CS 113
Introduction to Computer Science I
Spring 2015

Narain Gehani

Welcome

- Course will focus on learning Java with an emphasis on problem solving
- Programming is a key tool of our profession
  - Must know it well
  - Even if one is not going to be a programmer – need to manage programmers / software development
- Programming is a participatory activity and not a spectator activity!

Welcome

Syllabus
(handed out separately)

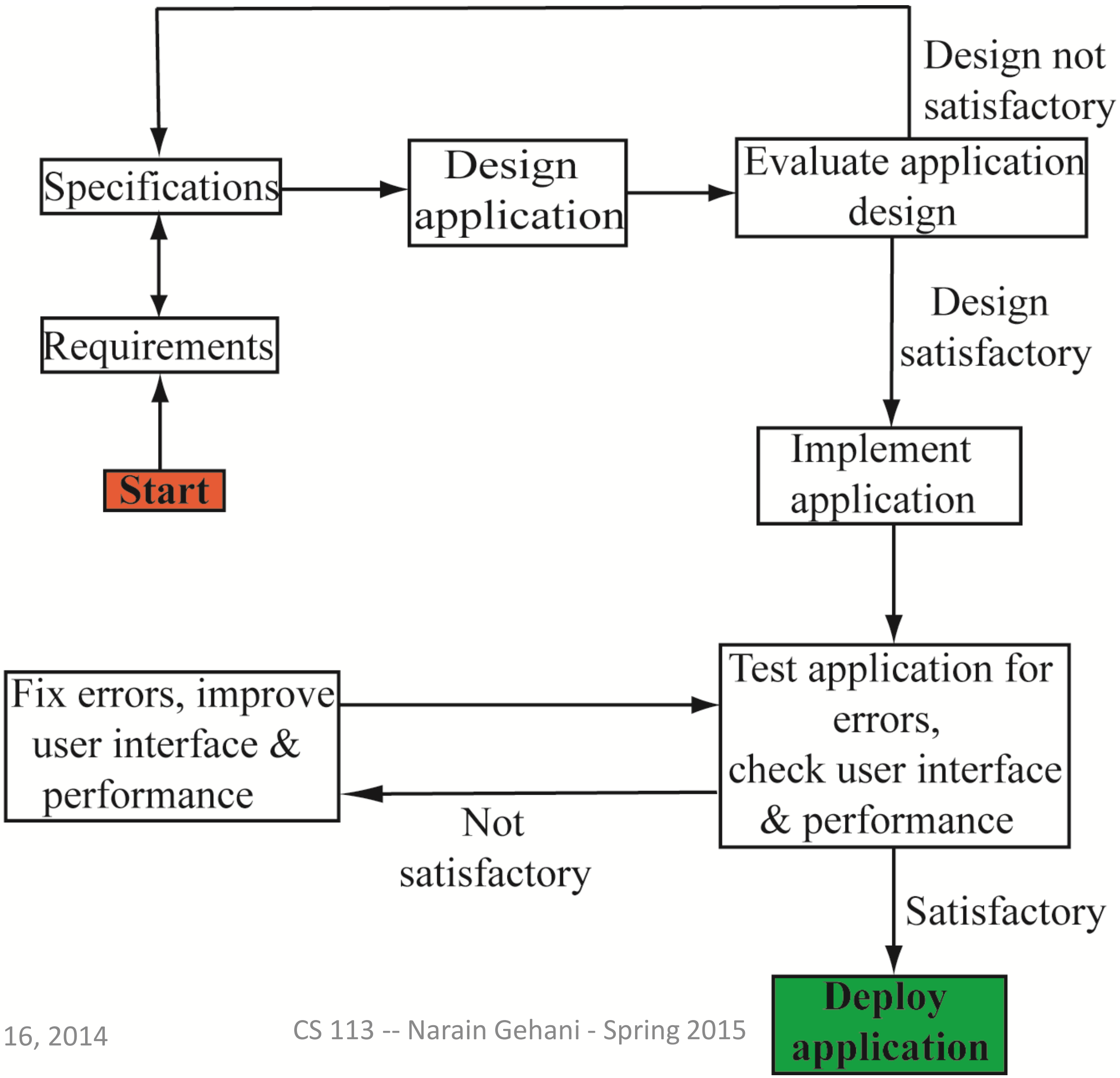
Lecture + Recitations

- Lectures: Concepts
- Recitations: Reinforcing of Concepts + Practice

Software Development

- Requirements (what the customer wants/needs)
- Specifications (what the software should do)
- Design (how the software is organized and how it should work)
- Implementation (converting design to code)
- Testing & Debugging
- Production

Software Design & Implementation



Problem Solving

- Goal of software development → problem solving
  - Use of a particular language is just a means to an end.
  - Typically one should pick the best language – but often the environment dictates what language is to be used.
  - Java + Libraries is a great candidate for software development.

Java Programming Language

- What is Java?
  - Industrial strength programming language
  - Created circa 1995 by Sun (which was acquired by Oracle)
  - General-purpose object-oriented computer programming language

Java Programming Language (contd.)

- Programs can be ported easily to other computers
  - code compiled to run on a virtual machine
  - code can run on any computer that has the virtual machine
  - virtual machines found on all sorts of computers and devices
- Extensive library that is rapidly growing

Java vs. Python

- Python
  - Dynamic typing (variable types based on use) – easy to use for beginners
  - Variable declarations not needed
  - Indentation used to indicate subcomponents (blocks of code)
  - No programmer "overhead" to get the programming machinery going
  - compact code

Java vs. Python (contd.)

- Java
  - Compiled code can be moved to other computers that have the virtual machine
    - Most devices including mobile devices have the Java virtual machine preinstalled
  - Uses braces to indicate blocks of code – makes it easy for Java and programmers to catch errors
  - Indentation has no meaning for the Java compiler – meant for human readability
  - Static typing – makes it easier for compiler to catch errors
  - Verbose (as we shall see)
  - Automatic conversion to string

C++

- C++
  - Compatible with C
  - Designed for systems programming
    - Pointers to memory locations
  - Compiled for specific machine – fast
  - Supports multiple inheritance

Java vs. C++ (contd.)

- Java
  - General application programming language
  - No pointers
  - No gotos – trying to prevent spaghetti code
  - Array bounds checking
  - Strong & better typing
  - Better error handling
  - Portable because code compiled for virtual machines
  - Garbage collection (automatic memory management)

Java Libraries

- Java's strength comes from the large number of libraries
- Language is relatively simple – but made complex by the large number library facilities.
- As reference for details about v7 & v8 libraries, please see

  `docs.oracle.com/javase/7/docs/api/overview-summary.html`

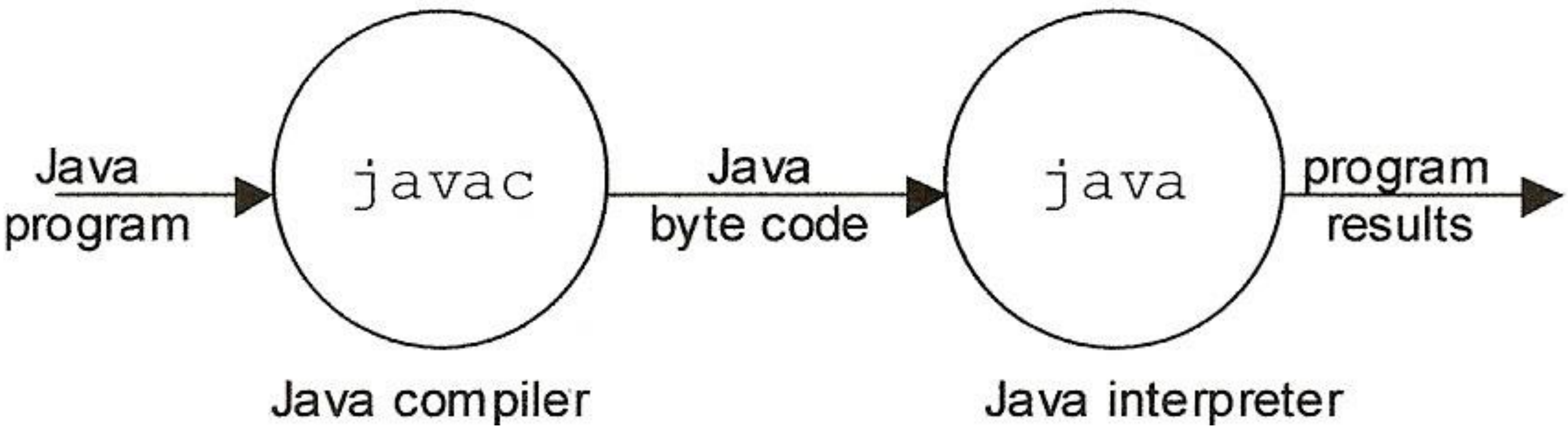
  `docs.oracle.com/javase/8/docs/api/overview-summary.html`

Compiling & Running Java Programs (Practice in Recitation)

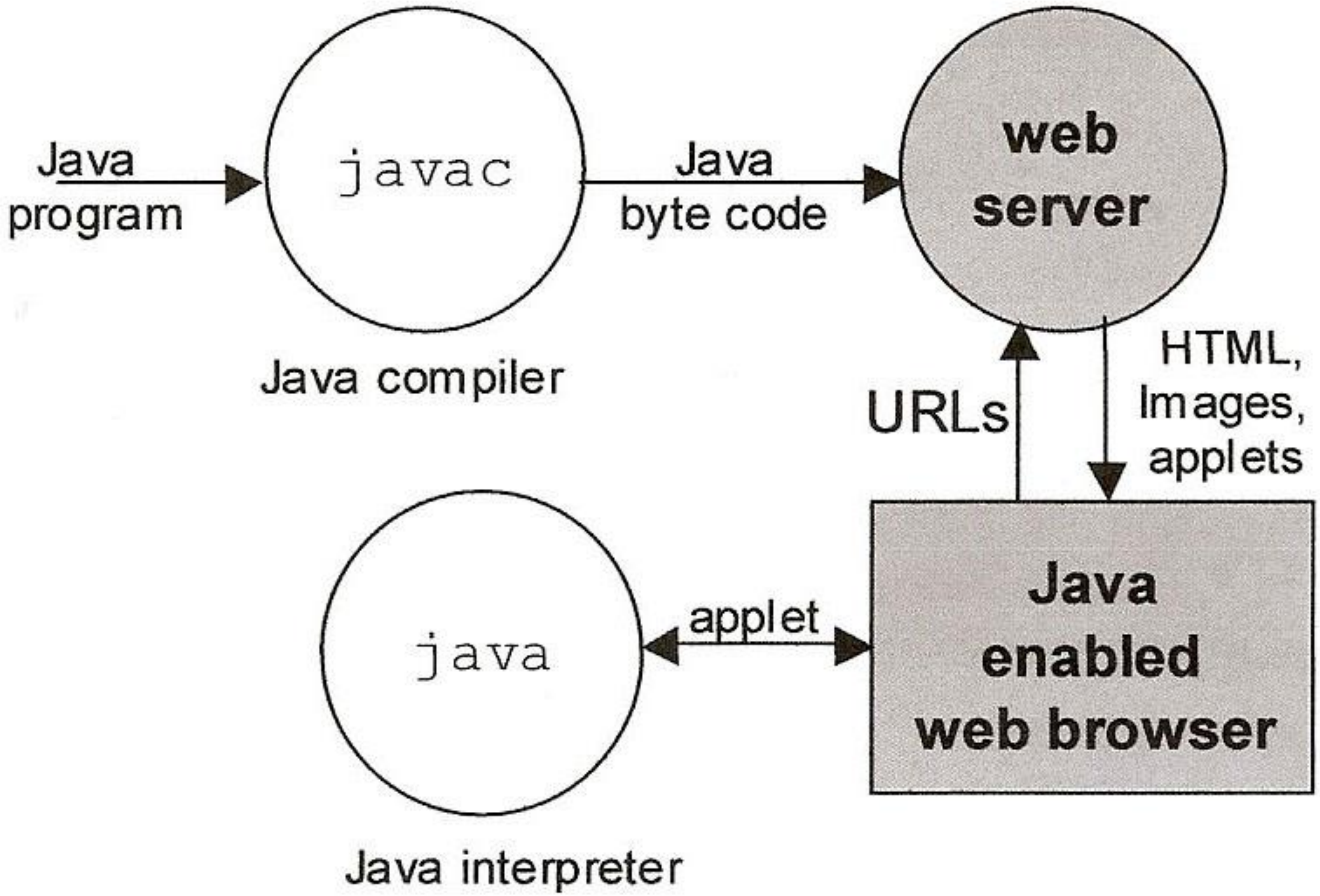
Developing Java Programs

- Three options
  - Command-line interface
  - jGrasp Integrated Development Environment
  - Eclipse Integrated Development Environment
- I will discuss the command-line & jGrasp very briefly.
  - They will be discussed more in the recitations
  - You can use Eclipse – you are on own in this case

Compiling & Running Java Programs



Compiling & Running Java Applets



Installing Java Standard Edition (SE)
(if you want to install Java on your computers)

- Follow Java SE download instructions at the Oracle website

  `www.oracle.com/technetwork/java/javase/downloads`

- To use the Java compiler (`javac`) and the Java intepreter (`java`) directly from the Windows *Command Window*:
  - set the Windows environment variable **`PATH`** to point to the **`bin`** folder in the Java software.

Setting Up the Command-line Interface

- Set up the **PATH** environment variable to enable use of **javac** & **java** from any directory (folder) in the command window:

- Go to Start (Windows 7)
  --> Control Panel
  --> System & Security
  --> Change Settings
  --> Advanced
  --> Environment Variables
  --> edit **PATH** variable

- Find location of the Java **bin** directory that has the **javac** and **java** commands and add it to the **PATH** variable.
- In my case I appended the following to the **PATH** variable

;C:\Program Files\Java\jdk1.7.0_05\bin

jGrasp IDE

- IDE = Integrated Programming Environment
- Integrated functionality for Java Program Development, specifically for
  - creating,
  - editing,
  - compiling,
  - running,
  - debugging,
  - file and folder manipulation,
  - …

Downloading jGRASP
(if you want to install jGRASP on your computers)

- To get jGRASP
  - Go to www.jgrasp.org
  - Download, and install on your PC/Laptop
  - There is also a tutorial you can download

The First Java Program

Printing "Hello World!"

- To print “Hello World!”, we would like to tell Java something like

```
print("Hello World!")
```

Printing "Hello World" (contd.)

- But this is not enough.
- We have to tell Java which print "method" we are talking about and where it is to be found

```
System.out.println("Hello World!");
```

The First Program – HelloWorld.java

- This is not enough – we need to supply some more details
- And the program must be stored in a file that matches the class name

```
// The First Program - hello world
// Author: N Gehani

    class HelloWorld {
        public static void main(String[] args) {
            System.out.println("Hello World!");
        }
    }
```

Our first complete Java program!!!

The First Program - HelloWorld

- Lot of baggage for a simple program!
- Relatively simple core language but libraries make it complex!
- Learn libraries as needed
- File name must be the same as the class name

The First Program – HelloWorld.java (contd.)

- When the program is “run” or “executed”, it will print

```
Hello World!
```

- From what you understand about this program, what can you do to extend it?

Command-line Compiling & Running `HelloWorld.java`

- We need two programs to run a Java program, `javac` and `java`.
  - `javac` (Java Compiler) takes a Java program, (*fileName*`.java`) and produces byte code (which it stores in *fileName*`.class`)
  - `java` (Java interpreter) uses the byte code to run the program

Command-line Compiling & Running `HelloWorld.java` (contd.)

- Assume that the Java program has been written.
- Position the Windows *Command Window* to the folder/directory containing the program.
- Compile program as

  ```
  javac HelloWorld.java
  ```

  which produces the byte code file

  ```
  HelloWorld.class
  ```

- Now you can run the program (its byte code) as

  ```
  java HelloWorld
  ```

Using jGRASP

- Click on the folder icon to create a folder, say `FirstJavaProgram`
- Start creating your first Java program file

```
File -> New -> Java
```

- Lets say we write the program `HelloWorld` in this file

```
// The First Program - hello world
// Author: N Gehani
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Using jGRASP (contd.)

- Now we need to save the file. Save the file (for the first time) giving it a name that matches the name of the class in the file

```
File -> Save As -> HelloWorld
```

Subsequent saves can be

```
File -> Save
```

or by simply double clicking on the diskette icon

- Compile program as

```
Build -> Compile
```

or simply by clicking the icon. This produces the executable program

```
HelloWorld.class
```

- Run program as

```
Build -> Run
```

or simply by clicking on the Runner icon 

Using jGRASP (contd.)

- If the program, say `HelloWorld.java` exists already, then navigate jGRASP to the folder `HelloWorld` containing it
- Compile `HelloWorld.java` to produce

  `HelloWorld.class`

- Run the program

Command-line arguments with jGRASP

- In the Build tab, check Run Arguments
- Compile Java program and you will see a toolbar at the top
- Supply file name in the toolbar
- Run program

Java Convention + Organization

- Java files have the extension `.java`
- Compiled Java (byte code) files have the extension `.class`
- Java file name must match the class name.
- Keep each program – all pieces – in its own folder/directory – otherwise, you have to tell Java run-time system or jGRASP where to find the missing pieces (using `CLASSPATH` & project facilities).

Object-Oriented Design

- Java is an object-oriented programming language.
- In an object-oriented programming model, program entities are modeled as objects that interact with each other.
- Java provides facilities to specify objects and facilities to interact with them.
- For example, when writing a banking application, some objects in the program could be
  - customer
  - account
  - manager

Classes & Objects

- A class is the template of an object.
- An object is an instantiation of a class
- A class specifies
  - Variables to hold data
  - Methods specifying how objects can be queried or changed.
- Each stand-alone Java program must have a `main` method (Java applets which are programs embedded in web browsers do not have a `main` method)
- The `main` method represents the Java program (it may call other methods, including those associated with other objects to perform expected functionality).

Comments / Notes (explanations in a Java Program)

- There are 2 styles of writing comments (notes) in Java programs
  - One-line comments begin with // and end with end of line.
  - Multiple-line comments begin with /* and end with */

Java Program Structure

```
Comments + import statements
public class ClassName
{
```

class header

class body

Class body contains program statements (declarations and actions) which can be intermingled with comments.

```
}
```

Java Program Structure (One `main` method per program)

```
Comments + import statements
public class ClassName
{
    //  comments about the method
    public static void main(String[] args)
    {
                                              method header
                method body

    }

}
```

Command-line Arguments

- When running a Java program, we may want it run with different input data on different occasions
- One way of doing this is to give the program data (or location of the data) when starting the program
- This technique of supplying data is called supplying data using "command-line" arguments.

Command-line Arguments (contd.)

- `String args[]` in a **`main`** method

```
public static void main(String[] args) {
 …
}
```

is an "array of strings" for holding command-line arguments

Command-line Arguments – `HelloCustom.java`

```
// Using Command-Line Arguments

class HelloCustom {
   public static void main(String[] args){
      System.out.println("Hello " +
            args[0] + ", Good Morning!" );
   }
}
```

Command-line Arguments (Contd.)

Having compiled `HelloCustom.java` using the command `javac`, the command

```
java HelloCustom Mike
```

runs the program `HelloCustom` program while passing the argument `Mike` to it producing the output

```
Hello Mike, Good Morning!
```

Note: `java` is the Java interpreter / virtual machine
`Mike` is the command-line argument

Command-line Arguments (Contd.) Using jGrasp

- Build using **Run Arguments** option

  Opens a toolbar to pass arguments
- Enter `Mike` in tool bar
- Then **Run** produces the output

```
Hello Mike, Good Morning!
```

Exception (Error) Handling

- If a command-line argument is not supplied to the program `HelloCustom` which is expecting one, `HelloCustom`
  - goes in a tizzy because it finds no value in `args[0]` ,
  - raises an "exception", and
  - terminates.
- This is not good. We have choices. We can
  - "handle" the exception or
  - explicitly check for the missing argument

  and take corrective action.

HelloCustomException.java

```
// Using Command-Line Arguments

class HelloCustomException {
   public static void main(String[] args) {
      try {
         System.out.println("Hello " + args[0]
                           + ", Good Morning!" );
      }
      catch (ArrayIndexOutOfBoundsException e) {
         System.err.println("Sorry, you must specify\n" +
                        "the person to be greeted\n" +
                        "as the command argument!");
         System.err.println("\nException " + e);
      }
      System.exit(0);
   }
}
```

HelloCustomException.java

Running `HelloCustomException` with no command-line argument as

```
java HelloCustomException
```

generates the following output:

```
Sorry, you must specify
the person to be greeted
as the command argument!

Exception java.lang.ArrayIndexOutOfBoundsException: 0
```

HelloCustomException.java

Of Special Interest

- The exception handling statement

  `try {…} catch(`*exception***`) {…}`**

- The `err` stream and the `exit` function

```
System.err.println("\nException " + e);
System.exit(0);
```

Exception (Error) Handling

```
try {
  statements that may cause an error
} catch (exception) {
    error-handling-statements that fix
    error or stop program
}
```

Output & Input streams

- Output
  - by default, goes to the screen (display)
  - can be "redirected" to a file
- Input
  - by default is taken from the keyboard
  - can be redirected from a file to the program
- Two kinds of output streams
  - Normal output stream
  - Error output stream
  - Each can be redirected to a file (same or separate)

Redirecting output & input
(Command-line program execution)

- Output sent to the `System.out` stream (screen) is redirected to a file using the redirection operator “>” as in

  ```
  java HelloWorld >result
  ```

  but output sent to the `System.err` stream will still show up on the screen

- Output to `System.err` can also be redirected to a file

  ```
  java HelloWorld >result 2>errorfile
  ```

  but is often not done so that you can see the error

- A file can be used as a substitute for input from the keyboard using the "<" redirection operator as in

  `java` *JavaProgram <inputFile*

Terminating the Program

- The method call

  `System.exit(`*statusCode*`)`

  terminates the program.

- In our case, it would have happened anyway by program execution flowing to the end of the program.
- It is good to use `System.exit()` to terminate a program especially when other programs are monitoring the program.

- Status code is accessible to higher-level programs and operating systems (OS) to determine program failure or success.
  – Status code 0 indicates normal termination
  – Non-zero status code indicates failure.

Human Readable Programs

- Making programs human readable is important.
  - Indent programs to show sub levels
  - Use mnemonic names
  - Write comments

- Lets again look at the indented version of

  `HelloCustomException.java`

  first and then a version with no indentation

HelloCustomException.java (good indentation)

```
// Using Command-Line Arguments

class HelloCustomException {
   public static void main(String[] args) {
      try {
         System.out.println("Hello " + args[0]
                              + ", Good Morning!" );
      }
      catch (ArrayIndexOutOfBoundsException e) {
         System.err.println("Sorry, you must specify\n" +
                           "the person to be greeted\n" +
                           "as the command argument!");
         System.err.println("\nException " + e);
      }
      System.exit(0);
   }
}
```

HelloCustomExceptionNoIndent.java (no indentation)

```
// Using Command-Line Arguments
// Author: N Gehani

class HelloCustomExceptionNoIndent {
public static void main(String[] args) {
try {System.out.println("Hello " + args[0] + ",
Good Morning!" );}
catch (ArrayIndexOutOfBoundsException e) {
System.err.println("Sorry, you must specify\n" +
"the person to be greeted\n" +
"as the command argument!");
System.err.println("\n***Exception*** " + e);
} System.exit(0);}}
```

Human Readable

Computer can read

`HelloCustomExceptionNoIndent.java`

and "understand" it, but it is no good for humans!

Human Readable (contd.)

- The two versions are functionally equivalent but very different from a variety of fronts:
  - Readability (programmer and others)
  - Testing
  - Debugging
  - Maintenance

Software Development Using Stepwise Refinement

Customer wants a Calculator

- Customer
  - I would like to you to build me a calculator
- Information Technology Expert
  - What should the calculator be able to do
- Customer
  - Add, subtract, multiply, exponentiation
- Information Technology Expert
  - What about division?
  - Why exponentiation?

Requirements

- Develop a simple calculator which takes
  - two real operands, a and b,
  - and an operator opr (+, -, *, /) and
  - computes the result

a opr b

Stepwise Refinement (contd.) Calculator (Refinement Level 0)

```
while (true) {
    read a, opr, b;
    result = a opr b;
    print result;
}
```

Stepwise Refinement (contd.) Calculator (Refinement Level 1)

```
while (true) {
   read a, opr, b;
   switch(opr) {
   case '+':  result = a + b; break;
   case '-':  result = a - b; break;
   case '*':  result = a * b; break;
   case '/':  result = a / b; break;
   }
   print result;
}
```

Stepwise Refinement (contd.) Calculator -- Questions

- At this stage, we have some questions & thoughts:
  - How/when is the calculation session to be ended
  - What is to be done about bad real (float) inputs?
  - What is to be done about a bad operator?

Time to go back to the customer and/or make make some decisions

Stepwise Refinement (contd.) Calculator – Answers / Decisions

- End session on end of input indication (Cntrll-Z on Windows Systems & Ctrl-Z on Linux / Apple systems):
  - Catch exception `NoSuchElementException` which is raised when there is no more input and input was expected.
- Dealing with bad operand inputs:
  - Catch `InputMismatchException` which is raised when expecting a floating-point number and the input is not a floating number
- Bad operator will have to be handled with an explicit check because there is no notion of a bad character when reading a single character

Calculator Specificaton (Refinement Level 2)

```
while (true) {
    try {
        read a, opr, b;
        if (opr is not one of +, -, *, /)
          start again;
        switch(opr) {
        case '+':  result = a + b; break;
        case '-':  result = a - b; break;
        case '*':  result = a * b; break;
        case '/':  result = a / b; break;
        }
        print result;
    } catch (InputMismatchException) {shut down}
      catch (NoSuchElementException) {shut down}
}
```

Stepwise Refinement (contd.) Calculator

Ready to implement program in Java

Stepwise Refinement (contd.) Calculator in Java

```
// A simple calculator

// a OPR b
// OPR = + | - | * | /

import java.util.*;
        // for class Scanner and for exception
        // for NoSuchElementException which is
        // raised when there are no more tokens and for
        // InputMismatchException which is raised when
        // the input number does not match a
        // floating number as expected
```

Calculator in Java (contd.)

```
public class Calculator {
   public static void  main(String[] args) {
      Scanner stdin = new Scanner(System.in);
      System.out.println("Ready to calculate!\n“
                 + "Enter Control-Z when done!");

       double a = 0, b = 0, result = 0;
       char opr = ' ';
```

Stepwise Refinement (contd.) Calculator in Java (contd.)

```
while(true) {
  try {
    System.out.print("Enter Operand 1: ");
    a = stdin.nextDouble();
    System.out.print("Enter Operator: ");
    opr = stdin.next().charAt(0);
    if ((opr != '+') && (opr != '-')
         && (opr != '*')  && (opr != '/')) {
       System.out.println("Bad Operator = " + opr +
            "\nMust be one of + - * /");
                continue;
    }
    System.out.print("Enter Operand 2: ");
    b = stdin.nextDouble();
```

Stepwise Refinement (contd.)
Calculator in Java (contd.)

```
switch(opr) {
  case '+':   result = a + b; break;
  case '-':   result = a - b; break;
  case '*':   result = a * b; break;
  case '/':   if (b == 0) {
                 System.out.println(
                         "Cannot Divide by Zero.");
                 continue;
               }
                result = a / b; break;
}
System.out.printf("Result = %.2f\n", result);
```

Stepwise Refinement (contd.) Calculator in Java (contd.)

```
        } catch(InputMismatchException e) {
              System.out.println("Input Does not ” +
                     “match a floating point number");
              System.out.println("Shutting down");
              System.exit(0);
        } catch(NoSuchElementException e) {
              System.out.println("Shutting down");
              System.exit(0);
        }
     }
   }
}
```

Using the Calculator

```
Ready to calculate!
Enter Control-Z when done!
Enter Operand 1: 33
Enter Operator: 11
Bad Operator = 1
Must be one of + - * /
Enter Operand 1: 33
Enter Operator: /
Enter Operand 2: 11
Result = 3
Enter Operand 1: <eof>
Shutting down
```

Nuts & Bolts of Java Programs

Java Programs

- Java programs consist of a set of instructions called *statements*
- Statements are composed of
  – Identifiers
  – Literals
  – Types
  – Variables & Constants
  – Expressions
  – Operators

Java Programs (contd.)

- Statements can be
  - *declaration* statements that define properties, objects, etc.
  - *executable* statements that perform actions such as assign values, perform conditional actions

Java Identifiers

- *identifiers* are the "words" in a program
  - Made from letters, digits, the underscore character (_), and the dollar sign
  - Cannot begin with a digit
- Java is *case sensitive*: `Total`, `total`, and `TOTAL` are different identifiers
- By convention, programmers use different case styles for different types of identifiers – constants, variables, class names, method names, etc.

Java Reserved Words / Identifiers (can only be used as predefined by Java)

abstract	**else**	**interface**	**switch**
assert	**enum**	**long**	**synchronized**
boolean	**extends**	**native**	**this**
break	**false**	**new**	**throw**
byte	**final**	**null**	**throws**
case	**finally**	**package**	**transient**
catch	**float**	**private**	**true**
char	**for**	**protected**	**try**
class	**goto**	**public**	**void**
const	**if**	**return**	**volatile**
continue	**implements**	**short**	**while**
default	**import**	**static**	
do	**instanceof**	**strictfp**	
double	**int**	**super**	

White Space

- Spaces, blank lines, and tabs are called white space.
- White space is used to separate words and symbols in a program.
- Extra white space is ignored.
- White space is used to enhance program readability, e.g., by using consistent indentation

Syntax and Semantics

- The ***syntax rules*** (grammar) of a programming language specify how a program is constructed from
  - symbols,
  - reserved words, and other
  - identifiers.
- The ***semantics*** of a program specify the meaning of the program – what the program does.
- A syntactically correct program may not be logically (semantically) correct.

Types of Errors

- Compile-Time Errors
  - Syntax errors, basic problems (such as uninitialized variables)
  - Compiler flags them before the program is run
- Run-Time Errors
  - Errors found during program execution, such as trying to divide by zero, accessing uninitialized array elements, infinite loops, …
- Logical Errors
  - A program runs, but produce incorrect results, infinite loops, …

Literals
(notations for specifying fixed values)

- characters
- strings
- integers
- floating-point
- boolean

Character Literals

- Single character enclosed in single quotes

```
'a'  'b'  '4' '\n'
```

- An escape sequence (indicated by backslash `\`) is used for specifying characters that have a special meaning

Escape Sequences for Specifying Special Characters

Escape Sequence	Character
`\b`	backspace
`\t`	horizontal tab
`\n`	line feed
`\f`	form feed
`\r`	carriage return
`\"`	double quote
`\'`	single quote
`\\`	backslash

String Literals

- String literals are character sequences enclosed in double quotes.

`""` (null string)

`"Hello World!“`

`"Hello World!\n"`

`"\""` (string consisting of the quote character)

Integer Literals

`22` (decimal)

`0x1f` (hexadecimal)

`026` (octal)

`0L` (long)

`22l` (long)

`0X1F8l`(long hexadecimal)

Real / Floating-Point Literals

float = single precision

double = double precision

`2.19f` (float)

`1.0e+23F` (float)

`3.1416` (double)

`777E+23` (double)

boolean (logical) literals

- `true`
- `false`

Types (of values)

- Primitive
  - Predefined types – building blocks for defining structured (complex) types, e.g., array & class (object) types
- Enumerated – A type whose values are user-defined identifiers
- Array
  - A one-dimensional array is a sequence of elements that are accessed by specifying their position. An n-dimensional array type is a sequence of (n-1)-dimensional arrays.
- Class
  - A class (object) type is a user-defined type (can also be a type predefined in a Java library).
- Interface
  - Similar to class types except that they consist only of method specifications (no body). Interfaces are implemented using classes.

Primitive Types

- `char`
- `int` (also `byte`, `short`, `long`)
- `float`, `double`
- `boolean`
- `String` (technically not a primitive type; it's an object type but given its strong supporty in Java `String` can be thought of as a primitive type.)

Enumerated Type Example

```
enum Student{freshman, sophomore,
             junior, senior}
```

- `Student` is a user-defined enumerated type
- It has values

  `Student.freshman`,

  `Student.sophomore`, etc.

- Better than using 1, 2, etc. to represent `freshman`, `sophomore`, etc.

Variables & Variable Declarations
(storing values)

- A *variable* is a name for a container (memory location) that holds a value.
- Variables must be declared for them to come into existence
- Variable declarations have the form

type identifier [= *initial-value*] ***;***

where

  - *identifier* is the variable name
  - *type* specifies the kind of value that will be stored in the memory location associated with *identifier*
  - square brackets [ ] are used to specify an optional initial value
  - here = is the initialization operator; it does double duty as the assignment operator
- Variable values can be changed with the assignment operator as we shall see soon.

Variable Declaration Examples

```
char terminator = ';';
double balance;
int n;
int i = 0;
String name;
String greeting = "Good Morning!";
boolean finished = false;
```

Mulitple Variables in One Declaration

- Another form of the variable declaration:

  type { *identifier* [= *initial-value*] }$^+$ *;*

  where {*x*}$^+$ specifies 1 or more occurrences of *x*.

- Example:

```
int i = 0, j = 0, k;
```

Default Initial Values

- Java does initializes variables by default but in general it is better to use explicit initialization
  - good for readability

Variable Type	Default Initial Value
`int`	0
`double`	0.0
`String`	`null`
`boolean`	`false`

Constants (Final Values)

- Symbolic name for a literal value
- Format of a constant declaration

  `final` ***type identifier*** = ***value***;

- Examples:

```
final int MAX = 1024;
final String NJIT =
   "New Jersey Institute of Technology";
```

- `final` → value cannot be changed

Scope of Variables & Constants

- A variable's or constant's scope (the area in which it can be referenced) depends on where it is declared.
- If declared in a
  - **class** (outside a method), then it can be referenced any where in the class (or in any of its methods)
  - **method**, then it can be referenced in the method
    - if there is a name collision with a variable/constant at the class level, then it will hide the class-level variable/constant
  - **block** (delimited by curly braces) inside a method, then it can be referenced in the block
    - if there is a *name collision* with a variable/constant at the class or method level, then the block-level variable or constant will hide its class-level or method-level counterpart variable or constant.

Expressions

- As mentioned, expressions are constructed using variables, constants, literals, method calls, and operators
- In describing expression composition I will use the notation

  → means ***is composed of***

  a | b means ***either a or b***

Expression Rules / Grammar

- Expressions have the form

  expression → expression operator expression

  expression → (expression)

  expression → unaryOperator expression | expression unaryOperator

  expression → literal | simpleIdentifier | compositeIdentifier

- Examples of expressions

  `a+b`, `x+1`, `-x`, `(-b+Math.sqrt(b))/(2*a)`

Statement & Expressions

- A Java statement is a "standalone" executable unit within a program.
- Java provides many statements to specify programmer actions.
- In addition, any expression can be made into a statement by terminating it with a semicolon

Operators

Operators

- Operators are used for constructing expressions.
- Operators take one, two, or three values to produce a result value.
- They are classified as
  - Unary such as +, - (take one operand)
  - Binary such as *, /, +, - (take two operands)
  - Tertiary such as the conditional operator ?: (takes there operands)

Assignment Operator

- The assignment operator $=$ is used to assign values to variables (overriding previous values, if any)
- For example, the assignment expression

    variable $=$ *expression*

  assigns the value of the *expression* to *variable*
- A Java expression is composed of variables, constants, literals, *method calls*, and operators.
- *variable* must have been declared *a priori*.
- There are other assignment operators, but $=$ is the one used most often

Assignment Expression → Assignment Statement

- An assignment expression (actually, any expression) can be made into a statement by terminating it with a semicolon

  variable = *expression* ;

- This form of assignment statement is the one you will see most often.

Assignment Examples

```
switch(opr) {
   case '+':   result = a + b; break;
   case '-':   result = a - b; break;
   case '*':   result = a * b; break;
   case '/':   if (b == 0) {
                 System.out.println(
                  "Cannot Divide by Zero.");
                 continue;
                }
               result = a / b; break;
}
```

Assignment Examples (contd.)

```
scan = new Scanner(System.in);

number_of_disks = scan.nextInt();

i = j = k = n;
```

Assignment Expression vs. Assignment Statement

```
boolean B = false;
if (B = true)
    System.out.println("Assignment"
    + " expression sets B to true");
```

This code will print

```
Assignment expression sets B to true
```

Assignment Expression vs. Assignment Statement (contd.)

```
boolean B = false;
if (B == true)
   System.out.println("Comparison" +
              " finds B to be false");
```

- The above code will print nothing.
- The above `if` statement is equivalent to

```
if (B)
   System.out.println("Comparison" +
              " finds B to be false");
```

String Concatenation Operator +

- Takes two strings as operands

  $String_1 + String_2$

  and joins (concatenates) them to produce another string
- If one operand of the concatenation operator + is not a string, e.g., it is a number, then Java tries to convert it (if possible) to a string and then joins them

  String + number

  number + String

- Java provides facilities to automatically convert numbers to strings
- If Java cannot convert the non-string operand to a string then it flags an error
- For object (user-defined) types, users need to specify how to convert the object to a string

String Concatenation Examples

```
"Factorial of " + 3
   → "Factorial of 3"

"First" + "Last"
   → "FirstLast"

"First " + "Last"
   → "First Last"

"Line1\n" + "Line2\n" + "Line3"
   → "Line1\nLine2\nLine3"
```

String Concatenation Examples (contd.)

- Assuming `n` has the value 3, then

```
"Factorial of " + n + " = "
```

→

```
"Factorial of 3 = "
```

- Do we have to use the concatenation operator here?
- Why can we not write

```
"Factorial of + n + = "
```

String Concatenation Examples (contd.)

