Lectures 1-3

Java Crash Course

Introducing Java Generics

Reading and outline

- Chapter 1: Java Fundamentals, Pages 3-64

  Slides 3-76
- Chapter 4: Classes and Interfaces,
  Pages 193-206

  Slides 77-112
- Chapter 5: Generics, Pages 270-278

  Slides 113-130

Review of Java Fundamentals

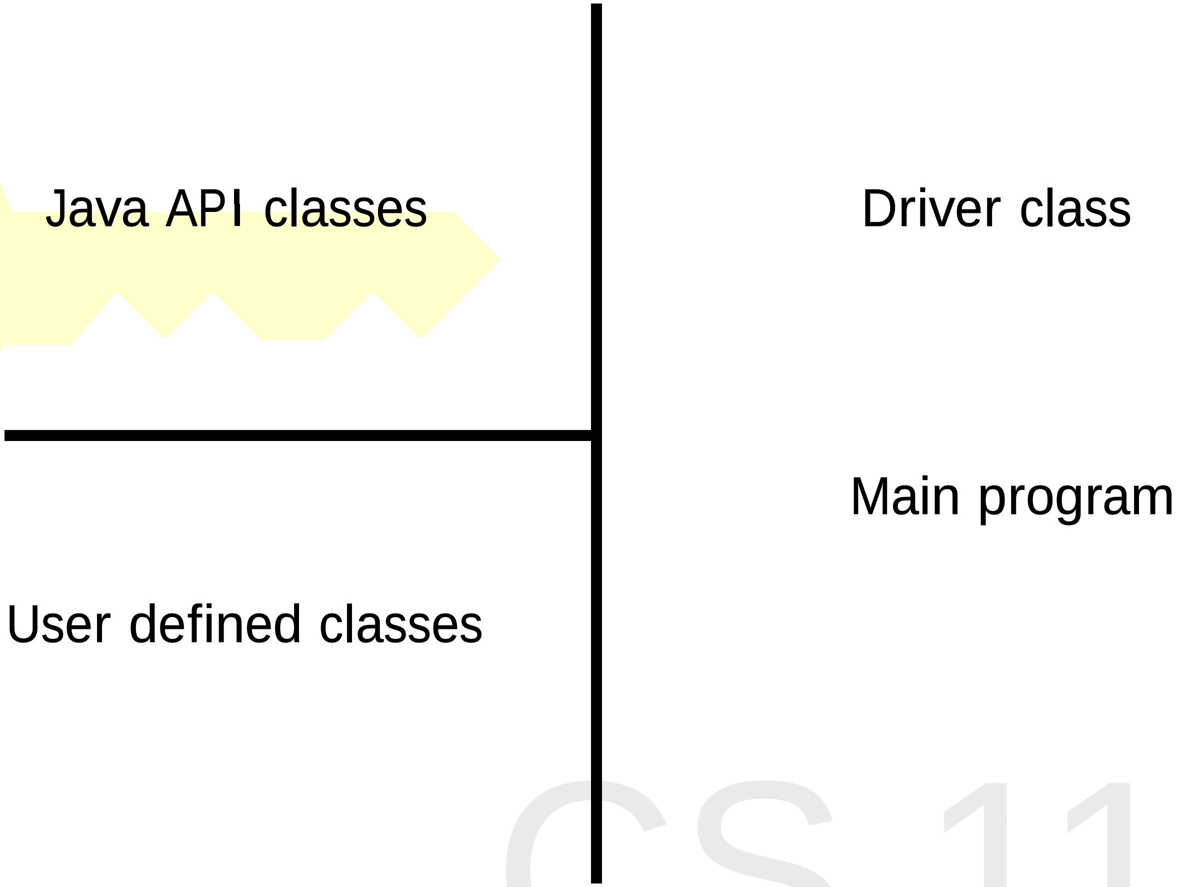
1.1 Program structures
1.2 Language basics
1.3 Selection statements
1.4 Iteration statements
1.5 Useful Java classes
1.6 Java Exceptions
1.7-8 Text and File I/O

Java 1.5 Features

Program Structure

- A typical Java program consists of
  - User written classes
  - Java Application Programming Interface (API) classes
- The program should have one class with a main method
- Java program basic elements:
  - Packages
  - Classes
  - Data fields
  - Methods

The vertical and horizontal lines



Java API classes

Driver class

Main program

User defined classes

Hello world!

```
public class Prog {
    static public void main(String[] args) {
        System.out.println("Hello world!");    // S.o.p. or simply Sop
    } //end main
} // end Prog
```

- This java program must be named as Prog.java
- To compile this program, type:

  javac Prog.java

- You should be:
  - In the window of the “command prompt" of your PC
  - Or on the sunfire system
  - And in the directory where the Prog.java file is
- Successful compilation will produce Prog.class
- To run the program, enter

  java Prog

Using command line arguments

```
public class Prog {
    static public void main(String[] args) {
        System.out.println("Hello " + args[0] + ", I am " + args[1] );
    }
}
```

- Enter:

```
java Prog John Ang
```

- Output:

```
Hello John, I am Ang
```

Packages

- Provide a mechanism for grouping related classes
- Java assumes all classes in a particular package are contained in the same directory
- Java API consists of many predefined packages
- import statement
  - Allows us to “use” classes contained in packages
  - Q: Must we import before use?
- Usually we need to
  - import java.util.*;
  - import java.io.*;
- Package java.lang is imported implicitly

Primitive and Reference Types

- Java has eight primitive data types:
  - byte, short, int, long, float, double, char, boolean
- All non-primitive data types are called reference or class types
- For example in

  ```
  String greeting;
  ```

  the String variable greeting is a reference variable
- All reference variables are pointers initialized with null (0) if they are not initialized with specific values. Accessing them causes a NullPointerException. So right now, greeting is a null pointer.

  ```
  greeting = "Hello"; // greeting now points to "Hello"
  ```

Aliases

- Two variables referencing the same instance of a class are aliases

e.g. in

```
String shouting = greeting;
```

the String variable shouting is an alias of greeting

Classes (1)

- A class definition includes:
  - The keyword class
  - An optional extends clause
  - An optional implements clause
  - And the class body

- Every Java class is a subclass of either:
  - another Java class when an extends clause is used, or
  - Object class when an extends clause is not used

- Place each class's definition in a separate file, if possible (Not when you have to submit to courseMarker or doing sit-in lab)

Classes (2)

- Each Java class defines a new data type
- A class specifies data (variables or constants) and methods available for instances of the class. Both data and methods are class members.

class = data + methods

- An object in Java is an instance of a class
  - new operator: to create an object from a class
  - Java recycles memory space taken up by objects that a program no longer references – garbage collection
- A class’s data fields should be private

Membership categories of a class

- Public members can be accessed without any restriction through any object containing the members
- Private members can be accessed by methods of the class
  - ! It can be accessed from sibling objects
    - E.g., s1 and s2 are two Student objects, and we may call s1.compareTo(s2):

```
int compareTo(Student s) {
        if (this.CAP < s.CAP) return -1;
        if (this.CAP == s.CAP) return 0;
        return 1; }
```

- Protected members can be accessed by methods of both the class and any derived class

Accessing Members: The Dot (select) operator

- Let obj be an object with members data d and function f()
- To use obj 's members, the dot operator (or select operator) is used:

```
obj.d        // to access the data
obj.f()      // to call the function
```

- If the members are static members of a class C, then we use C.d and C.f()
- Example:

```
Math.sqrt(…)
System.out.println()
```

Methods (1)

Each method performs one well-defined task

- Valued method
  - Returns a value
  - Body must contain a return expression
- Void method
  - Does not return a value
  - Body must not contain return expression;

```
return;   // is valid
```

Methods (2)

- Syntax of a method declaration

```
access-modifier use-modifiers return-type
method-name (list of variables) {
    method-body
}
```

