wrappers*—to wrap a primitive valued variable in an object's structure
- *The Throwable and Exception Classes*—An exception is associated with an unusual, often unpredictable event, detectable by software or hardware, that requires special processing.

# Exceptions

- One system unit raises or **throws and exception**, and another unit **catches the exception** and processes (handles) it.
  - **Throw an exception**—Interrupt the normal processing of a program to raise an exception that must be handled or rethrown by surrounding block or code.

# Exceptions Continued

– **Catch an exception**—Code that is executed to handle a thrown exception is said to catch the exception.

# Catching and Handling an Exception

```
try
{
  month = Integer.parseInt(dataFile.readLine());
  day = Integer.parseInt(dataFile.readLine());
  year = Integer.parseInt(dataFile.readLine());
}
catch (IOException readExcp)
{
  outFile.println("There was trouble reading in month, day, year.");
  outFile.println("Exception: " + readExcp.getMessage());
  System.exit();
}
```

# Unchecked Exception

An exception of the `RunTimeException` class, it does not have to be explicitly handled by the method within which it might be raised.

# The `ArrayList` Class

- Part of the `java.util` package.
- Functionality is similar to that of an array.
- However, an array list can grow and shrink; its size is not fixed for its lifetime.

# Array Lists

- Hold variables of type `Object`.
- Capacities grow and shrink, depending on need.
- An array list has a size indicating how many objects it is currently holding, and
- Has a capacity indicating how many elements the underlying implementation could hold without having to be increased.

# `ArrayList` Operations

| Method Name | Parameter Type | Returns | Operation Performed |
|---|---|---|---|
| `ArrayList` | (none) | | Constructs an empty array list of capacity 10. |
| `ArrayList` | `int` | | Constructs an empty array list of the capacity indicated by the parameter. |
| `add` | `int`, `Object` | `void` | Inserts the specified `Object` at the specified position; shifts all subsequent elements to the right one place. |
| `add` | `Object` | `void` | Inserts the specified `Object` at the end. |
| `ensureCapacity` | `int` | `void` | Increases the capacity of the array list to at least the specified capacity, if it is currently less than the specified capacity. |
| `get` | `int` | `Object` | Returns the element at the specified position. |
| `isEmpty` | (none) | `boolean` | Returns `true` if the array list is empty, `false` otherwise. |
| `remove` | `int` | `Object` | Removes the element at the specified position, shifts all subsequent elements to the left one place, and returns the removed element. |
| `size` | (none) | `int` | Returns current size. |
| `trimToSize` | (none) | `void` | Trims the capacity of the array list to its size. |

# Use an array list when

1. Space is an issue.
2. Execution time is not an issue.
3. The amount of space required changes drastically from one execution of the program to the next.
4. The actively used size of an array list changes a great deal during execution.

# Use an array list when *(Cont'd)*

5. The position of an element in the array list has no relevance to the application.
6. Most of the insertions and deletions to the array list take place at the size index.

# What Makes a Class an ADT?

**the Circle record**

```
public class Circle
{
  public int xValue;
  public int yValue;
  public float radius;
  public boolean solid;
}
```