```
System.out.println("Ready to calculate!\n" +
                   "Enter Control-Z when done!");
```

is equivalent to

```
System.out.println("Ready to
calculate!\nEnter Control-Z when done!");
```

- Do we have to use the concatenation operator here?
- Why?

Arithmetic Operators

Operator	What does it do?
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Remainder

Relational Operators

- Relational operators compare two values and return `true` or `false`:

`==`	equal to
`!=`	not equal to
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

Logical Operators

- *Logical operators* operate on boolean operands and produce a boolean value as the result

`!`	Logical NOT
`&&`	Logical AND
`||`	Logical OR

- `!` is a unary operator (operates on one operand)
- `&&` and `||` are binary operators (operate on two operands)

Semantics of the NOT operator !

a	**!a**
`true`	`false`
`false`	`true`

Semantics of AND and OR operators && and ||

a	**b**	**a && b**	**a || b**
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

- Operators && and || are short-circuit operators – the second operand is evaluated only if necessary.
- Need to be careful as this can cause problems in some cases.

Increment Operators (Prefix & Postfix)

- Prefix

  `++a` Increment `a` and then use value of `a`

  `--a` Decrement `a` and then use value of `a`

- Postfix

  `a++` Use value of `a` and then increment it

  `a--` Use value of a and then decrement it

Assignment Operators

- Simple assignment operator = (discussed already)
- Besides the simple assignment operator =, Java has many other assignment operators which do double duty by performing an
  - operation followed by
  - assignment
- For example

```
total += amount
```

does addition followed by assignment

```
total = total + amount
```

Assignment Operators (contd.)

- An assignment operator operates as follows:
  - First the entire right-hand expression is evaluated
  - Then the result is combined with the variable on the left side
  - The combined value is assigned to the variable

- Therefore

```
result /= (total-MIN) % num;
```

is equivalent to

```
result = result / ((total-MIN) % num);
```

Some Assignment Operators

Operator	Example	Equivalent To
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`%=`	`x %= y`	`x = x % y`

Evaluating Expressions

- Operator precedence & associativity determine the order in which operators in an expression are evaluated (applied)
- Each operator is assigned a precedence & associativity
  - Higher precedence operators are evaluated first
  - Operators with the same precedence are evaluated in left-to -right or right-to-left order depending upon their associativity (i.e., whether they associate left to right or right to left)

Operator Precedence & Associativity

<table>
<tr><th>Operator Type</th><th>Operators</th><th>Comments</th></tr>
<tr><td>primary</td><td><code>. [] ()</code></td><td>field, array, & method</td></tr>
<tr><td>postfix unary</td><td><code>++ --</code></td><td></td></tr>
<tr><td>prefix unary</td><td><code>+ - ++ -- ~ !</code></td><td>right-to-left association</td></tr>
<tr><td>cast / allocate</td><td><code>(</code>cast-type<code>) new</code></td><td>right-to-left association</td></tr>
<tr><td>multiplicative</td><td><code>* / %</code></td><td></td></tr>
<tr><td>additive</td><td><code>+ -</code></td><td></td></tr>
<tr><td>bit shift</td><td><code><< >> >>></code></td><td></td></tr>
<tr><td>relational</td><td><code>< <= >= > instanceof</code></td><td></td></tr>
<tr><td>equality</td><td><code>= !=</code></td><td></td></tr>
<tr><td>bit and</td><td><code>&</code></td><td></td></tr>
<tr><td>bit exclusive xor</td><td><code>^</code></td><td></td></tr>
<tr><td>bit inclusive or</td><td><code>|</code></td><td></td></tr>
<tr><td>logical and</td><td><code>&&</code></td><td></td></tr>
<tr><td>logical or</td><td><code>||</code></td><td></td></tr>
<tr><td>conditional</td><td><code>?:</code></td><td>right-to-left association</td></tr>
<tr><td>assignment</td><td><code>= *= /= %= += -= <<=</code>
<code>>>= >>>= &= ^= |=</code></td><td>right-to-left association</td></tr>
</table>

Operator Associativity

- Unless specified explicitly in the previous table, operators associate left-to-right

More Stepwise Refinement

Stepwise Refinement – A Must for Large Program Development (Review)

- Start by writing in psuedo code a, description of the program to be written – Level 0
- Level 0 → Level1
  - add more details (refine the Level 0 description)
- Level 1 → Level2 …
- Stop when writing it in a programming language becomes obvious

Will continue to illustrate stepwise refinement with examples

Write a Program that computes Average, Min, & Max of a list of numbers

Develop program using pseudo code

and

stepwise refinement

Program that computes Average, Min, & Max of a list of numbers (Level 0)

Initialize variables to:

- count numbers in list
- track the sum of the numbers read for computing Average
- track minimum and maximum

Read numbers in the list updating count, sum, minimum, maximum

Print minimum, maximum, and average

Program that computes Average, Min, & Max of a list of numbers (Level 1)

```
if (there is input) read first number x
else exit
sum = x; n = 1; min = max = x;
while (there is input) {
    read next number x
    update min and max
    n++
    sum = sum + x;
}
print min, max, average
```

Program that computes Average, Min, & Max of a list of numbers (Implementation)

```
import java.util.*;  // for class Scanner
public class AverageMinMax {
   public static void  main(String[] args) {
      Scanner stdin = new Scanner(System.in);
      System.out.println(
            "Ready to compute.\n" +
            "Enter Control-Z when done!");
      int n = 0;
       // Java requires initialization here
      double sum, min, max, x;
      sum = min = max = x = 0;
       // Java requires initialization here
```

Average, Min, & Max (contd.)

```
if (stdin.hasNextDouble()) {
   min = max = sum = stdin.nextDouble();
        //why reinitialize min & max when
        //they are already set to 0?
   n = 1;
}
else {
   System.out.println("No Input!\nBye.");
   System.exit(0);
}
while (stdin.hasNextDouble()) {
   x = stdin.nextDouble();
   if (x < min) min = x;
   if (x > max) max = x;
   sum = sum + x;
   n++;
```

Average, Min, & Max (contd.)

```
   System.out.println("Number of Values = " + n);
      System.out.println("Minimum = " + min);
      System.out.println("Maximum = " + max);
      System.out.println("Average = " + sum/n);
   }
}
```

Average, Min, & Max – Sample Interaction

```
Ready to compute.
Enter Control-Z when done!
-2 3 2 -3 4 -16 12
<eof>
Number of Values = 7
Minimum = -16.0
Maximum = 12.0
Average = 0.0
```

Roots of a Quadratic Equation Example

- Illustrates
  - Stepwise refinement
  - Reading data using class `Scanner`
  - 2 ways of writing to the standard output (display) (will recommend the C-style of output facility)
  - Use of arithmetic operators
  - Use of a `Math` function

Roots of a Quadratic Equation

Write a program that computes the roots of the quadratic equation

$$ax^2 + bx + c = 0$$

It should

- takes as inputs a, b, and c
- exits if a is equal to 0 or if b^2-4ac < 0, otherwise
- prints the two roots of the quadratic equation.

Roots of a Quadratic Equation (contd.)

The two roots of the quadratic equation

$$ax^2 + bx + c = 0$$

where

- a, b, c are constants with $a \neq 0$ and $b^2-4ac \geq 0$,
- x is a "math" variable

are given by

$$x = (-b + \text{sqrt}(b^2-4ac))/2a$$

and

$$x = (-b - \text{sqrt}(b^2-4ac))/2a$$

Roots of a Quadratic Equation (contd.)

- Method to compute the square root is

  `Math.sqrt(double` *number*`)`

  – where `Math` is the name of a class that contains the `sqrt` function

Roots of a Quadratic Equation (Refinement Level 0)

Read the coefficients a, b, c

Check coefficients for restrictions

Compute root x1

Compute root x2

Print x1, x2

Roots of a Quadratic Equation (Refinement Level 1)

```
Scanner stdin = new Scanner(System.in)
a = stdin.nextDouble()
b = stdin.nextDouble()
c = stdin.nextDouble()
If (a == 0) then print error and exit
if ((b*b - 4*a*c) < 0) then print error and exit
x1 = (-b + Math.sqrt(b*b - 4*a*c))/(2*a);
x2 = (-b - Math.sqrt(b*b - 4*a*c))/(2*a);
Print x1, x2
```

Roots of Quadratic Equation
The Java Program

```
// Compute the 2 roots of the quadratic equation
// ax2 + bx + c = 0 where a, b, c are constants with
// a != 0, b*b-4ac >= 0,
// and x is a variable

// very little error checking is done in the program
// 3 ways of printing output

// Author: N. Gehani

import java.util.*;  // for class Scanner
```

Roots of Quadratic Equation (contd).

```
public class QuadRoots{
   public static void  main(String[] args) {
      Scanner stdin = new Scanner(System.in);
      System.out.println("Lets compute Roots!\n");
      System.out.println("Enter constants, a, b, & c");

      double a = stdin.nextDouble();
      double b = stdin.nextDouble();
      double c = stdin.nextDouble();

      if (a == 0) {
         System.out.println("a cannot be 0!\nExiting");
         System.exit(0);
      }
      if ((b*b - 4*a*c) < 0) {
         System.out.println("b*b - 4*a*c cannot be < 0!\nExiting");
         System.exit(0);
      }
      double x1 = (-b + Math.sqrt(b*b - 4*a*c))/(2*a);
      double x2 = (-b - Math.sqrt(b*b - 4*a*c))/(2*a);
```

Roots of Quadratic Equation (contd).

```
// printing using automatic conversion
// of numbers to strings
System.out.println("Roots of quadratic equation\n" +
         a + "x*x + " + b + "x + " + c + " = 0\n" +
         "x = " + x1 + " and x = " + x2);
// printing using C style print facility
System.out.printf("Roots of quadratic equation\n" +
      "%.2fx*x + %.2fx + %.2f = 0\n" +
      "x = %.2f and x = %.2f\n", a, b, c, x1, x2);
  }
}
```

Roots of a Quadratic Equation (contd.)

- `Math.sqrt()`
- Pros and cons of the 2 print facilities ?
  - Lets look at the output

Roots of a Quadratic Equation (contd.)

```
Lets compute Roots!

Enter the 3 constants, a, b, and c
1 7 2
Roots of quadratic equation
1.0x*x + 7.0x + 2.0 = 0
x = -0.2984378812835757 3 and x = -6.701562118716424
Roots of the quadratic equation
1.00x*x + 7.00x + 2.00 = 0
x = -0.30 and x = -6.70
```

`Scanner` Class

- `import java.util.*`
- Can be used to read from input/output streams, files, and strings
- Method `hasNext()` used to check if there is any more input.
- Control-Z (^Z) indicates end of input on an interactive basis on Windows System (^D on Linux systems)

```
Scanner (InputStream source)
Scanner (File source)
Scanner (String source)
```
Constructors: sets up the new scanner to scan values from the specified source.

```
String next()
```
Returns the next input token as a character string.

```
String nextLine()
```
Returns all input remaining on the current line as a character string.

```
boolean nextBoolean()
byte nextByte()
double nextDouble()
float nextFloat()
int nextInt()
long nextLong()
short nextShort()
```
Returns the next input token as the indicated type. Throws `InputMismatchException` if the next token is inconsistent with the type.

```
boolean hasNext()
```
Returns true if the scanner has another token in its input.

```
Scanner useDelimiter (String pattern)
Scanner useDelimiter (Pattern pattern)
```
Sets the scanner's delimiting pattern.

```
Pattern delimiter()
```
Returns the pattern the scanner is currently using to match delimiters.

```
String findInLine (String pattern)
String findInLine (Pattern pattern)
```
Attempts to find the next occurrence of the specified pattern, ignoring delimiters.

FIGURE 2.7 Some methods of the Scanner class

Executable Statements

Statements

- Statements are the smallest standalone elements in a program written in a language such as Java.
- Two kinds
  - Declaration Statements (specify properties)
  - Executable Statements (specify actions)
- **Note**: The outermost statement of a Java program is the class declaration.

Statements (contd.)

- Declarations specify variable properties such as type, visibility, & possibly their initial values.
- Executable statements specify actions such as check for a condition and then perform an assignment.

Declarations can be intermingled with executable statements

Blocks

- Blocks are used to group multiple statements into one logical statement.
- Blocks have the form

```
{
  sequence of statements
}
```

- Blocks can be used wherever a single statement is allowed.
- Blocks are routinely used.

Null Statement

- The empty or null statement consists of just the semicolon:

```
;
```

- The empty statement does nothing; it is used occasionally.
- An example of its use is in a `for` loop
  - Loop condition "does all the work"
  - There is no need for the loop body to "do any work".
- Java requires the loop body to be a statement even if the body does no work
  - the empty statement serves the purpose.

Expression Statements

- Expressions can be converted to a statement by simply appending a semicolon
- Expressions → statements
  - assignment (simple & compound assignment)
  - increment & decrement
  - object creation
  - method calls

Simple Assignment Statement
(seen earlier)

- Simple assignment statements

  variable = expression ;

  is an assignment expression converted to a statement by adding a semicolon.

- The use of simple assignments is so common that programmers tend to think of assignment as a statement as opposed to being an operator.
- We will do the same.

Compound Assignment Statement

Compound assignment statements have the form

variable ***operator*****=** *expression* ;

where ***operator*****=** denotes a compound assignment operator:

`*=  /=  %=  +=  -=` etc.

Two examples of compound assignment statements:

```
x += i; y *= y;
```

These assignments are equivalent to the simple assignments

```
x = x + i; y = y * y;
```

Calling Methods

- Method is an object (class) operation
  - invoked or called to interact with an object (class) to perform some service
- Method invocations / calls have the form

  variable-or-className `.` *method* `(` `[` *arguments* `]` `)`

  where *variable* specifies an object.

- Examples seen before (`Math` and `Scanner` are class names, and `stdin` is a variable):

```
double x1 = (-b + Math.sqrt(b*b - 4*a*c))/(2*a);
Scanner stdin = new Scanner(System.in);
                           // constructor called
…
double a = stdin.nextDouble();
```

Method Call Statements

- Method call statements are simply method calls followed by a semicolon:

  variable-or-className **.** *method* ([*arguments*]) **;**

- From within the body of a method, another method associated with the same object can be called without specifying the object or class explicitly:

  method ([*arguments*]) **;**

- The value returned by a method in a method call statement, if any, is discarded.

Conditional Execution

- The `if` and `switch` statements are used for conditionally executing statements based on the value of a expression.
  - The `if` statement provides 2-way branching
  - The `switch` statement provides multi-way branching.
- We have seen examples of both these statements already – but we will now discuss them in detail.

`if` Statement – 2 forms

- The first form:

  `if` (*expression*)
  statement$_{true}$

- If *expression* evaluates to true, *statement*$_{true}$ is executed; otherwise, the statement following the `if` is executed.

- As an example, consider the following `if` statement :

```
if (ch == '\n') {
     n++;
     return opr;
}
```

If `ch` is equal to the newline character `\n` then,

– `n` is incremented, and
– the method containing the `if` statement terminates by returning the value of `opr`.

`if` Statement (contd.)

- The second form of the `if` statement specifies statements to be executed when the specified *expression* evaluates to `true` and to `false`:

    `if (`*expression*`)`

    $\textit{statement}_{\textit{true}}$

    `else`

    $\textit{statement}_{\textit{false}}$

`if` Statement (contd.)

Here is an example code fragment:

```
// print result of looking for integer x in database
    if (found)
        System.out.printf("\t>>%d is in database\n", x);
    else
        System.out.printf("\t>>%d is not in database\n", x);
```

`if` Statement (contd.)

- `if` statements can be nested
- `if` statements are all about comparing data

We discuss data comparison next

Data Comparison

- Interesting issues arise in the comparison of
  - floating-point values for equality
  - characters
  - strings (alphabetical order)
  - object vs. comparing object references

Comparing Float Values

- Avoid comparing two `float` or `double` values for equality
  - you may expect the values to the same but they maybe slightly different depending how they were computed
- Two floating-point values are equal only if their underlying binary representations match exactly
- In many situations, it is OK if two floating-point numbers are "close enough" even if they are not exactly equal
- If the difference between the two floating-point values is less than some tolerance (e.g., 0.000001) they are considered to be equal

```
if (Math.abs(f1 - f2) < TOLERANCE)
    System.out.println ("Essentially equal");
```

Comparing Characters

- Java character set is based on the Unicode character set
- Unicode establishes a particular integer value for each character, and therefore an ordering
  - the digits (0-9) are contiguous and in order
  - the uppercase letters (A-Z) and separately lowercase letters (a-z) are contiguous and in order
- Relational operators can be used to compare characters as integers based on the underlying integer ordering

Characters	Unicode Values
0 – 9	48 through 57
A – Z	65 through 90
a – z	97 through 122

Comparing Strings
String method `equals`

- We cannot use the relational operators to compare strings.
- But we can use methods provided by the `String` class type to compare strings.
- The `equals` method can be called with strings to determine if two strings are identical.
  - Returns `true` if the strings are identical (case must match) and `false` otherwise.
- Example (note how `equals` is used):

```
String name1 = "Joe";
String name2 = "Jim";
   ...
if (name1.equals(name2))
    System.out.println ("Same name");
```

Comparing Strings (contd.) String method `compareTo`

- The method call `name1.compareTo(name2)` returns (based on Unicode values)
  - zero if `name1` and `name2` are identical
  - a negative value if `name1` is less than `name2`
  - a positive value if `name1` is greater than `name2`

Characters	Unicode Values
0 – 9	48 through 57
A – Z	65 through 90
a – z	97 through 122

Comparing Strings – An Example

```
public class binaryTree {
    private String NodeName;
    private binaryTree left, right;
    …
    public boolean findName(String name) {
        int compareResult =
                NodeName.compareToIgnoreCase(name);
        if (compareResult == 0) return true;
        else if (compareResult < 0)  {
                if (right == null) return false;
                    return right.findName(name);
            }
            else {
                if (left == null) return false;
                    return left.findName(name);
            }
     }
}
```

Lexicographic Ordering (ordering based on character set)

- Lexicographic ordering is not strictly alphabetical when uppercase and lowercase characters are mixed
- For example, "`Great`" comes before "`fantastic`" because uppercase letters come before lowercase letters in Unicode

Characters	Unicode Values
0 – 9	48 through 57
A – Z	65 through 90
a – z	97 through 122

Loop Statements
(for repeated execution)

- Java has 3 kinds of loop statements
  - `while` statement
  - `for` statement
  - `do` statement

`while` Loop Statement

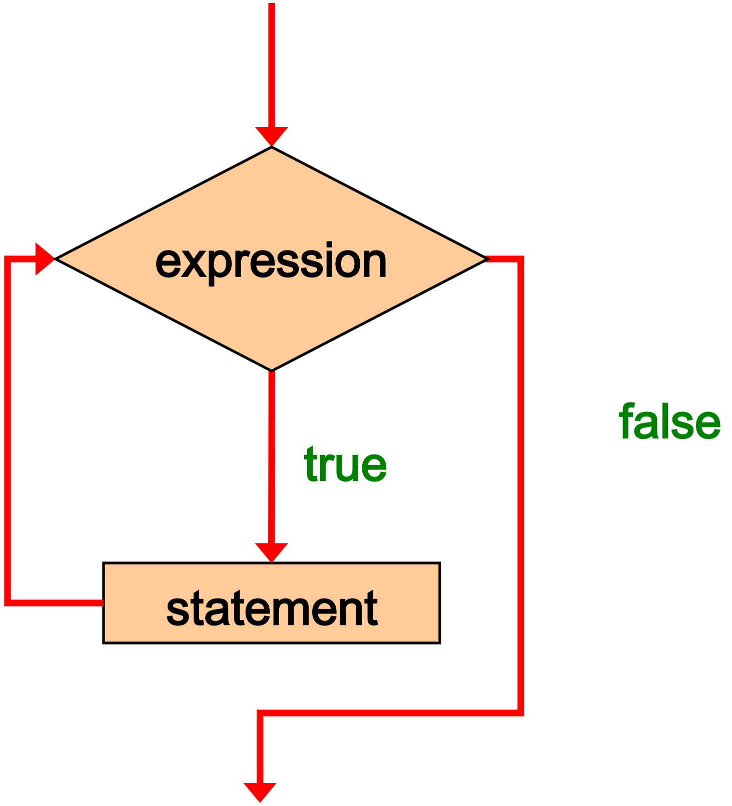
- The `while` loop is used to specify repeated execution of a *statement* as follows:

  `while` (*expression*)

  statement

- The `while` loop body, i.e., *statement*, will be executed repeatedly as long as *expression* evaluates to true.

`while` Loop Execution



`for` Loop

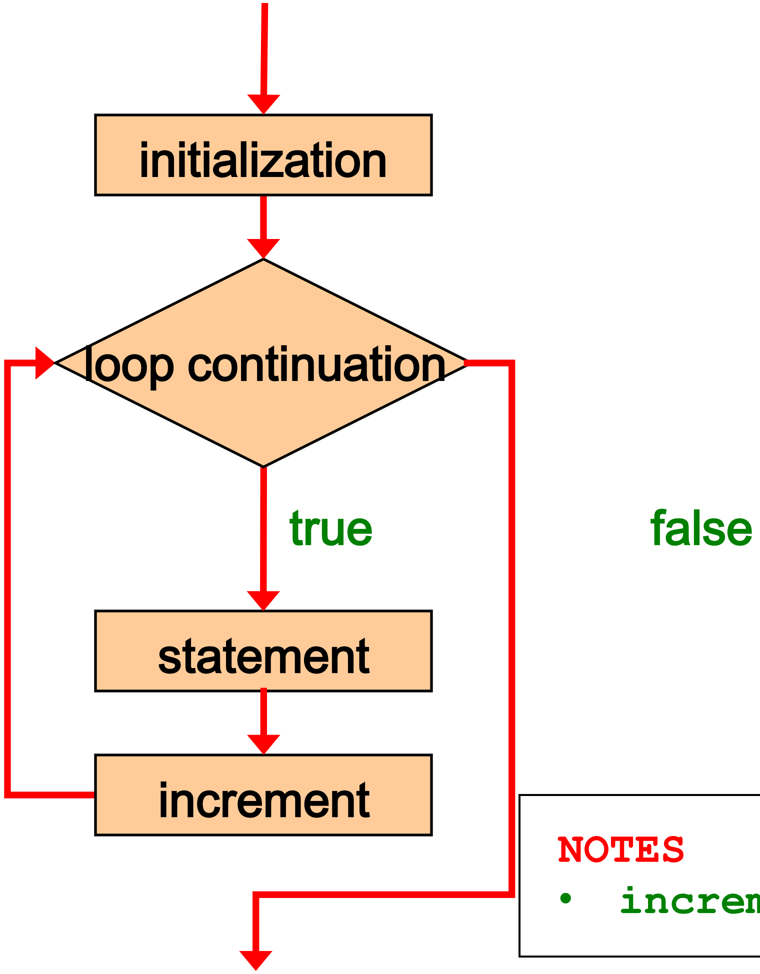
- The `for` statement has the form

  `for (`*initialization*`;` *loop continuation expression*`;` *increment*`)`
  statement

- The `for` statement starts execution
  - with some initial values
  - these are updated with each execution of the loop statement .
  - the loop is executed as long as these values satisfy the termination criteria

- Specifically
  - The *initialization* expression specifies initial values for variables used in the loop body, i.e., statement,
  - the *increment* expression updates these variables after each execution of the loop body, and
  - the *loop continuation* expression, if satisfied, represents the condition for continuing loop execution.

`for` loop execution



NOTES
- `increment/decrement`

The `for` Loop

- A `for` loop is equivalent to the following `while` loop :

```
initialization;
while(loop continuation)
{
    statement;
    increment;
}
```

`for` Loop (contd.)

- Consider, as an example, the following `for` statement:

```
for (int i=0; i < args.length; i++) {
    System.out.println(args[i]);
}
```

- This `for` statement
  - starts off with variable `i` equal to zero and
  - is executed as long as `i` is less than the length of array `args`.
- After each execution of the loop body, `i` is incremented by 1 prior to evaluation of the termination condition.
- Notice that here variable `i` is declared in the loop header.
  - therefore it can only be referenced inside the loop and not outside.

`for` Loop (contd.)

- Each expression in the header of a `for` loop is optional
  - If the initialization is left out, no initialization is performed
  - If the termination condition is left out, it is always considered to be `true`, and therefore creates a potentially infinite loop.
    - Some action must be taken inside the loop if it is to terminate
  - If the increment is left out, no increment operation is performed. **Note**: Instead of an increment, the loop can have a decrement.

`for` Loop (contd.)

- The following form of the `for` statement is often used to express loops that either
  - never terminate or
  - terminate with an explicit exit using a `break` or `return` statement or by throwing an exception.

```
for (;;) {
     …
}
```

- The above statement is equivalent to

```
for (; true ;) {
     …
}
```

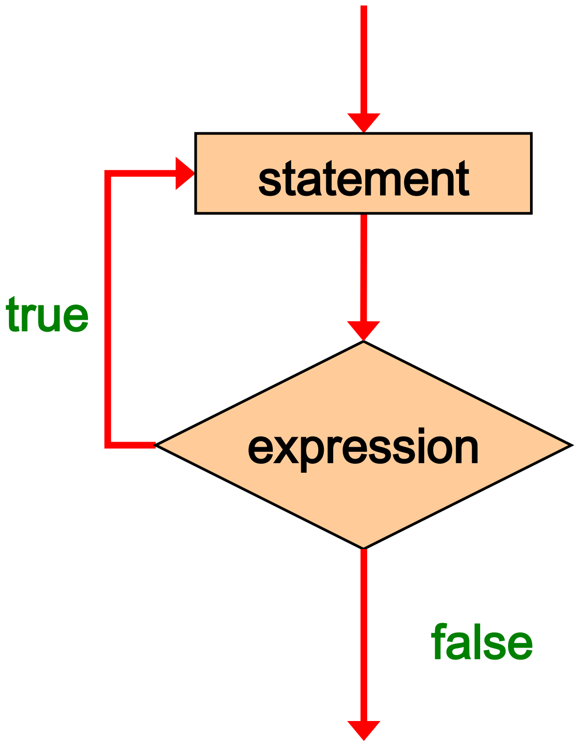
`do-while` loop

- In the `do-while` loop statement , the loop continuation test is performed after executing the loop body:

```
do
   statement
while (expression);
```

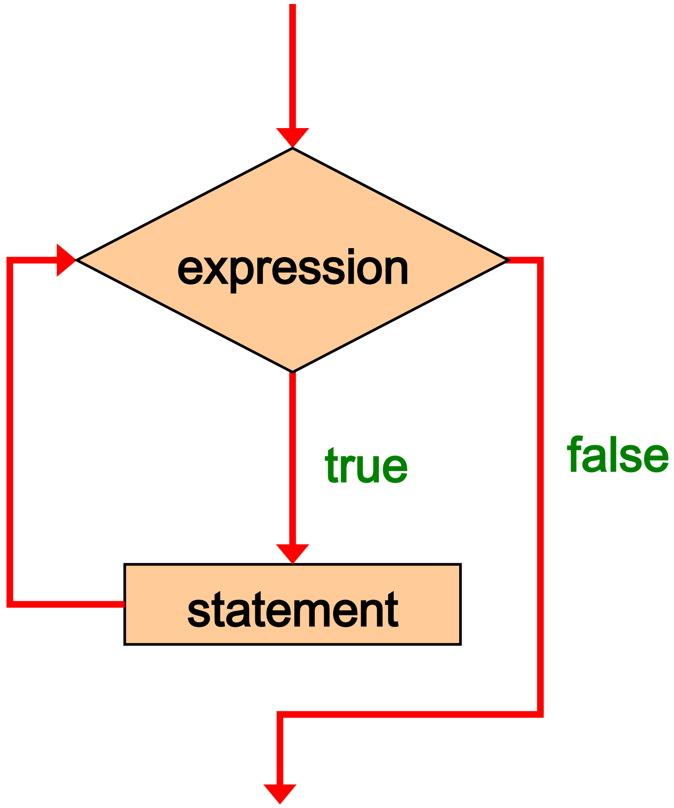
- Unlike the `for` and `while` loop statements, the body of a `do-while` loop is executed at least once regardless of the value of the `do-while` expression.

`do-while` Loop execution

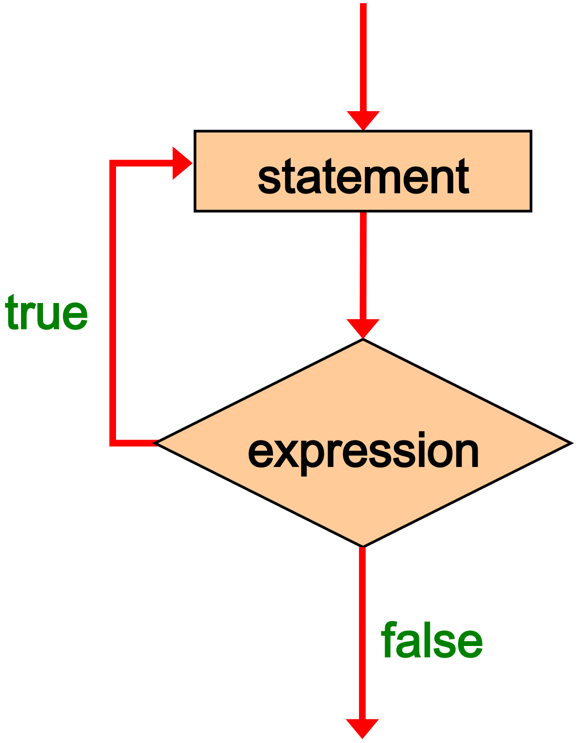


Comparing `while` and `do-while`

The `while` Loop



The `do-while` Loop



do-while example

```
//********************************************************************
//  ReverseNumber.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a do loop.
//********************************************************************

import java.util.Scanner;

public class ReverseNumber
{
   //-----------------------------------------------------------------
   //  Reverses the digits of an integer mathematically.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int number, lastDigit, reverse = 0;

      Scanner scan = new Scanner(System.in);
```

`do-while` example (contd.)

```
        System.out.print ("Enter a positive integer: ");
        number = scan.nextInt();

        do
        {
            lastDigit = number % 10;
            reverse = (reverse * 10) + lastDigit;
            number = number / 10;
        }
        while (number > 0);

        System.out.println("That number reversed is " + reverse);
    }
}
```

do-while → while
(it was not necessary to use do-while)

```
      System.out.print ("Enter a positive integer: ");
      number = scan.nextInt();

      while (number > 0);
      {
         lastDigit = number % 10;
         reverse = (reverse * 10) + lastDigit;
         number = number / 10;
      }
      System.out.println("That number reversed is " + reverse);
   }
}
```

Infinite loops

- Have to be careful of "infinite" loops. For example:

```
int left = 0, right = word.length() - 1;
while (left < right) {
    …
  left--; right++;
    // should be left++; right--;
}
```

Infinite loops (contd.)

- You might see code like this

```
while(true)
   statement
```

- Looks like an infinite loop.
- Is one – unless, *statement* is, for example, a block statement that containing a
  - `break` statement,
  - `return` statement or
  - `System.exit()` method call

  which is executed to exit the loop or terminate the program

Infinite loops (contd.)

How can you find an infinite loop?

Palindrome Word Checker

Develop program using pseudo code

and

stepwise refinement

Palindrome Word Checker

- What is a palindrome?
- Some examples?
- The palindrome word checker code shown next illustrates
  - an `if` statement
  - a `while` statement
  - an `if` statement nested in a `while` statement
  - a block statement

Palindrome Stepwise Refinement Level 0

Read word to be checked

If word reads same going forward
  and backwards then the word is a palindrome
  (need only do this for the left and right halves)

If word is palindrome

then print word is a palindrome

else print word is not a palindrome

Palindrome Stepwise Refinement Level 1

If word is not given as a command-line argument

then ask the user to supply the word on the keyboard

Assume word is a palindrome by setting variable palindrome = true;

// we will scan from the left and right to compare characters

let left pointer = position of first character in the word

let right pointer = position of the last character in the word

while (left < right) {

    if the left character is equal to the right character

    then advance left and right pointers

    else word is not a palindrome - set palindrome = false and exit loop

}

if (word is a palindrome) then print word is a palindrome

else print word is not a palindrome

Palindrome Word Checker

(illustrates `if`, `while`, `if` nested in `while`, block stmt.)

```
import java.util.*;  // for class Scanner
public class Palindrome{

   public static void  main (String[] args) {
      boolean palindrome = true;
      String word;

      if (args.length == 1) { // word in command-line?
         word = args[0];
      }
      else {  // get word to be checked
         Scanner scan = new Scanner(System.in);
         System.out.print("Enter word to be checked: ");
         word = scan.next();
      }
```

Palindrome Word Checker (contd,)

```
   int left = 0, right = word.length() - 1;
   while (left < right) {
       if (word.charAt(left) == word.charAt(right)) {
           left++; right--;
       }
       else {
           palindrome = false; break;
       }
    }
    if (palindrome)
     System.out.println(word + " is a palindrome");
    else
     System.out.println(word + " is NOT a palindrome");

  }
}
```

Palindrome Word Checker

- The palindrome word checker shown does not handle
  - mixed case words
  - multiple words (blanks ?)
- How will you fix it?
  - will discuss it again in more detail

`break` Statement

- The `break` statement

```
break;
```

is used to exit a loop or a `switch` statement.

- In both cases, the statement following the loop or `switch` statement is executed next

`continue` Statement

- The `continue` statement

```
continue;
```

is used in loops to exit the current loop iteration (execution) and start the next iteration.

`return` Statement

- The `return` statement has two forms

  `return;`

  `return` *expression*`;`

- It is used to terminate execution of a method (regardless of whether or not the `return` statement is inside a loop or a `switch` statement).
- The second form returns the value of the *expression* as the result of the method call

Iterators

- An *iterator* is an object that allows a collection of items to be processed one at a time
- An iterator class (a class that implements the `iterator` interface) must have at least two methods
  - a `hasNext()` method that returns `true` if there is at least one more item to process and `false` otherwise.
  - a `next()` method that returns the next item

Iterators (contd.)

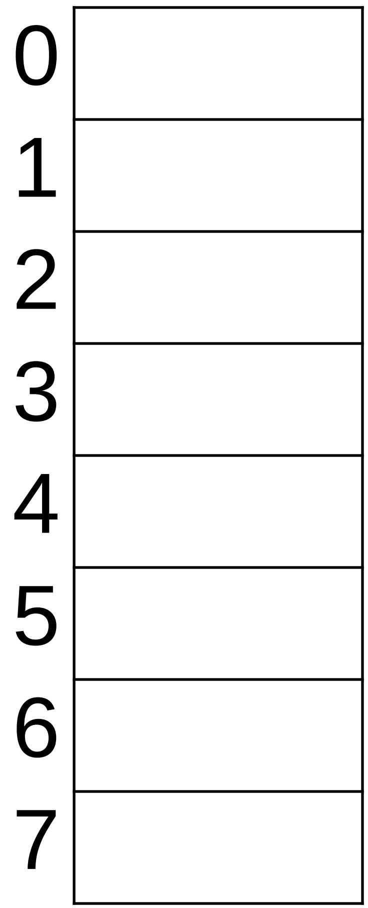
- Several classes in the Java standard class library are iterator classes
- For example, the `Scanner` class is an iterator. Its
  - `hasNext()` method returns `true` if there is more data to be scanned
  - `next()` method returns the next token as a string
- The `Scanner` class also has variations on the `hasNext()` and `next()` methods for specific data types (e.g., `hasNextDouble()` and `nextDouble()`)

Arrays
"Element Sequences"

Arrays

- A one-dimensional array is a sequence of elements that are accessed by specifying their position.
- An n-dimensional array type is a sequence of (n-1)-dimensional arrays
- Arrays are used to structure data (they are one type of data structure)

1-Dimensional Array



2-Dimensional Array
Elements accessed by specifying (row, col)

	0	1	2	3
0				
1				
2				
3				
4				
5				
6				
7				

1- D Array

This array was named `scores` when created

Each element is referenced with a numeric *index* (also called *subscript*)

	0	1	2	3	4	5	6	7	8	9
`scores`	79	87	94	82	67	98	87	81	74	91

- An array of size N is indexed from 0 to N-1
- This array is of size 10 – holds 10 values (79, 87, …) that have subscripts 0 to 9
- Was reated as

```
int[] scores = new int[10];
```

- Array elements have to be assigned values explicitly

Array Elements

- An array element is referenced using the array name followed by a number (or integer expression), called the index or subscript in brackets, e.g.,

```
scores[0]
```

- Array elements can be used like ordinary variables

```
scores[1] = 79; scores[1] = 87;
scores[first] = scores[first] + 2;
mean = (scores[0] + scores[1])/2;
System.out.println("Top = " + scores[5]);

pick = scores[10];
```

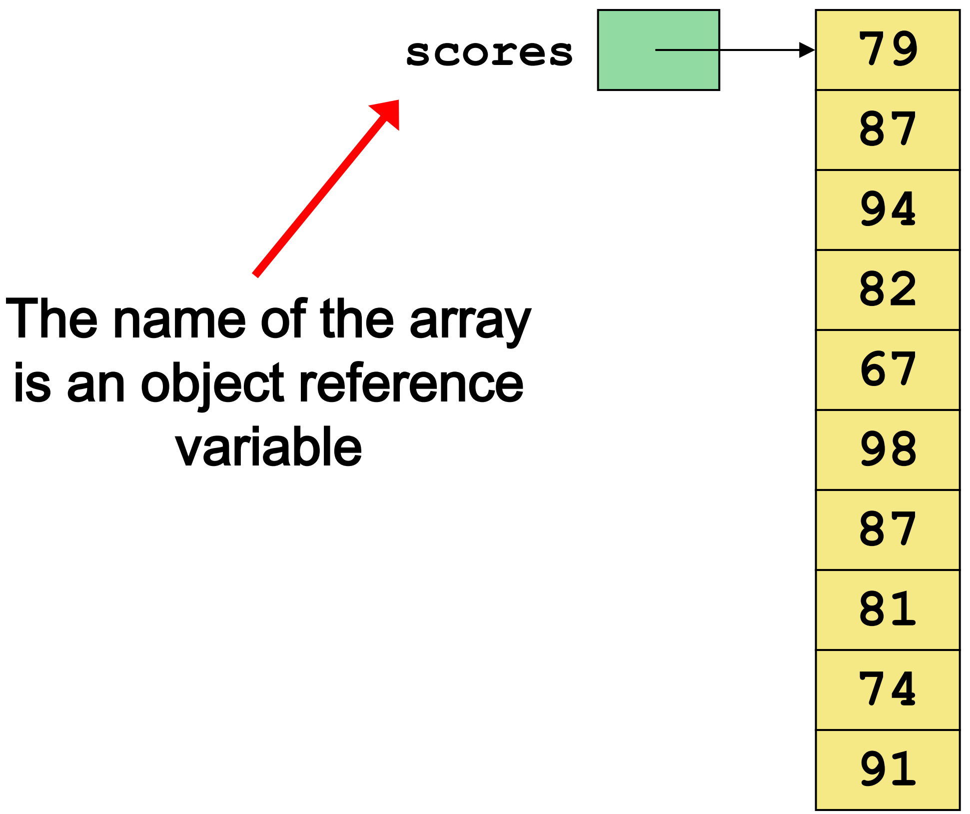
- Array scores is of size 10. Then why is the last statement not correct?

Array Element Types

- An array element type can be a primitive type or an "object reference"
- Arrays store multiple values of the same type
- We can create , for example, an array of
  - integers,
  - characters,
  - `String` objects,
  - `Coin` objects, etc.

An array is an object

- Variable `scores`, which we declared earlier, actually contains a reference to the array:



Array Creation

- Array creation requires two steps:
  - Array declaration
  - Array allocation
- These two steps can be combined into one

Array Variable Declarations

```
int[] year; char[] line;
double[][] sales;
String[] names;
int[][] chessBoard;
```

- `year` is declared as a one-dimensional (1-d) array of `int` elements,
- `line` is declared as a 1-d array of `char` elements,
- `sales` is declared as a 2-d array with `double` elements,
- `names` is declared as a 1-d array of `String` elements, and
- `chessBoard` is declared as a 2-d array of `int` elements

Array variables have been declared
but
but arrays have not been created yet, i.e., no storage has been allocated for them!

Array Creation (Allocation)

