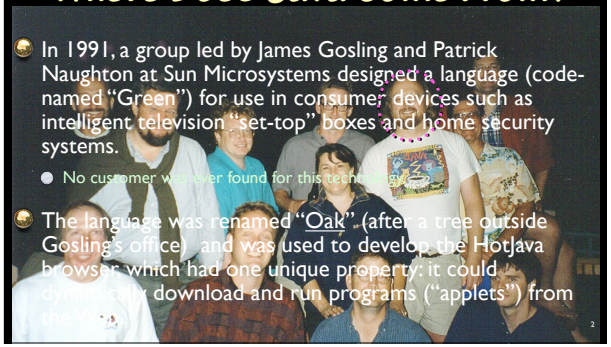
Computer Science E-10b: Introduction to Computer Science Using Java, 11

Dr. Henry H. Leitner

1

Where Does Java Come From?



- In 1991, a group led by James Gosling and Patrick Naughton at Sun Microsystems designed a language (code-named "Green") for use in consumer devices such as intelligent television "set-top" boxes and home security systems.
  - No customer was ever found for this technology.
- The language was renamed "Oak" (after a tree outside Gosling's office) and was used to develop the HotJava browser which had one unique property: it could dynamically download and run programs ("applets") from the Internet.

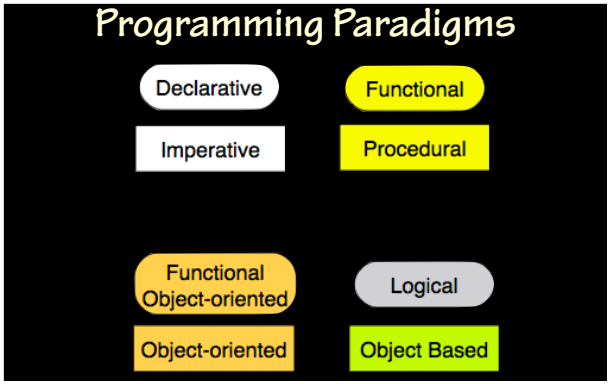
2

The "Other" Truth About Java

3

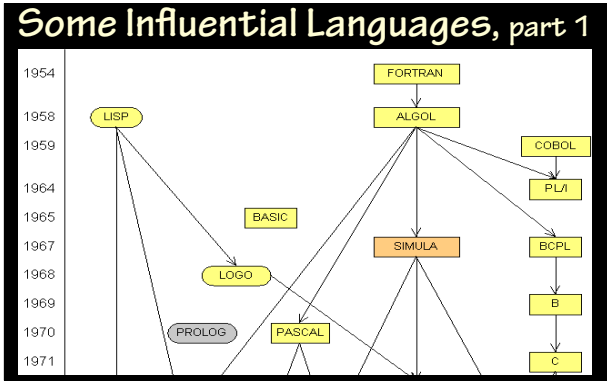
Programming Paradigms

4



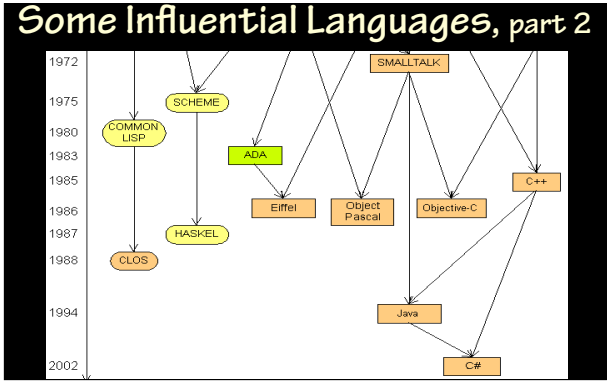
Some Influential Languages, part 1

5



Some Influential Languages, part 2

6



Important Features of Java

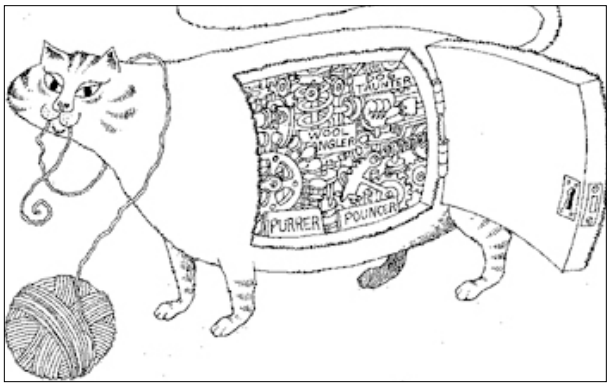
7

- Simple
- Multi-threaded
- Recycles memory automatically
- Distributed and secure
- Robust
- Portable; programs are compiled into byte code.
  - Byte code is platform-neutral; it is executed by a program that pretends it's a computer based on the byte-code instruction set, namely a Java virtual machine.

What is OOP?

8

- Object-oriented programming technology can be summarized by three key concepts:
  - Objects that provide encapsulation of procedures and data; it allows hiding of the implementation details of an object from the clients of the object.
  - Classes that implement inheritance within class hierarchies.
  - Messages that support polymorphism across objects; it allows the same code to be used with several different types of objects and behave differently depending on the actual type being used.

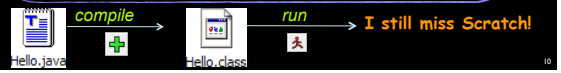


9

A Trivial Java Application

10

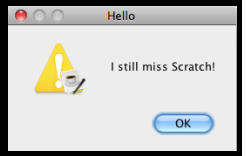
```
class Hello
{
   public static void  main  ( String [ ] args)
   {
      System.out.println ( " I still miss Scratch! ");

   }
}
```



10

Trivial Application with a GUI

11




```
import javax.swing.JOptionPane;

class GUIapp
{
  public static void main (String [] args)
  {
     JOptionPane.showMessageDialog (null,
        "I still miss Scratch!", "Hello",
         JOptionPane.WARNING_MESSAGE);
  }
}
```

11

A Trivial Java Applet

12

```
import javax.swing.*;        // File TrivialApplet.java
 import java.awt.*;
 public class TrivialApplet extends JApplet
 {
    public void paint( Graphics g )
    {
        g.setFont (new Font("Times", Font.BOLD, 50));
        g.setColor (Color.RED);
        g.drawLine ( 15, 10, 240, 10 );
        g.drawLine ( 15, 70, 240, 70 );
        g.setColor (Color.BLUE);
        g.drawString( "I *really* miss Scratch!", 50, 50 );
    }
  }

<html>          <!-- File TrivialApplet.html -->
   <applet code= "TrivialApplet.class" width=450 height=500>
   </applet>
 </html>
```

12

Identifiers in Java

13

- Identifiers must begin with a letter and may contain additional letters and digits.
  - In place of a letter you may use $, _, or letters from other alphabets (such as Greek, Japanese, Arabic, etc. — all part of the Unicode character set).
- Keywords and literals cannot be used:
  - abstract, assert, boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, enum, extends, false, final, finally, float, for, goto, if, implements, import, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, true, try, void, volatile, while

Strings

14

- ... are objects containing a sequence of characters.
  - To construct a `String`: start and end with a double-quote `"`
  - Examples:

```
"hello"
"This is a rather looong string!"
```

- Restrictions:
  - May not span multiple lines.

```
"This is not
a legal String."
```

  - May not contain a " character.

```
"This is not a "legal" String either."
```


Escape Sequences

15

- A special sequence of characters used to represent certain characters in a string. For example,

`\t`	tab character
`\n`	new line character
`\"`	quotation mark character
`\\`	backslash character

- Example: What does this print?

```
System.out.println("\\hello\nhow\tare \"you\"?\\\\");
```

Unicode Example \u

16

	037	038	039	03A	03B	03C	03D	03E	03F
0	Ͱ 0370		ΐ 0390	Π 03A0	ΰ 03B0	π 03C0	ϐ 03D0	Ϣ 03E0	ϰ 03F0
1	ͱ 0371		Α 0391	Ρ 03A1	α 03B1	ρ 03C1	ϑ 03D1	ϣ 03E1	ϱ 03F1
2	Ͳ 0372		Β 0392		β 03B2	ς 03C2	ϒ 03D2	Ϥ 03E2	ϲ 03F2
3	ͳ 0373		Γ 0393	Σ 03A3	γ 03B3	σ 03C3	ϓ 03D3	ϥ 03E3	ϳ 03F3

Operator Precedence, part 1

17

[] . (…) ++, --	array indexing object member reference method invocation & expr evaluation postfix increment, decrement	L to R
++, -- +, - ~, !	prefix increment, decrement unary plus, minus bitwise NOT, logical NOT	R to L
new (*type*)	object instantiation type-cast	R to L
*, /, %	multiplication, division, remainder	L to R

Operator Precedence, part 2

18

+ -	addition and string concatenation subtraction	L to R
<<, >> >>>	left shift, right shift right shift with zero fill	L to R
<, >, >=, >= instanceof	less-than, greater-than, etc. type comparison	L to R
==, !=	equal, not-equal	L to R
&	bitwise AND, boolean AND	L to R
^	bitwise XOR, boolean XOR	L to R
|	bitwise OR, boolean OR	L to R
&&	logical AND	L to R
||	logical OR	L to R

Operator Precedence, part 3

19

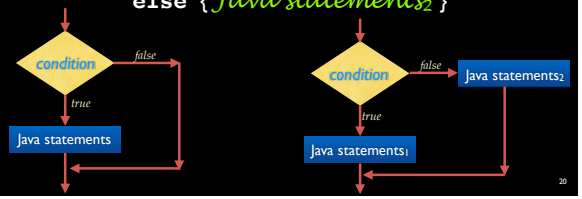
? :	conditional operator	R to L
=	assignment	R to L
+=	addition / string concatenation	
-=	subtraction, then assignment	
*=	multiplication, then assignment	
/=	division, then assignment	
%=	remainder, then assignment	
<<=	left shift, then assignment	
>>=	right shift (sign), then assignment	
>>>=	right shift (zero), then assignment	
&=	bitwise / boolean AND, assignment	
^=	bitwise / boolean XOR, assignment	
|=	bitwise / boolean XOR, assignment	

Control Flow in Java, part 1

20

Decision-Making

- `if` (*condition*) { *Java statements* }
- `if` (*condition*) { *Java statements*$_1$ }
  `else` { *Java statements*$_2$ }



20

Control Flow in Java, part 2

21

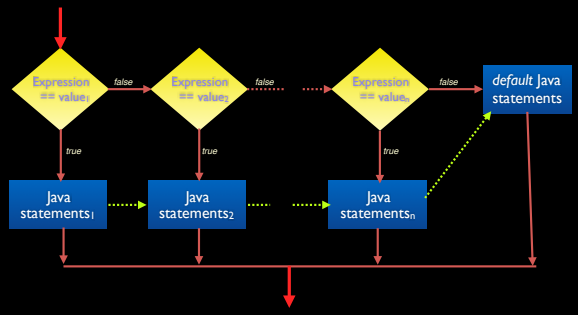
Decision-Making

```
switch (expression of certain types)
{  case value1: Java statements1 ;
   case value2: Java statements2 ;
      ...
      ...
   case valuen: Java statementsn ;

   default: default Java statements;
}
```

21

Control Flow in switch Statement

22



Control Flow in Java, part 3

23

Looping

- `for` (*initialize; boolean expr; increment*) { *Java statements* }
- `while` (*condition*) { *Java statements* }
- `do` { *Java statements* } `while` (*condition*);

Branching

- `continue` *optional-label*;
- `break` *optional-label*;
- `return` *optional-expression*;

23

Primitive Data Types in Java

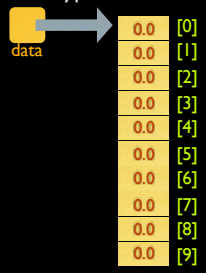
24

Type	Contains	Default	Size	Minimum	Maximum
boolean	true or false	false	1 bit		
char	Unicode character	\u0000	16 bits	\u0000	\uFFFF
byte	signed int	0	8 bits	-128	127
short	signed int	0	16 bits	-32768	32767
int	signed int	0	32 bits	-2,147,483,648	2,147,483,647
long	signed int	0	64 bits	-2^{63}	$2^{63} - 1$
float	IEEE 754 floating pt.	0	32 bits	-3.4 E38	3.4 E38
double	IEEE 754 floating pt.	0	64 bits	-1.7 E308	1.7 E308

A Quick Review of Arrays

25

- An array is a sequence of values of the same type
  - Construct array: new double[10]
  - Store in variable of type double[]: double [] data = new double[10];
- When array is created, all values are initialized depending on array type:
  - *Numbers:* 0, 0.0
  - *boolean:* false
  - *Object References:* null



Single-Dimensioned Arrays, *continued*

26

- If array data has n elements, the first one is data[0] and the last is data[n-1] . Array elements contain primitive data, or they contains references (pointers) to objects.
- What does the following code do?

```
String [] strArr = new String [5];
for (int k = 0; k < strArr.length; k++)
  { strArr[k] = "Hello" + k + 1;  }
```

- You can't directly
  - change the size of an array at execution time
  - compare arrays for equality using ==
  - print an array simply using print or println

26

The `java.util.Arrays` class

27

Method Name	Description
binarySearch (array, value)	*returns the index of the given value in a SORTED array*
copyOf (array, newLength)	*returns a new copy of an array (possibly truncated or padded)*
equals (array1, array2)	*returns true if the 2 arrays contain the same elements*
fill (array, value)	*sets every element of an array to a particular value*
toString (array)	*returns a string representation of the contents of the specified array*

27

Arrays Get Passed by Reference

28

- Changes made in the method are also seen by the caller. E.g.,

```
public static void main (String [] args)
 {   int[] iq = {3, 5, 7};
     increase (iq, 3);
     System.out.println ( Arrays.toString(iq) );
 }

public static void increase (int [] a, int f) {
 for (int i = 0; i < a.length; i++)
       {  a[i] *= f;    }
}
```

- Output?

28

Two-Dimensional Arrays

29

- Arrays can be used to store data in two dimensions (2D) like a spreadsheet
  - Rows and Columns
  - Also known as a "matrix"

	Gold	Silver	Bronze
Canada	1	0	1
China	1	1	0
Germany	0	0	1
Korea	1	0	0
Japan	0	1	1
Russia	0	2	1
U.S.A.	2	1	0

29

Declaring Two-Dimensional Arrays

30

- Use two "pairs" of square braces
  - `final int COUNTRIES = 7;`
  - `final int MEDALS = 3;`
  - `int [][] counts = new int [COUNTRIES][MEDALS];`
- You can alternatively initialize the array when it gets declared

```
int [][] counts =
{
  { 1, 0, 1 },
  { 1, 1, 0 },
  { 0, 0, 1 },
  { 1, 0, 0 },
  { 0, 1, 1 },
  { 0, 2, 1 },
  { 2, 1, 0 }
};
```

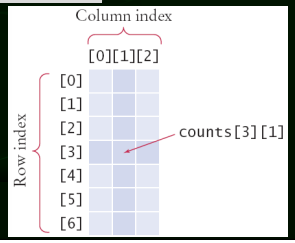
	Gold	Silver	Bronze
Canada	1	0	1
China	1	1	0
Germany	0	0	1
Korea	1	0	0
Japan	0	1	1
Russia	0	2	1
U.S.A.	2	1	0

Accessing Elements

31

- Use two index values (Row then Column):

```
int value = counts[3][1];
```



31

To Print

32

- Use nested for-loops
  - Outer: i'th row
  - Inner: j'th column

```
for (int i = 0; i < COUNTRIES; i++)
{
  // Process the ith row
  for (int j = 0; j < MEDALS; j++)
  {
    // Process the jth column in the ith row
    System.out.printf("%8d", counts[i][j]);
  }
  System.out.println(); // Start a new line at the end of the row
}
```

32

Locating Neighboring Elements

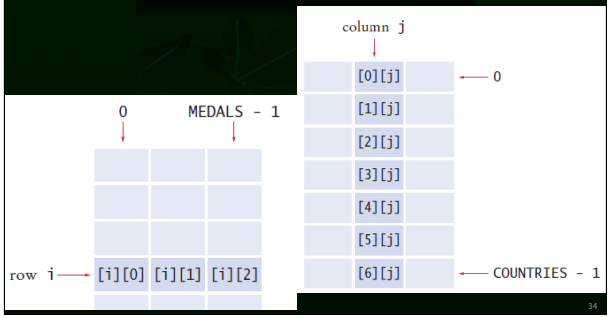
33

- Some programs that work with two-dimensional arrays need to locate the elements that are adjacent to an element
- This task is particularly common in games
- You are at loc i, j
- Watch out for edges!
  - No negative indexes!
  - Not off the 'board'

[i-1] [j-1]	[i-1] [j]	[i-1] [j+1]
[i] [j-1]	**[i] [j]**	[i] [j+1]
[i+1] [j-1]	[i+1] [j]	[i+1] [j+1]

Adding Rows and Columns

(see program Medals.java)



34

34

Review of Objects (briefly)

Object: An entity that contains **data** (variables inside the object) and **behavior** (methods inside the object).

- You interact with the methods
- The data is hidden in the object
- A class is the data type of objects

Methods (behavior)

Fields (state)

Creating (instantiating) an object:

```
Type objectName = new Type(optional arguments);
```

Calling an object's method:

```
objectName.methodName(optional arguments);
```

35

35

Class java.lang.String

- `int length()`
- `String trim()`
- `boolean equalsIgnoreCase(String other)`
- `int compareTo(String other)`
- `String substring(int begin)`
  - `String substring(int begin, int pastEnd)`
- `String toLowerCase()`
- `String toUpperCase()`
- `boolean endsWith(String suffix)`
- `boolean startsWith(String prefix)`
  - `boolean startsWith(String prefix,int offset)`
- `boolean equals(String other)`

See StringTest.java

36

36

Class java.lang.StringBuilder

37

- 3 constructors
- `int length()`
- `int capacity()`
- `char charAt (int index)`
- `char setCharAt (int index, char ch)`
- `StringBuilder append (String str)`
  - `StringBuilder append (char ch)`
- `StringBuilder insert (int offset, String str)`
  - `StringBuilder insert (int offset, char ch)`
- `StringBuilder reverse ()`
- `String substring (int start)`
  - `String substring (int start, int end)`

See StringBuilderTest.java

37

Where are the Other Classes?