**the Date ADT**

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

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

  public int yearIs()
  {
    return year;
  }

  public int monthIs()
  {
    return month;
  }

  public int dayIs()
  {
    return day;
  }
}
```

# Access Modifiers

| Modifier | Visibility |
|---|---|
| `public` | Within the class, subclasses in the same package, subclasses in other packages, everywhere |
| `protected` | Within the class, subclasses in the same package, subclasses in other packages |
| `package` | Within the class, subclasses in the same package |
| `private` | Within the class |

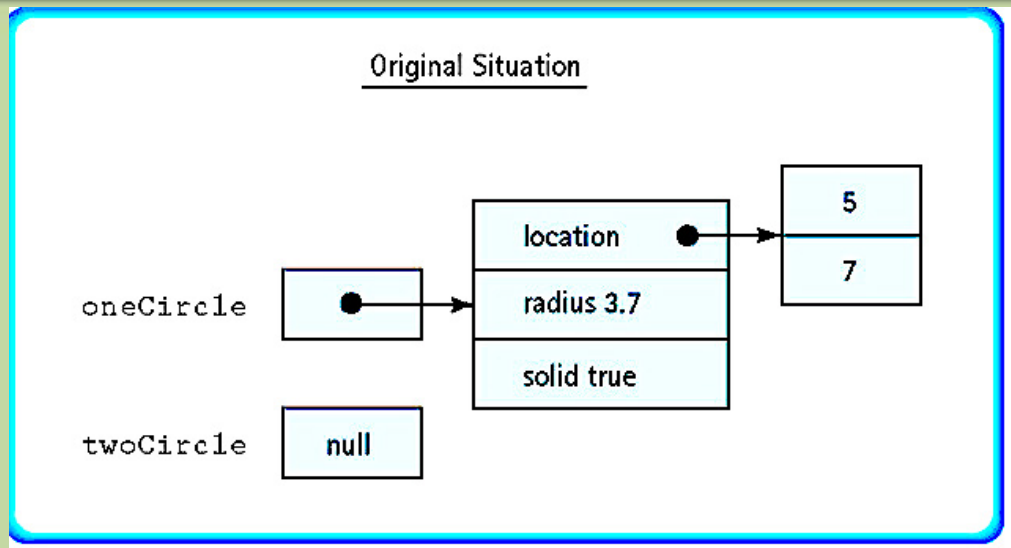
# Shallow Copy

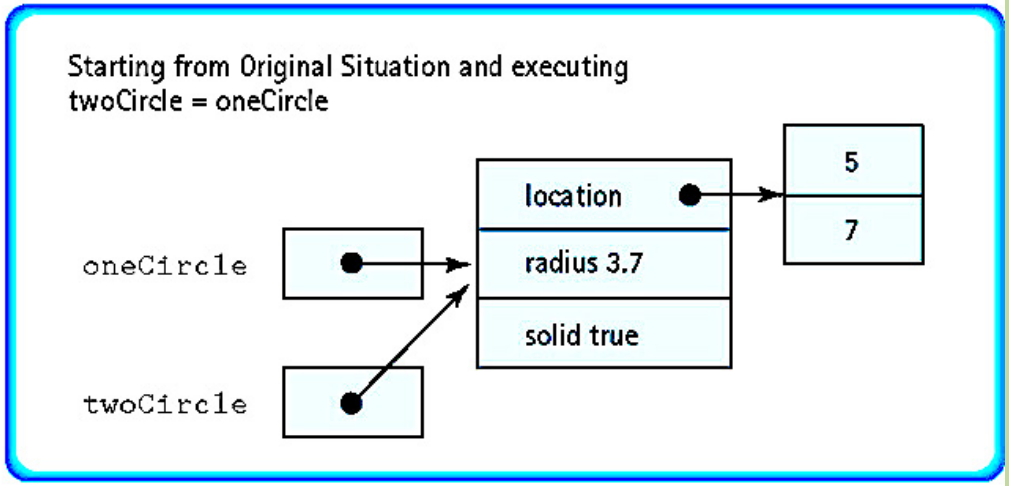
An operation that copies a source class instance to a destination class instance, simply copying all references so that the destination instance contains duplicate references to values that are also referred to by the source.

# Deep Copy

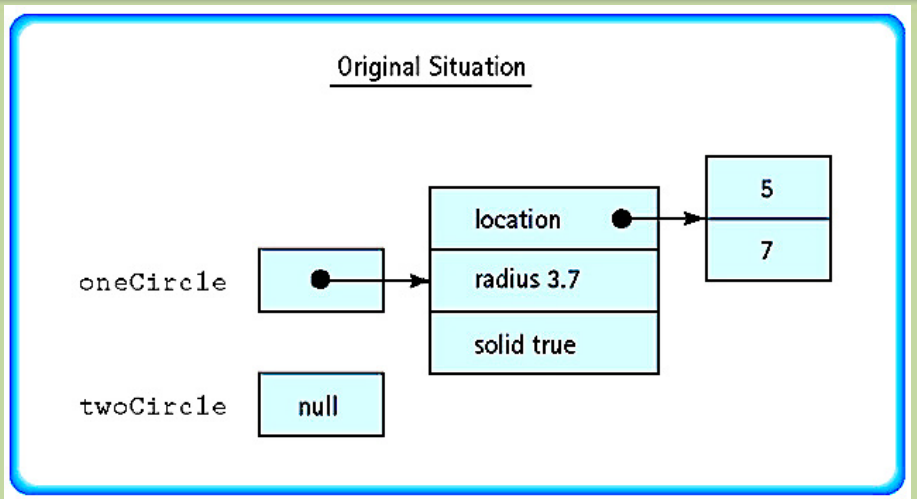
An operation that copies one class instance to another, using observer methods as necessary to eliminate nested references and copy only the primitive types that they refer to. The result is that the two instances do not contain any duplicate references.

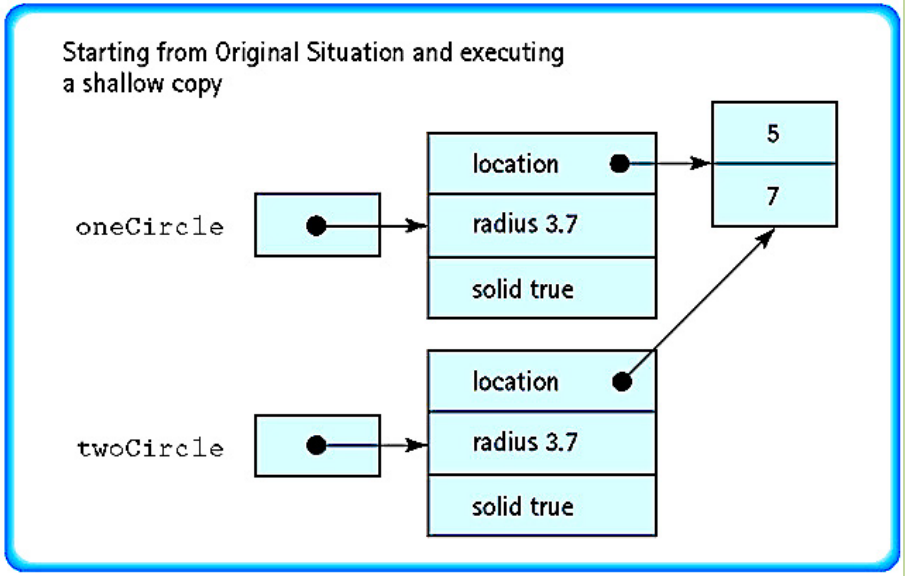
# Copying Objects I



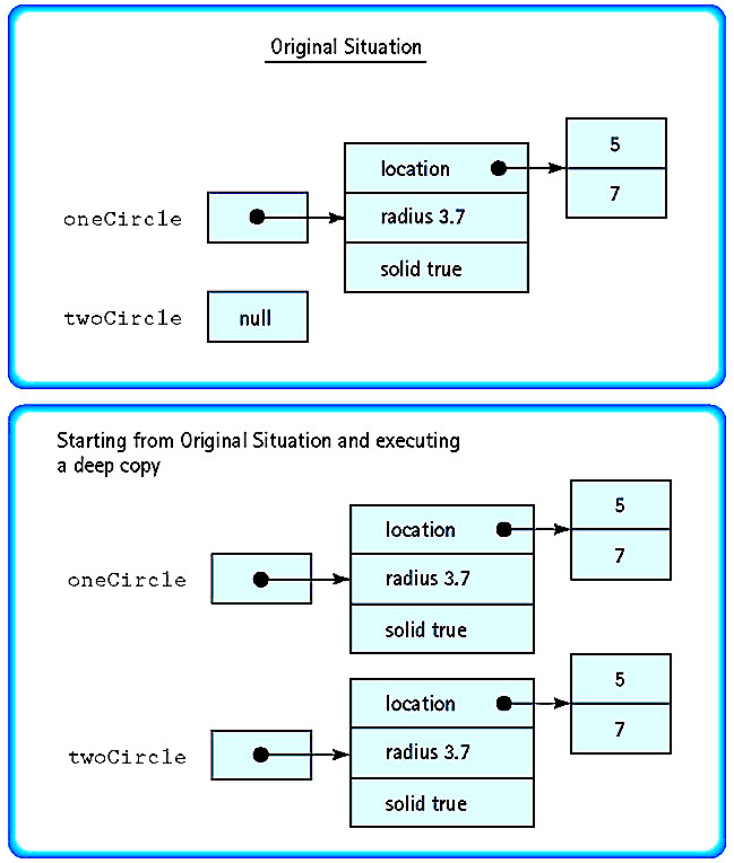



# Copying Objects II



# Copying Objects III



# Defining Our Own Exceptions

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