```
year = new int[12]; line = new char[80];
sales = new double [12][365]; // leap year?
names = new String[2];
chessBoard = new int[8][8];
```

- `year` now refers to as one-dimensional (1-d) array of 12 `int` elements,
- `line` now refers to a 1-d array of 80 `char` elements,
- `sales` now refers to a 2-d array of 12 x 365 `double` elements,
- `names` now refers to a 1-d array of `String` elements, and
- `chessBoard` now refers to 2-d array of 8 x 8 `int` elements

Array Creation (contd.)

- Separate Declaration and Allocation :

```
boolean[] flags;
flags = new boolean[20];
```

- Combined Declaration and Allocation

```
int[] weights = new int[2000];
double[] prices = new double[500];
char[] codes = new char[1750];
```

For Loops & Arrays

- In the next example, instead of the *for-each* loop as used in the book

```
for (int score : scores)
    System.out.println(score);
```

to read the values of all elements of array `scores`, we use the more general `for` loop

```
for (int i = 0; i < scores.length; i++)
    System.out.println(scores[i]);
```

where `scores.length` is the size of the array `scores`.

`for-each` loop notes

- The *for-each* version of the `for` loop is used for processing array elements:

```
for (int score : scores)
     System.out.println(score);
```

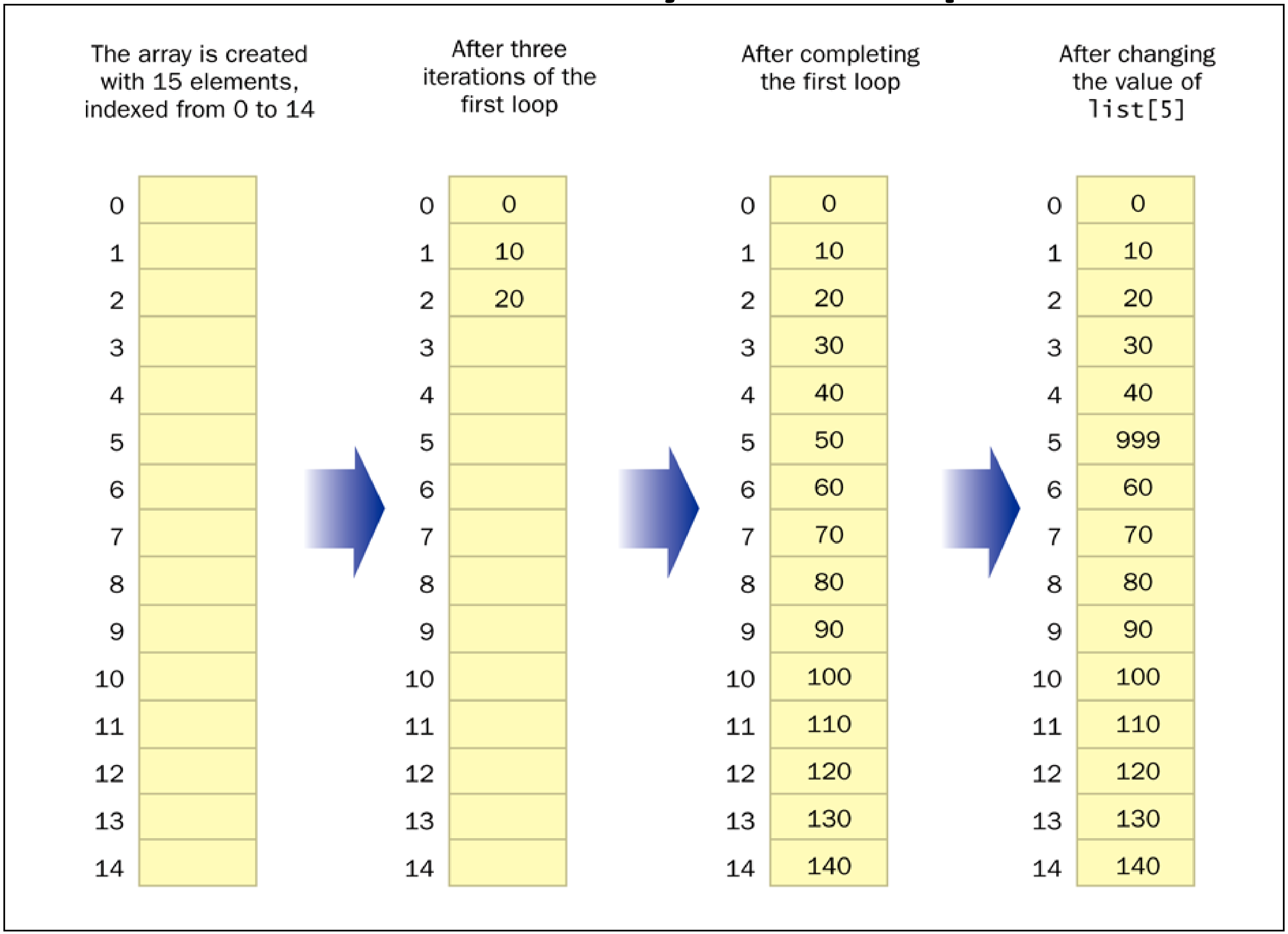
- Its use is appropriate only when processing all array elements, i.e., starting with the first element at index 0
- It cannoy be used to set /change array values.
- Consequently, we will not be using *for-each* loops. We will use the more general `for` loop.

Output

0   10   20   30   40   999   60   70   80   90   100   110   120   130   140

```java
public class BasicArray
{
   //-----------------------------------------------------------------
   //  Creates an array, fills it with various integer values,
   //  modifies one value, then prints them out.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int LIMIT = 15, MULTIPLE = 10;

      int[] list = new int[LIMIT];

      //  Initialize the array values
      for (int index = 0; index < LIMIT; index++)
         list[index] = index * MULTIPLE;

      list[5] = 999;  // change one array value

      //  Print the array values
      for (int index = 0; index < LIMIT; index++)
         System.out.print (list[index] + "   ");
   }
}
```

Basic Array Example



Bounds Checking

- Once an array, say of size N, is created, it has a fixed size
- The index used to reference an array element must specify a valid element
- In our case, the index value must be in range 0 to N-1
- If the index is out of range, called out of bounds) the Java interpreter will throw the exception

  `ArrayIndexOutOfBoundsException`

  – If this exception is not handled, the program will terminate
  – This is called *automatic bounds checking*

Array Bounds Checking

- If array `codes`, for example, is of size 100 , it has elements numbered from 0 to 99.
- If `count` == 100, then the following array reference will cause an exception to be thrown:

```
System.out.println(codes[count]);
```

- It's common to introduce *off-by-one errors* when using arrays:

problem

```
for (int index=0; index <= 100; index++)
    codes[index] = index*50 + epsilon;
```

Array Length

- Each array object has an attribute called `length` that yields the size of the array

- It is referenced using the array name:

```
scores.length
```

Array Initializer Lists

- An *initializer list* can be used to create and "populate" an array in one step. Two examples:

```
int[] units = {147, 323, 89, 933, 540,
                      269, 97, 114, 298, 476};

char[] grades = {'A', 'B', 'C', 'D', ’F'};
```

- The size of array
  - `units` is 10 (value of `units.length`) and
  - `grades` is 5 (value of `grades.length`)

Array Initializer Lists (contd.)

- Note that when an initializer list is used:
  - the `new` operator is not used
  - no size value is specified

- The size of the array is determined by the number of items in the list

- An initializer list can be used only in an array declaration

More Array Examples With Explicit Initialization

```
int[] year = {1997, 1998, 1999, 2000};
char[] line = {'d', 'a', 't', 'e'};
float[][] sales2
      = {{97,1}, {97,2}, {97,3},{97, 4}};
String[] names = {"Lisa", "Tom"};
```

Array Elements

- In the above declarations,
  - `year` is initialized to refer to a 1-d `int` array with 4 elements numbered from 0 to 3.
  - `sales` is initialized to refer to a 2-d array with 8 elements numbered from 0, 0 to 3, 1.
- 2-d array elements are referenced by specifying the row and column as [row][column
- Here are 2 examples illustrating array elements references:

```
year[0], sales2[2][1]
```

- If a non-existing array element is referenced, e.g., year[11], then exception

```
ArrayIndexOutOfBoundsException
```

is thrown.

Array Length

- The length of an array can be determined by referencing its **`length`** attribute
- For example, assuming arrays `names` and `sales` are declared as shown above, the following expressions

```
names.length
sales.length
sales[0].length
```

- yield the lengths of
  - the 1-d array `names`,
  - the first dimension of the 2-d array `sales`, and
  - the second dimension of the sales (i.e., 2).
- Note that a 2-d array is an array whose elements are 1-d arrays.

```java
//********************************************************************
//  Primes.java       Author: Lewis/Loftus/modified by Gehani
//
//  Demonstrates the use of an
//  initializer list for an array.
//********************************************************************

public class Primes
{
   //-----------------------------------------------------------------
   //  Stores some prime numbers in an array and prints them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int[] primeNums = {2, 3, 5, 7, 11, 13, 17, 19};

      System.out.println ("Array length: " + primeNums.length);

      System.out.println ("The first few prime numbers are:");

      for (int i = 0; i < primeNums.length; i++)
         System.out.print(primeNums[i] + "  ");
   }
}
```

Array Example - ReverseList

- Write a program that reads a list of integers and prints them in the reverse order.
- The data is stored in a file that is supplied as a command-line argument and has the form

$$n\ x_1\ x_2\ \ldots\ x_n$$

where n is the number of x values that follow.

Reading & Printing Arrays

```
// Reads and Prints a list of n integers stored in a file
// specified as a command-line argument in the format
// n x1 x2 x3 ... xn
// Array printed in two ways - element-by-element
// and default array conversion to string

import java.io.*;  // for class File
import java.util.*; // for  class Scanner

public class  PrintArray {
  public static void main(String args[]) throws IOException {

      // SET UP FILE FROM WHICH TO READ DATA
      if (args.length == 0) {
        System.out.println("Please supply data file" +
                           " as command-line argument");
         System.exit(0);
      }
      File inputDataFile = new File(args[0]);
      Scanner inputFile = new Scanner(inputDataFile);
```

`for` Loop Example – Read & Print array

```
// READ DATA FROM FILE
      int n = inputFile.nextInt();
      int list[] = new int[n];

      for (int i = 0; i < n; i++)
         list[i] = inputFile.nextInt();

// PRINT ARRAY EXPLICITLY
      for (int i = 0; i < n; i++)
         System.out.print(list[i] + " ");
      System.out.println();

// PRINT ARRAY EXPLICITLY
      for (int i = 0; i < n; i++)
         System.out.print(Integer.toString(list[i])+ " ");
      System.out.println();

// PRINT ARRAY USING JAVA CONVERSION - NO GOOD
      System.out.println(list);
      System.exit(0);
   }
}
```

`for` Loop Example – Input Data

```
11
1
2
3
4
5
6
7
8
9
10
11
```

`for` Loop Example – Output

```
1 2 3 4 5 6 7 8 9 10 11
1 2 3 4 5 6 7 8 9 10 11
[I@4ded4d06
```

ReverseList

Develop program using pseudo code

and

stepwise refinement

ReverseList – Level 0

if the data file is not specified as a command-line argument then exit

set up the file to read the list of numbers

read the size of list

create an array to hold the list

read the numbers into the array

print the numbers in the array in reverse order

Array Example – ReverseList (contd.)

```
// Reverses a list of n integers stored in a file
// given as a command-line argument in the format
// n x1 x2 x3 ... xn
// Author: N. Gehani

import java.io.*;  // for class File
import java.util.*; // class Scanner
public class ReverseList {
   public static void main(String[] args)
                                throws IOException {
     // must throw exception FileNotFoundException
     // or catch it (happens when opening file by
     // creating a new Scanner object)
```

Array Example – ReverseList (contd.)

```
// SET UP FILE FROM WHICH TO READ DATA

  if (args.length == 0) {
      System.out.println("Please supply data file" +
                        " as command-line argument");
      System.exit(0);
  }

  File inputDataFile = new File(args[0]);
  Scanner inputFile = new Scanner(inputDataFile);
```

Array Example – ReverseList (contd.)

```
// READ DATA FROM FILE
      int n = inputFile.nextInt();
      int list[] = new int[n];

      for (int i = 0; i < n; i++)
         list[i] = inputFile.nextInt();

// PRINT LIST IN REVERSE ORDER
      for (int i = n-1; i >= 0; i--)
         System.out.println(list[i]);

    }
}
```

Array Example – ReverseList (contd.)

- Can this list reversal be done with out an array?
- Justify your answer.

Program Example
Counting Lower case Letters in a String

Storing Statistics about Lower-case Letters

Characters	Unicode Values
0 – 9	48 through 57
A – Z	65 through 90
a – z	97 through 122

- We will store statistics of lower case occurrences in array lower:

```
int[] lower = new int[26];
```

- But we will need to map letters **a** to **z**, that is their integer representations from

  97 to 122 → 0 to 25

```java
// Author:Gehani - modfied version of Lewis/Loftus
// Reads a line & counts lowercase letter occurrences
import java.util.Scanner;
public class LowerCaseLetterCount {
   public static void main (String[] args){
      final int NUMCHARS = 26;
      int[] lower = new int[NUMCHARS];
      Scanner scan = new Scanner(System.in);
      char current;   // the current character being processed

      System.out.println ("Enter a sentence:");
      String line = scan.nextLine();

      //  Count letter occurrences
      for (int ch = 0; ch < line.length(); ch++) {
         current = line.charAt(ch);
         if (current >= 'a' && current <= 'z')
            lower[current-'a']++;
      }
      //  Print letter count
      for (int letter=0; letter < lower.length; letter++) {
         if (lower[letter] != 0)
            System.out.printf("\t%c: %d\n", letter + 'a', lower[letter]);
      }
   }
}
```

Object-Oriented Programming

Object-Oriented Programming

- Application development is simplified if developed in terms of the objects it deals with
- Objects can be
  - real like cars or
  - virtual like accounts

Java objects are instances of types called <u>classes</u>

Example: University Management System

- Objects
  - Students
  - Faculty
  - Staff
  - Accounts
  - Courses
- Some are related
  - Staff and Faculty are different types of Employees
- Each object has its own functionality

Example: University Management System (contd.)

- `Course` Class (Object Type)
  - Query: title, abstract, meeting times, …
  - Update: title, abstract, meeting times, …
- Not everything can be updated or changed
- Different users have different capabilities

What is a Class?

- Its an object type (template) which is used to create objects.
- A class provides
  - fields to store data (object state)
  - methods to query & update object state (object behavior)
- The object state comes into existence only after an object is created.
- A class can be used to create as many objects as needed.
- A program can have as many classes as the application needs.

Classes & Data Encapsulation

- Hiding internal state (i.e., fields) from a user of a class by requiring all interaction with an object be performed via the object's methods is known as *data encapsulation — a key principle of object-oriented programming*.
- Occasionally, fields may be accessed directly for ease of use especially if they are declared as constants or for reading *– the latter does weaken data encapsulation*

Predefined Classes

- Many predefined classes – some very important.
- Some examples:
  - **`String`**
  - **`Scanner`**
  - **`Math`**

Declaring & Creating Objects

- 3-step process prior to using objects
  - Defining the class (object type)
  - Declaring the object (actually a place holder for a reference to the object)
  - Creating the object (with the object allocator `new`) which yields a reference to the object
- An object declaration only creates a location to store the address of an the object
- The object must be explicitly created

Object Declaration Examples

- Example Classes
  - `String` is part of Java language
  - `binaryTree` is user-defined
  - `File` & `Scanner` are predefined in Java

- Declarations (no object/storage allocation)

```
String name, searchName;
binaryTree bTree;
File inputDataFile;
Scanner inputFile, scan;
```

Object Creation Examples

Objects are created using the object allocator `new`

```
bTree = new binaryTree();

inputDataFile = new File(args[0]);

inputFile = new Scanner(inputDataFile);

scan = new Scanner(System.in);

searchName = scan.nextLine();
                    //method call
```

Combined Object Declaration & Creation

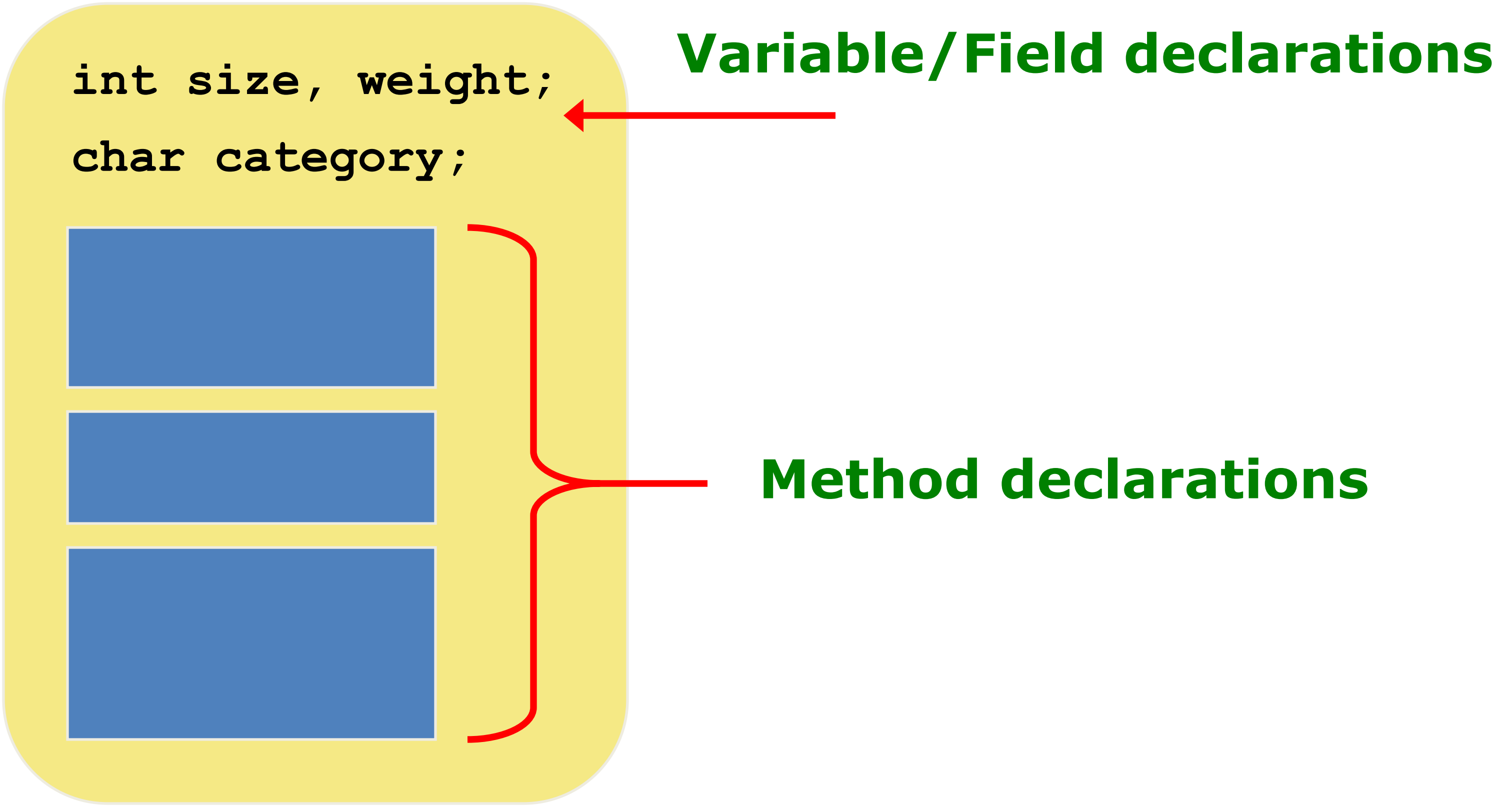
```
binaryTree bTree = new binaryTree();
File inputDataFile = new File(args[0]);
Scanner inputFile = new
Scanner(inputDataFile);
Scanner scan = new Scanner(System.in);
```

Objects (Class Type Instances)

- An object
  - stores its state in its own *fields* (variables) and
  - exposes its behavior through *methods* (functions).
- Methods operate on (query & update) an object's internal state and are used for communicating with the object.

Classes (Object Type)

- A class can contain data declarations and method declarations



Simplified Variable/Field Declarations Introduced Earlier

- Variable declarations introduced earlier

  type { *identifier* [= *initial-value*] }$^+$ *;*

  where the variable name is *identifier*
- We have seen some examples

```
final int MAX = 1024;
int[][] chessBoard;
int list[] = new int[n];
int i = 0, j = 0, k;
```

More on Variable/Field Declarations

- Declarations can have a modifier that controls access

  [*modifier*] *type* { *identifier* [= *initial-value*] }$^+$

- *modifier* can be
  - `final`: specifies a constant "variable"
  - `static`: specifies a class (not an object) variable
  - `public`: variable/field can be accessed from everywhere
  - `private`: variable/field can be accessed only from within class
  - `protected`: variable/field can be accessed only from the class or classes "derived" from it and the package containing it
  - unspecified, i.e., no modifier: variable/field can be accessed from the class and the package containing it

Simple Class Definition Format

```
class classname {
    field-declarations
    constructor-definitions
    method-definitions
}
```

- We will first examine class use and leave class definitions and their general format to later.

Methods & Method Invocation

- A *method* is a collection of statements grouped together to perform an operation on object.
- Method invocations have the form
  - *variable.method*([*arguments*])
  - *expression.method*([*arguments*])
  - *className.method*([*arguments*])

  where *variable* and *expression* refer to an object. The class name is used for **`static`** methods which are not associated with an object.
- Methods may or may not return a value.

Note: [**x**] means **x** is optional

Constructors

- Constructors are specialized object initialization methods that are called when an object is created as in, e.g.,

  … **`new`** className ([*arguments*])

- Constructors have the same name as the class containing them.
- Constructors do not return a value.
- A class can have multiple constructors – the one used is based on the arguments supplied (number & type) when creating the object.
- If no constructor is explicitly defined, Java provides a default constructor (without any arguments)

Method & Constructor Invocation Examples

```
char opr; double a;

Scanner stdin = new Scanner(System.in);
a = stdin.nextDouble();
opr = stdin.next().charAt(0);
```

Constructor Invocation

- Example:

```
Scanner stdin = new Scanner(System.in);
```

- A `Scanner` object is created & assigned to `stdin`.
- It is initialized by the `Scanner` constructor which is given `System.in` as an argument.
  – `System.in` is of type `InputStream`.
- Class `Scanner` has several constructors (see specification)
- The constructor used above is the one that takes one argument of type `InputStream`
  – argument can also be of type `File` or `String` which will invoke different constructors

Constructor Invocation (contd.)

- We refer to the field `in` as `System.in` using the class name `System` instead of an object name.
  - This is because `in` is declared as a **`static`** field in the definition of class `System`.
  - `static` fields are associated with the class and not with class objects.
  - `static` fields are used
    - to share data between object instances
    - when a class is used as a "packaging" facility and will typically not be used to create objects

More on static fields and methods later.

Method Invocation

- Example:

```
double a = stdin.nextDouble();
```

- Method `nextDouble()` of the `Scanner` object `stdin` is invoked.
- `nextDouble()` returns a `double` value
  - to be precise, it returns the next value in the input stream which must be a `double`.

Method Invocation (contd.)

- Example:

```
opr = stdin.next().charAt(0);
```

- The *dot* operator associates left to right , so the above expression of the right side of the assignment is equivalent to

```
(stdin.next()).charAt(0)
```

- `stdin` is an object of class type `Scanner.`
- The `next()` method of class `Scanner` is first called to retrieve a token (of type `String`) from the input stream.
- Then the `String` method `charAt()` is applied to the token to retrieve its first character.

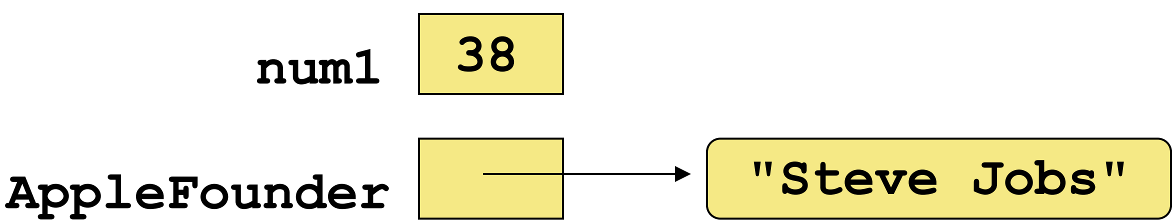
Fields

- Fields are referenced using the dot operator
- Field references have the form
  - *variable.fieldName*
  - *expression.fieldName*
  - *className.fieldName* (for `static` fields)
- They can be used in expressions

Object Variables

- A primitive type variable stores a value itself, but an object variable stores the address (location) of the object

- An object variable can be thought of as
  - containing a pointer to the location of the object or
  - a reference to the object



Primitive Type Variables – Assignment

The value assigned is stored in the variable

Before:

`num1`	38
`num2`	96

```
num2 = num1;
```

After:

`num1`	38
`num2`	38

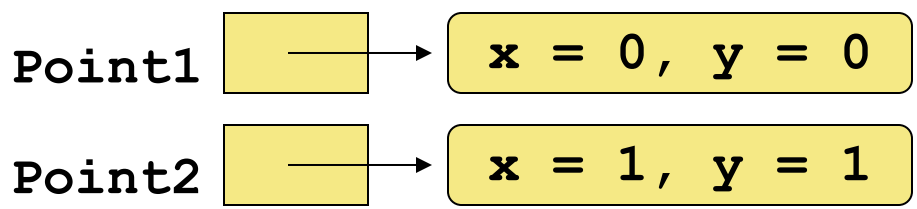
Class Point

```
public class Point {
    double x, y;
    public Point(int x, int y) {
        x = 0; y = 0;
    }
    …
}

…

Point1 = new Point(0, 0);
Point2 = new Point(1, 1);
```

Object Variables – Assignment

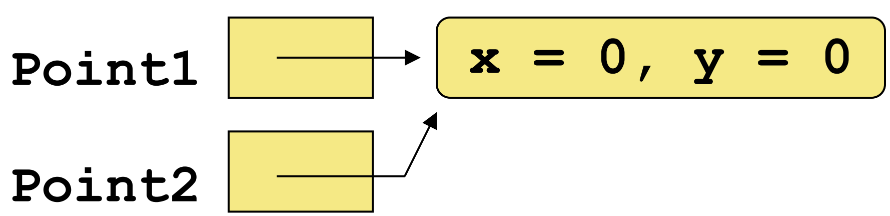
Assignment means copying the object address.

Before:




```
Point2 = Point1;
```

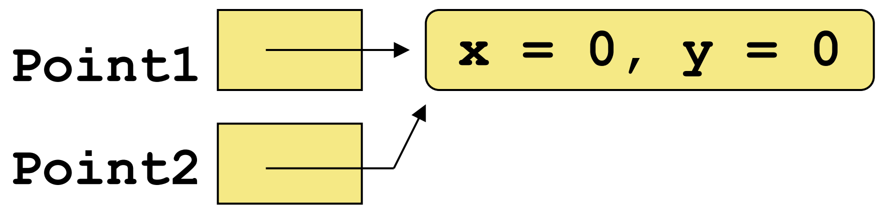
After:



Object Variables – Assignment

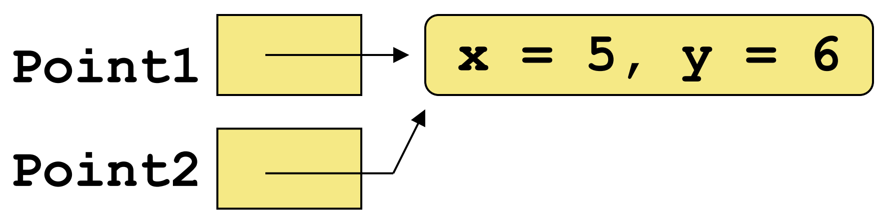
Assignment means copying the object address.

Before:




```
Point1.x = 5;
Point2.y = 6;
```

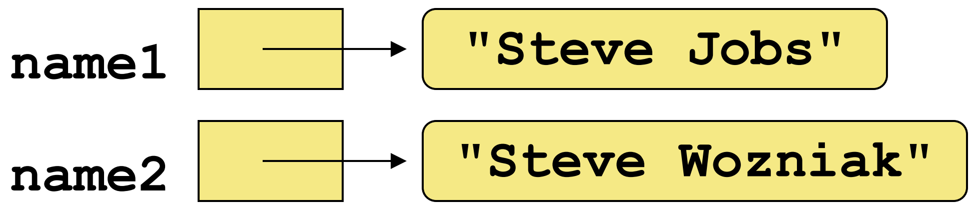
After:



Object Variables – Assignment

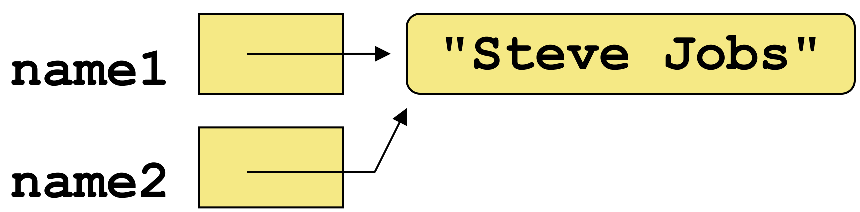
Assignment means copying the object address.

Before:




```
name2 = name1;
```

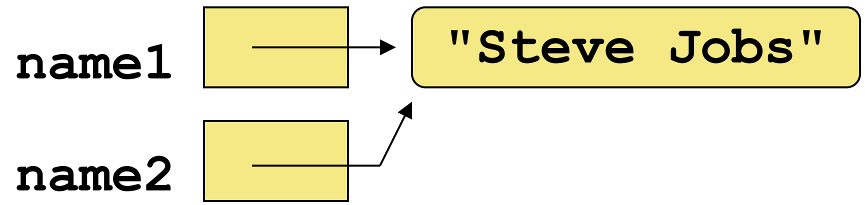
After:



Strings are Immutable (special objects)

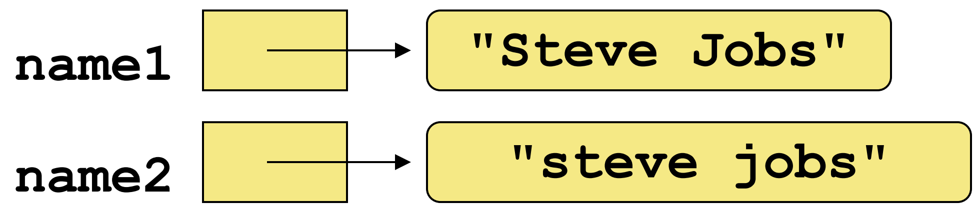
A string object cannot be modified – a new object is created

Before:

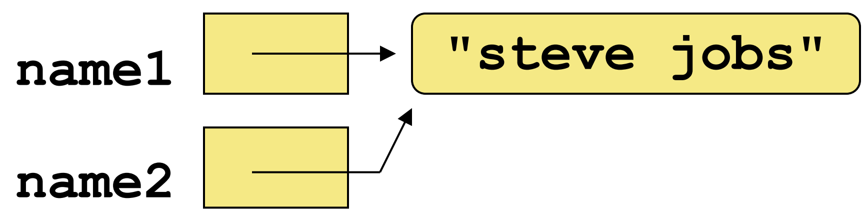



```
name2 = name1.toLowerCase();
```

After:



NOT:



Object Variables – Aliases

- Two or more object variables that refer to the same object are called *aliases* of the object.
- Aliases can be useful, but care is needed in tracking who refers to what
- As seen, changing an object through one alias changes it for all of its aliases.

Comparing Objects

- The comparison

  $$objectReference_1 == objectReferece\ _2$$

  returns `true` if the two object references are aliases of the same object (but not otherwise even if they point to objects with the equal values) and `false` otherwise.
- Method `equals` is pre-defined for all objects
  - Unless redefined in a class definition, it is equivalent to the == operator
  - It is redefined in the `String` class to compare characters in the two strings
  - When defining a new class, `equals` should be redefined if needed and as appropriate

Garbage & its Collection

- When an object no longer has any valid references to it, it can no longer be accessed in the program

```
binaryTree bTree1 = new binaryTree();
binaryTree bTree2 = new binaryTree();
…
bTree2 = bTree1;
```

- `bTree2` and `bTree1` now point to the same object – the one pointed to by `bTree1`.
- The object originally pointed to by `bTree2` cannot be accessed any more.
- Its address (reference to it) is lost. It is therefore *garbage*
- Java performs automatic garbage collection periodically, returning an object's memory to the system for future use

String Class

The String Class

- Although `String` is a class, its support in Java is so deep that it can be thought of as a primitive type.
- For example, the concatenation operator `+` is part of Java.
- Also, when allocating `String` objects it is not necessary to use the object allocator `new` which is required of other class objects, e.g.,

```
System.out.print("Enter Operator: ");
String greeting = "Good Morning!"
final String NJIT = "New Jersey Institute of Technology";
```

- Each string literal is a `String` object

`String (String str)`
Constructor: creates a new string object with the same characters as `str`.

`char charAt (int index)`
Returns the character at the specified `index`.

`int compareTo (String str)`
Returns an integer indicating if this string is lexically before (a negative return value), equal to (a zero return value), or lexically after (a positive return value), the string `str`.

`String concat (String str)`
Returns a new string consisting of this string concatenated with `str`.

`boolean equals (String str)`
Returns true if this string contains the same characters as `str` (including case) and false otherwise.

`boolean equalsIgnoreCase (String str)`
Returns true if this string contains the same characters as `str` (without regard to case) and false otherwise.

`int length ()`
Returns the number of characters in this string.

`String replace (char oldChar, char newChar)`
Returns a new string that is identical with this string except that every occurrence of `oldChar` is replaced by `newChar`.

`String substring (int offset, int endIndex)`
Returns a new string that is a subset of this string starting at index `offset` and extending through `endIndex-1`.

`String toLowerCase ()`
Returns a new string identical to this string except all uppercase letters are converted to their lowercase equivalent.

`String toUpperCase ()`
Returns a new string identical to this string except all lowercase letters are converted to their uppercase equivalent.

FIGURE 3.1 Some methods of the `String` class

Revisiting Palindrome Word Checker

- How will you handle mixed case?
- How will you handle multiple word palindromes (i.e., how will you handle blanks)?

<u>Clue</u>

Look at the specification of class `String`

Palindrome Word Checker (contd.)

- Consider using
  - Method `toLowerCase()`
  - Method `replace()` (to replace blanks with null strings)
  - Method `nextLine()` (to read the whole line – can contain words separated by blanks)

To See Details

Look at specifications of classes `String` & `Scanner`

Palindrome Word Checker (contd.)

- To handle upper and lower case and blanks
  - Convert the word to lower case
  - Replace blanks with null strings

```
String word2 =
    word.toLowerCase().replace(" ", "");
```

- Now use `word2` instead of `word` for checking if the word(s) supplied represent a palindrome.

Standard Class Library
(and looking at a couple of classes)

Package	Provides support to
java.applet	Create programs (applets) that are easily transported across the Web.
java.awt	Draw graphics and create graphical user interfaces; AWT stands for Abstract Windowing Toolkit.
java.beans	Define software components that can be easily combined into applications.
java.io	Perform a wide variety of input and output functions.
java.lang	General support; it is automatically imported into all Java programs.
java.math	Perform calculations with arbitrarily high precision.
java.net	Communicate across a network.
java.rmi	Create programs that can be distributed across multiple computers; RMI stands for Remote Method Invocation.
java.security	Enforce security restrictions.
java.sql	Interact with databases; SQL stands for Structured Query Language.
java.text	Format text for output.
java.util	General utilities.
javax.swing	Create graphical user interfaces with components that extend the AWT capabilities.
javax.xml.parsers	Process XML documents; XML stands for eXtensible Markup Language.

FIGURE 3.2 Some packages in the Java standard class library

The Math Class

- Part of package `java.lang`
- Contains methods that perform math functions:
  - absolute value, square root, exponentiation, trigonometric functions, generate random numbers
- `Math` class methods are `static` methods
  - Static methods are invoked using the class name (in this case using the name `Math`)
  - no `Math` object is needed, e.g.,
- Examples

```
value = Math.cos(90) + Math.sqrt(delta);
```

```
static int abs (int num)
```
Returns the absolute value of num.

```
static double acos (double num)
static double asin (double num)
static double atan (double num)
```
Returns the arc cosine, arc sine, or arc tangent of num.

```
static double cos (double angle)
static double sin (double angle)
static double tan (double angle)
```
Returns the angle cosine, sine, or tangent of angle, which is measured in radians.

```
static double ceil (double num)
```
Returns the ceiling of num, which is the smallest whole number greater than or equal to num.

```
static double exp (double power)
```
Returns the value e raised to the specified power.

```
static double floor (double num)
```
Returns the floor of num, which is the largest whole number less than or equal to num.

```
static double pow (double num, double power)
```
Returns the value num raised to the specified power.

```
static double random ()
```
Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

```
static double sqrt (double num)
```
Returns the square root of num, which must be positive.

FIGURE 3.5 Some methods of the Math class

Generating Pseudo-random Numbers

- Two ways of generating random numbers
  - Use method `Math.random()` to generate random `double` values between 0 (inclusive) and 1.0 (exclusive)
  - Use methods of the `Random` class to generate random `int`, `long`, `double`, … values.
- When using the `Random` class, the user creates a `Random` object and uses that to create a series of random numbers.
  - use different `Random` objects for different series of random numbers.

The Random Class
(Examples of Use)

```
import java.util.*;
Random generator = new Random();
int num1; float num2;

num1 = generator.nextInt();
        // Random integer
num1 = generator.nextInt(10);
        // Random integer between 0 & 9
```

The Random Class (contd.)

```
num2 = generator.nextDouble();
      // A random double between 0 and 1
      // includes 0 but not 1.0
num2 = generator.nextDouble() * 6;
      // 0.0 to 5.999999
```

Details of printing output

using

`printf()`

and more

Printing Output

- We have seen 2 ways of printing output
  - Using `print()` / `println()` along with default conversions of values to `String`
  - Using `printf()` and its format facilities

2 printing options

```
// printing using automatic conversion
// of numbers to strings
System.out.println("Roots of quadratic equation\n" +
         a + "x*x + " + b + "x + " + c + " = 0\n" +
         "x = " + x1 + " and x = " + x2);
// printing using C style print facility
System.out.printf("Roots of quadratic equation\n" +
      "%.2fx*x + %.2fx + %.2f = 0\n" +
      "x = %.2f and x = %.2f\n", a, b, c, x1, x2);
```

2 printing options (contd.)

```
System.out.println("Roots of quadratic equation\n"+
         a + "x*x + " + b + "x + " + c + " = 0\n" +
         "x = " + x1 + " and x = " + x2);
```

```
Roots of quadratic equation
1.0x*x + 7.0x + 2.0 = 0
x = -0.29843788128357573 and x = -6.701562118716424
```

- Java automatically converts the double variables `a`, `b`, `c`, `x1`, `x2` to strings using the method `Double.toString()`.
- `Double` is a wrapper class for the primitive type `double`
- We will be talking more about wrapper classes – each primitive type has a wrapper class wit a variety of methods for conversion etc.

Printing with `printf()`

Formatted printing for the Java language is heavily inspired by C's `printf()` *function* – Oracle's Java documentation

- The `printf()` method has the form

  System.out.printf(*formatString*, arg_1, …, arg_n)

  The format string specifies how the arguments arg_i are to be printed

Printing with `printf()` (contd.)