38

- Java programs contain one or more classes that contain definitions of methods (each is a sequence of instructions that describes how to carry out a computation). Every Java app contains at least one class with a method named **main**.
- Java classes are grouped into packages. If you use a class from a package other than java.lang
  - "import" the class; or
  - use the FULL name (both package and class names) whenever you use a class. For example,

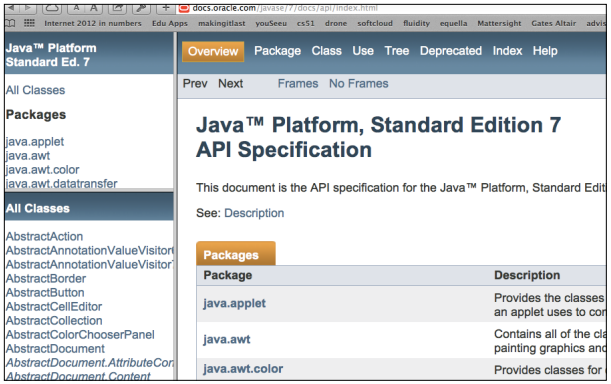
```
java.util.Scanner s =
    new  java.util.Scanner (System.in);
```

38

Java API Packages

39

- Java contains many predefined classes grouped into packages. Together, these packages are referred to as the Java applications programming interface (Java API). Some examples:
  - java.applet
  - javax.swing (contains classes for creating GUIs)
  - java.awt, java.awt.event and java.awt.image
  - java.io (contains classes for dealing with files)
  - java.util (contains Scanner class)
  - java.net
  - java.lang (automatically imported by the compiler)



40

Some of the New Features in Java ≥ 1.5

- Enumeration Types
- Autoboxing
- Variable Length Parameter Lists
- New ***for-each*** Loop
- Class **Scanner**
- Formatted output using **printf**
- Generic Types

41

Enumerated Types

- An **enum**erated type creates a new type of variable and establishes all possible values by listing them. Common examples include *compass directions* (values of NORTH, SOUTH, EAST, and WEST) and the *days of the week*.

```
enum Grade { A, B, C, D, F };

Grade myGrade = Grade.A;

if (myGrade == Grade.F)
  System.out.println("You fail");

System.out.println(myGrade);
```

42

42

Enumerated Types, *continued*

43

- No invalid value can be assigned
- Internally, each value has a corresponding ordinal value, starting at 0
  - You cannot use the ordinal values directly. Instead you can use the `ordinal()` method
- Java also provides a `name()` method
- enum types cannot be declared locally
- You can pass enum variables and values as arguments

Enumerated Type Example

44

```
class Enumerated
{
  enum Grade { A, B, C, D, F}

  public static void main(String [] args)
  {  Grade myGrade = Grade.A;

     if (myGrade == Grade.F)
        System.out.println("You fail");
     else System.out.println("You pass");

     System.out.println("myGrade: " + myGrade);
     System.out.println("myGrade.ordinal(): " +
              myGrade.ordinal());
     System.out.println("myGrade.name(): " +
              myGrade.name());
  }
}
```


Autoboxing

45

- ... the automatic conversion from a primitive value to the corresponding "wrapper" object

```
Integer myInt;

int number = 5;
myInt = number;
```

- *Unboxing* converts from an object to the primitive type, e.g.,

```
Integer myInt2 = new Integer(15);

int number = myInt2;
```

Variable Length Arguments

46

- Java methods can now accept a *variable number* of arguments
- These arguments will be stored in an *array*. For example,

```
public static void main (String [] args)
{
   myMethod ("A", "B", "C", "D");
}

public static void myMethod (String ... data)
{
   for (int i = 0; i < data.length; i++)
      {  System.out.println (data[i]);  }
}
```

OUTPUT
A
B
C
D

46

Variable Length Arguments, cont'd.

47

- Methods can accept other parameters, in addition to the *variable length parameter*
- The variable length parameter must come last in the argument list
- Methods can accept only one variable length parameter, e.g.,

```
void myMethod (int a, double b,
               int ... numbers)
 {
```

47

The New "For Each" Loop

48

- To iterate through a sequence of elements, such as those in an array or an ArrayList:

```
int [] list = {1, 2, 3, 4, 5};

for (int num : list)
{
  System.out.println(num);
}
```

OUTPUT
1
2
3
4
5

48

Keyboard Input (the old way)

49

```
import java.io.*;

class Hello
{
  public static void main (String [] args)
   throws IOException
  {
    BufferedReader br = new BufferedReader( new
              InputStreamReader(System.in));

    System.out.print("How old are you? ");
    int age = Integer.parseInt(br.readLine());

    System.out.println("Wow, next year you'll "
                       + "be " + (age+1) + "!");
  }
}
```


Keyboard Input a la Java ≥1.5

50

```
import java.util.Scanner;

class Hello

{

  public static void main (String [] args)
  {
     Scanner keyboard = new Scanner(System.in);

     System.out.print("How old are you? ");
     int age = keyboard.nextInt();

     System.out.println("Wow, next year you'll "
                        + "be " + (age+1) + "!");

  }
}
```


Scanner Class

51

- Scanner helps "iterate" over collection of items
  - **`String nextLine()`**
    - Returns the remaining keyboard input line as a **String**. Note: '\n' is read and discarded.
  - **`boolean nextBoolean()`**
  - **`double nextDouble()`**
  - **`int nextInt()`**
  - others …
  - Preview: these methods throw an unchecked *InputMismatchException* if the next token isn't of the correct type.

More on the Scanner Class

52

- boolean methods hasNextInt(), etc. Returns true if the next "token" in this scanner's input can be interpreted as an int value in the specified radix using the nextInt() method. The scanner does not advance past any input.
- **`useDelimiter (String aString)`**
  - Makes aString the only *delimiter* used to separate input. *aString* is often a "regular expression."
- **`next()`**
  - Returns input up to, but not including, the first *delimiter* (any whitespace character is the *default*)

Program to Demonstrate useDelimiter()

53

```
import java.util.Scanner;
class Delimit
{  public static void main(String [] args)
   {
     Scanner in = new Scanner(System.in);
     String word1, word2;
     in.useDelimiter ("#");
     System.out.print ("Enter two words: ");
     word1 = in.next();   word2 = in.next();

     System.out.println ("FIRST you entered:  " + word1);
     System.out.println ("THEN you entered: " + word2);
   }
}
```

OUTPUT Example1

```
Enter two words: one#two#
one was entered first
Then you entered two
```

OUTPUT Example2

```
Enter two words: one
two#
three#
one
two was entered first
Then you entered
three
```


Regular Expressions (regex)

54

- A *regular expression* is a sequence of characters that define a search pattern. Usually this pattern is used by algorithms that perform "find" or "find and replace" operations, or for input validation. They contain both
  - literal characters
  - *meta*-characters such as: `*  ?  [  ]  |`
- In Delimit.java, try these delimiters: `##` `[#?\t]` `[#?\t]+`
- Write code to read "words" intelligently using a regular expression as delimiter. Then try using `" "` as delimiter.

Formatted Output Using printf

55

- System.out.printf ("***format string***", $expr_1$, $expr_2$, … $expr_n$);
  - The ***format string*** contains ordinary characters that get printed, and *placeholders* that tell how each value should be printed.
  - Each *placeholder* begins with a `%` character and ends with a letter that indicates the format type. For example, `%d` or `%x`
  - Flags, width, and .precision specifiers are options BEFORE the type that indicate exceptions to the default output. For example,

`%,d`	format as decimal integer with commas
`%-9.2f`	format as floating-point, left-justified with 2 digits of precision

55

printf Example

56

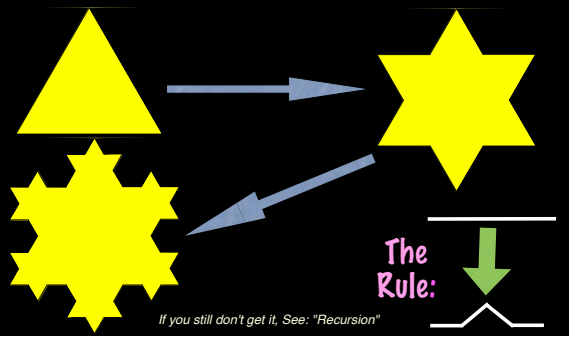
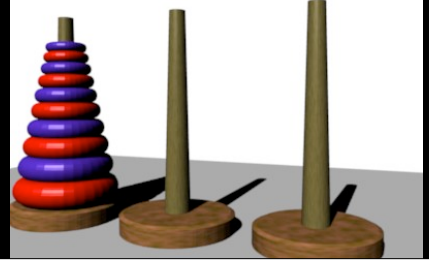
```
class PrintfDemo
{
   public static void main (String [] args)
   {
      System.out.printf ("I have %.2f bugs to fix!\n",
                          47568.09876);
      System.out.printf ("I have %,.2f bugs to fix!\n",
                         47568.09876);
      System.out.printf ("I have %,15.3f bugs to fix!\n",
                         47568.09876);
    }
}
```

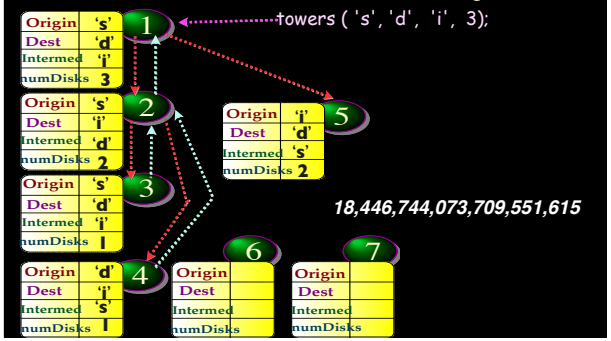
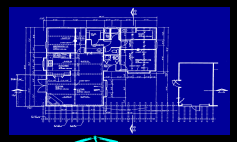
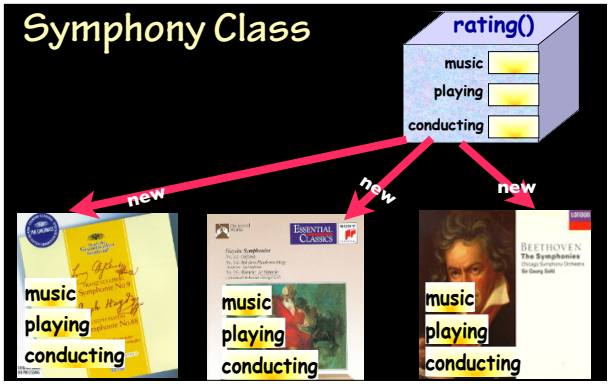
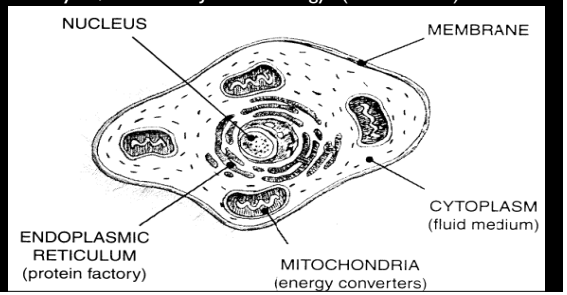
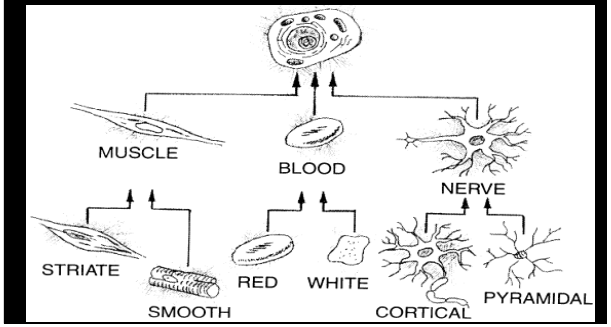
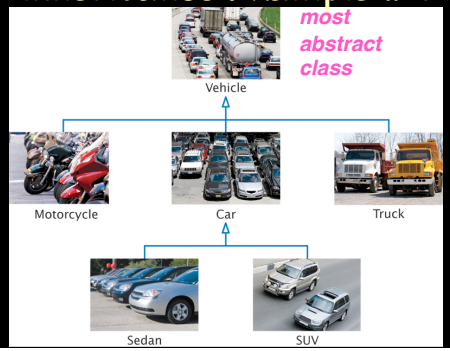
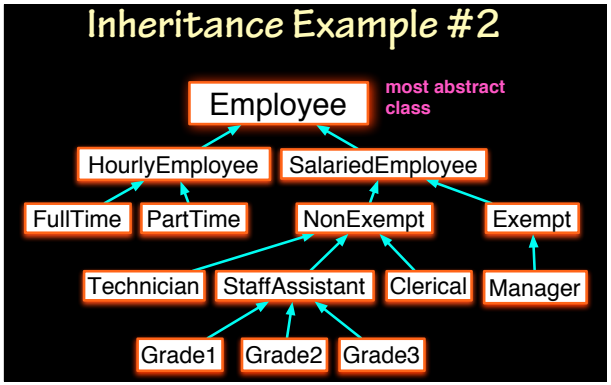
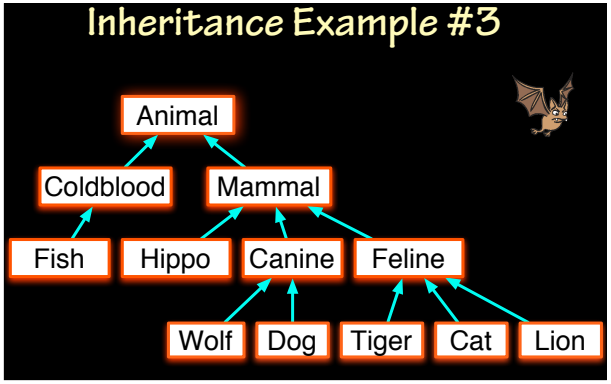
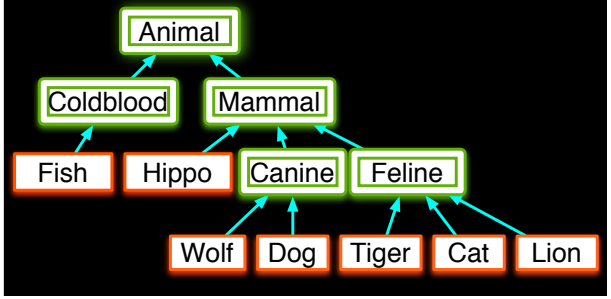
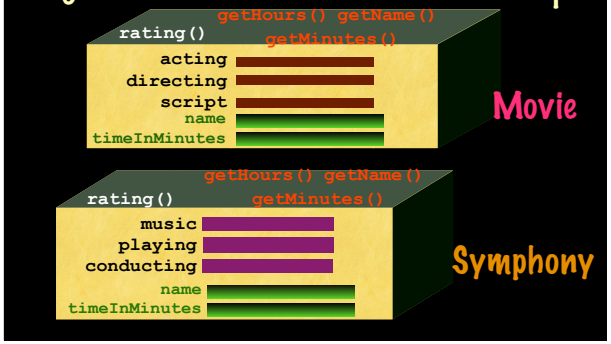
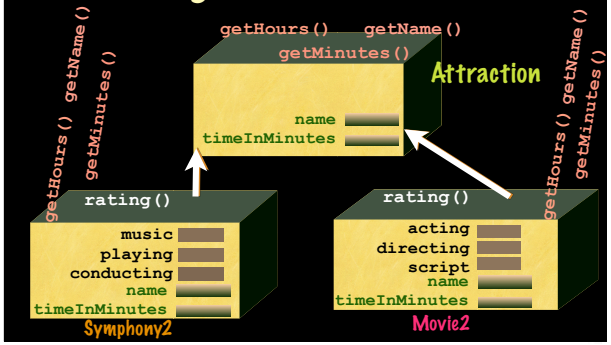
56

Simple Recursive Methods

57

- A ***recursive definition*** is a statement in which something is defined in terms of *smaller (simpler) versions of itself.*
- A recursive alternative to the built-in method `Math.pow` to compute x^n :
  - Base case:
  - Recursive case[s]:
  - ```
    static double power (double x, int n)
    {    // finish this off
    ```

57
    ```

Printing Strings in Reverse

58

```
public static void reverse (Scanner input)
 {
    if (input.hasNextLine())
    {
       String line = input.nextLine();
       reverse (input);
       System.out.println (line);
       return;
    }
    else return;
 }                                  // see file Reverse.java
```

Recursion in Koch Snowflake

59



When to Use Recursion

60

- Often, the decision to use recursion is suggested by the "appropriate" nature of the problem itself:
  - It must be possible to decompose the original into simpler instances of the same problem.
  - Once each of these simpler subproblems has been solved, it must be possible to combine these solutions to produce a solution to the original problem.
  - As the large problem is broken down into successively less complex ones, those subproblems must eventually become so simple that they can be solved without further subdivision.

60

General Scheme for Recursion

61

```
data-type recursiveSolve (instanceOfaProblem i)
{
   if (instance i is trivial)  { return (solve i directly); }
   else
    {
         Break instance i into smaller instances i1, i2, ... in
         data-type s1 = recursiveSolve (i1);
         data-type s2 = recursiveSolve (i2);
                  …
         return the result of combining/assembling the sn solutions;
    }
}
```


Towers of Hanoi

62

- The book Mathematical Recreations and Essays (11th ed.) gives a history of the puzzle, introduced in 1883:
  - *"In the great temple of Benares … rests a brass plate in which are fixed 3 diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed 64 disks of pure gold, the largest disk resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Brahmah. Day and night unceasingly the priests transfer the disks from one diamond needle to another according to the fixed and immutable laws of Brahmah, which require that the priest on duty must not move more than 1 disk at a time, and that he must place this disk on a needle so that there is no smaller disk below it. When the 64 disks shall have been thus transferred from the needle on which at the creation God placed them, to one of the other needles, tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish!"*

Towers of Hanoi Model

63

18,446,744,073,709,551,615

http://haubergs.com/hanoi



Towers of Hanoi, continued

- Recursive algorithm:

