Java Programming

Binnur Kurt
binnur.kurt@ieee.org

Istanbul Technical University
Computer Engineering Department

Version 0.0.3

About the Lecturer

- **BSc**

  İTÜ, Computer Engineering Department, 1995
- **MSc**

  İTÜ, Computer Engineering Department, 1997
- Areas of Interest
  - Digital Image and Video Analysis and Processing
  - Real-Time Computer Vision Systems
  - Multimedia: Indexing and Retrieval
  - Software Engineering
  - OO Analysis and Design

Welcome to the Course

- Introduction
  - Name
  - Company affiliation
  - Title, function, and job responsibility
  - Programming experience
  - Reasons for enrolling in this course
  - Expectations for this course
- Course Hours
  - 08:30─14:30
  - No lunch break

Course Overview

1. Java Technology
2. Object-Oriented Programming
   Encapsulation, Class, Method, Attribute,
   Accessing Object Members, Constructor
3. Identifiers, Keywords, and Types
   Java Keywords, Primitive Types, Variables,
   Declarations, Assignment, Reference Type
   Constructing and initializing Objects,
   Assigning Reference Types, Pass-by-Value
4. Expressions and Flow Control
   Variable and Scope, Initializing Variables, Operators,
   Logical Operators, Branching Statement,
   Looping Statement, Special Loop Flow Control

5. Arrays
   Declaring and Creating Arrays, Initialization of Arrays
   Multidimensional Arrays, Resizing and Copying Arrays
6. Class Design
   Inheritance, Access Control, Method Overriding, Super
   Polymorphism, Virtual Method Invocation, instanceof
   Casting Objects, Overloading Constructors,
   Object and Class Classes, Wrapper Classes
7. Advanced Class Features
8. When things go Wrong: Exceptions
9. Text-Based Applications
10. Building Java GUIs
11. GUI Event Handling
12. GUI-Based Applications
13. Threads
14. Advanced I/O Streams
15. Networking

1 Java Technology

Objectives

Java Technology 1

- Describe the key features of Java Technology
- Define the terms class and application
- Write, compile, and run a simple Java Technology application
- Describe the Java Virtual Machine's function
- Define Garbage collection
- List the three tasks performed by the Java platform that handle code security

What is Java Technology?

Java Technology 1

- A programming language
- A development environment
- An application environment
- A deployment environment

Primary Goals of the Java Technology

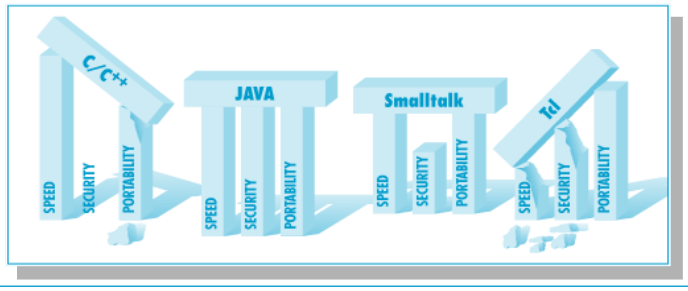
Java Technology 1

- Provides an easy-to-use language by
  - Avoiding the pitfalls of other languages
  - Being object-oriented
  - Enabling users to create streamlined and clear code
- Provides an interpreted environment for
  - Improved speed of development
  - Code portability

Primary Goals of the Java Technology

Java Technology 1

- Enables users to run more than one thread of activity,
- Load classes dynamically; that is, at the time they are actually needed,
- Supports dynamically changing programs during runtime by loading classes from disparate sources,
- Furnishes better security



Primary Goals of the Java Technology

Java Technology 1

- The following features fulfill these goals:
  - JVM,
  - Garbage Collection,
  - Code security

The Java Virtual Machine

Java Technology 1

- Provides hardware platform specifications,
- Reads compiled byte codes that are platform independent,
- Is implemented as software or hardware,
- Is implemented in a Java technology development tool or a Web browser.

The Java Virtual Machine

Java Technology 1

- JVM provides definitions for the
  - Instruction set (central processing unit [CPU])
  - Register set
  - Class file format
  - Stack
  - Garbage-collected heap
  - Memory area

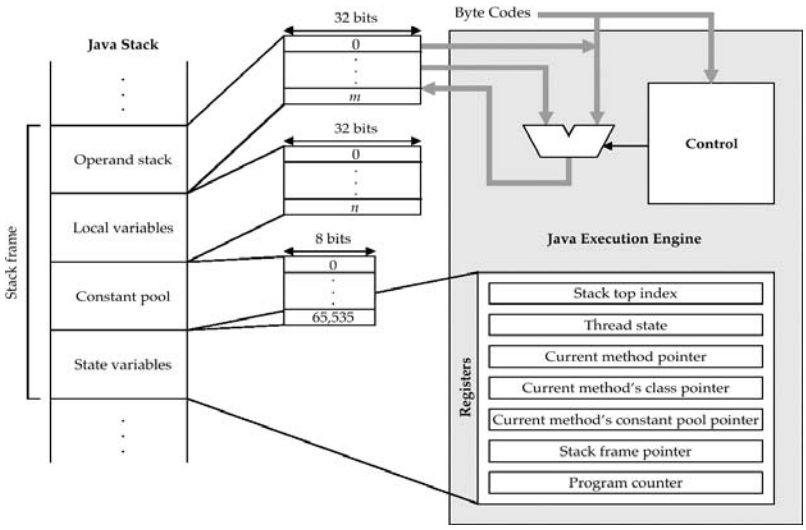
The Java Virtual Machine

Java Technology 1

- The Java Virtual Machine
  - **Bytecodes** that maintain proper type discipline form the code.
  - The majority of type checking is done when the code is compiled.
  - Every implementation of the **JVM** approved by Sun Microsystems must be able to run any compliant class file.

Java Technology 1

Java Virtual Machine Architecture



Java Technology 1

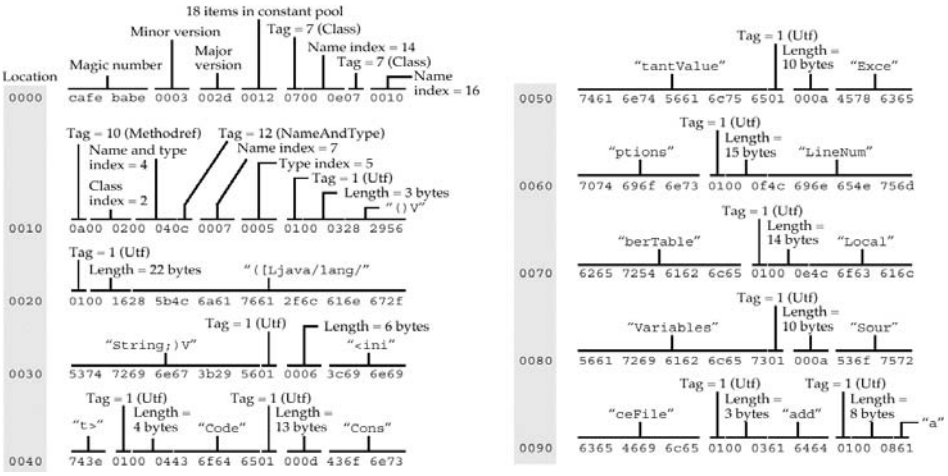
Java Program and Compiled Class File

```
// This is file add.java

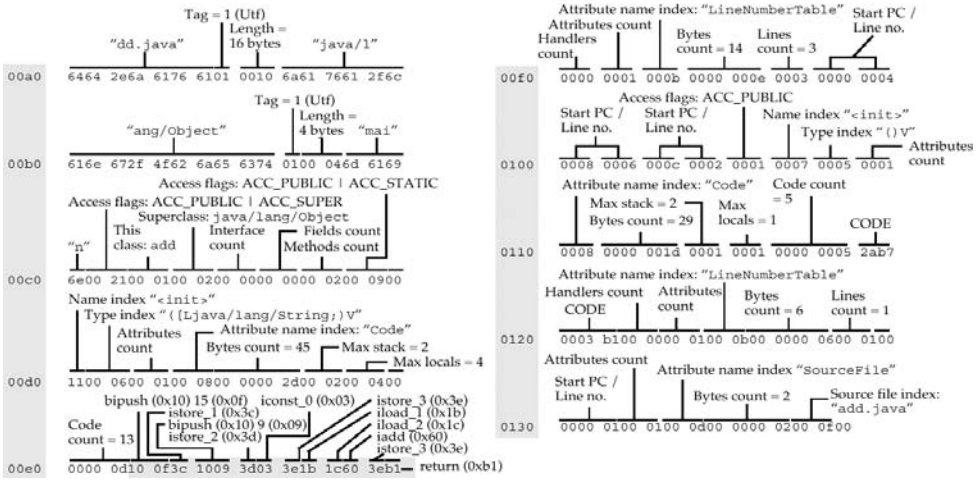
public class add {
    public static void main(String args[]) {
        int x=15, y=9, z=0;
        z = x + y;
        }
    }
```

```
0000 cafe babe 0003 002d 0012 0700 0e07 0010   .............
0010 0a00 0200 040c 0007 0005 0100 0328 2956   ............()V
0020 0100 1628 5b4c 6a61 7661 2f6c 616e 672f   ...([Ljava/lang/
0030 5374 7269 6e67 3b29 5601 0006 3c69 6e69   String;)V...<ini
0040 743e 0100 0443 6f64 6501 000d 436f 6e73   t>...Code...Cons
0050 7461 6e74 5661 6c75 6501 000a 4578 6365   tantValue...Exce
0060 7074 696f 6e73 0100 0f4c 696e 654e 756d   ptions...LineNum
0070 6265 7254 6162 6c65 0100 0e4c 6f63 616c   berTable...Local
0080 5661 7269 6162 6c65 7301 000a 536f 7572   Variables...Sour
0090 6365 4669 6c65 0100 0361 6464 0100 0861   ceFile...add...a
00a0 6464 2e6a 6176 6101 0010 6a61 7661 2f6c   dd.java...java/l
00b0 616e 672f 4f62 6a65 6374 0100 046d 6169   ang/Object...mai
00c0 6e00 2100 0100 0200 0000 0000 0200 0900   n.!.............
00d0 1100 0600 0100 0800 0000 2d00 0200 0400   ..........-.....
00e0 0000 0d10 0f3c 1009 3d03 3e1b 1c60 3eb1   .....<..=.>..`>.
00f0 0000 0001 000b 0000 000e 0003 0000 0004   ................
0100 0008 0006 000c 0002 0001 0007 0005 0001   ................
0110 0008 0000 001d 0001 0001 0000 0005 2ab7   ..............*.
0120 0003 b100 0000 0100 0b00 0000 0600 0100   ................
0130 0000 0100 0100 0d00 0000 0200 0f00        ..............
```

A Java Class File



A Java Class File (Cont'd)



A Java Class File (Cont'd)

```
ClassFile {
        u4 magic;
        u2 minor_version;
        u2 major_version;
        u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count-1];
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
        field_info fields[fields_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
    }
```

Java Technology 1

Byte Code for Java Program

Location	Code	Mnemonic	Meaning
0x00e3	0x10	bipush	Push next byte onto stack
0x00e4	0x0f	15	Argument to bipush
0x00e5	0x3c	istore_1	Pop stack to local variable 1
0x00e6	0x10	bipush	Push next byte onto stack
0x00e7	0x09	9	Argument to bipush
0x00e8	0x3d	istore_2	Pop stack to local variable 2
0x00e9	0x03	iconst_0	Push 0 onto stack
0x00ea	0x3e	istore_3	Pop stack to local variable 3
0x00eb	0x1b	iload_1	Push local variable 1 onto stack
0x00ec	0x1c	iload_2	Push local variable 2 onto stack
0x00ed	0x60	iadd	Add top two stack elements
0x00ee	0x3e	istore_3	Pop stack to local variable 3
0x00ef	0xb1	return	Return

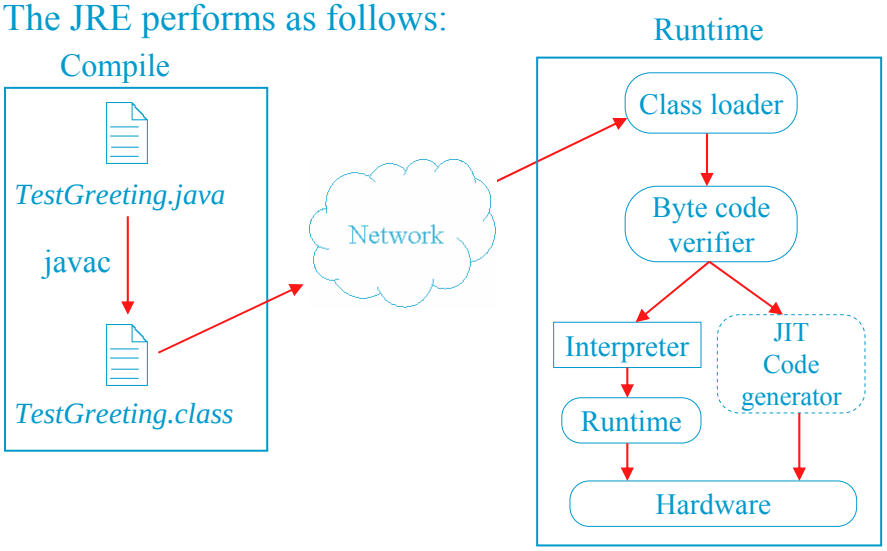
Java Technology 1

Garbage Collection

- Allocated memory that is no longer needed should be deallocated
- In other languages, deallocation is the programmer's responsibility
- The Java programming language provides a system-level thread to track memory allocation
- Garbage collection
  - Checks for and frees memory no longer needed
  - Is done automatically
  - Can vary dramatically across JVM implementations

Java Runtime Environment

The JRE performs as follows:



The Java Runtime Environment

Java Technology 1

- Performs three main tasks:
  - Loads code – Performed by the class loader
  - Verifies code – Performed by the bytecode verifier
  - Executes code – Performed by the runtime interpreter

The Class Loader

Java Technology 1

- Loads all classes necessary for the execution of a program
- Maintains classes of the local file system in separate "namespaces"
- Prevents spoofing

The Bytecode Verifier

Java Technology 1

- Ensures that
  - The code adheres to the JVM specification
  - The code does not violate system integrity
  - The code causes no operand stack overflows or underflows
  - The parameter types for all operational code are correct
  - No illegal data conversions (the conversion of integers to pointers) have occurred

A Basic Java Application: TestGreeting.java

Java Technology 1

```
1 //
2 // Sample "Hello World" application
3 //
4 public class TestGreeting{
5     public static void main (String[] args) {
6             Greeting hello = new Greeting();
7             hello.greet();
8     }
9 }
```

Greeting.java

```
1 // The Greeting class declaration.
2 public class Greeting {
3     public void greet() {
4         System.out.println("hi");
5     }
6 }
```

Java Technology 1

Running and Compiling

- Compiling TestGreeting.java
  - javac TestGreeting.java
- Greeting.java is compiled automatically
- Running an application
  - java TestGreeting
- Locating common compile and runtime errors

Java Technology 1

Compile-Time Errors

- `javac: Command not found`
- `Greeting.java:4: cannot resolve symbol`
  `symbol : method printl (java.lang.String)`
  `location: class java.io.PrintStream`
  `System.out.printl("hi");`
- `TestGreet.java:4: Public class TestGreeting must be defined in a file called "TestGreeting.java".`

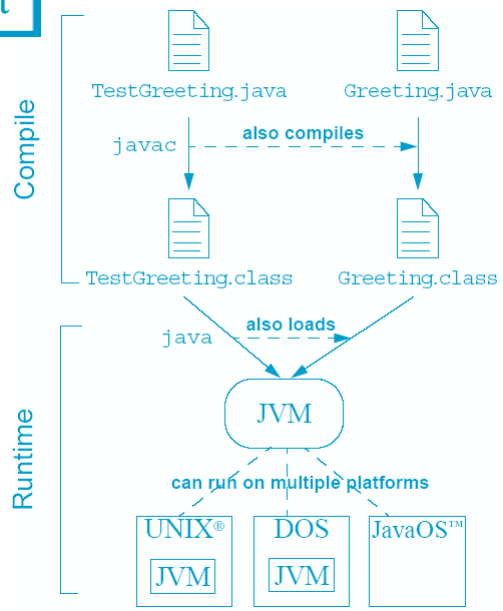
Java Technology 1

Run-Time Errors

- `Can't find class TestGreeting`
- `Exception in thread "main"`
  `java.lang.NoSuchMethodError: main`

Java Technology 1

Java Runtime Environment



1# JAVA TECHNOLOGY

1. Compile Test1.java
2. Compile Test2.java
3. Compile Test3.java
4. Compile Test4.java

Sun microsystems
HANDS-ON LAB

2 Object Oriented Programming

Objectives

- Define modeling concepts: abstraction, encapsulation, and packages
- Discuss why you can reuse Java technology application code
- Define class, member, attribute, method, constructor, and package
- Use the access modifiers private, and public as appropriate for the guidelines of the encapsulation
- Invoke a method on a particular object

Objectives

- In a Java program, identify the following:
  - The package statement
  - The import statements
  - Classes, methods, and attributes
  - Constructors
- Use the Java technology application programming interface (API) online documentation

Software Engineering

Toolkits / Frameworks / Object APIs (1990s – up)				
Java 2 SDK	AWT / Swing	Jini	Java Beans	JDBC

Object-Oriented Languages (1980s – up)					
SELF	Smalltalk	Common Lisp Object System	Effiel	C++	Java

Libraries / Functional APIs (1960s – early 1980s)				
NASTRAN	TCP/IP	ISAM	X-Windows	OpenLook

High-Level Languages (1950s –up)				Operating Systems (1960s – up)			
Fortran	LISP	C	COBOL	OS/360	UNIX	MacOS	MS-Windows

Machine Code (late 1940s – up)

The Analysis and Design Phase

- Analysis describes what the system needs to do:
  - Modeling the real-world: actors and activities, objects, and behaviors
- Design describes how the system does it:
  - Modeling the relationships and interactions between objects and actors in the system
  - Finding useful abstractions to help simplify the problem or solution

Abstraction

- Functions – Write an algorithm once to be used in many situations
  - Objects – Group a related set of attributes and behaviors into a class
  - Frameworks and APIs – Large groups of objects that support a complex activity:
  - Frameworks can be used “as is” or be modified to extend the basic behavior

Classes as Blueprints for Objects

- In manufacturing, a blueprint describes a device from which many physical devices are constructed
- In software, a class is a description of an object:
  - A class describes the data that each object includes
  - A class describes the behaviors that each object exhibits
- In Java technology, classes support three key features of object-oriented programming (OOP):
  - Encapsulation
  - Inheritance
  - Polymorphism

Declaring Java Technology Classes

- Basic syntax of a Java class:

```
< modifiers> class < class_name> {
   [< attribute_declarations>]
   [< constructor_declarations>]
   [< method_declarations>]
}
```

- Example:

```
public class Vehicle {
   private double maxLoad;
   public void setMaxLoad(double value{
        maxLoad = value;
   }
}
```

Declaring Attributes

- Basic syntax of an attribute:

  < modifiers> <type> <name>;

- Examples:

```
public class Foo {
   private int x;
   private float y = 10000.0F;
   private String name = "Bates Motel";
}
```


Declaring Methods

- Basic syntax of a method:

```
<modifiers> <return_type> <name>
   ([< argument_list>]) {
        [< statements>]
   }
```

- Examples:

```
public class Dog {
   private int weight;
   public int getWeight() {
        return weight;
   }
   public void setWeight(int newWeight) {
        weight = newWeight;
   }
}
```

Accessing Object Members

- The "dot" notation:

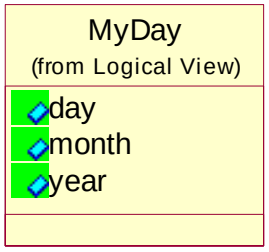
```
<object>.<member>
```

- This is used to access object members including attributes and methods
- Examples:

```
d.setWeight(42);
d.weight = 42; // only permissible
               //if weight is public
```


Information Hiding

- The Problem:

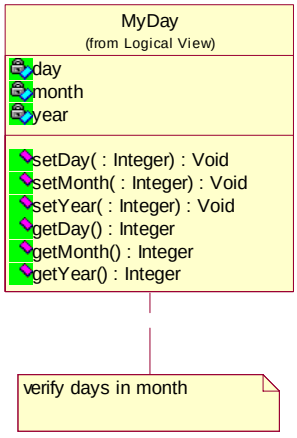


Client code has direct access to internal data:

```
MyDate d = new MyDate();
d.day = 32;
// invalid day
d.month = 2; d.day = 30;
// plausible but wrong
d.day = d.day + 1;
// no check for wrap around
```

Information Hiding

- The Solution:

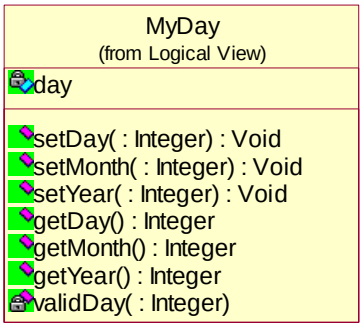


Client code must use setters/getters to access internal data:

```
MyDate d = new MyDate();
d.setDay(32);
// invalid day, returns false
d.setMonth(2);
d.setDay(30);
// plausible but wrong
d.setDay(d.getDay() + 1);
```


Encapsulation

- Hides the implementation details of a class
  - Forces the user to use an interface to access data
  - Makes the code more maintainable



Declaring Constructors


```
1 public class Dog {
2   private int weight;
3
4   public Dog() {
5         weight = 42;
6   }
7
8   public int getWeight() {
9         return weight;
10  }
11  public void setWeight(int newWeight) {
12    weight = newWeight;
13  }
14}
```


The Default Constructor

- There is always at least one constructor in every class.
- If the writer does not supply any constructors, the default constructor is present automatically:
  - The default constructor takes no arguments
  - The default constructor has no body
- Enables you to create object instances with new Xxx()without having to write a constructor.

Source File Layout

- Basic syntax of a Java source file:

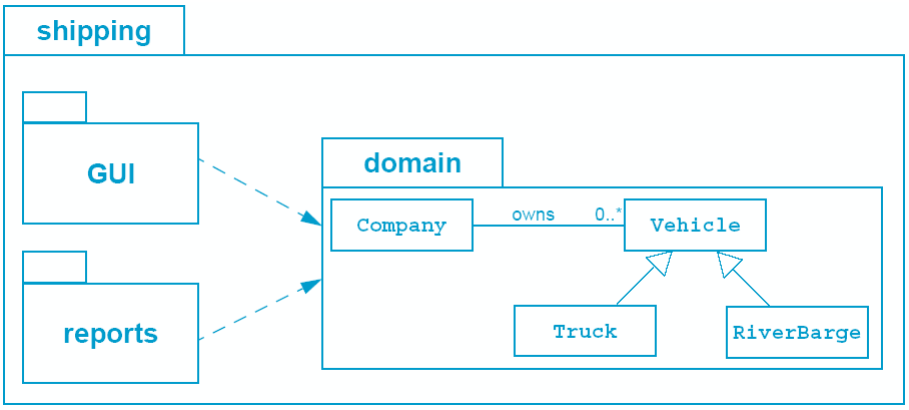
```
[< package_declaration>]
[< import_declarations>]
< class_declaration>+
```

- Example, the VehicleCapacityReport.java file:

```
package shipping.reports;
import shipping.domain.*;
import java.util.List;
import java.io.*;
public class VehicleCapacityReport {
    private List vehicles;
    public void generateReport(Writer output)
    {...}
}
```


Software Packages

- Packages help manage large software systems.
- Packages can contain classes and sub-packages.



The package Statement

- Basic syntax of the package statement:

```
package < top_pkg_name>[.< sub_pkg_name>]*;
```

- Example:

```
package shipping.reports;
```

- Specify the package declaration at the beginning of the source file.
- Only one package declaration per source file.
- If no package is declared, then the class "belongs" to the default package.
- Package names must be hierarchical and separated by dots.

The `import` Statement

- Basic syntax of the import statement:

```
import
<pkg_name>[.<sub_pkg_name>].<class_name>;
```

- OR

```
import <pkg_name>[.< sub_pkg_name>].*;
```

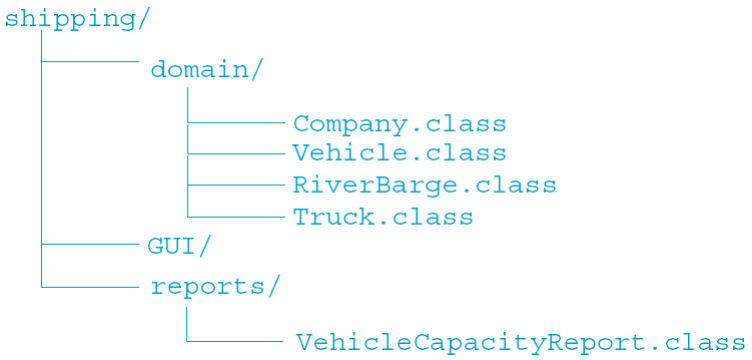
- Examples:

```
import shipping.domain.*;
import java.util.List;
import java.io.*;
```

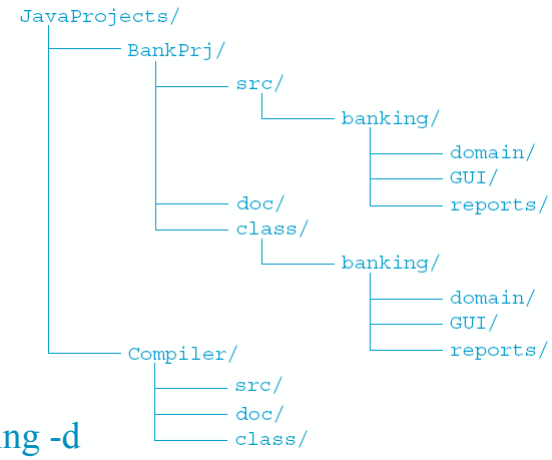
- Precedes all class declarations
- Tells the compiler where to find classes to use

Directory Layout and Packages

- Packages are stored in the directory tree containing the package name.
- Example, the "shipping" application packages:



Development



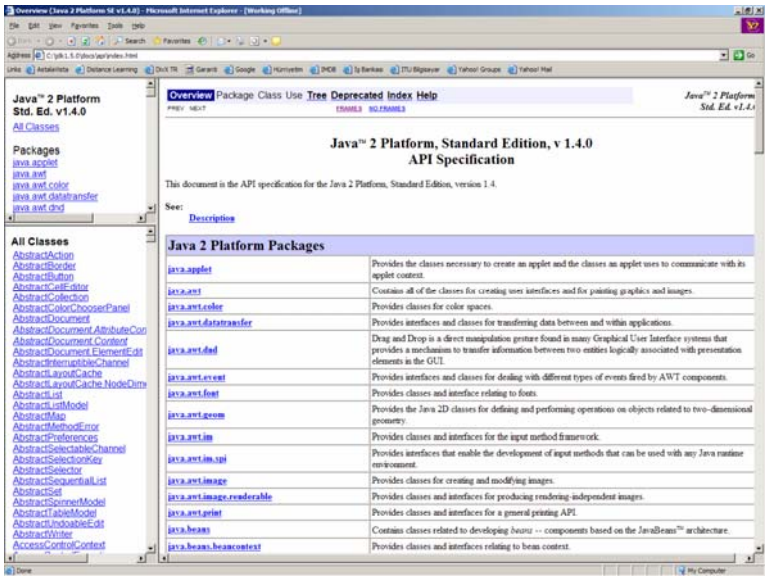
- Compiling using -d

```
cd JavaProjects/BankPrj/src
javac -d ../class banking/domain/*.java
```

Using the Java API Documentation

- A set of Hypertext Markup Language (HTML) files provides information about the API.
- One package contains hyperlinks to information on all of the classes.
- A class document includes the class hierarchy, a description of the class, a list of member variables, a list of constructors, and so on.

Example API Documentation Page



Declaring Java Technology Classes

```
public class Circle {
    private double x, y, r;  // The center and the radius of the circle
    public Circle ( double x, double y, double r ) {
        this.x = x;  this.y = y;   this.r = r;
    }
    public void setCenter(double a,double b){ x=a ; y=b ; }
    public void setRadius(double R){ r=R; }
    public double circumference( ) { return 2 * 3.14159 * r; }
    public double area( ) { return 3.14159 * r*r; }
}
```


Declaring Attributes

```
public class Circle {
    private double x, y, r;  // The center and the radius of the circle
    public Circle ( double x, double y, double r ) {
        this.x = x;  this.y = y;   this.r = r;
    }
    public void setCenter(double a,double b){ x=a ; y=b ; }
    public void setRadius(double R){ r=R; }
    public double circumference( ) { return 2 * 3.14159 * r; }
    public double area( ) { return 3.14159 * r*r; }
}
```

Declaring Methods


```
public class Circle {
   private double x, y, r;  // The center and the radius of the circle
   public Circle ( double x, double y, double r ) {
      this.x = x;  this.y = y;   this.r = r;
   }
   public void setCenter(double a,double b){ x=a ; y=b ; }
   public void setRadius(double R){ r=R; }
   public double circumference( ) { return 2 * 3.14159 * r; }
   public double area( ) { return 3.14159 * r*r; }
}
```


Accessing Object Members

- The "dot" notation : <object>.<member>
- This is used to access object members including attributes and methods
- Examples:

```
c.setCenter(8.7,23.5) ;
c.setRadius(3.14) ;
double a = c.area() ;
```

Information Hiding

```
Circle c ;
...
c = new Circle();
...
c.x = 2.0;
c.y = 2.0;
c.r = 1.0;
```


Declaring Constructors

```
public class Circle {
  private double x, y, r;  // The center and the radius of the circle
  public Circle ( double x, double y, double r ) {
     this.x = x;  this.y = y;   this.r = r;
  }
  public void setCenter(double a,double b){ x=a ; y=b ; }
  public void setRadius(double R){ r=R; }
  public double circumference( ) { return 2 * 3.14159 * r; }
  public double area( ) { return 3.14159 * r*r; }
}
```

The Default Constructor

- There is always at least one constructor in every class,
- If the writer does not supply any constructors, the default constructor is present automatically:
  - The default constructor takes no arguments,
  - The default constructor has no body.
- Enables you to create object instances with `new ClassName()` without having to write a constructor.

2# Object-Oriented Programming

- Exercise-1: “Java 2 Platform API Specification”
- Exercise-2: “Encapsulation”
- Exercise-3: “Creating a Simple Bank Package”

Sun microsystems
HANDS-ON LAB

3 Identifiers, Keywords & Types

Objectives

- Use comments in a source program
- Distinguish between valid and invalid identifiers
- Recognize Java technology keywords
- List the eight primitive types
- Define literal values for numeric and textual types
- Define the terms *class*, *object*, *member variable*, and *reference variable*
- Create a class definition for a simple class containing primitive member variables
- Declare variables of class type
- Construct an object using new
- Describe default initialization
- Access the member variables of an object using the dot notation

Comments

►Three permissible styles of comment in a Java technology program are :

// comment on one line

/* comment on one

or more lines */

/** documenting comment */

JAVADOC

Javadoc is a tool that parses the declarations and documentation comments in a set of source files and produces a set of HTML pages describing the classes, inner classes, interfaces, constructors, methods, and fields.

```
javadoc -sourcepath $(SRCDIR)
        -overview $(SRCDIR)/overview.html
        -d /java/jdk/build/api
        -use
        -splitIndex
        -windowtitle $(WINDOWTITLE)
        -doctitle $(DOCTITLE)
        -header $(HEADER)
        -bottom $(BOTTOM)
        -group $(GROUPCORE)
        -group $(GROUPEXT)
```

JAVADOC TAGS

Javadoc parses special tags when they are embedded within a Java doc comment. These doc tags enable you to autogenerate a complete, well-formatted API from your source code.