  - Variables appeared in method header are formal parameters
  - Variables that the method is called to work on are arguments or actual parameters

- Arguments are passed by value, i.e., whatever value stored in an argument will be passed to the corresponding formal parameter
- As a result, for arguments that are objects and arrays, a reference is copied to the corresponding formal parameters

Constructors

- A constructor looks like a method. It has the same name as the class but no return type
- A default constructor has no parameter
- A constructor is executed when an object is to be created
- ! If a class does not specify a constructor, it will be given a default constructor by the compiler
- When a class has multiple constructors, best practices show that one of them should be properly implemented and called by other constructors
- To invoke a constructor, use the new operator:

```
Circle c = new Circle(5);
Circle(4);     // invalid, as it cannot be called as a method
```

Accessors, Mutators, and Facilitators

- Accessor methods
  - to find out the values of the private member variables
- Mutator methods
  - to change the values of the private member variables
- Facilitator methods
  - to do some computation without modifying any member variable

Example: class Circle

```
public class Circle {
   private radius;
   public Circle() { this (1);}                    // default constructor
   public Circle (int r) { radius = r; }           // another constructor
   public getRadius() { return radius;}            // accessor
   public setRadius(int r) {radius = r;}           // mutator
   public double area() { return …;}               // facilitators
   public double perimeter() { return …;}
}
```

Review of Java Fundamentals

- 1.1 Program structures
- 1.2 Language basics
- 1.3 Selection statements
- 1.4 Iteration statements
- 1.5 Useful Java classes << Here
- 1.6 Java Exceptions
- 1.7-8 Text and File I/O

- Java 1.5 Features

Useful Java Classes: String (1)

- Declaration examples:

```
String title;                        // just a null pointer
String title1 = “Walls and Mirrors”;
```

- Assignment example:

```
title = “Walls and Mirrors”;
```

- String length example:

```
title.length();
```

- Referencing a single character

```
title.charAt(0);
```

- Comparing strings

```
s1.compareTo(s2);                    // Should not use s1 == s2, why?
                                     // <0, s1 < s2
                                     // =0, s1 == s2
                                     // >0, s1 > s2
```

Useful Java Classes: String (2)

Concatenation “cat” example:

```
String monthName = "December";
int day = 31;
int year = 07;
String date = monthName + " " + day + ", 20" + year;
        // date = "December 31, 2007"?
```

Q: What do we get for the following statements?

```
String date1 = monthName + day + year;
String date2 = monthName + (day + year);
```

- A: December3107   December38
- B: December38   December3107
- C: December3107   December3107

Class StringBuilder

Once a String object is created, it cannot be changed.

All methods modifying a String object return new String objects.

StringBuilder allows a string object to be modified without creating new string objects.

```
StringBuilder msg = new StringBuilder(“Rover");
msg.setCharAt(0,'R');                    // Rover
msg.append(", roll over!");              // Rover, roll over!
msg.insert(7, "Rover, ");                // Rover, Rover, roll over!
msg.delete(0,7);                         // Rover, roll over! range is [0,7)
msg.replace(7,16, "come here");          // Rover, come here!
String finalMsg = msg.toString();        // String from StringBuilder
```

Drawbacks of arrays

- The length of an array must be declared when the array is created, and cannot be changed
- If an array is not full, we must keep track of the last position currently in use
- Inserting a new item into an array necessitates pushing down the elements at and below the insertion point
- Similarly, deleting an element necessitates pulling elements up to fill the gap.

Q: Is there an alternative?

For an array a[], use a.length.
For an object (such as a String) s, use method call s.length(), or s.size()

Class Vector

- The Vector class implements a growable array of objects.
- Like an array, it contains components that can be accessed using an integer index.
- However, the size of a Vector can grow or shrink as needed to accommodate adding or removing items after the Vector has been created.

boolean	**add**(E o) Appends the specified element to the end of this Vector.
void	**add**(int index, E element) Inserts the specified element at the specified position in this Vector.
int	**capacity**() Returns the current capacity of this vector.
void	**clear**() Removes all of the elements from this Vector.
boolean	**contains**(Object elem) Tests if the specified object is a component in this vector.
E	**elementAt**(int index) Returns the component at the specified index.
int	**indexOf**(Object elem) Searches for the first occurence of the given argument
boolean	**isEmpty**() Tests if this vector has no components.
E	**remove**(int index) Removes the element at the specified position in this Vector.
boolean	**remove**(Object o) Removes the first occurrence of the specified element in this Vector
E	**setElementAt**(E obj, int index) Sets the component at the specified `index` to be the specified object.
int	**size**() Returns the number of components in this vector.

Example: Vector and enhanced for

```
import java.util.*;
public class MyVector {
  public static void main(String[] args) {
    Vector<String> band = new Vector<String>(10);
    band.add("Paul");
    band.add("Pete");
    band.add("John");
    band.add("George");
    System.out.println(band);
    band.remove("Pete");
    System.out.println(band);
    System.out.println("At index 1: " + band.elementAt(1));
    band.add(2, "Ringo");
    for (String s : band) System.out.print (s + " "); // enhanced for loop
    System.out.print ( "size: " + band.size());
    System.out.println ( "capacity: " + band.capacity());
  } }
```

```
[Paul, Pete, John, George]
[Paul, John, George]
At index 1: John
Paul John Ringo George
size: 4 capacity: 10
```

1.6 Exceptions

1.1 Program structures
1.2 Language basics
1.3 Selection statements
1.4 Iteration statements
1.5 Useful Java classes
1.6 Java Exceptions
1.7-8 Text and File I/O

What Is An Exception?
Why Is Exception Handling Important?
Types of Exceptions
- Throwing Exceptions
- Ordinary Exceptions
- Runtime Exceptions

finally clause – Forced Execution
User-defined exception classes

Example: Divide by zero

```
public static void main(String[] args) {
   int a, b;
   Scanner sc = new Scanner(System.in);
   System.out.print( "Enter a and b: ");
   a = sc.nextInt();
   b = sc.nextInt();
   int c = a / b;
   System.out.println("c is " + c );
}
```

As the program does not indicate how to deal with division by zero, the system will take over the control and crash the program after the message is printed.

```
Enter a and b: 3    0

Exception in thread "main" java.lang.ArithmeticException: / by zero

at DivideByZero.main(DivideByZero.java:10)
```

What is An Exception?

- An exception is an event that disrupts the normal flow of a program. This event is usually described by an object.
- Exception handling is the process of detecting and responding to an exception
- An error is a specific type of exception that a program is usually unable to handle.

Why is it Important?

- It provides a means to separate exception-handling code from normal functioning code.
- It provides a way to organize and differentiate between different types of abnormal conditions.

Without exception

```
Scanner sc = new Scanner (System.in);
System.out.println("Enter number of donuts:");
int donutCount = sc.nextInt();
System.out.println("Enter number of glasses of milk:");
int milkCount = sc.nextInt();
if (milkCount < 1)
    System.out.println("No milk! Go buy some!");
else {
    double donutPerGlass = donutCount / (double)milkCount;
    System.out.println("You have " + donutPerGlass + " donuts for
        each glass of milk.");
    }
```

With exception, the yellow code can be re-written as:

Using exception

```
try {                             // normal situation in try block
    double donutPerGlass = donutCount / (double)milkCount;
    System.out.println("You have " + donutPerGlass
            + " donuts for each glass of milk.");
} // end try
catch (ArithmeticException e) {   // exception in catch block
     System.out.println(e.getMessage());
     System.out.println("Go buy some!");
} // end catch
```

Throwing an exception

- When an exception is detected by a program, it can be thrown with a *throw* statement.
- A *throw* statement can appear anywhere
- The code that deals with an exception is said to catch (handle, deal with) the exception.