```
static void towers(char origin, char dest,
           char intermed, int numDisks)
{
   if ( numDisks == 1 )   // base case
      System.out.println ("Move disk #1  from "
              + origin + " to " +
                     dest );
   else
   {
      towers (                                    );
      System.out.println ("Move disk #"
               + numDisks + " from " + origin +
               " to " + dest);

      towers (                                    );
   }
```


Flow of Control in Hanoi.java



Numbers as Character Strings

- To recursively print the int n in decimal as a character sequence, view n as consisting of the units digit (n % 10), and everything else (n / 10):
  - "if needed," print (n/10) recursively
  - print the units digit, (n % 10)
- Now substitute any "base" from binary to decimal for 10, and write the whole thing in Java:
  -
```
static void writeBase (int n, int base)
{  if (n >= base)  writeBase (
     char [] digits = {'0', '1', '2', '3', '4',
                       '5', '6', '7', '8', '9'};
     System.out.print ( digits [n % base] );
}
```

Classes and Objects Revisited

67

- **class**: A program entity that represents
  - A program, *or*
  - A template for a new type of objects. E.g., the `Movie` class is a template for creating new `Movie` objects.
- **object**: An entity that combines state and behavior.
- ***object-oriented programming*** (**OOP**): Programs that perform their behavior as interactions between objects.

iPod Blueprint Analogy

68

state:
current song
volume
battery life

behavior:
power on/off
change station/song
change volume
choose random song



iPod #1
state:
song = "True Romance"
volume = 17
battery life = 12.5 hrs
behavior:
power on/off
change station/
song

iPod #2
state:
song = "Letting You"
volume = 9
battery life = 3.41 hrs
behavior:
power on/off
change station/
song

iPod #3
state:
song = "Discipline"
volume = 24
battery life = 1.8 hrs
behavior:
power on/off
change station/
song

The Movie Class

69

Symphony Class



70

What Goes in a Class Definition?

- Class definitions can include class and instance variables/constants plus class and instance methods. Each can have "public" or "private" access. For example,

```
class Symphony
 {
      private int music, playing, conducting;                  // instance variables
       private static int numberOfSymphonies = 0;              // class variable
      public Symphony (int m, int p, int c)                   // constructor
        { music = m;  playing = p;  conducting = c;  numberOfSymphonies++; }
      public static int  getNumberOfSymphonies ()
        {   return numberOfSymphonies;   }                    //  class method
     public double rating (double scale)                      // instance method
        {   return ( music +  playing + conducting) * scale;  }
 }
```

71

71

72

Objects as Biological Cells

73

- See Taylor, David: *Object Technology* (2nd edition)



Cell Hierarchies and Inheritance

74



What is Inheritance?

75

- **inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior.
  - a way to group related classes
  - a way to share code between ≥ 2 classes
- One class can *extend* another, absorbing its data/behavior.
  - **superclass**: The parent class being extended.
  - **subclass**: The child class that extends the superclass and inherits its behavior.
  - Subclass gets a copy of every *non-private* field and method from superclass

Inheritance Example #1

76



Inheritance Example #2

77



Inheritance Example #3

78



Abstract Classes (a Preview)

79



Why Use Inheritance? An Example

80



Inheritance Example, continued

81

- Suppose both the **Movie** class and the **Symphony** class each contain **timeInMinutes** and **name** instance variables, along with companion methods **getMinutes()**, **setMinutes()**, etc.
  - The problem is that these instance variables and instance methods are exact duplicates in both classes. *Maintaining multiple copies makes software development and maintenance difficult as you try to correct bugs, add features, improve performance, change behavior, etc.*
- Instead: tie together classes in hierarchies such that instance features declared in one class automatically appear in instances belonging to another; this allows one to create "natural "category hierarchies by creating a subclass to direct-superclass relationship, using the pattern:

```
class SubclassName   extends
   SuperclassName  {   …   }
```

Movies and Symphonies are Attractions

82

- For example,
  `public class Movie2 extends Attraction { … }`
- One can say that movies and symphonies are both "attractions," and define both instance variables and appropriate *getter*/*setter* methods just in this class:
- ```
  public class Attraction
  {  protected String name;
     protected int timeInMinutes;
     public Attraction () { timeInMinutes = 75;}
     public int getHours () {  ????   }
     public void setMinutes(int d) {minutes=d; }
  }
  ```
- A class will inherit *public* and *protected* instance variables and methods from all of its superclasses.
- See .java files Attraction, Movie2, Symphony2, and DemonstrateInheritance

82

Extending the Attraction Class

83



Shadowing

84

- A class can have only one immediate superclass.
- When a subclass-superclass chain contains multiple instance methods with the same name, argument number, and argument types, the one closest to the target instance in the subclass-superclass chain is the one executed. All others are shadowed / overridden. Consider
  - ```
    public class StevenSpielbergMovie extends  Movie2
    {
       public int rating()  { return 10 + acting +script; }
    }
    ```
  - ```
    StevenSpielbergMovie m = new StevenSpielbergMovie();
    int i = m.rating();        // which rating method is invoked?
    ```

84
    ```

Abstract Classes

85

- To prevent the creation of instances of the **Attraction** class, define it as **abstract**.
- You can, however, create **Attraction** variables and assign to them either **Movie2** or **Symphony2** instances.
  - WHY is that legal?
- When you mark a class with the **final** keyword, such a class cannot be extended.
  - All classes form an inverted tree with the **Object** class at the root. **Final** classes appear as leaves.

85

Abstract Methods

86

- Problem identified:
  - `Attraction x = new Movie2();`
    `System.out.println (x.rating());` // bug
- We need to define method rating() for **Attractions**
  - Undesirable, since this **rating()** method is never called.
  - Fortunately, Java allows you to define rating() as an **abstract** method of the abstract **Attraction** class: `public abstract int rating();`
  - Once you have defined an abstract method, Java forces you to define corresponding non-abstract methods in certain sub-classes of the abstract class.

86

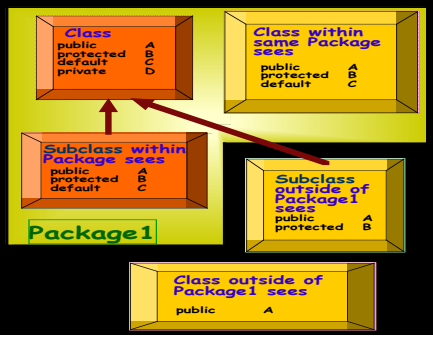
It's All So … final

87

- A final variable means you *cannot change* its value; this includes parameters as well as non-static variables
- A final method means you *cannot override* the method
- A final class means you *cannot extend* the class (i.e., you can't make a subclass)

Scoping Visibility Rules

88



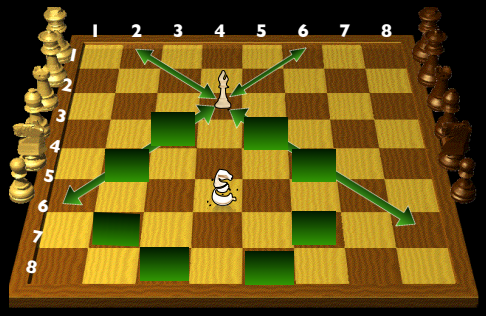
Access Specifiers

89

- A **public** feature (data or method) can be accessed outside the class definition. A public class can be accessed outside the package in which it's declared.
- A **protected** feature can be accessed only within the class definition in which it appears, within other classes in the same package, or within the definition of subclasses.
- A **private** feature can be accessed only within the class definition in which it appears.
- A **default** feature can be accessed by subclasses and by other classes in the same package.

Placing a Piece on a Chessboard

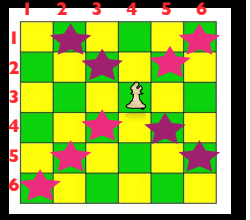
90



How a Bishop Attacks

- For convenience, define
  - `columnDiff = pieceColumn - indexColumn;`
  - `rowDiff = pieceRow - indexRow;`
- Then the diagonal `\` satisfies …
- And the diagonal `/` satisfies …
- For other squares,

```
if  (    // ???
      System.out.print(" B");
else  System.out.print(" W");
```



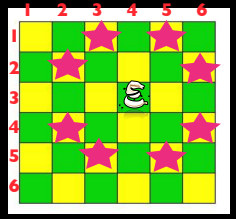
91

91

How a Knight Attacks

- Again we calculate
  - `columnDiff = pieceColumn - indexColumn;`
  - `rowDiff = pieceRow - indexRow;`
- Now the condition for printing a * is
  - `if (`

```
        System.out.print( " *" );
```



92

92

Constructors and Inheritance

- A child class can add new data and methods, but can also **override** methods defined in the parent class (so long as the name, return type and arguments match).
- When you want a constructor to hand one or more arguments to another constructor in the direct superclass, you put as the FIRST statement in the subclass constructor a statement consisting of
  - `super (`*argument list for parent's constructor*`);`
- If, on the other hand, you want a constructor to call another constructor IN THE SAME CLASS explicitly:
  - `this (`*argument list for other constructor*`);`

93

93

The super Constructor

94

- A call to the base class constructor can never use the name of the base class, but uses the keyword `super` instead
- A call to `super` must always be the first action taken in a constructor definition
- An instance variable cannot be used as an argument to `super`
- See files `C.java` and `SuperDemo.java`

The super Constructor, *cont'd.*

95

- If a derived class constructor does not include an invocation of `super`, then the no-argument constructor of the base class will automatically be invoked
  - This can result in an error if the base class has not defined a no-argument constructor
- Since the inherited instance variables should be initialized, and the base class constructor is designed to do that, then an explicit call to `super` should always be used
- NOTE: SUPER CAN BE USED TO INVOKE ORDINARY (non-constructor) METHODS IN A SUPERCLASS

The this Constructor

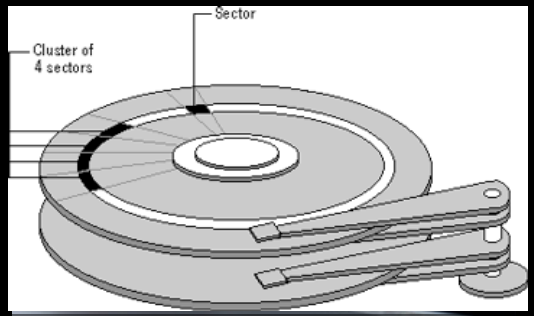
96

- Within the definition of a constructor for a class, `this` can be used as a name for invoking another constructor in the same class
  - The same restrictions on how to use a call to `super` apply to the `this` constructor
- If it is necessary to include a call to both `super` and `this`, the call using `this` must be made first, and then the constructor that is called must call `super` as its first action

File Input and Output

97

Internal View of Hard Drive



98

The Original Hard Drive

1956: IBM ships the first hard drive in the RAMAC 305 system. The drive holds 5MB of data at $10,000 a megabyte. The system is as big as two refrigerators and uses 50 24-inch platters.

99

File Input/Output (I/O)

100

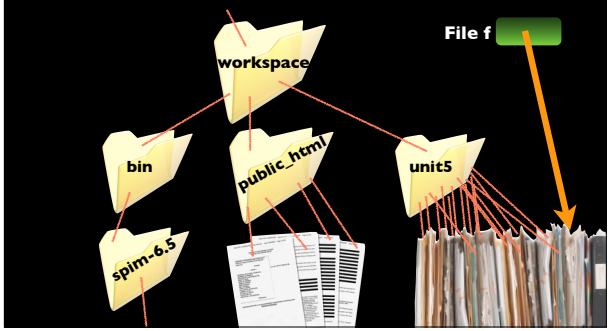
```
import java.io.*;
```

- Create a `File` object to get info about a file on your drive (without actually creating a new file on the hard disk.)

```
{
    File f = new File ("data.txt");
    if (f.exists() && f.canRead()) {
       System.out.println("length = "
                          + f.length());
    }
}
```

Files and Directories Example

101



Useful Methods of File Objects

102

Method	Description
isDirectory ()	true if file represents a directory/folder
isFile ()	true if file represents an actual file
getAbsolutePath ()	full path of file's location
getName ()	pathless name of file as String
delete()	deletes given file or directory
listFiles ()	returns array of File objects from dir
length()	# of characters in this file
mkdirs()	creates directory, if it doesn't exist
renameTo(newName)	changes this file's name to newName

Reading Files

103

- To read a file, pass a `File` when constructing a `Scanner`.

```
Scanner variable = new Scanner (new
                   File ("actual file name"));
```

  - Example:

```
File f = new File ("mydata.txt");
Scanner input = new Scanner (f);
```

  - or (shorter):

```
Scanner input = new Scanner (new
                File ("mydata.txt"));
```

103

File Paths

104

- **absolute path**: specifies a drive or a top "/" folder, e.g.,

```
C:/Documents/smith/hw6/stuff.txt
```

- **relative path**: does not specify any top-level folder, e.g.

```
mydata.txt
```

or

```
input/kinglear.txt
```

  - Assumed to be relative to the *current directory*: `Scanner input = new Scanner (new File("data/readme.txt"));`
  - If our program is in `H:/hw6`, `Scanner` will look for `H:/hw6/data/readme.txt`

104

Compiler Error with Files

105

```
import java.io.*;     // for File
import java.util.*;   // for Scanner
public class ReadFile {
    public static void main(String[] args) {
        Scanner input = new Scanner (new
                        File ("data.txt"));
        String text = input.next();
        System.out.println(text);
    }
}
```

- The compiler outputs the following error:

```
ReadFile.java:6: unreported exception
  java.io.FileNotFoundException; must be caught or declared
  to be thrown Scanner input = new Scanner(new
  File("data.txt"));
```

105

Exceptions

- exception: An object representing a …
  - dividing …
  - calling `substring` on a `String` and …
  - attempting to read the wrong type of value from …
  - trying to read a file that does not exist
- We say that a program with an error "*throws*" an exception.
  - It is also possible to "*catch*" (handle) an exception.
- checked exception: An error that must be handled by our program — or else!

106

The `throws` clause

- **`throws` clause**: Keywords on a method's header that state that it may generate an exception (and will not handle it).
  - Syntax:

```
static type methodName (params) throws Exceptiontype { ...
```

  - Example:

```
static void foobar (String [] s, int q)
  throws FileNotFoundException { …
```

  - Like saying, *"I hereby announce that this method might throw a FileNotFoundException, and I accept the consequences if this happens."*

107

Easy File I/O: CopyFile.java

- To read text: use Scanner & File classes
  - File reader = new File ("input.txt");
    Scanner in = new Scanner (reader);
  - Use the Scanner methods to read data from file: next(), nextLine(), nextInt(), and nextDouble()
- To write to a file, construct a PrintWriter:
  - PrintWriter out = new PrintWriter ("output.txt");
  - If file already exists, it is emptied before the new data are written into it; if file doesn't exist, an empty file is created
  - Use print and println to write into a PrintWriter
  - You must close a file when you are done processing it!

108

Input tokens

- **token**: A unit of user input, separated by whitespace.
  - A `Scanner` splits a file's contents into tokens.
- If an input file contains the following:

```
23   3.14
  "John Smith"
```

The `Scanner` can interpret the tokens as:

Token	Type(s)
23	int, double, String
3.14	double, String
"John	String
Smith"	String

Files and input cursor

- Consider a file `weather.txt` that contains this text:

```
  16.2  23.5
19.1 7.4  22.8
 18.5 -1.8 14.9
```

- A `Scanner` views all input as a stream of characters:

```
16.2  23.5\n19.1 7.4  22.8\n 18.5 -1.8 14.9\n
^
```

- **input cursor**: The current position of the `Scanner`.

Consuming Tokens

- **consuming input**: Reading input and advancing the cursor.
  - Calling `nextInt()`, etc. moves the cursor past the current token. For example,

```
  16.2  23.5\n19.1 7.4  22.8\n 18.5 -1.8 14.9\n
```



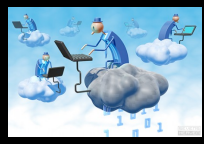
```
double d = input.nextDouble();    // 16.2
String s = input.next();    // "23.5"
```

File Input Problem

112

- Consider input file `weather.txt`:

```
16.2   23.5
    19.1 7.4  22.8
 18.5  -1.8   14.9
```



- Write a program that prints the change in temperature between each pair of neighboring days.