```
// printing using C style print facility
System.out.printf("Roots of quadratic equation\n" +
     "%.2fx*x + %.2fx + %.2f = 0\n" +
     "x = %.2f and x = %.2f\n", a, b, c, x1, x2);
```

```
Roots of the quadratic equation
1.00x*x + 7.00x + 2.00 = 0
x = -0.30 and x = -6.70
```

- Format specifications begin with `%` sign.
- The number of format specifications must match the number of arguments.
- The first `%.2f`, for example, specifies that the first argument, `a`, is to be printed as a floating-point number with 2 fractional digits (and as many digits on the left of the decimal point as needed).
- `\n` specifies a new line

Printing with `printf()` (contd.)

Format Specification	Semantics
`%b`	boolean
`%c`	character
`%d`	decimal integer (`%`*width*`d`)
`%e`	Float/double number (scientific notation)
`%f`	Float/double number (`%`*width.precision*`f`)
`%n`	new line
`%s`	string
`%%`	prints `%`

Printing (Writing) Output to a file

```
File outFile = new File("FILE NAME");
PrintStream outStream = null;
try {// open the file for writing
    outStream = new PrintStream(outFile);
}
catch (FileNotFoundException e) {
    System.out.println("File %s not found!", "FILE NAME");
    System.exit(0);
}
    …
outStream.println("STRING TO BE WRITTEN");
    …
outStream.printf("FORMAT STRING", ARG1, ... );
    …
outStream.close();
```

Printing Objects

- Java converts , when needed, an object type automatically to a `String`
- It uses the default method
- `Object.toString()`
- Class `Object` is the root class of all class types
- The default `toString()` method does not print very useful information (prints the class name + address of the object).
- Each class should provide its own `toString()` method

Printing Arrays

- Arrays are printed element-by-element.
  - No default way of printing the whole array

Defining Classes

Writing Classes

- The programs we have written in previous examples have used classes defined in the Java standard class library
- Now we will begin to design programs that use classes that we write.
- The class that contains the `main` method is just the starting point of a program
- True object-oriented programming is based on defining classes that represent objects with well-defined characteristics and functionality

Some Examples of User-Defined Classes
(not all possible methods are listed)

Class	Attributes	Operations
Student	Name Address Major Grade point average	Set address Set major Compute grade point average
Rectangle	Length Width Color	Set length Set width Set color
Aquarium	Material Length Width Height	Set material Set length Set width Set height Compute volume Compute filled weight
Flight	Airline Flight number Origin city Destination city Current status	Set airline Set flight number Determine status
Employee	Name Department Title Salary	Set department Set title Set salary Compute wages Compute bonus Compute taxes

General Form of Class Definitions

[*modifiers*] `class` *classname* {

    [*static-initializers*]

    [*field-declarations*]

    [*constructor-definitions*]

    [*method-definitions*]

}

Note: [*x*] means *x* is optional

Classes and Objects

- As discussed earlier, a class defines object *state* and *behavior*
- Consider a six-sided die (singular of dice)
  - It's state can be defined as the face value that is showing
  - It's primary behavior is that it can be rolled
- We will represent a die by designing a class called `Die` that models this state and behavior
  - The class will serve as the blueprint for a die (`Die` object)
- With this class, we can instantiate as many dies (`Die` objects) as we need for any particular program
  - In some games you need one die, in others more

Classes and Objects (contd.)

- The values of the data specified in the class definition specifies the state of an object
- The functionality of the methods define the behavior of the object
- Initial values are defined by constructors (if provided)

The `toString` Method

- It's good practice to define a `toString()` method for each class
- The `toString()` method returns a string that represents the object in some way
- It is called automatically when an object is to be converted to a string, e.g., when it is passed to the `println()` method

`Die` Class – An Example

- We define a `Die` class to represent dies
- In the `Die` class, we will specify the die state to be an integer field (attribute) called `faceValue`
  - represents the current face value (the state of the die)
- We will define several methods to implement die behavior and examine the face value.
  - One method implements “rolling” a die by setting `faceValue` to a random number between 1 and 6

```java
//********************************************************************
//  Die.java       Author: Lewis/Loftus
//
//  Represents one die (singular of dice) with faces showing values
//  between 1 and 6.
//********************************************************************

public class Die
{
   private final int MAX = 6;  // maximum face value

   private int faceValue;  // current value showing on the die

   //-----------------------------------------------------------------
   //  Constructor: Sets the initial face value.
   //-----------------------------------------------------------------
   public Die()
   {
      faceValue = 1;
   }
```

continued on next slide

```java
//-----------------------------------------------------------------
//  Rolls the die and returns the result.
//-----------------------------------------------------------------
public int roll()
{
   faceValue = (int)(Math.random() * MAX) + 1;
   return faceValue;
}

//-----------------------------------------------------------------
//  Face value mutator.
//-----------------------------------------------------------------
public void setFaceValue (int value)
{
   faceValue = value;
}

//-----------------------------------------------------------------
//  Face value accessor.
//-----------------------------------------------------------------
public int getFaceValue()
{
   return faceValue;
}
```

continued on next slide

continue

```
    //------------------------------------------------------------------
    //  Returns a string representation of this die.
    //------------------------------------------------------------------
    public String toString()
    {
       String result = Integer.toString(faceValue);

       return result;
    }
}
```

The `Die` Class (contd.)

- The `Die` class contains two fields (attributes or variables):
  - constant `MAX` that represents the maximum face value
  - integer `faceValue` that represents the current face value
- The `roll()` method uses the `random()` method of the `Math` class to determine a new face value.
  - We could also have used the `Random` class (which we saw earlier)
- There are also methods to explicitly set and retrieve the current face value at any time

The `Die` Class

(Modifying the code shown in the book)

- The initial value should be randomly set and not to 1 – in the constructor
- `Math.random()` generates a `double` value instead of an `int`.
  - The value returned needs to be converted to an integer.
- We will use the `Random` class to generate integer random numbers in the constructor and in the roll() method
- We will use the `String.format()` method in `toString()` to convert the integer `faceValue` to a `String` – similar to the `printf()` method.
- Do not need the extra variable `result` in the original code.

The `Die` Class (contd.)

```
import java.util.*;
…
private Random generator;
…

public Die() {
   generator = new Random();
   faceValue = generator.nextInt(6) + 1;
}
public int roll() {
   return faceValue = generator.nextInt(6) + 1;
}
…
public String toString() {
    return String.format("%d", faceValue);
}
```

Class `Point` – Another Example

- We define a class `Point` for storing 2-D points using Cartesian coordinates (i.e., x and y)

Class Point (contd.)

```
public class Point {
   private double x, y;
   public Point() {
      x = 0; y = 0;
   }
   public Point(double a, double  b) {
      x = a; y = b;
   }

   public String toString(){
      return "x = " + x + ", y = " + y;
   }
}
```

Class Point (contd.)

- Class `Point` has
  - 2 constructors
  - 1 method `toString()` that will be used by Java to automatically convert `Point` values to strings
    - Uses the default `double` → `String` conversion

- `Point` object creation examples:

```
Point p1 = new Point();
Point p2;
p2 = new Point(1, 5);
```

Class Point (contd.)

- Printing `Point` values

```
System.out.printf("%s\n", p2);
```

The above will print

```
x = 1.0, y = 5.0
```

- What methods is class `Point` missing?

What is Class `Point` Missing

- Methods to set and get `x` and `y` values
- Methods to update `x` and `y` values
- Utility methods, for example, a method to compute the distance between two points

Using Class Point

```
import java.util.*;  // for class Scanner
public class PointDemo {
   public static void  main(String[] args) {
      Scanner stdin = new Scanner(System.in);
      Point p1, p2; double x, y;

      System.out.print("Enter point 1 - x & y coordinates: ");
      x = stdin.nextDouble();
      y = stdin.nextDouble();
      p1 = new Point(x, y);

      System.out.print("Enter point 2 - x & y coordinates: ");
      x = stdin.nextDouble();
      y = stdin.nextDouble();
      p2 = new Point (x, y);

      System.out.println("Point p1: " + p1);
      System.out.println("Point p2: " + p2);
   }
}
```

Class `Point` (contd.)

Example Input/Output

```
Enter point 1 - x & y coordinates:  3 4
Enter point 2 - x & y coordinates: -11 33.99
Point p1: x = 3.0, y = 4.0
Point p2: x = -11.0, y = 33.99
```

Extending Class `Point` & Printing `double` Values

- We will add methods to set and retrieve `x` & `y` coordinates
- We will add a method to compute the distance between two points
- We will print the distance in two ways using
  - default `double` to `String` conversion and
  - explicit `double` to `String` conversion.

Extending Class `Point`

```
public class Point {
   private double x, y;
   public Point() { x = 0; y = 0; }
   public Point(double a, double  b) { x = a; y = b;}
   public void setX(double a) { x = a; }
   public void setY(double b) { y = b; }
   public double getX() { return x; }
   public double getY() { return y; }

   public double distance(Point p) {
      return Math.sqrt((x-p.x)*(x-p.x)+(y-p.y)*(y-p.y));
   }

   public String toString(){
      return "x = " + x + ", y = " + y;
   }
}
```

Using the Extended `Point` Class & Printing `double` Values

```
import java.text.*;  // for number format
import java.util.*;  // for class Scanner
public class PointDemo {
   public static void  main(String[] args) {
      Scanner stdin = new Scanner(System.in);

      Point p1, p2; double x, y;

      System.out.print("Enter point 1 - x & y coordinates: ");
      x = stdin.nextDouble(); y = stdin.nextDouble();
      p1 = new Point(x, y);

      System.out.print("Enter point 2 - x & y coordinates: ");
      x = stdin.nextDouble(); y = stdin.nextDouble();
      p2 = new Point (x, y);
```

Using the Extended `Point` Class & Printing `double` Values (contd.)

```
        System.out.println("Point p1: " + p1);
        System.out.println("Point p2: " + p2);

        System.out.println("Distance between points is " +
            p1.distance(p2)+
            " (using default double to string conversion)");

        System.out.printf("%s %.3f %s\n",
             "Distance between points is ",
             p1.distance(p2),
             " (using explicit double to string conversion)");
    }
}
```

Using the Extended `Point` Class & Printing `double` Values (contd.)

```
Enter point 1 - x & y coordinates: 2 3
Enter point 2 - x & y coordinates: 3 4
Point p1: x = 2.0, y = 3.0
Point p2: x = 3.0, y = 4.0
Distance between points is
1.4142135623730951 (using default double to
string conversion)
Distance between points is 1.414 (using
explicit double to string conversion)
```

Printing `double` values

- Default primitive type to `String` conversion looks OK sometimes
- But you may need to use explicit conversions to make the output look appropriate or pretty

Extending Class `Point` Further to Compute the Area of a Triangle

- Define a method that computes the area of a triangle.

  Area of a Triangle = SQRT(s(s-a)(s-b)(s-c))

  where a, b, c are the lengths of the sides and

  s=(a+b+c)/2

  Note: s = perimeter/2

- Do it two ways – first as a `static` method and then as a normal method
  - `public static double area(Point p, Point q, Point r)`
  - `public double area(Point q, Point r)`
- Normal method Illustrates either explicit or implicit use of the special variable `this`.
- Pros and cons of these two versions of `area`?

Roots of a Quadratic Equation Example – Object Version

- We will rewrite our Quadratic Equations program in object-oriented programming style.
- We will define a class `QuadEqn` to represent quadratic equations
- Why?
  – Class `QuadEqn` can be used by others
  – Can put it in a library
  – Each object of class `QuadEqn` can hold a quadratic equation allowing a program to conveniently handle more than one quadratic equation

Quadratic Equation Class

```
// QuadEqn represents the quadratic equation
//    Ax2 + Bx + C = 0
// where a, b, c are constants with A != 0,
// B*B - 4*A*C >= 0, and x is a variable


public class QuadEqn {
   private double a, b, c;

   public QuadEqn(double A, double B, double C) {
      if (A == 0) {
         System.out.println("a cannot be 0!\nExiting");
         System.exit(0);
      }
      if ((B*B - 4*A*C)< 0) {
         System.out.println("B*B-4*A*C < 0!\nExiting");
         System.exit(0);
      }
      a = A; b = B; c = C;
  }
```

Quadratic Equation Class (contd.)

```
 public void setA (double A) {
    if (A == 0) {
      System.out.println("A cannot be 0!\nExiting");
      System.exit(0);
    }
    a = A;
}
public void setB (double B) {b = B;}
public void setC (double C) {c = C;}
public double getA () {return a;}
public double getB () {return b;}
public double getC () {return c;}
```

Quadratic Equation Class (contd.)

```
    public double root1 (){
        if ((b*b - 4*a*c)< 0) {
            System.out.println("b*b-4*a*c < 0!\nExiting");
            System.exit(0);
        }
        return (-b + Math.sqrt(b*b - 4*a*c))/(2*a);
    }
    public double root2 () {
        if ((b*b - 4*a*c)< 0) {
            System.out.println("b*b-4*a*c < 0!\nExiting");
            System.exit(0);
        }
        return (-b - Math.sqrt(b*b - 4*a*c))/(2*a);
    }
}
```

Using Class QuadEqn

```
import java.util.*;  // for class Scanner
public class QuadObjectRoots{
   public static void  main(String[] args) {
      Scanner stdin = new Scanner(System.in);
      System.out.println("Lets compute Roots!\n");
      System.out.println("Enter the 3 constants, a, b, & c");

      double a = stdin.nextDouble();
      double b = stdin.nextDouble();
      double c = stdin.nextDouble();
      QuadEqn q = new QuadEqn(a, b, c);

      // printing using C style print facility
      System.out.printf("Roots of the quadratic equation\n" +
                        "%.2fx*x + %.2fx + %.2f = 0\n" +
                        "x = %.2f and x = %.2f\n",
                        a, b, c, q.root1(), q.root2());
   }
}
```

Private vs. Public Variables

- Why did we declare variables `a`, `b`, and `c` private in `QuadEqn` as

```
private double a, b, c;
```

- Had we declared them as

```
public double a, b, c;
```

then they could have been referenced directly, from outside class `QuadEqn`, e.g.,

```
QuadEqn q = new QuadEqn(5.0, 25.0, 3);
        …
q.a …  q.b … q.c
```

- We would not be able to ensure that the variables are set to the right values.
- For example, we would not be able to ensure that `q.a` is never set to 0 . For example, the user could just write

```
q.a = 0;
```

Quadratic Equation Class – Pluses

- Separates quadratic equation functionality from the program that needs to use it.
- Supports modularization.
- Class `QuadEqn` can be easily used by others.
- Can be tested separately from programs that use it.

Classes – Review

- The values of the attributes (fields in the class) define the state of an object created using the class

- The functionality of the methods (class operations) defines the behavior of the object

- Any given program will probably not use all methods of a given class

Scope of Variables/Fields & Constants (review)

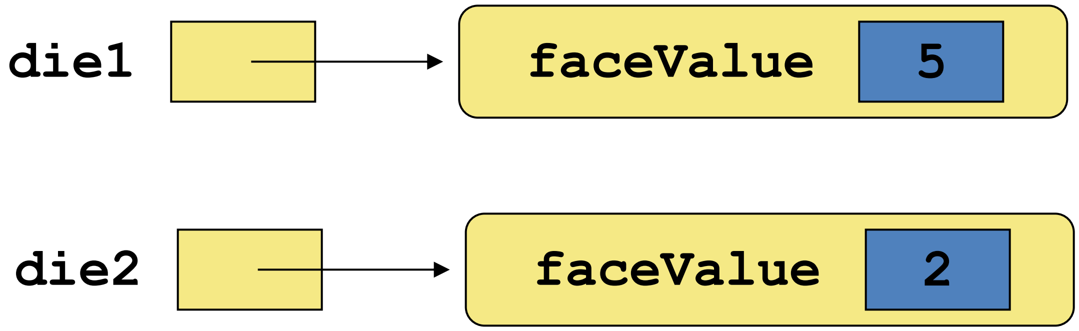
- The *scope* of a data item is the area in a program in which it can be referenced (used)
- Data items declared at the class level can be referenced by all methods in that class
- Data items declared within a method can be used only in that method
- Data items declared within a method are classified as *local data items*
- In the `Die` class (the book version), variable `result` is declared inside the `toString()` method – it is local to that method and cannot be referenced anywhere else

Instance Data Items

- A variable declared at the class level (such as `faceValue`) is also called *instance data item*
- Each instance (object) has its own instance variables
- A class declares the type of the data, but it does not allocate memory for it
- Memory is allocated when the object is created
  - For example, each time a `Die` object is created, a new `faceValue` variable is created as well
- The objects of a class share method definitions and static attributes/variables but each object has its own data space (instance variables)

Instance Data

- We can depict the two `Die` objects as follows:



Each object maintains its own `faceValue` variable, and thus its own state

Two Views of a Class

- **Developer view**
  - **Internal**: the details of the variables and methods of the class that defines it
- **User (Client) View**
  - **External**: the services (methods) that an class provides which defines its behavior

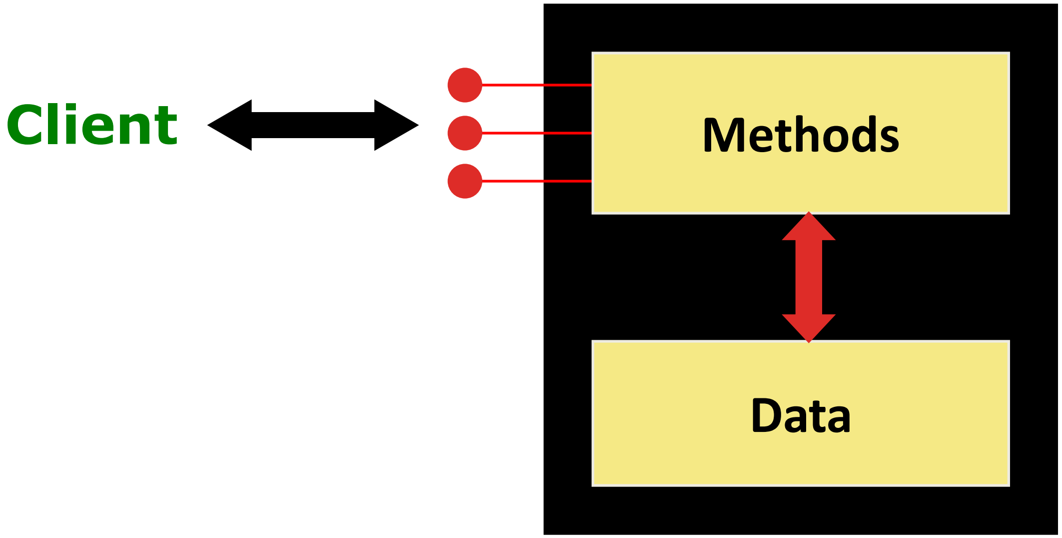
Encapsulation

- An object is an *encapsulated* entity, providing a set of specific services (specified by its class)
  - These services define the object *interface*
- One object (called the *client*) may use another object for the services it provides
- The client object may request services (call its methods), but it need not be aware of how those services are implemented
- Any changes to the object's state (its variables) should be made by the object's methods

Encapsulation (contd.)

- An encapsulated object can be thought of as a *black box* – its inner workings are hidden from the client

- The client (object) invokes the interface methods and they manage the instance data (object state)



Visibility Modifiers (review)

- Visibility modifiers are used to ensure encapsulation
- They are used in declarations to specify characteristics of a field or method:
  - `final`: specifies a constant
  - `static`: specifies a class (not an object) variable / method
  - `public`: can be accessed from everywhere
  - `private`: can be accessed only from within class
  - `protected`: can be accessed only from the class or classes derived from it and the package containing it
  - *no modifier*: can be accessed from the class and the package containing it

Visibility Modifiers (contd.)

- Public variables/fields violate encapsulation because they allow the object user (client) to modify the field values directly
  - Allows them to bypass checks that would be performed when using methods to modify field values
- It is OK to give constants public visibility, which allows them to be referenced directly outside of the class
  - Public constants do not violate encapsulation because, although the user can access them, their values cannot be changed

Visibility Modifiers (contd.)

- Methods that provide services of an object should be declared with public visibility so that they can be invoked by object users (clients).

- Public methods are also called *service methods*

- A method created simply to assist a service method is called a *support method*

- Since a support method is not intended to be called by a client, it should not be declared with public visibility

Visibility Modifiers

	`public`	`private`
Variables	**Violate encapsulation**	**Enforce encapsulation**
Methods	**Provide services to clients**	**Support other methods in the class**

Accessors & Mutators (or Getters & Setters) Methods

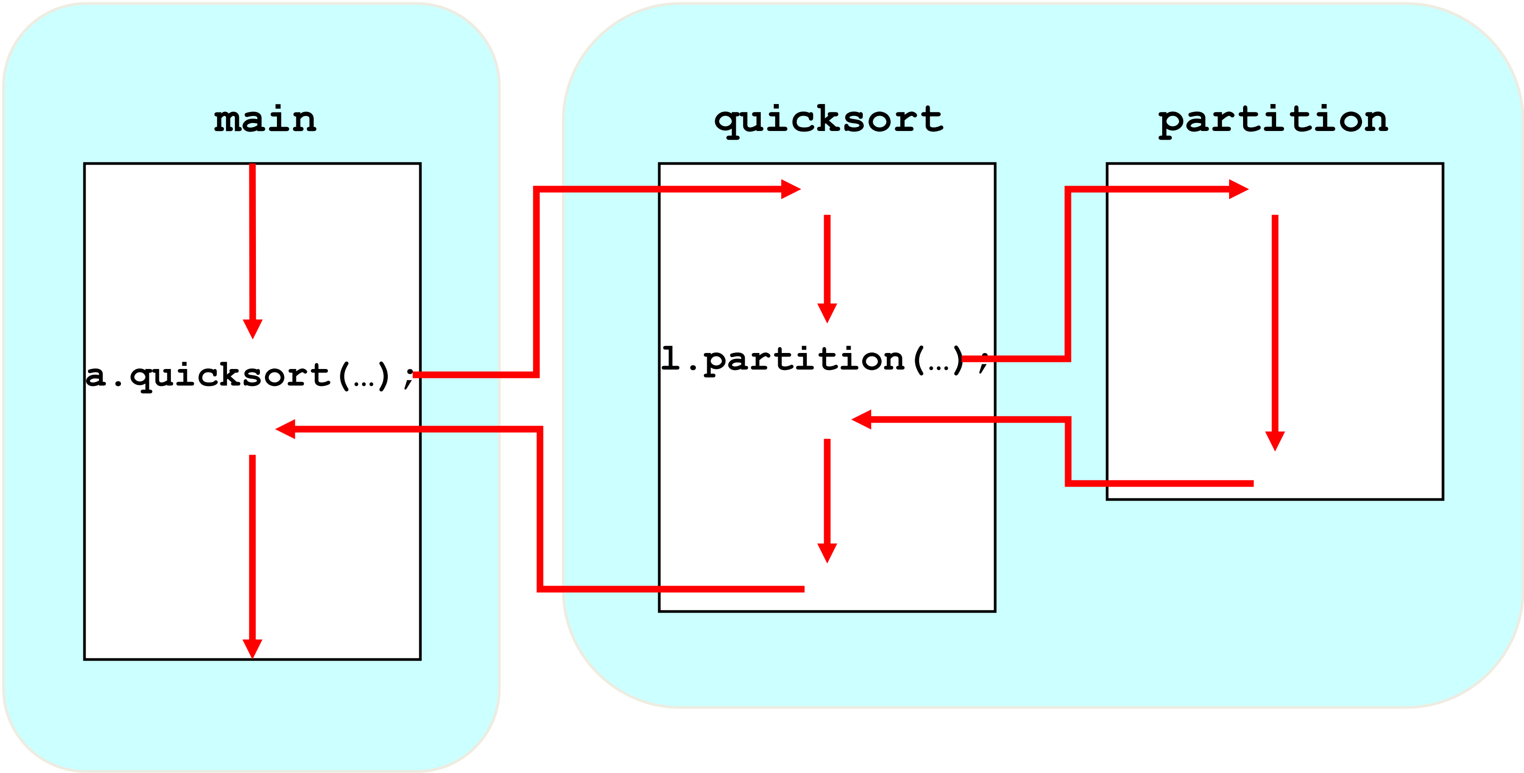
- Because object instance data is private, a class usually provides services to access and modify data values
  - An accessor (getter) method returns the current value of a variable
  - A mutator (setter) method changes the value of a variable
- Accessor and mutator methods typically have the form `getX` and `setX`, respectively, where X is the name of the field (instance variable).

Method Declarations

- A *method declaration* specifies the code that will be executed when the method is invoked (called)
- When a method is invoked, the flow of control jumps to the method and executes its code
- When execution is completed, the flow returns to the place where the method was called and continues
- The invocation may or may not return a value, depending upon how the method is defined

Method Control Flow

- If the called method is in the same class, only the method name is needed



Method Header

- A method declaration begins with a *method header*

```
public char calc(int num1, int num2)
```

visibility modifer

return type

method name

parameter list

The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal parameter*

Method Body

- The method header is followed by the *method body*

```
public char calc(int num1, int num2)
{
    int sum = num1 + num2;
    char result = message.charAt(sum);

    return result;
}
```

The return expression must be consistent with the return type

`sum` **and** `result` **are local variables**

They are created each time the method is called, and are destroyed when it finishes executing

The `return` Statement

- The *return type* of a method indicates the type of value that the method sends back to the caller

- A method that does not return a value has a `void` return type

- A `return` statement specifies the value that will be returned

  `return` *expression*;

- The expression type must conform to the return type

Parameters

- When a method is called, the *actual parameters* in the invocation are copied to the *formal parameters* in the method header.
- *Actual parameters* are also referred to as *arguments*
- *Formal parameters* are also referenced simply as *parameters*
- Values of arguments are simply copied to the parameters
  - Changing values of parameters in the method does not change the value of the arguments
  - Called *passing arguments by value*.

Arguments are Passed by Value

```
public class MISC {
    public static void add1(int x) {
        x = x + 1;
    }

    public static void Add1ToArrayElements(int[] y) {
         for (int i = 0; i < y.length; i++) {
             y[i]++;
         }
    }

}
```

Arguments are Passed by Value (contd.)

```
public class testclass {
    public static void main(String[] args) {
        //passing a primitive type as an argument
        int a = 0;
        System.out.printf("Before Adding 1: a = %d\n", a);
        MISC.add1(a);
        System.out.printf("After adding 1:  a = %d\n", a);

        ...

     }
}
```

RESULT

Before Adding 1: a = 0
After adding 1: a = 0

Arguments are Passed by Value (contd.)

```
public class testclass {
    public static void main(String[] args) {
        ...

        //passing an object type (array) as an argument
        int[] p = {1, 2, 3};
        System.out.printf("\nBefore Adding 1: ");
        for (int i = 0; i < p.length; i++)
            System.out.printf("p[%d] = %d ", i,  p[i]);
        MISC.Add1ToArrayElements(p);
        System.out.printf("\nAfter Adding 1: ");
        for (int i = 0; i < p.length; i++)
            System.out.printf("p[%d] = %d ", i,  p[i]);

    }
}
```

Arguments are Passed by Value (contd.)

```
Result
```

Before Adding 1: p[0] = 1 p[1] = 2 p[2] = 3
After Adding 1: p[0] = 2 p[1] = 3 p[2] = 4

Arguments are Passed by Value (Summary)

- Method parameters are *passed by value* – "from the method call to the method body"
- A copy of the *actual parameter* (*aka* argument) is stored into the *formal parameter* (specified in the method header, *aka* parameter)
  - When an primitive type literal or variable is passed to a method, the argument value is copied to the parameter
  - When an object variable is passed to a method, the arguments and parameters become aliases of each other pointing to the same object
- Depending upon whether a parameter type is of a basic type or of an object type and what a method does with a parameter the value of the argument may or may not change

Local Data

- Local variables are variables declared inside a method
- The formal parameters of a method behave as *local variables*
- When the method finishes, all local variables are destroyed (including the formal parameters)
- **Note**: instance variables (defining the object state) – variables declared at the class level – exist as long as the object exists

Main (Driver) Programs

- A `main` (*driver)* program drives the use of other, perhaps more interesting, parts of a program

- Test drivers are used to test all or parts of the software

Class Bank `Account` – An Example

- We will model a bank's basic financial operations tracking bank accounts
- A bank account will be implemented as a class named `Account`
- Objects of the `Account` type will track individual accounts recording information such as
  - the account number,
  - the current balance, and
  - the name of the owner
- Operations (services) supported include deposits and withdrawals, and adding interest

```java
//********************************************************************
//  Account.java  Author: Lewis/Loftus – toString() modified by Gehani
//  Represents a bank account with basic services such as deposit
//  and withdraw.
//********************************************************************

public class Account
{
   private final double RATE = 0.035;  // interest rate of 3.5%

   private long acctNumber;
   private double balance;
   private String name;

   //-----------------------------------------------------------------
   //  Sets up the account by defining its owner, account number,
   //  and initial balance.
   //-----------------------------------------------------------------
   public Account (String owner, long account, double initial)
   {
      name = owner;
      acctNumber = account;
      balance = initial;
   }
```

continued on next slide

```java
//-----------------------------------------------------------------
//  Deposits the specified amount into the account. Returns the
//  new balance.
//-----------------------------------------------------------------
public double deposit (double amount)
{
   balance = balance + amount;
   return balance;
}

//-----------------------------------------------------------------
//  Withdraws the specified amount from the account and applies
//  the fee. Returns the new balance.
//-----------------------------------------------------------------
public double withdraw (double amount, double fee)
{
   balance = balance - amount - fee;
   return balance;
}
```

continued on next slide

```java
//------------------------------------------------------------------
   //  Adds interest to the account and returns the new balance.
   //------------------------------------------------------------------
   public double addInterest ()
   {
      balance += (balance * RATE);
      return balance;
   }
   //------------------------------------------------------------------
   //  Returns the current balance of the account.
   //------------------------------------------------------------------
   public double getBalance () { return balance; }

   //------------------------------------------------------------------
   //  Returns a one-line description of the account as a string.
   //  Modified from original by Gehani
   //------------------------------------------------------------------
   public String toString ()
   {
       return String.format("Name = %s, Acct Num = %d, Balance = $%.2f",
                              name, acctNumber, balance);
   }
}
```

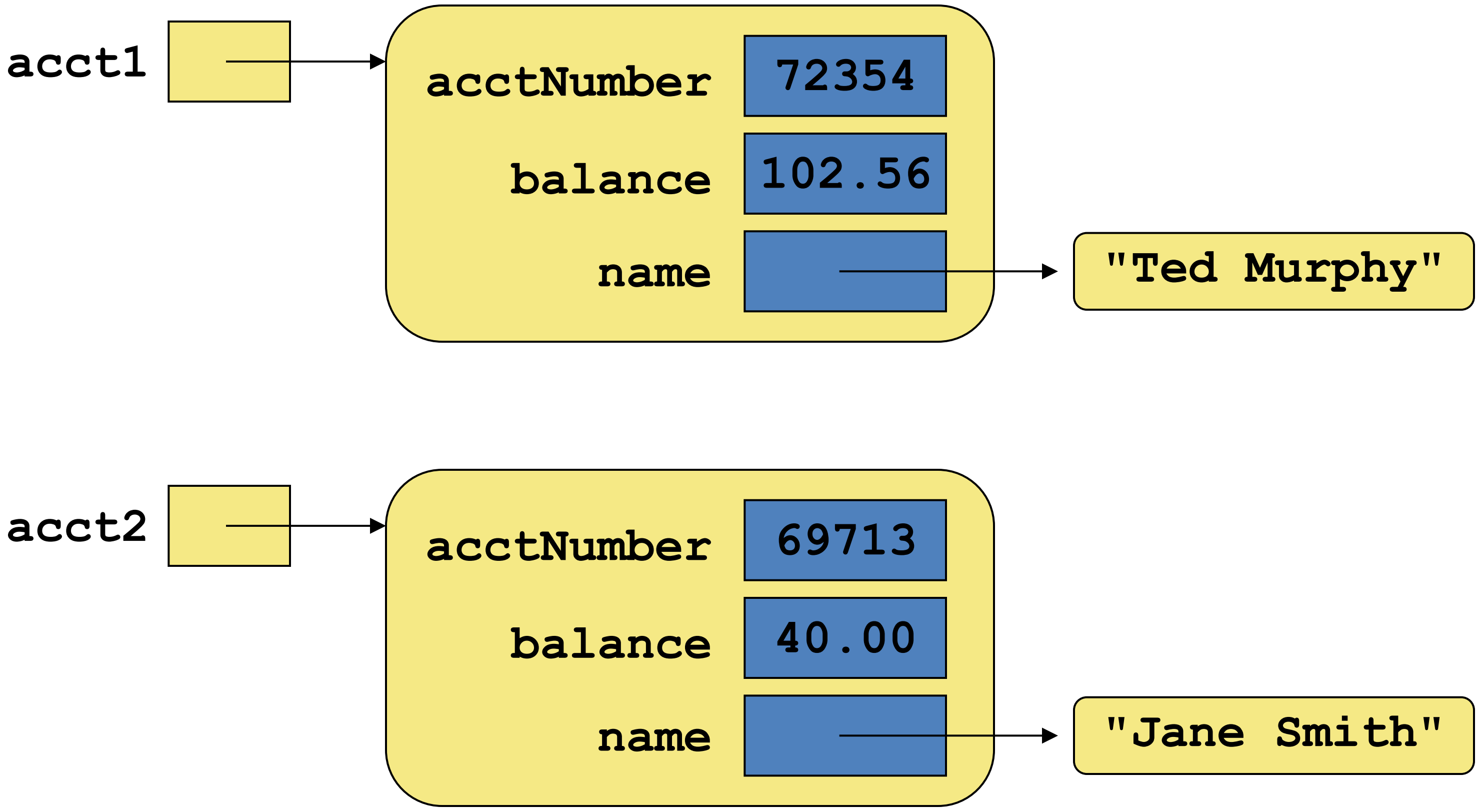
continued on next slide

```java
//********************************************************************
//  Transactions.java       Author: Lewis/Loftus
//
//  Demonstrates the creation and use of multiple Account objects.
//********************************************************************

public class Transactions
{
   //-----------------------------------------------------------------
   //  Creates some bank accounts and requests various services.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Account acct1 = new Account ("Ted Murphy", 72354, 102.56);
      Account acct2 = new Account ("Jane Smith", 69713, 40.00);
      Account acct3 = new Account ("Edward Demsey", 93757, 759.32);

      acct1.deposit (25.85);

      double smithBalance = acct2.deposit (500.00);
      System.out.println ("Smith balance after deposit: " +
                          smithBalance);
```

continued on next slide

```
      System.out.println ("Smith balance after withdrawal: " +
                          acct2.withdraw (430.75, 1.50));

      acct1.addInterest();
      acct2.addInterest();
      acct3.addInterest();

      System.out.println ();
      System.out.println (acct1);
      System.out.println (acct2);
      System.out.println (acct3);
   }
}
```

Bank Account Example



Bank Account Example

- There are some improvements that can be made to the `Account` class
  - Getter and setter methods could have been defined for all data items
  - The design of some methods could also be more robust, such as verifying that the `amount` parameter to the `withdraw()` method is positive
- Other improvements?

Constructors Revisited

- A constructor has no return type specified in the method header, not even `void`
- A constructor does not return a value – it just initializes the object created
- A common error is to put a return type on a constructor, which makes it a "regular" method that happens to have the same name as the class
- The programmer does not have to define a constructor for a class
- If no constructor has not been defined for a class, Java defines a *default constructor* that accepts no parameters

Wrapper Classes

Wrapper Classes

- When working with numbers, we typically use primitive types.
- There are, however, reasons to use objects in place of primitives (necessitated by the fact that Java is an object-oriented programming language.)
- Java provides "wrapper" classes for "wrapping" each primitive in an object thus converting the primitive to an object.
- The wrapping and unwrapping is often done by the Java compiler.
  - If you use a primitive where an object is expected, the compiler boxes the primitive in its wrapper class automatically – and vice versa (unboxes).

Using Wrapper Classes

There are several reasons why one might use a wrapper object rather than a primitive value:

- An argument of a method that expects an object
- To use constants defined in the wrapper class, such as `MIN_VALUE` and `MAX_VALUE`, that provide the upper and lower bounds of the data type.
- To use wrapper class methods for converting values to and from other primitive types, for converting to and from strings, and for converting between number systems (decimal, octal, hexadecimal, binary).
- When using collections or any polymorphic (generic) class / interface (these topics are not covered in this course)

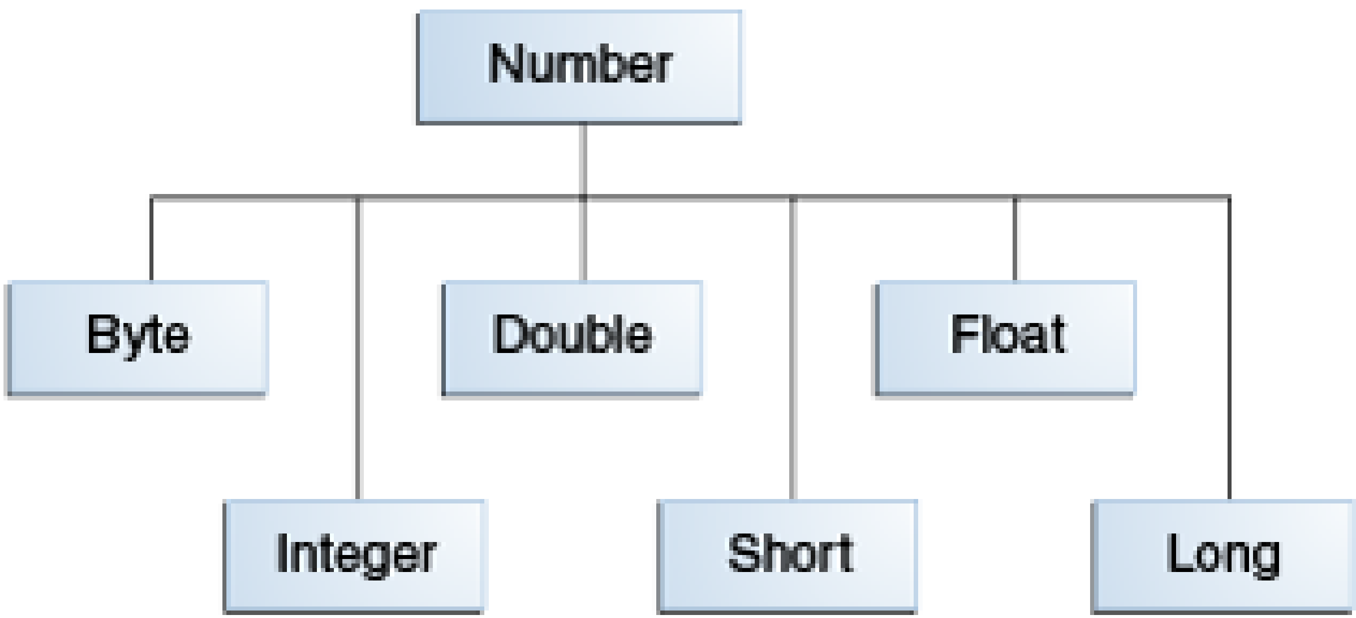
Wrapper Classes (contd.)

Package `java.lang` contains a wrapper class corresponding to each primitive type to allow them to be converted to objects and vice versa

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Wrapper Classes (contd.)

Numeric wrapper classes are subclasses of the abstract class `Number`:



Wrapper Classes – Autoboxing

- *Autoboxing* is the automatic conversion of a primitive value to a corresponding wrapper object:

```
Integer obj;
int num = 42;
obj = num;
```

- The assignment creates the appropriate `Integer` object
- The reverse conversion (called *unboxing*) also occurs automatically as needed

Wrapper Classes (contd.)

- Each wrapper class has `static` methods that enable easy manipulation of primitive type values.
  - For example, the `Integer` class contains a method to convert an integer stored in a `String` to an `int` value:

```
num = Integer.parseInt(str);
```

The `Integer` Wrapper Class

```
Integer (int value)
```
Constructor: creates a new Integer object storing the specified value.