Tag	Introduced in JDK/SDK
@author	1.0
{@docRoot}	1.3
@deprecated	1.0
@exception	1.0
{@link}	1.2
@param	1.0
@return	1.0

Tag	Introduced in JDK/SDK
@see	1.0
@serial	1.2
@serialData	1.2
@serialField	1.2
@since	1.1
@throws	1.2
@version	1.0

Semicolons, Blocks, and White Space

► A statement is one or more lines of code terminated by a semicolon (;) :

```
totals = a + b + c
                + d + e + f ;
```

►A block is a collection of statements bound by opening and closing braces:

```
{
   x = y + 1 ;
   y = x + 1 ;
}
```

Semicolons, Blocks, and White Space

- A *block* can be used in a *class* definition

```
public class Date {
   private int day;
   private int month;
   private int year;
}
```

- Block statements can be nested
- Any amount of *whitespace* is allowed in a Java program


```
while ( i < large ) {
   a = a + i ;
   // nested block
   if ( a == max ) {
      b = b + a ;
      a = 0 ;
   }
   i = i + 1 ;
}
```

Identifiers

- Are names given to a variable, class, or method
- Can start with a letter, underscore(_), or dollar sign($)
- Are case sensitive and have no maximum length
- Examples:

```
identifier
userName
user_name
_sys_var1
$change
```


Java Keywords

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

const ve **goto**, Java'da tanımlı olmasalar da değişken isimleri olarak kullanılamazlar.

Primitive Types

- The Java programming language defines eight primitive types
  - Logical - **boolean**
  - Textual - **char**
  - Integral - **byte**, **short**, **int**, and **long**
  - Floating - **double** and **float**

Logical – boolean

- The boolean data type has two **literals**, `true` and `false`.
- For example, the statement

```
boolean truth = true;
```

declares the variable `truth` as `boolean` type and assigns it a value of `true`.

Textual – char and String

`char`

- Represents a 16-bit Unicode character
- Must have its literal enclosed in single quotes(' ')
- Uses the following notations:

`'a'`	The letter a
`'\t'`	A tab
`'\u????'`	A specific Unicode character (????) is replaced with exactly four hexadecimal digits. For example, '\u03A6' is the Greek letter phi[Φ]

Textual – char and String

String

- Is not a primitive data type; **it is a class**
- Has its literal enclosed in double quotes (" ")

```
"The quick brown fox."
```

- Can be used as follows:

```
String greeting = "Good Morning !!\n";
String err_msg = "Record Not Found !";
```

```
// declares and initializes a char variable
char ch = 'A' ;

// declares two char variables
Char ch1,ch2 ;

// declares two String variables and initializes them
String greeting = "Good Morning !!\n" ;
String errorMessage = "Record Not Found !!\n" ;

// declares two String variables
String str1,str2 ;
```

Note: initial values for str1 and str2 are null. Without initialization

```
System.out.println(str1) ;
```

causes to print null.

Integral — byte, short, int, and long

- Uses three forms - decimal, octal, or hexadecimal
  - `2` The decimal value is two.
  - `077` The leading zero indicates an octal value.
  - `0xBAAC` The leading `0x` indicates a hexadecimal value.
- Has a default `int`
- Defines `long` by using the letter `"L"` or `"l"`

Integral – byte, short, int, and long

Each of the integral data types have the following range:

Integer Length	Name or Type	Range
8 bits	byte	-2^7 to 2^7-1
16bits	short	-2^{15} to $2^{15}-1$
32bits	int	-2^{31} to $2^{31}-1$
64bits	long	-2^{63} to $2^{63}-1$

Floating Point – float, double

- Default is double
- Floating point literal includes either a decimal point or one of the following:
  - `E or e (add exponential value)`
  - `F or f (float)`
  - `D or d (double)`

`3.14`	A simple floating-point value (a `double`)
`6.02E23`	A large floating-point value
`2.718F`	A simple `float` size value
`123.4E+306D`	A large `double` value with redundant D

Floating Point – float, double

Floating-point data types have the following ranges:

Float Length	Name or Type
32 bits	float
64 bits	double


```
public class Assign {
      public static void main(String[] args){
          int x,y ;
          float z = 3.4115 ;
          boolean truth = true ;
          char c ;
          String str ;
          String str1 = "bye"  ;
          c = 'A' ;
          str = "Hi out there!" ;
          x = 6 ;
          y = 1000 ;
      }
}
```

```
y = 3.1415926;  // 3.1415926 is not an int; it
                // requires casting and decimal will be truncated

w = 175,000 ;  // Comma symbol cannot appear

truth = 1 ;  // this is a common mistake
             // made by C/C++ programmers

z = 3.1415926 ;  // Can't fit double into a
                 // float; This requires casting
```


Java Reference Types

- Beyond primitive types all others are of reference types
- A *reference variable* contains a handle to an object.
- Example:

next slide

```
public class MyDate {
        private int day = 1 ;
        private int month = 1 ;
        private int year = 1923 ;
        public MyDate(int day,int month,int year) {
           this.day = day ;
           this.month = month ;
           this.year = year ;
        }
        public void print(){...}
 }
public class TestMyDate {
     public static void main(String[] args){
           MyDate today = new MyDate(22,7,1964) ;
     }
}
```


Constructing and Initializing Objects

►Calling *new ClassName()* to allocate space for the new object results in:

- Memory allocation: Space for the new object is allocated and instance variables are initialized to their default values,
- Explicit attribute initialization is performed
- A constructor is executed
- Variable assignment is made to reference the object

►Example:

```
MyDate my_birth = new MyDate(11,7,1973) ;
```

Memory Allocation and Layout

- A declaration allocates storage only for a reference:

```
MyDate my_birth = new MyDate(11,7,1973) ;
```

my_birth	?????

- Use the new operator to allocate space for MyDate:

```
MyDate my_birth = new MyDate(11,7,1973) ;
```

my_birth	?????
day	0
month	0
year	0

Explicit Attribute Initialization

- Initialize the attributes:

```
MyDate my_birth = new MyDate(11,7,1973) ;
```

my_birth	?????
day	1
month	1
year	1923

- The default values are taken from the attribute declaration in the class.

Executing the Constructor

- Execute the matching constructor:

```
MyDate my_birth = new MyDate(11,7,1973) ;
```

my_birth	?????
day	11
month	7
year	1973

- In the case of an overloaded constructor, the first constructor may call another.

Assigning a Variable

- Assign the newly created object to the reference variable:

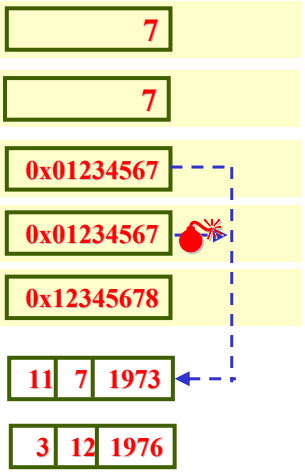
```
MyDate my_birth = new MyDate(11,7,1973) ;
```

my_birth	0x01abcdef
day	11
month	7
year	1973

Assigning Reference Types

Consider the following code fragment:

```
int x = 7 ;
int y = x ;
MyDate s = new MyDate(11,7,1973) ;
MyDate t = s ;
t = new MyDate(3,12,1976) ;
```



Pass-by-Value

- The Java programming language only passes arguments by value,
- When an object instance is passed as an argument to a method, the value of the argument is a reference to the object,
- The contents of the object can be changed in the called method, but the object reference is never changed.
- Example:

↳ next slide

```
public class PassTest {
        public static void changeInt(int myValue) {
            myValue = 55 ;
        }
        public static void changeObjectRef(MyDate ref) {
            ref = new MyDate(1,1,2000) ;
        }
        public static void changeObjectAttr(MyDate ref) {
            ref.setDay(4) ;
        }
        public static void main(String[] args){
            MyDate date ;
            int val ;
            val = 11 ;
            changeInt(val) ;
            System.out.println("Int value is: "+val) ;
            date = new MyDate(22,7,1964);
            changeObjectRef(date) ;
            date.print();
            changeObjectAttr(date);
            date.print();
        }
}
```


The this Reference

- Here are a few uses of the this keyword:
  - To reference local attribute and method members within a local method or constructor
  - The keyword `this` distinguishes a local method or constructor variable from an instance variable
  - To pass the current object as a parameter to another method or constructor

1

```
public class Circle {
    public double x, y, r;  // The center and the radius of the circle
    public Circle ( double x, double y, double r ) {
        this.x = x;  this.y = y;   this.r = r;
    }
    public double circumference( ) { return 2 * 3.14159 * r; }
    public double area( ) { return 3.14159 * r*r; }
}
```


2

```
public class Circle {
    public double x, y, r;
    // An instance method.  Returns the bigger of two circles.
    public Circle bigger(Circle c) {
        if (c.r > this.r) return c; else return this;
    }
    // A class method.  Returns the bigger of two circles.
    public static Circle bigger(Circle a, Circle b) {
        if (a.r > b.r) return a; else return b;
    }
    // Other methods omitted here.
}
```

Java Programming Language Coding Conventions

- Packages:

```
package shipping.object ;
```

- Classes

```
class AccountBook ;
```

- Interfaces

```
interface Account
```

- Methods

```
balanceAccount()
```


Java Programming Language Coding Conventions

- Variables:

```
currentCustomer
```

- Constants:

```
HEAD_COUNT
MAXIMUM_SIZE
```

3# Identifiers, Keywords, and Types

- Exercise-1: “Investigating Reference Assignment”
- Exercise-2: “Creating Customer Accounts”

Sun microsystems
HANDS-ON LAB

4 Expressions & Flow Control

Objectives

- Distinguish between instance and local variables
- Describe how instance variables are initialized
- Identify and correct a Possible reference before assignment compiler error
- Recognize, describe, and use Java operators
- Distinguish between legal and illegal assignments of primitive types
- Identify boolean expressions and their requirements in control constructs

- Recognize assignment compatibility and required casts in fundamental types
- Use `if`, `switch`, `for`, `while`, and `do` constructions and the labeled forms of break and continue as flow control structures in a program

Variables and Scope

- Local variables are
  - Variables which are defined inside a method and are called *local*, *automatic*, *temporary*, or *stack* variables
  - Created when the method is executed and destroyed when the method is exited
  - Variables that must be initialized before they are used or compile-time errors will occur

Variable Scope Example # 1

```
{
    int x = 12;
    /* only x available */
        {
          int q = 96;
          /* both x & q available */
        }
    /* only x available */
    /* q is out of scope */
}
```

Variable Scope Example # 2

```
public class ScopeExample {
    private int i=1 ;
    public void firstMethod(){
        int i=4, j=5 ;
        this.i = i + j ;
        secondMethod(7) ;
    }
    public void secondMethod(){
        int j=8 ;
        this.i = i + j ;
    }
}
```



```
public class TestScoping {
  public static void main(String []args){
      ScopeExample scope = new ScopeExample();
      scope.firstMethod() ;
}
```

Initializing Variables

variables defined outside of a method are initialized ***automatically***

boolean	**false**
char	'\u0000' **(null)**
byte	**(byte)0**
short	**(short)0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**

Initializing Variables

While variables defined outside of a method are initialized automatically, local variables ***must*** be initialized manually before use:

```
public class doComputation {
  int x = (int) (Math.random() * 100) ;
  int y ;
  int z ;
  if (x > 50){
     y = 9 ;
  }
  z = x + y ;
}
```

Operators

The Java operators are similar in style and function to those of C and C++:

Separator	. [] () ; ,
Associative	Operators
R to L	++ -- + - ~ ! (*data type*)
L to R	* / %
L to R	+ -
L to R	<< >> >>>
L to R	< > <= >= instanceof
L to R	== !=

Associative	Operators
R to L	&
L to R	^
L to R	|
L to R	&&
L to R	||
R to L	?:
R to L	= *= /= %= += -= <<= >>= >>>= &= ^= |=

Logical Operators

- The `boolean` operators are:
  - ! – NOT
  - & – AND
  - | – OR
  - ^ – XOR
- The short-curcuit `boolean` operators are:
  - && – AND
  - || – OR
- You can use these operators as follows:

```
MyDate d ;
if ((d != null) && (d.day > 31)) {
   // do something with d
}
```


Most Java operators are taken from other languages and behave as expected.

Relational and logical operators return a **`boolean`** result. The value **`0`** is not automatically interpreted as **`false`** and non-zero values are not automatically interpreted as **`true`**.

```
int i = 1 ;
if (i) // generates a compile error
if (i != 0) // correct
```

```
MyDate d ;
if ((d != null) && (d.day > 31)) {
    // do something with d
}
```

The second sub-expression is skipped when the first subexpression is false

if ((d != null) && (d.day > 31)) is SAFE

if ((d != null) & (d.day > 31)) is NOT safe

a=0

```
import java.io.* ;

public class Evaluate {
    public static void main(String[] args) {
        int a=1,b=1,c=3,d=10,y ;
        y = b + ( a==1 || ++b==d || ++d==11 ? b : d ) + 1 ;
        System.out.println("y=") ;
        System.out.println(y) ;
    }
}
```

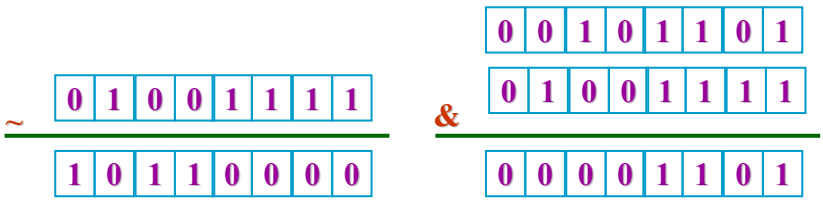
Bitwise Logical Operators

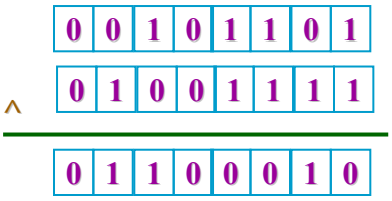
- The integer *bitwise* operators are:

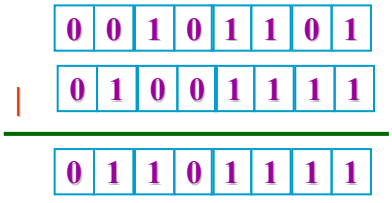
  ~ – Complement & – AND

  ^ – XOR | – OR

- Byte-sized examples:







Right-Shift Operators >> and >>>

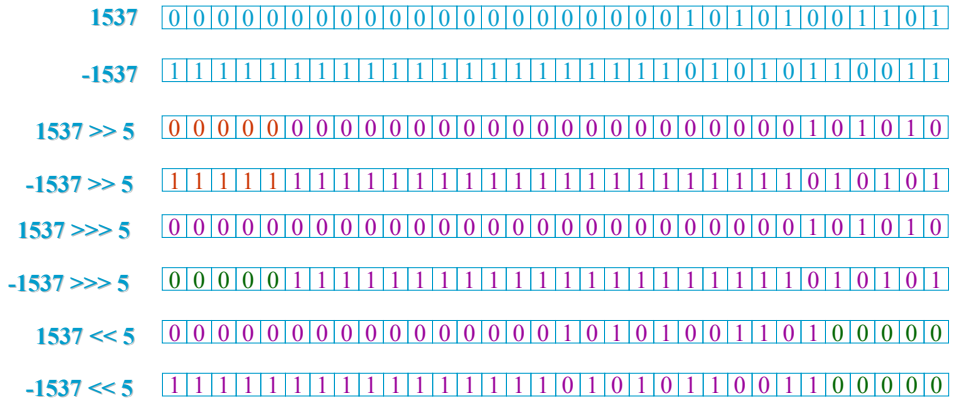
- *Arithmetic* or *signed* right-shift (>>) is used as follows:
  - **128 >> 1** returns $128/2^1=64$
  - **256 >> 4** returns $256/2^4=16$
  - **-256 >> 4** returns $-256/2^4=-16$
  - The sign bit is copied during the shift.
- A *logical* or *unsigned* right-shift (>>>) is:
  - Used for bit patterns.
  - The sign bit is not copied during the shift.

1	0	1	0	1	1	0	1	>> 2 =	1	1	1	0	1	0	1	1
1	0	1	0	1	1	0	1	>>> 2 =	0	0	1	0	1	1	0	1

Left-Shift Operator (<<)

- Left-shift (<<) works as follows:
  - **128 << 1** returns $128*2^1=256$
  - **16 << 2** returns $16*2^2=64$

Shift Operator Examples



String Concatenation With +

- The + operator
  - Performs `String` concatenation
  - Produces a new String:

```
String salutation = "Dr.";
String name = "Pete " + " " + "Seymour";
String title = salutation + " " + name;
```

- One argument must be a `String` object
- Non-strings are converted to `String` objects automatically

Casting

►If information is lost in an assignment, the programmer must confirm the assignment with a typecast.

►The assignment between `long` and `int` requires an explicit cast.

```
long bigValue = 99L;
int squashed = bigValue; //Wrong, needs a cast
int squashed = (int) bigValue; // OK

int squashed = 99L; //Wrong, needs a cast
int squashed = (int) 99L;  // OK, but...
int squashed = 99; // default integer literal
```


Promotion and Casting of Expressions

►Variables are automatically promoted to a longer form

(such as `int` to `long`)

►Expression is *assignment compatible* if the variable type is at least as large (the same number of bits) as the expression type.

```
long bigval = 6 ; // 6 is an int type, OK
int smallval = 99L ; // 99L is a long, illegal

double z = 12.414F; // 12.414F is float, OK
float z1 = 12.414; // 12.414 is double, illegal
```

Assignment compatibility between integer types

to \ from	byte	char	short	int	long
byte	✓	💣	💣	💣	💣
char	💣	✓	💣	💣	💣
short	✓	✓	✓	💣	💣
int	✓	✓	✓	✓	💣
long	✓	✓	✓	✓	✓

✓ Assignable
💣 Cast needed

```
short a, b, c ;
a = 1 ;
b = 2 ;
c = a + b ;     c = (short) (a + b) ;
```

For binary operators, such as the + operator, when the two operands are of primitive numeric types, the result is at least an **int** and has a value calculated by promoting the operands to the result type or promoting the result to the wider type of the operands. This might result in **overflow** or **loss of precision**.

Branching Statements

The `if`, `else` statement syntax:

```
if (boolean expression) {
    statement or block;
}

if (condition is true) {
    statement or block;
} else {
    statement or block;
}
```


Example

```
int count ;
count = getCount() ;
if (count<0) {
    System.out.println("Error: count value is negative.");
} else if (count > getMaxCount()) {
    System.out.println("Error: count value is too big.");
} else {
    System.out.println("There will be"+count+ "people for lunch today.");
}
```

```
if ( x > 5 )
  if ( y > 5 )
     System.out.println( "x and y are > 5" );
else
  System.out.println( “x is <= 5" );
```

```
if ( x > 5 ){
  if ( y > 5 )
     System.out.println( "x and y are > 5" );
}
else
  System.out.println( "x is <= 5" );
```

Branching Statements

The `switch` statement syntax is:

```
switch (expr1) {
   case expr2:
      statements;
   break;
   case expr3:
      statements;
   break;
   default:
      statements;
   break;
}
```

Common Programming Error

```
int i = 1 ;
switch (i) {
    case 1: System.out.println("bir") ;
    case 2: System.out.println("iki") ;
    break;
        case 3: System.out.println("uc") ;
        case 4: System.out.println("dort") ;
    default: System.out.println("on") ;
    break;
}
```


Looping Statements

The for statement

```
for (init_expr; boolean testexpr; alter_expr) {
     statement or block;
}
```

```
int faktoriyel,i;
for(i=1,faktoriyel=1;i<=9;i++)
   faktoriyel *= i;
System.out.println("9!=" + faktoriyel) ;
```

Looping Statements

The **`while`** loop:

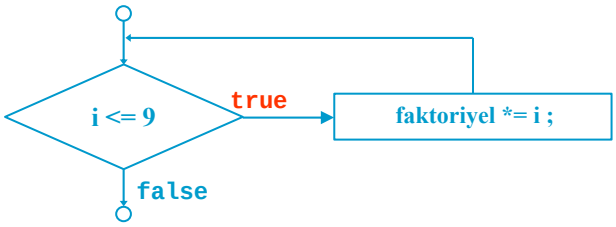
```
while (boolean test) {
     statement or block;
}
```



```
int faktoriyel = 1, i=1;
while ( i <= 9 ){
    faktoriyel *= i;

    i++;

}
System.out.println("9!=" + faktoriyel);
```



Looping Statements

The **do/while** loop:

```
do {
     statement or block;
} while (boolean test);
```

```
int faktoriyel = 1, i=1;
do {
   faktoriyel *= i;
   i++ ;
} while ( i <= 9 );
System.out.println("9!=" + faktoriyel)
```


Special Loop Flow Control

- `break [label];`
- `continue [label];`
- `label: statement; // where statement should be a loop.`

The **break** statement:

```
do {
    statement or block;
    if (condition is true)
      break;
    statement or block;

} while (boolean expression);
```


The **continue** statement:

```
do {
    statement or block;
    if (condition is true)
      continue;
    statement or block;

} while (boolean expression);
```

The **break** statement with a label named **outer**:

```
outer:
do {
statement or block;
   do {
       statement or block;
       if (condition is true)
         break outer;
       statement or block;

   } while (boolean expression);
statement or block;
} while (boolean expression);
Statement;
```


The **continue** statement with a label named **test**:

```
test:
do {
statement or block;
   do {
       statement or block;
       if (condition is true)
         continue test;
       statement or block;

   } while (boolean expression);
statement or block;
} while (boolean expression);
```

3# Identifiers, Keywords, and Types

- Exercise-1: "Using Loops and Branching Statements"
- Exercise-2: "Modifying the withdraw Method"
- Exercise-3: "Using Nested Loops"

An integer number is said to be a perfect number if its factors, including 1 (but not the number itself), sum to the number. For example, 6 is a perfect number because 6=1+2+3. Write a main function that determines all the perfect numbers between 1 and 5000. Print the factors of each perfect number to confirm that the number is indeed perfect.

```
import java.io.* ;

public class PerfectNumber {
    public static void main(String[] args) {
        int i,j,sum             ;

        for(j=2;j<5001;j++){
          sum = 1 ;
          for(i=2 ;i<=j/2;i++)
            if( (j % i) == 0 ) sum += i ;
          if(sum == j){
           System.out.print("\n"+j + " is a perfect number, since "+j+"= ");
           for(i=1;i<=j/2;i++)
             if ( (j%i) == 0 )
               System.out.print(i+" ");
          }
        }
  }
}
```


5 ARRAYS

Objectives

5 Arrays

- ▶Declare and create arrays of primitive, class, or array types
- ▶Explain why elements of an array are initialized
- ▶Given an array definition, initialize the elements of an array
- ▶Determine the number of elements in an array
- ▶Create a multidimensional array
- ▶Write code to copy array values from one array type to another

Declaring Arrays

5 Arrays

- Group data objects of the same type
- Declare arrays of primitive or class types

```
char s[];
Point p[];
char [] s;
Point [] p;
```

- Create space for a reference
- An array is an object; it is created with **new**.

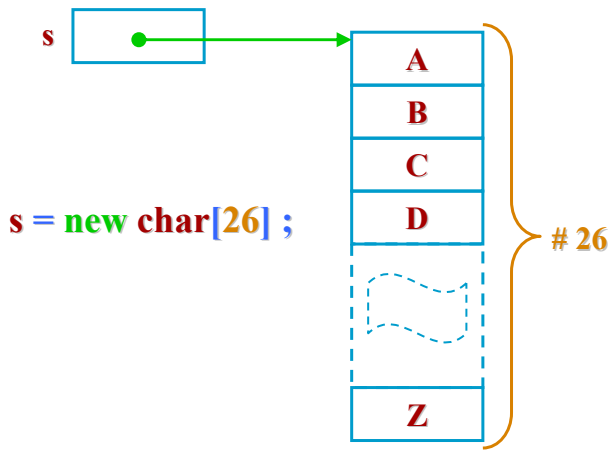
Creating Arrays

1

Use the **new** keyword to create an array object.

For example, a primitive (`char`) array:

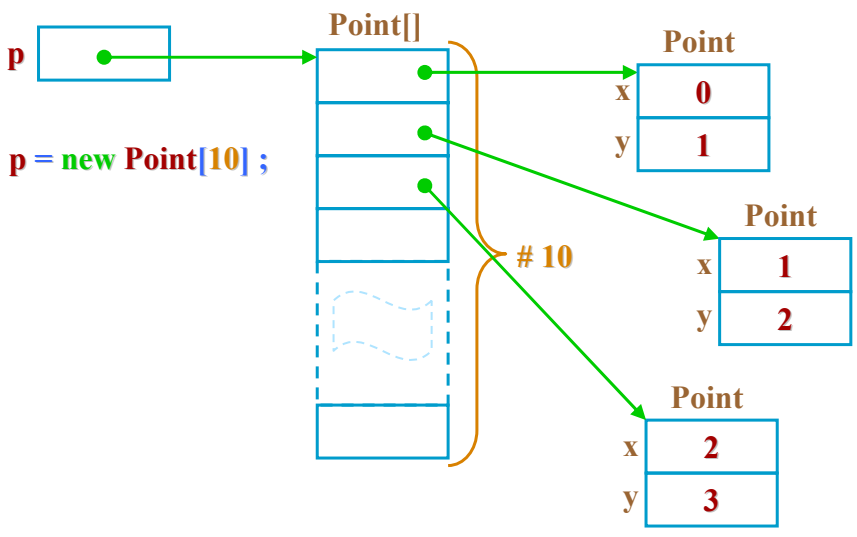
```
public char[] createArray() {
      char[] s;
      s = new char[26] ;
      for (int i=0; i<26; i++){
          s[i] = (char) ('A'+i) ;
      }
}
```

Creating Arrays

Another example, an object array:

```
public Point[] createArray() {
        Point[] s;
        p = new Point[10] ;
        for (int i=0; i<10; i++){
            p[i] = new Point(i,i+1) ;
        }
}
```

Initializing Arrays

- Initialize an array element
- Create an array with initial values :

```
String names[] ;
names = new String[3] ;
names[0]="Georgianna“;
names[1]="Jen“;
names[2]="Simon“;
```

❶

```
String names[] = {
"Georgianna",
"Jen",
"Simon"
};
```

❷


```
MyDate dates[] ;
dates = new MyDate[3] ;
dates[0]= new MyDate(22,7,1964);
dates[1]= new MyDate(1,1,2000);
dates[2]= new MyDate(22,12,1964);
```

❶

```
MyDate dates[]= {
      new MyDate(22,7,1964),
      new MyDate(22,1,1964),
      new MyDate(1,7,2000)
};
```

❷

Multidimensional Arrays

- Arrays of arrays

```
int twoDim [][] = new int [4][];
twoDim[0] = new int[5];
twoDim[1] = new int[5];
```

rectangular array

```
int twoDim [][] = new int [][4];
```

illegal

Multidimensional Arrays

- Non-rectangular arrays of arrays

```
twoDim[0] = new int[2];
twoDim[1] = new int[4];
twoDim[2] = new int[6];
twoDim[3] = new int[8];
```

- Array of four arrays of five integers each

```
int twoDim[][] = new int[4][5];
```

Array Bounds

All array subscripts begin at 0 :

```
int list[] = new int [10];
for (int i = 0; i < list.length; i++){
  System.out.println(list[i]);
}
```


Array Resizing

- Cannot resize an array
- Can use the same reference variable to refer to an entirely new array

```
int elements[] = new int[6];
elements = new int[10];
```

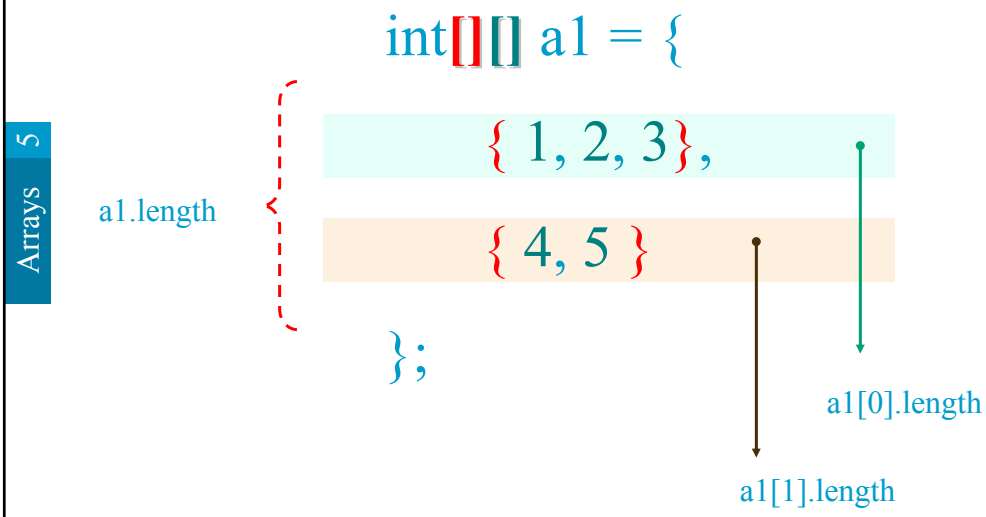
Copying Arrays

The System.arraycopy() method

```
//original array
int elements[] = { 1, 2, 3, 4, 5, 6 };
:
// new larger array
int hold[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
// copy all of the elements array to the hold
// array, starting with the 0th index
System.arraycopy(elements, 0, hold, 0,elements.length);
```



```
public class Arrays {
  public static void main(String[] args) {
        int[] a1 = { 1, 2, 3, 4, 5 };
        int[] a2;
        a2 = a1;
        for(int i = 0; i < a2.length; i++)
           a2[i]++;
        for(int i = 0; i < a1.length; i++)
           prt("a1[" + i + "] = " + a1[i]);
  }
}
```



5# Arrays

- Exercise-1: "Manipulating Arrays"
- Exercise-2: "Using Arrays to Represent Multiplicity"

Practice Session

write a method

```
public static int[] ins(int[] a,int x)
```

which takes a sorted integer array, **a**, and an integer value **x**, and then returns another sorted array, which includes elements of **a** and **x**.


```
import java.io.* ;

public class Insert {
    public static int [] ins(int[] a,int x){
        int  r[],i,p;
        r = new int[a.length+1] ;
        for(i=0;i<a.length;i++){
          if(a[i]>x)
            break;
          r[i] = a[i] ;
        }
        r[i] = x ;
        i++;
        for(;i<=a.length;i++)
          r[i] = a[i-1] ;
        return r ;
    }
```

```
public static void print(int a[]){
        System.out.print("\n"+a[0]);
        for(int i=1;i<a.length;i++)
            System.out.print(","+a[i]);
    }
    public static void main(String[] args) throws IOException {
        int c[]={1,2,3,4,6,7,8,9};
        int[] d ;
        int e = 5 ;
        d = ins(c,e) ;
        print(c);
        print(d);
    }
}
```


6 CLASS DESIGN

Objectives

► Define *inheritance*, *polymorphism*, *overloading*, *overriding*, and *virtual method invocation*,

►Use the access modifiers *protected* and "*package-friendly*"

►Describe the concept of constructor and method overloading,

►Describe the complete object construction and initialization operation

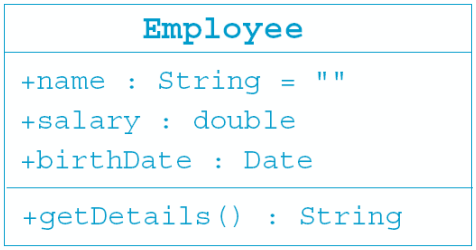
Objectives

►In a Java program, identify the following:

- Overloaded methods and constructors
- The use of *this* to call overloaded constructors
- Overridden methods
- Invocation of *super* class methods
- Parent class constructors
- Invocation of parent class constructors

Subclassing

The `Employee` class

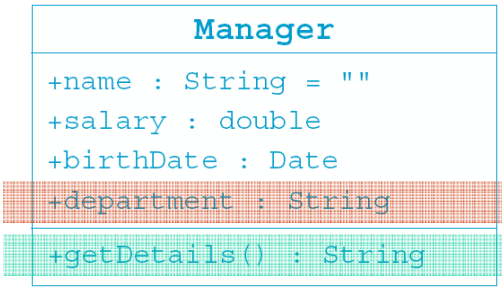



```
public class Employee {
    public String name="";
    public double salary;
    public Date birthDate ;
    public String getDetails(){...}
}
```

Class Design 6

Subclassing

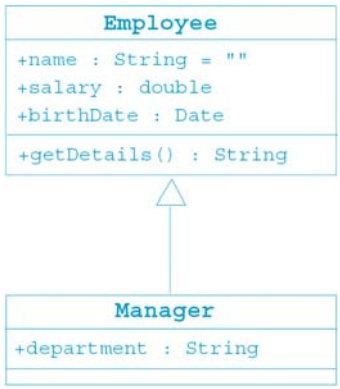
The `Manager` class




```
public class Manager {
    public String name="";
    public double salary;
    public Date birthDate ;
    public String department;
    public String getDetails(){...}
}
```

Class Design 6

Subclassing



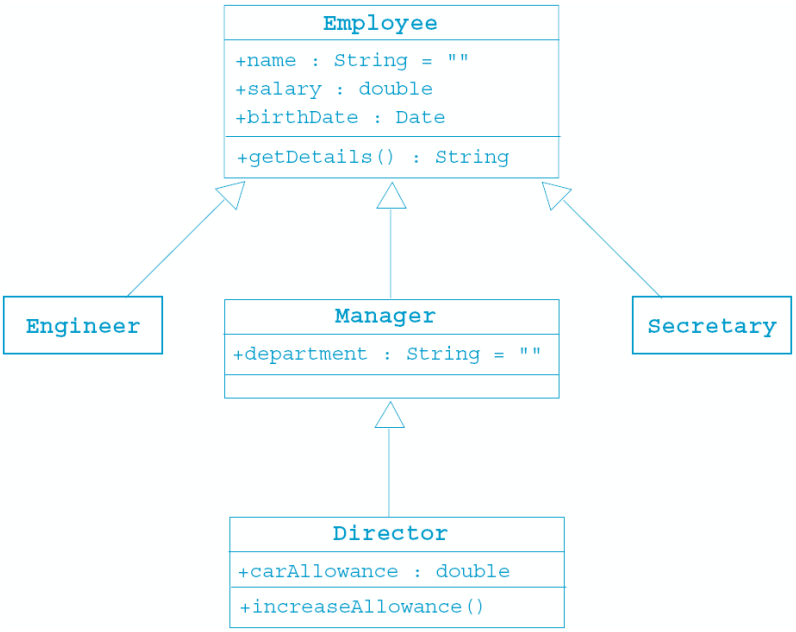

```
public class Employee {
    public String name="";
    public double salary;
    public Date birthDate ;
    public String getDetails(){...}
}
```

```
public class Manager extends Employee{
    public String department ;
}
```


Single Inheritance

- When a class inherits from only one class, it is called *single inheritance*.
- Single inheritance makes code more reliable.
- *Interfaces* provide the benefits of multiple inheritance without drawbacks.
- Syntax of a Java Class:

```
<modifier> class <name> [extends <superclass>] {
        <declarations>*
}
```



Access Control

Modifier	Same Class	Same Package	Subclass	Universe
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

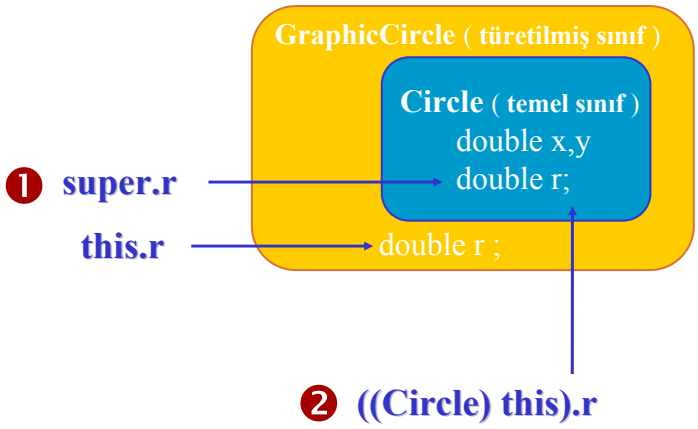
Overriding Methods

- A subclass can modify behavior inherited from a parent class.
- Subclass can create a method in a subclass with a different functionality than the parent's method but with the same
  - Name
  - Return type
  - Argument list

The `super` Keyword

- `super` is used in a class to refer to its superclass.
- `super` is used to refer to the member variables of superclass.
- Superclass behavior is invoked as if the object was part of the superclass.
- Behavior invoked does not have to be in the superclass; it can be further up in the hierarchy.

```
public class GraphicCircle extends Circle {
   Color outline, fill;
   float r;  // New variable.  Resolution in dots-per-inch.
   public GraphicCircle(double x, double y, double rad, Color o, Color f){
      super(x, y, rad); outline = o; fill = f;
   }
   public void setResolution(float resolution) { r = resolution; }
   public void draw(DrawWindow dw) {
       dw.drawCircle(x, y, r, outline, fill); }
}
```


```
public class A{
    int x ;
}
public class B extends A {
    int x ;
}
public class C extends B{
    int x ;
}
```

x

this.x

super.x

((B)this).x

((A)this).x

super.super.x

Polymorphism

►*Polymorphism* is the ability to have many different forms; for example, the Manager class has access to methods from Employee class.