```
16.2 to 23.5, change = 7.3
23.5 to 19.1, change = -4.4
19.1 to 7.4, change = -11.7
...
```


Now to Actually Handle Errors

113

- Traditional approach: Method returns error code
  - Problem: Forget to check for error code, so failure notification may go undetected
  - Problem: Calling method may not be able to do anything about failure, so program must fail too and let its caller worry about it. Also many method calls would need to be checked
- Instead of programming for success, as in `x.doSomething()` you oftentimes program for failure:

```
if (!x.doSomething())
   { System.out.print
      ( "Aaaaarrrgggggghhh!!!" ); return false; }
```


Throwing an Exception

114

```
if (amount > balance)
{
    throw new IllegalArgumentException (
          "Amount exceeds balance!");
}
balance = balance - amount;
```

This line is not executed when exception is thrown

When you throw an exception, the normal flow of control is terminated.

115

"Catching" an Exception

116

- try
  {
  Scanner in = new Scanner (new File ("x.dat"));
  String input = in.next();
  process (input);
  }
  catch (IOException e)
  {
  System.out.println ("Cannot open file!");
  }

Constructor can throw FileNotFoundException

When IOException is thrown, execution resumes here

Catching Multiple Exceptions

117

- Surround Java code that can throw exceptions with a "try block" …

```
try
{
   String filename = . . .;
   Scanner in = new Scanner(new File(filename));
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
   System.out.println("Input was not a number");
}
```

FileNotFoundException

NoSuchElementException

NumberFormatException

Write 'catch blocks' for each possible exception.

It is customary to name the exception parameter either 'e' or 'exception' in the catch block. See ***Exc.java***

The Ariane Rocket Incident

118

Unmanned European rocket expl on first flight

June 4, 1996
Web posted at: 12:00 p.m. EDT (1600 GMT)

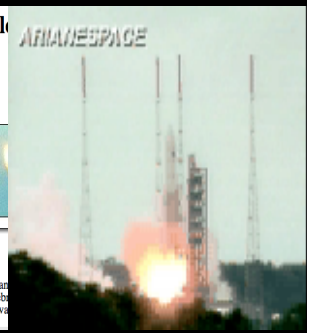
KOUROU, French Guiana
(CNN) -- Europe's newest unmanned satellite-launching rocket, the Ariane 5, intentionally was blown up Tuesday just seconds after taking off on its maiden flight.


The rocket exploded seconds after launching

A spokesman for Arianespace said the rocket was destroyed by its controllers.

RELATED SITES & STORIES

"It's been confirmed that the vehicle was deliberately destroyed by the safety people," said spokesman "Why? I don't know. There were no, repeat no, injuries. Most of the debr down in the mangroves and the sea, and the winds carried the fumes awa people out over the sea."



Creating Your Own Exceptions

119

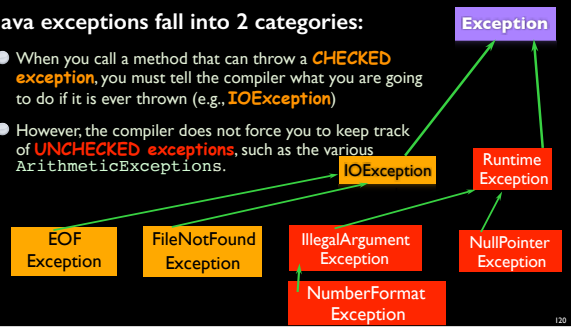
- See `ArrayExpansionDemo.java`
- To shut a program down, use `System.exit(0);`
- Create (and throw) your own exceptions:
  - Define: `class StrangeNewException extends Exception { }`
  - `throw (new StrangeNewException () )`
  - `catch (StrangeNewException e) { … }`
- See programs `E.java`, `E2.java`

119

[Un]checked Exceptions

120

- Java exceptions fall into 2 categories:
  - When you call a method that can throw a **CHECKED exception**, you must tell the compiler what you are going to do if it is ever thrown (e.g., **IOException**)
  - However, the compiler does not force you to keep track of **UNCHECKED exceptions**, such as the various `ArithmeticExceptions`.



120

[Un]checked Exceptions, cont'd.

121

- Categories aren't perfect; e.g.,
  - `Scanner's nextInt()` throws unchecked `InputMismatchException`
  - Programmer cannot prevent users from entering incorrect input, but this choice makes the class easy to use for beginning programmers
- Deal with checked exceptions principally when programming with files and "streams"
  - For example, you need to deal with the FileNotFoundException when you attempt to open a file for "reading" information.

PRIMITIVE Text File Input

122

- To read bytes from an input file, use pattern
  - `FileInputStream` *stream variable* `=`
    `new FileInputStream ("file specification");`
- To read characters from an input file,
  - `InputStreamReader` *reader variable* `=`
    `new InputStreamReader(`*stream variable*`);`
- To read characters more efficiently (see *IO.java*),
  - `BufferedReader` *bufReader variable* `=`
    `new BufferedReader(`*reader variable*`)`
- You still need to catch I/O exceptions; e.g., use `FileNotFoundException` or `IOException` class.

http://docs.oracle.com/javase/7/docs/api/index.html

123

java.io

Class InputStreamReader

java.lang.Object
  java.io.Reader
    java.io.InputStreamReader

All Implemented Interfaces:

Closeable, AutoCloseable, Readable

Direct Known Subclasses:

FileReader

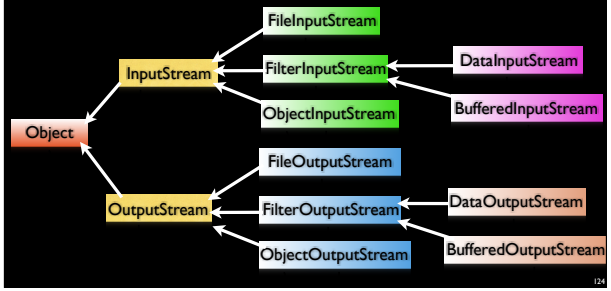
---

public class InputStreamReader

Part of the java.io Hierarchy

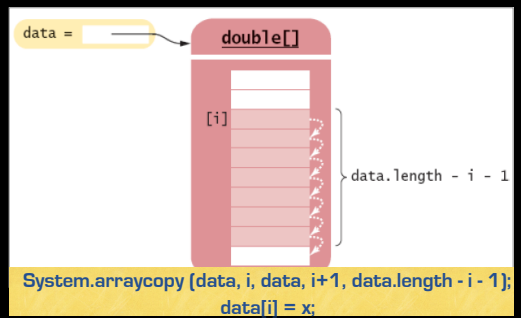
124

- Binary I/O Streams in Java (see Binary.java)



Array Difficulties: Inserting

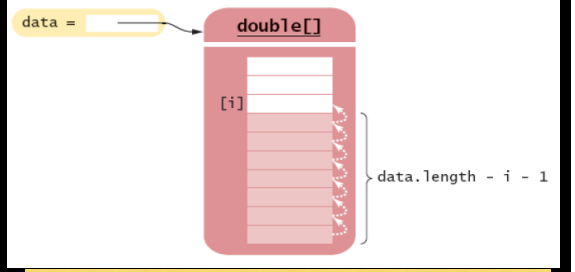
125




```
System.arraycopy (data, i, data, i+1, data.length - i - 1);
data[i] = x;
```

Array Difficulties: Removing

126




```
System.arraycopy (data, i + 1, data, i, data.length - i - 1);
```

Array Difficulties: Growing

127

- If the array is full and you need more space, you can "grow" the array:

1. Create a new, larger array.

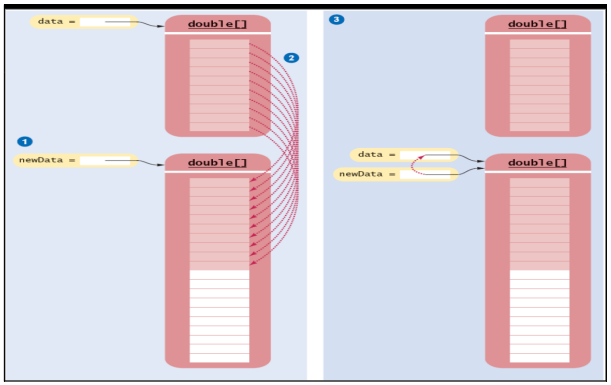
```
double[] newData = new double[2 * data.length];
```

2. Copy all elements into the new array

```
System.arraycopy(data, 0,newData, 0,data.length);
```

3. Store the reference to the new array in the array variable

```
data = newData;
```



128

ArrayLists: an Array Alternative

129

- The ArrayList class manages a sequence of objects
- Can grow and shrink as needed
- ArrayList class supplies methods for many common tasks, such as inserting and removing elements
- The ArrayList class is a *generic* class: ArrayList<T> collects objects of type T:
  - `ArrayList<Movie2> myMovies = new ArrayList<Movie2>();`

ArrayList Methods

130

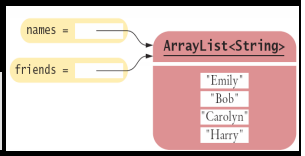
- To insert new objects into an *ArrayList*, use method *add*:
  - **myMovies.add (new Movie2 ("The Shape of Water"));**
  - **myMovies.add (new Movie2 ("The Post"));**
  - **myMovies.add (1, new Movie2 ("Fifty Shades of ... Java"));**
- `size()` *yields number of elements*
- `get()` *retrieves one element*
  - **index starts at 0**
  - **IndexOutOfBoundsException can occur**
  - **see file *Remove.java* (which contains a subtle bug)**

Copying an ArrayList

131

- Remember that *ArrayList* variables hold a reference to an *ArrayList* (just like arrays)
- Copying a reference:

```
ArrayList<String> friends =
            names;
friends.add("Harry");
```



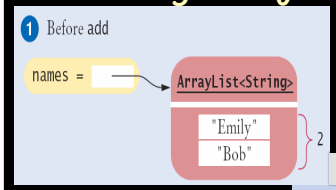
- To make a copy, pass the reference of the original *ArrayList* to the constructor of the new one:

reference

```
ArrayList<String> newNames = new ArrayList<String>(names);
```

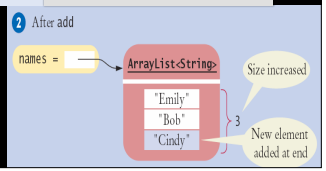
Adding ArrayList Elements

132

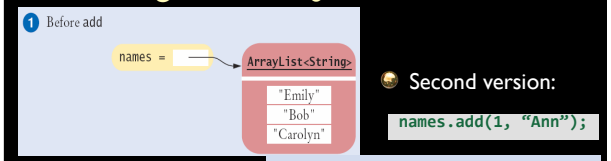



```
names.add("Cindy");
```

- The *add* method has two versions. First version: pass a new element to add to the end



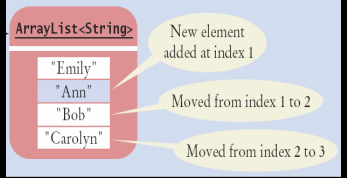
Adding an ArrayList Element



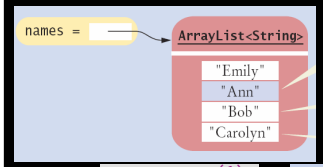
- Second version:

```
names.add(1, "Ann");
```

- Pass a location (index) and the new value to add
- Moves all other elements

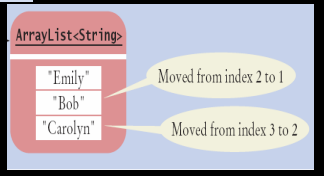


Removing ArrayList Elements




```
names.remove(1);
```

- Pass a location (index) to be removed; all other elements are then moved



ArrayList Methods, *continued*

- `set (int index, E element)` **overwrites an existing value.**
- `remove (int index)` **removes an element**
- **isEmpty () returns true if list has no elements**
- **contains (Object element) returns true if the list contains the specified element**
- **toArray () returns an array of Object with all of the elements of the ArrayList**

See ArrayListDemo.java

Using Vectors (see VectorDemo.java)

136

- The `Vector` type is very similar to an ArrayList but might be slightly more inefficient due to synchronization. It has some additional methods:
  - `Vector<type>` *vector name* `=` `new Vector<type>();`
- To add elements into a vector, use one of
  - *vector name*`.addElement(`*object*`)` `// or add`
  - *vector name*`.insertElementAt(`*object*`,0)`
  - *vector name*`.insertElementAt(`*object*`,`*index*`)`
- To retrieve an element from the front or back,
  - *vector name*`.firstElement()`
  - *vector name*`.lastElement()`

136

Using Vectors, continued

137

- To retrieve an element or replace an element, use
  - *vector name*`.elementAt(`*index*`)` `// or get`
  - *vector name*`.setElementAt(`*anObject*`,` *index*`)`
- To know the number of elements in a vector,
  - *vector name*`.size()`
- Elements of a vector might be instances of the `Object` class, so you may need to do casting:
  - `(` `(`*class name*`)` *vector element* `)`
- To use every element of a vector in turn,
  - `for ( Object o :` *vector name*`) { … }`

137

Stacks

138

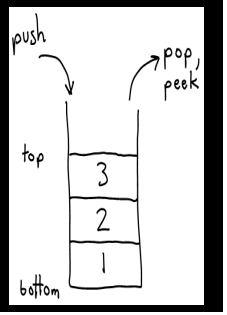
- A collection based on the principle of adding elements and retrieving them in the opposite order.

138

Stack Operations

139

- Basic stack operations:
  - **push**: Add an element to the top.
  - **pop**: Remove the top element.
- Last-In, First-Out ("LIFO")
- The elements are stored in order of insertion, but we do not think of them as having indexes.
- We can only add/remove/examine the last element added (the "top").



Inheritance vs. Composition

140

- If you find yourself using the phrase an X is a Y when describing the relation between two classes, then the first class is a subclass of the second.
- If you find yourself using the phrase an X has a Y when describing the relation between two classes, then instances of the second class appear as parts of instances of the first class.
- Deciding between these two relations is not always straightforward. See Stack.java and Stack2.java and StackDemo.java

Stacks in Computer Science

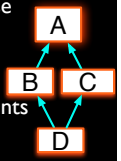
141

- Programming languages and compilers:
  - method calls are placed onto a stack (call=push, return=pop)
  - compilers use stacks to evaluate expressions
- Matching up related pairs of things:
  - find out whether a string is a palindrome *see ParenChecker.java*
  - examine a file to see if its braces { } and other operators match
  - convert "infix" expressions to "postfix" or "prefix"
- Sophisticated algorithms:
  - searching through a maze with "backtracking"
  - many programs use an "undo stack" of previous operations

Interfaces

142

- Java does not allow "multiple inheritance" because it introduces problems as well as benefits. E.g.,



- Fortunately, Java allows you to impose requirements on a class from multiple classlike interfaces. An interface is like an abstract class: it can hold abstract method definitions that force other classes to implement them. But it has NO instance variables (though it can have static final members).
  - All methods in an interface are abstract (they each have a name, parameters, and a return type, but no implementation)
  - All methods in an interface are automatically public.

142

Interface Example

143

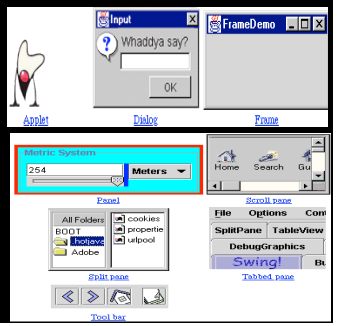
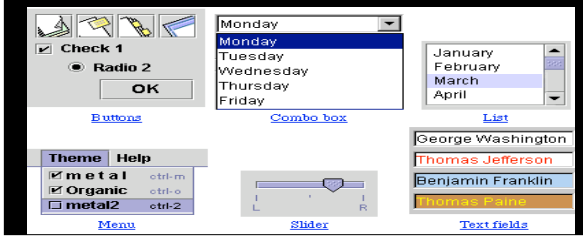
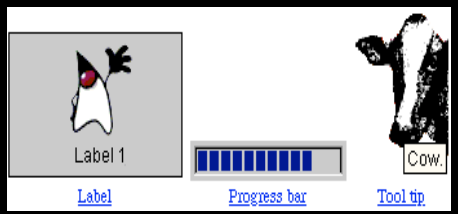
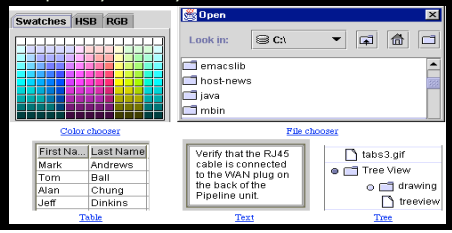
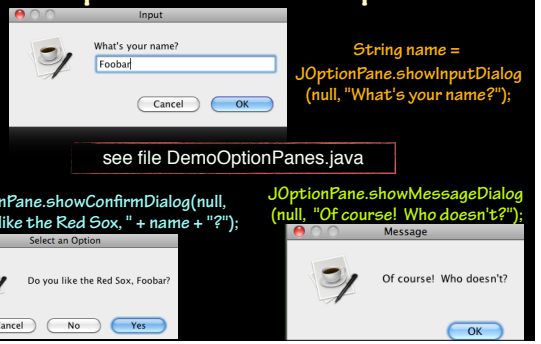
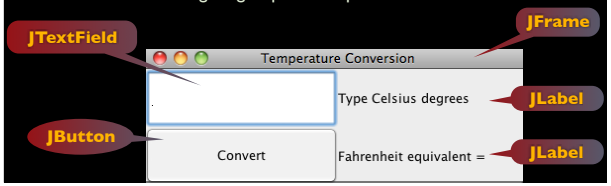
- For example, the java.lang package defines a Comparable interface as
  - `public interface Comparable<T>`
    `{  public int compareTo (T other); }` *// no implementation*
- If you want an interface to impose requirements on a particular class, don't extend it; instead implement it:
  - `public class` *someClassName* `implements` *InterfaceName1, InterfaceName2* `{ … }`
  - ```
    public class Movie3 extends Attraction
          implements Comparable<Movie3> {  …
      public int compareTo (Movie3 otherMovie)
      { if (rating() < otherMovie.rating()) return -1;
        else if (rating() > otherMovie.rating()) return 1;
        else return 0;  }    …  }
    ```