Syntax for throwing an exception:

```
throw new ExceptionClass (stringArgument);
```

If there is no specific message to show, use

```
throw new ExceptionClass ();
```

Anatomy of exception handling

General layout for handling exceptions:

```
try {                          // try block
   statement (s);              // exceptions might be thrown
}                              // followed by one or more catch block
```

```
catch (ExceptionClass1  identifier) { // a catch block
   statement (s);              // Do something about the exception
} catch (ExceptionClass2  identifier) { // another catch block
   statement (s);              // Do something about the exception
}
```

```
finally {                      // finally block – for cleanup code
   statement (s);
}
```

Types of Exceptions

There are three types of exceptions:

- Ordinary Exceptions are exceptions that occur at predictable locations

  For example: *file not found* exception

- Runtime exceptions are exceptions whose location will be nailed down during runtime

  For example, a *null pointer* exception or a *divide by zero* exception

- Errors are exceptions that are catastrophic

  For example, *running out of memory*

Class Exception

When an exception thrown in a program is a descendant of the class Exception, the program must have code to handle (check) it, and it is called a checked exception.

Checked exceptions are ordinary exceptions:

- FileNotFoundException
- IOException
- Most user defined exceptions

Class RuntimeException

When an exception class is derived from RuntimeException class, it does not need to be caught in a catch block nor specified in a throw clause of a method.

Examples include:

- ArrayIndexOutOfBoundException
- NullPointerException
- ArithmeticException
- NoSuchElementException
- IndexOutOfBoundException
- ClassCastException
- UnsupportedOperationException

- Also known as unchecked exceptions
- RuntimeException is a subclass of Exception

Class Error

- Treated as an unchecked exception. It is beyond the program's control and the program has to be terminated.

  E.g.

  OutOfMemoryError

  VirtualMachineError

Catching more than one Exception

```
try {
    TroubleMaker1 ();                   // may throw an exception
    TroubleMaker2 ();                   // may throw another exception
    …
}
catch (DivideByZeroException e) { // more specific exception
  System.out.println (e.getMessage ());
}
catch (AnotherException e) {        // another specific exception
  System.out.println (e.getMessage ());
}
catch (Exception e) {               // general exception considered last
  e.printStackTrace();
}
```

class Throwable

- Throwable is the mother of all exception classes
- It is the common super class of Exception and Error
- If the most general exception is to be specified, use Throwable.

Forced Execution with finally

- To clean up or release some resources (such as closing files or release some structures to memory pool), a finally clause is provided.

```
try {
    int x = 100;
    for (int n = 10; n >= 0; n--)  System.out.println (x / n);
    return;
}
catch (ArithmeticException e) {System.out.println (e.getMessage ());}
finally { // Always executed even when a return statement is in try block
   System.out.println (“Can’t get around me!”);
}
```

- Note: "divide by zero" will throw ArithmeticException

Declaring Exception throwing

A method must declare which exceptions are thrown in its header if:

1. exceptions have been thrown in it;
2. exceptions might be thrown by methods it calls and it does not catch them.

```
public void thisIsTrouble1 () throws anException {
    ...
    throw new anException ();
}
```

```
public void thisIsTrouble2 () throws Exception1, Exception2 {
    method1();    // may throw Exception1
    method2();    // may throw Exception2
} // Exception1 and Exception2 are not handled by this method
```

User defined Exceptions (1)

```
class DivideByZeroException extends Exception {
  public DivideByZeroException () {
     super ("Divide By Zero Exception!");
  }
  public DivideByZeroException (String msg) {
     super (msg);
  }
}
```

User defined Exceptions (2)

```
public class DivideByZero1 {
public static void main(String[] args) {
   int a, b, c = 100;
   Scanner sc = new Scanner(System.in);
   System.out.println( "Enter a and b: ");
   a = sc.nextInt();
   b = sc.nextInt();
   try { c = divide(a, b); }
   catch (DivideByZeroException e) {System.out.println (e.getMessage ()); }
   System.out.println("c is " + c );
}
static int divide (int a, int b) throws DivideByZeroException{
  if (b == 0) throw new DivideByZeroException(“My own exception");
  return a/b;}
}
```

When b = 0,
the output is:

```
My own exception
c is 100
```

Assertions

An assertion is a statement of truth

For example, in

```
double sqrt(double x) {
assert x>0;
…
}
```

when sqrt() is called with a negative x, the program will terminate:

```
Exception in thread "main" java.lang.AssertionError
```

If we use assert x>0 : x;

then the error message printed when sqrt(-5.1) is executed will be:

```
Exception in thread "main" java.lang.AssertionError: -5.1
```

Assert statements will be executed only when they are enabled:

```
java –ea MyProgram
```

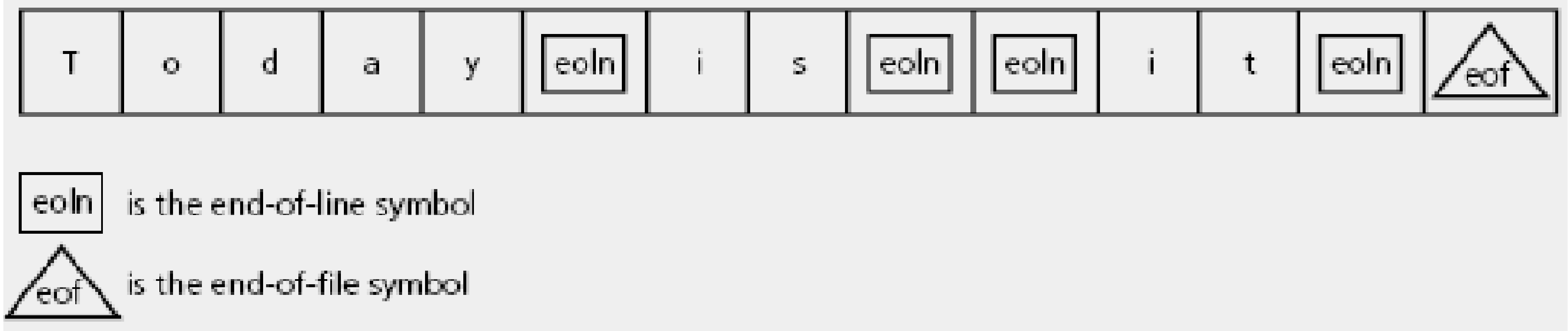
File Input and Output

- File
  - Sequence of components of the same type that resides in auxiliary storage
  - Can be large
  - Can exist after program exits
- Vs. arrays
  - Files grow as needed; arrays have a fixed size
  - Files provides both sequential and random access; arrays provide random access
- File types
  - Text and binary (general or nontext) files

Text Files

- Designed for people
  - Flexible and easy to use
  - Not efficient with respect to computer time and storage
- End-of-line symbol
  - Creates the illusion that a text file contains lines
- End-of-file symbol
  - Follows the last component in a file
- Scanner class can be used to read text files

Example



T	o	d	a	y	eoln	i	s	eoln	eoln	i	t	eoln	eof

eoln is the end-of-line symbol

eof is the end-of-file symbol

Today
is

it

FileReader and BufferedReader

- Open a stream from a file for read, use class FileReader
  - This may throw a FileNotFoundException
  - Good to handle exceptions with try…catch
- Stream is usually embedded within an instance of class BufferedReader which buffers characters so as to provide efficient reading
  - This class provides text processing capabilities such as readLine

Example: Reading from text file

```
BufferedReader input;             // starts as null pointer
String inputLine;
try {
   input = new BufferedReader(new FileReader("Ages.dat"));
   while ((inputLine = input.readLine()) != null) {        // eof?
     ...                          // process line of data
   }
}                                 // end try
catch (IOException e) {
   System.out.println(e);
   System.exit(1);                // I/O error, the program exits
}                                 // end catch
```