```
byte byteValue ()
double doubleValue ()
float floatValue ()
int intValue ()
long longValue ()
```
Return the value of this Integer as the corresponding primitive type.

```
static int parseInt (String str)
```
Returns the int corresponding to the value stored in the specified string.

```
static String toBinaryString (int num)
static String tohexString (int num)
static String toOctalString (int num)
```
Returns a string representation of the specified integer value in the corresponding base.

FIGURE 3.9 Some methods of the `Integer` class

Packages

- One plus about Java is that it there are many class libraries (sets of classes) – these are not part of the Java but they serve to extend Java.
- The class libraries that come with Java are also known as the Java Application Programming Interface (API)
- We have already used Java API by having used several classes
  - `System`
  - `Scanner`
  - `Math`
  - `String`

Packages (contd.)

Package	Purpose
java.lang	General support
java.applet	Creating applets for the web
java.awt	Graphics and graphical user interfaces
javax.swing	Additional graphics capabilities
java.net	Network communication
java.util	Utilities
javax.xml.parsers	XML document processing

Packages (contd.)

- Packages are imported using the `import` declaration, for example, classes in packages `java.io` and `java.util` are imported as

```
import java.io.*;  // for class File
import java.util.*; // for class Scanner
```

- We could be specific

```
import java.util.Scanner;
```

- Package `java.lang` is a standard package
  - No need to explicitly import it, e.g., for classes `System`, `String`, `Math`, the wrapper classes, …
  - Has classes that are needed for writing Java applications

Creating a Package

- Suppose you want to create a package named `dataStructures`. Then put the statement

```
package dataStructures;
```

as the first line in the files defining the classes, etc. that are to be in the package, e.g., in the files

```
stack.java
queue.java
lists.java
listElements.java
```

- By default, in the absence of a `package` statement, the class ends up in an unnamed package.
- Unnamed packages are typically used only for small applications

Finding +Swapping + Sorting

Finding + Swapping + Sorting

Concept Discussion

- Finding an element in a list
- Swapping two values
- Sorting (selection sort, insertion sort)

Swapping

- Swapping requires three assignment statements and a temporary storage location
- E.g., to swap values of variables `first` & `second`:

```
temp = first;
first = second;
second = temp;
```

Swapping (contd.)

Arguments are Passed by Value (why will this not work?)

```
//... file MISC.java
class MISC {
    public static void swap(int x, int y) {
        int temp;
        temp = x; x = y; y = temp;
    }
}

// ... file test.java
int a = 0, b = 1;
system.out.printf("a = %d, b = %d", a, b);  //??
MISC.swap(a, b);
system.out.printf("a = %d, b = %d", a, b);  // ??
```

More Conditionals

`switch` Statement

- Uses an expression to select from a set of alternatives (each a sequence of statements) for execution.
- The `switch` statement has the form

```
switch(expression) {
case label_1:
  statements_1; break;
case label_2:
  statements_2; break;
  ...
case label_n:
  statements_n; break;
[default:
  statements_default]
}
```

- Default alternative is optional
  - Does not need a `break` statement.

`switch` Statement (contd.)

- The `switch` labels correspond to possible values of the `switch` expression.
  - `switch` expression type must be one of `char`, `byte`, `short`, or `int`.
  - Each label must have the same type as the `switch` expression.
  - The `default` label covers values of the `switch` expression not specified by the other labels.
  - No two labels can have the same value.
- Executing the `break` statement terminates the `switch` statement.
  - the statement following the `switch` statement is executed next.
- The `break` statement is not required by Java which allows control to follow to flow to the next alternative (if any).
  - Good practice to ensure that each alternative terminates with a `break` statement to avoid "flow through" errors – the `default` alternative is an exception.

`switch` Statement Example

```
char opr;
double num = 0, result = 0;
   …
switch(opr) {
case '+':
    result = result + num; break;
case '-':
    result = result - num; break;
case '*':
    result = result * num; break;
case '/':
    result = result / num; break;
case 'c':
    result = num; break;
default:
    System.out.println("Bad Operator!");
}
```

The Conditional Operator

- The *conditional* operator has the form:

  condition ? $\text{expression}_{\text{true}}$: $\text{expression}_{\text{false}}$

- If ***condition* is true**, the value of the conditional expression will be ***expression**$_{\textit{true}}$*

- If ***condition* is false**, the value of the conditional expression will be ***expression**$_{\textit{false}}$*

The Conditional Operator (contd.)

- The conditional operator is similar to an `if` statement, except that it is an expression

- For example, we can rewrite the `if` statement

```
if (a > b)
    max = a;
else
    max = b;
```

using the conditional operator as

```
max = ((a > b) ? a : b);
```

Recursion

Recursion

- Occurs when
  - a method calls itself for additional computation
  - method A calls method B which calls method A, …
- Like infinite loops we can have infinite recursion

Factorial – An Example of Recursion

$$\text{Factorial}(0) = 1$$

$$\text{Factorial}(n) = 1 \times 2 \times \ldots \times n$$

or

$$\text{Factorial}(n) = n \times \text{Factorial}(n-1)$$

Factorial
Iterative & Recursive Versions

- We will implement factorial both
  - iteratively (using loops) and
  - using recursion

Factorial (contd.)

```
// Factorial series -- recursive & iterative choices
// factorial of 0: 0! == 1
// factorial of n >= 1: n! == 1 x 2 x 3 x ... x n-1 x n
//                         ==    n * n-1!

// f(1) = 0; f(2) = (1), f[n] = f[n-2] + f[n-1}

public class Factorial {

  public static int factRecursive(int n) {
      if (n <0 ) {
         System.err.println("Error: Factorial" +
                                   " number must be >=0");
         System.exit(0);
      }
      switch(n) {
         case 0:  return 1;
         default: return n * factRecursive(n-1);
      }
   }
```

Factorial (contd.)

```
public static int factIterative(int n) {
  if (n <0 ) {
    System.err.println("Error: " +
           "Factorial number must be >0");
    System.exit(0);
  }

  int fact = 1;
  for (int i = 1; i <= n; i++)
    fact = fact * i;
  return fact;
 }
}
```

Factorial – Main Program

```
// use: FactorialDemo n (n is the factorial number)
//  or  FactorialDemo (will ask for n)
import java.util.*;  // for class Scanner
public class FactorialDemo {
   public static void  main (String[] args) {
      int n;
      if (args.length == 1) { // Factorial num in cmd-line?
         n = Integer.parseInt(args[0]);
      }
      else {  // ask for number of disks
         Scanner scan = new Scanner(System.in);
         System.out.print("Enter Factorial Number: ");
         n = scan.nextInt();
      }
      System.out.println("Factorial Number " + n +
         " recursively computed = " +
         Factorial.factRecursive(n));
      System.out.println("Factorial Number " + n +
         " iteratively computed = " +
         Factorial.factIterative(n));
   }
}
```

No `break` Statement Used in the `switch` Statement

```
switch(n) {
case 0:  return 1;
default: return n*factRecursive(n-1);
}
```

Infinite Recursion

- If we removed the check that ensures that factorial is not computed with a negative number, we will have infinite recursion

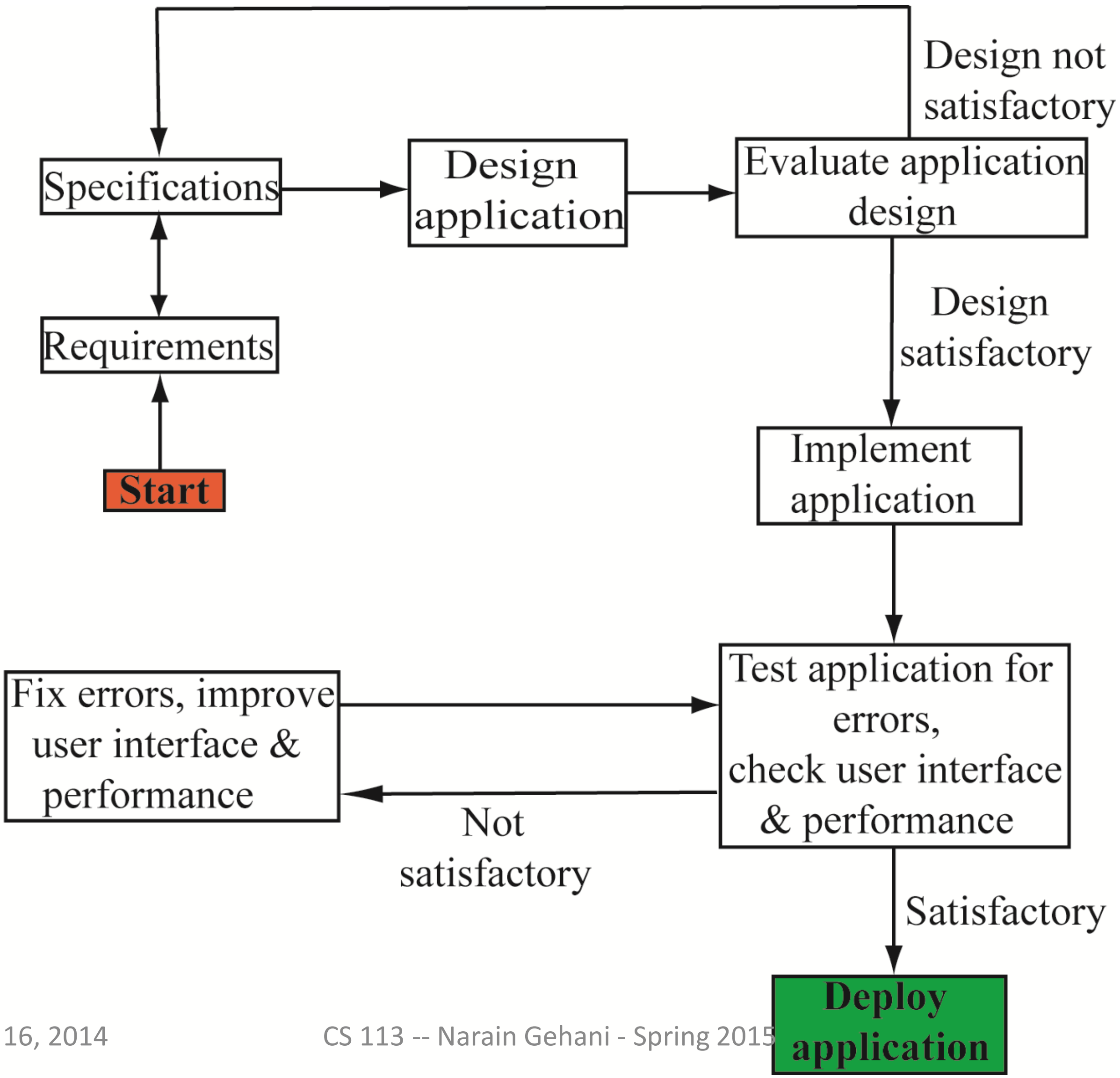
Object-Oriented Design

Software Development (revisiting)

- Requirements (what customer wants)
- Specifications (what the software should do)
- Design (object-oriented) (how the software is organized and how it should work)
- Implementation (converting design to code)
- Testing & Debugging
- Production

Note: In the book, requirements & specifications are considered to be one activity

Software Design & Implementation



Requirements

- *Software requirements* is what you get / extract from the customer.

- The customer may or may not know what is
  - wanted,
  - needed,
  - appropriate, or
  - cost effective

- May require several interactions with the customer to figure our an appropriate set of requirements

- Figuring out the customer needs will lead to a happy customer.

Specification

- Software requirements are likely to be informal and vague.
- These must be converted to clear and precise specifications which an be given to a programmer
- The specifications specify
  - what is to be done by the software
  - **NOT** how it is to be done.

Design

- A *software design* specifies how a program will accomplish the specifications
- A software design specifies how the solution can be broken down into manageable pieces and what each piece will do
- An object-oriented design determines which classes and objects are needed, and specifies how they will interact
- Low-level design details include how individual methods will accomplish their tasks

Implementation

- *Implementation* is the process of translating a design into source code
- Novice programmers often think that writing code is the heart of software development, but actually it should be the least creative step
- Almost all important decisions are made during requirements and design stages
- Implementation should focus on coding details, including style guidelines and documentation

Object-Oriented Desgin

- The core activity of object-oriented design is determining the classes (object types) that will make up the solution
- The classes may be part of a class library, reused from a previous project, or newly written
- One way to identify potential classes is to identify the objects discussed in the requirements
- Objects are generally nouns, and the services that an object provides are generally verbs

Identifying Classes

- Our calculator example is too simple for specifying a set of classes
- Let us consider a university management system
  - what classes will be need
  - are the classes related to each other?

Identifying Classes & Objects

- Sometimes it is challenging to decide whether or not something should be represented as a class
- For example, should an employee's address be represented as a set of instance variables or as an object of class `Address`
- The more you examine the problem and its details the more clear these issues become
- When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities

Identifying Classes and Objects (contd.)

- We want to define classes with the proper amount of detail
- For example, it may be unnecessary to create separate classes for each type of appliance in a house
- It may be sufficient to define a more general `Appliance` class with appropriate instance data
- It all depends on the details of the problem being solved

Identifying Classes and Objects (contd.)

- Part of identifying the classes we need is the process of *assigning responsibilities* to each class
- Every activity that a program must accomplish must be represented by one or more methods in one or more classes
- In early stages it is not necessary to determine every method of every class – begin with primary responsibilities and evolve the design

Code Reviews, Testing, & Debugging

Code Reviews

- A *code review* is a meeting in which several people examine a design document or section of code
- It is a common and effective form of human-based testing
- Presenting a design or code to others
  - makes us think more carefully about it
  - provides an outside perspective
- Code reviews are sometimes called *code inspections* or *code walkthroughs*

Testing

- The goal of testing is to find errors
- As we find and fix errors, it increases our confidence in the correctness of the program.
- We can never really be sure that all errors have been eliminated

Testing cannot show the absence of errors, only their presence!

- So when do we stop testing?
  - Conceptual answer: Never
  - Cynical answer: When we run out of time
  - Better answer: When we are willing to risk that an undiscovered error still exists

Test Cases

- A *test case* is a set of inputs coupled with the expected results
- Often test cases are organized formally into *test suites* which are stored and reused as needed
- For medium and large systems, testing must be a carefully managed process
- Many organizations have a separate Quality Assurance (QA) department to lead testing efforts

Defect and Regression Testing

- *Defect testing* is the execution of test cases to uncover errors
- The act of fixing an error may introduce new errors
- After fixing a set of errors we should perform *regression testing* – running previous test suites to ensure new errors haven't been introduced
- It is not possible to create test cases for all possible input and user actions
- Therefore we should design tests to maximize their ability to find problems

Black-Box Testing

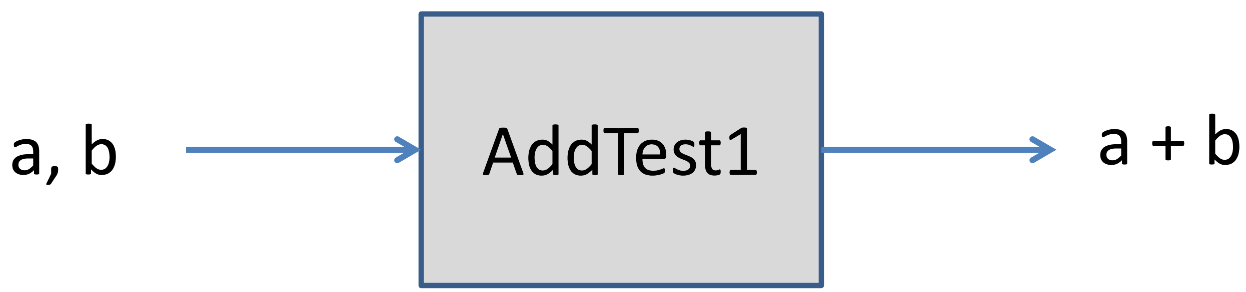
- In *black-box testing*, test cases are developed without considering the internal logic (code of a program)
  – Test cases are based on the input and expected output
- Input can be organized into *equivalence categories*
  – Two input values in the same equivalence category would produce similar results
- A good test suite will cover all equivalence categories and focus on the boundaries between categories

White-Box Testing

- *White-box testing* focuses on the internal structure of the code
- The goal is to ensure that every path through the code is tested
- Paths through the code are governed by any conditional or looping statements in a program
- A good testing effort will include both black-box and white-box tests

AddTest1.java – Simple Adder

- Adds two numbers
- How do we check if it is correct?



AddTest1.java – Simple Adder

```
import java.util.*;  // for class Scanner
public class AddTest1{
   public static void  main(String[] args) {
      Scanner stdin = new Scanner(System.in);
      System.out.println("Ready to ADD 2 numbers!\n" +
                               "Enter Control-Z when done!");
      double a = 0, b = 0;
      while(true) {
         if (stdin.hasNext()) a = stdin.nextDouble();
         else System.exit(0);

         if (stdin.hasNext()) b = stdin.nextDouble();
         else System.exit(0);

         double result  = a + b;
         System.out.printf("%.2f + %.2f = %.2f\n",
                                  a, b, result);
      }
   }
}
```

Testing (contd.)

- Let us introduce a bug in the code
- Whenever the first input is between 300 and 310, the result will be 1 less than it should be

AddTest2.java – Flawed Adder

```
import java.util.*;  // for class Scanner
public class AddTest2{
   public static void  main(String[] args) {
      Scanner stdin = new Scanner(System.in);
     System.out.println("Ready to ADD 2 numbers!\n" +
                        "Enter Control-Z when done!");
      double a = 0, b = 0;
      while(true) {
         if (stdin.hasNext()) a = stdin.nextDouble();
         else System.exit(0);

         if (stdin.hasNext()) b = stdin.nextDouble();
         else System.exit(0);

         double result = a+b;
         if ((a >= 300) && (a <= 310)) result--;
                                    // flawed!!!!

         System.out.printf("%.2f + %.2f = %.2f\n",
                                   a, b, result);
      }
   }
}
```

Testing (contd.)

- Not likely that we will find this bug by black box testing.
- We need to be able to see the code – white box testing

Debugging

- *Debugging* is the process of determining the cause of a problem and fixing it
- Use a debugger which lets you control program execution
- Use print statements

Debugging in jGRASP

- Nice Debugger
- Specify breakpoints – left most part of window when editing a Java program – thin grey strip – mouse turns a red dot – click to set break point
- Breakpoints can be set only on executable statements
- Build each file in Debug mode, and then Run in debug mode
- Program will stop at each break point. You can see variable values
- You can step line-by-line or back to the breakpoints

jGrasp Debugger (recitation topic)

- Tutorial at

  jgrasp.org/tutorials187/06_Debugger.pdf

More about Classes & Objects

Class Definitions – `static` Modifier

- Methods and variables are typically associated with class instances, i.e., objects.
- But they can be associated with a class itself
- If a class has all static methods and static constants, then it is being used for "packaging" them together in one place.

Class Data Items

- Each object has its own instance variables
- However, class variables are shared by all objects.
  - They are declared as `static` variables
- Suppose, for example, we want to count the number of instances of class `Player` (or accounts as another example) that are created.
- This count will be represented as
  - class data
  - not object data

Static Variables
Counting Number of Players Example

Static variable `nPlayers` tracks data that cuts across all instances, i.e., it is associated with the class `Player`:

```
public class Player {
   static int nPlayers = 0;

   // … more items

   public Player() { nPlayers++; }
   public static int numberOfPlayers()
                 { return nPlayers; }
   public void quit() { nPlayers--; }

   //more methods
}
```

Static Variables (contd.)

- Each time a `Player` object is allocated, for example, as in

```
Player p;
…
p = new Player();
```

`nPlayers` is incremented by the class `Player` constructor tracking the number of players created .

Note: `nPlayers` must be initialized in the declaration! Why?

Static Variables (contd.)

- Static variables are typically initialized in their declarations.
- Initializing static variables within constructors is not appropriate because the static variables
  - will be reinitialized every time an object is allocated
  - unless they are initialized conditionally.

Static Methods

- Static methods are associated with a class.
- They are specified using the `static` modifier in their declarations.
- They are invoked with the class name, without the need for creating an instance of the class, for example

```
Player.numberOfPlayers()
```

- Static methods cannot reference instance data.
- **A simple rule** for specifying a method to be `static` is that the method does not depend upon the object state as in case of method `numberOfPlayers()`.

Class Relationships

- Classes in a software system can have various types of relationships to each other

- Three of the most common relationships:
  - Dependency: A *uses* B
  - Aggregation: A *has-a* B
  - Inheritance: A *is-a* B

- We will discuss inheritance in detail in later

Rational Number Class – An Example

- A rational number is a value that can be represented as the ratio of two integers
- The following example defines a class called `RationalNumber`
- Several methods of the `RationalNumber` class accept another `RationalNumber` object as a parameter – **`RationalNumber` depends upon itself** (An illustration of dependency)

```
//********************************************************************
//  RationalTester.java       Author: Lewis/Loftus
//
//  Driver to exercise the use of multiple Rational objects.
//********************************************************************

public class RationalTester
{
   //-----------------------------------------------------------------
   //  Creates some rational number objects and performs various
   //  operations on them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      RationalNumber r1 = new RationalNumber (6, 8);
      RationalNumber r2 = new RationalNumber (1, 3);
      RationalNumber r3, r4, r5, r6, r7;

      System.out.println ("First rational number: " + r1);
      System.out.println ("Second rational number: " + r2);
```

continued on next slide

```java
        if (r1.isLike(r2))
           System.out.println ("r1 and r2 are equal.");
        else
           System.out.println ("r1 and r2 are NOT equal.");

        r3 = r1.reciprocal();
        System.out.println ("The reciprocal of r1 is: " + r3);

        r4 = r1.add(r2);
        r5 = r1.subtract(r2);
        r6 = r1.multiply(r2);
        r7 = r1.divide(r2);

        System.out.println ("r1 + r2: " + r4);
        System.out.println ("r1 - r2: " + r5);
        System.out.println ("r1 * r2: " + r6);
        System.out.println ("r1 / r2: " + r7);
     }
  }
```

Output

```
First rational number: 3/4
Second rational number: 1/3
r1 and r2 are NOT equal.
The reciprocal of r1 is: 4/
r1 + r2: 13/12
r1 - r2: 5/12
r1 * r2: 1/4
r1 / r2: 9/4
```

```java
//********************************************************************
//  RationalNumber.java       Author: Lewis/Loftus
//
//  Represents one rational number with a numerator and denominator.
//********************************************************************

public class RationalNumber
{
   private int numerator, denominator;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the rational number by ensuring a nonzero
   //  denominator and making only the numerator signed.
   //-----------------------------------------------------------------
   public RationalNumber (int numer, int denom)
   {
      if (denom == 0)
         denom = 1;

      // Make the numerator "store" the sign
      if (denom < 0)
      {
         numer = numer * -1;
         denom = denom * -1;
      }
```

continued on next slide

```java
        numerator = numer;
        denominator = denom;

        reduce();
    }

    //-----------------------------------------------------------------
    //  Returns the numerator of this rational number.
    //-----------------------------------------------------------------
    public int getNumerator ()
    {
        return numerator;
    }

    //-----------------------------------------------------------------
    //  Returns the denominator of this rational number.
    //-----------------------------------------------------------------
    public int getDenominator ()
    {
        return denominator;
    }
```

continued on next slide

```java
//-----------------------------------------------------------------
//  Returns the reciprocal of this rational number.
//-----------------------------------------------------------------
public RationalNumber reciprocal ()
{
   return new RationalNumber (denominator, numerator);
}

//-----------------------------------------------------------------
//  Adds this rational number to the one passed as a parameter.
//  A common denominator is found by multiplying the individual
//  denominators.
//-----------------------------------------------------------------
public RationalNumber add (RationalNumber op2)
{
   int commonDenominator = denominator * op2.getDenominator();
   int numerator1 = numerator * op2.getDenominator();
   int numerator2 = op2.getNumerator() * denominator;
   int sum = numerator1 + numerator2;

   return new RationalNumber (sum, commonDenominator);
}
```

continued on next slide

```java
//-----------------------------------------------------------------
//  Subtracts the rational number passed as a parameter from this
//  rational number.
//-----------------------------------------------------------------
public RationalNumber subtract (RationalNumber op2)
{
   int commonDenominator = denominator * op2.getDenominator();
   int numerator1 = numerator * op2.getDenominator();
   int numerator2 = op2.getNumerator() * denominator;
   int difference = numerator1 - numerator2;

   return new RationalNumber (difference, commonDenominator);
}

//-----------------------------------------------------------------
//  Multiplies this rational number by the one passed as a
//  parameter.
//-----------------------------------------------------------------
public RationalNumber multiply (RationalNumber op2)
{
   int numer = numerator * op2.getNumerator();
   int denom = denominator * op2.getDenominator();

   return new RationalNumber (numer, denom);
}
```

continued on next slide

```java
//-----------------------------------------------------------------
//  Divides this rational number by the one passed as a parameter
//  by multiplying by the reciprocal of the second rational.
//-----------------------------------------------------------------
public RationalNumber divide (RationalNumber op2)
{
   return multiply (op2.reciprocal());
}

//-----------------------------------------------------------------
//  Determines if this rational number is equal to the one passed
//  as a parameter. Assumes they are both reduced.
//-----------------------------------------------------------------
public boolean isLike (RationalNumber op2)
{
   return ( numerator == op2.getNumerator() &&
            denominator == op2.getDenominator() );
}
```

continued on next slide

```java
//--------------------------------------------------------------------
//  Returns this rational number as a string.
//--------------------------------------------------------------------
public String toString ()
{
   String result;
   if (numerator == 0)
      result = "0";
   else
      if (denominator == 1)
         result = numerator + "";
      else
         result = numerator + "/" + denominator;
   return result;
}
```

continued on next slide

```java
//--------------------------------------------------------------------
//  Reduces this rational number by dividing both the numerator
//  and the denominator by their greatest common divisor.
//--------------------------------------------------------------------
private void reduce ()
{
   if (numerator != 0)
   {
      int common = gcd (Math.abs(numerator), denominator);

      numerator = numerator / common;
      denominator = denominator / common;
   }
}
```

continued on next slide

continue

```
   //-----------------------------------------------------------------
   //  Computes and returns the greatest common divisor of the two
   //  positive parameters. Uses Euclid's algorithm.
   //-----------------------------------------------------------------
   private int gcd (int num1, int num2)
   {
      while (num1 != num2)
         if (num1 > num2)
            num1 = num1 - num2;
         else
            num2 = num2 - num1;

      return num1;
   }
}
```

GCD – Euclids' Algorithm

- Let x and y be positive numbers
- if x == y

  GCD(x, y) == GCD(x, x) == x
- if x > y

  GCD(x, y) == GCD(x-y, y)
- if x < y

  GCD(x, y) == GCD(x, y-x)

Aggregation

- In the following example, a `Student` object is composed, in part, of `Address` objects
- A student has an address (in fact each student has two addresses)

```java
//********************************************************************
//  Student.java       Author: Lewis/Loftus
//
//  Represents a college student.
//********************************************************************

public class Student
{
   private String firstName, lastName;
   private Address homeAddress, schoolAddress;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this student with the specified values.
   //-----------------------------------------------------------------
   public Student (String first, String last, Address home,
                   Address school)
   {
      firstName = first;
      lastName = last;
      homeAddress = home;
      schoolAddress = school;
   }
```

continued on next slide

```
   //-----------------------------------------------------------------
   //  Returns a string description of this Student object.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result;

      result = firstName + " " + lastName + "\n";
      result += "Home Address:\n" + homeAddress + "\n";
      result += "School Address:\n" + schoolAddress;

      return result;
   }
}
```

```java
//********************************************************************
//  Address.java       Author: Lewis/Loftus
//
//  Represents a street address.
//********************************************************************

public class Address
{
   private String streetAddress, city, state;
   private long zipCode;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this address with the specified data.
   //-----------------------------------------------------------------
   public Address (String street, String town, String st, long zip)
   {
      streetAddress = street;
      city = town;
      state = st;
      zipCode = zip;
   }
```

continued on next slide

```
    //-----------------------------------------------------------------
    //  Returns a description of this Address object.
    //-----------------------------------------------------------------
    public String toString()
    {
       String result;

       result = streetAddress + "\n";
       result += city + ", " + state + "  " + zipCode;

       return result;
    }
}
```

```java
//********************************************************************
//  StudentBody.java       Author: Lewis/Loftus
//  Demonstrates the use of an aggregate class.
//********************************************************************
public class StudentBody
{
   //-----------------------------------------------------------------
   //  Creates some Address and Student objects and prints them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Address school = new Address ("800 Lancaster Ave.", "Villanova",
                                    "PA", 19085);
      Address jHome = new Address ("21 Jump Street", "Lynchburg",
                                   "VA", 24551);
      Student john = new Student ("John", "Smith", jHome, school);

      Address mHome = new Address ("123 Main Street", "Euclid", "OH",
                                   44132);
      Student marsha = new Student ("Marsha", "Jones", mHome, school);

      System.out.println (john);
      System.out.println ();
      System.out.println (marsha);
   }
}
```

```
//******************                        **********************
//  StudentBody.java
//  Demonstrates the
//******************                        **********************
public class StudentB
{
   //----------------                       ----------------------
   //  Creates some A                       and prints them.
   //----------------                       ----------------------
   public static void
   {
      Address school                        er Ave.", "Villanova",
                                            ;
      Address jHome =                       et", "Lynchburg",

      Student john =                        ", jHome, school);

      Address mHome =                       eet", "Euclid", "OH",

      Student marsha                        ones", mHome, school);

      System.out.println (john);
      System.out.println ();
      System.out.println (marsha);
   }
}
```

Output

```
John Smith
Home Address:
21 Jump Street
Lynchburg, VA  24551
School Address:
800 Lancaster Ave.
Villanova, PA  19085

Marsha Jones
Home Address:
123 Main Street
Euclid, OH  44132
School Address:
800 Lancaster Ave.
Villanova, PA  19085
```

The Special Variable `this`

Class `Point` (defined earlier)

```
public class Point {
   private double x, y;
   public Point() { x = 0; y = 0; }
   public Point(double a, double  b) { x = a; y = b;}
   public void setX(double a) { x = a; }
   public void setY(double b) { y = b; }
   public double getX() { return x; }
   public double getY() { return y; }

   public double distance(Point p) {
      return Math.sqrt((x-p.x)*(x-p.x)+(y-p.y)*(y-p.y));
   }

   public String toString(){
      return "x = " + x + ", y = " + y;
   }
}
```

The Special Variable `this`

- Within a method, the special variable `this` refers to the object invoking the method. Thus method

```
public double getX() { return x; }
```

equivalent to writing it as

```
public double getX() { return this.x; }
```

- In the method call shown below

```
Point P = new Point(3.0, 9.0);
double Px, Py;
…
Px = P.getX();
```

the special variable `this` refers to the object `P`.

The Special Variable `this` (contd.)

Within a method, the special variable `this` refers to the object invoking the method. Thus the following constructor in class `Point`

```
Point(double a, double  b) {
      x = a; y = b;
}
```

could have been written as

```
Point(double a, double  b) {
      this.x = a; this.y = b;
}
```

The Special Variable `this` (contd.)

- The special variable `this` comes in handy when the object associated with the invocation has to be
  - passed as an argument to another method (good perhaps necessary use) or
  - specified explicitly for disambiguation as shown below (weak use).
- Suppose, for example, that the parameters of the `Point` constructor are also named `x` and `y`.
  - Then we could use `this` to disambiguate between the variables and the parameters as follows:

```
Point(double x, double  y) {
    this.x = x; this.y = y;
}
```

Interface Classes

- Interfaces specify requirements for classes that implement it thus guaranteeing functionality
- Classes implementing an interface must define the methods specified in the interface.
- An interface specification is like a class specification but it
  - uses the keyword `interface` and
  - contains only method declarations and no method definitions (bodies).
- An interface cannot be used by itself.
  - It must be implemented by a class
  - Multiple classes can implement the same interface.
- An interface variable can refer to an object of any class that implements the interface.

Interface Location – an Example

- To illustrate interfaces, we will define an `interface` type `Location` that represents requirementsfrom types representing a point on a 2-dimensional plane.
- We will look at two implementations of `Location` and illustrate their use.

Interface Location – an Example

- Interface `Location` specifies
  - methods `x()` and `y()` that yield the `x` and `y` coordinates of a location and
  - method `distance()` that compute sthe distance between two locations.
- Here is its specification:

```
public interface Location {
    double x();
    double y();
    double distance(Location loc);
}
```

Interface Location (contd.)

- A location on a 2-D plane can be represented using
  - Cartesian coordinates (the familiar x and y coordinates) or
  - polar coordinates (the distance of a point from the origin and the angle with respect to the x axis).
- For these two systems, we will define classes `Cartesian` and `Polar` that will implement interface `Location` by defining
  - methods `x()` and `y()`, and
  - method `distance()`

to meet the requirements of interface `Location`.

Class Cartesian

```
class Cartesian implements Location {
   private double x, y;
   public Cartesian(double x, double y) {
      this.x = x; this.y = y;
   }
   public double x() {return x;}
   public double y() {return y;}
   public double distance(Location loc) {
      double dx = x() - loc.x();
      double dy = y() - loc.y();
      return Math.sqrt(dx*dx + dy*dy);
   }
}
```

Note

- `Cartesian` implements the methods `x()`, `y()`, and `distance()` specified by interface `Location`.
- It could implement more methods

Class Polar

```
class polar implements Location {
   private double r, theta;
   public polar(double r, double theta) {
                                   //theta in radians
        this.r = r; this.theta = theta;
  }
   public double r() {return r;}
   public double theta() {return theta;}
   public double x() {return r * Math.cos(theta);}
   public double y() {return r * Math.sin(theta);}
   public double distance(Location p) {
      double dx = x() - p.x();
      double dy = y() - p.y();
      return Math.sqrt(dx*dx + dy*dy);
   }
}
```

Note

- Like `Cartesian`, `Polar` implements the methods `x()`, `y()`, and `distance()`.

Using `Location` Interface

```
Cartesian c1 = new Cartesian(0, 0);
Cartesian c2 = new Cartesian(1, 1);
System.out.printf("Distance = %.2f\n",
                  c1.distance(c2));

Polar p1 = new Polar(0, 0);
Polar p2 = new Polar(1, 0.7854);
System.out.printf("Distance = %.2f\n",
                  p1.distance(p2));
```

- The output of the above code is

```
Distance = 1.41
Distance = 1.00
```

Method Design

Method Design

- Identify object types (classes)
- Identify their behavior
- Model the methods on their behavior
- Let us consider a class `flightReservation` for use by an airline reservation system.
  – What would be some methods?

Method Decomposition

- Methods should be, to the extent possible, small – easy to understand
- Large methods should be decomposed into smaller methods
- A `public` method may call one or more `private` *support* methods to help it accomplish its goal
- Support methods might call other support methods if appropriate

Method Overloading

Method Overloading

- *Method overloading* is the use of a single name for multiple methods of a class.
- For Java to be able to figure out which method is being called, each method must have a unique signature, i.e.,
  - unique combination of number, type, and sequence of formal parameters.
  - the signature does not include the method return type.

Method Overloading

- Java determines which method is being invoked by analyzing the parameters. Consider the two methods of class `MISC`

```
public static double tryMe(int x)
{
    return x + .375;
}
```

```
public static double tryMe(int x, double y)
{
    return x*y;
}
```

Invocation

```
double result = MISC.tryMe(25, 4.32)
```

Method Overloading

- The `println` method in class `System.out` is overloaded as it accepts arguments of different types, e.g.,

```
System.out.println(String s)
System.out.println(int i)
System.out.println(double d)
```

  and so on...

- The following lines invoke different versions of the `println` method (variable `total` is of type `double`):

```
System.out.println("The total is:");
System.out.println(total);
```

More about Arrays

Arrays as Parameters

- An entire array can be passed as a parameter to a method
- As discussed, it is the reference to the array that is passed, making the array argument and the corresponding parameter aliases of each other
- Therefore, changing an array element within the method changes the original element

Arrays of Objects

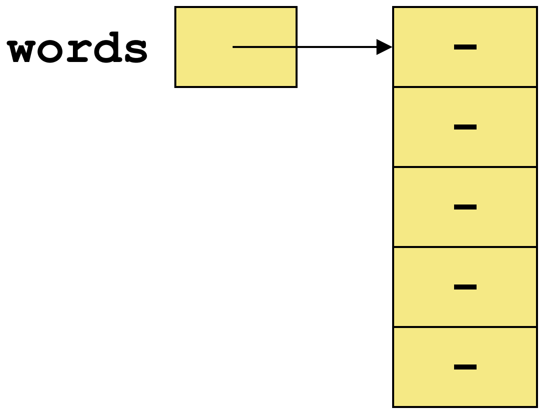
- The elements of an array can be object references
- The following array declaration reserves space to store 5 references to `String` objects