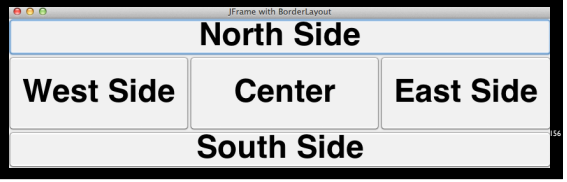
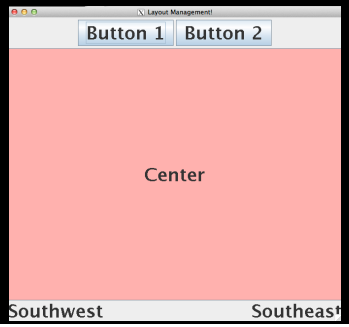
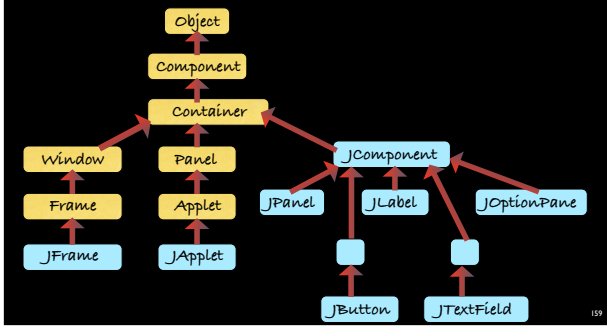
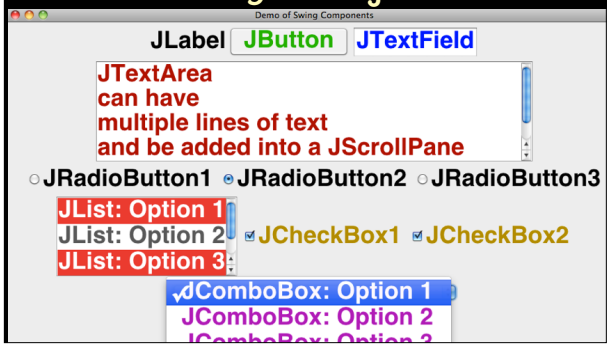
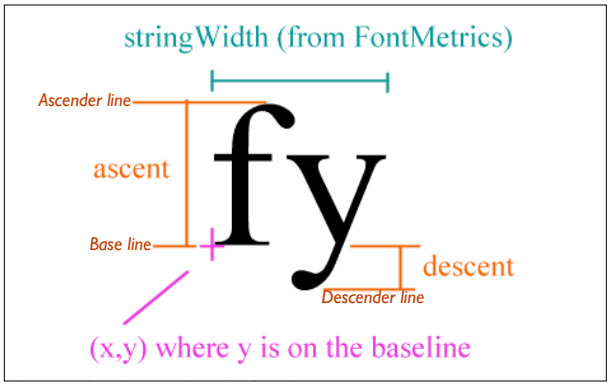
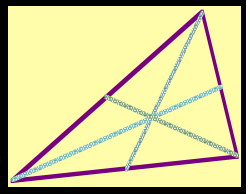
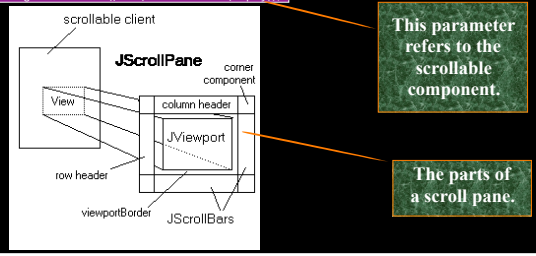
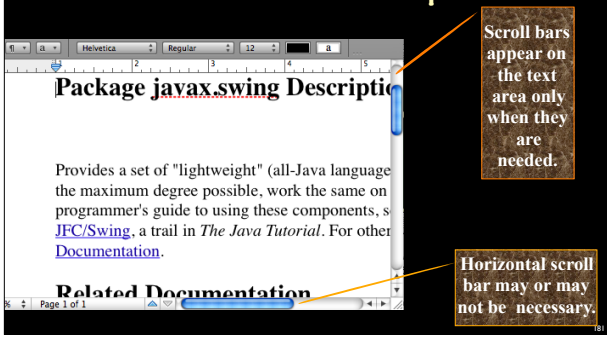
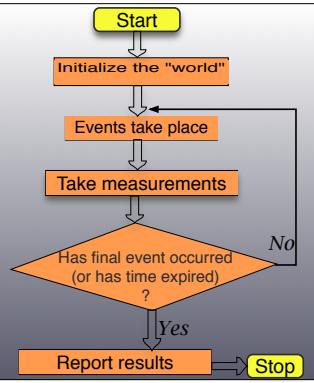
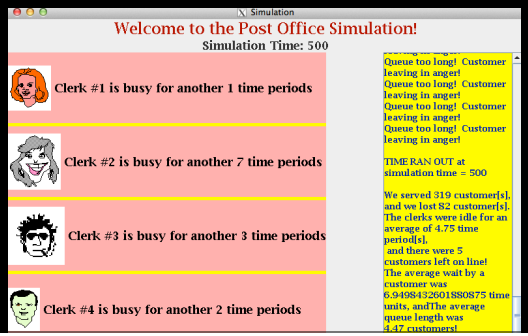
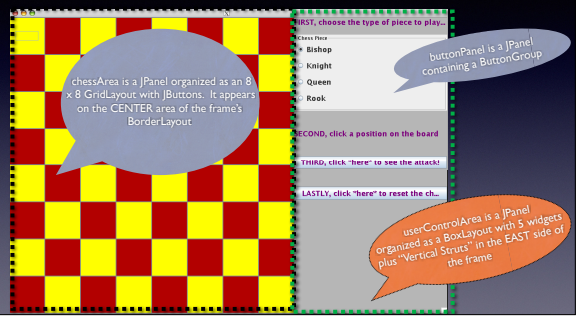
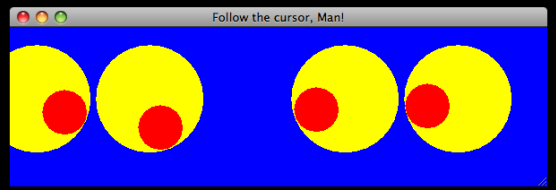
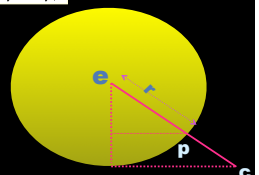
143

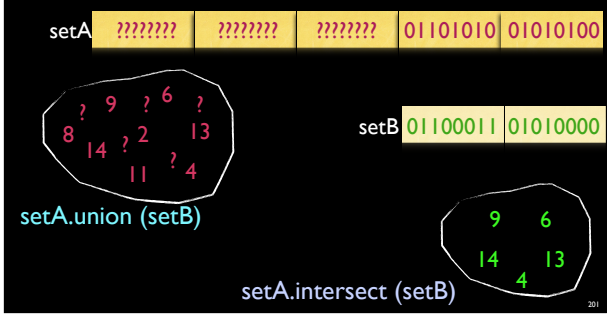
The Serializable Interface

144

- You can save an object's state in a file by "serializing" the object. (See Serial.java)
  - You need an ObjectOutputStream from java.io constructed from a FileOutputStream
  - Then use method writeObject (*anObject*)
- Restore serialized objects by constructing an ObjectInputStream from a FileInputStream
  - Use method readObject() and do a "typecast"


    ```

Building GUIs in Java

145

- A graphical user interface (GUI) presents a pictorial interface to a program, giving it a distinctive "look and feel." Consistent and intuitive GUIs can help individuals to use programs more productively and to learn new apps faster.
- GUIs are built from components (or "widgets") — visual objects the user interacts with via the mouse or keyboard.
  - **Label**: An area where uneditable text can be displayed
  - **Button**: An area that triggers an event when clicked
  - **TextField**: An area in which the user inputs data using the keyboard; it can also display information
  - **Choice**: A drop-down list of items user can select from

GUI Elements

146

- **Top-Level Containers**
  - The components at the top of any GUI containment hierarchy.
- **General-Purpose Containers**
  - Intermediate containers that can be used under many different circumstances.



GUI Widgets

147

- **Basic Controls**
  - Atomic components that exist primarily to get input from the user; they generally also show simple state.



GUI Widgets, continued

148

- **Uneditable Information Displays**
  - Atomic components that exist solely to give the user information.



Editable GUI Elements

149

- **Editable Displays of Formatted Information**
  - Atomic components that display highly formatted information that can be optionally edited by the user.



Simplest Graphical Container

150

- An "option pane" is a simple graphical box that appears on screen to present a message or to request input from the user. The JOptionPane class has methods:

`showMessageDialog (parent, message)`	*displays message string in a box on the screen*
`showInputDialog (parent, message)`	*displays input box with message & returns user input as a String*
`showConfirmDialog (parent, message)`	*displays Yes/No/Cancel box with given message and returns one of 3 possible constants*

JOptionPane Examples



String name =
JOptionPane.showInputDialog
(null, "What's your name?");

see file DemoOptionPanes.java

JOptionPane.showConfirmDialog(null,
"Do you like the Red Sox, " + name + "?");

JOptionPane.showMessageDialog
(null, "Of course! Who doesn't?");

Onscreen GUI Elements

- Most GUIs are <u>not</u> composed of *option panes*; they are too limited. Instead, complex GUIs contain:
  - **frame**: A graphical window on the screen.
  - **components**: GUI widgets such as buttons or text fields.
  - **containers**: Logical groups of components.



Opening a "Window"


```
import javax.swing.JFrame;    // a subclass of java.awt.Frame

public class FrameTest
{
   public static void main(String[] args)
   {  JFrame frame = new JFrame ("Frame Demo");
      frame.setSize(300, 320);
      frame.setVisible(true);
   }
}
```

JFrames, continued

154

- An ins… …uch as a border… Applic…
- Some J…
  - Cons…
  - set… …ot)
  - set…
  - set… …t
  - set…
  - set…

More on JFrames

155

- In Java, a **`Container`** is a **`Component`** that can contain other **`Component`**s. JFrames inherit methods from several superclasses:

```
java.lang.Object
   |
   +--java.awt.Component
         |
         +--java.awt.Container
               |
               +--java.awt.Window
                     |
                     +--java.awt.Frame
                           |
                           +--javax.swing.JFrame
```

- The hard work of organizing elements within a container is the task of the *layout manager*. It determines: the overall size of the container, the size of each element in the container, and the spacing and positioning of the elements.

java.awt.BorderLayout

156

- Every container, by default, has a layout manager — an object that implements the LayoutManager interface — but you can easily replace it with another one.
  - **java.awt.BorderLayout**: arranges elements along the north, south, east, west, and center of the container. All extra space is placed in the center area. See program BorderLayoutDemo.java



Other Layout Managers

157

- **javax.swing.BoxLayout**: arranges elements in 1 row or 1 column.
- **java.awt.CardLayout**: arranges elements like a stack of cards, with one visible at a time
- **java.awt.FlowLayout**: arranges left to right, top-bottom
- **java.awt.GridLayout**: arranges elements in a 2-dimensional grid of equally sized cells
- **java.awt.GridBagLayout**: arranges elements in a grid of variable-sized cells (complicated)
- Note: The Swing `Border` and `BorderFactory` classes can be used to put borders around almost any GUI element — this can make groupings of components more apparent and help guide (and inform) the user.
- See programs LayoutManagerDemo.java and GridBagLayoutDemo.java

Layout Mystery Problem

158



Partial Class Hierarchy

159



Methods of a Component

160

- A *component* (e.g., JButton) is an object having a graphical representation that can be displayed on the screen and can interact with the user.
  - `Color getForeground ()`
  - `Color getBackground ()`
  - `Font getFont()`
  - `void setBackground (Color c)`
  - `void setForeground (Color c)`
  - `void setFont (Font f)`
  - `boolean isVisible()`
  - `void setVisible (boolean b)`
  - `void setSize (int width, int height)`
  - `void setName (String name)`

Color Class: *See program* ColorDemo.java

161

- Colors enhance the appearance of a program and help to convey meaning.
- Note every color is created from a red, blue and green component (RGB), each an *integer* between 0 and 255 or a *float* value between 0.0 and 1.0.
- Color constants: `ORANGE`, `PINK`, `CYAN`, `MAGENTA`, `YELLOW`, `BLACK`, `WHITE`, `GRAY`, `LIGHTGRAY`, `DARKGRAY`, `RED`, `GREEN`, `BLUE` (RGB value of 0, 0, 255)
- `Color` Methods: several constructors; `int getRed()`, `int getGreen()`, `int getBlue()`; `getColor` and `setColor()`; `brighter()`, `darker()`;

SwingDemo.java

162



Event-Driven Programming

- An "event" is an object representing a user's interaction with a GUI component (e.g., clicking on a component or moving the mouse over it).
- A "listener" is an object that is notified when an event occurs, and executes code to respond to that event. To organize event "listening," you use different event listener classes. For example,
  - An ActionEvent is generated from clicking a JButton, selecting a menu item in a JMenu, hitting the Enter key (or return) in a JTextField. To such components use addActionListener (instance of class that implements the ActionListener interface), which requires defining *void actionPerformed (ActionEvent e)*

Events and Listeners

- Mouse movement, mouse clicks & keystrokes cause different kinds of "events" to be generated. A Java program must install *event listener* objects in order to be notified about only certain of these events.
  - To listen to window events, add a "window listener" object to the frame. The **WindowListener** interface has 7 methods, or you can just implement certain of these methods via the **WindowAdapter** "convenience class."
  - A "mouse listener" must implement all of the **MouseListener** interface (5 methods). Or it can implement just certain methods via the **MouseAdapter** class.
  - When the user clicks a button, presses Return while typing in a text field, or chooses a menu item, an **ActionEvent** is generated. A listener must implement the **ActionListener** interface (which contains one method).

Event - Listener Examples

Event Type	Listener Type	Some Methods
Window Events (e.g., JFrame jf)	jf.addWindowListener (*object of class that implements Windowlistener or extends WindowAdapter*);	windowClosing windowIconified windowOpened *4 others*
Action Events (e.g., JButton jb)	jb.addActionListener (*object of class that implements ActionListener*);	actionPerformed
Mouse Events (e.g., Component c)	c.addMouseListener (*object of class that implements MouseListener or extends MouseAdapter*);	mouseEntered mouseClicked mouseExited mousePressed mouseReleased

Window Events

166

- A WindowEvent is generated when you
  - open a window for the first time,
  - close it,
  - iconify it,
  - activate it, etc.

- To such components, use addWindowListener(instance of class that implements the WindowListener interface or extends the WindowAdapter class) .
- To listen to window events, add a "window listener" object to the frame. The **WindowListener** interface has 7 methods; or one may implement certain of these methods via the **WindowAdapter** "convenience class."
  - See ActionEventDemo.java for an example

Implementing WindowListener Interface

167

- The first step in preparing an application to respond to Window events is to define a "listener" class that implements the WindowListener interface:

```
import java.awt.event.*;
public class WindowChatterBox  implements WindowListener
{
   public void windowClosing (WindowEvent e )
    {  System.out.println ("Window got CLOSED!");  System.exit(0);  }
   public void windowActivated ( WindowEvent e )          { }
   public void windowClosed ( WindowEvent e )             { }
   public void windowDeactivated ( WindowEvent e )        { }
   public void windowDeiconified ( WindowEvent e )        { }
   public void windowIconified ( WindowEvent e )
      {  System.out.println ("Window got iconified!");  }
   public void windowOpened (WindowEvent e)               { }
}
```

167

Some java.awt.event Classes

168

Components	Events	Description
Button, JButton	ActionEvent	User clicked button
CheckBox, JCheckBox	ItemEvent	User toggled a checkbox
ScrollBar, JScrollBar	AdjustmentEvent	User moved the scrollbar
Component, JComponent	ComponentEvent	Component was moved or resized
	FocusEvent	Component acquired or lost focus
	KeyEvent	User typed a key
TextField, JTextField	ActionEvent	User typed Enter key
Window, JWindow	WindowEvent	User manipulated window

168

Some javax.swing.event Classes

169

Components	Events	Description
JPopupMenu	PopupMenuEvent	User selected a choice
JComponent	AncestorEvent	An event occurred in an ancestor
JList	ListSelectionEvent	User double-clicked a list item
	ListDataEvent	List's contents were changed
JMenu	MenuEvent	User selected menu item
JTextComponent	CaretEvent	User clicked in text
JTable	TableModelEvent	Items added/removed from table
	TableColumnModelEvent	A table column was moved

169

[Anonymous] Inner Classes

170

- An inner class is one that's defined inside of another class. The syntax is ugly, but it provides a useful way of creating classes and objects "on the fly." The body of the class definition is put right after the **new** operator, as illustrated:

```
…
JFrame jf = new JFrame ("test frame");
jf.setSize(400, 300);
jf.setVisible(true);
jf.addWindowListener   (
   new WindowAdapter()
    {
       public void windowClosing (WindowEvent e)
       {  System.exit(0);   }
    }                  );
…
```

170

Listening to Mouse Events

171

- An event is a change in status that can initiate a responsive action. For example, when you click a mouse button, you change the status of the mouse, thus launching a mouse event.
- Mouse events can be trapped for any GUI widget that derives from Component. Each of the following takes a MouseEvent object as its argument (which has the x,y coordinates of where the event occurred).
  - The MouseListener methods are mousePressed, mouseClicked, mouseReleased, mouseEntered and mouseExited.
  - The MouseMotionListener methods are mouseDragged and mouseMoved.

171

The Graphics Object

172

- To draw shapes and lines, you need access to a Graphics object.
- This object is like a paintbrush; it can be dipped into any color, e.g., if g is the Graphics object:
  g.setColor (Color.RED);
- The Graphics object is usually obtained inside the paintComponent method of a JComponent or JPanel

```
g.fillRect (10, 30, 60, 35);
g.fillOval (80, 40, 50, 70);
```

How to Draw in a Window

173

- With Swing, the preferred method is to create a dedicated drawing area as a subclass of `JPanel`:

```
public class subclassName extends JPanel
{   …
    public void paintComponent (Graphics g)
     { // NEVER invoke paintComponent directly
        super.paintComponent( g );   // SOMETIMES needed
       // your drawing code goes here
          …
     }
}
```

- If you draw on a Canvas, override the `paint()` method, not `paintComponent`.