FileWriter and PrintWriter

- To output to a text file, need to open an output stream to the file
- Use class FileWriter to open an output stream
- For ease of output operation, an output stream is usually embedded within an instance of class PrintWriter
  - That provides methods print and println

Example: Output to text file

```
try {
    PrintWriter output = new PrintWriter(new FileWriter("Results.dat"));
    output.println("Results of the survey");
    output.println("Number of males: " + numMales);
    output.println("Number of females: " + numFemales);
    // other code and output appears here...
} // end try
catch (IOException e) {
    System.out.println(e);
    System.exit(1); // I/O error, the program exits
} // end catch
```

java Cat outfile infile1 infile2 ...

```
public class Cat {
  public static void main(String[] args) {
    PrintWriter out;
    int i;
    try {                                    // can generate an exception, need try
      out = new PrintWriter (new FileWriter (args[0] ));        // first file
      for (i = 1; i < args.length; ++i) {
         Scanner sc = new Scanner(new File( args[i] ));         // remaining files
         out.println("\n\t\tcontent of " + args[i] + " :\n");
         while (sc.hasNextLine()) {
           String line = sc.nextLine();
           out.println(line); }
         sc.close(); }
      out.close(); }
    catch (FileNotFoundException e) { … }                       // end catch
    catch (IOException e) { … }                                 // end catch
}}
```

Miscellaneous file methods

- Closing a file

```
myStream.close();
```

- Appending to a text file

```
PrintWriter ofStream = new PrintWriter(new FileWriter
   ("Results.dat", true));
```

Review of Java Fundamentals

- 1.1 Program structures
- 1.2 Language basics
- 1.3 Selection statements
- 1.4 Iteration statements
- 1.5 Useful Java classes
- 1.6 Java Exceptions
- 1.7-8 Text and File I/O

- Java 1.5 Features << Here

New features of Java 5

- Class Scanner (taught in CS1101)
- Auto-boxing / Unboxing
  - supports automatic conversion between primitive data types and the corresponding wrapper objects
- Simplified for Loop for traversing collections and arrays
- Definition of classes with Generic Types

The Scanner class

- A scanner breaks input into tokens using a delimiter pattern, which by default matches with whitespace.
- The resulting tokens may then be converted into values of different types using the various next methods.

Example: Reading from System.in

```
import java.util.Scanner;
Scanner sc = new Scanner(System.in);
System.out.println("Enter your height in feet and inches:");
int feet = sc.nextInt();
int inches = sc.nextInt();
```

The user could type either

```
6 2
```

on one line, or

```
6                  2
```

on one line, or

```
6
2
```

on two lines.

Example: Reading from a text file

```
Scanner sc = new Scanner(new File("data.txt"));
while (sc.hasNextLine()) {
      String line = sc.nextLine();
      System.out.println(line);
}
sc.close();
```

You can use this in the revision lecture’s lab exercise

Advantages of the Scanner class

- Data entered from a keyboard are in ASCII code and they have to be converted to internal format such as int, float, or double.
- If sc is a Scanner object connected to the input channel , then we can use sc.nextInt() to get the next integer, sc.nextFloat() to get the next float, etc with automatic data conversion.

Q: If there are some numbers in a string (i.e., they are in ASCII), can we use a Scanner object to extract and convert them?

Yes, we can, if we have a way to turn a string into a Scanner object.

Example: Reading from a String

```
String input = "red fish    blue     fish";
Scanner sc = new Scanner(input);
System.out.println(sc.next());
System.out.println(sc.next());
System.out.println(sc.next());
System.out.println(sc.next());
sc.close();
```

```
red
fish
blue
fish
```

Example: Extracting info from a String

```
Scanner sc = new Scanner(“Now  2.14  30”);
String s;
double d;
int x;
s = sc.next();
d = sc.nextDouble();
x = sc.nextInt();
// s == “Now”, d == 2.14, x == 30
```

- Note that if the string is “Now 2.14and30”, then 2.14 cannot be extracted as above, as there is no delimiter after 2.14. An InputMismatchException will be thrown

When should we use Scanner?

When we have to process one line of input at a time.

Example: When coefficients of equations ax^2+bx+c are entered 1 line per equation, it may look like the following:

```
2   3               // linear equation
4   5    6          // quadratic equation
```

sc.nextInt() won't tell us whether 4 is the third number of the first line or the first number of the second line. That is, the below input will return exactly the same information.

```
2   3    4           // quadratic equation
5   6                // linear equation
```

Both inputs are also no different from 2 3 4 5 6 for nextInt()

Example: Solving Input Ambiguity

```
Scanner sc = new Scanner(System.in);            // This is for reading
while (sc.hasNextLine()) {                      // process 1 line at a time
    String line = sc.nextLine();
    Scanner scLine = new Scanner(line);         // This is for tokenization
    int i = 0;
    int coef[3];
    while (scLine.hasNextInt()) {
        coef[i] = scLine.nextInt();
        ++i;                                    // get coefficients
    }
     // solve the equation
     // Q: How do you know whether you are dealing with a
     // linear or quadratic equation?
}
```

Example: Specifying delimiters

```
String input = "red,fish,blue,fish";
Scanner sc = new Scanner(input);
sc.useDelimiter(",");
System.out.println(s.next());
System.out.println(s.next());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

```
red
fish
blue
fish
```

To think about:
What happens if the string has spaces in it?
Do they show up as part of the s.next tokens or not?
Try it out yourself.

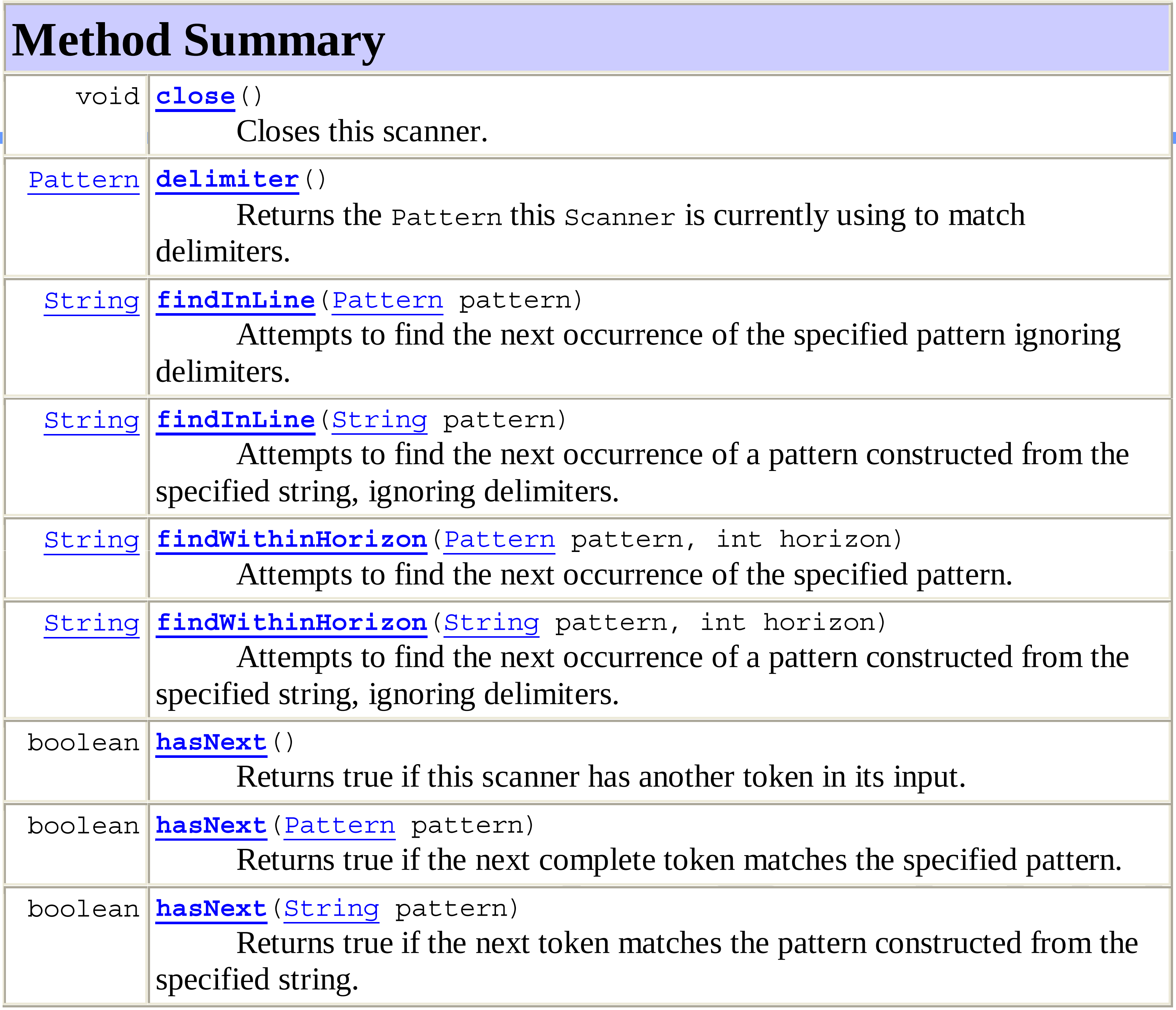
Example: Detecting the end of file