►An object has only one form.

►A variable has many forms; it can refer to objects of different forms.

►Polymorphism is a runtime issue.

►Overloading is a compile-time issue.

```
import java.io.* ;

public class Square {
      protected float edge=1 ;
      public Square(int edge)    {
         this.edge = edge ;
      }
      public float area()   {
         return edge * edge ;
      }
      public void print()   {
         System.out.println("Square Edge="+edge);
      }
}
```



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

public class Cube extends Square {
      public Cube(int edge){
         super(edge) ;
      }
      public float area() {
         return 6.0F * super.area() ;
      }
      public void print()      {
         System.out.println("Cube Edge="+edge);
      }
}
```

```
public class PolymorphSample {
        public static void main(String[] args) {
                Square[] sq ;
                sq = new Square[5] ;
                sq[0] = new Square(1) ;
                sq[1] = new Cube(2) ;
                sq[2] = new Square(3);
                sq[3] = new Cube(4) ;
                sq[4] = new Square(5) ;
                for (int i=0;i<5;i++)  sq[i].print();
        }
}
```



```
public class A {
  public int i = 1;
  public int f() { return i;}
}
public class B extends A {
 public int i = 2;
 public int f(){ return -i;}
}
```

```
public class override_test {
    public static void main(String args[]) {
        B b = new B();
        System.out.println(b.i);
        System.out.println(b.f());
        A a = (A) b;
        System.out.println(a.i);
        System.out.println(a.f());
    }
}
```

polymorphism

Virtual Method Invocation

- Compile-time and run-time type

```
Square S = new Square(1.0) ;
Cube   C = new Cube(1.0) ;
S.area();
C.area();
```

compile-time type

- Virtual method invocation:

```
Square q = new Cube(1.0) ;
q.area();
```

run-time type


```
Employee e = new Manager() //legal
e.department = "Finance" //illegal

Employee [] staff = new Employee[1024];
staff[0] = new Manager();
staff[1] = new Employee();
```

Rules About Overridden Methods

- Must have a return type that is identical to the method it overrides
- Cannot be less accessible than the method it overrides
- Cannot throw more exceptions than the method it overrides


```
public class Parent {
    public void doSomething() {}
}
public class Child extends Parent {
    private void doSomething() {}
}
public class UseBoth {
    public void doOtherThing() {
        Parent p1 = new Parent();
        Parent p2 = new Child();
        p1. doSomething();
        p2. doSomething();
    }
}
```

Heterogeneous Collections

- Collections with a common class are called *homogenous* collections.

```
MyDate[] dates = new MyDate[2] ;
dates[0] = new MyDate(22,12,1964)  ;
dates[1] = new MyDate(22,7,1964)  ;
```

- Collections with dissimilar objects is a *heterogeneous* collection:

```
Employee[] staff = new Employee[1024] ;
staff[0] = new Manager() ;
staff[1] = new Employee() ;
staff[2] = new Engineer(),
```


Polymorphic Arguments

- Since a Manager *is an* Employee:

```
// In the Employee class
public TaxRate findTaxRate(Employee e) {
}
// Meanwhile, elsewhere in the application class
Manager m = new Manager();
:
TaxRate t = findTaxRate(m);
```

The instanceof Operator

```
public class Employee extends Object
public class Manager extends Employee
public class Contractor extends Employee
-----------------------------------------
public void method(Employee e) {
    if (e instanceof Manager) {
       // Gets benefits and options along with salary }
    else if (e instanceof Contractor) {
       // Gets hourly rates
    }
    else {
       // temporary employee
    }
}
```


Casting Objects

- ▶Use **instanceof** to test the type of an object.
- ▶Restore full functionality of an object by casting.
- ▶Check for proper casting using the following guidelines:
- ▶Casts up hierarchy are done implicitly.
- ▶Downward casts must be to a subclass and is checked by compiler.
- ▶The reference type is checked at runtime when runtime errors can occur.

```
public void doSomething(Emplyee e) {
    if(e instanceof Manager) {
        Manager m = (Manager) e ;
        System.out.println("This is the manager of"+
                                m.getDepartment())  ;
    }
    // rest of operation
}
```


Overloading Method Names

- It can be used as follows:

```
public void print(int i)
public void print(float f)
public void print(String s)
```

- Argument lists *must* differ.
- Return types *can* be different, but it is not sufficient for the return type to be the only difference. The argument lists of overloaded methods must differ.

Overloading Constructors

- As with methods, constructors can be overloaded.
- Example:

  public Employee(String name, double salary, Date DoB)

  public Employee(String name, double salary)

  public Employee(String name, Date DoB)

- Argument lists *must* differ.
- You can use the this reference at the first line of a constructor to call another constructor.


```
public class Employee {
    private static final double BASE_SALARY = 15000.0 ;
    private String name ;
    private double salary ;
    private Date birthDate ;
    public Employee(String name, double salary, Date DoB) {   1
        this.name = name ;
        this.salary = salary ;
        this.birthDate = DoB ;
    }
    public Employee(String name, double salary){              2
        this(name,salary,null) ;
    }
```

```
        public Employee(String name, Date DoB) {          ❸
            this(name,BASE_SALARY,DoB) ;
        }
        public Employee(String name){                     ❹
            this(name,BASE_SALARY) ;
        }
        // more Employee code...
    }
```


Example # 2

```
public class Circle {
    public double x, y, r;
    public Circle ( double x, double y, double r ) {
        this.x = x; this.y = y; this.r = r;
    }
    public Circle ( double r ) { x = 0.0; y = 0.0; this.r = r; }
    public Circle ( Circle c ) { x = c.x; y = c.y; r = c.r; }
    public Circle ( ) { x = 0.0; y = 0.0; r = 1.0; }
    public double circumference ( ) { return 2 * 3.14159 * r; }
    public double area ( ) { return 3.14159 * r*r; }
}
```

```
Circle c1 = new Circle ( 1.414, -1.0, .25 ) ;
Circle c2 = new Circle (3.14) ;
Circle c3 = new Circle () ;
Circle c4 = new Circle (c3) ;
Circle c4 = new Circle (new Circle(1.0)) ;
```


Constructors Are Not Inherited

►A subclass inherits all methods and variables from the superclass (parent class).

►A subclass does not inherit the constructor from the superclass.

►Two ways to include a constructor are

- Use the default constructor
- Write one or more explicit constructors

Invoking Parent Class Constructors

►To invoke a parent constructor, you must place a call to `super` in the first line of the constructor

►You can call a specific parent constructor by the arguments that you use in the call to `super`

►if no `this` or `super` call is used in a constructor, then the compiler adds an implicit call to super() that calls the parent no argument constructor(which could be the "default" constructor)

►if the parent class defines constructors, but does not provide a no argument constructor, then a compiler error message is issued.


```
public class Manager extends Employee {
    private String department ;
    public Manager(String name, double salary, String dept) {
        super(name,salary) ;
        department = dept ;
    }
    public Manager(String name, String dept){
        super(name) ;
        department = dept ;
    }
    public Manager(String dept) {
        department = dept ;                                  super() ;
    }
}
```

Eğer türetilmiş sınıfta bir kurucu fonksiyon tanımlı değil ise derleyici bir tane yaratır. Yaratılan bu kurucu fonksiyon temel sınıfın kurucu fonksiyonunu çağırır :

```
class A {
   int i;
   public A() {
    i = 3;
  }
}
class B extends A {
   // Default constructor: public B() { super(); }
}
```

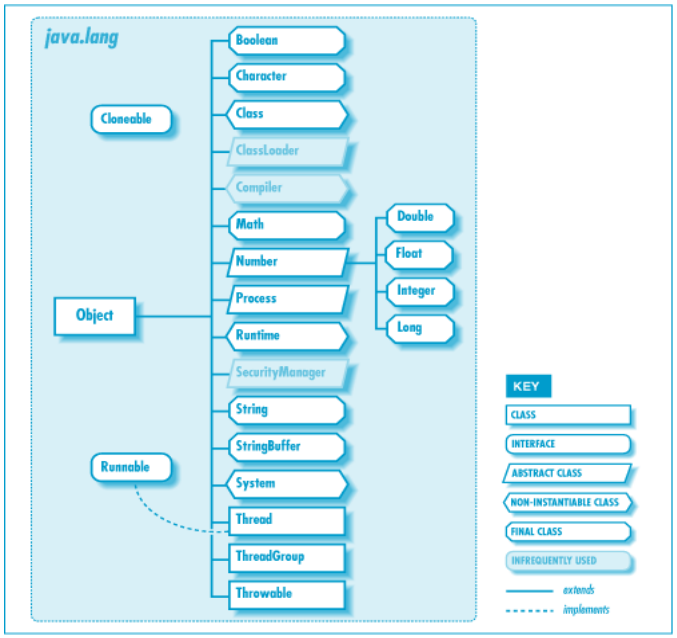

The Object Class

- The Object class is the root of all classes in Java
- A class declaration with no extends clause, implicitly uses "extends Object"

```
public class Employee {
   ...
}
```

is equivalent to:

```
public class Employee extends Object {
   ...
}
```



The class Class

Classes in Java source code are represented at run-time by instances of the `java.lang.Class` class. There's a Class object for every class you use; this Class object is responsible for producing instances for its class

Classes in the Java language have a run-time representation. There is a class named Class, instances of which contain run-time class definitions. If you're handed an object, you can find out what class it belongs to. In a C or C++ program, you may be handed a pointer to an object, but if you don't know what type of object it is, you have no way to find out. In the Java language, finding out based on the run-time type information is straightforward.

```
String myString = “Try!" ;
Class c = myString.getClass();
or
Class c = String.class;
```

```
String s = “Relations between IMF and Turkey";
Class strClass = s.getClass();
System.out.println( strClass.getName() );
// prints "java.lang.String"
String s2 = (String) strClass.newInstance();
```



```
try{
    Class c = Class.forName("java.lang.String") ;
    Object o ;
    String s ;

    o = (Object) c.newInstance();
    if(o instanceof String){
      s = (String) o ;
      System.out.println(s ) ;
    }
  }
  catch(Exception e){
    System.out.println("something is wrong.") ;
  }
```

The == Operator Compared With equals

- The == operator determines if two references are identical to each other (that is, refer to the same object).
- The `equals` method determines if objects are "equal" but not necessarily identical.
- The `Object` implementation of the equals method uses the == operator.
- User classes can override the `equals` method to implement a domain-specific test for equality.
- Note: You should override the `hashCode` method if you override the `equals` method.


```
public class MyDate {
    private int day ;
    private int month ;
    private int year ;
    public MyDate(int day, int month, int year){
       this.day = day ; this.month = month ; this.year = year ;
    }
    public boolean equals(Object o) {
       boolean result = false ;
       if( (o != null) && (o instanceof MyDate) ){
          MyDate d = (MyDate) o ;
         if( (day == d.day) && (month == d.month) && (year == d.year) )
            result = true ;
       }
       return result ;
    }
```

```
    public int hashCode() {
        return (
                    (new Integer(day).hashCode())
                ^ (new Integer(day).hashCode())
                ^ (new Integer(day).hashCode())
               ) ;
    }
}
```



```
public class TestEquals {
    public static void main(String[] args) {
        MyDate date1 = new MyDate(13, 3, 1976) ;
        MyDate date2 = new MyDate(13, 3, 1976) ;

        if( date1 == date2 )
          System.out.println("date1 is identical to date2") ;
        else
          System.out.println("date1 is not identical to date2") ;

        if( date1.equals(date2) )
          System.out.println("date1 is equal to date2") ;
        else
          System.out.println("date1 is not equal to date2") ;
```

```
        System.out.println("set date2 to date1") ;
                date2 = date1 ;
                if( date1 == date2 )
                  System.out.println("date1 is identical to date2") ;
                else
                  System.out.println("date1 is not identical to date2") ;
        }
}
```


The `toString` Method

- Converts an object to a `String`.
- Used during string concatenation.
- Override this method to provide information about a user-defined object in readable format.
- Primitive types are converted to a `String` using the wrapper class's `toString` static method.

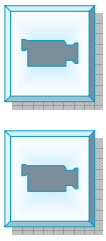
```
String one = String.valueOf( 1 );
String two = String.valueOf( 2.0f );
String notTrue = String.valueOf( false );

String date = String.valueOf( new Date() );
System.out.println( date );
// Wed Jul 11 12:46:16 GMT+03:00 2001

date = null;
System.out.println( date );
// null
```


Wrapper Classes

- Look at primitive data elements as Objects



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

```
int pInt = 500 ;
Integer wInt = new Integer(pInt) ;
int p2 = wInt.intValue();
```

```
public class StringTest {
  public static void main(String[] args)  {
        String s = "123";
        Integer wInt = new Integer(Integer.parseInt(s)) ;
        System.out.println(  wInt )                    ;
        System.out.println(  wInt.intValue() )         ;
        System.out.println(  wInt.floatValue() )       ;
              System.out.println(  wInt.toString() )   ;
         }
}
```



```
public class StringTest{
  public static void main(String[] args) {
       String s = "-123.45";
       Double wDouble = new Double(Double.parseDouble(s));
       System.out.println(  wDouble ) ;
       System.out.println(  wDouble.intValue() )       ;
       System.out.println(  wDouble.toString() )       ;
  }
}
```

6# Class Design

- Exercise-1: “Creating Subclasses of Bank Accounts”
- Exercise-2: “Creating Customer Accounts”

7 ADVANCED CLASS FEATURE

Objectives

- Describe static variables, methods, and initializers
- Describe final classes, methods, and variables
- Explain how and when to use abstract classes and methods
- Explain how and when to use inner classes
- Distinguish between static and non-static inner classes
- Explain how and when to use an interface

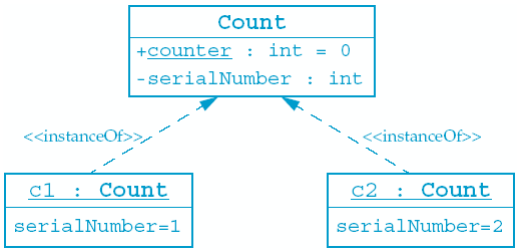
The static Keyword

- The `static` keyword is used as a modifier on variables, methods, and inner classes.
- The `static` keyword declares the attribute or method is associated with the class as a whole rather than any particular instance of that class.
- Thus `static` members are often called "class members", such as "class attributes" or "class methods".

Class Attributes

- Are shared among all instances of a class

```
1 public class Count {
2   private int serialNumber;
3   public static int counter = 0;
4
5   public Count() {
6     counter++;
7     serialNumber = counter;
8   }
9 }
```



Class Attributes

- Can be accessed from outside the class if marked as public without an instance of the class

```
1 public class OtherClass {
2       public void incrementNumber() {
3               Count.counter++;
4       }
5 }
```

Class Methods

- You can invoke static method without any instance of the class to which it belongs.


```
1 public class Count {
2 private int serialNumber;
3 private static int counter = 0;
4
5 public static int getTotalCount() {
6 return counter;
7 }
8
9 public Count() {
10 counter++;
11 serialNumber = counter;
12 }
13 }
```

```
1 public class TestCounter {
2 public static void main(String[] args) {
3 System.out.println("Number of counter is "
4 + Count.getTotalCount());
5 Count count1 = new Count();
6 System.out.println("Number of counter is "
7 + Count.getTotalCount());
8 }
9 }
```

The output of the TestCounter program is:

Number of counter is 0

Number of counter is 1

```
public class Circle {
 static int num_circles = 0;
 public double x, y, r;
  public Circle(double x, double y, double r) {
      this.x = x; this.y = y; this.r = r;
      num_circles++;
    }
    public Circle(double r) { this(0.0, 0.0, r); }
    public Circle(Circle c) { this(c.x, c.y, c.r); }
    public Circle() { this(0.0, 0.0, 1.0); }
    public double circumference() { return 2 * 3.14159 * r; }
    public double area() { return 3.14159 * r*r; }
}
```



```
public class Circle {
   public static final double PI = 3.14159265358979323846;
   public double x, y, r;
   // ... etc....
}

public double circumference() { return 2 * Circle.PI * r; }
```

derleyici

```
public class Circle {
  double x, y, r;
  public boolean isInside(double a, double b) {
     double dx = a - x;
     double dy = b - y;
     double distance = Math.sqrt(dx*dx + dy*dy);
     if (distance < r) return true;
     else return false;
   }
  // Constructor and other methods omitted.
}
```



```
public class Circle {
   public double x, y, r;
   // An instance method.  Returns the bigger of two circles.
   public Circle bigger(Circle c) {
      if (c.r > r) return c; else return this;
   }
   // A class method.  Returns the bigger of two circles.
   public static Circle bigger(Circle a, Circle b) {
      if (a.r > b.r) return a; else return b;
   }
  // Other methods omitted here.
}
```

1.
```
Circle a = new Circle(2.0);
Circle b = new Circle(3.0);
Circle c = a.bigger(b);
b.bigger(a);
```

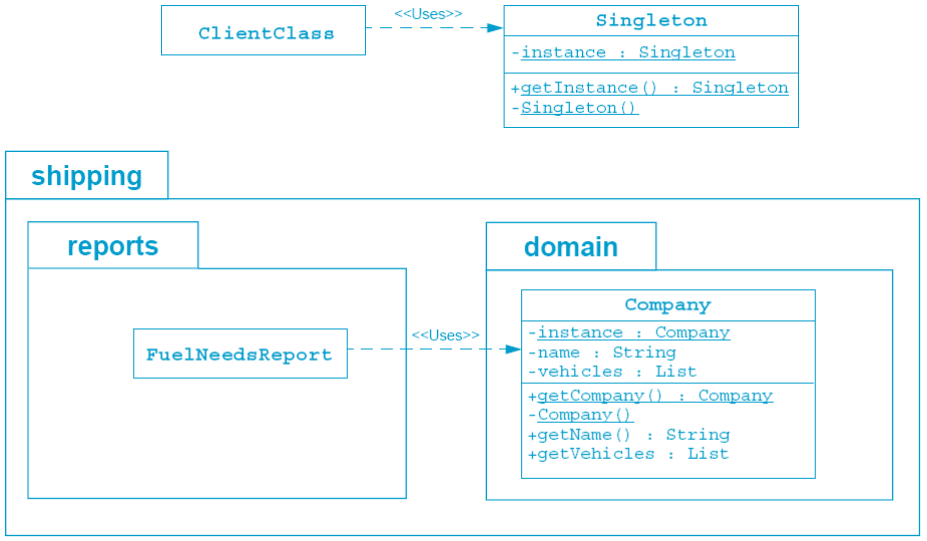
2.
```
Circle a = new Circle(2.0);
Circle b = new Circle(3.0);
Circle c = Circle.bigger(a,b);
```


Static Initializers

- A class can contain code in a static block that does not exist within a method body.
- Static block code executes only once, when the class is loaded.
- A static block is usually used to initialize static (class) attributes

```
public class Circle {
    static private double sines[] = new double[1000];
    static private double cosines[] = new double[1000];
    static {
            double x, delta_x;
            delta_x = (Circle.PI/2)/(1000-1);
            for(int i = 0, x = 0.0; i < 1000; i++, x += delta_x) {
             sines[i] = Math.sin(x);
             cosines[i] = Math.cos(x);   }
     }
      // The rest of the class omitted.
}
```


The Singleton Design Pattern



Implementing the Singleton Design Pattern

```
1 package shipping.domain;
2
3 public class Company {
4 private static Company instance = new Company();
5 private String name;
6 private Vehicle[] fleet;
7
8 public static Company getCompany() {
9 return instance;
10 }
11
12 private Company() {...}
13
14 // more Company code ...
15 }
```


Usage Code

```
1   package shipping.reports;
2
3   import shipping.domain.*;
4
5   public class FuelNeedsReport {
6       public void generateText(PrintStream output) {
7           Company c = Company.getCompany();
8           // use Company object to retrieve the fleet vehicles
9       }
10  }
```

The `final` Keyword

- You cannot subclass a `final` class.
- You cannot override a `final` method.
- A `final` variable is a constant.
- You can set a final variable once, but that assignment can occur independently of the declaration; that is called "blank final variable."
  - A blank final instance attribute must be set in every constructor.
  - A blank final method variable must be set in the method body before being used.

`final` Classes

- The Java programming language allows you to apply the keyword final to classes. if you do this, the class cannot be inherited. For example, the class `java.lang.String` is a `final` class.
- This is done for security reasons.

`final` Methods

►You can also make individual methods as final. Methods marked final cannot be overridden. for security reasons, you should make a method final if the method has an implementation that should not be changed and is critical to the consistent state of the object.

►Methods declared final are sometimes used for optimization. The compiler can generate code that causes a direct call to the method, rather than the usual virtual method invocation that involves a runtime lookup.

`final` Variables

►If a variable is marked as final, the effect is to make it a constant. Any attempt to change the value of final variable causes a compiler error:

```
public class Bank{
  private static final double DEFAULT_INTEREST_RATE = 3.2 ;
  ... // more declarations
}
```

7# Advanced Class Features

►Exercise-1: "Working with the `static` and `final` Keywords"

►Modify the Bank class to Implement the Singleton Design Pattern

Yokedici Fonksiyonlar


```
protected void finalize() throws IOException {
    if (fd != null) close();
}
```

finalize fonksiyonu, C++'ın aksine, nesnenin erimi dışına çıkılınca derleyici tarafından değil, çöp toplayıcı süreç tarafından, yürütme zamanında çalıştırılır. Java nesnenin ne zaman bellekten kaldırılacağını garanti etmez. Bu nedenle nesne belirli bir süre daha erişilebilir durumda olacaktır.

Nesnelerin Yokedilmesi

- **Çöp toplama**
  - bir süreç kullanılmayan bellek alanlarını saptar : Java Grabage Collector : düşük-seviyeli süreç
  - delete ?

```
String processString(String s)
{
   StringBuffer b = new StringBuffer(s);
   return b.toString();
}
```


Kullanılan Bellek Alanlarının Çöplüğe Atılması

```
public static void main(String argv[])
{
   int big_array[] = new int[100000];
   int result = compute(big_array);
   big_array = null;
   for(;;) handle_input();
}
```

Yokedici Fonksiyon Zinciri

```
public class Point {
    ...
    protected void finalize() {
        ...
        super.finalize() ;
    }
}
public class Circle extends Point {
    ...
    protected void finalize() {
        ...
        super.finalize() ;
    }
}
```


Soyut Sınıflar (=abstract classes)

Soyut sınıflar, sadece metod isimlerini içeren ancak metodların gövdelerinin tanımlanmadığı sınıflardır. Soyut sınıftan bir nesne yaratılamaz:

```
public abstract class Shape {
    public abstract double area();
    public abstract double circumference();
}
```

Soyut sınıflar C++'daki **pure virtual** sınıflara karşılık düşmektedir:

```
class Shape {
    public :
      double area()=0;
      double circumference()=0;
}
```

```
class Circle extends Shape {
    protected double r;
    protected static final double PI = 3.14159265358979323846;
    public Circle() { r = 1.0; }
    public Circle(double r) { this.r = r; }
    public double area() { return PI * r * r; }
    public double circumference() { return 2 * PI * r; }
    public double getRadius() { return r; }
}
```



```
class Rectangle extends Shape {
    protected double w, h;
    public Rectangle() { w = 0.0; h = 0.0; }
    public Rectangle(double w, double h){ this.w=w;this.h=h;}
    public double area() { return w * h; }
    public double circumference() { return 2 * (w + h); }
    public double getWidth() { return w; }
    public double getHeight() { return h; }
}
```

```
Shape[] shapes = new Shape[3];
shapes[0] = new Circle(2.0);
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);
double total_area = 0;
for(int i = 0; i < shapes.length; i++)
   total_area += shapes[i].area();
```


interface

Java'nın çoklu kalıtımı desteklemediğini hatırlayınız! Bunun anlamı Java'da birden fazla temel sınıf içeren bir sınıf türetemezsiniz. Java çoklu kalıtımın işlevini yerine getiren başka bir yapıya sahiptir : **interface**.