173

Drawing Methods of a Graphics Object

174

- void **drawLine** (int x1, int y1, int x2, int y2)
- void **drawRect** (int x, int y, int width, int height)
- void **fillRect** (int x, int y, int width, int height)
- void **fillRoundRect** (int x, int y, int width, int height)
- void **clearRect** (int x, int y, int width, int height)
- void **drawOval** (int x, int y, int width, int height)
- void **fillOval** (int x, int y, int width, int height)
- void **drawArc** (int x, int y, int width, int height, int startA, int arcA)
- void **drawString** (String s, int x, int y)
- void **setFont** (Font f)
- void **setColor** (Color c)

174

Fonts (see Surprise.java)

175

- To write a string, add a **drawString** statement to a paint or paintComponent method (defined in a subclass of **JPanel** or **Canvas**):
  - *graphicsContext*.**drawString (aString, x, y);**
- To determine the height of a font or width of a string, create an instance of the **FontMetrics** class and examine it's instance variables:
  - **FontMetrics** *var* = *graphicsContext*.**getFontMetrics();**
  - *var*.**getHeight ()**
  - *var*.**stringWidth (***aString***)**
- *graphicsContext*.setFont (
  new Font ("*fontName*", Font.*style*, *size*));

176



Testing a Theorem with Graphics

177

- We can write methods that operate on points and lines in such a way that the underlying algebra remains hidden. E.g., to test the theorem: "The medians of a triangle meet at a point"
- Suppose we have 4 arrays:
  - `Point [] vertices = new Point [3];`
  - `Point [] midpoints = new Point [3];`
  - `Line [] medians = new Line [3];`
  - `Point [] intersections = new Point [3];`



JSlider Controls

178

- JSliders enable the user to select from a range of integer values. When oriented horizontally, the minimum value is at the extreme left; vertical ones have the minimum value at the bottom.
  - The slider can show both major and minor tick marks between them. The # of values between the tick marks is controlled with setMajorTickSpacing(int n) & setMinorTickSpacing(int n) . Use setPaintTicks(true) .
- `JSlider (int orientation, int min, int max, int value)`
- `int getValue ()` *// Returns slider's current value.*
- `void setValue (int n)` *// Sets slider's current value.*

178

JSlider Controls, continued

179

- JSliders generate ChangeEvents when the user interacts with the control. An object of a class that implements interface ChangeListener and defines method public void stateChanged (ChangeEvent e) can respond to ChangeEvents.

```
final JSlider s = new JSlider (
  SwingConstants.HORIZONTAL, 5, 200, 50);
s.addChangeListener ( new ChangeListener()
    { public void stateChanged(ChangeEvent e)
        {   System.out.println(s.getValue() ); }
    }
                    );      //see SliderTest.java
```

179

Scroll Panes

180

- A JScrollPane is an object that manages scrolling within a window or JTextArea.

```
this.getContentPane().add(new JScrollPane(display));
```



This parameter refers to the scrollable component.

The parts of a scroll pane.

180

Scroll Pane Example



JTextAreas (see TextIO.java)

- A **`JTextArea`** is a multiline text area that can be used for either input or output. It is almost identical to the AWT component, TextArea, except it does not contain scrollbars by default.
  - `public JTextArea (String text, int rows, int columns)`
  - If jf is a defined JFrame, and jta is a defined JTextArea, then **`jf.add (new JScrollPane(jta));`** will add scrollbars to the text area.
- Additional useful methods:
  - `void append (String str)`
  - `int getRows()`
  - `int getColumns()`
  - `int getColumnWidth()`

Simulation



The Queue Class

PostOffice Simulation



Animation (see Animate.java)

- The abstract class Image is the superclass of all classes that represent graphical images. You cannot create an object of class Image; instead you request that an Image be loaded (from a local file or a URL) via the ImageIcon class:

```
static ImageIcon [] mouse;
mouse = new ImageIcon [15];
for (int i=0; i < 15; i++)
  mouse[i] = new ImageIcon ( "T" + (i+1)  + ".gif");
```

- You can also use method `paintIcon(Component c, Graphics g, int x, int y)` to draw the individual images on screen. See ScratchAnimation.java

Menus (see MenuTest.java)

187

- Menus allow a program to grow without cluttering the interface.
- A **JMenuBar** is a horizontal list of menus.
- A **JMenu** is a clickable area on the menu bar that is associated with a **JPopupMenu**, a small window that displays **JMenuItem**s.
- **JSeparator**s are used to group menu items.

Menu Example (MenuTest.java)

188

```
JMenuBar bar = new JMenuBar();                  // create menubar
setJMenuBar( bar );                             // set the menubar for the JFrame
                                                // create File menu and Exit menu item
JMenu fileMenu = new JMenu ( "File" );
fileMenu.setMnemonic ( 'F' );
JMenuItem aboutItem = new JMenuItem( "About..." );
aboutItem.setMnemonic ( 'A' );
aboutItem.addActionListener (
  new ActionListener() {
    public void actionPerformed( ActionEvent e )
    {  JOptionPane.showMessageDialog( MenuTest.this,
       "This is an example\nof using menus",
       "About", JOptionPane.PLAIN_MESSAGE );
    }
                    }   );
fileMenu.add (aboutItem );
JMenuItem exitItem = new JMenuItem( "Exit" );
```

Keyboard equivalent for specific menus and menu items

Action listeners are associated with menu items.

JMenuItem is added to a JMenu

Chessboard GUI

189



Eyes that Follow the Mouse

190

- EyeDemo.java creates a JPanel that contains an instance variable (cursor) that contains the location of the mouse, and two instance variables of type PairOfEyes (e1 and e2).



Eyes that Follow the Mouse

191

- Consider the distance **d** from point e to point c is $\sqrt{(c.x - e.x)^2 + (c.y - e.y)^2}$
- We also know r is the radius of the eye. Using similar triangles, we see the center of the pupil is at (p.x, p.y), where
  p.x= e.x+ (c.x-e.x) * r/d and
  p.y = e.y+ (c.y-e.y) * r/d
  (Actually, r is EYE_RADIUS-PUPIL_RADIUS .)



See PairOfEyes.java and EyeDemo.java

191

Sets

192

- In mathematics, an unordered collection of distinct elements is called a set.
  - Elements can be added, located and removed.
  - Sets don't have duplicates.
  - The order of the objects doesn't really matter (unlike an array or an ArrayList)
- In Java, sets are implemented both as "hash tables" and as "trees." Both HashSet and TreeSet implement the Set interface.

Defining a Set Class in Java

193

- By defining set objects as a class we can use sets as an *abstract data type* in our Java programs, without having to worry about implementation details! See file TestSets.java The main program loop looks roughly like this:
  - case 1: … setA.readSet(); … break;
  - case 2: … setB.readSet(); … break;
  - case 3: … System.out.print (setA.intersect (setB)); …
  - case 4: … System.out.print (setA.union (setB)); …
  - case 5: … System.out.print (setA.difference (setB)); …
  - default: …
- Additional methods for adding and removing a single element from a set (among others) are also defined (in addition to *union*, *difference*, and *intersect*).

193

Representing a Set Using a Single Byte

194

- A byte (8 bits) is the smallest addressable unit of main memory. The same byte (bit pattern) can represent various things:

1	0	1	1	0	1	1	1
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

  - an unsigned integer
  - a signed integer
  - an ASCII character
  - the SET whose elements are …

194

Basic Set Operations

195

- Set A = {2, 3, 4, 7} A = 1 0 0 1 1 1 0 0

  Set B = {2, 4, 5} B = 0 0 1 1 0 1 0 0

  0 0 0 1 0 1 0 0
- The UNION of the two sets **(A + B)** consists of all elements that are in *either* A or B. In Java, the operation on bytes can be expressed as …
- The INTERSECTION of the two sets **(A * B)** consists of all elements that are in *both* A and B. In Java, this operation can be expressed as …

195

Set Operations, *continued*

196

- The SYMMETRIC DIFFERENCE of two sets consists of elements that are in one set but not the other. In our example, this is {3, 5, 7}. This is expressed as …
- The COMPLEMENT of a set consists of all elements not in the set. The complement of b can be expressed in Java as …
- The DIFFERENCE of two sets (A - B) consists of elements that are in A but not in B. In Java this operation is …
- In Java there are bitwise operations that incorporate assignment: `|=`, `&=`, `^=`
- These are useful for setting, clearing, or complementing a single bit within a byte. For example,
  - `a |= (1 << n)` `//sets what???`

196

Instance Data Members

197

- To have sets of a fixed size (say, those whose elements range from 0 to 255, requiring 256/8 = 32 bytes):

```
class Bitset
{  private byte byteArray[] = new byte[32];
   …
}
```

- But to have sets of different sizes, use

```
class Bitset
{   private int maxSize;
    private byte [] byteArray;
    …
```

197

Representing Larger SETs

198

- Use nbyte consecutive bytes of memory to represent a set with 8*nbyte possible members.
  - For example, with 4 bytes we can have members from 0 to 31:

  - To find the byte in which element n is represented, the address we need is **byteArray [???]** and the bit number at that address is … **???**
- Now we can implement primitives like

```
void setBit (int n)   // turn on appropriate bit for value n
{   int whichByte =  n / 8;
    int whichBit =   n % 8;
    byteArray[whichByte]  |=  (1 << whichBit);
}
```

198

More Primitive Operations

199

- See file Bitset.java for the details:

```
boolean getBit ( int n )          // Returns the n'th bit value
{    int whichByte = n / 8;
     int whichBit = n % 8;
     return ( (byteArray [whichByte] &  (1 << whichBit) )  != 0 );
}

void clearBit ( int n )           // CLEARS the n'th bit
{    int whichByte = n / 8;
     int whichBit = n % 8;
     byteArray [whichByte]  &=  ( (1 << whichBit) ^ 255);
}
```


Constructors for a SET

200

- The constructor that we use most of the time calculates how many bytes of memory are needed and allocates them using **new**. The set is cleared automatically to all zeroes, yielding the empty set:
  - ```
    Bitset (int size)
    {  maxSize = size;
       int nbyte = (size + 7) / 8;
       byteArray = new byte[nbyte];
    }
    Bitset ()
    {  maxSize = 0;
       byteArray = null;
    }
    ```

In case we declare an array of sets, we define a 0-arg constructor and an initialization function (named `setSize`). See Bitset.java for details.

How Intersection/Union Works

201


    ```

Defining the Operators

202

- The union, intersection and difference operations act on 2 sets to produce a third set. What's tricky is making sure these methods work correctly when they operate on sets of different sizes. For example,

```
Bitset union (Bitset setB)
{  Bitset temp = new Bitset (this.maxSize  > setB.maxSize ? this :  setB);

    int nbyte = Math.min (byteArray.length, setB.byteArray.length);
    for (int i = 0;  i < nbyte; i++)
    {
        temp.byteArray[i] = (byte)  (byteArray[i] | setB.byteArray[i]);
    }
    return temp;
}
```


Other Useful Set Functions

203

- Frequently we need to operate on a single element of a set: to add it to the set, remove it from the set, or test whether it is present in the set. All these things are easy to do in terms of primitive **Bitset** operations:

```
boolean member  (int i)
{
    if (i >= maxSize)  return false;
    else return (getBit (i) );
}
```

```
void  include (int i)
{  if  (i >= maxSize) error("Too big!");  setBit(i);  }
```

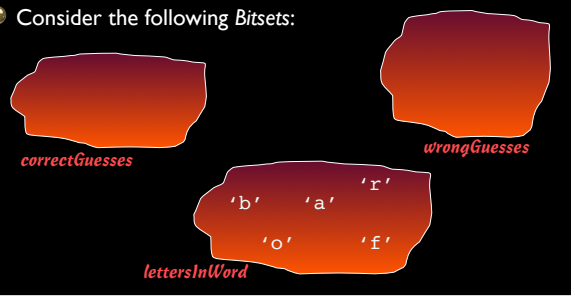
```
void exclude (int i)
{  if (i  >= maxSize) error ("Too big!");  clearBit(i);  }
```

- All the above check parameter i's validity.

Hangman: A Bitset Application

204

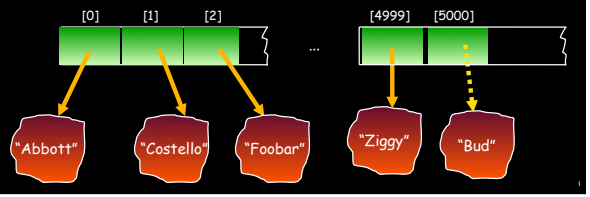
- Consider the following *Bitsets*:



Limitations of Arrays, revisited

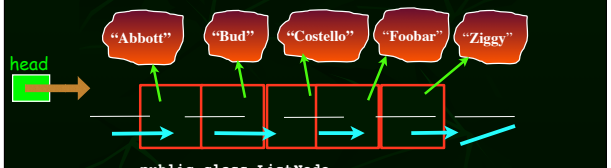
1

- Consider "Employee" objects, sorted alphabetically by last name.
  - How do we add / delete objects from the array?



Linked-List as an Alternative

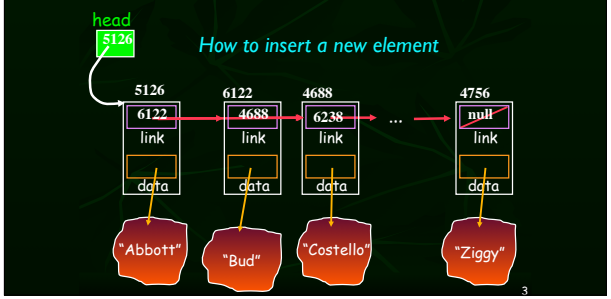
2




```
public class ListNode
{
     String  data;    // the green arrows
     ListNode link;   // the blue arrows
}
```

Linked-Lists, continued

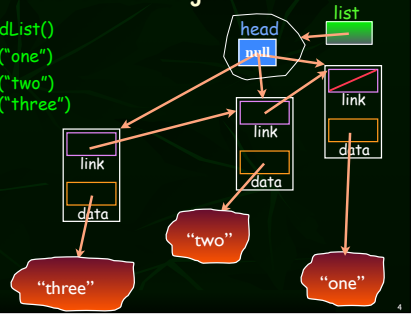
3



LinkedListDemo.java

4

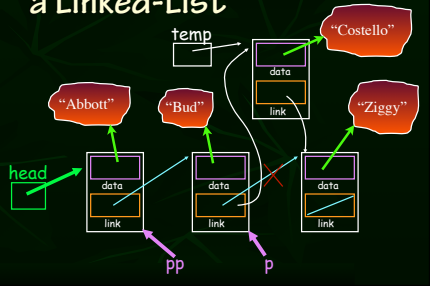
list = new StringLinkedList()
list.addNodeToStart ("one")
list.addNodeToStart ("two")
list.addNodeToStart ("three")



Inserting in the Middle of a Linked-List

5

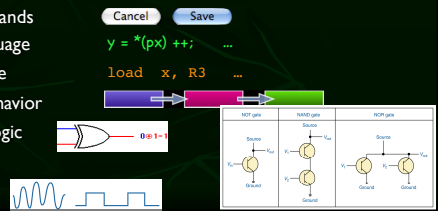
pp.setLink(temp);
temp.setLink(p);



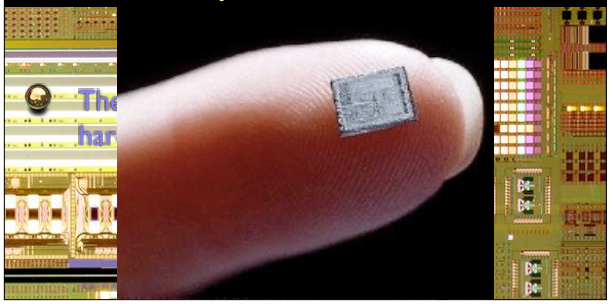
Managing Complexity

6

- ABSTRACTION: *Simplifying assumptions that ignore or gloss over details that could be articulated in a more detailed analysis.*
- Examples from computing
  - user-level commands
  - higher-level language
  - machine language
  - register-level behavior
  - clocked digital logic
  - semiconductors
  - analog voltages

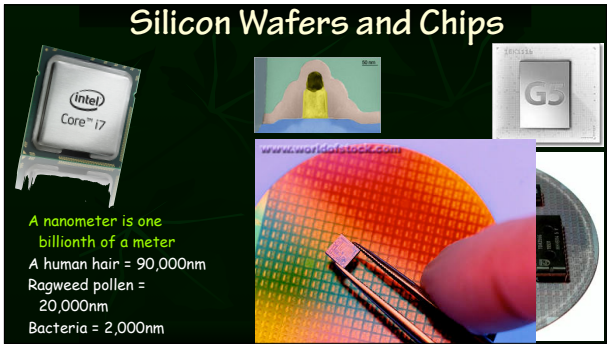


Computer Hardware



7

Silicon Wafers and Chips



A nanometer is one billionth of a meter

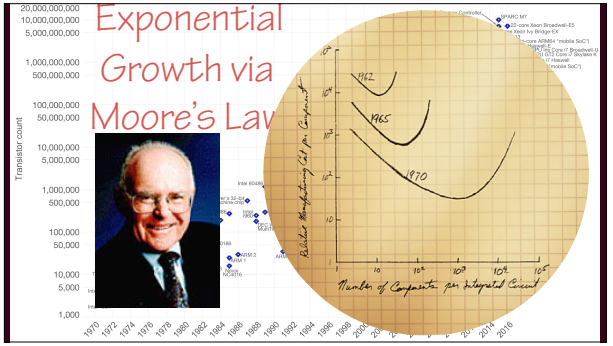
A human hair = 90,000nm

Ragweed pollen = 20,000nm

Bacteria = 2,000nm

8

Exponential Growth via Moore's Law



9

History of the Transistor

10

Machine Architecture

11

- The MIPS Processor Family
  - A RISC architecture embodied by the R2000, R3000, R4000 and R6000 processors.
  - MIPS Technologies = principal architect of embedded 32- and 64-bit RISC processors, licensed to system OEMs, semiconductor manufacturers, etc.
  - See Britton: "Mips Assembly Language Programming" and **http://www.cs.wisc.edu/~larus/spim.html**
- Processor Performance: *Time per Task* = C * T * I
  - C = Cycles per instruction
  - T = Time per cycle (clock speed)
  - I = Instructions per Task

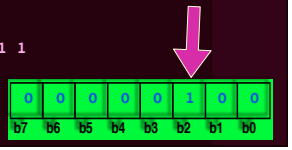
Number Systems

12

- **Unsigned Binary Numbers**

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 0 1 1
0 0 0 0 0 1 0 0
        …
        …
1 1 1 1 1 1 1 1
```

- **Bit Numbering**



Binary Arithmetic

13

- The usual algorithm works, with 1 + 1 = 0 (carry 1):

```
          1 1
    0 0 0 0 0 1 0 1
+   0 0 0 0 1 1 1 0
    0 0 0 1 0 0 1 1
```

- Carry out of the MSB means the sum is too large to represent.
- Conversion of binary to decimal involves adding up the decimal values of the powers of 2 corresponding to the 1 bits. For example,
  - 0 1 0 1 1 0 1 0

Signed Integers

14

- How many different integers can be represented using N bits?
- To "negate" an integer: form the two's-complement by
  - Taking the logical complement
  - Adding one to the logical complement
  - For example,

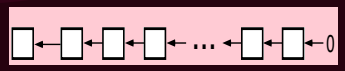
```
  0 0 0 0 0 1 1 0
  1 1 1 1 1 0 0 1
+                 1
  1 1 1 1 1 0 1 0
```

- What about negating zero?
- What is 1 1 1 1 1 1 1 1 ?
- What is 1 0 0 0 0 0 0 0 ?