```
public static void main(String[] args) {
  Scanner sc = new Scanner(System.in);
  while (sc.hasNext()) {
       // …
       }
  System.out.println("End of file encountered");
}
```

You can also use

```
       sc.hasnextInt(),
       sc.hasnextLine(), …
```

to detect whether it is at the end of file (eof) http://java.sun.com/j2se/1.5.0/docs/api/java/util/Scanner.html



Method Summary	
void	**close**() Closes this scanner.
Pattern	**delimiter**() Returns the `Pattern` this `Scanner` is currently using to match delimiters.
String	**findInLine**(Pattern pattern) Attempts to find the next occurrence of the specified pattern ignoring delimiters.
String	**findInLine**(String pattern) Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
String	**findWithinHorizon**(Pattern pattern, int horizon) Attempts to find the next occurrence of the specified pattern.
String	**findWithinHorizon**(String pattern, int horizon) Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
boolean	**hasNext**() Returns true if this scanner has another token in its input.
boolean	**hasNext**(Pattern pattern) Returns true if the next complete token matches the specified pattern.
boolean	**hasNext**(String pattern) Returns true if the next token matches the pattern constructed from the specified string.

Tokenization using split()

StringTokenizer is now a legacy class. Use split() in String class to split a string into an array of tokens

```
String s = "This,is,,string";
String[] tokens = s.split(",");        // delimiter ,
for (int i = 0; i < tokens.length; ++i) {
  System.out.println(tokens[i]);
}
```

```
This
is

string
```

- Q: What do you get if s = "red,fish,blue,fish"; in the above code?

Tokenization using indexOf()

Q: How do you extract the percentage embedded in the following line?

```
String line = "File1-123(71%)";
int is = line.indexOf("(");                 // start index
int ie = line.indexOf("%");                 // end index
String ss = line.substring(is+1,ie);        // range: [is+1, ie)
int p1 = Integer.parseInt(ss);
```

Tip: Learn how to use Scanner, StringTokenizer, and String

Auto-boxing / Auto-unboxing

- auto-boxing: converting a primitive value to its corresponding wrapper object
- auto-unboxing: converting a wrapper object to its corresponding primitive value

```
Integer n = 28;          // auto-boxing
int x = n;               // auto-unboxing
```

Example: Boxing and unboxing

```
  Scanner sc = new Scanner(System.in);
  System.out.print("What is his age? ");
A int hisAge = sc.nextInt();
  System.out.print("What is her age? ");
B Integer herAge = sc.nextInt();                              // boxing?
C Integer ageDifference = Math.abs(hisAge – herAge);       // (un)boxing?
D System.out.println("He is  " + hisAge + ", she is " + herAge +
      ": a difference of " + ageDifference + ".");
```

Q: Which line represent boxing?

Enhanced for Loop (1)

The enhanced for loop simplifies the traversal of a collection (i.e., array, Vector, Stack, Queue). To traverse an int array a[], the statement

```
for (int i : a) { … }
```

is a short form of the fully specified for loop:

```
for (int j = 0; j < a.length; ++j) {
  int i = a[j];
  … // print or compute using I
}
```

! When we write

```
for (int i : a) i = i + 10;
```

a[j] is not changed at all. So an enhanced for loop cannot be used to modify a collection

Example: Enhanced for Loop (2)

```
import java.util.*;
class Test_for_each_loop {
  public static void main (String [] args) {
    int a [ ] =  {2, 2, 3, 4, 5};
    // To add all elements of a
    int result = 0;
    for (int i : a) result += i;
    System.out.println ("Result is " + result);
    // To print the array
    for (int i : a) System.out.println (i);
```

Note: "i" is NOT the loop counter

Classes and Interfaces

Chapter 1: Java Fundamentals,
Pages 3-64
Slides 3-76

Chapter 4: Classes and Interfaces,
Pages 193-206
Slides 77-112

Chapter 5: Generics,
Pages 270-278
Slides 113-130

Superclasses and subclasses
The use of the keyword super
Abstract classes
Polymorphism and dynamic binding
Interfaces

Class Inheritance

- Inheritance: To derive new classes by extending existing classes

When a class c1 is derived from another class c2, then

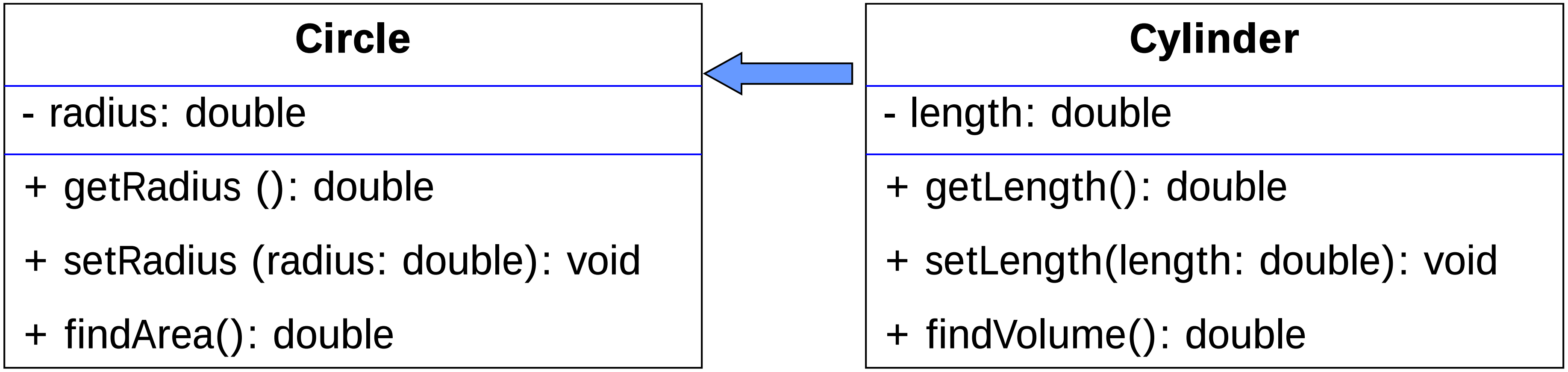
- c1 is called a subclass (child class) of c2, and
- c2 is called the superclass (parent class) of c1.

Not in syllabus. Slide 76 – 104 included for reference only

Class Inheritance (in UML)

Superclass

Subclass



Circle

- radius: double

+ getRadius (): double

+ setRadius (radius: double): void

+ findArea(): double

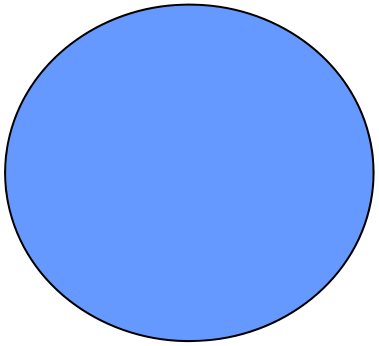
Cylinder

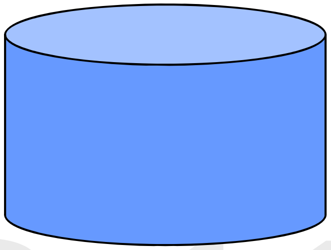
- length: double

+ getLength(): double

+ setLength(length: double): void

+ findVolume(): double





class CircleWithAccessors