```
public interface Drawable {
   public void setColor(Color c);
   public void setPosition(double x, double y);
   public void draw(DrawWindow dw);
}
```

Uses of Interfaces

- Interfaces are useful for
  - Declaring methods that one or more classes are expected to implement
  - Determining an object's programming interface without revealing the actual body of the class
  - Capturing similarities between unrelated classes without forcing a class relationship
  - Describing "function-like" objects that can be passed as parameters to methods invoked on other objects


```
public class DrawableRectangle extends Rectangle implements Drawable {
    private Color c;
    private double x, y;
    public DrawableRectangle(double w, double h) { super(w, h); }
    public void setColor(Color c) { this.c = c; }
    public void setPosition(double x, double y) { this.x = x; this.y = y; }
    public void draw(DrawWindow dw) {
        dw.drawRect(x, y, w, h, c);
    }
}
```

```
Shape[] shapes = new Shape[3];
Drawable[] drawables = new Drawable[3];
DrawableCircle dc = new DrawableCircle(1.1);
DrawableSquare ds = new DrawableSquare(2.5);
DrawableRectangle dr = new DrawableRectangle(2.3, 4.5);
shapes[0] = dc;   drawables[0] = dc;
shapes[1] = ds;   drawables[1] = ds;
shapes[2] = dr;   drawables[2] = dr;
double total_area = 0;
for(int i = 0; i < shapes.length; i++) {
   total_area += shapes[i].area();
   drawables[i].setPosition(i*10.0, i*10.0);
   drawables[i].draw(draw_window);
}
```


Çoklu interface

```
public class DrawableScalableRectangle
        extends DrawableRectangle
        implements Drawable, Scalable {
   // The methods of the Scalable interface must be implemented here.
}
```

interface ve Kalıtım

```
public interface Transformable
                extends Scalable, Rotateable, Reflectable
{
}
public interface DrawingObject extends Drawable, Transformable {
}
public class Shape implements DrawingObject
{ ...
}
```

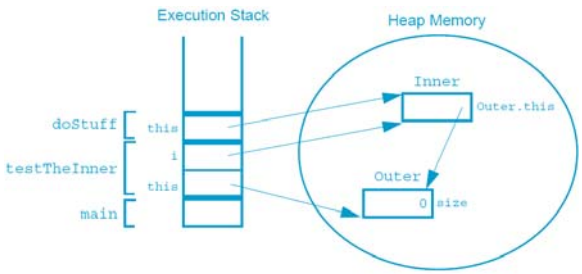

interface içinde sabit tanımlama

```
class A { static final int CONSTANT1 = 3; }
interface B { static final int CONSTANT2 = 4; }
class C implements B {
   void f() {
      int i = A.CONSTANT1;
      int j = CONSTANT2;
   }
}
```

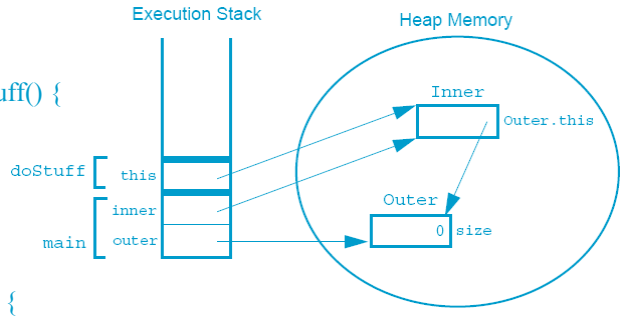
Inner Classes

- Were added to JDK 1.1
- Allow a class definition to be placed inside another class definition
- Group classes that logically belong together
- Have access to their enclosing class's scope

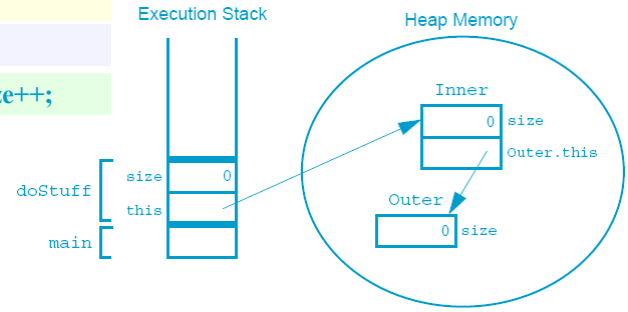

```
public class Outer1 {
    private int size ;
    /* Declare an inner class called "Inner" */
    public class Inner {
        public void doStuff() {
            // Inner class has access to 'size' from Outer
            size++;                                   Outer1.this.size++;
        }
    }
    public void testTheInner() {
      Inner i = new Inner() ;
      i.doStuf() ;
    }
}
```



```
public class Outer2 {
    private int size ;
    public class Inner {
        public void doStuff() {
            size++;
        }
    }
}
public void testTheInner() {
    public static void main(String[] args) {
      Outer2  outer = new Outer2() ;
      Outer2.Inner inner = outer.new Inner();
      inner.doStuff() ;
    }
}
```




```
public class Outer3 {
    private int size ;
    public class Inner {
        private int size ;
        public void doStuff(int size) {
            size++;
            this.size++;
            Outer3.this.size++;
        }
    }
}
```



```
1 public class Outer4 {
2     private int size = 5;
3
4     public Object makeTheInner(int localVar) {
5         final int finalLocalVar = 6;
6
7         // Declare a class within a method!?!
8         class Inner {
9             public String toString() {
10                return ("#<Inner size=" + size +
11                // " localVar=" + localVar + // ERROR: ILLEGAL
12                "finalLocalVar=" + finalLocalVar + ">");
13            }
14        }
15
16        return new Inner();
17 }
```


Inner Class Example

```
19    public static void main(String[] args) {
20        Outer4 outer = new Outer4();
21        Object obj = outer.makeTheInner(47);
22        System.out.println("The object is " + obj);
23    }
24 }
```

Properties of Inner Classes

►The class name can be used only within the defined scope, except when used in a qualified name. The name of the inner class must differ from the enclosing class.

►The inner class can be defined inside a method. Any variable, either a local variable or a formal parameter, can be accessed by methods within an inner class provided that the variable is marked as final.

Properties of Inner Classes

►The inner class can use both class and instance variables of enclosing classes and local variables of enclosing blocks.

►The inner class can be defined as abstract.

►Only inner classes can be declared as private or protected.

►An inner class can act as an interface implemented by another inner class.

Properties of Inner Classes

►Inner classes that are declared static automatically become top-level classes.

►Inner classes cannot declare any static members; only top-level classes can declare static members.

►An inner class wanting to use a static must declare static in the top-level class.

7# Advanced Class Features

►Exercise-2: "Working with Interfaces and Abstract Classes"

Sun microsystems
HANDS-ON LAB

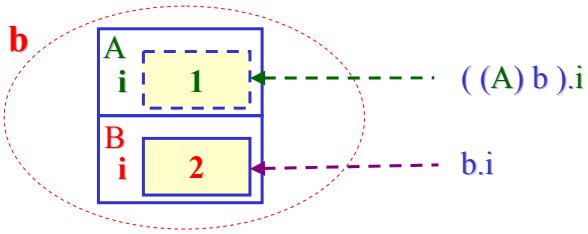
Example Revisited # 1

```
public class A {
  public int i = 1;
  public int f() { return i;}
}
public class B extends A {
 public int i = 2;
 public int f(){ return -i;}
}
```

```
public class override_test {
    public static void main(String args[]) {
        B b = new B();
        System.out.println(b.i);
        System.out.println(b.f());
        A a = (A) b;
        System.out.println(a.i);
        System.out.println(a.f());
    }
}
```

polymorphism

B b = new B() ;



Polymorphism does not work on data members but methods:

b.f()

((A)b).f()

Q: How can one access to B::i using a = (A)b ?

A: Use the operator : instanceof !

```
if (a instanceof B)
{
    B bb ;
    bb = (B) a ;
    System.out.println(b.i);
}
```



```
Employee e = new Manager() //legal

e.department = "Finance" //illegal
```

page.158

```
if (e instanceof Manager)
{
    Manager m ;
    m = (Manager) e ;
    m.department = "Finance" //legal
}
```

Example Revisited # 2

```
public class A{
    int x = 1 ;
    public int f() {return x ;}
}
public class B extends A {
    int x = 2 ;
    public int f() {return 2*x ;}
}
public class C extends B{
    int x = 3 ;
    public int f() {return ? ;}
}
```

- x
- this.x
- super.x
- ((B)this).x
- ((A)this).x
- super.super.x


```
public class polymorphism_test {
    public static void main(String args[]) {
        CC c = new CC();
        System.out.println(c.x);
        System.out.println(((BB)c).x);
        System.out.println(((AA)c).x);
        System.out.println(c.f());
        AA a = (AA) c;
        System.out.println(a.x);
        System.out.println(a.f());
        BB b = (BB) c;
        System.out.println(b.x);
        System.out.println(b.f());
        b = (BB) a ;
        System.out.println(b.x);
        System.out.println(b.f());
        Class cls = a.getClass();
        System.out.println(cls.getName()) ;
    }
}
```

8 EXCEPTIONS

Objectives

- Define exceptions
- Use `try`, `catch` and `finally` statements
- Describe exception categories
- Identify common exceptions
- Develop programs to handle your own exceptions

Exceptions

- The exception class defines mild error conditions that your program encounters.
- Exceptions can occur when
  - The file you try to open does not exist
  - The network connection is disrupted
  - Operands being manipulated are out of prescribed ranges
  - The class file you are interested in loading is missing
- An error class defines serious error conditions


```
1   public class HelloWorld {
2      public static void main (String args[]) {
3      int i = 0;
4
5      String greetings [] = {
6           "Hello world!",
7           "No, I mean it!",
8           "HELLO WORLD!!"
9       };
10
11     while (i < 4) {
12        System.out.println (greetings[i]);
13        i++;
14     }
15   }
16  }
```

```
Hello world!
No, I mean it!
HELLO WORLD!!
java.lang.ArrayIndexOutOfBoundsException
        at HelloWorld.main(HelloWorld.java:12)
Exception in thread "main" Process Exit...
```

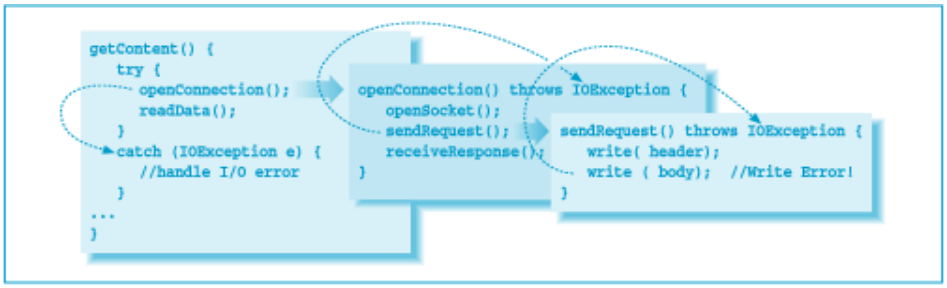

The try and catch Statements

```
try {
    // code that might throw a particular exception
}
catch (MyExceptionType e) {
    // code to execute if a MyExceptionType exception is thrown
}
catch (Exception e) {
    // code to execute if a general Exception exception is thrown
}
```

Call Stack Mechanism

►If an exception is not handled in the current try/catch block, it's thrown to the caller of that method.

►If the exception gets back to the main method and is not handled there, the program is terminated abnormally.



Call Stack Mechanism

►Consider a case where a method calls another method named openConnection(), and this, in turn calls another method named sendRequest(). If an exception occurs in sendRequest(), it is thrown back to openConnection(), where a check is made to see if there is a catch for that type of exception. if no catch exists in openConnection(), the next method in the call stack, main(), is checked. if the exception is not handled by the last method on the call stack, then a runtime error occurs and the program stops executing.

The `finally` Statement

What if we have some clean up to do before we exit our method from one of the catch clauses? To avoid duplicating the code in each catch branch and to make the cleanup more explicit, Java supplies the finally clause. A finally clause can be added after a try and any associated catch clauses. Any statements in the body of the finally clause are guaranteed to be executed, no matter why control leaves the try body:

next slide


```
try {
   // Do something here
}
catch ( FileNotFoundException e ){
   ...
}
catch ( IOException e ) {
   ...
}
catch ( Exception e ) {
   ...
}
finally {
   // Cleanup here
}
```

If the statements in the try execute cleanly, or even if we perform a return, break, or continue, the statements in the finally clause are executed. To perform cleanup operations, we can even use try and finally without any catch clauses:

```
try {
 // Do something here
 return;
}
finally {
 System.out.println("Do not ignore me!");
}
```



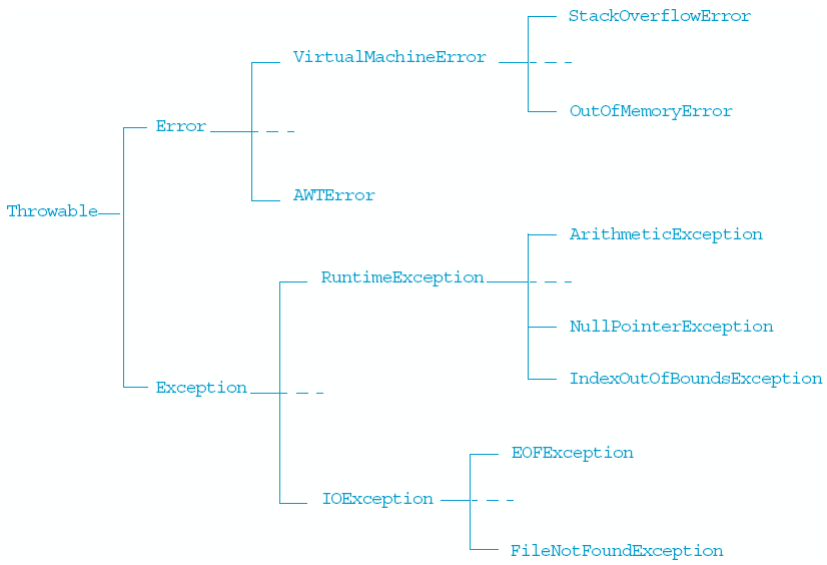
```
public class HelloWorldRevisited {
   public static void main (String args[]) {
                    int i = 0;
                    String greetings [] = {
                              "Hello world!",
                              "No, I mean it!",
                              "HELLO WORLD!!" };
     while (i < 4) {
           try {
              System.out.println (i+" "+greetings[i]);
           } catch (ArrayIndexOutOfBoundsException e){
             System.out.println("Re-setting Index Value");
             break;
           } finally {
             System.out.println("This is always printed");
           }
         i++;
     } // end while()
   } // end main()
}
```

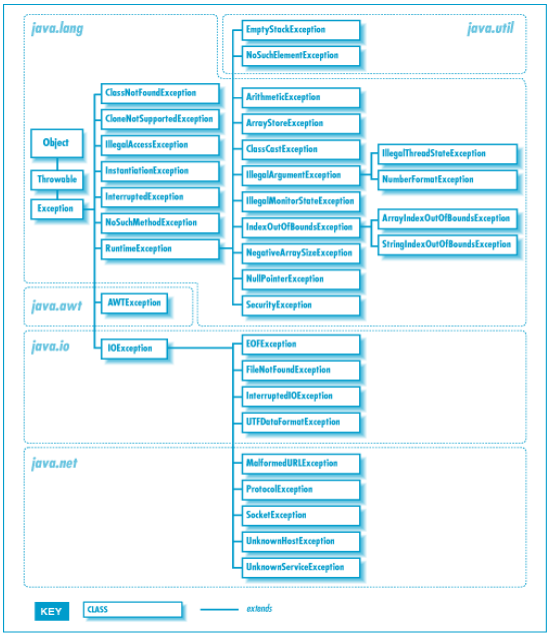
Exception Categories

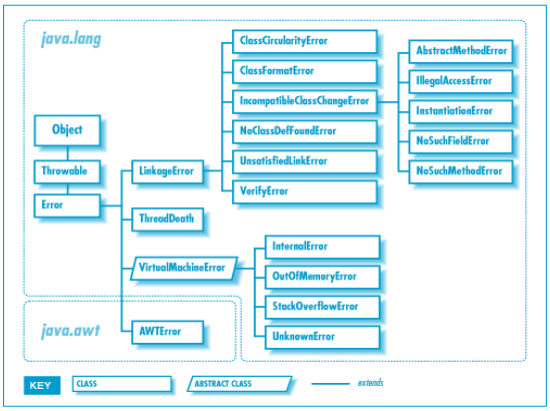
►`Error` indicates a severe problem from which recovery is difficult, if not impossible. An example is running out of memory. A program is not expected to handle such conditions.

►`RuntimeException` indicates a design or implementation problem. That is, it indicates conditions that should never happen if the program is operating properly.An `ArrayOutOfBoundsException` exception, for example, should never be thrown if the array indices do not extend past the array bounds. This would also apply, for example, to referencing a null object variable.

►Other exceptions indicate a difficulty at runtime that is usually caused by environmental effects and can be handled. Examples include a file not found or invalid URL exceptions. Because these usually occur as a result of user error, you are encouraged to handle them.




java.lang
java.util
java.awt
java.io
java.net
Object
Throwable
Exception
EmptyStackException
NoSuchElementException
ClassNotFoundException
CloneNotSupportedException
IllegalAccessException
InstantiationException
InterruptedException
NoSuchMethodException
RuntimeException
ArithmeticException
ArrayStoreException
ClassCastException
IllegalArgumentException
IllegalMonitorStateException
IndexOutOfBoundsException
NegativeArraySizeException
NullPointerException
SecurityException
IllegalThreadStateException
NumberFormatException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
AWTException
IOException
EOFException
FileNotFoundException
InterruptedIOException
UTFDataFormatException
MalformedURLException
ProtocolException
SocketException
UnknownHostException
UnknownServiceException
KEY
CLASS
extends


java.lang
java.awt
Object
Throwable
Error
LinkageError
ThreadDeath
VirtualMachineError
AWTError
ClassCircularityError
ClassFormatError
IncompatibleClassChangeError
NoClassDefFoundError
UnsatisfiedLinkError
VerifyError
AbstractMethodError
IllegalAccessError
InstantiationError
NoSuchFieldError
NoSuchMethodError
InternalError
OutOfMemoryError
StackOverflowError
UnknownError
KEY
CLASS
ABSTRACT CLASS
extends

Common Exceptions

- `ArithmeticException`

```
int i = 12 / 0 ;
```

- `NullPointerException`

```
Date d = null ;
System.out.println(d.toString()) ;
```

- `NegativeArraySizeException`
- `ArrayIndexOutOfBoundsException`
- `SecurityException`
  - Access a local file
  - Open a socket to the host that is not the same host that served the applet
  - Execute another program in runtime environment

The Handler or Declare Rule

- Handle exceptions by using the `try-catch-finally` block
- Declare that the code causes an exception by using the `throws` clause
- A method may declare that it `throws` more than one exception
- You do not need to handle or declare runtime exceptions or errors.

Method Overriding and Exceptions

The overriding method:

►Can throw exceptions that are subclasses of the exceptions being thrown by the overridden method

For example, if the superclass method throws an `IOException`, then the overriding method can throw an `IOException`, a `FileNotFoundException` (a subclass of `IOException`), but not an Exception (the superclass of `IOException`)


```
public class TestA {
   public void methodA() throws RuntimeException {
      // do some number crunching
   }
}
public class TestB1 extends TestA {
   public void methodA() throws ArithmeticException {
      // do some number crunching
   }
}
public class TestB2 extends TestA {
   public void methodA() throws Exception {
      // do some number crunching
   }
}
```

```
public class TestMultiA {
   public void methodA()
                        throws IOException, RuntimeException {
      // do some IO stuff
   }
}
public class TestMultiB1 extends TestMultiA {
   public void methodA()
    throws FileNotFoundException, UTFDataFormatException,
           ArithmeticException {
      // do some number crunching
   }
}
```



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

public class TestMultiB2 extends TestMultiA {
   public void methodA()
    throws FileNotFoundException, UTFDataFormatException,
           ArithmeticException, SQLException {
      // do some IO, number crunching, and SQL stuff
   }
}
```

```
public class ServerTimedOutException extends Exception {
        private int port;
        public ServerTimedOutException (String reason,int port){
            super(reason);
            this.port = port;
        }
        public int getPort() {
             return port;
        }
}
```

To throw an exception of the above type, write

```
throw new ServerTimedOutException("Could not connect",60) ;
```



```
public void connectMe(String serverName)
                              throws ServerTimedOutException  {
  int success;
  int portToConnect = 80 ;
  success = open(serverName,portToConnect) ;
  if( success == -1 )
    throw new ServerTimedOutException("Could not connect",
                                                  portToConnect);
}
public void findServer() {
      try {
          connectMe(defaultServer) ;
      }   catch (ServerTimedOutException e) {
            System.out.println("Server timed out, trying alternative") ;
            try{
                connectMe( alternativeServer) ;
            } catch (ServerTimedOutException e1) {
                System.out.println("Error : " + e1.getMessage() +
                                          "connecting to port" + e1.getPort()) ;
                }
          }
}
```

```
1  // Fig. 14.10: UsingExceptions.java
2  // Demonstrating the getMessage and printStackTrace
3  // methods inherited into all exception classes.
4  public class UsingExceptions {
5     public static void main( String args[] )
6     {
7        try {
8           method1();
9        }
10       catch ( Exception e ) {
11          System.err.println( e.getMessage() + "\n" );
12
13          e.printStackTrace();
14       }
15    }
16    public static void method1() throws Exception
17    {
18       method2();
19    }
20
21    public static void method2() throws E
22    {
23       method3();
24    }
25    public static void method3() throws E
26    {
27       throw new Exception( "Exception thrown in method3" );
28    }
29 }
```

Call **method1**, which calls **method2**, which calls **method3**, which throws an exception.

getMessage prints the **String** the **Exception** was initialized with.

printStackTrace prints the methods in this order:

method3
method2
method1
main

(order they were called when exception occurred)


```
Exception thrown in method3
java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:28)
        at UsingExceptions.method2(UsingExceptions.java:23)
        at UsingExceptions.method1(UsingExceptions.java:18)
        at UsingExceptions.main(UsingExceptions.java:8)
```

9 TEXT-BASED APPLICATIONS

Objectives

- Write a program that uses command-line arguments and system properties
- Write a program that reads from *standard input*
- Write a program that can create, read, and write files
- Write a program that uses sets and lists
- Write a program to iterate over a collection

Command-Line Arguments

- Any Java technology applications can use command-line arguments
- These string arguments are placed on the command lien to launch the Java interpreter, after the class name:

```
java TestArgs arg1 arg2 "another arg"
```

- Each command-line arguments is placed in the args array that is passed to the static `main` method:

```
public static void main(String[] args)
```



```
public class TestArgs {
   public static void main(String[] args) {
      for ( int i=0 ; i<args.length ; i++ ) {
          System.out.println("args[" + i +
                             "] is '" +
                             args[i] + "'") ;
      }
  }
}
```

java TestArgs arg1 arg2 "another arg"

System Properties

- System properties is a feature that replaces the concept of *environment variables* (which is platform-specific)
- The `System.getProperties` method returns a `Proeprties` object
  - System.getProperties()
  - System.getProperties(String)
  - System.getProperties(String,String)
- The `getProperty` method returns a `String` representing the value of the named property.
- Use the –D option to include a new property.

The Properties Class

- The `Properties` class implements a mapping of names to values (a `String` to `String` map).
- The `propertyNames` method returns an `Enumeration` of all property names.
- The `getProperty` method returns a `String` representing the value of the named property.
- You can also read and write a properties collection into a file using `load` and `store`.

```
import java.util.Properties ;
import java.util.Enumeration ;
public class TestProperties {
   public static void main(String[] args) {
    Properties props = System.getProperties() ;
    Enumeration prop_names = props.propertyNames();
    while( prop_names.hasMoreElements() ){
      String prop_name=(String)prop_names.nextElement() ;
      String property = props.getProperty(prop_name) ;
      System.out.println("property '" + prop_name +
                              "' is '" + property + "'") ;
      }
  }
}
```



```
property 'java.runtime.name' is 'Java(TM) 2 Runtime Environment, Standard Edition'
property 'sun.boot.library.path' is 'F:\jdk1.3\jre\bin'
property 'java.vm.version' is '1.3.0-C'
property 'java.vm.vendor' is 'Sun Microsystems Inc.'
property 'java.vendor.url' is 'http://java.sun.com/'
property 'path.separator' is ';'
property 'java.vm.name' is 'Java HotSpot(TM) Client VM'
property 'file.encoding.pkg' is 'sun.io'
property 'java.vm.specification.name' is 'Java Virtual Machine Specification'
property 'user.dir' is 'C:\'
property 'java.runtime.version' is '1.3.0-C'
property 'java.awt.graphicsenv' is 'sun.awt.Win32GraphicsEnvironment'
property 'os.arch' is 'x86'
property 'java.io.tmpdir' is 'C:\DOCUME~1\kurt\LOCALS~1\Temp\'
property 'line.separator' is ' '
property 'java.vm.specification.vendor' is 'Sun Microsystems Inc.'
property 'java.awt.fonts' is ''
property 'os.name' is 'Windows 2000'
```

```
String osName= (System.getProperties()).getProperty("os.name");

switch ( osName.hashCode() ) {
   case -113889189 :
        System.out.println(“Program has not been tested on this os!") ;
    break;
   default:
          System.out.println("Ok.") ;
}
```


Console I/O

- `System.out` allows you to write to “standard output”.
  - It is an object of type `PrintStream`.
- `System.in` allows you to read from “standard input”.
  - It is an object of type `InputStream`.
- `System.err` allows you to write to “standard error”
  - It is an object of type `PrintStream`.

Writing to Standard Output

- The `println` methods print the argument and a newline '\n'.
- The `print` methods print the argument without a newline
- The `print` and `println` methods are overloaded for most primitive types (`boolean`, `char`, `int`, `long`, `float`, and `double`) and for `char[]`, `Object`, and `String`.
- The `print(Object)` and `println(Object)` methods call the `toString()` method on the argument.

Reading From Standard Input

```
import java.io.* ;
public class KeyboardInput {
    public static void main(String[] args) throws IOException {
      String s                                          ;
      InputStreamReader ir = new InputStreamReader(System.in) ;
      BufferedReader in = new BufferedReader(ir)                ;
      System.out.println("Enter an integer : ")  ;
     try{

      while( (s = in.readLine()) != null )  )
            System.out.println("Read: "+ s) ;

         in.close();
      }
      catch( IOException e) {
           e.printStackTrace();
      }
    }
}
```

Files and File I/O

- The `java.io` package
- Creating `File` Objects
- Manipulating `File` Objects
- Reading and writing to file streams

Creating a New File Object

- `File myFile ;`
- `myFile = new File("myfile.txt") ;`
- `myFile = new File("MyDocs","myfile.txt") ;`
- Directories are treated just like files in Java; the File class supports methods for retrieving an array of files in the directory
- ```
  File myDir = new File("MyDocs") ;
  myFile = new File(myDir,"myfile.txt") ;
  ```

The class File defines platform-independent methods for manipulating a file maintained by a native file system. However, it does not allow you to access the contents of the file.
  ```

File Tests and Utilities

- `File` names :
  - String getName()
  - String getPath()
  - String getAbsolutePath()
  - String goParent()
  - boolean renameTo(File newName)
- `File` tests :
  - boolean exists()
  - boolean canWrite()
  - boolean canRead()
  - boolean isFile()
  - boolean isDirectory()

- `File` information
  - long lastModified()
  - long length()
  - boolean delete()
- `File` utilities :
  - boolean mkdir()
  - String[] list()

File Stream I/O

- File input:
  - Use the `FileReader`. class to read characters
  - Use the `BufferedReader` class to use the `readLine` method
- File output:
  - Use the `FileWriter` class to write characters
  - Use the `PrintWriter` class to use the `print` and `println` methods


```
import java.io.* ;
public class ReadFile {
    public static void main(String[] args) {
      File file = new File(args[0])                                    ;
      try {
         BufferedReader in = new BufferedReader(new FileReader(file)) ;
         String s ;
         s = in.readLine() ;
         while( s != null ) {
               System.out.println("Read: " + s ) ;
               s = in.readLine() ;
         }
         in.close() ;
      } catch ( FileNotFoundException e1) {
          System.err.println("File not found: " + file);
      } catch ( IOException e2 ) {
          e2.printStackTrace() ;
      }
    }
}
```

```
import java.io.* ;
public class WriteFile {
    public static void main(String[] args) {
      File file = new File(args[0])                                    ;
      try {
        BufferedReader in = new
                    BufferedReader(new InputStreamReader(System.in)) ;
        PrintWriter out = new PrintWriter(new FileWriter(file)) ;
        String s ;
        System.out.print("Enter file text.   ") ;
        System.out.println("[Type ctrl-d (or ctrl-z) to stop.]") ;
        while( (s=in.readLine()) != null ) {
            out.println(s) ;
        }
        in.close() ;
        out.close();
      } catch ( IOException e) {
        e.printStackTrace() ;
      }
    }
}
```


The Math Class

- The `Math` class contains a group of static math functions
  - Truncation: `ceil`, `floor`, and `round`
  - Variations on max, min, and abs
  - Trigonometry: `sin`, `cos`, `tan`, `asin`, `atan`, `toDegrees`, and `toRadians`
  - Logarithms: `log` and `exp`
  - Others: `sqrt`, `pow`, and `random`
  - Canstants: `PI` and `E`

The String Class

- `String` objects are immutable sequences of Unicode characters.
- Operations that create new strings: `concat`, `replace`, `substring`, `toLowerCase`, `toUpperCase`, and `trim`
- Search operations: `endsWith`, `startWith`, `indexOf`, and `lastIndexOf`.
- Comparisons: `equals`, `equalsIgnoreCase`, and `compareTo`
- Others: `charAt`, and `length`

Methods That Create New Strings

- `String concat(String s)` – returns a new string consisting of `this` string followed by the s string
- `String replace(char old, char new)` – returns a new string that is copy of `this` string with the `new` string replacing all occurences of the `old` string
- `String substring(int start,int end)` – returns a portion of `this` string starting at the `start` index and ending at `end`.
- `String toLowerCase()` – returns a new string consisting of `this` string converted to lowercase
- `String toUpperCase()` – returns a new string consisting of `this` string converted to uppercase

Search Methods

- `boolean endsWith(String s)` – returns `true` if `this` string ends with s
- `boolean startsWith(String s)` – returns `true` if `this` string starts with s
- `int indexOf(String s)` – returns index within `this` string that starts with s
- `int indexOf(int ch)` – returns index within `this` string of the first occurence of the character ch
- `int indexOf(String s, int offset)` – returns index within `this` string that matches with s starting at `offset`
- `int indexOf(int ch, int offset)` – returns index within `this` string of the first occurence of the character ch starting at `offset`

Comparison Methods

- `boolean equals(String s)` – returns `true` if `this` string is equal to string s
- `boolean equalsIgnoreCase(String s)` – returns `true` if `this` string is equal to (ignoring case) the string s
- `int compareTo(String s)` – performs a lexical comparison between `this` string and s; returns a negative `int` if `this` string is less than s, a positive `int` if `this` string is greater than s, or 0 if the two strings are equal

The StringBuffer Class

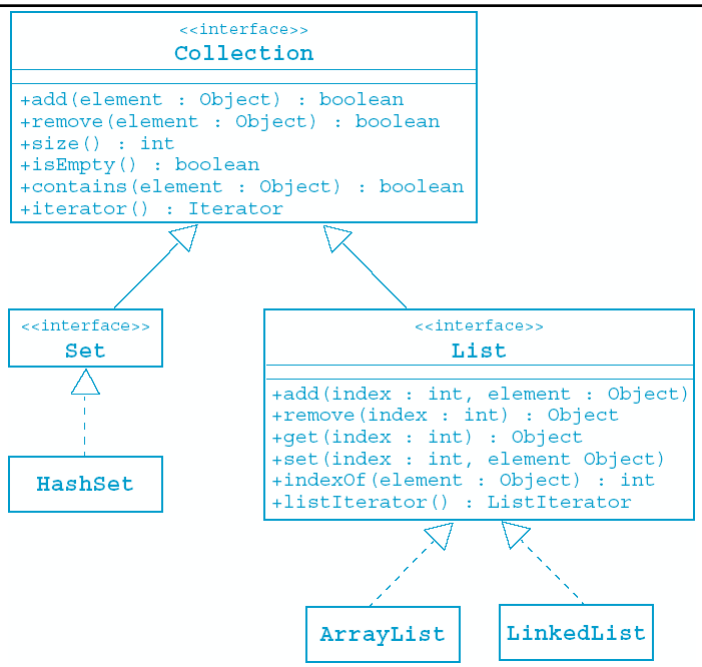
- `StringBuffer` objects are mutable sequences of Unicode characters
- Constructors:
  - `StringBuffer()` – creates an empty buffer
  - `StringBuffer(int capacity)` – creates an empty buffer with a specified initial capacity
  - `StringBuffer(String initialString)` – creates a buffer that initially contains the specified string
- Modification operations: `append`, `insert`, `reverse`, `setCharAt`, and `setLength`

Modifications Methods

- `StringBuffer append(String s)` – Modifies `this` string buffer by appending the s string onto the end of the buffer
- `StringBuffer insert(int offset, String s)` – Modifies `this` string buffer by inserting the s string into the buffer at the specified offset location.
- `StringBuffer reverse()` – Reverses the order of the string buffer
- `void setCharAt(int index, char ch)` – Modifies `this` string buffer by changing the character at the location specified by `index` to the specified character, `ch`.
- `void setLength(int newLength)`

The Collections API

- A *collection* is a single object representing a group of objects known as its elements
- Collection API contains interfaces that group objects as a:
  - `Collection` – A group of objects called elements; any specific ordering and allowance of duplicates is specified by each implementation
  - `Set` – An unordered collection; no duplicates are permitted
  - `List` – An ordered collection; duplicates are permitted



Set Example

```
import java.util.* ;
public class SetExample {
   public static void main(String[] args) {
      Set set = new HashSet();
      set.add("one") ;
      set.add("second") ;
      set.add("3rd") ;
      set.add(new Integer(4)) ;
      set.add(new Float(5.0F)) ;
      set.add("second") ;
      set.add(new Integer(4)) ;
      System.out.println(set) ;
   }
}
```


List Example

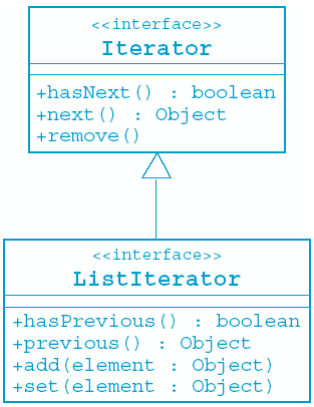
```
import java.util.* ;
public class ListExample {
   public static void main(String[] args) {
      List list = new ArrayList();
      list.add("one") ;
      list.add("second") ;
      list.add("3rd") ;
      list.add(new Integer(4)) ;
      list.add(new Float(5.0F)) ;
      list.add("second") ;
      list.add(new Integer(4)) ;
      System.out.println(list) ;
   }
}
```

Iterators

- Iteration is the process of retrieving every element in a collection
- An `Iterator` of a `Set` is unordered
- A `ListIterator` of a `List` can be scanned forwards (using the `next` method) or backwards (using the `previous` method):