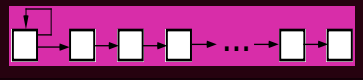
Doubling / Halving By Shifting

15

- *Doubling* is accomplished via "arithmetic left shift"
  - Slide all bits one place to the LEFT, and make the new b_0 equal to 0.



- *Halving* is accomplished via "arithmetic right shift"
  - Slide all bits one place to the RIGHT, and make the new b_{n-1} stay the same.



Hexadecimal (base 16)

16

- The 16 patterns of 4 bits are as follows:

Binary	0000	0001	0010	0011	0100	0101	0110	0111
Decimal	0	1	2	3	4	5	6	7
Hex	0	1	2	3	4	5	6	7

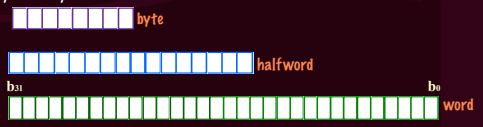
Binary	1000	1001	1010	1011	1100	1101	1110	1111
Decimal	8	9	10	11	12	13	14	15
Hex	8	9	A	B	C	D	E	F

- So an 8-bit -1 would be FF. To distinguish such numbers from identifiers, we precede them by "0x" in C/C++ and in Java.
  - `0x69 =`

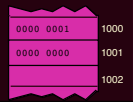
Data Sizes

17

- The smallest unit of data that can be stored in / retrieved from memory is 1 byte = 8 bits



- Memory addresses are byte addresses; ask for the byte at location 1001, and you'll get it.
  - *Word* addresses must be divisible by 4
  - *Halfword* addresses must be divisible by 2



The "Endian" Wars

18

- Big Endian: Most significant byte is in lower-numbered address
  - e.g., M680x0, IBM mainframes, PowerPC, SPARC

- Little Endian: Least significant byte is in lower-numbered address
  - e.g., Intel 80x86 and Core Duo, DEC Alpha & VAX

- Total incompatibility between the two! BUT …
  - MIPS is bi-endian!

The MIPS "RISC" Architecture

19

- Few, simple instructions
- Load/store architecture (move in/out of registers, rather than to/from memory)
- Simple addressing
- Why RISC?
  - Optimize speed of functions that get used most
  - Small processor design (rapid design cycle)
  - Rely on smart compilers to produce good code!
- MIPS Registers (all are 32-bit)
  - general purpose: r0 - r31 (but r0 always contains 0)
  - these may be given other names, depending on usage conventions. (e.g., r29 ↔ $sp)

MIPS Registers, continued

20

- $t0 - $t9 (temporaries. Use them at will, but subroutines may clobber them)
- $s0 - $s7 (saved registers. Well-behaved subroutines will not permanently change them.)
- $a0 - $a3 (argument registers. Up to 4 subroutine arguments go here.)
- $v0 - $v1 (Subroutines return values here.)
- $sp (stack pointer)
- $ra (return address of subroutine)
- +5 other numbered general purpose registers. In addition, we have
  - special registers **LO** / **HI** for fixed-point multiplication/division
  - a special *floating-point* registers, f0 - f31
  - a special program counter register named **PC**

MIPS Registers: the Full Story

21

Register name	Register #	Usage
$zero	0	constant 0
$at	1	reserved for assembler
$v0	2	expr eval & function result
$v1	3	expr eval & function result
$a0	4	argument 1 to a function
$a1	5	argument 2 to a function
$a2	6	argument 3 to a function
$a3	7	argument 4 to a function
$t0	8	temporary (not preserved)

MIPS Registers, continued

22

Register name	Register #	Usage
$t1	9	temporary (not preserved)
$t2	10	temporary (not preserved)
$t3	11	temporary (not preserved)
$t4	12	temporary (not preserved)
$t5	13	temporary (not preserved)
$t6	14	temporary (not preserved)
$t7	15	temporary (not preserved)
$s0	16	saved temporary (preserved)
$s1	17	saved temporary (preserved)
$s2	18	saved temporary (preserved)

MIPS Registers, continued

23

Register name	Register #	Usage
$s3	19	saved temporary (preserved)
$s4	20	saved temporary (preserved)
$s5	21	saved temporary (preserved)
$s6	22	saved temporary (preserved)
$s7	23	saved temporary (preserved)
$t8	24	temporary (not preserved)
$t9	25	temporary (not preserved)
$k0	26	reserved for OS kernel
$k1	27	reserved for OS kernel
$gp	28	pointer to global area

MIPS Registers, continued

24

Register name	Register #	Usage
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (to caller)

Note: MIPS also has 32 *floating-point* registers. Two registers are paired for double precision numbers. Odd numbered registers cannot be used for arithmetic or branching, just as part of a double precision register pair. There are also special registers for a "program counter" (PC) and for multiplication/division (LO and HI)

Two Instructions

25

- **ADD**
  - e.g., **add $t2, $t0, $t1** # *(t2) <— (t0) + (t1)*
    - first 2 operands must be registers, but third can be "immediate" data — stored in the instruction
  - e.g., **add $t2, $t0, 64** # *(t2) <—(t0) + 64*
    - there is an actual **ADDI** instruction, used by the assembler when the 3rd arg to **ADD** is a 16-bit constant.
- LI (load immediate)
  - e.g., **li $t0, 176** # *(t0) <— 176*
  - stores an integer into a register
  - this is an example of a "*pseduo*-instruction". The assembler translates this into real instructions.

Assembly Language Programming

26

- **Adding 1 + 2**

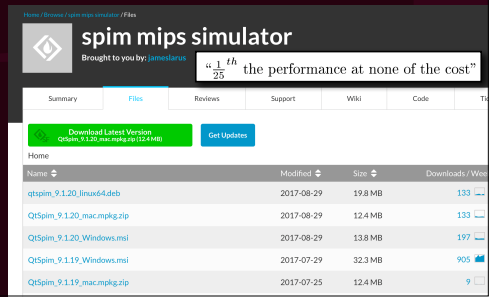
```
main:                   # SPIM starts execution at main.
  li    $t1, 1          # load 1 into $t1.
  li    $t2, 2          # load 2 into $t2.
  add   $t0, $t1, $t2   # $t0 <- $t1 + $t2.
  li    $v0, 10         # syscall code 10 is for exit
  syscall               # make the syscall
```

- **SYSCALLs**
  - are calls to support routines that do I/O, exit gracefully, etc. Code for function to be performed is passed in $v0, e.g.
    - **code 5 = read integer into $v0**
    - **code 1 = print the integer in $a0**

https://sourceforge.net/projects/spimsimulator/

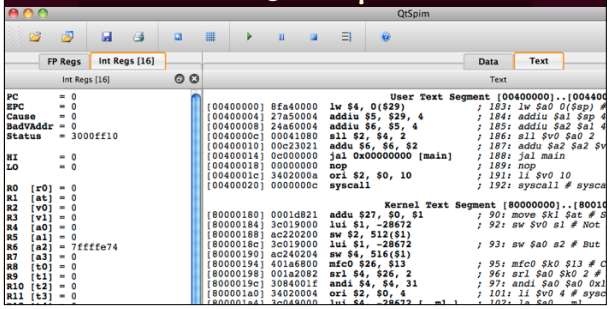
27

Download SPIM



Using QtSpim

28



29

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $a0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

Add2.asm (sum of 2 integers)

30

```
main:  #     $t0 and $v0    - used to hold the 2 numbers.
       ## Get first number from user, put into $t0.
       li      $v0, 5         # load syscall read_int into $v0.
       syscall                # make the syscall.
       move  $t0, $v0         # move the number read
       ## Get second number from user, leave it in $v0.
       li      $v0, 5         # load syscall read_int into $v0.
       syscall                # make the syscall.
       add     $a0, $t0, $v0  # compute the sum.
       li      $v0, 1         # load syscall print_int into $v0.
       syscall                # make the syscall.
       li      $v0, 10        # syscall code 10 is for exit.
       syscall                # make the syscall.
```

More Instructions

31

- To move data between registers and memory:
  - **LW / LH / LB** — load word / halfword / byte
  - **SW / SH / SB** — store word / halfword/ byte
- LA — load address (e.g., `LA $t0, labelOfData`)
  - puts 32-bit address of label in $t0. In practice, labelOfData assembles to something like 4097($at), so actual address cannot be determined till runtime
- Arithmetic / Logical Instructions
  - ADD and SUB
  - MUL *$rd, $rs1, $rs2* vs. MULT *$r1, $r2*
  - DIV *$r1, $r2*
  - AND, OR, XOR, NOT
  - SLL, SRL, SRA

Examples of *Directives*

32

- Assembler directives all begin with "."
- A C-style string, null-terminated, is produced by the directive
  - `hello_msg: .asciiz "Hello\n"`
- **.byte 0xFF** *#produces one byte containing 1111 1111*
- **.ascii "ABC"** *# produces 3 bytes with ASCII codes*
- **.text** *# assemble what follows into "program space,"*
- **.data** *# vs. assemble what follows into "data space"*
- Printing an *asciiz* string (see file hello.asm)

```
.text
la   $a0, hello_msg
li   $v0, 4
syscall
```

Branch Instructions

33

- `b lab` *# unconditional branch to instruction labelled lab*
- `beq, bne, blt, bgt, ble, bge`
  - e.g., bgez $r, lab #branch if $r ≥0
  - Note: lab is actually converted to a 16-bit offset by assembler — so branches are limited to "distances" of about $\pm 2^{15}$ words (added to PC)
- Unsigned Comparisons
  - **e.g., compare** `1111 … 1111 to 0000 … 0000`
  - `bgeu, bgtu, bleu, bltu`
- Miscellaneous Data Movement
  - `move` $rd, $rs *# like LW, but for registers only*
  - `mfhi` $rd *# also MTHI, MFLO, MTLO*

Decision-Making

34

- **IF ($t0 < $t1) THEN *do-first* else *do-second***

```
    …
    bge     $t0, $t1, to_ge
        # Execution arrives here only if ($t0) < ($t1)
        # … code for do-first …
    b       end_if_code
to_ge:          # Execution arrives here only if ($t0) ≥ ($t1)
                # … code for do-second …
                # 2 forks merge back together
end_if_code:
```

- Example: Use a "moving pointer" to find the length of an ASCIIZ string stored at **the_string**
- Example: Check if a string is a "palindrome"

Converting a String to a Number

35

- Assume $t0 points to a string of one or more decimal digits:

- Algorithm:
  - $t2 <— 0
  - $t1 <- ($t0); $t0 ++
  - $t1 < '0' OR > '9' —> Done!
  - $t2 <- $t2 * 10 + ($t1 - '0')
- See program **atoi-1.asm**. This program does NOT handle negative numbers; produces 0 result if there are no digits; does not check overflow; does not check for illegal input characters!

Addressing Memory

36

- MIPS is a "load-store" architecture.
  - Bare machine allows only one addressing mode, **constant ($register)**
- However, SPIM (the virtual machine) allows all of the following, along with "pseudo-instructions," such as **abs**
  - **($register)**
  - **a constant**
  - **a symbolic label**
  - **a symbolic label ± a constant**
  - **a symbolic label ± a constant ($register)**

Detecting Overflow

37

- Multiplication of two 32-bit integers produces a 64-bit product in the special registers **HI** and **LO**
- The **MUL** "instruction" expands into more than one instruction that can produce the right result if it can be represented in 32 bits.
- Use the underlying **MULT** instruction to check for a possible overflow condition:

```
li    $t4, 10        #no immediate MULT
mult  $t2, $t4
mfhi  $t5
bnez  $t5, overflow
mflo  $t2
bltz  $t2, overflow  #product is in $t2
```

Subroutine Linkage

38

- The environment of a function includes
  - **values of the function (parameters/arguments)**
  - **values of local variables**
  - **a record of where to return to when this activation of the function completes**
- The environment is kept on the stack in a block called the *stack frame* (one per activation)
- Example:

```
f1 ( … )
   {  …  f2(…)  …  }

f2 ( … )
   {  …  f1( … )  …  }
```

Euclid's Algorithm as Subroutine (see Euclid.asm)

39

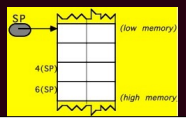
```
gcd:   div    $a0, $a1   # divide $a0 by $a1
       mfhi   $t0        # the remainder is in HI - get it to check
       beqz   $t0, done  # if the remainder is 0, you're done!  Go to done…
       move   $a0, $a1   # if the remainder is not 0, put $a1 into $a0
       move   $a1, $t0   # put the remainder into $a1
       b      gcd        # go back to step 1 (div)

done:  move   $v0, $a1   # we have the answer - put it in $v0 to return
       jr     $ra        # jump back to main
```

Pointers & Offset/Displacement Addressing for LOADs & STOREs

40

- lw $rt, 16-bit-offset($rb)
  - sign-extend offset to 32 bits; then add it to
  - the value in $rb to give address of word to be moved
  - Note: $rb is typically $sp (stack pointer), or $at (a global pointer, reserved for use by the assembler)
- The *Stack* grows from larger addresses to smaller ones

Global space is pre-allocated as a result of calculations made at assembly time.



Register Conventions

41

- $t0 - $t9 are CALLER-saved; if the *caller* wants them preserved, it must do the saving (in its stack frame) since the *callee* is under no obligation to preserve them.
- $s0 - $s7 are CALLEE-saved; every subroutine is obliged to preserve them. So a routine that wants to use them MUST save them on entry, and restore them on exit.
- Caller-Callee Protocol:
  - *CALLER* preparation for calling
    - put parameters into $a0 - $a3
    - if *caller* needs any of $t0 - $t9 saved, put them in caller stack frame
    - execute a jal CALLEE, which puts the address after the instruction itself into $ra

Caller-Callee Protocol,

42

- *CALLEE* preamble
  - Create stack frame
    - $sp <— $sp - <frame size>
  - Save in the frame
    - $fp
    - any of $s0 - $s7 used by callee (known at compiling/coding time)
    - $ra, unless this routine does not call any other (leaf routine)
  - $fp <— $sp + <frame size> *# $fp is typically old $sp*
- BODY of CALLEE
  - local variables are referred to as displacements off $sp — i.e., positions within the "frame"

Caller-Callee Protocol, *continued*

43

- *CALLEE* return preparation
  - $v0 <— <return value>
  - restore callee-saved registers ($fp, $s0-$s7, $ra)
  - restore stack pointer: $sp <— $sp + <frame size>
  - return using JR $ra
- *CALLER* cleanup
  - restore caller-saved registers ($t0-$t9), if any
  - return value is in $v0
- Understanding RECURSION: see fib-s.asm
- Note: A function that requires additional temporary storage of size that is not known till runtime (e.g., a sequence of input characters), can use the stack.

Recursive Fibonacci

44

```
      move    $s0, $a0
      blt     $s0, 2, fib_base_case   # if n < 2 ...
      sub     $a0, $s0, 1             # compute fib (n - 1)
      jal     fib
r1:   move    $s1, $v0                # s1 <- fib (n - 1).
      sub     $a0, $s0, 2             # compute fib (n - 2)
      jal     fib
r2:   move    $s2, $v0                # $s2 <- fib (n - 2).
      add     $v0, $s1, $s2           # fib (n -1) + fib (n -2)
      b       fib_return
fib_base_case:  li      $v0, 1
fib_return:     ...                   # deal with stack frame
                jr      $ra
```

44

Arithmetic and Logical Instructions

45

Absolute value

```
abs rdest, rsrc          pseudoinstruction
```

Put the absolute value of register `rsrc` in register `rdest`.

Addition (with overflow)

```
add rd, rs, rt
```

0	rs	rt	rd	0	0x20
6	5	5	5	5	6

Addition (without overflow)

```
addu rd, rs, rt
```

0	rs	rt	rd	0	0x21
6	5	5	5	5	6

Put the sum of registers `rs` and `rt` into register `rd`.

Addition immediate (with overflow)

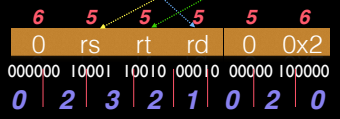
```
addi rt, rs, imm
```

8	rs	rt	imm
6	5	5	16

Assembly to Machine Language

46

```
add $v0,$s1,$s2
add $2,$17,$18
```



47

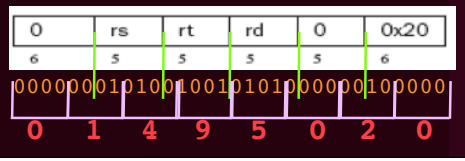
`add $t2, $t2, $t1`

is assembled into

`add $10, $10, $9`

and is of the form

`add rd, rs, rt`

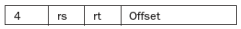


Other Instruction Formats

48

Branch on equal

`beq rs, rt, label`



Jump Instructions

Jump

`j target`

Unconditionally jump to the instruction at target.

Jump and link

`jal target`

RISC vs CISC

49

- As VLSI improvements made it possible to squeeze more components onto a processor chip, more complex instruction sets became possible.
  - e.g., The VAX's POLY X, D, ADDR instruction (the description of this takes 4 pages in the manual)
- Such complex instructions are not implemented by custom hardware, but by programs in a special language called microcode that controls movement and operations on data in the internal processor registers.
  - Microcode ("firmware") is prepared "at the factory" and frozen into the processor chip.