```
// CircleWithAccessors.java: A circle class with accessor methods
public class CircleWithAccessors {
  private double radius;
  public CircleWithAccessors()              // default constructor
    { this(1.0); }                          // Q: what’s going on here?
  public CircleWithAccessors(double r)      // constructor
    { radius = r; }
  public double getRadius()                 // accessor
    { return radius; }
  public void setRadius(double newRadius) // mutator
    { radius = newRadius;}
  public double findArea()                  // facilitator
    { return radius * radius * 3.14159;}
}
```

Cylinder1 extends CircleWithAccessors

```java
// Cylinder1.java: Class definition for Cylinder
public class Cylinder1 extends CircleWithAccessors {
   private double length;
   public Cylinder1() { this(1.0, 1.0); }
   public Cylinder1(double r, double l) {
      super(r); // Call superclass' constructor, CircleWithAccessors(r)
      length = l;
   }
   public double getLength() { return length; }
   public double findVolume() { return findArea() * length;}
}
```

Testing Inheritance

```
// TestCylinder.java: Use inheritance.
public class TestCylinder {
  public static void main(String[] args) {
    // Create a Cylinder object and display its properties
    Cylinder1 myCylinder = new Cylinder1(5.0, 2.0);
    System.out.println("The length is " + myCylinder.getLength());
    System.out.println("The radius is " + myCylinder.getRadius());
    System.out.println("The volume of the cylinder is " +
                    myCylinder.findVolume());
    System.out.println("The area of the circle is " +
                    myCylinder.findArea());
  }
}
```

Q: Which methods are inherited?

getRadius() and findArea()

Using the Keyword super

- The keyword super refers to the superclass of the class. It can be used in two ways:
  - To call a superclass constructor
  - To call a superclass method

Calling Superclass Constructors

- To call a superclass constructor, use
  - super() or super(parameters)

- A subclass's constructor will always invoke super() if super() or super(parameters) is not invoked explicitly in the constructor.

- The statement super () or super (parameters) must appear as the first line of the subclass constructor if it is called.

Example: Superclass Constructors

```
class C3 {
  public C3 () {      // constructor
    System.out.println ("C3's default constructor");
} }
class C2 extends C3 {
  public C2 () {      // implicitly call C3's constructor
    System.out.println ("C2's default constructor");
} }
public class C1 extends C2 {
  public C1 () {      // implicitly call C2's constructor
    System.out.println ("C1's default constructor");
  }
  public C1 (int n) { // implicitly call C2's constructor
    System.out.println ("C1's constructor");
} }
public static void main (String [] args)
   { new C1(1); } // Q: What’s the output? Which constructor do we call?
}
```

```
C3's default constructor
C2's default constructor
C1's constructor
```

Superclass default constructor

- If a superclass defines constructors other than a default constructor,

  then the subclass cannot use the default constructor of the superclass as the superclass does not have one.

```
class B {
   public B (String name) {         // non-default constructor
     System.out.println ("B's non-default constructor");
   }       // Q: Will the compiler give B a default constructor?
}
public class A extends B {
    // class A cannot be compiled as the default constructor
    // of A given by the compiler has a call to the default
    // constructor of B which does not exist
}
```

Calling Superclass Methods

- The keyword super also can be used to refer to methods other than the constructor in the superclass.
- For example, in Cylinder1 class, if it has defined the findArea() method, then in order to call the findArea() method in the superclass CircleWithAccessors , super is needed:

```
double findVolume () {
   return super.findArea() * length;
}
```

Accessing super-super class attributes

```
class A                  { int x = 77; }
class B extends A        { int x = 88; }
class C extends B {
    int x = 99;
    void printing () {
      System.out.println ("X is " + x);
      // Q: How do you access the value 88?
      System.out.println ("Super X is " + super.x);
      // Q: How do you access the value 77?  super.super.x?
      System.out.println ("Super Super X is " + ((A) this).x);
   }
}
class SuperSuper {
   public static void main (String [] args) {
     new C ().printing ();
   }
}
```

Invalid!
Use cast

Method Overriding

- A subclass inherits methods from a superclass.

- Sometimes it is necessary for the subclass to redefine the methods from the superclass. This is called method overriding.

Example: Method Overriding (1)

```
// Cylinder2.java: New cylinder class that overrides the findArea()
public class Cylinder2 extends CircleWithAccessors {
private double length;
public Cylinder2() { length = 1.0;  }          // Where is super()?
public Cylinder2(double radius, double l) {
    super (radius);
    length = l;
  }
public double getLength() { return length;}
public double findArea() {                      // method overriding
    return 2 * super.findArea() + 2 * getRadius() * Math.PI * length;
  }
public double findVolume() {return super.findArea() * length; }
}
```

Q: Do we have to specify super.getRadius()?

Example: Method Overriding (2)

```
// TestOverrideMethod.java: Test the Cylinder class that overrides
// its superclass's methods.
public class TestOverrideMethod {
  public static void main(String[] args) {
    Cylinder2 myCylinder = new Cylinder2(5.0, 2.0);
    System.out.println("The length is " + myCylinder.getLength());
    System.out.println("The radius is " + myCylinder.getRadius());
    System.out.println("The surface area of the cylinder is "+
                                                  myCylinder.findArea());
    System.out.println("The volume of the cylinder is "+
                                                  myCylinder.findVolume());
  }
}
```

Abstract Classes

- In the inheritance hierarchy, classes become more specific and concrete with each new subclass.
- Moving from a subclass to superclasses, the classes become more general and less specific.
- When a class is so general that an instance cannot be created, it is an abstract class.

Abstract methods and classes

- An abstract class is a class that has at least one abstract method
- An abstract method is a method declared as abstract, not implemented, and all derived class must eventually implement

```
public abstract class GeometricObject {
  public abstract double findArea();
  public abstract double findPerimeter();
  public double semiperimeter() {
      return findPerimeter ( ) / 2;
    }
}
```

Abstract class GeometricObject

```
public abstract class GeometricObject {
  private String color = "white";
  private boolean filled;
  protected GeometricObject() { }             // default constructor
  protected GeometricObject (String c, boolean f) {
    color = c;
    filled = f;
  }
  public String getColor() { return color; }  // instantiated methods
  public void setColor(String c) { color = c; }
  public boolean isFilled() { return filled; }
  public void setFilled(boolean f) { filled = f; }
  public abstract double findArea();          // abstract methods
  public abstract double findPerimeter();
}
```

class Circle extends GeometricObject

```
public class Circle extends GeometricObject {
   private double radius;
   public Circle() { this(1.0); }
   public Circle(double radius) { this(radius, "white", false); }
   public Circle(double r, String color, boolean filled) {
      super(color, filled);
      radius = r;
   }
   public double getRadius()          { return radius;}
   public void setRadius(double r) { radius = r; }
   public double findArea()           { return radius*radius*Math.PI;}
   public double findPerimeter()   { return 2*radius*Math.PI;}
   public boolean equals(Circle circle) { return radius == circle.getRadius(); }
   public String toString()           { return "[Circle] radius = " + radius;}
}
```

class Rectangle extends GeometricObject (1)