```
List list = new ArrayList() ;
Iterator elements = list.iterator();
while( elements.hasNext() )
   System.out.println( elements.next() )
```


The Iterator Interface Hierarchy



Collections in JDK 1.1

- `Vector` implements the `List` interface
- `Stack` is a subclass of `Vector` and supports the `push`, `pop`, and `peek` methods
- `Hashtable` implements the `Map` interface
- `Enumeration` is a variation on the `Iterator` interface:
  - An enumeration is returned by the `elements` method in `Vector`, `Stack`, and `Hashtable`

10 Building Java GUIs

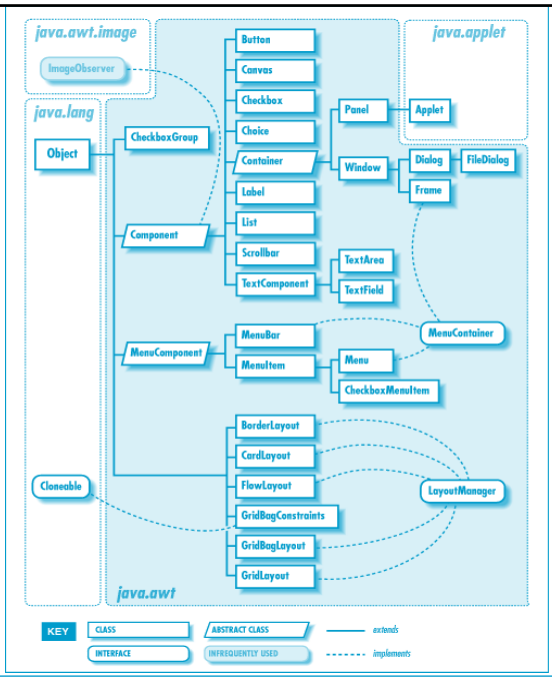
Objectives

- Describe the AWT package and its components
- Define the terms *containers*, *components* and *layout managers*, and how they work together to build a graphical user interface (GUI)
- Use layout managers
- Use the flow, border, grid, and card layout managers to achieve a desired dynamic layout
- Add components to a container
- Use the frame and panel containers appropriately

Abstract Window Toolkit (AWT)

- Provides basic GUI components which are used in all Java applets and applications
- Contains super classes which can be extended and their properties inherited; classes can also be abstract
- Ensures that every GUI component that is displayed on the screen is a subclass of the abstract class `Component` or `MenuComponent`
- Contains `Container` which is an abstract subclass of `Component` and which includes two subclasses:
  - `Panel`
  - `Window`

User-interface classes of AWT Package



Containers

- Add components with the `add()` method
- The two main types of containers are `Window` and `Panel`
- A `Window` is a free floating window on the display. There are two important types of Window:
  - `Frame` – window with a title and corners you can resize
  - `Dialog` – cannot have a menu bar, you cannot resize
- A `Panel` is a container of GUI components that must exist in the context of some other container, such as a window or applet

Positioning Components

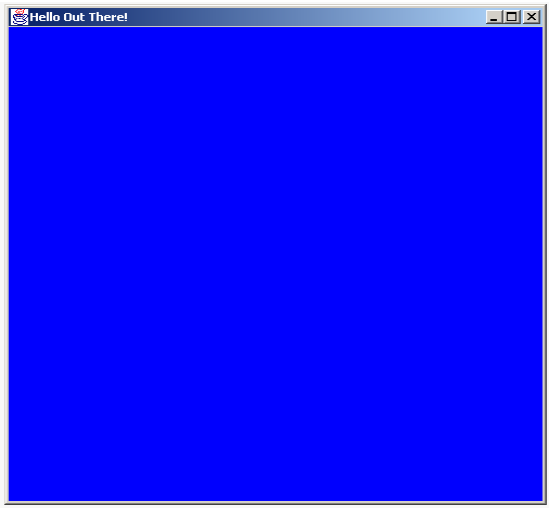
- The position and size of a component in a container is determined by a layout manager.
- You can control the size or position of components by disabling the layout manager.

You must then use `setLocation()`, `setSize()`, or `setBounds()` on components to locate them in the container.

Frames

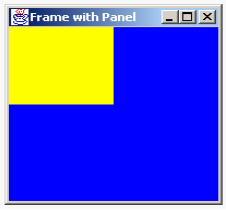
- Are subclasses of `Window`
- Have title and resize corners
- Inherit from Component and add components with the *add* method
- Are initially invisible, use `setVisible(true)` to expose the frame
- Have `BorderLayout` as the default layout manager
- Use the `setLayout` method to change the default layout manager

```
import java.awt.*;
    public class MyFrame extends Frame {
    public static void main (String args[]) {
            MyFrame fr = new MyFrame("Hello Out There!");
            // Component method setSize()
            fr.setSize(500,500);
            fr.setBackground(Color.blue);
            fr.setVisible(true); // Component method show()
    }
    public MyFrame (String str) {
            super(str);
    }
}
```

Panels

- Provide a space for components
- Allow subpanels to have their own layout manager
- Add components with the add method




```
public class FrameWithPanel extends Frame {
        public FrameWithPanel (String str) {
                super (str);
        }
        public static void main (String args[]) {
          FrameWithPanel fr =
              new FrameWithPanel ("Frame with Panel");
        Panel pan = new Panel();
        fr.setSize(200,200);
        fr.setBackground(Color.blue);
         fr.setLayout(null); //override default layout manager
        pan.setSize (100,100);
        pan.setBackground(Color.yellow);
        fr.add(pan);
        fr.setVisible(true);
    }
}
```

Container Layout

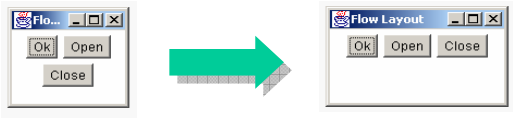
- `FlowLayout`–The default layout manager of Panel and Applet
- `BorderLayout`–The default manager of `Window`, `Dialog`, and `Frame`
- `GridLayout`–A layout manager that provides flexibility for placing components
- `CardLayout`
- `GridBagLayout`

The `FlowLayout` Manager

- Default layout manager for the `Panel` class
- Components are added from left to right
- Default alignment is centered
- Uses components' preferred sizes
- Uses the constructor to tune behavior

```
public class MyFlow {
        private Frame f;
        private Button button1, button2, button3;

        public static void main (String args[]) {
                MyFlow mflow = new MyFlow ();
                mflow.go();
        }
        public void go() {
           f = new Frame ("Flow Layout");
           f.setLayout(new FlowLayout());
           button1 = new Button("Ok");
           button2 = new Button("Open");
           button3 = new Button("Close");
           f.add(button1);
           f.add(button2);
           f.add(button3);
           f.setSize (100,100);
           f.setVisible(true);
        }
}
```

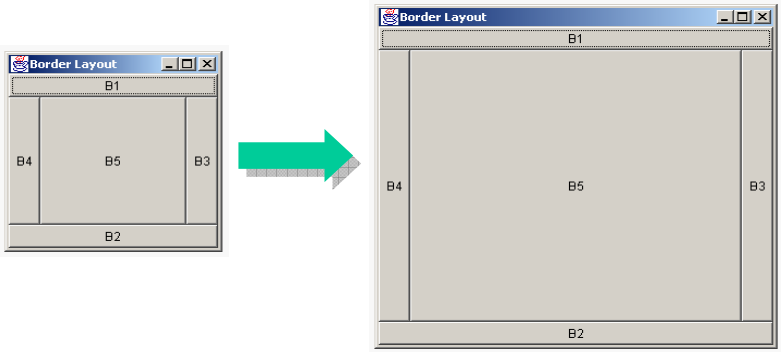


The BorderLayout Manager

- Default layout manager for the `Frame` class
- Components are added to specific regions
- The resizing behavior:
  - North, South, and Center regions adjust horizontally
  - East, West, and Center regions adjust vertically

```
public class BorderExample {
        private Frame f;
        private Button bn, bs, bw, be, bc;
        public static void main(String args[]) {
            BorderExample guiWindow2 = new BorderExample();
            guiWindow2.go();
        }
        public void go() {
            f = new Frame("Border Layout");
            bn  =  new Button("B1");
            bs  =  new Button("B2");
            be  =  new Button("B3");
            bw =  new Button("B4");
            bc  =  new Button("B5");

            f.add(bn, BorderLayout.NORTH);
            f.add(bs, BorderLayout.SOUTH);
            f.add(be, BorderLayout.EAST);
            f.add(bw, BorderLayout.WEST);
            f.add(bc, BorderLayout.CENTER);
            f.setSize (200, 200);
            f.setVisible(true);
          }
}
```

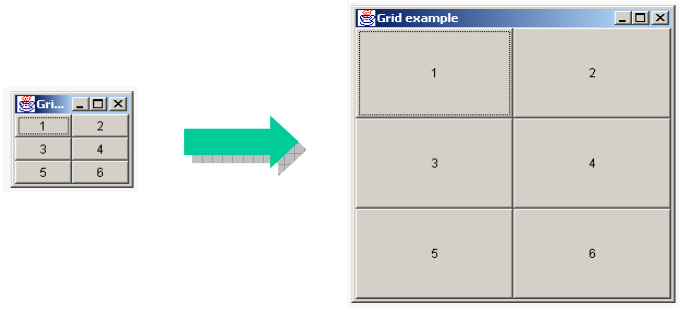



The GridLayout Manager

- Components are added left to right, top to bottom.
- All regions are equally sized.
- The constructor specifies the rows and columns.


```
public class GridExample {
        private Frame f;
        private Button b1, b2, b3, b4, b5, b6;
        public static void main(String args[]) {
                GridExample grid = new GridExample();
                grid.go();
        }
        public void go() {
                f = new Frame("Grid example");
                f.setLayout (new GridLayout (3, 2));
```

```
                b1 = new Button("1");
                b2 = new Button("2");
                b3 = new Button("3");
                b4 = new Button("4");
                b5 = new Button("5");
                b6 = new Button("6");
                f.add(b1);
                f.add(b2);
                f.add(b3);
                f.add(b4);
                f.add(b5);
                f.add(b6);
                f.pack();
                f.setVisible(true);
            }
        }
```

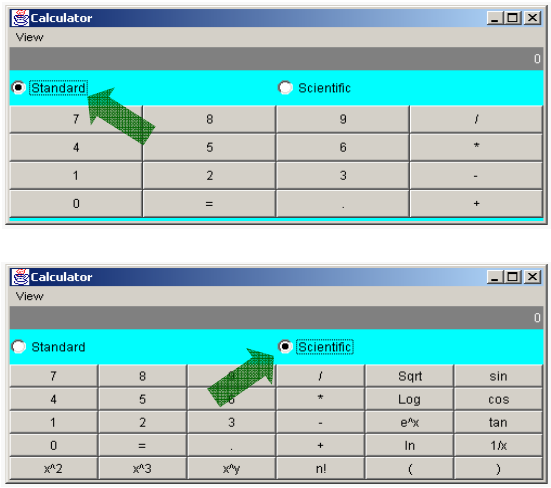
The CardLayout Manager

►The `CardLayout` manager arranges components into a "deck" of cards where only the top card is visible.

►Each card is usually a container such as a panel and each card can use any layout manager


```
 // add deck
deck = new Panel() ;
cardManager = new CardLayout() ;
deck.setLayout(cardManager) ;
      // add Card 1
     standardCalculator = new Panel() ;
     standardCalculator.setLayout(new GridLayout(4,4)) ;
     standardButtons = new Button[16] ;
      for(int i=0;i<16;i++)  {
         standardButtons[i] = new Button(standardString[i]) ;
         standardButtons[i].addActionListener(this) ;
         standardCalculator.add(standardButtons[i])   ;
     }
     deck.add(standardCalculator,"standard") ;
```

```
// add Card 2
    scientificCalculator = new Panel() ;
    scientificCalculator.setLayout(new GridLayout(5,6)) ;
    scientificButtons = new Button[30] ;
    for(int i=0;i<30;i++) {
        scientificButtons[i] = new Button(scientificString[i]) ;
        scientificButtons[i].addActionListener(this) ;
        scientificCalculator.add(scientificButtons[i])   ;
     }
    deck.add(scientificCalculator,"scientific")        ;
```

The GridBagLayout Manager

- A Layout manager similar to `GridLayout`.
- Unlike `GridLayout` each component size can vary and components can be added in any order

11 GUI Event Handling

Objectives

- Write code to handle events that occur in a GUI
- Describe the concept of adapter classes, including how and when to use them
- Determine the user action that originated the event from the event object details
- Create the appropriate interface and event handler methods for a variety of event types

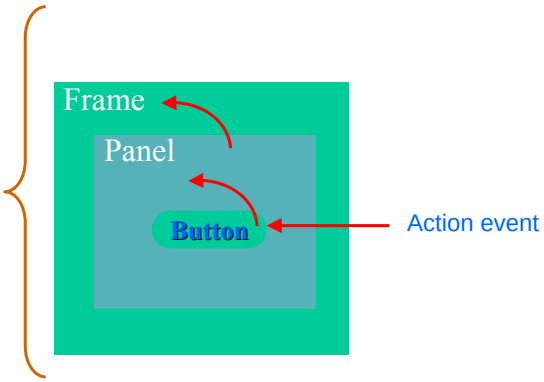
What is an Event?

- Events - Objects that describe what happened
- Event sources - The generator of an event
- Event handlers - A method that receives an event object, deciphers it, and processes the user's interaction

Hierarchical Model (JDK1.0)

- Is based on containment

action()
lostFocus()
mouseExit()
gotFocus()
mouseDown()
mouseMove()
keyDown()
mouseDrag()
mouseUp()
keyUp()
mouseEnter()



- Advantages
  - It is simple and well suited to an object-oriented programming environment.
- Disadvantages
  - An event can only be handled by the component from which it originated or by one of the containers of the originating component.
  - In order to handle events, you must either subclass the component that receives the event or create a handleEvent() method at the base container.

Delegation Model (JDK1.1)

► Events are sent to the component from which the event originated, but it is up to each component to propagate the event to one or more registered classes called listener. Listeners contain event handlers that receive and process the event. In this way, the event handler can be in an object separate from the component. Listeners are classes that implement the EventListener interface.

► Events are objects that are reported only to registered listeners. Every event has corresponding listener interface that mandates which methods must be defined in a class suited to receiving that type of event. The class that implements the interface defines those methods, and can be registered as a listener.

► Events from components that have no registered listeners are not propagated.

Delegation Model

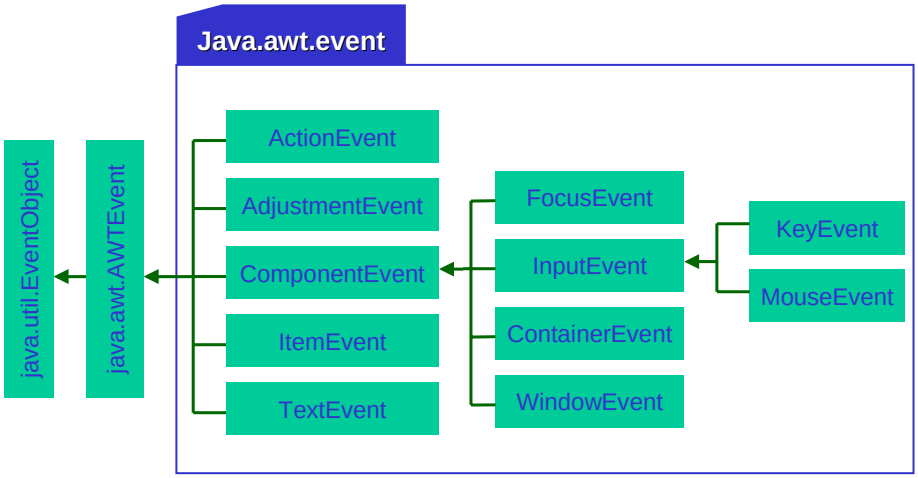
►Client objects (handlers) register with a GUI component they want to observe.

►GUI components only trigger the handlers for the type of event that has occurred

►Most components can trigger more than one type of event

►Distributes the work among multiple classes

Event Categories



Event Class	Listener Interface	Listener Methods
ActionEvent	ActionListener	actionPerformed()
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged()
ComponentEvent	ComponentListener	componentHidden()
		componentMoved()
		componentResized()
		componentShown()
ContainerEvent	ContainerListener	componentAdded()
		componentRemoved()
FocusEvent	FocusListener	focusGained()
		focusLost()
ItemEvent	ItemListener	itemStateChanged()
KeyEvent	KeyListener	keyPressed()
		keyReleased()
		keyTyped()

MouseEvent	MouseListener	mouseClicked()
		mouseEntered()
		mouseExited()
		mousePressed()
		mouseReleased()
MouseMotionEvent	MouseMotionListener	mouseDragged()
		mouseMoved()
TextEvent	TextListener	textValueChanged()
WindowEvent	WindowListener	windowActivated()
		windowClosed()
		windowClosing()
		windowDeactivated()
		windowDeiconified()
		windowIconified()
		windowOpened()

Component	Events Generated	Meaning
Button	ActionEvent	User clicked on the button
Checkbox	ItemEvent	User selected or deselected an item
CheckboxMenuItem	ItemEvent	User selected or deselected an item
Choice	ItemEvent	User selected or deselected an item
Component	ComponentEvent	Component moved, resized, hidden, or shown
	FocusEvent	Component gained or lost focus
	KeyEvent	User pressed or released a key
	MouseEvent	User pressed or released mouse button, mouse entered or exited component, or user moved or dragged mouse. Note: MouseEvent has two corresponding listeners, MouseListener and MouseMotion Listener.

Container	ContainerEvent	Component added to or removed from container
List	ActionEvent	User double-clicked on list item
	ItemEvent	User selected or deselected an item
MenuItem	ActionEvent	User selected a menu item
Scrollbar	AdjustmentEvent	User moved the scrollbar
TextComponent	TextEvent	User changed text
TextField	ActionEvent	User finished editing text
Window	WindowEvent	Window opened, closed, iconified, deiconified, or close requested

Example

```
import java.awt.* ;
import java.awt.event.* ;
public class TwoListener implements MouseMotionListener,
MouseListener
{
    private Frame f   ;
    private TextField tf;

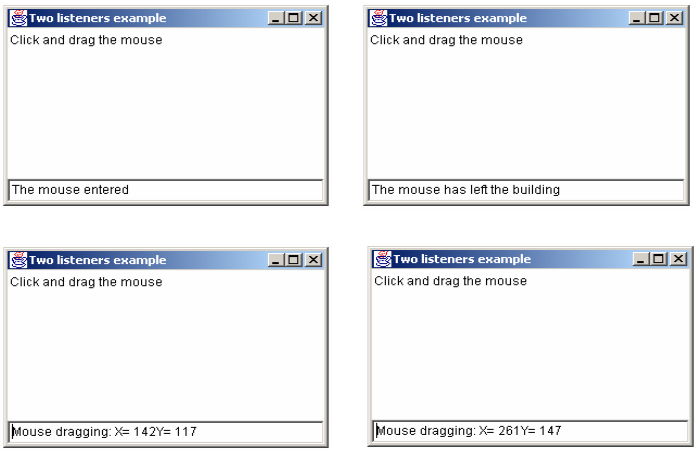
     public TwoListener() {
          f = new Frame("Two listeners example") ;
          tf = new TextField(30) ;
     }
```

```
public void launchFrame()  {
        Label label = new Label("Click and drag the mouse") ;
        f.add(label, BorderLayout.NORTH) ;
        f.add(tf, BorderLayout.SOUTH) ;
        f.addMouseMotionListener(this) ;
        f.addMouseListener(this) ;
        f.setSize(300,200) ;
        f.setVisible(true) ;
    }
 // These are MouseMotionListener events
 public void mouseDragged( MouseEvent e)   {
        String s = "Mouse dragging: X= " + e.getX() + "Y= " + e.getY() ;
        tf.setText(s) ;
  }
```



```
    public void mouseEntered( MouseEvent e)  {
        String s = "The mouse entered" ;
        tf.setText(s) ;
    }
    public void mouseExited( MouseEvent e)  {
        String s = "The mouse has left the building" ;
        tf.setText(s) ;
    }
    // Unused MouseMotionListener method
    // All methods of a listener must be present in the
    // class even if they are not used
    public void mouseMoved(MouseEvent e) { }
    // Unused MouseListener methods
    public void mousePressed(MouseEvent e) { }
    public void mouseClicked(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
```

```
    public static void main(String[] args)          {
            TwoListener two = new TwoListener() ;
          two.launchFrame();
     }
}
```

Multiple Listeners

- Multiple listeners cause unrelated parts of a program to react to the same event.
- The handlers of all registered listeners are called when the event occurs

Event Adapters

- The listener classes that you define can extend adapter classes and override only the methods that you need.
- Example:

```
import java.awt.* ;
import java.awt.event.* ;
public class MouseClickHandler extends MouseAdapter {
        public void mouseClicked( MouseEvent e) {
          // do stuff with the mouse click...
        }
}
```

Event Handling Using Anonymous Classes

- You can include an entire class definition within the scope of an expression.
- This approach defines what is called an anonymous inner class and creates the instance all at once.
- For example:

↳next slide


```
import java.awt.* ;
import java.awt.event.* ;

public class TestAnonymous {
    private Frame f  ;
    private TextField tf;

    public TestAnonymous() {
            f = new Frame("Anonymous class example") ;
            tf = new TextField(30) ;
    }
    public void launchFrame() {
        Label label = new Label("Click and drag the mouse") ;
        f.add(label, BorderLayout.NORTH) ;
        f.add(tf, BorderLayout.SOUTH) ;
        f.addMouseMotionListener(new MouseMotionAdapter()
        {
            public void mouseDragged( MouseEvent e){
                String s = "Mouse dragging: X= " + e.getX() +
                        "Y= " + e.getY() ;
                tf.setText(s) ;
}
        }) ;
```

```
            f.addMouseListener( new MouseClickHandler(tf) ) ;
            f.setSize(300,200) ;
            f.setVisible(true) ;
        }

        public static void main(String[] args)
        {
            TestAnonymous obj = new TestAnonymous() ;
            obj.launchFrame();
        }
}
```



```
import java.awt.* ;
import java.awt.event.* ;
public class MouseClickHandler extends MouseAdapter {
            private TextField tf ;
            public static int count = 0 ;
            public MouseClickHandler(TextField tf)  {
                        this.tf = tf ;
            }
            public void mouseClicked( MouseEvent e) {
                count++;
                String s = "Mouse has been clicked " + count + " times so far." ;
tf.setText(s) ;
    }
}
```

Event Handling Using Inner Classes

- You can implement event handlers as inner class.
- This allows access to the private data of the outer class.
- For example:

↳next slide


```
import java.awt.* ;
import java.awt.event.* ;

public class TestInner {
      private Frame f   ;
      private TextField tf;

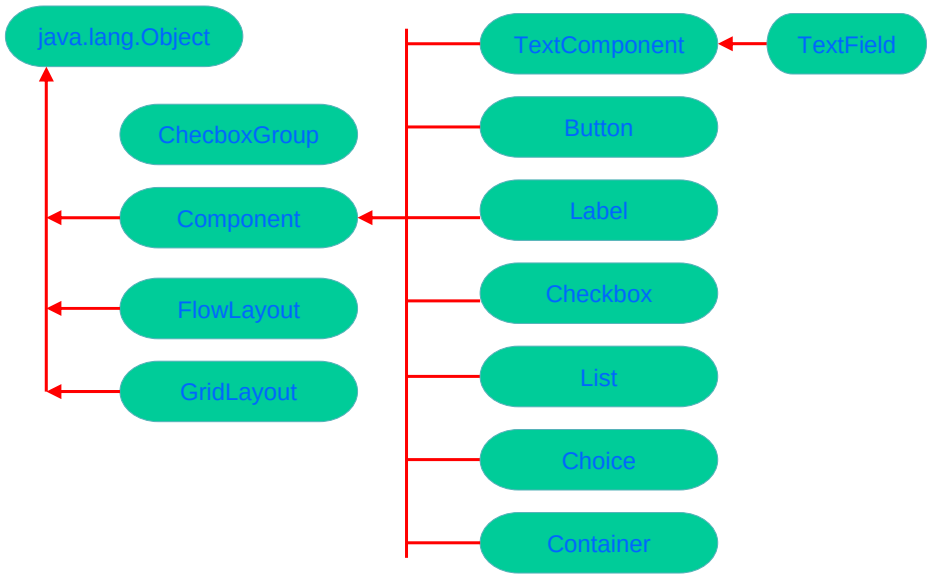
      public TestInner() {
            f = new Frame("Inner classes example") ;
            tf = new TextField(30) ;
      }
      public void launchFrame() {
            Label label = new Label("Click and drag the mouse") ;
            f.add(label, BorderLayout.NORTH) ;
            f.add(tf, BorderLayout.SOUTH) ;
            f.addMouseMotionListener(new MyMouseMotionListener()) ;
            f.addMouseListener(new MouseClickHandler(tf)) ;
            f.setSize(300,200) ;
            f.setVisible(true) ;
       }
```

```
class MyMouseMotionListener extends MouseMotionAdapter  {
        public void mouseDragged( MouseEvent e ) {
                String s = "Mouse dragging: X= " + e.getX() +
                        "Y= " + e.getY() ;
                tf.setText(s) ;
         }
         public static void main(String[] args)   {
            TestInner obj = new TestInner() ;
            obj.launchFrame();
         }
}
```


12 GUI-Based Applications

GUI-Based Applications

- Identify key AWT components
- Use AWT components to build user interfaces for real programs
- Control the colors and fonts used by an AWT component
- Use the Java printing mechanism



Label

- public Label()

  constructs an empty Label- text is not displayed

- public Label(String s)

  Constructs a Label that displays the text s with default left-justified alignment

- public Label(String s, int alignment)

  Label.LEFT, Label.CENTER, Label.RIGHT

- public String getText()
- public void setText(String s)
- public void setAlignment(int alignment)

TextField

- public TextField()

  constructs a TextField object

- public TextField(int columns)

  constructs an empty TextField object with specified number of columns

- public TextField(String s, int columns)
- public void setEchoChar(char c)
- public void setEditable(boolean b) // true == editable
- public String getText()
- public void setText(String s)

```
import java.awt.* ;
import java.awt.event.* ;

public class TextFieldApp implements ActionListener {
        private Frame myFrame       ;
        private Label myLabel       ;
        private Panel myPanel       ;
        private TextField password  ;
        private TextField tf                ;

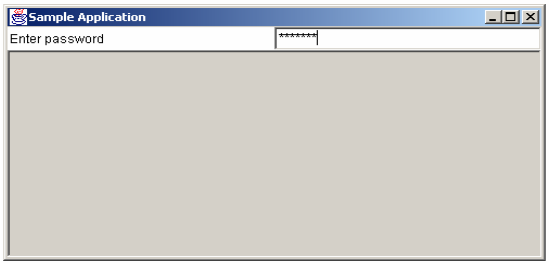
        public TextFieldApp ()          {
                myFrame = new Frame("Sample Application") ;
                myPanel = new Panel() ;
                // Label
                myLabel = new Label("Enter password") ;
                // TextField
                password = new TextField("") ;
                password.setEchoChar('*') ;
                password.addActionListener(this) ;
```

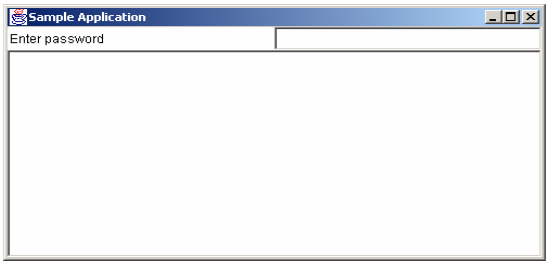


```
                // Panel
                myPanel.setLayout(new GridLayout(1,2)) ;
                myPanel.add(myLabel) ;
                myPanel.add(password) ;
                // Text Field
                tf = new TextField() ;
                tf.setEditable(false) ;
                myFrame.add(myPanel,BorderLayout.NORTH) ;
                myFrame.add(tf,BorderLayout.CENTER);
                // setSize and setVisible
                myFrame.setSize(500,256) ;
                myFrame.setVisible(true) ;
        }
        public void actionPerformed(ActionEvent e)    {
                String s ;
                password.setText("");
                s = e.getActionCommand() ;
                if(s.compareTo("keyword") == 0) tf.setEditable(true)    ;
                else  tf.setEditable(false)           ;

        }
```

```
public static void main(String[] args) {
                TextFieldApp tfa = new TextFieldApp() ;
        }
}
```



Choice

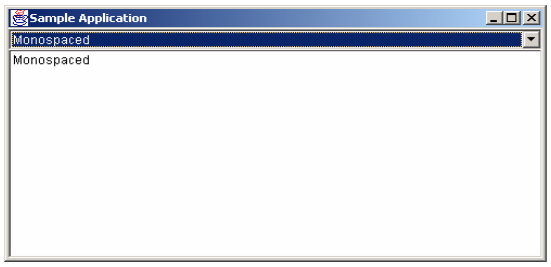
- public String getItem(int index)
- public synchronized void add(String s)
- public synchronized String getSelectedItem()
- public int getSelectedIndex()
- public synchronized String insert(String s, int index)
- public synchronized void remove(String s)

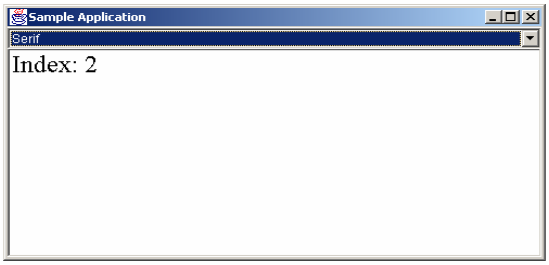

```
import java.awt.* ;
import java.awt.event.* ;
public class ChoiceApp implements ItemListener {
        private Frame myFrame      ;
        private TextField tf                    ;
        private Choice fonts       ;
         static int size = 10 ;
        public ChoiceApp () {
             myFrame = new Frame("Sample Application") ;
             // Choice
            fonts = new Choice() ;
            fonts.add( "Monospaced") ; // Courier
            fonts.add( "SansSerif") ; // Helvetica
            fonts.add( "Serif") ; // Times
            fonts.addItemListener(this) ;
```

```
                    // TextField
                    tf = new TextField(fonts.getItem(0), 30) ;
                    myFrame.add(fonts,BorderLayout.NORTH) ;
                    myFrame.add(tf,BorderLayout.CENTER);
                    // setSize and setVisible
                    myFrame.setSize(500,256) ;
                    myFrame.setVisible(true) ;
        }
        public void itemStateChanged( ItemEvent e)          {
                tf.setText( "Index: " + fonts.getSelectedIndex()) ;
                tf.setFont(new Font(fonts.getSelectedItem(),
                        tf.getFont().getStyle(),ChoiceApp.size++)) ;
        }
```



```
    public static void main(String[] args) {
            ChoiceApp ca = new ChoiceApp() ;
    }
}
```





Checkbox and CheckboxGroup

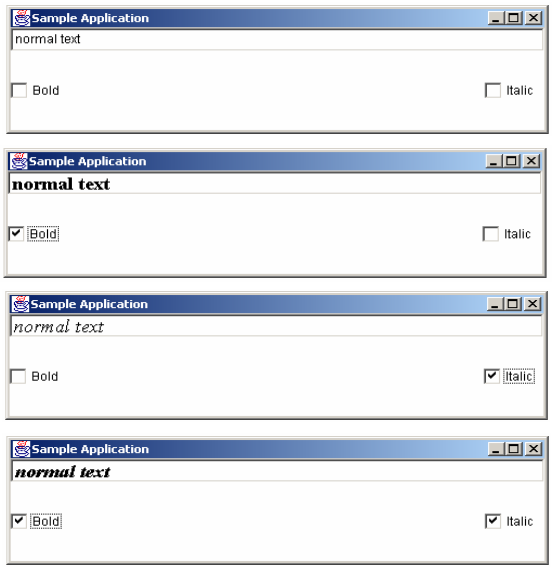
- public Checkbox(String s)
- public Checkbox(String s, CheckboxGroup c, boolean state)
- public CheckboxGroup()

```
import java.awt.* ;
import java.awt.event.* ;
public class CheckboxApp implements ItemListener {
        private Frame myFrame      ;
        private TextField tf                    ;
        private Checkbox bold,italic ;
        public CheckboxApp () {
                myFrame = new Frame("Sample Application") ;
                // Checkbox
                bold   = new Checkbox( "Bold" ) ;
                italic = new Checkbox( "Italic" ) ;
                bold.addItemListener(this) ;
                italic.addItemListener(this) ;
```