50

From the VAX Machine Architecture Manual

POLY

POLYNOMINAL EVALUATION

Purpose: allows fast calculation of math functions

Format: opcode arg.rx, degree.rw, tbladdr.ab

Operation:

```
tmp1 ← degree;
if tmp1 GRTU 31 then RESERVED OPERAND EXCEPTION;
tmp2 ← tbladdr;
tmp3 ← [(tmp2);          !tmp3 accumulates the
                         partial result
                         !tmp3 is of type X
if POLYH then –(SP) ← arg;
tmp4 ← 0;                !underflow flag for
                         original 11/780
while tmp1 GRTU 0 do
   begin                 !computation loop
   tmp5 ← [arg * tmp3];  !tmp5 accumulates new partial
                         result;
```

50

51

The VAX **POLY** Instruction, continued

Integer and Floating Point Instructions

```
   if FU EQL 1 then tmp4 ← 1;
                    !set underflow flag
                    (original 11/780)
   end;
tmp1 ← tmp1 - 1;
tmp2 ← tmp2 + [size of data type];
tmp3 ← tmp5;
                    !update partial result in tmp3
end;
if POLYF then
   begin
   R0 ← tmp3;
   R1 ← 0;
   R2 ← 0;
   R3 ← tmp2;
   end;
if POLYD or POLYG then
   mbegin
   R1'R0 ← tmp3;
   R2 ← 0;
   R3 ← tmp2;
```

Integer and Floating Point Instructions

Description: The table address operand points to a table of p coefficients; the coefficient of the highest order te polynomial is pointed to by the table address ope table is specified with lower order coefficients sto creasing addresses. The data type of the coefficie same as the data type of the argument operand.

Evaluation is carried out by Horner's method, and th of R0 (R1'R0 for POLYD and POLYG, R3'R2'R1'R0 fo are replaced by the result. The result computed is:

result = C[0] + x*(C[1] + x*(C[2] +...x*C [d])), w degree and x = arg.

The unsigned word degree operand specifies th numbered coefficient to participate in the evaluatio requires four longwords on the stack to store arg i instruction is interrupted.

Notes:

1. After execution:

   POLYF
   R0 = result
   R1 = 0

Problems with CISC

52

- Complex design
- Complex instructions are very rarely used
  - in part, because compilers don't want them
- Full instruction set must be maintained in new CPUs
- The microcode describes use of multiple registers, arithmetic units, etc. But most of the time, these functional units are idle!
- Cycles vs. Instructions
  - Each physical processor has a "cycle time" = the time for the smallest indivisible register transfer, arithmetic (etc.) operation. 2.5 gHz = 2.5 billion cycles/sec.
  - An instruction typically takes several cycles to execute completely. CISC = many cycles per instruction

Making a Fast RISC Machine

53

- Fast clock rate (time/cycle)
- Low cycles per instruction (cycle/inst)
  - Streamlining the the hardware can improve the CPI by factors of 5 or more
- Relatively low instruction counts (inst/task), optimizing compiler technology
- "Pipelining" to improve effective cycles/instr.
  - Goal: To keep as much of the processor busy at the same time as possible
  - First: Keep instruction types simple, so there are only a few different "ways" instructions work
    - e.g., except for MUL/DIV, all MIPS arithmetic/logical instructions work on 3 registers, or 2 registers plus immediate data

Pipelining

54

- Second: Execute different "parts" of several instructions simultaneously
- Hypothetical, simple, non-pipelined CPU:
  - `IF ID EX MEM WB`
    `IF ID EX MEM WB`
  - Each instruction takes 5 "clock ticks"
  - IF = instruction fetch from cache
  - ID = instruction decode
  - EX = execute (arithmetic, etc.)
  - MEM = memory operation (load/store)
  - WB = write back (changes to register, if any)
- With RISC, you can get 5 (or more) instructions in progress at once (R4000). Cycles/inst = 1.2 to 1.6

Pipelining, continued

55

Single-Issue, 5-Stage RISC Pipeline

Instruction	Pipeline Stages								
1	IF	ID	EX	MEM	WB				
2		IF	ID	EX	MEM	WB			
3			IF	ID	EX	MEM	WB		
4				IF	ID	EX	MEM	WB	
5					IF	ID	EX	MEM	WB

5 (or more) instructions in progress at once (R4000)

Typical of early & current RISCs; some used 4 stages

Attempt to issue (start) 1 instruction per cycle
Current & new systems: start 2 or more, sometimes!

Cycles/instruction = 1 (at best), 1.2-1.6 more typical

Instruction Scheduling

56

- What if the result from one instruction is used by the next instruction, but the first instruction isn't done?
  - Hardware stall / interlock
  - Insert "NO-OP"s in between to slow things down;
- Example: The instruction after a LOAD cannot use the result of that LOAD; similarly, the instruction after a branch is executed even if the branch is taken. For example,

```
lw   $R7,  X
?????                  #delay slot
add  $R7, $R7, 1
```

Reorder the Instructions!

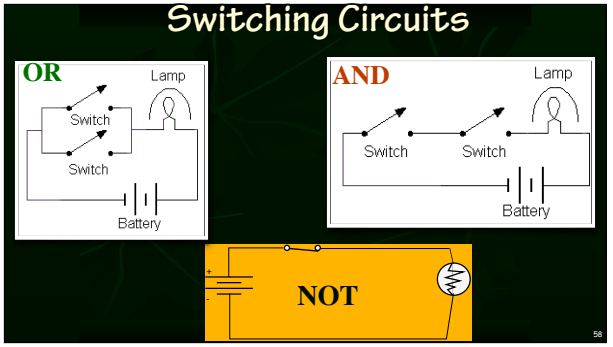
57

- a = b+ 1; if (c == 0) d = 0;

```
lw   $r2, b
nop
add  $r2, $r2, 1
sw    $r2, a                # a = b + 1
lw    $r3, c                # r3 = c
nop
bne    $r3,  zero, lab      # if (c != 0) skip
nop
sw     $zero, d             # if (c == 0), then d = 0
lab:
```

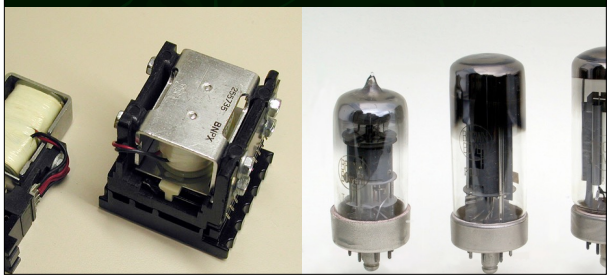
Switching Circuits

58



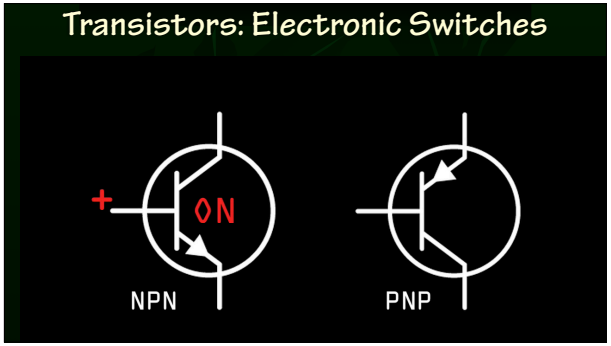
Electromechanical Relays and Vacuum Tubes

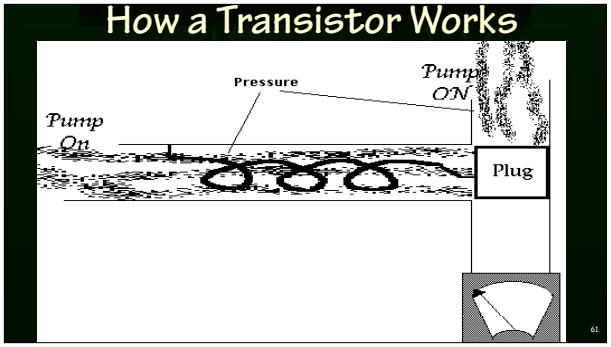
59



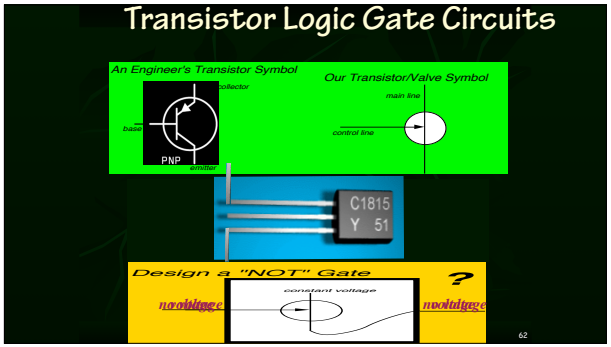
Transistors: Electronic Switches

60

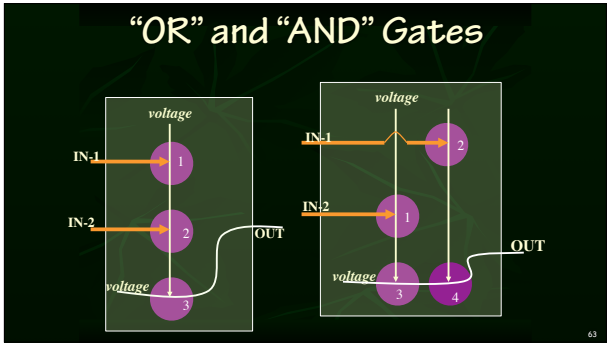



How a Transistor Works
Pressure
Pump ON
Pump On
Plug
61

61


Transistor Logic Gate Circuits
An Engineer's Transistor Symbol
collector
base
PNP
emitter
Our Transistor/Valve Symbol
main line
control line
C1815
Y 51
Design a "NOT" Gate
?
constant voltage
62

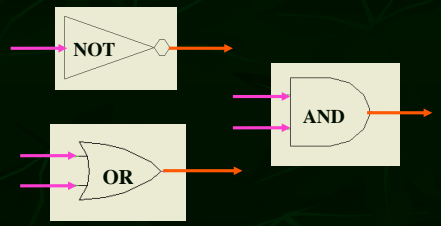
62


"OR" and "AND" Gates
voltage
IN-1
IN-2
1
2
3
OUT
voltage
voltage
IN-1
IN-2
1
2
3
4
OUT
63

63

Representing the Gates

64

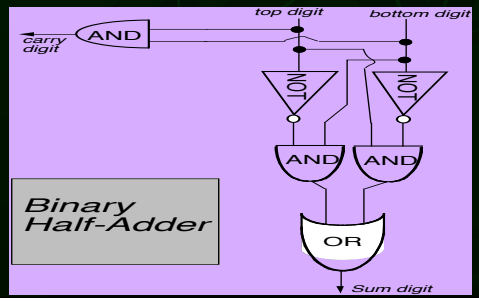


See http://math.hws.edu/TMCM/java/labs/xLogicCircuitsLab1.html

64

Binary Half-Adder

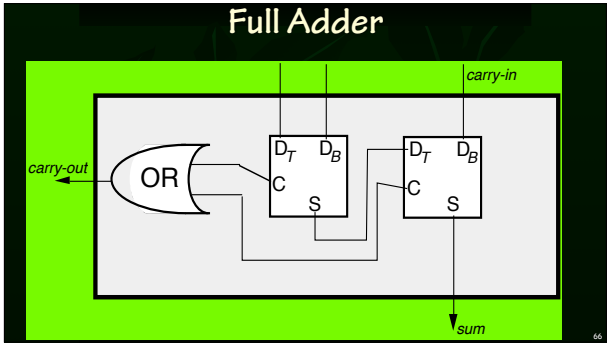
65



65

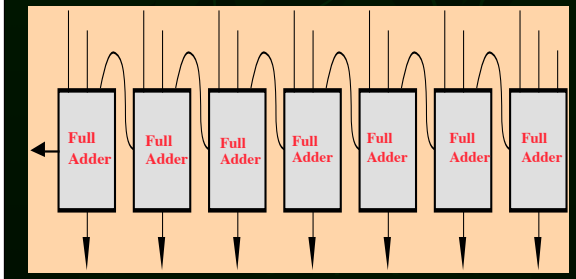
Full Adder

66



66

Circuit to Add 7-bit Numbers



67

67

Grammars

- A **language** is a (possibly infinite) set of legal character strings, specified by …
- … a **grammar**. It contains
  - A finite set of terminal symbols, T
  - A finite set of non-terminal symbols, N
  - A finite set of "rules" for specifying a language, each of the form
    Non-terminal-symbol ⇒ sequence of symbols chosen from the union of T + N

68

68

Grammar for English Subset

- R1: <Sentence> ⇒ <NounPhrase> <VerbPhrase> .
  | Who <VerbPhrase> ?
- R2: <NounPhrase> ⇒ <Article> <ModifiedNoun>
  | <ProperNoun>
- R3: <Article> ⇒ a | the
- R4: <ModifiedNoun> ⇒ <Noun> |
  <Adjective> <ModifiedNoun>
- R5: <Noun> ⇒ ball | book | baby
- R6: <Adjective> ⇒ big | blue | boring
- R7: <ProperNoun> ⇒ Jill | Obama | Jack
- R8: <VerbPhrase> ⇒ <TransitiveVerb> <NounPhrase>
- R9: <TransitiveVerb> ⇒ kicked | kissed | saw

69

69

Our Objective

- To interpret correctly a sequence of Java-like assignment statements such as

```
a = 21 + 5 * 4;
b = (a + 6) * (a - 3);
temp = (b - a) * -b;
ans = temp * (1 + temp);
.
```

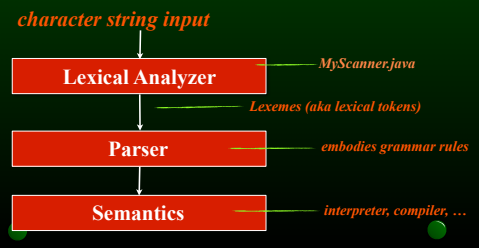
… and print the value of each variable assigned to; or compile into MIPS assembly language; or translate into LISP; or …

A Grammar For the Language

- <expr> ::= <term> { [+, -] <term> }*
- <term> ::= <factor> { [*, /] <factor>}*
- <factor> ::= <variable> | <number>
  | (<expr>) | - <factor>
- <asst> ::= <variable> = <expr> ;
- <program> ::= { <asst> } * .
- Note that <variable>, <number> are specified at the lexical level
- The meta-symbol ⇒ is used interchangeably with ::=
- Consider the grammatical structure of this string:

```
abc = 4 * 5 + 6 ;
```

Overview of the Compiler/Interpreter



Lexical and Parser Level

73

- **The Lexical Analyzer**
  - Breaks the input character string into units that are significant to the parser, i.e., are distinguishable in the grammar.
  - These units (lexemes or lexical tokens) are
    - more than individual characters (321, temp)
    - less than "expressions" such as 2+3
- **The Parser**
  - Analyzes the stream of lexemes according to the rules of the grammar.
  - Determines whether the input is legal and may develop a parse tree analysis.

File Parser.java

74

- **"Recursive-descent" parser**
  - A simple kind suitable only for certain grammars
  - There is a function for each non-terminal
  - Grammar has the property that if a choice must be made among alternative right-hand sides, they can be distinguished by looking at the next lexeme. For example,

<expr> ::= <term> { [+, -] <term> }* *becomes*

```
expr()
{   term();
    while (next lexeme is + or -) term();
      …
}
```

- *// Bookkeeping detail: fetches one lexeme too many*
  // so Scanner.java has an "unget" method

Interpreter/Compiler Semantics

75

- **Interpreter:**
  - The "meaning" of a term, factor, or expression is a number value.
  - The semantics of a rule like:
    - <expr> ::= <term> + <term>
      4 * 6 + 7 * 8
      β α γ
  - The characters from the input that correspond to the <term> and <term> on the right have numerical values β, γ
  - The numerical sum of β + γ is the value α associated with the <expr> as a whole.
- **Compiler:**
  - The "meaning" of an expression is the assembly language code which would, when executed, calculate the value of the expression.

The Lexical Analyzer (MyScanner.java)

76

- The smallest unit handled by the parser is a ***Lexeme***. These are returned by the scanner.
- Lexeme types:
  - **SYMBOL** *Variables*
  - **NUMBER** *Numeric atoms*
  - **OP** *Operators*
  - **ASSIGN** *Assignment operator*
  - **EOP** *End-of-program*
  - **LEFT_PAREN, RIGHT_PAREN**
  - **SEMICOLON**
  - **EOF** *End-of-File*
- Technically, a **Lexeme** has 3 instance variables: a type, a name, and an nval.

Semantic Routines

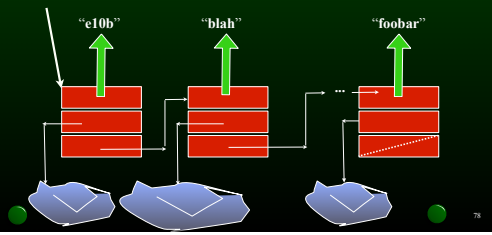
77

- Parser routines return AValues
- But first, what are AValues?
  - In IAValue: Just a single integer value
  - In CAValue: a regType, and a regNum (i.e., a register into which the generated code would compute the value at runtime)
    - 2 kinds of registers: temporaries (8 t's), and permanents (8 s's)
    - In this simple compiler, we just throw up our hands if we run out of registers
    - Symbols are associated with permanent registers.
    - Static arrays keep track of used registers .
  - In LispAValue, RPNAValue, EngAValue, there is a pName (print name) field. These 3 are quite similar.

The Symbol Table (Symtab.java)

78

- For every new identifier that is recognized by the parser, a *SymEntry* object is added on to a *table*:



Compiler Semantics (CAValue.java)

- There are two boolean arrays recording usage of registers at this point in compilation.
- temporaryRegUsage 
- permanentRegUsage 
- By declaring them static members of the CAValue class, we ensure there is just one copy of these arrays, shared by all CAValues.
  - See routines getTempReg(), getPermReg(), freeRegIfTempReg()
- Other Compiler Semantic Routines
  - genArithOpCode (String opcode, CAValue src1, CAValue src2)