```
public Date(int newMonth, int newDay, int newYear) throws DateOutOfBound-
sException
{
  if ((newMonth <= 0) || (newMonth > 12))
    throw new DateOutOfBoundsException("month must be in range 1 to 12");
  else
    month = newMonth;

  day = newDay;

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

# Re-Throwing an Exception

```
public class UseDates
{
  public static void main(String[] args) throws DateOutOfBoundsException
  {
    Date 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 Date(M, D, Y);

    // Program continues
  }
}
```

# Catching and Landing an Exception

```
public class UseDates
{
  public static void main(String[] args)
  {
    Date 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 Date(M, D, Y);
      DateOK = true;
    }
    catch(DateOutOfBoundsException OB)
    {
      output.println(OB.getMessage());
    }
  }

  // Program continues
}
```

# Options when dealing with error situations within our ADT method

- Detect and handle the error within the method itself.
- Throw an exception related to the error.
- Ignore the error situation.

# Designing ADTs

1. Determine the general purpose.
2. List the specific types of operations the application program performs.
3. Identify a set of public methods to be provided by the ADT class that allow the application program to perform the desired operations.

# Designing ADTs

4. Identify other public methods which help make the ADT generally usable.
5. Identify potential error situations and classify into
    1. Those that are handled by throwing an exception
    2. Those that are handled by contract
    3. Those that are ignored.

# Designing ADTs

6. Define the needed exception classes.
7. Decide how to structure the data.
8. Decide on a protection level for the identified data.
9. Identify private structures and methods.
10. Implement the ADT.
11. Create a test driver and test your ADT.