```
                // TextField
                tf = new TextField("", 30) ;
                myFrame.add(tf,BorderLayout.NORTH) ;
                myFrame.add(bold,BorderLayout.CENTER) ;
                myFrame.add(italic,BorderLayout.EAST);
                // setSize and setVisible
                myFrame.setSize(500,128) ;
                myFrame.setVisible(true) ;
        }
        public void itemStateChanged( ItemEvent e) {
           int valBold = (bold.getState() ? Font.BOLD : Font.PLAIN ) ;
           int valItalic = (italic.getState() ? Font.ITALIC : Font.PLAIN ) ;
tf.setFont(new Font("Serif",valBold+valItalic,18) );
        }
```

```
        public static void main(String[] args)
        {
                CheckboxApp ca = new CheckboxApp() ;
        }
}
```

```
import java.awt.* ;
import java.awt.event.* ;
public class GroupCheckboxApp implements ItemListener {
        private Frame myFrame      ;
        private TextField tf                    ;
        private Checkbox plain,bold,italic,boldItalic ;
        private CheckboxGroup fontStyle ;
        private Font boldFont,italicFont,plainFont,boldItalicFont ;
        public GroupCheckboxApp () {
          myFrame = new Frame("Sample Application") ;
          myFrame.setLayout(new FlowLayout());
          // Predefined Fonts
          boldFont = new Font("Serif",Font.BOLD,18) ;
          italicFont = new Font("Serif",Font.ITALIC,18) ;
          plainFont = new Font("Serif",Font.PLAIN,18) ;
boldItalicFont = new Font("Serif",Font.BOLD+Font.ITALIC,18) ;
```

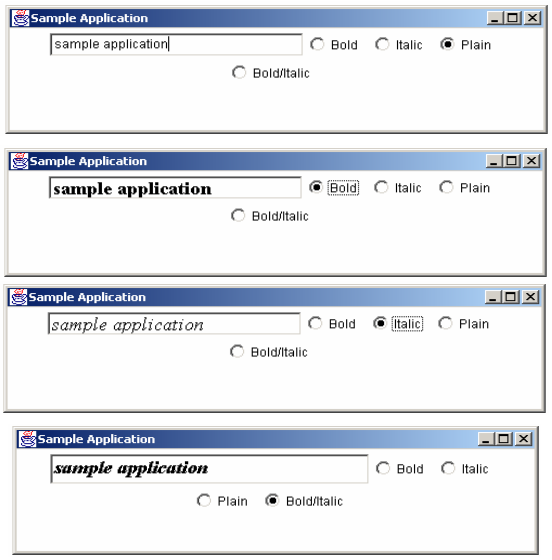


```
        // GroupCheckbox
        CheckboxGroup fontStyle = new CheckboxGroup() ;
        // Checkbox
        bold   = new Checkbox( "Bold" ,fontStyle,false) ;
        italic = new Checkbox( "Italic" ,fontStyle,false) ;
        plain  = new Checkbox( "Plain" ,fontStyle,true) ;
        boldItalic = new Checkbox( "Bold/Italic" ,fontStyle,false) ;
        // add ItemListener
        bold.addItemListener(this) ;
        italic.addItemListener(this) ;
        plain.addItemListener(this) ;
        boldItalic.addItemListener(this) ;
        // TextField
       tf = new TextField("", 30) ;
```

```
            myFrame.add(tf) ;
            myFrame.add(bold) ;
            myFrame.add(italic);
            myFrame.add(plain) ;
            myFrame.add(boldItalic);
            // setSize and setVisible
            myFrame.setSize(500,128) ;
            myFrame.setVisible(true) ;
        }
        public void itemStateChanged( ItemEvent e) {
            if( e.getSource() == plain )
              tf.setFont( plainFont);
            else  if( e.getSource() == bold )
              tf.setFont( boldFont);
            else  if( e.getSource() == italic )
              tf.setFont( italicFont);
            else if( e.getSource() == boldItalic )
              tf.setFont( boldItalicFont);
        }
```



```
    public static void main(String[] args)
    {
            GroupCheckboxApp ca = new GroupCheckboxApp() ;
    }
}
```



Single-Selection List

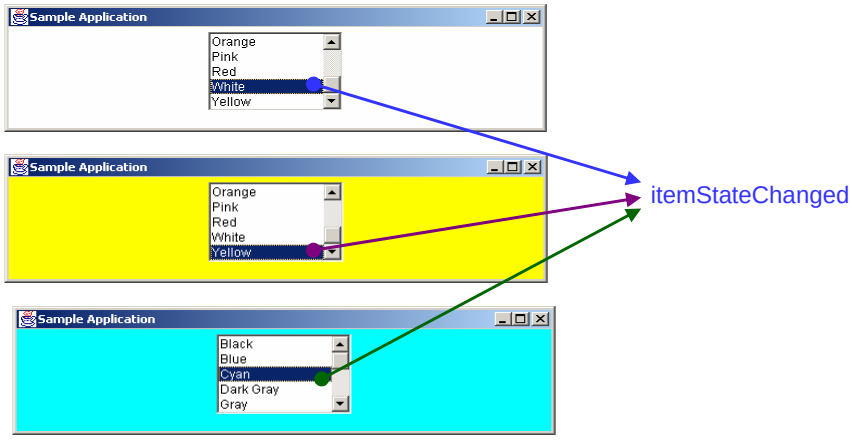
colorList = new List(5 , false) ; // single-select

```
public class SingleSelectListApp implements ItemListener {
    private Frame myFrame      ;
    private List colorList     ;
    private String[] colorNames = {
       "Black" , "Blue", "Cyan", "Dark Gray", "Gray", "Green",
       "Light Gray","Magenta","Orange", "Pink","Red","White","Yellow“ } ;
     private Color[] colors = {
        Color.black ,Color.blue,Color.cyan,Color.darkGray,Color.gray,
        Color.green,Color.lightGray,Color.magenta,Color.orange,
        Color.pink, Color.red, Color.white, Color.yellow  } ;
     public SingleSelectListApp ()  {
            myFrame = new Frame("Sample Application") ;
            myFrame.setLayout(new FlowLayout());
            // List
            colorList  = new List( 5 , false) ;
            // add ItemListener
            colorList.addItemListener(this) ;
            // add items to the list
            for(int i=0;i< colorNames.length;i++)
                colorList.add(colorNames[i]) ;
            myFrame.add(colorList)    ;
            // setSize and setVisible
            myFrame.setSize(500,128) ;
            myFrame.setVisible(true) ;
          }
```



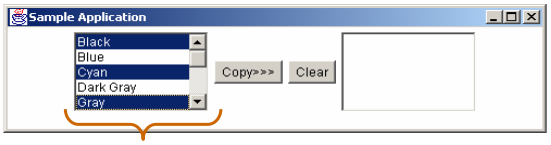
```
     public void itemStateChanged( ItemEvent e) {
          myFrame.setBackground(colors[colorList.getSelectedIndex()]) ;
     }

    public static void main(String[] args)  {
        SingleSelectListApp ssla = new SingleSelectListApp() ;
    }
}
```



Multiple-Selection List

colorList = new List(5 , true) ; // multiple-select



there are 5 items
3 items selected

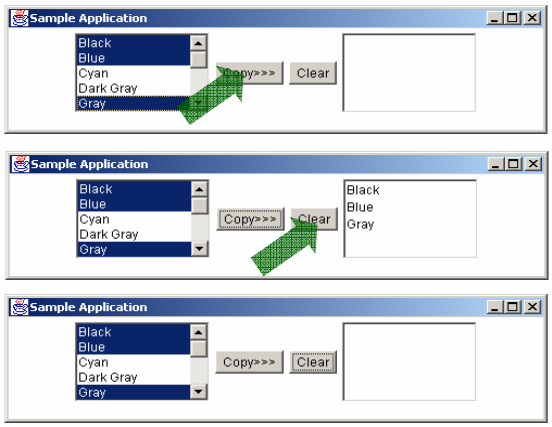
```
public class MultipleSelectListApp implements ActionListener {
        private Frame myFrame      ;
        private List colorList     ;
        private List copyList      ;
        private Button copy        ;
        private Button clear       ;

        public MultipleSelectListApp () {
            myFrame = new Frame("Sample Application") ;
            myFrame.setLayout(new FlowLayout());
            // Lists
            colorList  = new List( 5 , true) ;
            copyList  = new List( 5 , false) ;
            // Button
            copy = new Button("Copy>>>") ;
            copy.addActionListener(this) ;
            clear = new Button("Clear") ;
            clear.addActionListener(this) ;
            // add items to the list
            for(int i=0;i< colorNames.length;i++)
                colorList.add(colorNames[i]) ;
```



```
            myFrame.add(colorList)    ;
            myFrame.add(copy)         ;
            myFrame.add(clear)        ;
            myFrame.add(copyList)     ;
            // setSize and setVisible
            myFrame.setSize(500,128) ;
            myFrame.setVisible(true) ;
    }
    public void actionPerformed( ActionEvent e) {
            if ( e.getSource() == copy ) {
              String[] colors ;
              // get the selected states
              colors = colorList.getSelectedItems() ;
              // copy them to copyList
              for( int i=0 ; i < colors.length ; i++ )
                  copyList.add( colors[i] ) ;
            }else{
              copyList.clear();
            }
    }
```

```
  public static void main(String[] args)
  {
    MultipleSelectListApp ssla = new MultipleSelectListApp() ;
  }
}
```



TextArea

- public TextArea( )
- public TextArea(int rows, int columns)
- public TextArea(String s)
- public TextArea(String s, int rows, int columns)
- public TextArea(String s, int rows, int columns, int scrollbars)

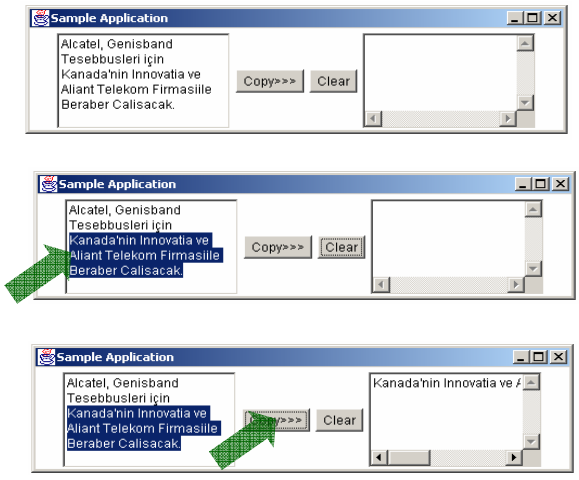
```
public class TextAreaApp implements ActionListener, TextListener {
        private Frame myFrame   ;
        private TextArea t1,t2      ;
        private Button copy         ;
        private Button clear        ;

        public TextAreaApp ()  {
          String s = "Alcatel, Genisband Tesebbusleri için " +
                     "Kanada'nin Innovatia ve Aliant Telekom Firmasi" +
                     "ile Beraber Calisacak." ;
          myFrame = new Frame("Sample Application") ;
          myFrame.setLayout(new FlowLayout());
          // TextArea
    t1  = new TextArea( s , 5 , 20 ,TextArea.SCROLLBARS_NONE ) ;
          t2  = new TextArea( 5 , 20 ) ;
          // Button
          copy = new Button("Copy>>>") ;
          copy.addActionListener(this) ;
          clear = new Button("Clear") ;
          clear.addActionListener(this) ;
          myFrame.add(t1)             ;
          myFrame.add(copy)                   ;
          myFrame.add(clear)                  ;
          myFrame.add(t2)             ;
```



```
          // setSize and setVisible
          myFrame.setSize(500,128) ;
          myFrame.setVisible(true) ;
        }
        public void textValueChanged( TextEvent e )
        {
            TextComponent source = (TextComponent) e.getSource() ;
                t2.setText( source.getText() ) ;
        }
        public void actionPerformed( ActionEvent e){
                if ( e.getSource() == copy ) {
                  t2.setText( t1.getSelectedText() )     ;
                 }else{
                  t2.setText("") ;
                 }
        }

        public static void main(String[] args) {
                 TextAreaApp taa = new TextAreaApp() ;
        }
}
```



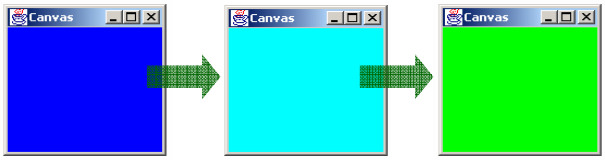
Canvas

- A canvas is a dedicated drawing area that can also receive mouse events.
- Class Canvas inherits from Component
- The paint method for a Canvas must be overriden to draw on the Canvas
- Drawing on a Canvas is performed with coordinates that are measured from the upper-left corner (0,0) of the Canvas.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class MyCanvas extends Canvas implements KeyListener {
        int index;
        Color[] colors =  {
        Color.black , Color.blue, Color.cyan, Color.darkGray, Color.gray,
        Color.green,Color.lightGray, Color.magenta, Color.orange,
        Color.pink, Color.red, Color.white,Color.yellow } ;
        public void paint(Graphics g) {
                g.setColor(colors[index]);
                g.fillRect(0,0,getSize().width,getSize().height);
        }
       public void keyTyped(KeyEvent ev) {
          index++;
          if (index == colors.length)
            index =0;
         repaint();
    }
   public void keyPressed(KeyEvent ev) {}
   public void keyReleased(KeyEvent ev) {}
```

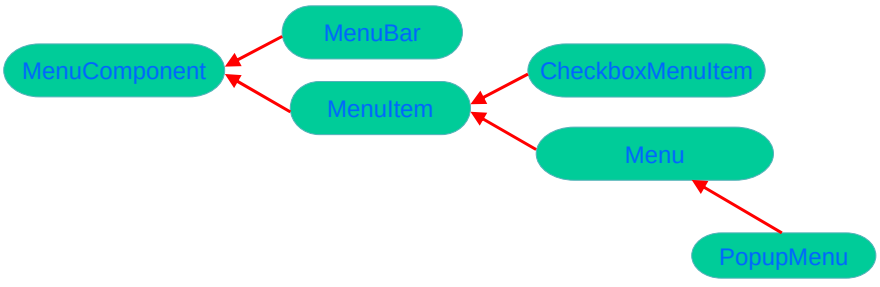


```
        public static void main(String args[]) {
                Frame f = new Frame("Canvas");
                MyCanvas mc = new MyCanvas();
                f.add(mc,BorderLayout.CENTER);
                f.setSize(150, 150);
                mc.requestFocus();
                mc.addKeyListener(mc);
               f.setVisible(true);
            }
   }
```

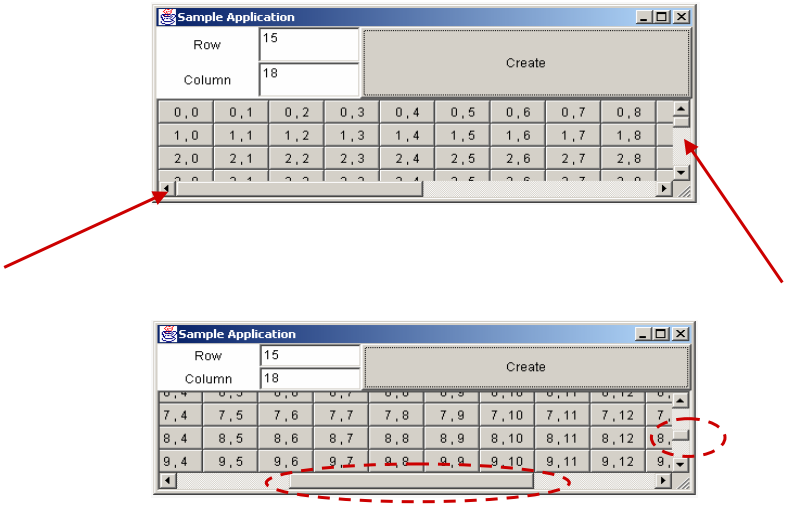


Menus with Frames

- Menus are an integral part of GUIs
- Menus allow the user to perform actions without unneccessarily cluttering a graphical user interface with extra GUI components
- Menus can only be used with Frames
- PopupMenus can be used with any GUI component



ScrollPane




```
import java.awt.* ;
import java.awt.event.* ;
public class ScrollPaneApp implements ActionListener, TextListener {
        private Frame myFrame      ;
        private Panel np           ;
        private Panel innerPanel   ;
        private ScrollPane sp      ;
        private Button[] buttons   ;
        private Label rlabel,clabel   ;
        private TextField rtf,ctf     ;
        private Button create      ;
        private int row=10,column=10    ;
        private Dialog d           ;
```

```
public ScrollPaneApp () {
    myFrame = new Frame("Sample Application") ;
    np = new Panel() ;
    np.setLayout(new GridLayout(2,2)) ;
    rlabel = new Label( "Row" , Label.CENTER ) ;
    clabel = new Label( "Column" , Label.CENTER ) ;
    rtf    = new TextField( 10 ) ;
    rtf.addTextListener(this)    ;
    ctf    = new TextField( 10 ) ;
    ctf.addTextListener(this)    ;
    np.add(rlabel)               ;
    np.add(rtf)                  ;
    np.add(clabel)               ;
    np.add(ctf)                  ;
```



```
        // ScrollPane
        sp = new ScrollPane() ;
        innerPanel = new Panel() ;
        sp.add(innerPanel)    ;
        // Button
        create = new Button("Create") ;
        create.addActionListener(this);
        // setSize and setVisible
        myFrame.add(np,BorderLayout.WEST) ;
        myFrame.add(create,BorderLayout.CENTER) ;
        myFrame.add(sp,BorderLayout.SOUTH) ;
        myFrame.setSize(500,200) ;
        myFrame.setVisible(true) ;
}
```

```
public void textValueChanged( TextEvent e ) {
        if ( e.getSource() == rtf ) {
            row = Integer.parseInt( rtf.getText() ) ;
        }else {
            column = Integer.parseInt( ctf.getText() ) ;
        }
}
```



```
public void actionPerformed( ActionEvent e) {
    if( (row>0) && (column>0) ) {
    innerPanel.setLayout(new GridLayout(row,column)) ;
    buttons = new Button[row*column] ;
    for(int i=0;i<row;i++)
      for(int j=0;j<column;j++) {
        buttons[i*column+j] = new Button( i + " , " + j) ;
          innerPanel.add(buttons[i*column+j]);
      }
     myFrame.pack();
}else {
```

```
            d = new Dialog(myFrame, "Ooops Dialog", false);
            Button b = new Button("OK") ;
            b.addActionListener(this) ;
            d.add(new Label("Oooppps..."),BorderLayout.CENTER);
            d.add(b,BorderLayout.SOUTH);
            d.pack();
            d.setVisible(true);
        }
    }
    public static void main(String[] args) {
        ScrollPaneApp spa = new ScrollPaneApp() ;
    }
}
```


Dialog

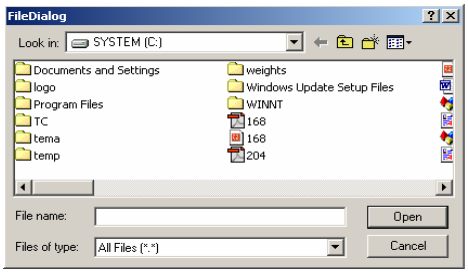
```
Dialog d = new Dialog(myFrame, "Dialog", false);
Button b = new Button("OK") ;
b.addActionListener(this) ;
d.add(new Label("Hello,I'm a Dialog"),BorderLayout.CENTER);
d.add(b,BorderLayout.SOUTH);
d.pack();
d.setVisible(true);
```

Creating a FileDialog

```
FileDialog d = new FileDialog(parentFrame, "FileDialog");

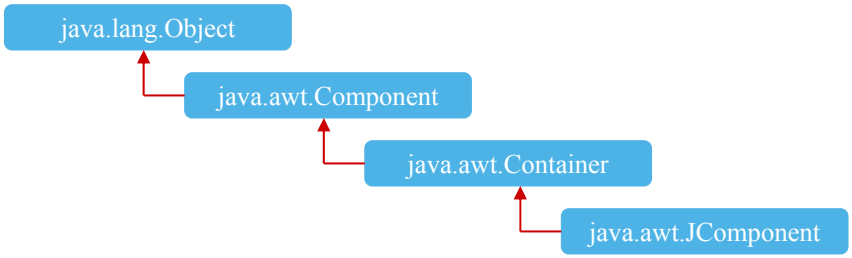
d.setVisible(true); // block here until OK selected

String fname = d.getFile();
```



A SWING

- Swing GUI components
  - Defined in package **javax.swing**
  - Original GUI components from Abstract Windowing Toolkit in **java.awt**
    - Heavyweight components - rely on local platform's windowing system for look and feel
  - Swing components are lightweight
    - Written in Java, not weighed down by complex GUI capabilities of platform
    - More portable than heavyweight components
  - Swing components allow programmer to specify look and feel
    - Can change depending on platform
    - Can be the same across all platforms



- **Component** defines methods that can be used in its subclasses (for example, **paint** and **repaint**)
- **Container** - collection of related components
  - When using **JFrames**, attach components to the content pane (a **Container**)
  - Method **add**
- **JComponent** - superclass to most Swing components
- Much of a component's functionality inherited from these classes

Swing Overview

- Some capabilities of subclasses of **JComponent**
  - Pluggable look and feel
  - Shortcut keys (mnemonics)
    - Direct access to components through keyboard
  - Common event handling
    - If several components perform same actions
  - Tool tips
    - Description of component that appears when mouse over it

JLabel

- Labels
  - Provide text instructions on a GUI
  - Read-only text
  - Programs rarely change a label's contents
  - Class **JLabel** (subclass of **JComponent**)
- Methods

```
18    label1 = new JLabel( "Label with text" );
```

  - **myLabel.setToolTipText("Text")**
    - Displays **"Text"** in a tool tip when mouse over label
  - **myLabel.setText("Text")**
  - **myLabel.getText()**

Icon

- Object that implements interface **Icon**

- One class is **ImageIcon** (**.gif** and **.jpeg** images)

```
Icon bug = new ImageIcon( "bug1.gif" );
```

- Display an icon with **setIcon** method (of class **JLabel**)
  - **myLabel.setIcon(myIcon);**

```
label3.setIcon( bug );
```


Alignment

- By default, text appears to right of image
- **`JLabel`** methods **`setHorizontalTextPosition`** and **`setVerticalTextPosition`**
  - Specify where text appears in label
  - Use integer constants defined in interface **`SwingConstants`** (**`javax.swing`**)
    - **`SwingConstants.LEFT, RIGHT, BOTTOM, CENTER`**

Another `JLabel` constructor

- **`JLabel( "Text", ImageIcon, Text_Alignment_CONSTANT)`**

JTextField, JPasswordField

- Single line areas in which text can be entered or displayed
- **JPasswordField**s show inputted text as an asterisk *

Swing A

- **JTextField** extends **JTextComponent**
  - **JPasswordField** extends **JTextField**

- When **Enter** pressed
  - **ActionEvent** occurs
  - Currently active field "has the focus"

- Methods
  - Constructors
    - **`JTextField( 10 )`**
      - Textfield with 10 columns of text
      - Takes average character width, multiplies by 10
    - **`JTextField( "Hi" )`**
      - Sets text, width determined automatically
    - **`JTextField( "Hi", 20 )`**
  - **`setEditable( boolean )`**
    - If **`false`**, user cannot edit text
    - Can still generate events
  - **`getPassword`**
    - Class **`JPasswordField`**
    - Returns password as an **`array`** of type **`char`**

Swing A

JButton

- Button
  - Component user clicks to trigger an action
  - Several types of buttons
    - Command buttons, toggle buttons, check boxes, radio buttons
- Command button
  - Generates **ActionEvent** when clicked
  - Created with class **JButton**
    - Inherits from class **AbstractButton**
    - Defines many features of Swing buttons
- **JButton**
  - Text on face called button label
  - Each button should have a different label
  - Support display of **Icon**s

- Methods of class **`JButton`**
  - Constructors

```
JButton myButton = new JButton( "Label" );
JButton myButton = new JButton( "Label", myIcon );
```

  - **`setRolloverIcon( myIcon )`**
    - Sets image to display when mouse over button
- Class ActionEvent
  - **`getActionCommand`**
    - Returns label of button that generated event

JCheckBox, JRadioButton

- State buttons
  - **JToggleButton**
    - Subclasses **JCheckBox**, **JRadioButton**
  - Have on/off (true/false) values

- Class **JCheckBox**

  - Text appears to right of checkbox
  - Constructor

```
JCheckBox myBox = new JCheckBox( "Title" );
```

Swing A

- When **JCheckBox** changes
  - **ItemEvent** generated
    - Handled by an **ItemListener**, which must define **itemStateChanged**
  - Register handlers with with **addItemListener**

```
private class CheckBoxHandler implements ItemListener {
        public void itemStateChanged( ItemEvent e ) {
}
```

  - **getStateChange**
    - Returns **ItemEvent.SELECTED** or **ItemEvent.DESELECTED**
- **JTextField**
  - Method **setText(fontObject)**
    - **new Font(name, style_CONSTANT, size)**
    - **style_CONSTANT - FONT.PLAIN, BOLD, ITALIC**
      - Can add to get combinations

Swing A

Swing A

- Radio buttons
  - Have two states: selected and deselected
  - Normally appear as a group
    - Only one radio button in the group can be selected at time
    - Selecting one button forces the other buttons off
  - Used for mutually exclusive options
  - **ButtonGroup** - maintains logical relationship between radio buttons
- Class **JRadioButton**
  - Constructor
    - **JRadioButton("Label", selected)**
    - If selected **true**, **JRadioButton** initially selected

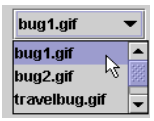
Swing A

- Class **JRadioButton**
  - Generates **ItemEvents** (like **JCheckBox**)
- Class **ButtonGroup**
  - **ButtonGroup myGroup = new ButtonGroup();**
  - Binds radio buttons into logical relationship
  - Method **add**
    - Associate a radio button with a group

    myGroup.add(myRadioButton)

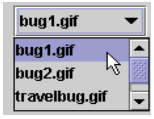
JComboBox

- Combo box (drop down list)
  - List of items, user makes a selection
  - Class **JComboBox**
    - Generate **ItemEvents**



- **JComboBox**
  - Constructor

    JComboBox (arrayOfNames)
  - Numeric index keeps track of elements
    - First element added at index **0**
    - First item added is appears as currently selected item when combo box appears



- **JComboBox** methods
  - **getSelectedIndex**
    - Returns the index of the currently selected item
    - **myComboBox.getSelectedIndex()**
  - **setMaximumRowCount(n)**
    - Set max number of elements to display when user clicks combo box
    - Scrollbar automatically provided
    - **setMaximumRowCount(3)**

JList

- List
  - Displays series of items, may select one or more
  - This section, discuss single-selection lists
- Class **JList**
  - Constructor **JList(arrayOfNames)**
    - Takes array of **Objects** (**String**s) to display in list
  - **setVisibleRowCount(n)**
    - Displays **n** items at a time
    - Does not provide automatic scrolling

- **JScrollPane** object used for scrolling

```
c.add( new JScrollPane( colorList ) );
```

  - Takes component to which to add scrolling as argument
  - Add **JScrollPane** object to content pane
- **JList** methods
  - **setSelectionMode(selection_CONSTANT)**
  - **SINGLE_SELECTION**
    - One item selected at a time
  - **SINGLE_INTERVAL_SELECTION**
    - Multiple selection list, allows contiguous items to be selected
  - **MULTIPLE_INTERVAL_SELECTION**
    - Multiple-selection list, any items can be selected

- **JList** methods
  - **getSelectedIndex()**
    - Returns index of selected item
- Event handlers
  - Implement interface **ListSelectionListener** (**javax.swing.event**)
  - Define method **valueChanged**
  - Register handler with **addListSelectionListener**

Multiple Selection List

- Multiple selection lists
  - **SINGLE_INTERVAL_SELECTION**
    - Select a contiguous group of items by holding *Shift* key
  - **MULTIPLE_INTERVAL_SELECTION**
    - Select any amount of items
    - Hold *Ctrl* key and click each item to select
- **JList** methods
  - **getSelectedValues()**
    - Returns an array of **Objects** representing selected items
  - **setListData(arrayOfObjects)**
    - Sets items of **JList** to elements in **arrayOfObjects**

JPanel

- Complex GUIs
  - Each component needs to be placed in an exact location
  - Can use multiple panels
    - Each panel's components arranged in a specific layout
- Panels
  - Class **JPanel** inherits from **JComponent**, which inherits from **java.awt.Container**
    - Every **JPanel** is a **Container**
  - **JPanels** can have components (and other **JPanels**) added to them
    - **JPanel** sized to components it contains
    - Grows to accomodate components as they are added

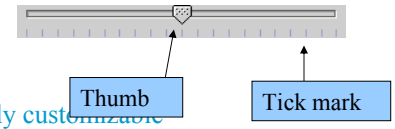
JTextArea

- **JTextArea**
  - Area for manipulating multiple lines of text
  - Like **JTextField**, inherits from **JTextComponent**
    - Many of the same methods
  - Does not have automatic scrolling
  - Methods
    - **getSelectedText**
      - Returns selected text (dragging mouse over text)
    - **setText(string)**
  - Constructor
    - **JTextArea(string, numrows, numcolumns)**
- **JScrollPane**
  - Provides scrolling for a component

- Initialize with component
  - **new JScrollPane(myComponent)**
- Can set scrolling policies (always, as needed, never)
- Methods **setHorizontalScrollBarPolicy**, **setVerticalScrollBarPolicy**
  - Constants:
  - **JScrollPane.VERTICAL_SCROLLBAR_ALWAYS**
  - **JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED**
  - **JScrollPane.VERTICAL_SCROLLBAR_NEVER**
    - Similar for **HORIZONTAL**
  - If set to **JScrollPane.HORIZONTAL_SCROLLBAR_NEVER**, word wrap

JSlider

- Select from a range of integer values



- Highly customizable
  - Snap-to ticks, major and minor ticks, labels
- When has focus (currently selected GUI component)
  - Use mouse or keyboard
    - Arrow or keys to move thumb, *Home*, *End*
- Have horizontal or vertical orientation
  - Minimum value at left/bottom, maximum at right/top
  - Thumb indicates current value

- Methods
  - Constructor
  - **JSlider(orientation_CONSTANT, min, max, initialValue)**
    - **orientation_CONSTANT**
      - **SwingConstants.HORIZONTAL**
      - **SwingConstants.VERTICAL**
    - **min, max** - range of values for slider
    - **initialValue** - starting location of thumb

```
diameter = new JSlider( SwingConstants.HORIZONTAL,
                        0, 200, 10 );
```


- Methods
  - **setMajorTickSpacing(n)**
    - Each tick mark represents **n** values in range
  - **setPaintTicks(boolean)**
    - **false** (default) - tick marks not shown
  - **getValue()**
    - Returns current thumb position
- Events
  - JSliders generates **ChangeEvent**s
    - **addChangeListener**
    - Define method **stateChanged**

JFrame

- **JFrame**
  - Inherits from **java.awt.Frame**, which inherits from **java.awt.Window**
  - **JFrame** is a window with a title bar and a border
    - Not a lightweight component - not written completely in Java
    - Window part of local platform's GUI components
      - Different for Windows, Macintosh, and UNIX
- **JFrame** operations when user closes window
  - Controlled with method **setDefaultCloseOperation**
    - Interface **WindowConstants** (**javax.swing**) has three constants to use
    - **DISPOSE_ON_CLOSE**, **DO_NOTHING_ON_CLOSE**, **HIDE_ON_CLOSE** (default)

- Windows take up valuable resources
  - Explicitly remove windows when not needed
  - Method **dispose** (of class **Window**, indirect superclass of **JFrame**)
    - Or, use **setDefaultCloseOperation**
  - **DO_NOTHING_ON_CLOSE** -
    - You determine what happens when user wants to close window
- Display
  - By default, window not displayed until method **show** called
  - Can display by calling method **setVisible(true)**
  - Method **setSize**
    - Set a window's size else only title bar will appear

Menu

- Menus
  - Important part of GUIs
  - Perform actions without cluttering GUI
  - Attached to objects of classes that have method **setJMenuBar**
    - **JFrame** and **JApplet**
  - **ActionEvent**s
- Classes used to define menus
  - **JMenuBar** - container for menus, manages menu bar
  - **JMenuItem** - manages menu items
    - Menu items - GUI components inside a menu
    - Can initiate an action or be a submenu
    - Method **isSelected**