```
String[] words = new String[5];
```

- However, it DOES NOT create the `String` objects themselves
- Initially, an array of objects holds `null` references
- Each object stored in an array must be instantiated separately

Arrays of Objects

- The `words` array when initially declared looks like:



- At this point, the following line of code would throw a `NullPointerException`:

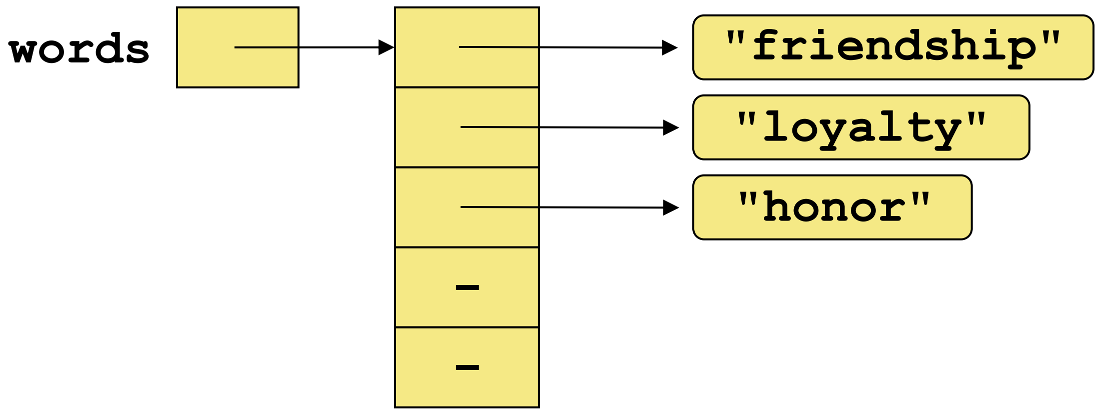
```
System.out.println(words[0]);
```

Arrays of Objects

- After some `String` objects have been stored in the array with the following statements

```
words[0] = "friendship"; words[1] = "loyalty";
words[2] = "honor";
```

the array looks like



Arrays of Objects

- The following declaration creates an array called `verbs` and fills it with five `String` objects created using string literals

```
String[] verbs = {"play", "work", "eat",
                  "sleep", "run"};
```

Arrays of Objects

- The following example creates an array of `Grade` objects, each with a string representation and a numeric lower bound
- The letter grades include plus and minus designations, so must be stored as strings instead of `char`

```java
//********************************************************************
//  Grade.java       Author: Lewis/Loftus
//  Represents a school grade.
//********************************************************************
public class Grade
{
   private String name;
   private int lowerBound;
   //-----------------------------------------------------------------
   //  Constructor: Sets up this Grade object with the specified
   //  grade name and numeric lower bound.
   //-----------------------------------------------------------------
   public Grade (String grade, int cutoff)
   {
      name = grade;
      lowerBound = cutoff;
   }

   //-----------------------------------------------------------------
   //  Returns a string representation of this grade.
   //-----------------------------------------------------------------
   public String toString()
   {
      return name + "\t" + lowerBound;
   }
```

continued on next slide

```java
//--------------------------------------------------------------------
//  Name mutator.
//--------------------------------------------------------------------
public void setName (String grade)
{
   name = grade;
}

//--------------------------------------------------------------------
//  Lower bound mutator.
//--------------------------------------------------------------------
public void setLowerBound (int cutoff)
{
   lowerBound = cutoff;
}
```

continued on next slide

```java
    //-----------------------------------------------------------------
    //  Name accessor.
    //-----------------------------------------------------------------
    public String getName()
    {
       return name;
    }

    //-----------------------------------------------------------------
    //  Lower bound accessor.
    //-----------------------------------------------------------------
    public int getLowerBound()
    {
       return lowerBound;
    }
}
```

```java
//*********************************************************
//  GradeRange.java       Author: Lewis/Loftus
//
//  Demonstrates the use of an array of objects.
//*********************************************************
public class GradeRange
{
   //-------------------------------------------------------
   //  Creates an array of Grade objects and prints them
   //-------------------------------------------------------
   public static void main (String[] args)
   {
      Grade[] grades =
      {
         new Grade("A", 95), new Grade("A-", 90),
         new Grade("B+", 87), new Grade("B", 85), new Grade("B-", 80),
         new Grade("C+", 77), new Grade("C", 75), new Grade("C-", 70),
         new Grade("D+", 67), new Grade("D", 65), new Grade("D-", 60),
         new Grade("F", 0)
      };
// modified by Gehani
      for(int i = 0; i < grades.length; i++)
         System.out.println(grades[i]);
   }
}
```

Output

```
A      95
A-     90
B+     87
B      85
B-     80
C+     77
C      75
C-     70
D+     67
D      65
D-     60
F      0
```

Variable Number of Parameters

- Suppose we want to create a method that accepts a variable number of arguments
- For example, consider method `average` of class `MISC` that returns the average of a varying number of integer parameters

```
// one call to average has 3 arguments
mean1 = MISC.average(42, 69, 37);

// second call to average has 7 arguments
mean2 = MISC.average(35, 43, 93, 23, 40, 21, 75);
```

Variable Number of Parameters (contd.) Possible Options ?

1. We can define overloaded versions of the `average` method
   - Downside: we'd need a separate version of the method for each additional parameter
2. We can define the method to accept an array of integers
   - Downside: we'd have to create the array and store the integers prior to calling the method each time
3. Solution: Use Java's *variable length parameter lists*

Variable Length Parameter Lists

- Using special syntax "..." in the formal parameter list, we can define a method that accept any number of parameters of the same type
- For each call, the parameters are automatically put into an array for easy processing in the method

Indicates a variable length parameter list

```
public double average(int ... list)
{
   ...
}
```

element type

array name

Variable Length Parameter Lists

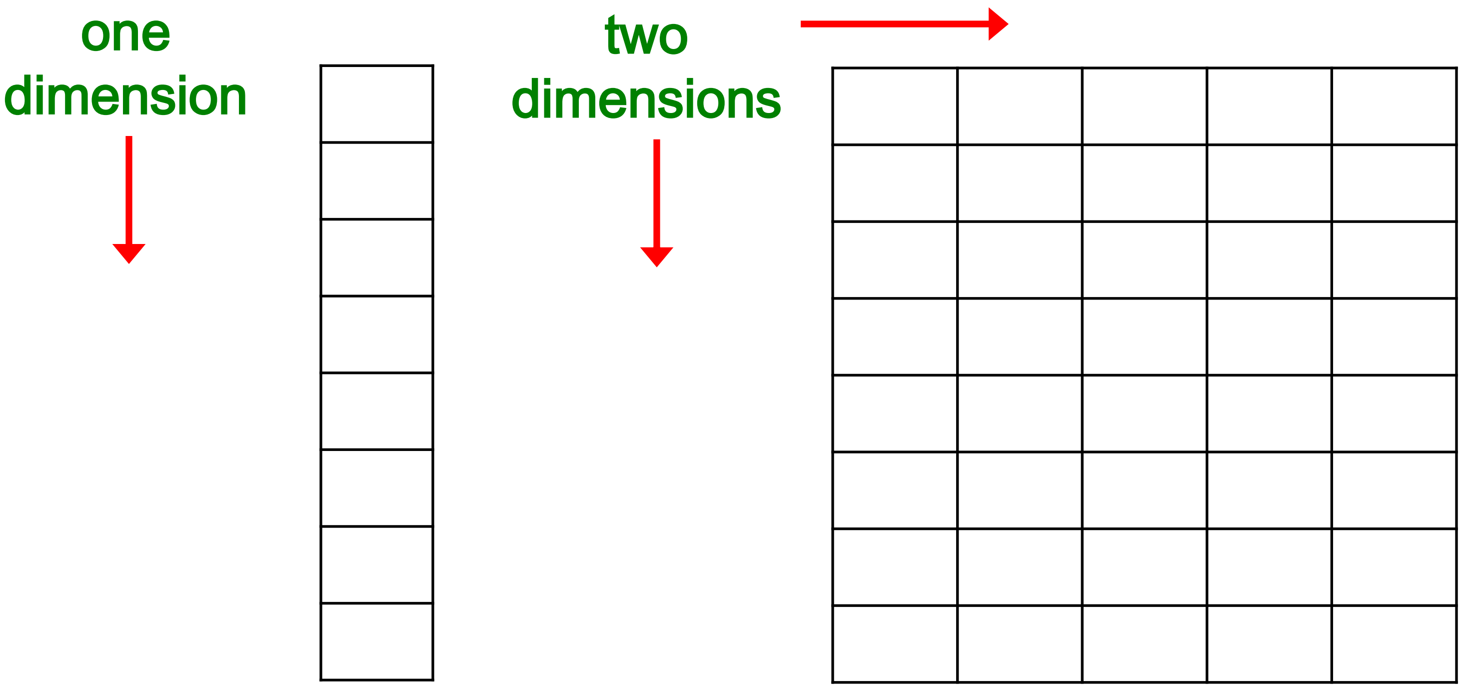
```
public static double average (int ... list)
{
   double result = 0.0;
   if (list.length != 0) {
      int sum = 0;
      for (int i = 0; i < list.length; i++)
         sum += list[i];
      result = (double) sum/ list.length;
   }
   return result;
}
```

Multi-dimensional Arrays

- An array can have many dimensions – if it has more than one dimension, it is called a *multi-dimensional array*
- Each dimension subdivides the previous one into the specified number of elements
- Each dimension has its own `length` constant
- Because each dimension is an array of array references, the arrays within one dimension can be of different lengths
  - these are sometimes called *ragged arrays*
  - no guarantee that a 2-d array will be a rectangle or a 3-d array box-shaped, etc.

Two-Dimensional Arrays

- A *one-dimensional array* can be thought of as a list (one column table) of elements
- A *two-dimensional array* can be thought of as a multi-column table of elements, with rows and columns



2-D Arrays

- To be precise, a 2-d array is an array of arrays
- A 2-d array is declared by specifying the size of each dimension separately:

```
int[][] table = new int[12][50];
```

- An array element is referenced using two indexes:

```
value = table[3][6]
```

- The array stored in one row can be specified using one index

2-D Arrays

Expression	Type	Description
`table`	**`int[][]`**	2-d array of integers, or an array of integer arrays
`table[5]`	**`int[]`**	array of integers
`table[5][12]`	**`int`**	integer

Soda Survey – 2-D Array Example

- Survey of 4 new flavors
- 10 testers each try each flavor and give a score
- Compute and print the average response
  - for each soda
  - of each tester

Soda Survey – 2-D Array Example

- We need a 4 row x 10 columns 2-d array to store the scores
  - each row corresponds to a soda
  - each column corresponds to a taster

```
int[][] scores =
        { {3, 4, 5, 2, 1, 4, 3, 2, 4, 4},
          {2, 4, 3, 4, 3, 3, 2, 1, 2, 2},
          {3, 5, 4, 5, 5, 3, 2, 5, 5, 5},
          {1, 1, 1, 3, 1, 2, 1, 3, 2, 4}
        };
```

- When using an initializer, no need to allocate a 2-d array using new

```
//--------------------------------------------------------
//  Determines & prints average of each row (soda) & each
//  column (respondent) of the survey scores.
//--------------------------------------------------------
public static void main (String[] args) {
   int[][] scores = { {3, 4, 5, 2, 1, 4, 3, 2, 4, 4},
                      {2, 4, 3, 4, 3, 3, 2, 1, 2, 2},
                      {3, 5, 4, 5, 5, 3, 2, 5, 5, 5},
                      {1, 1, 1, 3, 1, 2, 1, 3, 2, 4} };

   final int SODAS = scores.length;
   final int PEOPLE = scores[0].length;

   int[] sodaSum = new int[SODAS];
   int[] personSum = new int[PEOPLE];
```

continued on next slide

```
      for (int soda=0; soda < SODAS; soda++)
         for (int person=0; person < PEOPLE; person++)
          {
             sodaSum[soda] += scores[soda][person];
            personSum[person] += scores[soda][person];
          }

       System.out.println ("Averages:\n");

       for (int soda=0; soda < SODAS; soda++)
          System.out.printf("Soda # %d: %.1f\n", soda+1,
                     (double) sodaSum[soda]/PEOPLE);

       System.out.println ();
       for (int person=0; person < PEOPLE; person++)
          System.out.printf("Person # %d: %.1f\n", person+1,
                     (double) personSum[person]/SODAS);
    }
}
```

Output

```
Averages:

Soda #1: 3.2
Soda #2: 2.6
Soda #3: 4.2
Soda #4: 1.9

Person #1: 2.2
Person #2: 3.5
Person #3: 3.2
Person #4: 3.5
Person #5: 2.5
Person #6: 3
Person #7: 2
Person #8: 2.8
Person #9: 3.2
Person #10: 3.8
```

Matrices

- Matrices are 2-d arrays.
- Many matrix operations can be performed on matrices, e.g., multiplication, addition, subtraction
- We will implement multiplication
- I will ask you to do addition and subtraction which are simpler

Matrix (2-d Arrays) Multiplication

- Let X be a 2-d array with of size $m \times n$
- Let Y be a 2-d array of size $n \times p$
- The result of multiplying X and Y is an array Z of size $m \times p$ such that

  $$Z[i, j] = \sum_{k=1}^{n} X[i, k] * Y[k, j]$$

- In other words Z[i, j] is the sum of the products of the corresponding elements in the i^{th} row of X and the j^{th} column of Y

Matrix Multiplication – Input

- Input for the two matrices will be in a file
  - data for the first matrix will be followed by data for the second matrix
- Input specifying the first matrix will be in the format

  m n

  followed by

  m rows of n elements each
- and then for the second matrix in the same format

  n p

  followed by n rows of p elements each

Class Matrix

```
public class Matrix {
   int mat[][];
   public Matrix(int m, int n) {
      mat = new int[m][n];
   }
   public int getElement(int i, int j) {
       return mat[i][j];
   }
   public void setElement(int i, int j, int x) {
      mat[i][j] = x;
   }
   public int firstDim() {return mat.length; }
   public int secondDim() {
      return mat[0].length;
   }
```

continued on next slide

Class Matrix (contd.)

```
public Matrix multiply(Matrix y) throws MismatchMatrixBounds {
   Matrix x = this;  //easier to think x and y
   int m = x.mat.length;
   int n = 0;
   if (x.mat[0].length != y.mat.length)
      throw new MismatchMatrixBounds(
           "Matrix Bounds not satisfied for Multiplication");
   else
      n = y.mat.length;
   int p = y.mat[0].length;
   Matrix z = new Matrix(m, p);
   for (int i = 0; i < m; i++)
      for (int j = 0; j < p; j++) {
         z.mat[i][j] = 0;
         for (int k = 0; k < n; k++)
             z.mat[i][j] += x.mat[i][k] * y.mat[k][j];
      }
   return z;
}
```

continued on next slide

Exception MismatchMatrixBounds

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

continued on next slide

Class Matrix (contd.)

```
public String toString() {
      String s="";

      for(int i = 0; i < mat.length; i++){
         for(int j = 0;j < mat[0].length; j++)
            s += mat[i][j]+ " ";
         s += "\n";
      }
      return s;
   }
}
```

continued on next slide

Test Multiply Program

```
import java.io.*;  // for class File
import java.util.*; // class Scanner

public class  TestMultiply {

   public static void main(String args[])
            throws IOException, MismatchMatrixBounds {

// SET UP FILE FROM WHICH TO READ DATA

      if (args.length == 0) {
         System.out.println("Please supply data file" +
                            " as command-line argument");
         System.exit(0);
      }

      File inputDataFile = new File(args[0]);
      Scanner inputFile = new Scanner(inputDataFile);
```

Test Multiply Program (contd.)

```
// READ DATA FROM FILE
      int m = inputFile.nextInt();
      int n = inputFile.nextInt();
      Matrix x = new Matrix(m, n);

      int i, j;
     for(i = 0; i < m; i++)
         for(j = 0;j < n; j++)
            x.setElement(i, j, inputFile.nextInt());

      m = inputFile.nextInt();
      n = inputFile.nextInt();
      Matrix y = new Matrix(m, n);;

      for(i = 0; i < m; i++)
         for(j = 0;j < n; j++)
            y.setElement(i, j, inputFile.nextInt());
```

Test Multiply Program (contd.)

```
// COMPUTE MULTIPLY
      Matrix z = x.multiply(y);

// PRINT MATRICES
      System.out.println("X Matrix with dimensions " +
                         x.firstDim() + "x" + x.secondDim());
      System.out.println(x);

      System.out.println("Y Matrix with dimensions " +
                         y.firstDim() + "x" + y.secondDim());
      System.out.println(y);

      System.out.println("Z Matrix with dimensions " +
                         z.firstDim() + "x" + z.secondDim());
      System.out.println(z);
      System.exit(0);
   }
}
```

Data File for Matrix Multiply

```
1 5
1 1 1 1 1
5 1
1
1
1
1
1
```

Matrix Multiply Output

```
X Matrix with dimensions 1x5
1 1 1 1 1

Y Matrix with dimensions 5x1
1
1
1
1
1

Z Matrix with dimensions 1x1
5
```

More about classes

Inheritance, Abstract Classes

Inheritance

- *Inheritance* allows a software developer to *derive* a new class from an existing one
- The existing class is called the *parent* class, or the *superclass*, or the *base class*
- The derived class is called the *child* class, specialized class, or *subclass*
- A subclass inherits properties of its parent
  - its methods and data

Inheritance (contd.)

- The derived class can be customized by
  - adding new variables/fields and methods, or
  - modifying the inherited ones
- Three benefits of inheritance
  - software reuse – classes already defined are used to create new ones
  - variable of base class can point to object of base class and all classes derived from it
  - object of a derived class can be passed as an argument for a parameter of the base class type

Subclasses

Person → Employee → Manager → Executive

- What are some common characteristics as we do derivation?
- What do we add in terms of methods and data?

Subclasses (contd.)

- When defining a subclass, the superclass is specified using the extends-class clause, e.g.,

```
public class Employee extends Person {
     …
}
```

- Every class in Java has a superclass.
- If no superclass is explicitly specified, then Java assumes the superclass to be the special class `Object`.

Subclass vs. Superclass variables & methods

- Subclass variables with names identical to those of superclass variables override (hide) the superclass variables.
  - These superclass variables can be accessed in the subclass by qualifying their names with the keyword `super`.
- Similarly, subclass methods with signatures identical to superclass methods override the superclass methods
  - Methods declared as `final` in a superclass cannot be overridden.
  - Overridden superclass methods can be accessed in the subclass by qualifying their names with the keyword `super`.

Subclass & Superclass variables

- A variable of a class type, besides referring to objects of its class, can also refer to objects of its subclasses.
- A subclass object can be passed as an argument when an object of the superclass is expected.
  - This is because a subclass object has the same properties as the superclass object plus more.

Subclass Constructors

- A subclass constructor automatically calls the argument less constructor of the superclass to initialize the superclass part of the subclass object.
- If the superclass does not have an argument less constructor then another superclass constructor must be explicitly called.
- A superclass constructor can be explicitly called in the subclass constructor with the statement:

  `super([`*arguments*`]);`

- This statement must be the first statement in the subclass constructor body.

Inheritance Example

Consider, as a example, class `Person` which we will use to define a subclass:

```
public class Person {
    protected String First;
    protected String Last;

    Person(String first, String last)
        { First = first; Last = last; }

    void setFirstName(String first) { First = first; }
    String getFirstName() { return First; }
    void setLastName(String last) { Last = last; }
    String getLastName() { return Last; }

    void print() {
        System.out.println(First + " " + Last);
    }
}
```

Inheritance Example (contd.)

- An employee is a person with additional attributes, e.g., a
  - job title and
  - telephone extension.
- We will now derive class `Employee` from class `Person`.
- Besides, the above attributes, subclass `Employee` will have a new `print` method

Subclass to Show Inheritance

```
public class Employee extends Person {
    protected String Title;
    protected String Ext;

    Employee(String first, String last,
             String title, String ext) {
        super(first, last); //first line
        Title = title;
        Ext = ext;
    }
    void setTitle(String title) { Title = title; }
    String getTitle() { return Title; }
    void setExt(String ext) { Ext = ext; }
    String getExt() { return Ext; }

    void print() {
        super.print();
        System.out.println(Title);
    }
}
```

Inheritance Example (contd.) `print` method

- The `print` method in class `Employee` overrides the `print` method in its superclass `Person` (the two print methods have the same signature).
- To reference the `print` method of class `Person` we need to use the qualifier `super`. For example:

```
super.print();
```

- Within the body of class `Employee`, the unqualified identifier `print` refer to its `print` method.

Inheritance Example (contd.) `Employee` constructor

- Instead of invoking the superclass `Person` constructor in the `Employee` constructor,
  - we could have alternatively initialized the two variables `First` and `Last` of the superclass directly in the `Employee` constructor and
  - avoided calling the superclass constructor explicitly

Inheritance Example (contd.)

```
public class Employee extends Person {
    protected String Title;
    protected String Ext;

    Employee(String first, String last,
             String title, String ext) {
        First = first; Last = last;
        Title = title; Ext = ext;
    }

    void setTitle(String title) { Title = title; }
    String getTitle() { return Title; }
    void setExt(String ext) { Ext = ext; }
    String getExt() { return Ext; }

    void print() {
        super.print();
        System.out.println(Title);
    }
}
```

Inheritance Example (contd.)

- If the superclass constructor is not called explicitly:
  - superclass must have an argument less constructor (If no constructor is explicitly defined, Java supplies a default argument less constructor)
  - Java will call this implicitly when a subclass object is being created to initialize the subclass object.
- Thus in the second version of class `Employee`, Java requires an argument less constructor to be explicitly defined in class **`Person`**.
  - When an `Employee` object is created, part of it is a `Person` object and this part will, by default, be initialized by calling the argument less constructor in class `Person` (since we have not explicitly called a `Person` constructor).

Inheritance Example (contd.)

```
public class Person {
    protected String First;
    protected String Last;

    Person() {} //argument less constructor
    Person(String first, String last) {
        First = first;
        Last = last;
    }
    void setFirstName(String first) { First = first; }
    String getFirstName() { return First; }
    void setLastName(String last) { Last = last; }
    String getLastName() { return Last; }

    void print() {
        System.out.println(First + " " + Last);
    }
}
```

Inheritance Example (contd.)

- Here is some sample code illustrating the use of the two classes shown above:

```
Person p = new Person("Nina", "Masters");
p.print();

Employee e = new Employee("Nina", "Masters",
                          "Vice President", "4443");
e.print();
```

The output of the two `print` methods is:

```
Nina Masters
Nina Masters
Vice President
```

Overriding

- A subclass can *override* the definition of an inherited method in favor of its own

- The new method must have the same signature as the parent's method, but can have a different body

- The type of the object executing the method determines which version of the method is invoked

- **Note**: A super class object can refer to objects of its subclasses and their subclasses

Overriding (contd.)

- A method in the parent class can be invoked explicitly using the `super` reference

- If a method is declared with the `final` modifier, it cannot be overridden

- The concept of overriding can be applied to data and is called *shadowing variables*

- Shadowing variables should be avoided because it leads to confusing code

Overloading vs. Overriding

- Overloading deals with multiple methods with the same name in the same class, but with different signatures
  - Overloaded operations are typically written to have similar functionality
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
  - Overriding lets one define similar operations for different object types

The `Object` Class
(defined in the `java.lang` standard library package)

- All classes are derived from the `Object` class
- If a class is not explicitly defined as a subclass of an existing class, it is assumed to be derived from the `Object` class
- The `Object` class is the ultimate root of all class hierarchies

The `Object` Class (contd.)

- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString()` method is defined in the `Object` class
- Every time we define the `toString()` method, we are actually overriding an inherited definition
- The `toString()` method in the `Object` class is defined to return a string that contains the name of the object's class along with a hash code
  - This `toString()` will be used if a class does not define one unless it is derived from another class which defines a `toString()` method
  - The `toString()` of the `Object` class is not very useful
  - Need to define one for each class

The `Object` Class (contd.)

- The `equals()` method of the `Object` class returns true if the two arguments (two object references) are aliases of each other

- We can override `equals` in any class to define equality in some more appropriate way
  - As we have seen, the `String` class defines the `equals` method to return true if two `String` objects contain the same characters

  - The designers of the `String` class have overridden the `equals()` method inherited from `Object` in favor of a more useful version

`abstract` Classes

- `interface` classes are a restricted type of `abstract` classes
- `abstract` classes often contain both
  - methods with no definitions (like those in an interface), i.e., abstract methods and
  - methods with full definitions
- Unlike `interface` classes, `abstract` classes can be used to derive other classes
- members of an `interface` are public
- a class can implement multiple interfaces

Abstract Classes (contd.)

- `abstract` classes, like `interface` classes, cannot be used directly.
  - They can only be used to derive new classes.
- Defining a class to be `abstract` forces the subclass (unless the subclass is `abstract` also) to supply definitions for the `abstract` methods.
- Not all methods need to be abstract.
- A class containing an `abstract` method must be specified as abstract.

Abstract Class `Shape`

- As an example of an `abstract` class, consider class `Shape` that defines two abstract methods:

```
public abstract class Shape {
  abstract void draw();
  abstract double perimeter();
}
```

Abstract Class `Shape` (contd.)

- Class `Shape` is used as the superclass for classes specifying different object shapes (circles, squares, triangles, etc.).
- Class `Shape` is defined as `abstract` to force all classes that extend `Shape` to define methods `draw` and `perimeter`.
  - Method `draw` generates a line drawing of the object by a `Shape` subclass object (there can be no `Shape` objects as it is an abstract class).
  - Method `perimeter` prints the value of the perimeter of a `Shape` subclass object.

Abstract Class `Shape` (contd.)

- Any class that extends class `Shape` (provided it is not an `abstract` class itself) must supply definitions for the abstract methods `draw` and `perimeter`.
- Class `Square` shown next is one such class

Class `Square` (using an abstract class)

```
public class Square extends Shape {
    protected double Side;
    Square(double side) {
        Side = side;
    }

    public void draw() {
       // ...
    }
    public double perimeter() {
        return 4*Side;
    }

    public void setSide(double side) { Side = side; }
    public double getSide() { return Side;}

}
```

Final Classes

- A `final` class cannot be used to derive subclasses.
  - All methods of a `final` class are automatically treated as `final` methods that cannot be overridden in subclasses.
  - An object of a `final` class is truly an object of the class itself and not an object of one of its subclasses.
  - A user of a `final` class can be sure of getting the class specified and not some variation of it.
- The `String` class, e.g., is declared as `final`.
  - When an object of type `String` is passed as an argument to a method, the method can be sure that the object is of type `String` and not of a subclass of `String`.

Run-time Type Information

- A variable of the class type T can refer to
  - objects of objects of type T and
  - objects of class types derived from T.
- The `instanceof` operator can be used to determine the type of the object referenced by such a variable.
- Expression

a `instanceof` T

where *a* is a variable of a class type evaluates to `true` if the object referenced by a is of the class type T; otherwise, it evaluates to `false`.

Run-time Type Information (contd.)

- Allowing variables of a type, say T, to also refer to objects of subclasses of type T is useful in many situations.
- For example, this allows
  - an array to hold of objects of type T and of all subclasses derived from T , and
  - a method to accept similar types of objects as arguments.
- As mentioned, the `instanceof` operator can be used to determine the exact object type.
- When invoking a method on an object of type T, Java at run time, checks to see if the object is of type T or a type derived from T and invokes the appropriate method automatically.

Run-time Type Information (Contd.) Class `Shape`

- As an example illustrating the use of the `instanceof` operator, we will look at classes `Square` (shown earlier) and `Circle` that are both derived from the `abstract` class `Shape`:

```
public abstract class Shape {
    abstract void draw();
    abstract double perimeter();
}
```

Run-time Type Information (Contd.) Class `Circle` extending `Shape`

```
public class Circle extends Shape {
    protected double Radius;

    Circle(double radius) { Radius = radius; }

    public void draw() { // ... }

    public double perimeter() {
        return 2*Math.PI*Radius;
    }

    public void setRadius(double radius) {
        Radius = radius;
    }

    public double getRadius() { return Radius; }
}
```

Run-time Type Information (Contd.)

- Both classes `Square` and `Circle` are subclasses of the class `Shape`.
- Consider the following code:

```
Shape sh; Circle c;
Square s = new Square(5);
```

- Suppose that the `Square` object referenced by `s` is assigned to `sh`.

```
sh = s;
```

- Executing the statement that converts `sh` to a circle

```
c = (Circle) sh;
```

will be an error because `sh` is really a `Square` object.

- Such errors are not detectable at compile time but the Java interpreter will detect this at runtime. and throw the exception `java.lang.ClassCastException`.

Run-time Type Information (Contd.)

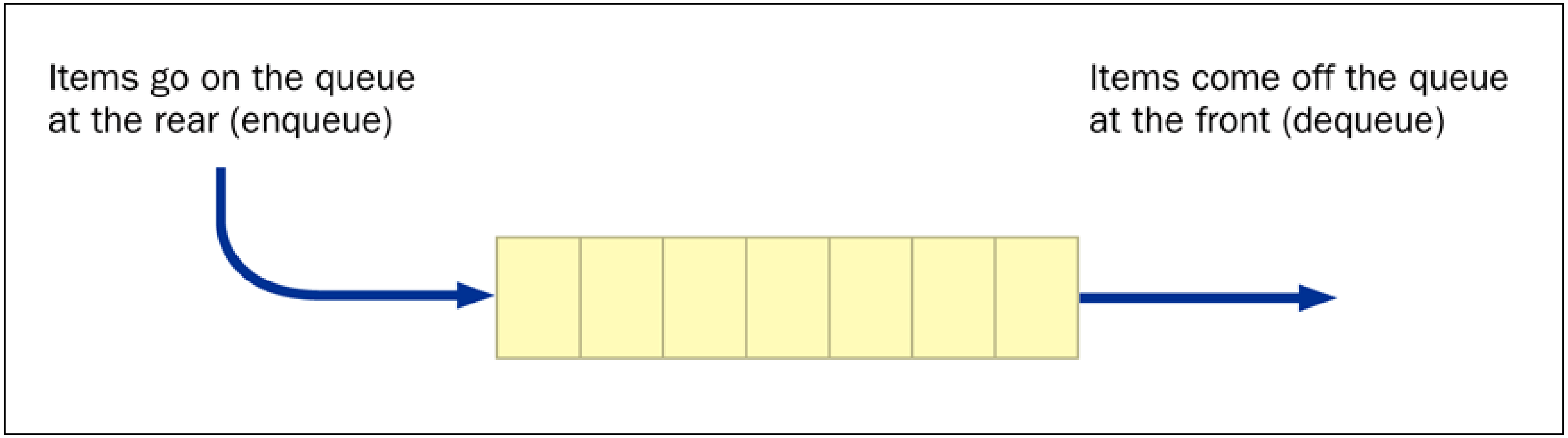
- Prior to converting a class object back to its type, the type of the object should be checked (if there is any doubt) using the `instanceof` operator.
- For example:

```
if (sh instanceof Circle) {
    c = (Circle) sh;
} else if (sh instanceof Square) {
    s = (Square) sh;
}
```

More Data Structures (besides arrays)

Queues

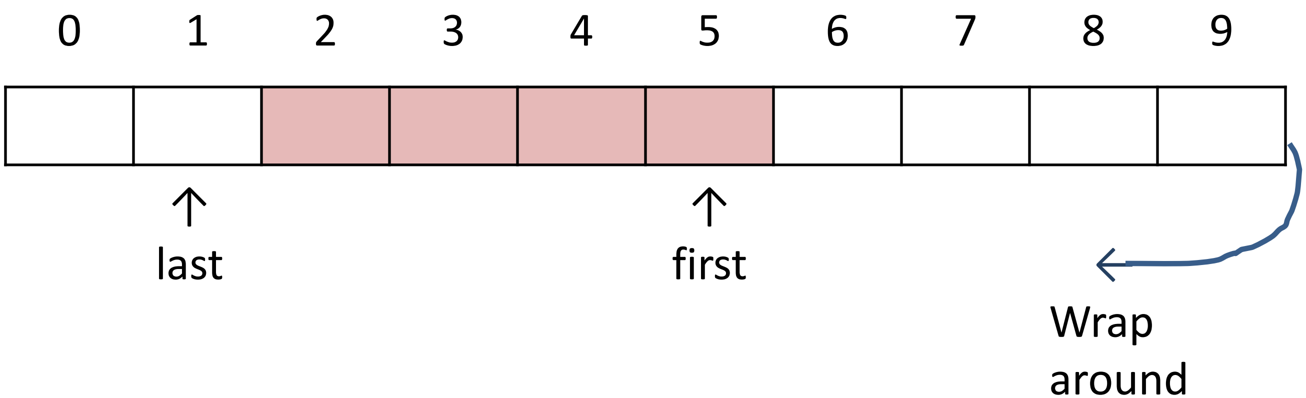
- A *queue* is a list that adds items only to the rear of the list and removes them only from the front
- It is a FIFO data structure: First-In, First-Out
- Analogy: a line of people at a bank teller's window



Queues

- A queue can be represented by an array.
- The remainder operator (%) to “wrap around the array” when the end of the array is reached

Queues



- **max == 10 → Max size of queue array q**
- **Current size of queue = n (now 4)**
- **n == 0 → queue empty**

Queues

- Classic operations for a queue
  - *enqueue*: add an item to the rear of the queue
  - *dequeue*: remove an item from the front of the queue
  - *empty*: returns `true` if the queue is empty and `false` otherwise
- Queues often are helpful in simulations or any situation in which items get "backed up" while awaiting processing

Class Queue

```
public class Queue {
   int q[]; //queue
   int max; // max size of q
   int n; // size of queue
   int first, last;
  public Queue(int size) {
      q = new int[size];
      max = size;
      n = first = last = 0;
   }

   public void enqueue (int item) {
      if (n < max) {
         n++; q[last] = item;
         last = (last+1) % max;  //circular array
      }
      else {
         System.out.printf("Error: Queue Full");
         System.exit(0);
      }
   }
```

Class Queue (contd.)

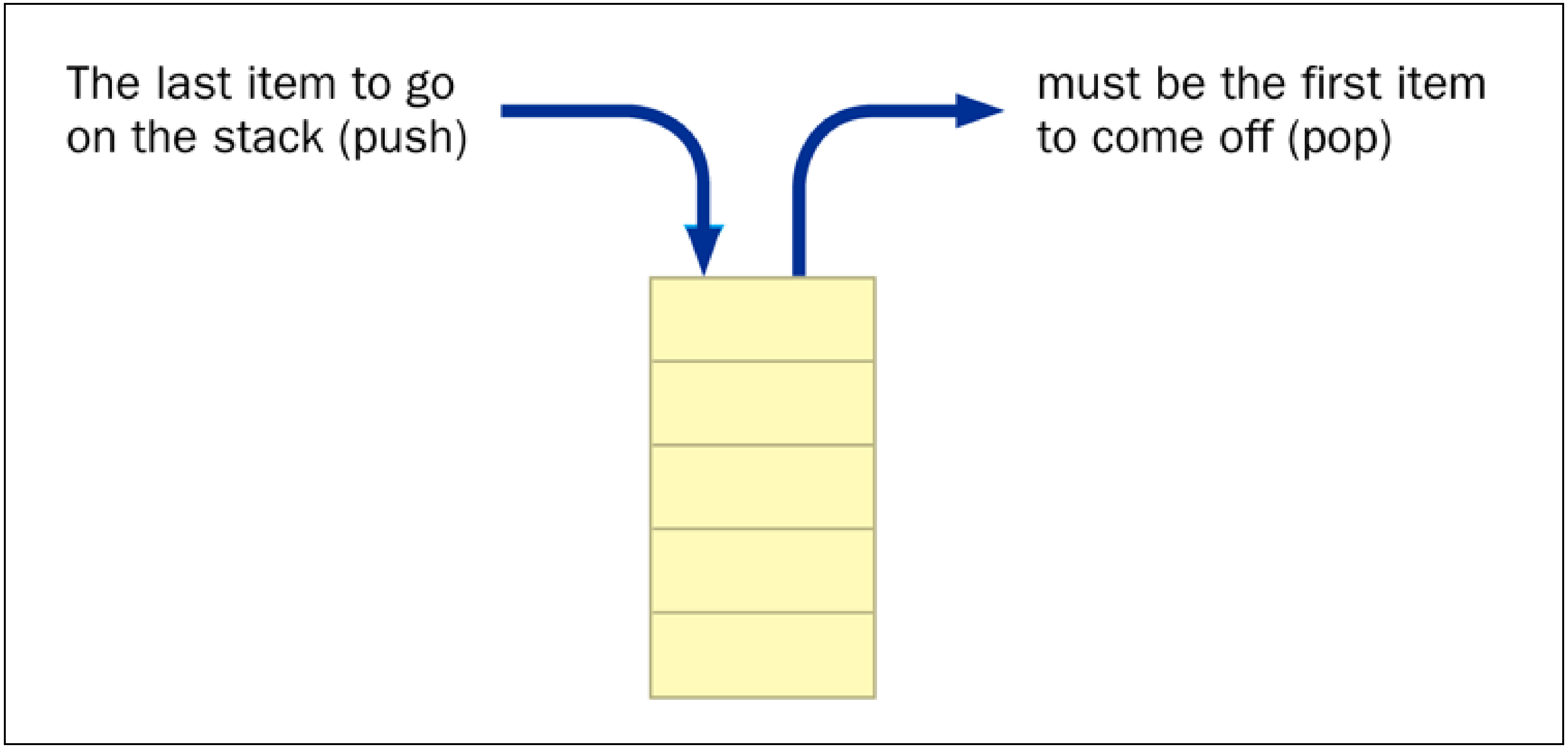
```
public int dequeue () {
      int item = 0;
      if (n > 0) {
         n--; item = q[first];
         first = (first+1) % max;  //circular array
      }
      else {
         System.out.printf("Error: Queue Empty");
         System.exit(0);
      }
      return item;
   }
   public boolean empty() {
      return n == 0;
   }
   public boolean full() {
      return n == max;
   }
}
```

Stacks

- Items to a stack are added and removed from only one end of a stack
- It is therefore LIFO: Last-In, First-Out
- Analogies: a stack of plates or a stack of books

Stacks

- Stacks often are drawn vertically:



Stacks

- Classic stack operations:
  - *push*: add an item to the top of the stack
  - *pop*: remove an item from the top of the stack
  - *peek* (or top): retrieves the top item without removing it
  - *empty*: returns `true` if the stack is empty; otherwise `false`.
  - *full*: returns `true` if the stack is full; otherwise `false`
- A stack can be represented by an array

Stack

```
public class Stack {
   int max, n; // max size & number of elements
   int top;  // top points to first empty slot
   int stk[];

   public Stack(int size) {
      max = size; n = 0; top = 0;
      stk = new int[max];
   }
   public void push(int item) {
      if (n < max) {
         n++; stk[top++] = item;
      }
      else {
         System.out.println("Error: Stack Full");
         System.exit(0);
      }
   }
```

Stack (contd.)