```
public class Rectangle extends GeometricObject {
  private double width;
  private double height;

  public Rectangle() { this(1.0, 1.0);}
  public Rectangle(double width, double height) {
    this(width, height, "white", false);       }
  public Rectangle(double w, double h,
    String color, boolean filled) {
    super(color, filled);
    width = w;
    height = h;
   }

  public double getWidth() {return width; }
  public void setWidth(double w) {width = w; }
```

class Rectangle extends GeometricObject (2)

```
  public double   getHeight() { return height; }
  public void     setHeight(double h) { height = h; }
  public double   findArea() { return width*height; }
  public double   findPerimeter() { return 2*(width + height); }
  public boolean equals(Rectangle rectangle) {
    return (width == rectangle.getWidth()) &&
      (height == rectangle.getHeight());
  }
  public String   toString() {
    return "[Rectangle] width = " + width + " and height = " + height;
  }
}
```

class Cylinder extends Circle (1)

```
public class Cylinder extends Circle {
  private double length;
  public Cylinder() { this(1.0, 1.0); }
  public Cylinder(double radius, double length) {
    this(radius, "white", false, length); }
  public Cylinder(double radius, String color, boolean filled, double l) {
    super(radius, color, filled);
    length = l;
  }

  public double getLength()             { return length;}
  public void   setLength(double l)     { length = l;}
```

class Cylinder extends Circle (2)

```
  public double findArea() {                    // overriding
    return 2*super.findArea()+(2*getRadius()*Math.PI)*length; }

  public double findVolume() {return super.findArea()*length; }

  public boolean equals(Cylinder cylinder) {
    return (this.getRadius() == cylinder.getRadius()) &&
      (length == cylinder.getLength());
  }

  public String toString() {
    return "[Cylinder] radius = " + getRadius() + " and length "
      + length;
  }
}
```

Abstract Classes and Interfaces

An abstract class can implement an interface as well

```
public abstract class GeometricObject implements Comparable
    <GeometricObject> {
  …
  public int compareTo (GeometricObject x) {
    if (findArea () == x. findArea ())
        return 0;
    else if (findArea () > x. findArea ())
          return 1;
        else
          return -1;
  }
}
```

Polymorphism – intuition

- A dog is an animal. A cat is also an animal. To describe them in classes, both Dog and Cat can be developed as subclasses of Animal class.
- All animals make noise. Given an animal (object), we can always call animal.makeNoise(). Since different animal makes different noise, the makeNoise() method in the Animal class is an abstract method.

- As a subclass of Animal,
Dog has to implement makeNoise() to bark,
and Cat has to implement makeNoise() to meow.
- When animal.makeNoise() is executed, polymorphism allows the correct version of makeNoise() to be called so that barking or meowing can be expected depending on whether an animal (object) is a dog or a cat.

Polymorphism

- Polymorphism literally means "*many forms*" (shapes). In Java, it is the ability to perform the operations according to the identity of an object instantiated from one of its related subclasses

- For example, a Cylinder, Circle, Rectangle are subclasses of GeometricObject
  - Thus a GeometricObject object has three forms. It may behave as a Cylinder, Circle, or Rectangle according to the true identity of this object

- Polymorphism can be realized through dynamic binding

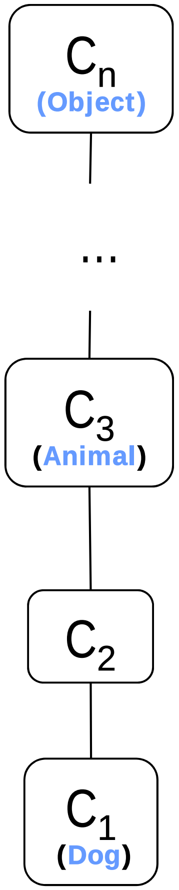
Dynamic Binding (1)

- A method may be defined in a superclass but overridden in a subclass.

- Which implementation of the method is used on a particular call will be determined dynamically by the Java Virtual Machine at runtime.

- This capability is known as dynamic binding, the binding of a method to its actual implementation during runtime.

Dynamic Binding (2)



Suppose an object o is an instance of C_1 with C_1 a subclass of C_2, C_2 a subclass of C_3, etc. C_n is the most general class and C_1 is the most specific class.

- In Java, C_n is the Object class. If we invoke a method m() through o, the Java Virtual Machine will search for this method in C_1, C_2, …, C_{n-1} and C_n until it is found, and the first found is invoked.
- If m() is found in C_1, then it is called immediately without moving up the inheritance hierarchy. This happens when we execute o.makeNoise() with o being a Dog object of C_1, regarded as a subclass of Animal class C_2

Dynamic Binding (3)

- Polymorphism allows methods to be used for a wide range of object arguments.

- We may pass an object as an argument of a method if the class of this object is a subclass of the class of the parameter

❗ The method invoked through this object is determined dynamically by the class of the argument, not by the class of the parameter.

Example: Dynamic Binding

```
public class TestPolymorphism {
  public static void main(String[] args) {
    GeometricObject geoObject1 = new Circle (5);
    GeometricObject geoObject2 = new Rectangle (5, 3);
    System.out.println("Do the two objects have the same area? "
                              + equalArea(geoObject1, geoObject2));
    displayGeometricObject(geoObject1);
    displayGeometricObject(geoObject2);
  }
  static boolean equalArea( GeometricObject o1,  GeometricObject o2) {
    return o1.findArea() == o2.findArea(); }
  static void displayGeometricObject( GeometricObject object) {
    System.out.println(object.toString());
    System.out.println("The area is " + object.findArea());
    System.out.println("The perimeter is " + object.findPerimeter());
  }
}
```

Interfaces

- The interface in Java consists only of public abstract methods and public static final fields.
- A class is said to implement an interface if it provides definitions for all of the abstract methods in the interface
- Each interface is compiled into a separate bytecode file, just like a regular class.

- We cannot create an instance of an interface, but

  we can use an interface as a data type for a variable, as the result of casting, etc.

- To define an interface called InterfaceName, use:

```
modifier interface InterfaceName {
    /* Constant declarations */
    /* Method signatures */
}
```

Implementing several interfaces

- Sometimes it is necessary to derive a subclass from several classes, thus inheriting their data and methods.
  Java, however, does not allow multiple inheritance.

- The extends keyword allows only one parent class. With interfaces, we can achieve the effect close to that of multiple inheritance by implementing several interfaces.

```
public class Vector<E> extends AbstractList<E> implements
List<E>, RandomAccess, Cloneable, Serializable
```

Q: What are the methods that are abstract in AbstractList<E>?

Interface Comparable<T>

- Suppose we want to design a generic method to find the larger of two objects, we can use the following interface in java.lang:

```
// Interface for comparing objects
package java.lang;
public interface Comparable<T> {
   public int compareTo (T o);
}
```

- The compareTo method determines the order of this object with the specified object o, and returns -1, 0 or +1 if this object is regarded as smaller, equal, or larger than the specified object o.
  - This is similar to the concepts of <, ==, > applicable to numbers and their wrappers

Comparable<String>

- When T is String, then the interface of interest is Comparable<String>. Any class that has implemented the method compareTo(String s) can claim to have implemented Comparable<String>.