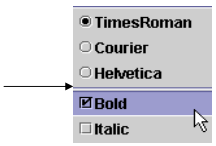
- Classes used to define menus (continued)
  - **JMenu** - manages menus
    - Menus contain menu items, and are added to menu bars
    - Can be added to other menus as submenus
    - When clicked, expands to show list of menu items
  - **JCheckBoxMenuItem** (extends **JMenuItem**)
    - Manages menu items that can be toggled
    - When selected, check appears to left of item
  - **JRadioButtonMenuItem** (extends **JMenuItem**)
    - Manages menu items that can be toggled
    - When multiple **JRadioButtonMenuItem**s are part of a group (**ButtonGroup**), only one can be selected at a time
    - When selected, filled circle appears to left of item

- Mnemonics
  - Quick access to menu items (File)
    - Can be used with classes that have subclass **`javax.swing.AbstractButton`**
  - Method **`setMnemonic`**

```
JMenu fileMenu = new JMenu( "File" )
fileMenu.setMnemonic( 'F' );
```

    - Press **`Alt + F`** to access menu
- Methods
  - **`setSelected( true )`**
    - Of class **`AbstractButton`**
    - Sets button/item to selected state

- Methods (continued)
  - **addSeparator()**
    - Of class **JMenu**
    - Inserts separator line into menu



- Dialog boxes
  - Modal - No other window can be accessed while it is open (default)
    - Modeless - other windows can be accessed

Dialogs

- **JOptionPane.showMessageDialog(parentWindow, text, title, messageType)**
- **parentWindow** - determines where dialog box appears
  - **null** - displayed at center of screen
  - Window specified - dialog box centered horizontally over parent

Swing A

```
JOptionPane.showMessageDialog( MenuTest.this,
                               "This is an example\nof using menus",
                               "About", JOptionPane.PLAIN_MESSAGE );
```


- Using menus
  - Create menu bar
    - Set menu bar for **JFrame**
      - **setJMenuBar(myBar);**
  - Create menus
    - Set Mnemonics
  - Create menu items
    - Set Mnemonics
    - Set event handlers
  - If using **JRadioButtonMenuItem**s
    - Create a group: **myGroup = new ButtonGroup();**
    - Add **JRadioButtonMenuItem**s to the group

Swing A

- Add menu items to appropriate menus
  - `myMenu.add( myItem );`
  - Insert separators if necessary:
    `myMenu.addSeparator();`
- If creating submenus, add submenu to menu
  - `myMenu.add( mySubMenu );`
- Add menus to menu bar
  - `myMenuBar.add( myMenu );`

13 Threads

Threads

Objectives

- Define a thread
- Create separate threads in a Java program, controlling the code and data that are used by that thread
- Control the execution of a thread and write platform-independent code with threads
- Describe the difficulties that might arise when multiple threads share data
- Use keyword synchronized to protect data from corruption
- Use wait() and notify() to communicate beween threads

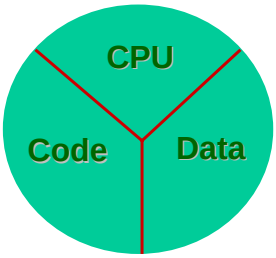
Threads

- What are threads?
  - Virtual CPU
- Remember “Garbage Collector”

Three Parts of a Thread

A thread or execution context is composed of three main parts:

- A virtual CPU
- The code that the CPU is executing
- The data on which the code works




```
public class ThreadTester {
        public static void main(String[] args) {
                HelloRunner r = new HelloRunner() ;
                Thread t = new Thread(r) ;
                t.start();
        }
}
class HelloRunner implements Runnable {
        int i ;
        public void run() {
                i = 0 ;
                while (true) {
                    System.out.println("Hello "+ i++ ) ;
                   if( i == 500 ) break;
                }
        }
}
```

Creating the Thread

Multithreaded programming

- Multiple threads from same Runnable instance
- Threads share same data and code
- public Thread()
- public Thread(String threadName)
- setName(), getName()

- Example

```
Thread t1 = new Thread(r) ;
Thread t2 = new Thread(r) ;
```


Starting the Thread

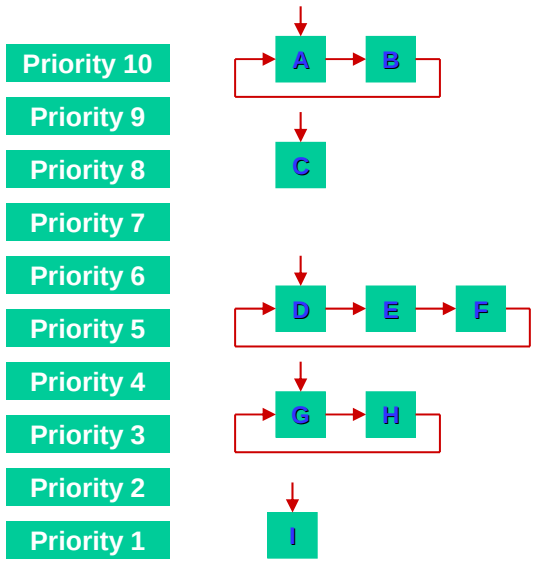
- Using the start() method
- Placing the thread in runnable state

Thread Scheduling

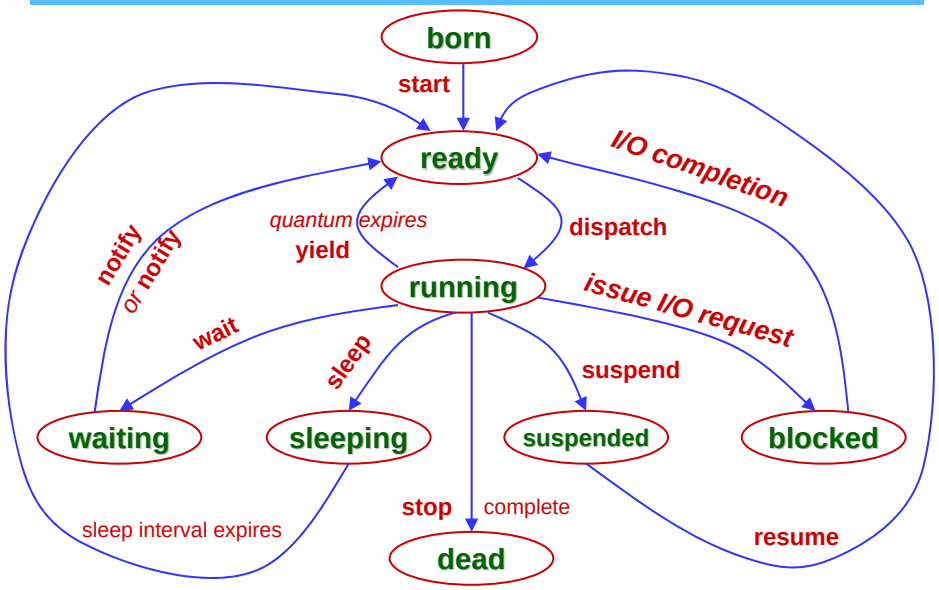
- In java technology, threads are usually preemptive, but not necessarily time-sliced
- Although Java is perphaps the world's most portable programming language, certain portions of the language are nevertheless platform dependent. In particular, there differences among Solaris and Windows-based implementations.
- The early Solaris Java platform runs a thread of a given priority to completion or until higher-priority thread becomes ready.
- In Windows implementations, threads are timesliced. This means that each thread is given a limited amount of time (called "quantum") to execute on a processor, and when that times expires the threadis made to wait while all other threads of equal priority get their chances to use their quantim in *round robin* fashion.

- Every Java thread has a priority in the range Thread.MIN_PRIORITY (a constant of 1) and Thread.MAX_PRIORITY (a constant of 10).
- By default, each thread is given priority Thread.NORM_PRIORITY (a constant of 5).
- A thread's priority can be adjusted with the setPriority method which takes an int argument. If the argument is notin the range 1 through 10 inclusive, then method setPriority throws an IllegalArgumentException.
- Method getPriority returns the thread's priority.

Round Robin



Thread States: Life Cycle of a Thread



- Born state
  - Thread just created
  - When **start** called, enters ready state
- Ready state (runnable state)
  - Highest-priority ready thread enters running state
- Running state
  - System assigns processor to thread (thread begins executing)
  - When **run** method completes or terminates, enters dead state
- Dead state
  - Thread marked to be removed by system
  - Entered when **run** terminates or throws uncaught exception

- Blocked state
  - Entered from running state
  - Blocked thread cannot use processor, even if available
  - Common reason for blocked state - waiting on I/O request
- Sleeping state
  - Entered when **sleep** method called
  - Cannot use processor
  - Enters ready state after sleep time expires
- Waiting state
  - Entered when **wait** called in an object thread is accessing
  - One waiting thread becomes ready when object calls **notify**
  - **notifyAll** - all waiting threads become ready

```
public class Runner implements Runnable {
        public void run()  {
                while (true)  {
                        // do lots of interesting stuff
                        // :
                  // give other threads a chance
                  try {
                     Thread.sleep(10) ;
                   } catch(InterruptedException e) {
                    // This thread's sleep was interrupted
                     // by another thread
                   }
                }
        }
}
```


Terminating a Thread

- When a thread completes execution and terminates, it cannot run again.
- You can stop a thread by using a flag that indicates that the run method should exit

```
public class Runner implements Runnable {
        private boolean timeToQuit = false ;
        public void run()  {
                while ( ! timeToQuit )   {
                        // :
                  }
        }
        public void stopRunning()
        {
                timeToQuit = true ;
        }
}
```



```
public class ThreadController
{
        private Runner r = new Runner() ,
        private Thread t = new Thread(r) ;
        public void startThread() {
                t.start() ;
        }
       public void stopThread()  {
                r.stopRunning() ;
      }
}
```

Thread.currentThread()

```
public class NameRunner implements Runnable
{
    public void run() {
       while (true)  {

       }
       System.out.println("Thread " + Thread.currentThread().getName()  +
                                  " completed" );
    }
}
```


Basic Control of Threads

- Testing threads
  - isAlive()
- Accessing thread prioritythreads
  - getPriority()
  - setPriority()
- Putting threads on holdthreads
  - Thread.sleep()
  - join()
  - Thread.yield

Thread.sleep()

When a running thread's sleep method is called, that thread enters the *sleeping* state. A sleeping thread becomes ready after the designated sleep time expires. A sleeping thread cannot use a processor even if one is available.

join()

waits for the Thread to which the message is sent to die before the current Thread can proceed

stop()

stops a thread by throwing a ThreadDeath object.

yield()

a thread call the yield method to give other threads a chance to execute. Actually, whenever a higher-priority thread becomes ready, the current thread is preempted, so a thread cannot yield to a higher-priority thread because the first thread will have been preempted when the higher-priority thread became ready. Similarly, yield always allows the highest-priority *ready* thread to run, so if only lower-priority threads are ready at the time of yield call, the current thread will be the highest-priority thread and will continue executing. Therefore, a thread yields to give threads of an equal priority a chance to run. On time-sliced system this is unnecessary because threads of equal priority will each execute for their quantum and other threads of equal priority will execute in round-robin fashion.

Other Ways to Create Threads

Threads 13

```
public class MyThread extends Thread {
        public void run() {
                while (true)  {
                }
        }
        public static void main(String[] args)  {
                Thread t = new MyThread() ;
                t.start() ;
        }
}
```


EXAMPLE

Threads 13

```
public class PrintThread extends Thread {
        private int sleepTime ;
        public PrintThread(){
            sleepTime = (int) (Math.random() * 5000.0 ) ;
            System.out.println("name "+getName()+";sleep:"+sleepTime);
        }
        public void run(){
                try {
                   Thread.sleep(sleepTime) ;
                }
                 catch(InterruptedException e) {
                   System.out.println( e.toString() ) ;
                }
                System.out.println( getName() ) ;
    }
}
```

```
public class PrintTest  {
        public static void main(String[] args)  {
                PrintThread thread1,thread2,thread3,thread4 ;
                thread1 = new PrintThread() ;
                thread2 = new PrintThread() ;
                thread3 = new PrintThread() ;
                thread4 = new PrintThread() ;

                thread1.start() ;
                thread2.start() ;
                thread3.start() ;
                thread4.start() ;
        }
}
```


Critical Section – Mutual Exclusion

```
public class MyStack {
        private int idx = 0 ;
        private char[] data = new char[6] ;
        public void push(char c) {
                data[idx] = c ;
                idx++;                          critical section
        }
        public char pop() {
                --idx;
                return data[idx] ;              critical section
        }
}
```

Thread t1 ;

```
private Stack s ;
:
s.push('a') ;
s.push('b');
:
```

Thread t2;

```
private Stack s ;
:
s.pop('a') ;
s.push('b');
:
```

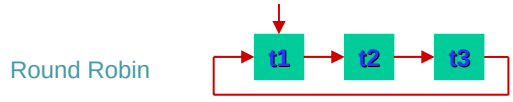
Thread t3;

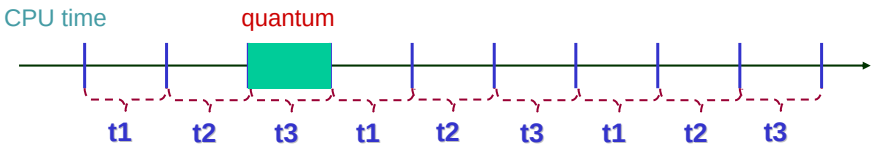
```
private Stack s ;
:
s.push('a') ;
s.push('b');
:
```

```
public void push(char c) {
1           data[idx] = c ;
2           idx++;
}
```



```
public void push(char c) {
1           data[idx] = c ;
2           idx++;
}
```





The Object Lock Flag

- Every object has a flag that can be thought of as a “lock flag” .
- `synchronized` allows interaction with the lock flag


```
public void push(char c) {
    synchronized (this) {
        data[idx] = c ;
        idx++;
    }
}
```

mutual exclution


```
public char pop() {
    synchronized (this) {
            --idx;
            return data[idx] ;
    }
```

mutual exclution

Releasing the Lock Flag

- Released when the thread passes the end of the synchronized() code block
- Automatically released when a break or exception is thrown by the synchronized() code block

synchronized – Putting It Together

- *All* access to delicate data should be synchronized.
- Delicate data protected by synchronized should be private.

```
public void push(char c) {
    synchronized (this) {
        data[idx] = c ;
        idx++;
    }
}
```

```
public synchronized void push(char c) {
        data[idx] = c ;
        idx++;
}
```

Deadlock

- Two threads waiting for a lock from the other
- It is not detected or avoided
- It can be avoided by:
  - Deciding on order to obtain locks
  - Adhering to this order throughout
  - Releasing locks in reverse order

Thread Interaction – `wait` and `notify`

- Scenario
  - Consider yourself and a cab driver as two threads
- The problem
  - How to determine when you are at your destination
- The solution
  - You notify the cabbie of your destination and relax
  - Cabbie drives and notifies you upon arrival at your destination

Thread Interaction

- wait() and notify()
- The pools
  - Wait pool
  - Lock pool

Monitor Model for Synchronization

- Leave shared data in a consistent state
- Ensure programs cannot deadlock
- Do not put threads expecting different notifications in the same wait pool

Producer/Consumer Relationship Without Thread Synchronization


```
public class SharedCell {
  public static void main( String args[] )  {
    HoldInteger h = new HoldInteger();
    ProduceInteger p = new ProduceInteger( h );
    ConsumeInteger c = new ConsumeInteger( h );
    p.start();
    c.start();
  }
}
```


PRODUCER

```
class ProduceInteger extends Thread {
  private HoldInteger pHold;
  public ProduceInteger( HoldInteger h )  {
    pHold = h;
  }
  public void run()  {
    for ( int count = 0; count < 10; count++ ) {
      // sleep for a random interval
      try {
        Thread.sleep( (int) ( Math.random() * 3000 ) );
      }
      catch( InterruptedException e ) {
        System.err.println( e.toString() );
      }
```

```
            pHold.setSharedInt( count );
             System.out.println( "Producer set sharedInt to " + count );
          }
          pHold.setMoreData( false );
        }
      }
```


CONSUMER

```
class ConsumeInteger extends Thread {
  private HoldInteger cHold;
  public ConsumeInteger( HoldInteger h )  {
    cHold = h;
  }
  public void run()  {
    int val;
    while ( cHold.hasMoreData() ) {
      // sleep for a random interval
      try {
        Thread.sleep( (int) ( Math.random() * 3000 ) );
      }
      catch( InterruptedException e ) {
        System.err.println( e.toString() );
      }
```

```
            val = cHold.getSharedInt();
            System.out.println( "Consumer retrieved " + val );
          }
        }
      }
```



```
class HoldInteger {
  private int sharedInt = -1;
  private boolean moreData = true;

  public void setSharedInt( int val ) { sharedInt = val; }

  public int getSharedInt() { return sharedInt; }

  public void setMoreData( boolean b ) { moreData = b; }

  public boolean hasMoreData() { return moreData; }
}
```

Producer/Consumer Relationship With Thread Synchronization


```
public class SharedCell {
  public static void main( String args[] )  {
    HoldInteger h = new HoldInteger();
    ProduceInteger p = new ProduceInteger( h );
    ConsumeInteger c = new ConsumeInteger( h );
    p.start();
    c.start();
  }
}
```


PRODUCER

```
class ProduceInteger extends Thread {
  private HoldInteger pHold;
  public ProduceInteger( HoldInteger h )  {
    pHold = h;
  }
  public void run()  {
    for ( int count = 0; count < 10; count++ ) {
      // sleep for a random interval
      try {
        Thread.sleep( (int) ( Math.random() * 3000 ) );
      }
      catch( InterruptedException e ) {
        System.err.println( e.toString() );
      }
```

```
            pHold.setSharedInt( count );
             System.out.println( "Producer set sharedInt to " + count );
          }
          pHold.setMoreData( false );
        }
      }
```


CONSUMER

```
class ConsumeInteger extends Thread {
  private HoldInteger cHold;
  public ConsumeInteger( HoldInteger h )  {
    cHold = h;
  }
  public void run()  {
    int val;
    while ( cHold.hasMoreData() ) {
      // sleep for a random interval
      try {
        Thread.sleep( (int) ( Math.random() * 3000 ) );
      }
      catch( InterruptedException e ) {
        System.err.println( e.toString() );
      }
```

```
            val = cHold.getSharedInt();
            System.out.println( "Consumer retrieved " + val );
          }
        }
      }
```


PRODUCER

```
class HoldInteger {
  private int sharedInt = -1;
  private boolean moreData = true;
  private boolean writeable = true;
  public synchronized void setSharedInt( int val )  {
    while ( !writeable ) {
      try {
        wait();
      }
      catch ( InterruptedException e ) {
        System.err.println( "Exception: " + e.toString() );
      }
    }
    sharedInt = val;
    writeable = false;
    notify();
  }
```

CONSUMER

```
public synchronized int getSharedInt()
{
   while ( writeable ) {
      try {
         wait();
      }
      catch ( InterruptedException e ) {
         System.err.println( "Exception: " + e.toString() );
      }
   }
   writeable = true;
   notify();
   return sharedInt;
}
```



```
   public void setMoreData( boolean b ) { moreData = b; }

   public boolean hasMoreData() { return moreData; }
}
```

Producer/Consumer Relationship
The Circular Buffer


```
import java.applet.Applet;
import java.awt.*;
import java.text.DecimalFormat;
public class SharedCell extends Applet {
  private TextArea output;
  public void init()  {
    setLayout( new BorderLayout() );
    output = new TextArea();
    add( output, BorderLayout.CENTER );
  }
  public void start()  {
    HoldInteger h = new HoldInteger( output );
    ProduceInteger p = new ProduceInteger( h );
    ConsumeInteger c = new ConsumeInteger( h );
    p.start();
    c.start();
  }
}
```


PRODUCER


```
class ProduceInteger extends Thread {
  private HoldInteger pHold;
  public ProduceInteger( HoldInteger h )  {
    pHold = h;
  }
  public void run()  {
    for ( int count = 0; count < 10; count++ ) {
      // sleep for a random interval
      try {
        Thread.sleep( (int) ( Math.random() * 3000 ) );
      }
      catch( InterruptedException e ) {
        System.err.println( e.toString() );
      }
```

```
            pHold.setSharedInt( count );
             System.out.println( "Producer set sharedInt to " + count );
          }
          pHold.setMoreData( false );
        }
      }
```


CONSUMER

```
class ConsumeInteger extends Thread {
  private HoldInteger cHold;
  public ConsumeInteger( HoldInteger h )  {
    cHold = h;
  }
  public void run()  {
    int val;
    while ( cHold.hasMoreData() ) {
      // sleep for a random interval
      try {
        Thread.sleep( (int) ( Math.random() * 3000 ) );
      }
      catch( InterruptedException e ) {
        System.err.println( e.toString() );
      }
```

```
            val = cHold.getSharedInt();
            System.out.println( "Consumer retrieved " + val );
          }
        }
      }
```



```
class HoldInteger {
  private int sharedInt[] = { -1, -1, -1, -1, -1 };
  private boolean moreData = true;
  private boolean writeable = true;
  private boolean readable = false;
  private int readLoc = 0, writeLoc = 0;
  private TextArea output;

  public HoldInteger( TextArea out )
  {
    output = out;
  }
```

```
public synchronized void setSharedInt( int val ) {
    while ( !writeable ) {
      try {
        output.append( " WAITING TO PRODUCE " + val );
        wait();
      }
      catch ( InterruptedException e ) {
        System.err.println( e.toString() );
      }
    }
    sharedInt[ writeLoc ] = val;
    readable = true;
    output.append( "\nProduced " + val +" into cell " + writeLoc );
    writeLoc = ( writeLoc + 1 ) % 5;
    output.append( "\twrite " + writeLoc +"\tread " + readLoc );
    printBuffer( output, sharedInt );
    if ( writeLoc == readLoc ) {
      writeable = false;
      output.append( "\nBUFFER FULL" );
    }
    notify();
  }
```



```
public synchronized int getSharedInt()  {
    int val;
    while ( !readable ) {
      try {
        output.append( " WAITING TO CONSUME" );
        wait();
      }
      catch ( InterruptedException e ) {
        System.err.println( e.toString() );
      }
    }
    writeable = true;
    val = sharedInt[ readLoc ];
    output.append( "\nConsumed " + val + " from cell " + readLoc );
    readLoc = ( readLoc + 1 ) % 5;
    output.append( "\twrite " + writeLoc + "\tread " + readLoc );
    printBuffer( output, sharedInt );
    if ( readLoc == writeLoc ) {
      readable = false;
      output.append( "\nBUFFER EMPTY" );
    }
    notify();
    return val;
  }
```

```
public void printBuffer( TextArea out, int buf[] )
  {
    DecimalFormat threeChars = new DecimalFormat( " #;-#" );
    output.append( "\tbuffer: " );

    for ( int i = 0; i < buf.length; i++ )
      out.append( " " + threeChars.format( buf[ i ] ) );
  }
  public void setMoreData( boolean b ) { moreData = b; }
  public boolean hasMoreData()  {
    if ( moreData == false && readLoc == writeLoc )
      return false;
    else
      return true;
  }
}
```


The `suspend` and `resume` Methods

- Have been deprecated in JDK 1.2
- Should be replaced with wait() and notify()

The `stop` Method

- Releases the lock before it terminates
- Can leave shared data in an inconsistent state
- Should be replaced with wait() and notify()


```
public class ControlledThread extends Thread {
        static final int SUSP=1;
        static final int STOP=2;
        static final int RUN=0;
        private int state = RUN;
        public synchronized void setState( int s){
                state = s;
                if (s == RUN)  notify();
        }
        public synchronized boolean checkState() {
            while(state == SUSP) {
                try {
                  wait();
                } catch (InterruptedException e) { }
            }
            if (state == STOP)  return false;
           return true;                                    }
```

```
public void run() {
        while(true) {
            doSomething();
            // be sure shared data is in
            // consistent state in case the
            // thread is waited or marked for
            // exiting from run().
            if (!checkState())      break;
        }
    }//of run
}//of producer
```

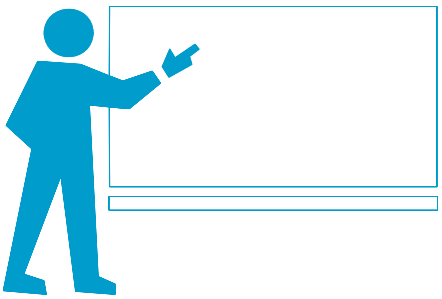

Thread Groups

- It is sometimes useful to identify various threads as belonging to a thread group
- Class ThreadGroup contains the methods for creating and manipulating thread groups
- public ThreadGroup(String groupName)
- public ThreadGroup(ThreadGroup parentThreadGroup, String groupName)
- public Thread(ThreadGroup threadGroup, Runnable runnableObject)
- public Thread(ThreadGroup threadGroup, Runnable runnableObject, String threadName)

ThreadGroup Methods

- activeCount() : reports the number of active threads in a thread group
- getMaxPriority() : return the maximum priority
- setMaxPriority() : sets a new maximum priority
- getName() : returns a String – **ThreadGroup**'s name
- getParent() : determines the parent
- parentOf()
- stop() : stops every Thread in the **ThreadGroup**
- destroy() : destroys a **ThreadGroup** and its child **ThreadGroup**s

Practice Session



Exercise # 1

- Open TicTacToe source
- Add new menu item to the Menu “Game” : “New Window”
- addActionListener to the Menu Item
- make modifications on the source code so that it can create one new game window when the menu item is selected

Exercise # 2

- Write a parallel search algorithm which finds a specified value in an array.
- Divide-and-Conquer !

```
public class SearchApp implements ActionListener {
    private Frame fr ;
    private TextArea ta ;
        private TextField tf ;
        private Button b   ;
        private int searchValue ;
        public SearchApp() {
            fr = new Frame("Search thread example") ;
          tf = new TextField() ;
          b = new Button("Search") ;
          b.addActionListener(this) ;
          tf.addActionListener(this) ;
          ta = new TextArea("",10,10) ;
          fr.setLayout(new GridLayout(3,1)) ;
          fr.add(tf) ;
          fr.add(ta) ;
          fr.add(b) ;
          fr.setSize(200,200) ;
          fr.setVisible(true) ;
        }
```



```
public void actionPerformed( ActionEvent e ) {
    if( e.getSource() == tf )
        searchValue = Integer.parseInt( tf.getText() ) ;
            else {
              SyncArray sa = new SyncArray(1000,10) ;
              Search[]  st = new Search[10] ;
              ta.setText("") ;
              for(int i=0;i<10;i++)
                st[i] = new Search(searchValue,sa,ta)  ;
              for(int i=0;i<10;i++)
                st[i].start();
                    try {
                       for(int i=0;i<10;i++)
                           st[i].join();
                    }catch(InterruptedException ev)  {}
              ta.append("Finished...")  ;
            }
    }
```

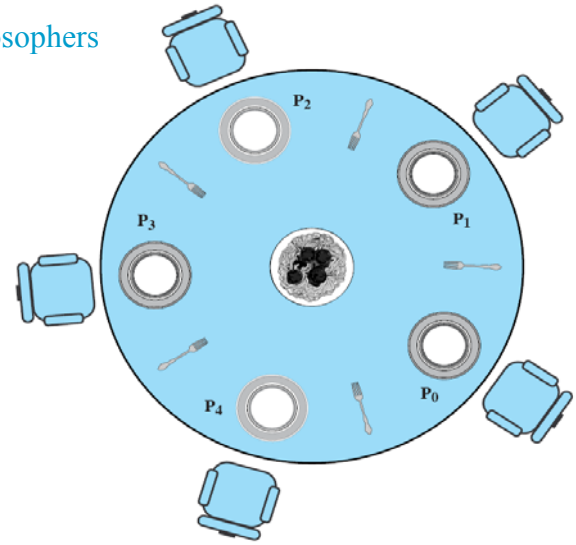
```
public class Search extends Thread {
    private SyncArray s   ;
    private TextArea  ta  ;
    private int searchValue ;
    public Search(int searchValue,SyncArray t,TextArea ta) {
        s = t ;
        this.ta = ta ;
        this.searchValue = searchValue ;
    }
    public void run() {
        int[] a ;
        int result=0,i=0 ;
        a = new int[2] ;
        a = s.getIndex() ;
        ta.append("Thread " + getName() + " is running\n") ;
        for(i=a[0];i<(a[0]+a[1]);i++) {
            result = s.compare(i,searchValue) ;
            if( result == SyncArray.FOUND ) break ;
            if( result == i ) break ;
            }
```



```
        if( result == i )
            ta.append(getName() + " found the search value.\n") ;
        else
            ta.append(getName() + " ends.\n") ;
    }
}
```

Exercise # 3

- Dining Philosophers




```
public class DiningPhilosopher {
    public static void main(String[] args) {
            Fork f = new Fork() ;
            philosopher p = new philosopher(f) ;

            Thread phi1 = new Thread(p,"0") ;
            Thread phi2 = new Thread(p,"1") ;
            Thread phi3 = new Thread(p,"2") ;
            Thread phi4 = new Thread(p,"3") ;
            Thread phi5 = new Thread(p,"4") ;

            phi1.start();
            phi2.start();
            phi3.start();
            phi4.start();
            phi5.start();
        }
}
```

```
public class philosopher implements Runnable {
    private Fork f;
    public philosopher(Fork f) {
        this.f = f ;
    }
    public void run() {
        int sleepTime ;
        int philosopherID ;
        philosopherID = getID() ;
        System.out.println("This is  philosopher " +
                                philosopherID + " thinking...") ;
        // thinking
        sleepTime = (int) (Math.random() * 5000.0) ;
        try {
           Thread.sleep(sleepTime) ;
        } catch(  InterruptedException e) {        }

        f.pickForks(philosopherID) ;
```



```
        // eat
        System.out.println("This is  philosopher " +
                                philosopherID + " eating...") ;
        sleepTime = (int) (Math.random() * 5000.0) ;
        try {
          Thread.sleep(sleepTime) ;
        } catch(  InterruptedException e) {}

        f.releaseForks(philosopherID) ;
    }
    public int getID(){
            String s = Thread.currentThread().getName() ;
            return Integer.parseInt(s) ;
    }
}
```

```
public class Fork{
        private int waitingToEnter = 0 ;
        private int[] waitingToPickFork ;
        private int[] fork ;
        public Fork(){
                fork = new int[5] ;
                waitingToPickFork = new int[5] ;
                for(int i=0;i<fork.length;i++) {
                        fork[i] = 1 ;
                        waitingToPickFork[i] = 0 ;
                }
        }
```



```
public synchronized void pickForks(int forkNumber){
        // wait for forks
        while ( fork[forkNumber] == 0 )
            try {
              waitingToPickFork[forkNumber]++ ;
              wait() ;
              waitingToPickFork[forkNumber]-- ;
            } catch(  InterruptedException e) {}
      while ( fork[(forkNumber+1) % 5 ] == 0 )
         try {
          waitingToPickFork[(forkNumber+1) % 5 ]++ ;
           wait() ;
           waitingToPickFork[(forkNumber+1) % 5 ]-- ;
         } catch(  InterruptedException e) {}
      fork[forkNumber] = 0 ;
      fork[(forkNumber+1) % 5] = 0 ;
}
```

```
    public synchronized void releaseForks(int forkNumber) {
        fork[forkNumber] = 1  ;
        if( waitingToPickFork[forkNumber] > 0 )
          notify() ;
        fork[(forkNumber+1) % 5] = 1  ;
        if( waitingToPickFork[(forkNumber+1) % 5] > 0 )
          notify() ;
    }
}
```


14 Advanced IO Stream

Advanced I/O Streams

Objectives

- Describe and use streams philosophy of the java.io package
- Construct file and filter streams, and use them appropriately
- Distinguish readers and writers from streams, and select appropriately between them.
- Examine and manipulate files and directories
- Read, write, and update text and data files
- Use the Serialization interface to persist the state of objects

I/O Fundamentals

- A *stream* can be thought of as a flow of data from a source or to a sink.
- A *source* stream initiates the flow of data, also called an input stream.
- A *sink* stream terminates the flow of data, also called an output stream.
- Sources and sinks are both *node streams*.
- Types of node streams are files, memory, and pipes between threads or processes.

	Byte Streams	Character Streams
Source Streams	`InputStream`	`Reader`
Sink Streams	`OutputStream`	`Writer`

- You can read from an input stream, but you cannot write to it.
- Conversely, you can write to an output stream, butyou cannot read from it.