```
 public int pop() {
      int item = 0;
      if (n > 0) {
         item = stk[top-1]; n--; top--;
      }
      else {
         System.out.println("Error: Stack Empty");
         System.exit(0);
      }
      return item;
   }
   public int peek() {
      if (n == 0) {
         System.out.println("Error: Stack Empty");
         System.exit(0);
      }
      return stk[top-1];
   }
   public boolean empty() { return n == 0; }
   public boolean full() { return n == max; }
}
```

Recursion, Searching & Sorting

Recursion

- Powerful divide-and-conquer tool
  - Divide problem into one or more subproblems (hopefully the smaller problems are simpler to solve)
  - Solve the subproblems
  - Use the solutions of the subproblems to construct a solution to the original problem
- Recursion comes into play as the strategy used to solve the original problem is used to solve the subproblems
  - There must be a solution to a subproblem that breaks the recursion

Recursion (contd.)

- Recursion in programming occurs when a method calls itself for additional computation
- Care needs to be taken when using recursion because just like infinite loops we can have infinite recursion
  - Recursion has to be terminated at some point

Examples of Recursion (contd.)

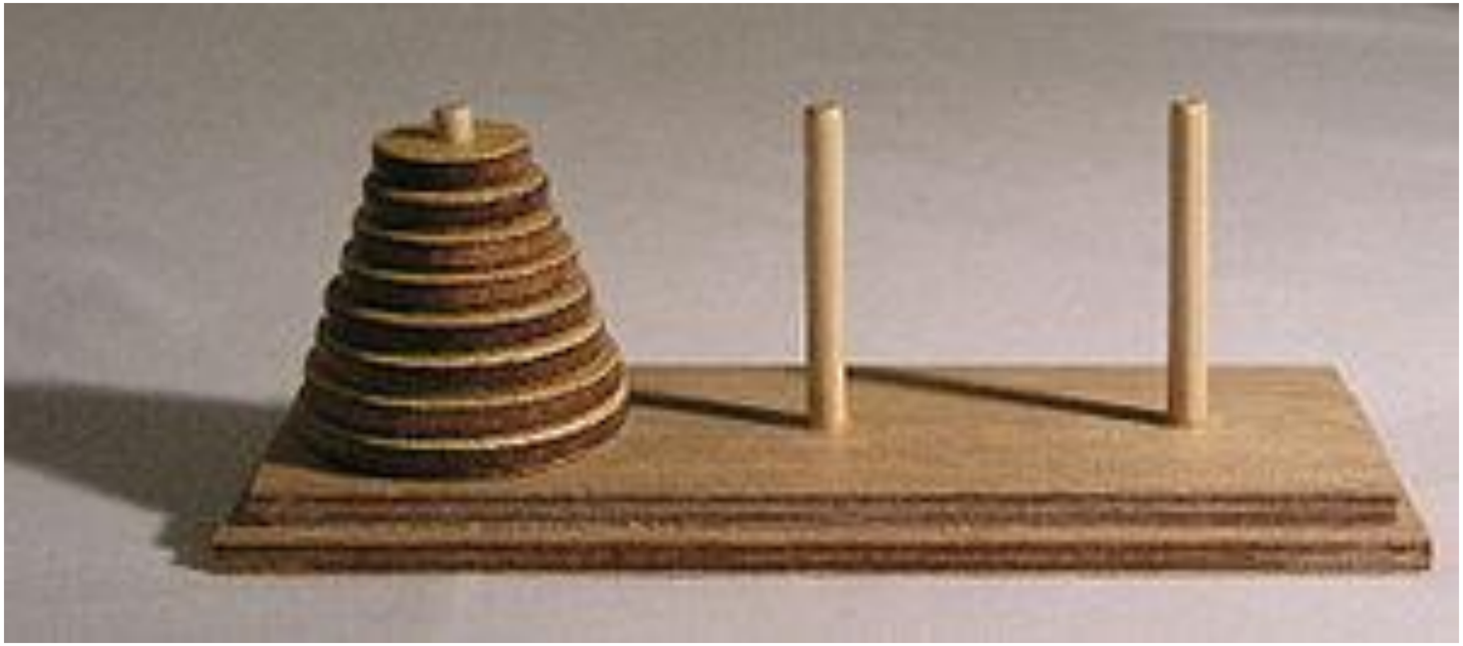
- Math Example
  - Factorial (seen before)
  - Towers of Hanoi
  - Fibonacci numbers
- Searching
  - Finding a number in a sorted list (like a old-style telephone directory)
- Sorting an Array

Recursion Example – Factorial
(we saw this before)

Factorial(0) = 1
Factorial(n) = n × Factorial(n-1)

- To solve Factorial(n) we need to solve the subproblem Factorial(n-1)
- If we solve the subproblem, we can use its solution to construct the solution to the problem
- Recursion breaks when we reach Factorial(0).

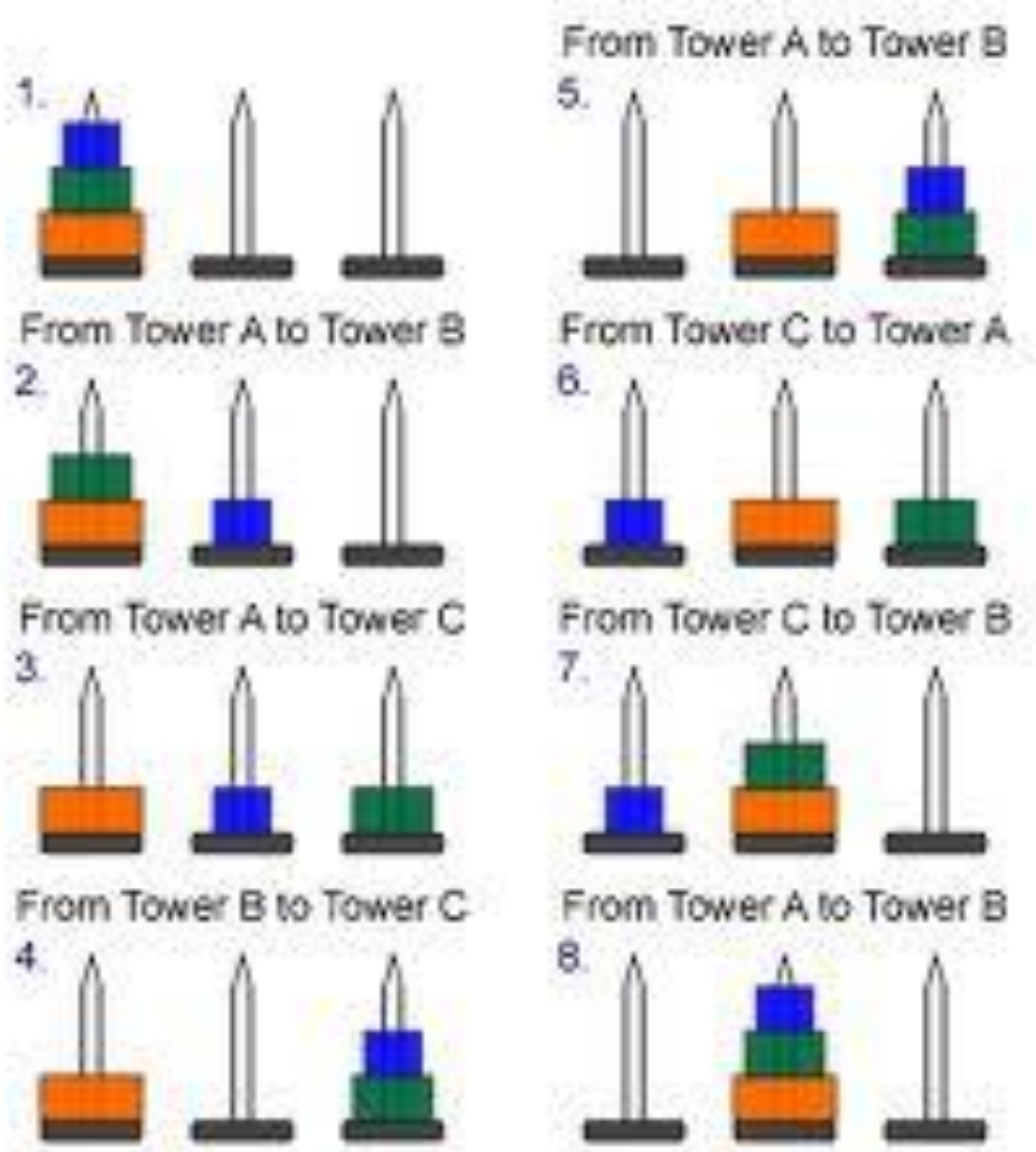
Towers of Hanoi (from Wikipedia)



Towers of Hanoi
Another Example of Recursion

- Three rods A, B, C
- Rod A has a pile of N disks of different sizes stacked up in decreasing size like a tower
- The task is to move the N disks from A to B using C
  - Disks can only be placed on one of the rods (not on the ground)
  - A larger disk cannot be placed on top of a smaller disk
  - Only one disk can be moved at a time

image
from
the
web



Towers of Hanoi

- Move N disks from A to B using C as a temporary holder
- This can be done as
  1. Move N-1 disks from A to C using B as a temporary holder
  2. Move disk N from A to B
  3. Move N-1 disks from C to B using A as a temporary holder

Towers of Hanoi (contd.)
(Moving N disks – adding check for 0)

- If N = 0, do nothing; otherwise
- Move N disks from A to B using C as a temporary holder
- This can be done as
  1. Move N-1 disks from A to C using B as a temporary holder
  2. Move disk N from A to B
  3. Move N-1 disks from C to B using A as a temporary holder

Towers of Hanoi (contd.)

```
// Towers of Hanoi
// use: Hanoi NumberOfDisks
//  or
//      Hanoi (will ask for Number of disks)

import java.util.*;  // for class Scanner
public class Hanoi {

   public static void  main(String[] args) {

      int number_of_disks;

      if (args.length == 1) { // number of disks in command-line?
         number_of_disks = Integer.parseInt(args[0]);
      }
      else {  // ask for number of disks
         Scanner scan = new Scanner(System.in);
         System.out.print("Enter Number of Disks to be moved: ");
         number_of_disks = scan.nextInt();
      }

      move(number_of_disks, 'A', 'B', 'C');
   }
```

continued on next slide

Towers of Hanoi (contd.)

```
 private static void move(int n,char X,char Y,char Z){
        // move n disks from X to Y using Z
      if (n != 0) {
         move(n-1, X, Z, Y);
         System.out.println("Move disk " + n +
                         " from " + X + " to " + Y);
         move(n-1, Z, Y, X);
      }
 }
}
```

Does the Program Towers of Hanoi Work?

- You can try understanding the algorithm or do some testing with
  - a negative number,
  - zero and
  - some positive numbers

Fibonacci Numbers – An example of Recursion

- Fibonacci series

  0, 1, 1, 2, 3, 5, …
- A Fibonacci number is the sum of the previous two numbers in the series
- Exceptional case: first two numbers in the series which are 0 and 1
- Fibonacci series is specified as

  f(1) = 0

  f(2) = 1

  f[n] = f[n-2] + f[n-1]

Fibonacci Numbers (contd.)

- The recursive computation of the Fibonacci numbers is straightforward and matches the definition
- The iterative computation is more challenging as we will see

Fibonacci Numbers (contd.)

```
public class Fibonacci {
   public static int fibRecursive(int n) {
      if (n <= 0 ) {
         System.err.println("Error: Fibonacci series"
                                   + " number must be > 0");
         System.exit(0);
      }
      switch(n) {
         case 1:
            return 0;
         case 2:
            return 1;
         default:
            return fibRecursive(n-2)+fibRecursive(n-1);
      }
   }
```

Fibonacci Numbers (contd.)

```
public static int fibIterative(int n) {
      int fn, fn1, fn2; // fn1 = f[n-1], fn2 = f[n-2];
      if (n <= 0 ) {
         System.err.println("Error: Fibonacci series number must be > 0");
         System.exit(0);
      }
      switch(n) {
         case 1 :return 0;
         case 2: return 1;
      }
      int i = 3;
      fn = 0;
      fn2 = 0; fn1 = 1; // fn2 == f[i-2]
                        // fn1 == f[i-1]

      while(i <= n) {      // at this point fn1 == F[i-1], fn2 == F[i-2]
         fn = fn1 + fn2;
         i++;
               // at this point NOT TRUE that fn1 == F[i-1], fn2 == F[i-2]
         fn2 = fn1; fn1 = fn;
               // now fn1 == F[i-1], fn2 == F[i-2]
      }
      return fn;
   }
}
```

Fibonacci Numbers (contd.) Demo Driver

```
// use: FibonacciDemo n
//  or
//      FibonacciDemo (will ask for the number n)
// where n is the number of the Fibonanci term to be
// computed

import java.util.*;  // for class Scanner
public class FibonacciDemo {
   public static void  main (String[] args) {
      int n;

      if (args.length == 1) { // Fibonnaci number in command-line?
         n = Integer.parseInt(args[0]);
      }
      else {  // ask for number of disks
         Scanner scan = new Scanner(System.in);
         System.out.print("Enter Fibonnaci Number to be computed: ");
         n = scan.nextInt();
      }
      System.out.println("Fibonacci Number " + n +
          " recursively computed = " + Fibonacci.fibRecursive(n));
      System.out.println("Fibonacci Number " + n +
          " iteratively computed = " + Fibonacci.fibIterative(n));
   }
}
```

Searching

Searching

- Linear search is typically used for unsorted items
- Binary search is used with sorted items to minimize the search time

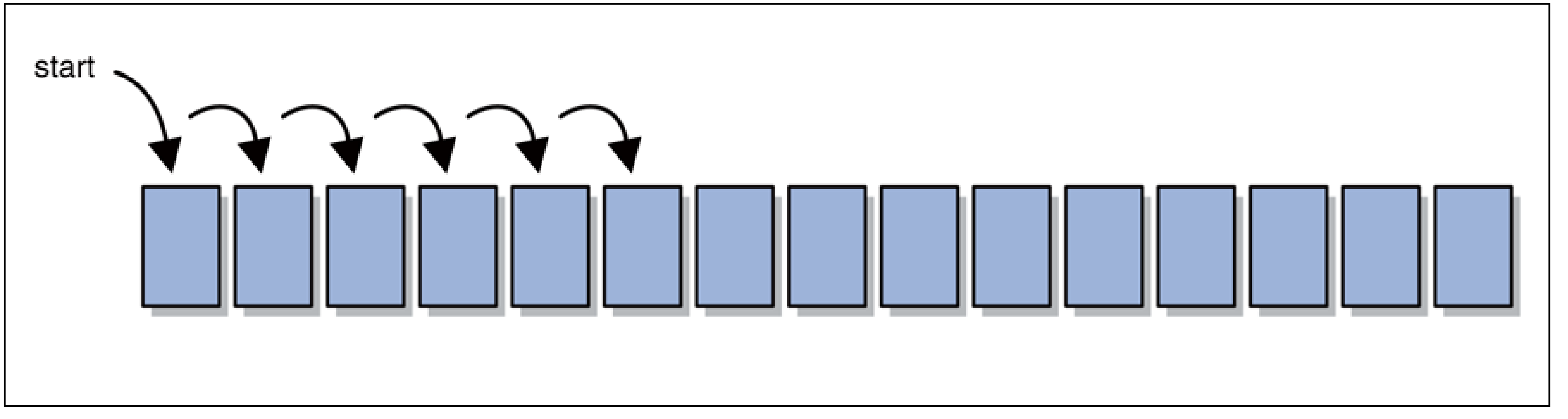
<u>Search time</u>

- The number of comparisons that need to be made depends upon the
  - data structure
  - search technique

Linear Search

- A linear search begins at one end of a list and examines each element in turn

- Eventually, either the item is found or the end of the list is encountered

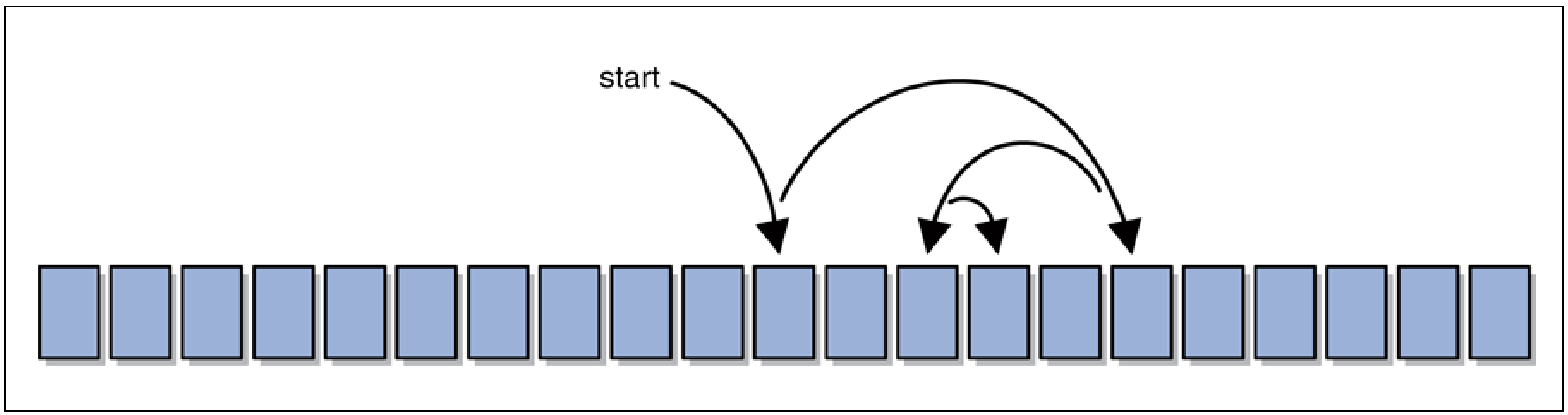


Binary Search

- A *binary search* assumes the list of items to be searched is sorted
- It eliminates half the list with a single comparison
  - A binary search first examines the middle element of the list – if it matches the target, the search is over
  - If not, only one half of the remaining elements need be searched
  - Since they are sorted, the target can only be in one half or the other

Binary Search

- Each comparison eliminates approximately half of the sorted list to be searched
- Eventually, the target is found if present



Search Example

- Class `Search` specifies two methods
  - `linearSearch()`
  - `binarySearch()`

  to search integer arrays
- Binary search is a natural candidate for a recursive implementation

Search Example (contd.)

```
public class Search {
   public static boolean linearSearch(int[] a, int searchKey) {
      for(int i = 0; i < a.length; i++)
         if (a[i] == searchKey)
            return true;
      return false;
   }
   //searching sorted arrays
   public static boolean binarySearch(int[] a, int searchKey) {
      return bSearch(a, 0, a.length-1, searchKey);
   }
```

Search Example (contd.)

```
private static boolean bSearch(int[] a, int left, int right, int key) {

      if (left > right) // no elements in array
         return false;

      int mid = (left+right)/2;

      if (a[mid] == key)
         return true;

      if (a[mid] < key)
         return bSearch(a, mid+1, right, key);
      else
         return bSearch(a, left, mid-1, key);

   }
}
```

Search Driver

```
import java.io.*;  // for class File
import java.util.*; // class Scanner
public class  searchTester {
   public static void main(String args[]) throws IOException {
      final int MAX_ARRAY_SIZE = 1024;
      int[] data = new int[MAX_ARRAY_SIZE];
                        // stores input data
      int size = 0; // populated with data read
      int[] a; // array to be populated for searching

      // read input from the file
      File inputDataFile = new File(args[0]);
      Scanner inputFile = new Scanner(inputDataFile);

      while(inputFile.hasNextInt() && size < MAX_ARRAY_SIZE) {
         data[size] = inputFile.nextInt();
         size++;
      }
      a = new int[size];
      for (int i = 0; i < size; i++) { a[i] = data[i];}
```

continued on next slide

Search Driver (contd.)

```
// print input data
System.out.println("Input Data Supplied");
System.out.println("Data Set Size = " + size);
for (int i = 0; i < size; i++) {
   System.out.print(data[i] + " ");
}
System.out.println();

// ask for the key
Scanner scan = new Scanner(System.in);
System.out.print("Enter key value for search: ");
int searchKey = scan.nextInt();
```

continued on next slide

Search Driver (contd.)

```
        // linear search
        if (Search.linearSearch(a, searchKey))
           System.out.println(searchKey +
              " is in the array - linear Search");
        else
           System.out.println(searchKey +
              " is NOT in the array - linear Search");

        // binary search
        if (Search.binarySearch(a, searchKey))
           System.out.println(searchKey +
              " is in the array - binary Search");
        else
           System.out.println(searchKey +
              " is NOT in the array - binary Search");
    }
}
```

Sample Data to Be Searched

```
-10
-9
-4
0
1
4
7
99
111
200
234
477
```

Sample Search

```
Input Data Supplied
Data Set Size = 12
-10 -9 -4 0 1 4 7 99 111 200 234 477
Enter key value for search: 99
99 is in the array - linear Search
99 is in the array - binary Search


Enter key value for search: 476
476 is NOT in the array - linear Search
476 is NOT in the array - binary Search
```

Effort Required for Searching

- Suppose we have an array of 1024 elements
- Linear search
  - in the worst case requires comparing all the 1024 elements
  - on the average requires comparing about 512 elements
- Binary search
  - in the worst case requires comparing at most 10 elements

Sorting

Sorting

- We will focus on sorting integer arrays for simplicity
- Sorting arrays with elements of different types is similar – the type must have equality and comparison operations
- Sorting can be ascending or descending (our examples will sort in ascending order)

Sorting (contd.)

- Many sorting techniques – we will discuss a few
  - Selection Sort
  - Insertion Sort
  - Quicksort
  - MergeSort

but first …

Swapping

Swapping

- To sort an array, we need to be able to *swap* the values of two variables (specifically values of two array elements)
- Swapping requires three assignment statements and a temporary storage location
- E.g., to swap values of variables `first` & `second`:

```
temp = first;
first = second;
second = temp;
```

Selection Sort

Sorting array A of size n in increasing order

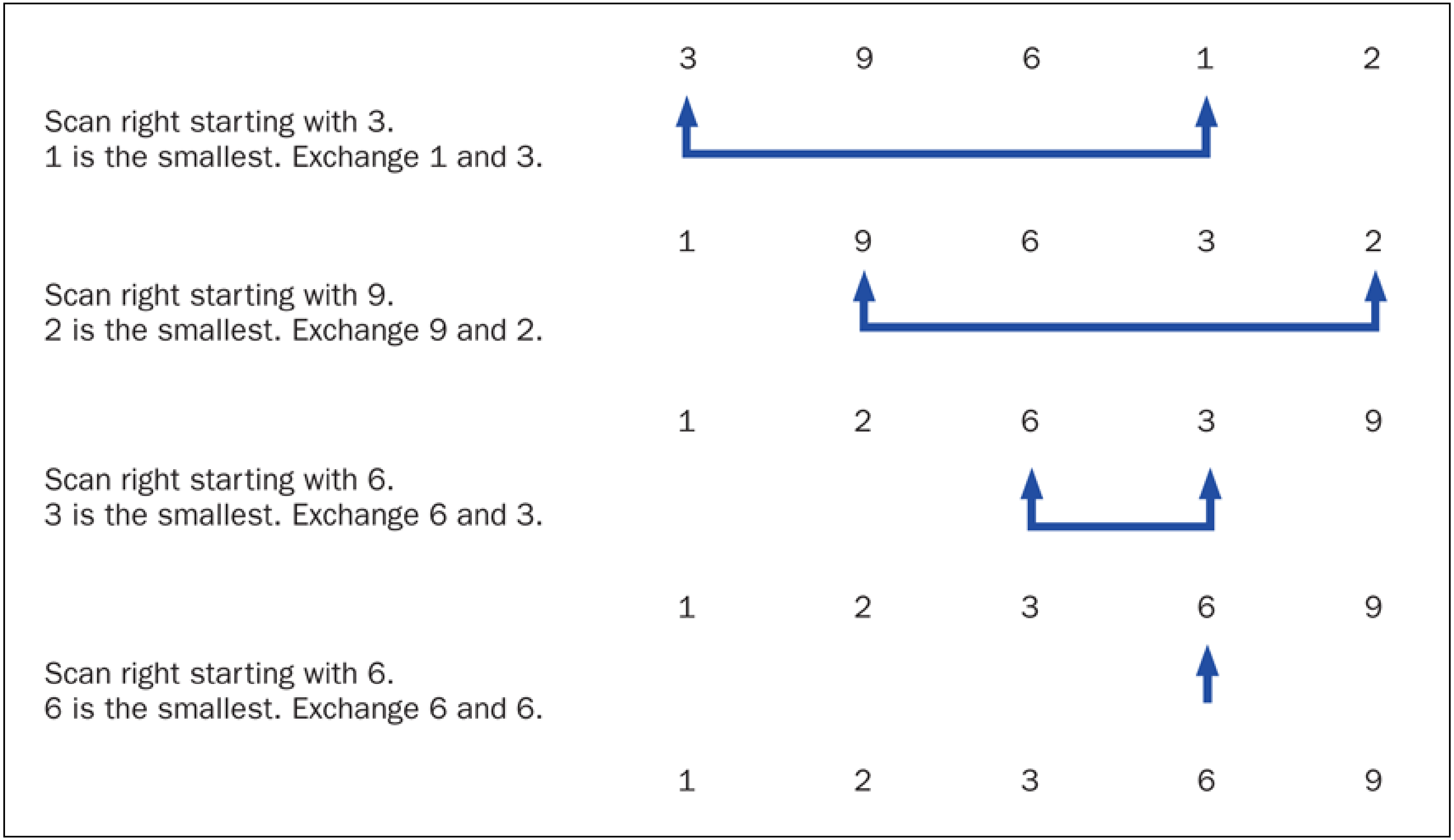
1. Start with the first element of the array, i.e., A[0], and exchange it with the smallest element in A[1:n-1] that is less than A[0]
2. Next, take the second element of the array, i.e., A[1] and exchange it with the smallest element in A[1:n-1] that is less than A[1]
3. Repeat with the next element each time until finished

Selection Sort (2nd Verson)

- If array A has only one element then do nothing,
- Otherwise:

  for(i = 1; i < A.length; i++)
  In the array[i:A.length-1] find the smallest element that is less than a[i-1] and swap them

Selection Sort



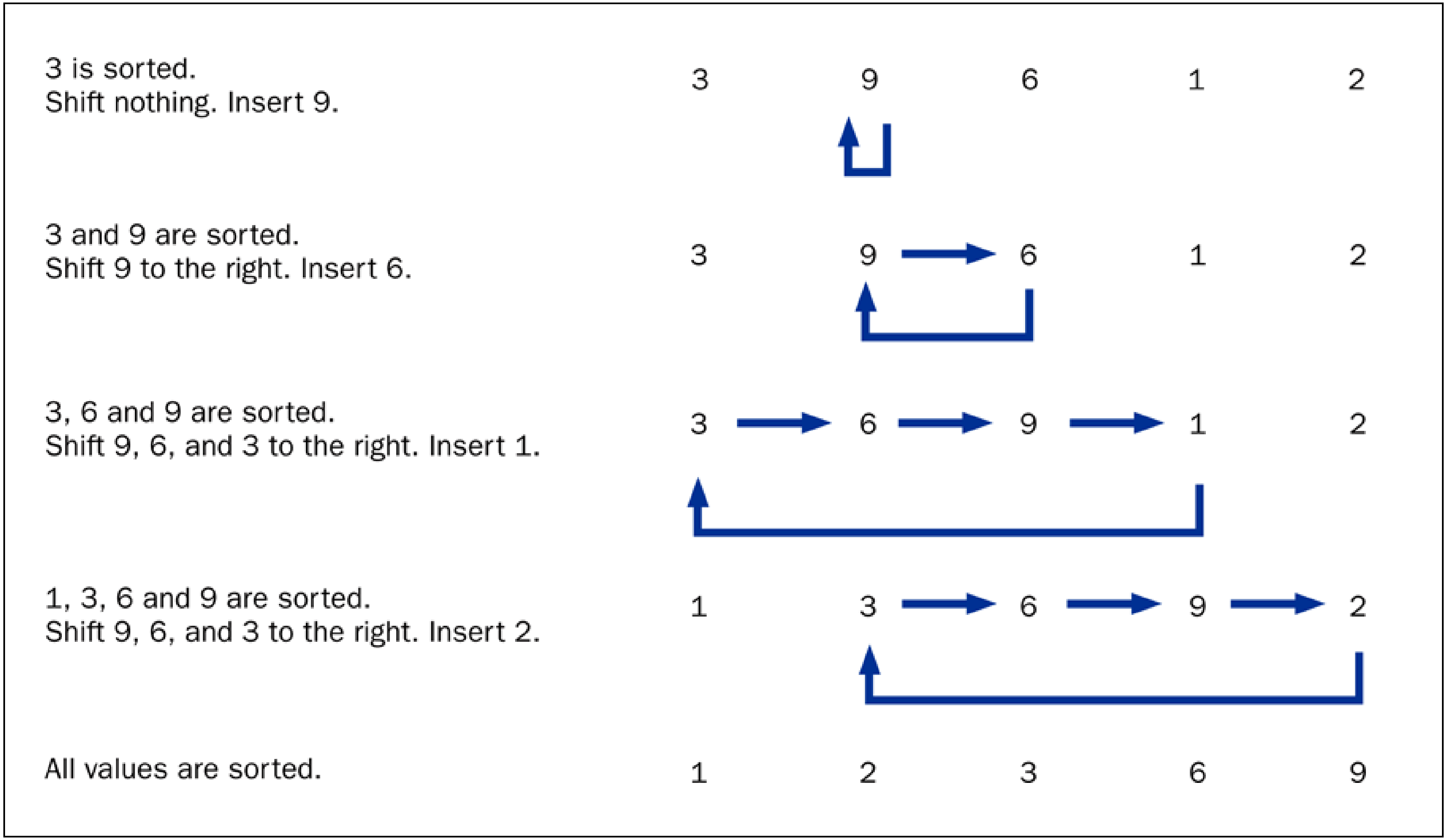
Selection Sort

```
public static void selectionSort(int a[], int low, int high) {

// if array a has only one element then do nothing
// for(i = 1; i < a.length; i++)
//   In the array[i:a.length-1] find the smallest element
//   that is less than a[i-1] and swap them

   int min, minIndex; int length = high-low+1;

   if (length == 1) return;
   for (int i = low+1; i < length; i++) {
      minIndex = i-1; min = a[minIndex];
      for (int j = i; j < length; j++) {
         if (a[j] < min) {
            min = a[j]; minIndex = j;
         }
      }
      int save = a[i-1];
      a[i-1] = a[minIndex];
      a[minIndex] = save;
   }
}
```

Insertion Sort

- Consider the first item of the array to be a sorted sublist (of one item)
- Insert the second item into the sorted sublist, shifting the first item, if needed, to make room to insert the new one to create a sorted sublist of 2
- Insert the third item into the sorted sublist of two items, shifting items as necessary – creating a sorted sublist of 3 items
- Repeat until all values are inserted into their proper positions

Insertion Sort



Insertion Sort algorithm in more detail

insertionSort(int a[], int low, int high)

Initially sublist a[low:low] is sorted (one item sublist)
Extend this sublist to be a[low:high] and we are done

We can write this as

```
for(i = low; i != high; i++)
    extend a[low:i] to include a[i+1]
```

initially a[low:i] is sorted because i == low
finally the whole array is sorted because a[low:i] is sorted and i == high

Insertion Sort algorithm (contd.)

extend a[low:i] to include a[i+1]

can be performed as follows

let temp = a[i+1]   // a[i+1] is free can now be used for other elements
find the location a[j] in a[low:i] where to insert temp
shift all elements a[j:i] one place to the right using up a[i+1]'s spot
a[j] = temp

find the location a[j] in a[low:i] where to insert temp

```
for (j = low; a[j] < temp; j++) // will stop before i+1 because temp == a[i+1]
    ;
```

shift all elements a[j:i] one place to the right using up a[i+1]'s spot

```
for (k = i; k>=j; k--)
    a[k+1] = a[k];
```

Insertion Sort

```
public static void insertionSort(int a[], int low, int high) {

   int i, j, k, temp;

   // Initially region a[low:low] is sorted
   // Extend this region to be a[low:high] and we are done
      for(i = low; i != high; i++) {

         // extend a[low:i] to include a[i+1]

         temp = a[i+1];

         // first locate where a[i+1] should be inserted
         for (j = low; a[j] < temp; j++) // note use of null stmt
               ;

         // shift all elements a[j:i] one place
         // to the right using a[i+1]'s spot

         for (k = i; k >= j; k--)
            a[k+1] = a[k];

         // insert
         a[j] = temp;
   }
}
```

QuickSort

```
public static void quickSort(int a[], int low, int high) {
    if a has 0 or 1 elements then return
    if a has 2 elements then swap them if they are not
            in order and return
    partition a into two parts by swapping elements
            so that elements in the left part have values
            less than elements in the right part
    quicksort(a, low, partitionPoint)
    quicksort(a, partitionPoint+1, high)
}
```

Partition

private static int partition(int a[], int left, int right)

partitions a in two parts and returns a partitionPoint such that
    elements in a[left:partitionPoint] <= elements in a[partitionPoint+1:right]

partitionValue = a[(left+right)/2] // any element of the array a
lf = left, rt = right;

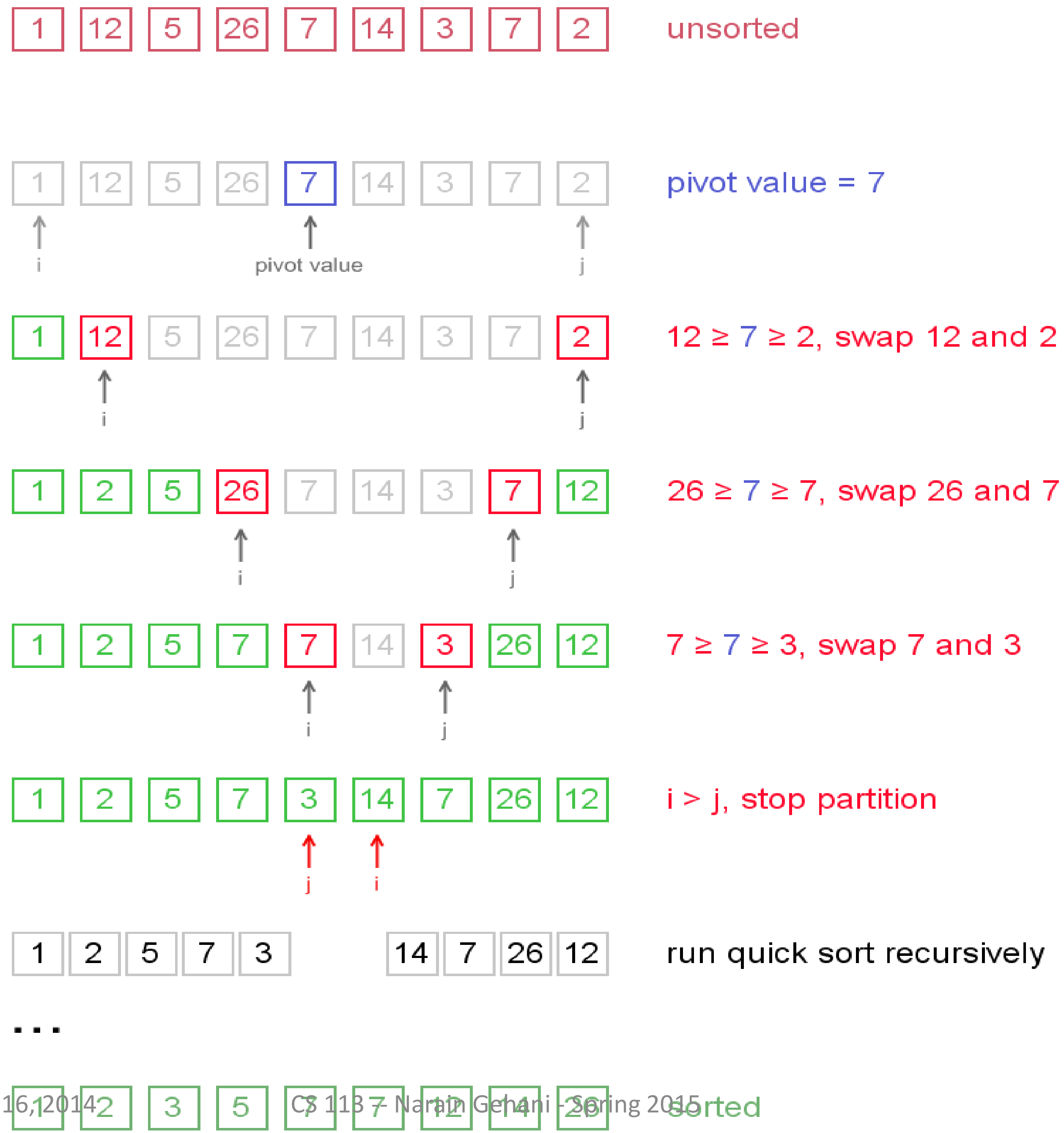
left partition must contain elements <= partitionValue

right partition must contain elements >= partitionValue

Initially, the left partition a[left:lf-1] is empty since lf == left, meaning lf-1 < left
Similarly the right partition a[rt+1: right] is empty since rt == right

```
while (lf < rt)  {  //partitions do not cover the whole array a)
   Extend the left partition (lf) by scanning until
       an element is found that should not be in the partition  (> partitionValue)
   Extend the right partition (rf) by scanning until
     an element is found that should not be in the partition (< partitionValue)
      Swap elements, lf++, rf--
}
return partition point rt
    (partitions are a[left:rt] and a [rt+1:right])
```

Partition image from the web



Quicksort

```
public static void quickSort(int a[], int low, int high) {
                               //sorts array[low:high]

   if ((high - low) <= 0)  // zero or one elements
      return;
   if ((high - low) == 1)  {// two elements

      if (a[low] > a[high]) {  // swap them
         int save = a[low];
         a[low] = a[high];
         a[high] = save;
      }
      return;
   }
   int partitionPoint = partition(a, low, high);
   quickSort(a, low, partitionPoint);
   quickSort(a, partitionPoint+1, high);
}
```

Partition

```
private static int partition(int a[], int left, int right) {
          /* partitions a in two parts and returns
             a partitionPoint such that
             elements in a[left:partitionPoint] <=
                  elements in a[partitionPoint+1:right]
         */


   int temp;
   int partitionValue = a[(left+right)/2];
                 // any element of the array a
   int lf = left, rt = right;

   while (true) {  // extend the partitions
      while (a[lf] < partitionValue) lf++;
      while (a[rt] > partitionValue) rt--;

      if (lf < rt) { // swap & advance pointers
           //left partition will get a value <= partitionValue
           // right partition will get a value >= partitionValue
         temp = a[lf]; a[lf] = a[rt]; a[rt] = temp;
         lf++; rt--;
      } else
        break;
   }
   return rt;
}
```

MergeSort

```
public static void mergeSort(int a[],int left,int right)
```

if a has only one element then return

otherwise, divide a into two parts mergesort each part and merge the sorted part as follows

mid = (low+high)/2

mergeSort(a, left, mid)
mergeSort(a, mid+1, right)

merge a[left:mid] and a[mid+1:right] into temp[]
copy temp[] back to a[left:right]

Sorting Complexity

- Selection, Bubble, Insertion sorts are $O(n^2)$ --- order n^2
  - number of comparisons is proportional to n^2
  - for 1024 elements → proportional to 1024 x 1024 comparisons
- Mergesort and Quicksort are O(n log n) – order n log n
  - Typically implemented recursively – easier to do this though they can be implemented iteratively
  - number of comparisons is proportional to n log n
  - for 1024 elements → proportional to 1024 x 10 comparisons

Exceptions

Exceptions

- An *exception* is an object that describes an unusual or erroneous situation
- The Java API has a predefined set of exceptions that can occur during execution
- Exceptions are *thrown* (raised) in a program
- An exception *thrown* can be dealt in one of three ways:
  - ignored
  - handled where it occurs
  - handled in an another place in the program

Exception Handling

- If an exception is ignored (not handled/caught) by the program, the program will terminate and produce an appropriate message

- The message includes a *call stack trace* that:
  – indicates the line on which the exception occurred
  – shows the method call trail that lead to the attempted execution of the offending line

Handling Exceptions

- Exceptions that might be raised when executing some code are handled by surrounding the code with a `try` statement