```
class A implements Comparable<String> {
  public int compareTo(String s) { return 0;}
   public static void main (String [] args) {
     A a = new A();
     System.out.println(a.compareTo(""));
  }
}
```

- Note that class A does not have anything to do with String other than having implemented the method compareTo(String s)

Using Interface As a Data Type

```
public class A {
  public static Comparable <String> max(String o1, String o2) {
    if (o1.compareTo(o2 ) > 0) return o1;
    else return o2; }
  public static void main (String [] args) {
     String s1 = “abcdef”; String s2 = “acdef”;

     // s3 supports all methods described in the interface Comparable
     <String>
     Comparable <String> s3 = max (s1, s2);

     // dynamic binding to toString() of String class is done here
     System.out.println (s3);
  } }
```

Note that String class implements Comparable<String> and it is valid to return a string object as a Comparable<String> object.

Interfaces vs. Abstract Classes

- Data
  - In an interface, all data are constants (keyword final is omitted)
  - An abstract class can have non-constant data fields.
- Methods
  - In an interface, all methods are not implemented
  - An abstract class can have concrete methods.
- Keyword abstract
  - In an interface, the keyword abstract in the method signature can be omitted
  - In an abstract class, it is needed for an abstract method.
- Inheritance
  - A class can implement multiple interfaces
  - A class can inherit only from one (abstract) class

Generics

Chapter 1: Java Fundamentals, Pages 3-64
Slides 3-76

Chapter 4: Classes and Interfaces,
Pages 193-206
Slides 77-112

Chapter 5: Generics, Pages 270-278
Slides 113-130 << Here

Strategies towards code reuse

We want to re-use code as much as possible

- Inheritance is one way to re-use contents of a class and its methods including the type of the data used.

- Another way is to make a class capable of handling the most general data type – Object.
    - Class Object, the most general class, is the mother (base class) of all classes

When class Object is used
(mainly in older versions of java)

- The frequent castings needed when class Object is used is a bit annoying:

```
class ArrayList { // old version of Java (jdk 1.3)
  void add (Object item) { …. }
  Object get (int i) { …. }
}
ArrayList myIntList = new ArrayList ();
myIntList.add (new Integer (88));
Integer x = (Integer) (myIntList.get (0));
```

Q: myIntList.add (new Double (1.2)); // Is it valid?

Variation: Generic class ArrayList<E>

```
class ArrayList <E> {
  void add (E item) { .... }
  E get (int i) { .... }
}

ArrayList <Integer> myIntList = new ArrayList <Integer> ();
myIntList.add(88);                    // boxing
Integer x = myIntList.get(0);
```

The compiler guarantees that myIntList.get() returns an Integer. It does not allow non-Integer objects (such as Double) to be added.

Motivation for Generics: Pair of int

To return an int from a method, simply declare its return type as int:

```
int max(int a, int b);
```

To return a pair of int, we have to return an object of the class PairOfInt :

```
class PairOfInt {
    int first;
    int second;
}
```

For example,

```
PairOfInt minAndMax (int [] a)
```

Q: What if a pair of float, or a pair of double is to be returned? Must classes PairOfFloat and PairOfDouble be defined as well?

Not if we use generics.

Generic classes

- A generic class describes a class in terms of object type parameters
- Usually these parameters are specified as T, E, or any short symbols in upper case.
- These parameters are replaced by actual data types when declaring an instance of the class.

Class OrderedPair<T> (1)

```
public class OrderedPair <T> {            // Note: T can not be a
  private T first, second;                // primitive data type
  public OrderedPair() {}
  public void setPair(T firstItem, T secondItem) {
    first = firstItem;
    second = secondItem;
  }
  public void changeOrder() {
    T temp = first;
    first = second;
    second = temp;
  }
  public String toString() { return "(" + first + ", " + second + ")"; }
} // end OrderedPair
```

Note: toString() is a very useful method of a class

Class OrderedPair<T> (2)

```
OrderedPair<String> fruit = new OrderedPair<String> ();
fruit.setPair("apple", "orange");
fruit. changeOrder();
OrderedPair<Integer> xyCoord = new OrderedPair<Integer> ();
                                   // Use object type!
xyCoord.setPair(1,2);              // boxing
xyCoord.changeOrder();
```

Q: Must first and second be always of the same data type?

Class Pair<S, T>

```
public class Pair <S, T> {
  private S first;
  private T second;
  public Pair(S firstItem, T secondItem) {
    first = firstItem;
    second = secondItem;
  }
  public String toString() { return "(" + first + ", " + second + ")"; }
} // end Pair
```

```
Pair<String,Double> price = new Pair<String,Double> ("apple", 0.5);
System.out.println(price);
```

Advantages of using Generics

- Generics provide increased readability and type safety.
- Generics are compiled once only and yet they can be used for any data type.
- Classes, Interfaces and Methods can all be made generics.

Generic array not supported

```
class genericArray <T> {
   private T[] anArray;
   static final int MAX = 50;
   genericArray () {
     anArray = new T[MAX];
   } }
```

```
Compilation error:
genericArray.java:6: generic array creation
anArray = new T[MAX];
          ^
1 error
```

To overcome this problem, use type casting:

```
class genericArray <T> {
    private Object  [] anArray;
    static final int MAX = 50;
    genericArray () {
    anArray = (T[]) new Object [MAX];
    }
}
```

Compiler generates the warning:

unchecked type casting error

Generic methods

Just like class and interface, methods can also be generic.

```
public class A {    // A is not generic, but the method below is
   public static <T> void swap (T a, T b) {
    T tmp;
    tmp = a;
    a = b;
    b = tmp;
  }
  static public void main(String[] args) {
    int a = 1;
    int b = 2;
    swap (a,b);
    System.out.println("a:" + a + " b:" + b);
 } //end main
} // end class Generic
```

a:1 b:2

Surprised?
What’s wrong here

Generic comparison (1)

First attempt:

```
public class A {
   public static <T> T max (T a, T b) {
      if (a < b) return b;
      else return a;
   }
}
```

Fails! < is only defined for numbers and their wrappers only.

To compare 2 objects obj1 and obj2, we should use

obj1.compareTo(obj2);

Generic comparison (2)

Second attempt:

```
public class A {
  public static <T> T max (T a, T b) {
     if (a.compareTo(b) < 0) return b;
     else return a;
   }
}
```

Failed again!

T must implement (extend) the Comparable<T> interface in order that compareTo() can be called through a of type T

Generic comparison (3)

Final attempt:

```
public class A {
     public static <T extends Comparable<? super T> >
     T max (T a, T b) {
    if (a.compareTo(b) < 0) return b;
    else return a;
  }
  static public void main(String[] args) {
     System.out.println("max(3,5):" + max(3,5));
 } //end main
} // end class Generic
```

Note: T extends Comparable<? super T>

means that T must support the compareTo() method which could have been implemented in any of the super class of T and inherited by T.

Some useful generic java classes

In the java.util package:

- ArrayList <E>
- Vector <E>
- List <E>   // interface
- LinkedList <E>
- Stack <E>
- Queue <E>   // interface

ArrayList <E>/ Vector<E> (1)

```
import java.util.*;
class Circle {
  private int radius;
  public Circle (int r) { radius = r; }
  public String toString () { return "C [" + radius + "]";}
}
class TestArrayListAndVector {
   public static void main (String [] args) {
     ArrayList <Integer>   ai = new ArrayList <Integer> ();
     ArrayList <Circle>     ac = new ArrayList <Circle> ();
     Vector      <Circle>     vc = new Vector <Circle> ();
```

ArrayList <E>/ Vector<E> (2)

```
    for (int i = 1; i <= 5; i++) {
      ai.add (i);                                      // auto boxing
      ac.add (new Circle (i));
      vc.add (new Circle (i));
    }
    for (int i : ai) System.out.print (i + "\t");      // auto unboxing
    System.out.println ();
    for (Circle c : ac) System.out.print (c + "\t");
    System.out.println ();
    for (Circle c : vc) System.out.print (c + "\t");
    System.out.println ();
  }
}
```

```
1       2       3       4       5
C [1]   C [2]   C [3]   C [4]   C [5]
C [1]   C [2]   C [3]   C [4]   C [5]
```