```
try {
  statements
}
catch (exception-type_1 variable_1)  {
  …
}
…
catch (exception-type_n variable_n) {
  …
}
[ finally { … } ]
```

Handling Exceptions (contd.)

- When an exception is thrown by the code in the `try` block, it is handled by the `catch` clause for that exception (if specified)
- The `finally` clause is optional and is always executed – exception or no exception

Throwing Exceptions

- Exceptions can be thrown by
  - by the Java run-time system or
  - explicitly by the program using the `throw` statement

Examples of Exception Propagation

```
import java.io.*;  // for class File
import java.util.*; // class Scanner

public class  ReverseList {
   public static void main(String args[]) throws IOException {
         // must propagate (pass on) the IO Exception
         // FileNotFoundException or catch it
         // (could happen when opening file
         // by creating a new Scanner object)

      // SET UP FILE FROM WHICH TO READ DATA
      if (args.length == 0) {
        System.out.println("Please supply data file" +
                           " as command-line argument");
         System.exit(0);
      }

      File inputDataFile = new File(args[0]);
      Scanner inputFile = new Scanner(inputDataFile);
       …
   }
}
```

Example of Exception Handling HelloCustomException.java

```
class HelloCustomException {
   public static void main(String[] args) {
      try {
         System.out.println("Hello " + args[0]
                                   + ", Good Morning!" );
      }
      catch (ArrayIndexOutOfBoundsException e) {
         System.err.println("Sorry, you must specify\n" +
                                "the person to be greeted\n" +
                                "as the command argument!");
         System.err.println("\nException " + e);
         System.exit(0);
      }
   }
}
```

- To ensure that referring to `args[0]` does not crash the program if no argument is supplied,
  - handle the exception or
  - check to make sure `args.length` is 1 before referring to `args[0]`

User Defined Exception & Throwing Exceptions

- We had seen examples of user-defined in matrix multiplication program
- We also saw how exceptions are thrown, i.e.,

`throw new` *UserDefinedException* (*arguments*)

Exception `MismatchMatrixBounds`

- Exception `MismatchMatrixBounds` is user-defined
- User-defined exceptions are derived from class `Exception`

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

Class Matrix
Throwing an exception

```
public Matrix multiply(Matrix y) throws MismatchMatrixBounds {
  Matrix x = this;  //easier to think x and y
  int m = x.mat.length;
  int n = 0;
  if (x.mat[0].length != y.mat.length)
   throw new MismatchMatrixBounds(
    "Matrix Bounds not satisfied for Multiplication");
  else
    n = y.mat.length;
  int p = y.mat[0].length;

      …
}
```

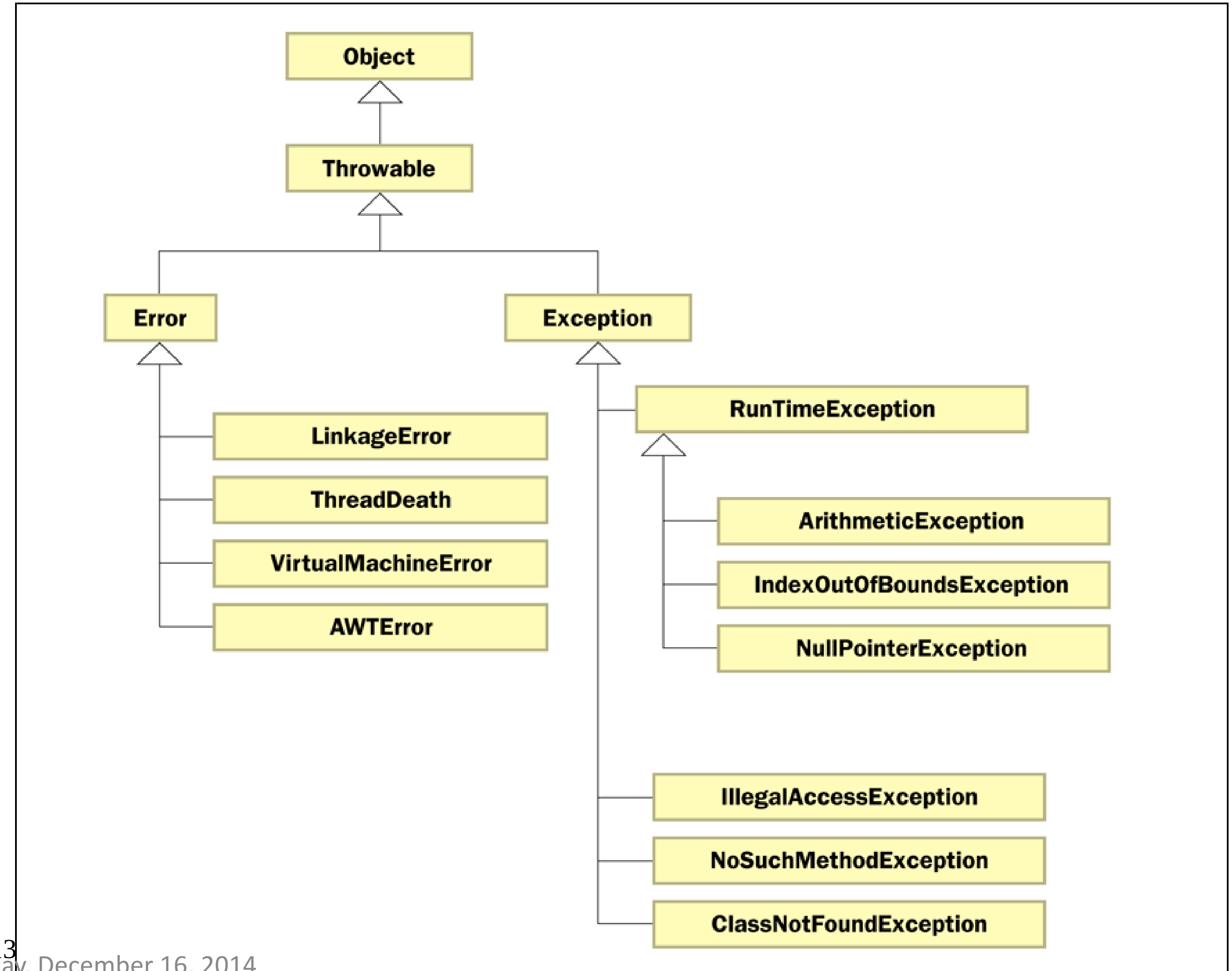
Not Handling Exceptions – Propagating them up

```
…
public class  TestMultiply {
   public static void main(String args[])
            throws IOException, MismatchMatrixBounds {
    …
      File inputDataFile = new File(args[0]);
      Scanner inputFile = new Scanner(inputDataFile);
    …
      // COMPUTE MULTIPLY
         Matrix z = x.multiply(y);
    …
}
```

The `Exception` Class Hierarchy

- Exception classes in the Java API are related by inheritance, forming an exception class hierarchy
- All exception classes are descendants of class `Throwable`
- A programmer can define an exception by extending the `Exception` class or one of its descendants

The Exception Class Hierarchy

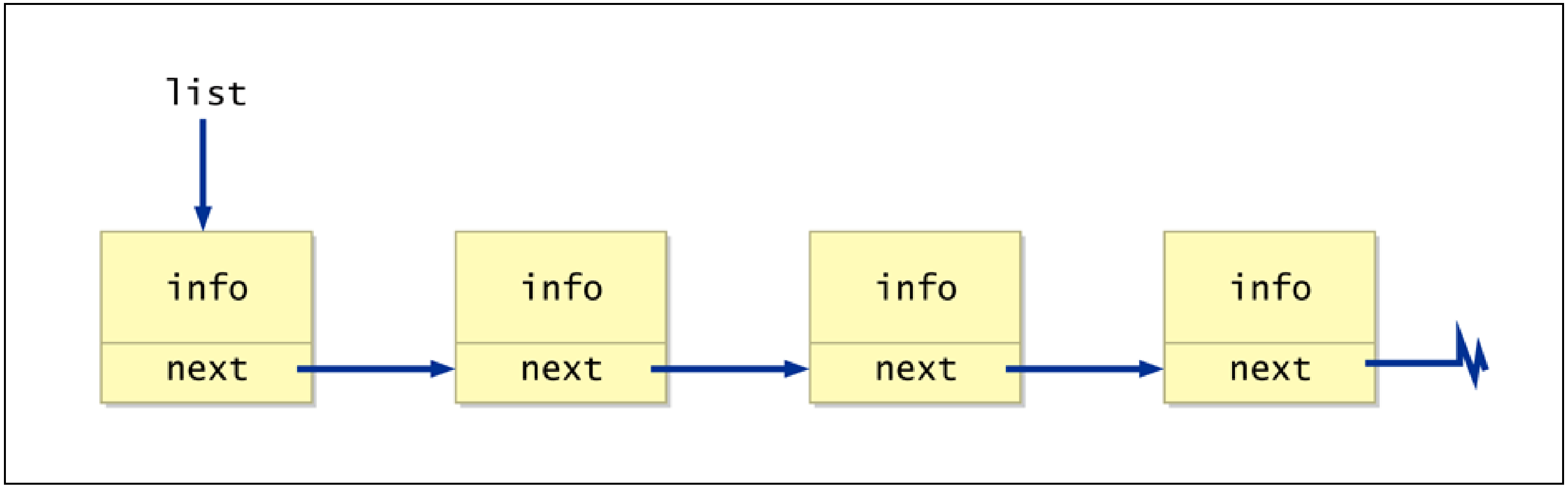


Even More Data Structures

Singly-Linked Lists

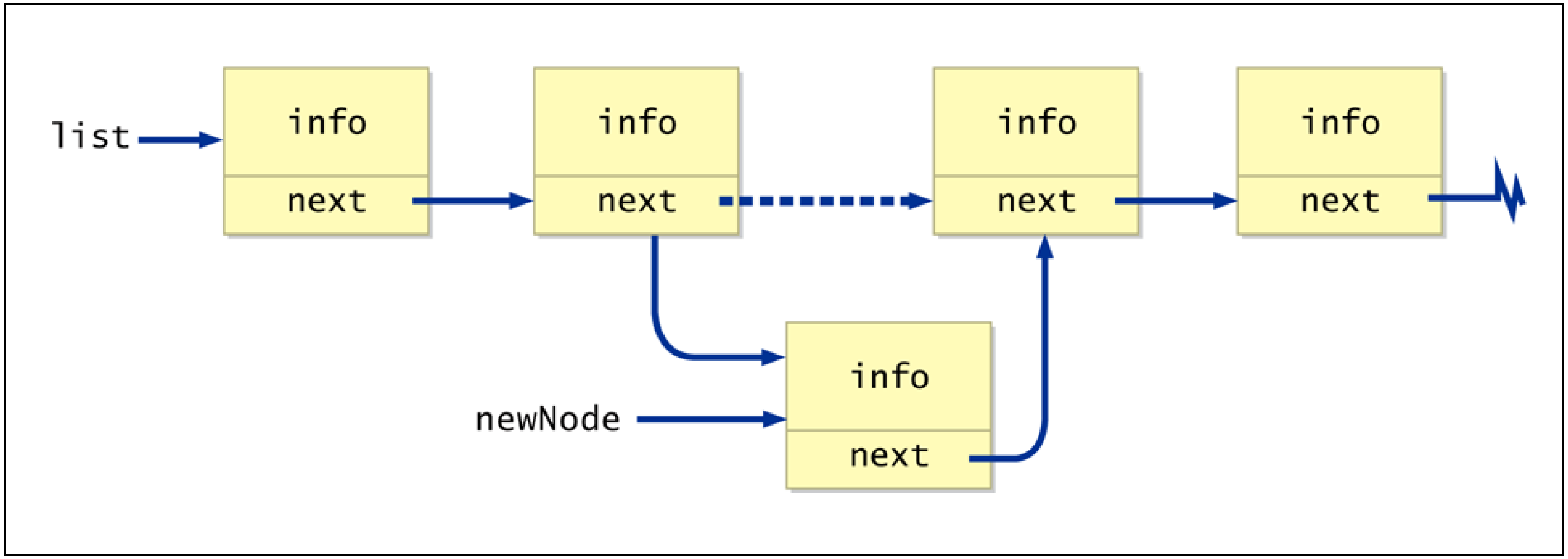
- Array elements can be accessed "randomly" using their index.
  - New elements cannot be inserted or deleted without shifting other elements
  - Typically, arrays have a bounded size.
- Lists do not have these issues but they support only sequential access.
- Each list element has two parts
  - one or more variables to store data
  - a reference to point to the next element

Singly-linked List



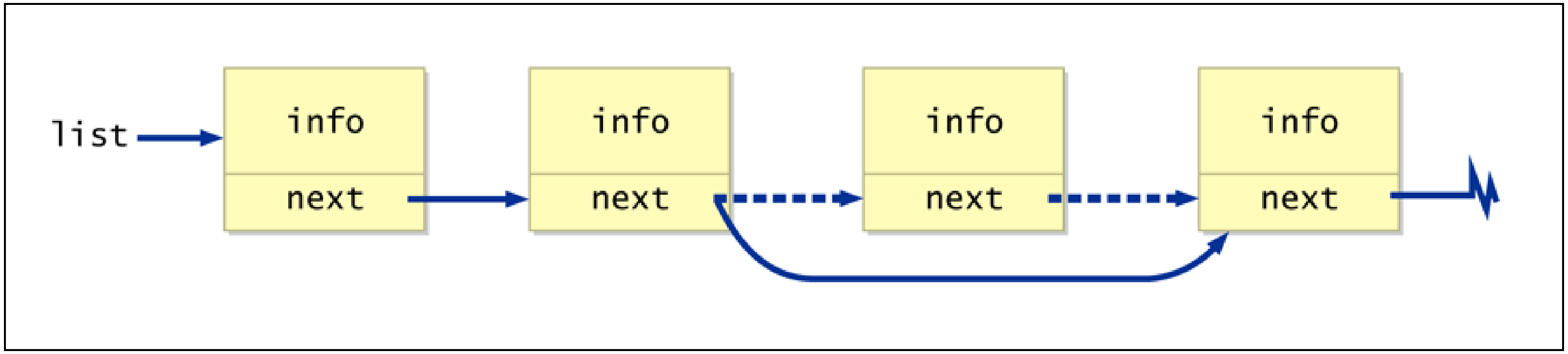
Inserting an Element

- A element can be inserted into a linked list with a few pointer changes:



Deleting an Element

- Likewise, a element can be removed from a linked list by changing the `next` pointer of the preceding node:



Linked List

- We will define a `list` class with operations `addfirst()`,`inList()`,`delete()`,and `toString()`.
- For simplicity our add operation is designed to add to the front of the list

Linked List Element

```
// For a change, we will allow direct
// access to the ListElement variables..
// Coding is  a bit easier

public class ListElement {
   int info;
   ListElement next;
   public ListElement(int x) {
      info = x;
      next = null;
  }
}
```

Linked List

```
public class List {
   ListElement head;
   public List() {
      head = null; // empty list
   }
   public void addFirst(int x) {
      ListElement newItem = new ListElement(x);
      if (head == null) {
         head = newItem;      }
      else {
          newItem.next = head;
          head = newItem;
      }
   }
   public boolean inList(int x) {
      ListElement itemRef = head;
      while (itemRef != null) {
         if (itemRef.info == x)
            return true;
         itemRef = itemRef.next;
      }
      return false;
   }
```

continued on next slide

Linked List (contd.)

```
public void delete(int x) {
      if (head == null) return;
      if (head.info == x) {head = head.next; return;}
      ListElement itemRef = head;
      while(itemRef.next != null) {
         if (itemRef.next.info == x) {
            itemRef.next = itemRef.next.next;
                                          // delete element
            return;
         }
         itemRef = itemRef.next;
      }
   }
   public String toString() {
      String s = "";
      ListElement itemRef = head;
      while (itemRef != null) {
         s = s + itemRef.info + " ";
         itemRef = itemRef.next;
      }
      return s;
   }
}
```

Binary Tree Definition

cslibrary.stanford.edu/110/BinaryTrees.html

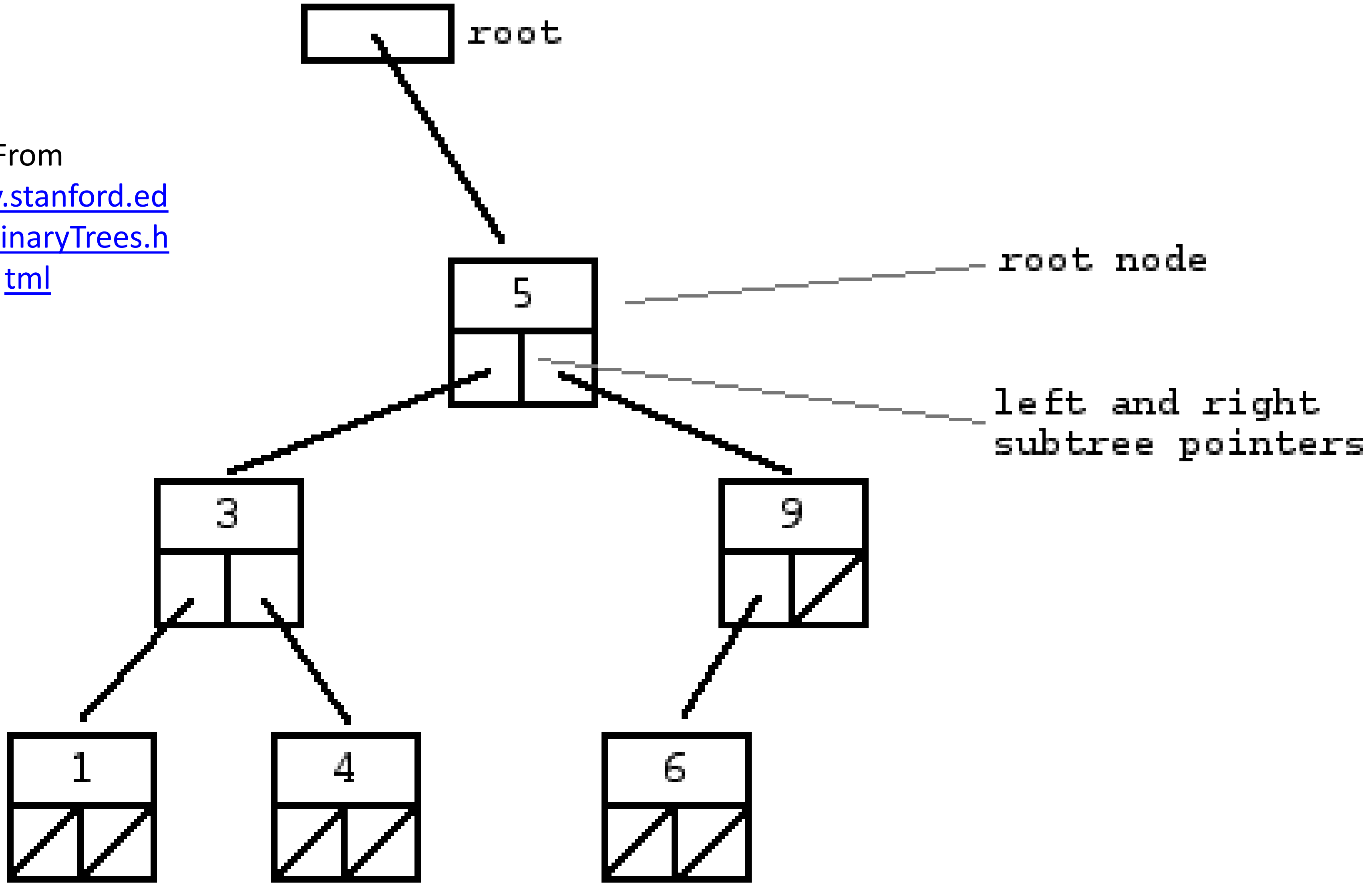
- A binary tree is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element.
  - The "root" pointer points to the topmost node in the tree.
  - The left and right pointers recursively point to smaller "subtrees" on either side.
  - A null pointer represents a binary tree with no elements -- the empty tree.

- The formal recursive definition is: A binary tree is
  - either empty (represented by a null pointer), or
  - made of a single node, where the left and right pointers (recursive definition ahead) each point to a binary tree

Ordered Binary Tree

- An ordered binary tree is a binary tree such that
  - the value of the left child (node) is less than that of the parent
  - the value of the right child (node) is greater than of the parent
- We will write a program to implement an ordered binary tree (we will call it simply binary tree omitting ordered)

From cslibrary.stanford.edu/110/BinaryTrees.html



Binary Tree

```
public class binaryTree {
   private String name;
   private binaryTree left, right;

   public binaryTree(String name) {

       this.name = name;
       left = right = null;

   }

   public binaryTree() {
       this.name = null;
       left = right = null;
   }
```

continued on next slide

Binary Tree (contd.)

```
public void addName(String name) {
    if (this.name == null) {
        this.name = name; return;
    }
    int compareResult =
            this.name.compareToIgnoreCase(name);

    if (compareResult <= 0){
        if (right == null)
           right = new binaryTree(name);
        else
           right.addName(name);
    }
    else {
        if (left == null)
            left = new binaryTree(name);
        else
            left.addName(name);
    }
```

continued on next slide

Binary Tree (contd.)

```
public boolean findName(String name) {

      int compareResult =
                this.name.compareToIgnoreCase(name);
      if (compareResult == 0)
         return true;
      else if (compareResult < 0) {
              if (right == null)
                 return false;
              return right.findName(name);
           }
           else {
              if (left == null)
                 return false;
              return left.findName(name);
           }
   }
```

continued on next slide

Binary Tree (contd.)

```
public void printInOrder() {
     // prints in sorted order
      if (left != null)
         left.printInOrder();
      System.out.println("name = " + name);
      if (right != null)
         right.printInOrder();
   }
}
```

Binary Tree Driver

```
import java.io.*;  // for class File
import java.util.*; // class Scanner
public class  binaryTreeTester {
   public static void main(String args[]) throws IOException {
      String name, searchName;
      binaryTree bTree = new binaryTree();

// READ DATA FROM FILE
      if (args.length == 0) {
         System.out.println(
               "Please supply data file as command-line argument");
         System.exit(0);
      }
      File inputDataFile = new File(args[0]);
      Scanner inputFile = new Scanner(inputDataFile);

      while(inputFile.hasNextLine()) {
         name = inputFile.nextLine();
         System.out.println("Input name = " + name);
         bTree.addName(name);
      }

      if (bTree != null) bTree.printInOrder();
```

continued on next slide

Binary Tree Driver (contd.)

```
//search for a name
  Scanner scan = new Scanner(System.in);
      System.out.print("Enter name for search: ");
      searchName = scan.nextLine();
      System.out.println("Search name = " +
                         searchName);

      if (bTree.findName(searchName))
         System.out.println(searchName +
                 " is in the binary Tree");
      else
         System.out.println(searchName +
                 " is NOT in the binary Tree");
   }
}
```

Binary Tree Input

Input File

```
Joan
Narain
Mark
Serena
Jim
Harry
Josh
Charles
```

Search name = **Serena**

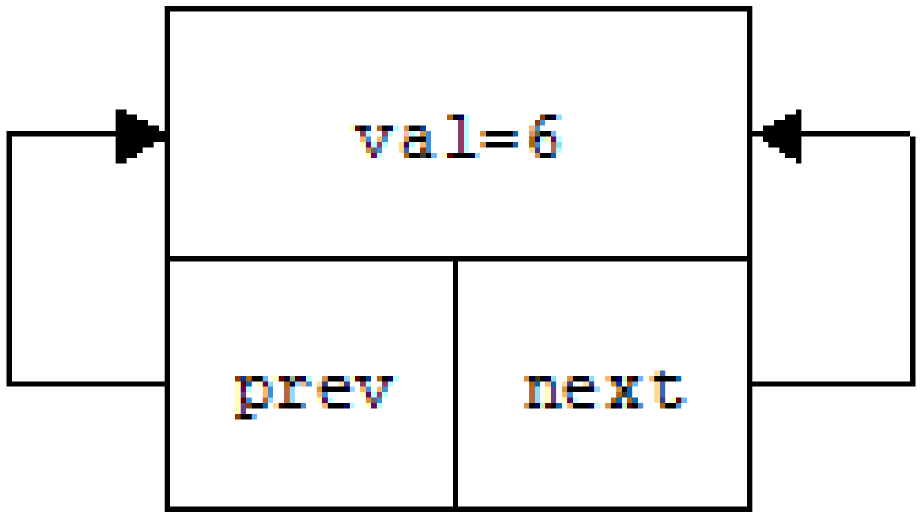
Serena is in the binary Tree

Circular Doubly-Linked List - Example

- Circular doubly-linked lists vs. singly linked lists
  - require more storage (2 references vs. 1)
  - easy traversal in either direction (one reference for each direction)
  - can go round the list starting from any point
  - easy insertions and deletions

Circular Doubly-Linked List (contd.)

- To give you a flavor of what a circular doubly-linked list looks like, here is one with a single element with the value 6

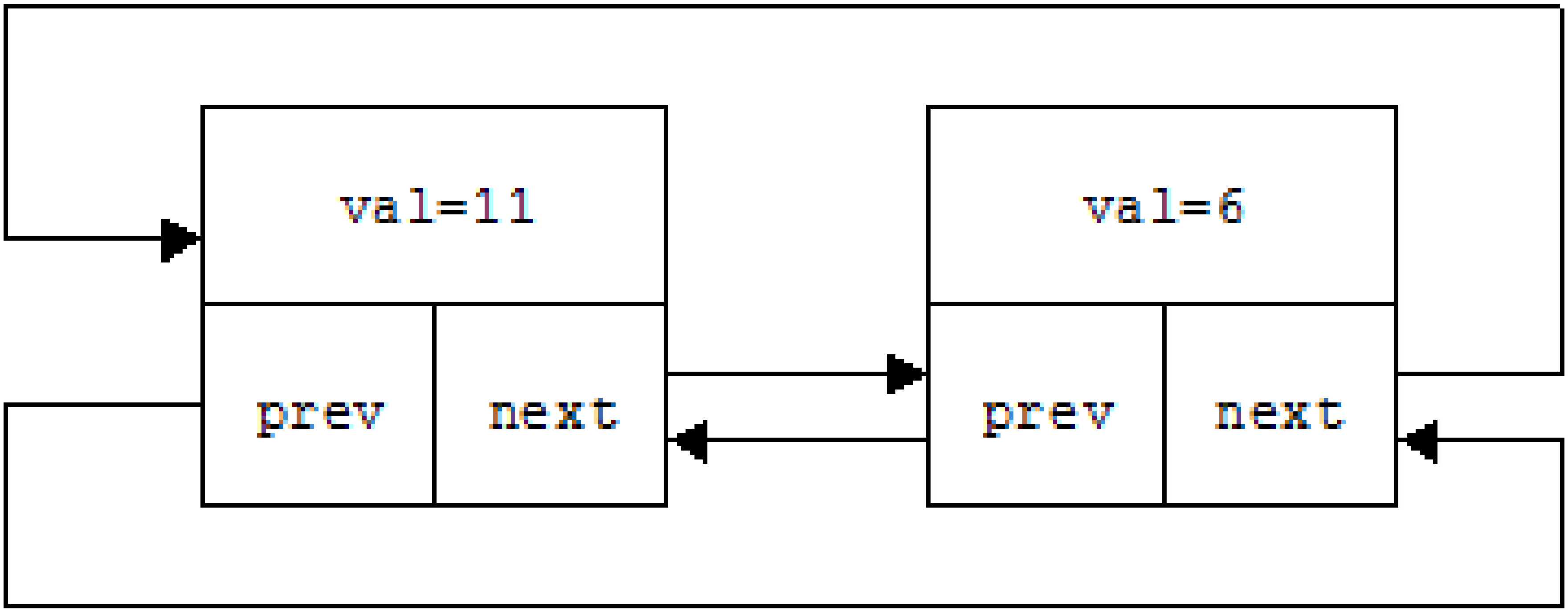


Circular Doubly-Linked List (contd.)

- Variables `prev` and `next` refer to the list element containing them because we are defining a circular list and there are no other elements.
- If we now add another element, say with the value 11, before the above list element, then the list will look like:

Circular Doubly-Linked List (contd.)

- 2 nodes

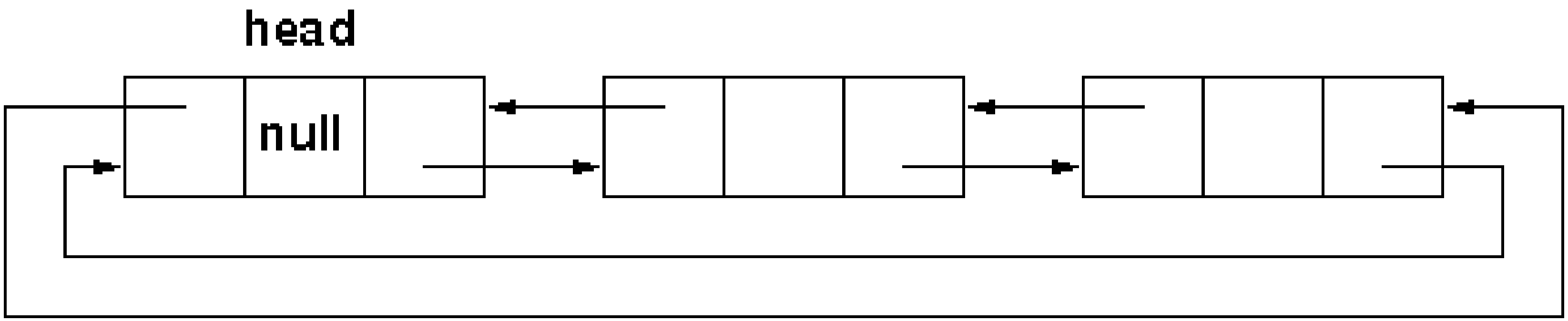


Circular Doubly-Linked List - Example

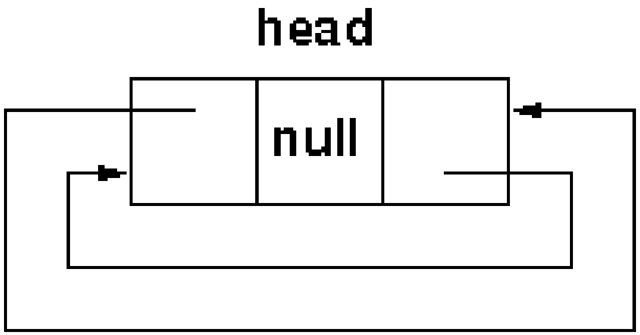
- Elements of our circular doubly-linked list will be of the class type called `dlList`.
- Each object of type dlList will contain the following three "field" variables:
  - val: stores the int list item value;
  - prev: points to the previous element in the list; and
  - next: points to the next element in the list.
- Although `dlList` objects will store `int` values (in variable `val`), `dlList` can be trivially modified (by changing the type of `val`) to store other types of data values.

DL List with a head node

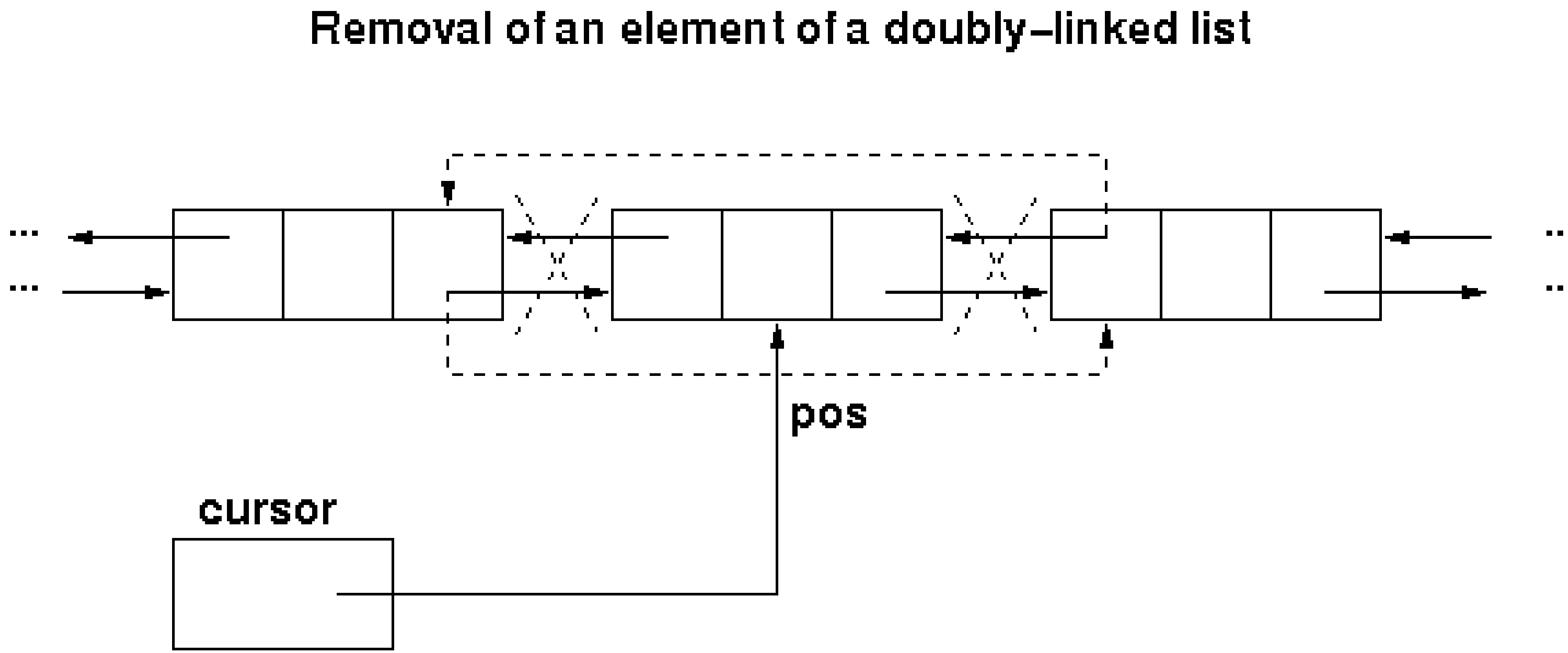
A doubly-linked list with 2 elements



A doubly-linked list with no elements



Removing an Element



Circular Doubly-Linked List (contd.) Constructors & Methods

- Operations implemented

```
void addFirst(int v)
int getFirst() throws EmptyList
        //get & remove from list
void printForward()
void printReverse()
void delete(int v)
```

- Other operations can be added, adding before or after an element

Circular Doubly-Linked List Element

```
public class dlListElement {
    int val;
    dlListElement prev, next;
    public dlListElement(int v) {
        val = v;
        prev = next = null;
    }
}
```

Circular Doubly-Linked List

```
public class dlList {
   dlListElement head;

   public dlList() {
      head = null;
   }
   public void addFirst(int v) {
      dlListElement newItem = new dlListElement(v);

      if (head == null) {
         head = newItem;
         head.next = head.prev = head;
                    // first element points to itself
      } else {
         newItem.prev = head;
         newItem.next = head.next;
         head.next.prev = newItem;
         head.next = newItem;
      }
   }
```

continued on next slide

Circular Doubly-Linked List (contd.)

```
public int getFirst() throws EmptyList {
         // get & remove the first item
         // from the list and delete it

   if (head == null) // list empty
      throw new EmptyList("getFirst No items");
   else {
      int v = head.val;
      head.next.prev = head.prev;
      head.prev.next = head.next;
      head = head.next;
      return v;
   }
}
```

continued on next slide

Circular Doubly-Linked List (contd.)

```
public void printForward() {
    if (head == null)
        return;
    dlListElement p = head;
    do  {  // start with head pointer
           // stop when u reach head
        System.out.print(p.val+ "  ");
        p = p.next;
    } while (p != head);
    System.out.println();
 }
```

continued on next slide

Circular Doubly-Linked List (contd.)

```
public void delete(int v) {
   if (head == null)
       return;
   if ((head.val == v) && head == head.next) {
                          //one element in list
      head = null; return;
   }
   dlListElement p = head.next;

   while (p.val != v) {
      if (p == head )return;
      p = p.next;
   }
   p.next.prev = p.prev;
   p.prev.next = p.next;

   if (p == head) head = p.next;
}
```

`ArrayList` Class

- `ArrayList` class provides the capabilities of both an array and a list
  - there is not need to specify the size of an ArrayList object
- It is a generic type which means that you can use it for objects of any object type.
- But not primitive types – for this one must use their wrapper classes.

```
public class ArrayList<E> { … }
```

- Type E must be specified when instantiating an ArrayList object, e.g.,

```
ArrayList<String> Countries = new ArrayList<String>;
```

`ArrayList` Class (Some Operations)

- **`ArrayList()`**: Constructor that creates an empty list
- **`boolean add(E obj)`**: append object `obj` (of type E) to the end of the list
- **`void add(int index, E obj)`**: inserts the object `obj` at the specified position in the list
- **`void clear()`**: Removes all elements from the list.
- **`boolean contains(Object obj)`**: Returns true if the list contains object `obj`
- **`E get(int index)`**: Returns the element at the specified position in the list
- **`indexOf(Object obj)`**: Returns the index of the first occurence of obj in the list (testing for equality using the `equals` method)
- **`boolean isEmpty()`**: returns true or false depending upon whether the list is empty or not
- **`E remove(int index)`**: removes the object at position index and returns it.
- **`int size()`**: returns the number of elements in the list

For-each Loops

- The for-each loop simplifies the processing of items in an iterator class such as `ArrayList`

- Consider the declaration

```
ArrayList<String> Countries =
            new ArrayList<String>;
```

- The following loop prints each country:

```
for (String Country: Countries)
   System.out.println(Country);
```

For-each Loops

- A for-each loop can be used on any object that implements the `Iterable` interface
- It eliminates the need to retrieve an iterator and call the `hasNext` and `next` methods explicitly

End

Creating a Java Archive

- Java archive files (.jar) allow you to store several files together in a single archive file.
  This is very handy to submit homework assignments.

  To create the .jar file, make sure that the project is open.
  Then, click on Project, and scroll down to Create Jar File for Project...
  A window will pop up.
  Select the boxes for Project File and Sources.
  Then click on Next.
  A new window will pop up.
  After determining the directory and name for the .jar file, select Create Jar.

  You can test if the .jar file was created properly by opening it.
  Before that, close your project and all files.
  Copy the .jar file to some other directory, and then select Project and scroll down to Jar / Zip Extractor.
  In the next window, select File \ Open Jar or Zip File and locate the .jar file.
  Select the file and open it.
  The list of files inside the .jar file will appear.
  Now click on File \ Extact Files and select the directory where you want to store the files.

  Before submitting a .jar file as the result of your homework, copy it to a separate directory, extract it inside jGRASP, and make sure that you can compile all the .java files and that you can run the program properly. Then it is time to submit the assignment.