Data Within Streams

- Java technology supports wo types of streams: character and byte.
- Input or output of character data is handled by readers and writers.
- Input and output of byte data is handled by input streams and output streams:
  - Normally, the term stream refers to a byte stream
  - The terms reader and writer refer to character streams

`InputStream` Methods

- The three basic read() methods
  - int read()
  - int read(byte[])
  - int read(byte[], int, int)
- The other methods
  - void close()
  - int available()
  - skip(long)
  - boolean markSupported()
  - void mark(int)
  - void reset(int)

`OutputStream` Methods

- The three basic write() methods
  - void write(int)
  - void write(byte[])
  - void write(byte[], int, int)
- The other methods
  - void close()
  - void flush()

`Reader` Methods

- The three basic `read` methods
  - int read()
  - int read(char[] cbuf)
  - int read(char[] cbuf, int offset, int length)
- The other methods
  - void close()
  - boolean ready()
  - skip(long)
  - boolean markSupported
  - void mark(int readAheadLimit)
  - void reset(int)

`Writer` Methods

- The three basic `write` methods
  - void write(int c)
  - void write(char[] cbuf)
  - void write(char[] cbuf, int offset, int length)
  - void write(String string)
  - void write(String string, int offset, int length)
- The other methods
  - · void close()
  - · void flush()

- Object
- File
- FileDescriptor
- StreamTokenizer
- InputStream
  - ByteArrayInputStream
  - SequenceInputStream
  - StringBufferInputStream
  - PipedInputStream
  - FileInputStream
  - FilterInputStream
    - DataInputStream
    - BufferedInputStream
    - PushBackInputStream
    - LineNumberInputStream

- Object
- File
- FileDescriptor
- StreamTokenizer
- OutputStream
  - ByteArrayOutputStream
  - PipedOutputStream
  - FileOutputStream
  - FilterOutputStream
    - DataOutputStream
    - BufferedOutputStream
    - PrintStream
- RandomAccessFile

- InputStream and OutputStream are abstract classes that define methods for performing input and output respectively. Their derived classes override these methods
- File input/output is done with FileInputStream and FileOutputStream.
- Pipes are synchronized communication channels between threads. A pipe is established bteween two threads. One thread sends data to another by writing to a PipedOutputStream. The target thread reads information from the pipe via a PipedInputStream.
- A PrintStream is used for performing output to the screen. System.out and System.err are PrintStream.

- FilterInputStream filters an InputStream and FilterOutputStream filters OutputStream. Filtering simply means that the filter stream provides additional functionality such as bufferin, monitoring line numbers or aggregating datat bytes into meaninful primitive data types.
- Reading data as raw bytes is fast but crude. Usually programs want to read data as aggregates of bytes that form an int, a float, a double, and so on. To accomplish this we use a DataInputStream.
- DataInput interface is implemented by class DataInputStream and class RandomAccessFile that each need to read primitive data types from a stream. DataInputStreams enable a program to read binary data from an InputStream.

- DataOutput interface is implemented by class DataOutputStream and class RandomAccessFile that each need to read primitive data types from a stream. DataInputStreams enable a program to write binary data from an InputStream.

- Buffering is an I/O-performance enhancement technique. With a BufferedOutputStream each output statement does necessarily result in an actual physical transfer of data to the output device. rather, each output operation is directed to a region in memory called a buffer that is large enough to hold the data of many output operations. Than actual output to the output device is performed in one large pysical output operation each time the buffer fills.

- With a BufferedInputStream many logical chunks of data from a file are read as one large physical input operation into a memory buffer. As a program requests each new chunk of data, it is taken fom the buffer. When the buffer becomes empty, the next actual physical input operation from the input device is performed .

- With a BufferOutputStream a partially filled buffer can be forced out to the device at any time with an explicit flush as follows:

```
testBufferedOutputStream.flush() ;
```

- RndomAccessFile is useful for direct-access applications such as tarnsaction-processing applications. With a sequential-access file each successive input/output request reads or writes the next consecutive set of data in the file. With a random-access file, each successive input/output request may be directed to any part of the file. Direct-access applications provide rapid access to specific data items in large files.

- Java stream I/O includes capabilities for inputting from byte arrays in memory and outputting to byte arrays in memory. A ByteArrayInputStream performs its inputs fom a byte array in memory. A ByteArrayOutputStream outputs to a byte array in memory.

- A StringBufferInputStream inputs from a StringBuffer object.
- A SequenceInputStream enables several InputStreams to be concatenated so that the program sees the group as one continuous InputStream. As the end of each input stream is reached, the stream is closed and the next stream in the sequence is opened.
- A LineNumberInputStream always knows what line number of the file is being read.

Creating a Sequential-Access File

- Java imposes no structure on a file.
- Thus, notions like “record” do not exist in Java files.
- Therefore, the programmer must structure files to meet the requirements of applications.

```
public class CreateSequentialFile implements ActionListener {
  // TextFields where user enters account number, first name,
  // last name and balance.
  private JFrame fr ;
  private JTextField accountField, firstNameField,
                     lastNameField, balanceField;

  private JButton enter,  // send record to file
                  done;   // quit program

  // Application other pieces
  private DataOutputStream output;
```



```
public CreateSequentialFile()  {
  fr = new JFrame( "Create Client File" );
  // Open the file
  try {
    output = new DataOutputStream(
             new FileOutputStream( "client.dat" ) );
  }
  catch ( IOException e ) {
    System.err.println( "File not opened properly\n" + e.toString() );
    System.exit( 1 );
  }
  fr.getContentPane().setLayout( new GridLayout( 5, 2 ) );
  // create the components of the Frame
  fr.getContentPane().add( new JLabel( "Account Number" ) );
  accountField = new JTextField();
  fr.getContentPane().add( accountField );
```

```
        fr.getContentPane().add( new JLabel( "First Name" ) );
        firstNameField = new JTextField( 20 );
        fr.getContentPane().add( firstNameField );
        fr.getContentPane().add( new JLabel( "Last Name" ) );
        lastNameField = new JTextField( 20 );
        fr.getContentPane().add( lastNameField );
        fr.getContentPane().add( new JLabel( "Balance" ) );
        balanceField = new JTextField( 20 );
        fr.getContentPane().add( balanceField );
        enter = new JButton( "Enter" );
        enter.addActionListener( this );
        fr.getContentPane().add( enter );
        done = new JButton( "Done" );
        done.addActionListener( this );
        fr.getContentPane().add( done );
        fr.setSize( 300, 150 );
        fr.show();
    }
```



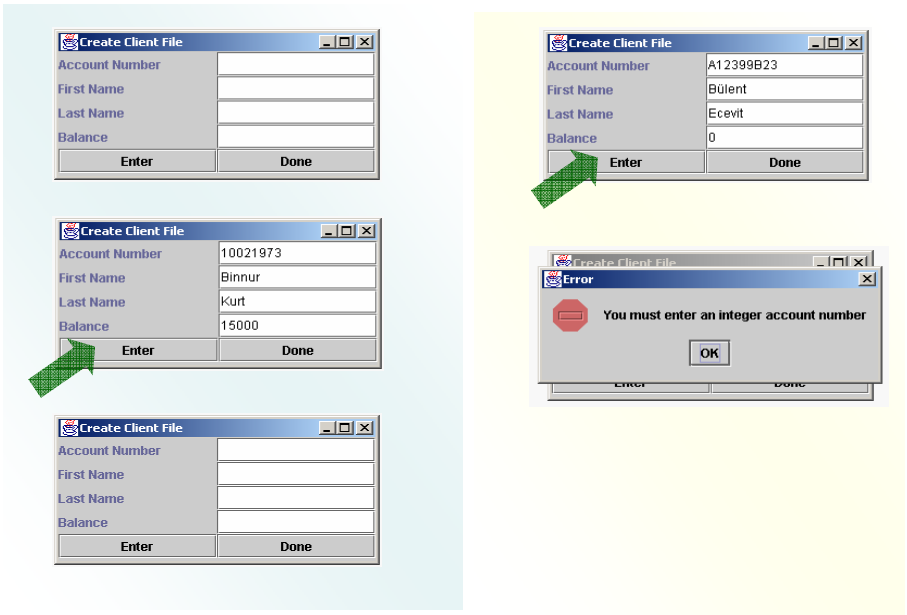
```
    public void addRecord()   {
        int accountNumber = 0;
        Double d;
        if ( ! accountField.getText().equals( "" ) ) {
            // output the values to the file
            try {
                accountNumber =  Integer.parseInt( accountField.getText() );
                if ( accountNumber > 0 ) {
                    output.writeInt( accountNumber );
                    output.writeUTF( firstNameField.getText() );
                    output.writeUTF( lastNameField.getText() );
                    d = new Double ( balanceField.getText() );
                    output.writeDouble( d.doubleValue() );
                }
```

```
            // clear the TextFields
            accountField.setText( "" );
            firstNameField.setText( "" );
            lastNameField.setText( "" );
            balanceField.setText( "" );
         }
         catch ( NumberFormatException nfe ) {
            JOptionPane.showMessageDialog( fr,"You must enter an
      integer account number" , "Error“ ,JOptionPane.ERROR_MESSAGE ) ;
         }
         catch ( IOException io ) {
            JOptionPane.showMessageDialog( fr,
      "Error during write to file" ,"Error" ,JOptionPane.ERROR_MESSAGE ) ;
            System.exit( 1 );
         }
      }
   }
```



```
   public void actionPerformed( ActionEvent e )   {
      addRecord();
      if ( e.getSource() == done ) {
         try {
            output.close();
         }
         catch ( IOException io ) {
         JOptionPane.showMessageDialog( fr,
                        "File not closed properly" + io.toString(),"Error" ,
            JOptionPane.ERROR_MESSAGE ) ;
         }
         System.exit( 0 );
      }
   }
```

```
    public static void main( String args[] )
    {
      CreateSequentialFile csf = new CreateSequentialFile();
    }
}
```

Reading Data from a Sequential-Access File

```
public class ReadSequentialFile implements ActionListener {
  //  TextFields to display accont number, first name
  // last name and balance.
  private JFrame fr ;
  private JTextField accountField, firstNameField,
                     lastNameField, balanceField;
  private JButton enter,  // get next record in file
                  done;   // quit program
  // Application other pieces
  private DataInputStream input;
```



```
public ReadSequentialFile()  {
  fr = new JFrame( “Read Client File" );
  // Open the file
  try {
    input = new DataInputStream(
            new FileInputStream( "client.dat" ) );
  }
  catch ( IOException e ) {
    System.err.println( "File not opened properly\n" + e.toString() );
    System.exit( 1 );
  }
  fr.getContentPane().setLayout( new GridLayout( 5, 2 ) );
  // create the components of the Frame
  fr.getContentPane().add( new JLabel( "Account Number" ) );
  accountField = new JTextField();
  fr.getContentPane().add( accountField );
```

```
        fr.getContentPane().add( new JLabel( "First Name" ) );
        firstNameField = new JTextField( 20 );
        fr.getContentPane().add( firstNameField );
        fr.getContentPane().add( new JLabel( "Last Name" ) );
        lastNameField = new JTextField( 20 );
        fr.getContentPane().add( lastNameField );
        fr.getContentPane().add( new JLabel( "Balance" ) );
        balanceField = new JTextField( 20 );
        fr.getContentPane().add( balanceField );
        next = new JButton( “Next" );
        next.addActionListener( this );
        fr.getContentPane().add( next );
        done = new JButton( "Done" );
        done.addActionListener( this );
        fr.getContentPane().add( done );
        fr.setSize( 300, 150 );
        fr.show();
    }
```



```
public void actionPerformed( ActionEvent e )
{
  if ( e.getSource() == next )
    readRecord();
  else
    closeFile();
}
```

```
public void readRecord()   {
  int account;
  String first, last;
  double balance;
  // input the values from the file
  try {
    account = input.readInt();
    first = input.readUTF();
    last = input.readUTF();
    balance = input.readDouble();
    accountField.setText( String.valueOf( account ) );
    firstNameField.setText( first );
    lastNameField.setText( last );
    balanceField.setText( String.valueOf( balance ) );
  }
```



```
  catch ( EOFException eof ) {
    closeFile();
  }
  catch ( IOException e ) {
      JOptionPane.showMessageDialog( fr,
      "Error during read from file "+e.toString() ,"Error" ,
      JOptionPane.ERROR_MESSAGE ) ;
      System.exit( 1 );
  }
}
```

```
  private void closeFile()  {
    try {
      input.close();
      System.exit( 0 );
    }
    catch ( IOException e ) {
       JOptionPane.showMessageDialog( fr,
                   "Error closing file" ,"Error" ,
       JOptionPane.ERROR_MESSAGE ) ;
      System.exit( 1 );
    }
  }
  public static void main( String args[] )  {
    ReadSequentialFile rsf = new ReadSequentialFile();
  }
}
```


Searching for a Record in a Sequential-Access File

```
public class CreditInquiry implements ActionListener {
  // application window components
  private JFrame fr ;
  private JTextArea recordDisplay;
  private JButton done, credit, debit, zero;
  private JPanel buttonPanel;
  private RandomAccessFile input;
  private String accountType;
  public CreditInquiry()  {
    fr = new JFrame( "Credit Inquiry Program" );
    // Open the file
    try {
      input = new RandomAccessFile( "client.dat", "r" );
    }
    catch ( IOException e ) {
      System.err.println( e.toString() );
      System.exit( 1 );
    }
```

```
        fr.setSize( 400, 150 );
        // create the components of the Frame
        buttonPanel = new JPanel();
        credit = new JButton( "Credit balances" );
        credit.addActionListener( this );
        buttonPanel.add( credit );
        debit = new JButton( "Debit balances" );
        debit.addActionListener( this );
        buttonPanel.add( debit );
        zero = new JButton( "Zero balances" );
        zero.addActionListener( this );
        buttonPanel.add( zero );
        done = new JButton( "Done" );
        done.addActionListener( this );
        buttonPanel.add( done );
        recordDisplay = new JTextArea( 4, 40 );
        // add the components to the Frame
        fr.getContentPane().add( recordDisplay, BorderLayout.NORTH );
        fr.getContentPane().add( buttonPanel, BorderLayout.SOUTH );

        fr.show();
    }
```



```
    public void actionPerformed( ActionEvent e )  {
      if ( e.getSource() != done ) {
        accountType = e.getActionCommand();
        readRecords();
      }
      else {      // Close the file
        try {
          input.close();
          System.exit( 0 );
        }
        catch ( IOException ioe ) {
            JOptionPane.showMessageDialog( fr,
             "File not closed properly " + ioe.toString() ,"Error" ,
             JOptionPane.ERROR_MESSAGE ) ;
          System.exit( 1 );
        }
      }
    }
```

```
public void readRecords() {
  int account;
  String first, last;
  double balance;
  DecimalFormat twoDigits = new DecimalFormat( "0.00" );
  // input the values from the file
  try {      // to catch IOException
    try {   // to catch EOFException
      recordDisplay.setText( "The accounts are:\n" );
      while ( true ) {
        account = input.readInt();
        first = input.readUTF();
        last = input.readUTF();
        balance = input.readDouble();
```



```
        if ( shouldDisplay( balance ) )
          recordDisplay.append( account + "\t" +
            first + "\t" + last + "\t" +
            twoDigits.format( balance ) + "\n" );
      }
    }
    catch ( EOFException eof ) {
      input.seek( 0 );
    }
  }
  catch ( IOException e ) {
    JOptionPane.showMessageDialog( fr,
    "Error during read from file " + e.toString() ,"Error" ,
    JOptionPane.ERROR_MESSAGE ) ;
    System.exit( 1 );
  }
}
```

```
public boolean shouldDisplay( double balance )
 {
   if ( accountType.equals( "Credit balances" ) &&
        balance < 0 )
     return true;
   else if( accountType.equals("Debit balances") &&
            balance > 0 )
     return true;
   else if ( accountType.equals( "Zero balances" ) &&
              balance == 0 )
     return true;

   return false;
 }
```



```
  // Instantiate a CreditInquiry object and start the program
   public static void main( String args[] )
   {
     CreditInquiry ci = new CreditInquiry();
   }
 }
```

Updating Sequential-Access Files

- Updating can be done, but it is awkward.
- First it should be copied to a new file
- It is updated there, and copied itself again.

Random-Access Files

- Sequential-access files are inappropriate for "instant-access" applications in which a particular record of information must be located immediately.
- Java imposes no structure on a file.
- Thus, notions like "record" do not exist in Java files.
- Therefore, the programmer must structure files to meet the requirements of applications.
- A variety of techniques can be used : "File Organization"
- Simplest one : all records in a file are of the same fixed length.

```
public class Record {
   private int account;
   private String lastName;
   private String firstName;
   private double balance;
   // Read a record from the specified RandomAccessFile
   public void read( RandomAccessFile file ) throws IOException {
      account = file.readInt();
      char first[] = new char[ 15 ];
      for ( int i = 0; i < first.length; i++ )
         first[ i ] = file.readChar();
      firstName = new String( first );
      char last[] = new char[ 15 ];
      for ( int i = 0; i < last.length; i++ )
         last[ i ] = file.readChar();
      lastName = new String( last );
      balance = file.readDouble();
   }
```



```
// Write a record to the specified RandomAccessFile
public void write( RandomAccessFile file ) throws IOException   {
   StringBuffer buf;
   file.writeInt( account );
   if ( firstName != null )
      buf = new StringBuffer( firstName );
   else
      buf = new StringBuffer( 15 );
   buf.setLength( 15 );
   file.writeChars( buf.toString() );
   if ( lastName != null )
      buf = new StringBuffer( lastName );
   else
      buf = new StringBuffer( 15 );
   buf.setLength( 15 );
   file.writeChars( buf.toString() );
   file.writeDouble( balance );
}
```

```
  public void setAccount( int a ) { account = a; }
  public int getAccount() { return account; }
  public void setFirstName( String f ) { firstName = f; }
  public String getFirstName() { return firstName; }
  public void setLastName( String l ) { lastName = l; }
  public String getLastName() { return lastName; }
  public void setBalance( double b ) { balance = b; }
  public double getBalance() { return balance; }
  // NOTE: This method contains a hard coded value for the
  // size of a record of information.
  public static int size() { return 72; }
}
```


Creating a Random-Access Files

```
public class WriteRandomFile implements ActionListener {
  // TextFields where user enters account number, first name,
  // last name and balance.
  private JFrame fr ;
  private JTextField accountField, firstNameField,
                     lastNameField, balanceField;
  private JButton enter,  // send record to file
                  done;  // quit program
  // Application other pieces
  private RandomAccessFile output;  // file for output
  private Record data;
```

```
public WriteRandomFile()   {
  fr = new JFrame( “Write Client File" );
  // Open the file
  try {
     output = new RandomAccessFile("client.dat“,”rw” );
  }
  catch ( IOException e ) {
    System.err.println( e.toString() );
    System.exit( 1 );
  }
  fr.getContentPane().setLayout( new GridLayout( 5, 2 ) );
  // create the components of the Frame
  fr.getContentPane().add( new JLabel( "Account Number" ) );
  accountField = new JTextField();
  fr.getContentPane().add( accountField );
```



```
  fr.getContentPane().add( new JLabel( "First Name" ) );
  firstNameField = new JTextField( 20 );
  fr.getContentPane().add( firstNameField );
  fr.getContentPane().add( new JLabel( "Last Name" ) );
  lastNameField = new JTextField( 20 );
  fr.getContentPane().add( lastNameField );
  fr.getContentPane().add( new JLabel( "Balance" ) );
  balanceField = new JTextField( 20 );
  fr.getContentPane().add( balanceField );
  next = new JButton( “Next" );
  next.addActionListener( this );
  fr.getContentPane().add( next );
  done = new JButton( "Done" );
  done.addActionListener( this );
  fr.getContentPane().add( done );
  fr.setSize( 300, 150 );
  fr.show();
}
```

```
public void actionPerformed( ActionEvent e )
{
  addRecord();

  if ( e.getSource() == done ) {
    try {
      output.close();
    }
    catch ( IOException io ) {
      JOptionPane.showMessageDialog( fr,
      "File not closed properly " + io.toString() ,"Error" ,
      JOptionPane.ERROR_MESSAGE ) ;
    }
    System.exit( 0 );
  }
}
```



```
public void addRecord()   {
  int accountNumber = 0;
  Double d;
  if ( ! accountField.getText().equals( "" ) ) {
    // output the values to the file
    try {
      accountNumber =
        Integer.parseInt( accountField.getText() );
      if ( accountNumber > 0 && accountNumber <= 100 ) {
        data.setAccount( accountNumber );
        data.setFirstName( firstNameField.getText() );
        data.setLastName( lastNameField.getText() );
        d = new Double ( balanceField.getText() );
        data.setBalance( d.doubleValue() );
        output.seek(
          (long) ( accountNumber-1 ) * Record.size() );
        data.write( output );
      }
```

```
            // clear the TextFields
            accountField.setText( "" );
            firstNameField.setText( "" );
            lastNameField.setText( "" );
            balanceField.setText( "" );
         }
         catch ( NumberFormatException nfe ) {
            JOptionPane.showMessageDialog( fr,
            "You must enter an integer account number","Error" ,
            JOptionPane.ERROR_MESSAGE ) ;
         }
         catch ( IOException io ) {
            JOptionPane.showMessageDialog( fr,
            "Error during write to file " + io.toString() ,"Error" ,
            JOptionPane.ERROR_MESSAGE ) ;
            System.exit( 1 );
         }
      }
   }
```


Reading Data Sequentially from a Random-Access File

```
public class ReadRandomFile implements ActionListener {
   // TextFields to display account number, first name,
   // last name and balance.
   private JFrame fr ;
   private JTextField accountField, firstNameField,
                      lastNameField, balanceField;
   private JButton next,   // get next record in file
                   done;   // quit program
   // Application other pieces
   private RandomAccessFile input;
   private Record data;
```

```
public ReadRandomFile()   {
  fr = new JFrame( “Read Client File" );
  // Open the file
  try {
     input = new RandomAccessFile("client.dat“,”r” );
  }
  catch ( IOException e ) {
    System.err.println( "File not opened properly\n" + e.toString() );
    System.exit( 1 );
  }
  data = new Record ;
  fr.getContentPane().setLayout( new GridLayout( 5, 2 ) );
  // create the components of the Frame
  fr.getContentPane().add( new JLabel( "Account Number" ) );
  accountField = new JTextField();
  fr.getContentPane().add( accountField );
```



```
  fr.getContentPane().add( new JLabel( "First Name" ) );
  firstNameField = new JTextField( 20 );
  fr.getContentPane().add( firstNameField );
  fr.getContentPane().add( new JLabel( "Last Name" ) );
  lastNameField = new JTextField( 20 );
  fr.getContentPane().add( lastNameField );
  fr.getContentPane().add( new JLabel( "Balance" ) );
  balanceField = new JTextField( 20 );
  fr.getContentPane().add( balanceField );
  next = new JButton( “Next" );
  next.addActionListener( this );
  fr.getContentPane().add( next );
  done = new JButton( "Done" );
  done.addActionListener( this );
  fr.getContentPane().add( done );
  fr.setSize( 300, 150 );
  fr.show();
}
```

```
public void actionPerformed( ActionEvent e )
{
  if ( e.getSource() == next )
    readRecord();
  else
    closeFile();
}
```



```
public void readRecord()  {
  DecimalFormat twoDigits = new DecimalFormat( "0.00" );
  // read a record and display
  try {
    do {
      data.read( input );
    } while ( data.getAccount() == 0 );
    accountField.setText( String.valueOf( data.getAccount() ) );
    firstNameField.setText( data.getFirstName() );
    lastNameField.setText( data.getLastName() );
    balanceField.setText( String.valueOf(
    twoDigits.format( data.getBalance() ) ) );
  }
  catch ( EOFException eof ) {
    closeFile();
  }
  catch ( IOException e ) {
    System.err.println( "Error during read from file\n" + e.toString() );
    System.exit( 1 );
  }
}
```

```
private void closeFile()
{
  try {
    input.close();
    System.exit( 0 );
  }
  catch ( IOException e ) {
    JOptionPane.showMessageDialog( fr,
    "Error closing file " + e.toString() ,"Error" ,
    JOptionPane.ERROR_MESSAGE ) ;
    System.exit( 1 );
  }
}
```


A Transaction-Processing Program

15 Networking

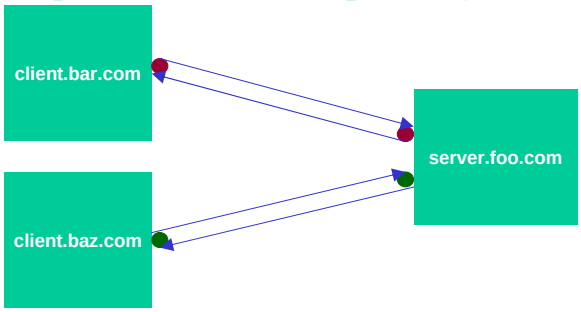
Networking

Objectives

- Develop code to set up the network connection
- Understand the TCP/IP protocol
- Use ServerSocket and Socket classes for implementing TCP/IP clients and servers

Networking

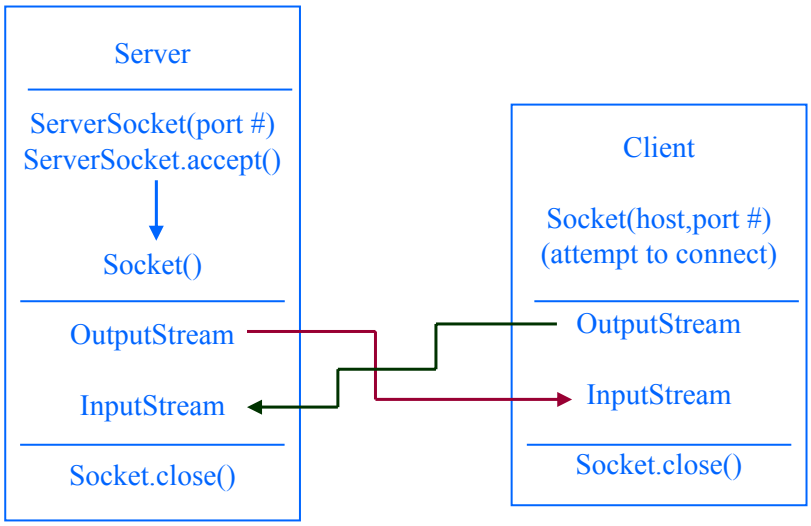
- Sockets
  - Sockets hold two streams
- Setting up the connection
  - Set up is similar to a telephone system



Networking with Java Technology

- Addressing the connection
  - Address or name of remote machine
  - Port number to identify purpose
- Port numbers
  - Range from 0 - 65535

Java Networking Model



Minimal TCP/IP Server

```
import java.net.*;
import java.io.*;
public class SimpleServer {
 public static void main(String args[]) {
                ServerSocket s = null;
                Socket s1;
                String sendString = "Hello Net World!";
                OutputStream s1out;
                DataOutputStream dos;
                // Register your service on port 5432
                try {
                        s = new ServerSocket(5432);
                } catch (IOException e) { }
```

```
                // Run the listen/accept loop forever
                while (true) {
                   try {
                        // Wait here and listen for a connection
                        s1=s.accept();
                       // Get a communication stream associated with the socket
                        s1out = s1.getOutputStream();
                        dos = new DataOutputStream (s1out);
                      // Send your string!
                      dos.writeUTF(sendString);
                       // Close the connection, but not the server socket
                       s1out.close();
                       s1.close();
                    } catch (IOException e) { }
                }
            }
        }
```



```
import java.net.*;
import java.io.*;
public class SimpleClient {
      public static void main(String args[]) throws IOException {
            int c;
            Socket s1;
            InputStream s1In;
            DataInputStream dis;
            // Open your connection to sunbert, at port 5432
            s1 = new Socket("127.0.0.1",5432);
            // The above statement could potentially throw an IOException.
            // Get an input file handle from the socket and read the input
            s1In = s1.getInputStream();
            dis = new DataInputStream(s1In);
            String st = new String (dis.readUTF());
            System.out.println(st);
            // When done, just close the connection and exit
            s1In.close();
            s1.close();
      }
}
```

16 Introduction to JDBC

Introduction to JDBC

- JDBC is a layer of abstraction that allows users to choose between databases
- JDBC allows you to write to a single API
- JDBC allows you change to a different database engine
- JDBC supports ANSI SQL-2 compatible database, but can be used on other databases

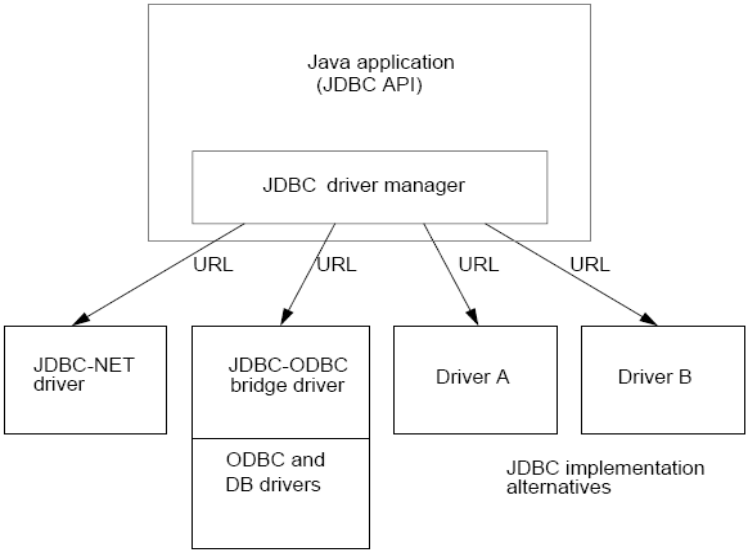
The Two Components of JDBC

- An implementation interface for database manufacturers
- An interface for application and applet writers.

JDBC Driver Interface

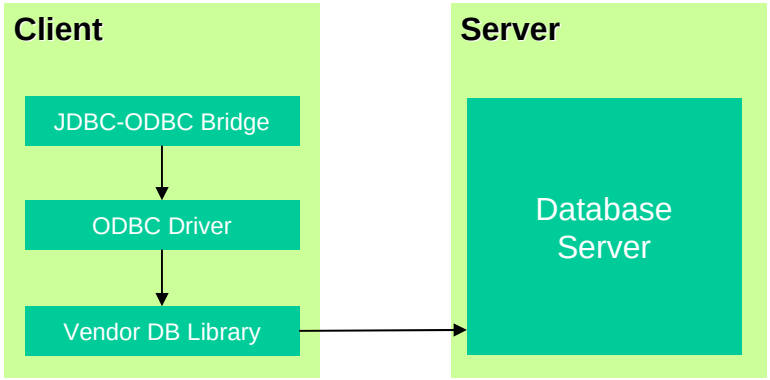
- The JDBC Driver interface provides vendor-specific
- implementations of the abstract classes provided by the JDBC API.
- Each vendor's driver must provide implementations of the following:
  - java.sql.Connection
  - java.sql.Statement
  - java.sql.PreparedStatement
  - java.sql.CallableStatement
  - java.sql.ResultSet
  - java.sql.Driver

JDBC Drivers



Type 1: JDBC-ODBC Bridge

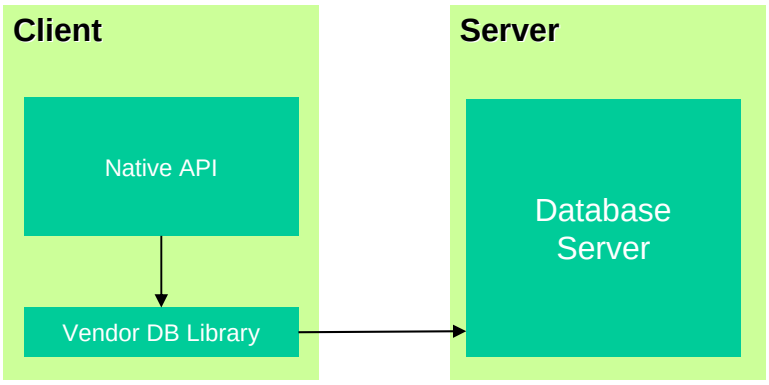
- JDBC-ODBC Bridge, translates all JDBC calls into ODBC (Open DataBase Connectivity) calls and sends them to the ODBC driver.



Type 2: Native-API/partly Java driver

- the native-API/partly Java driver converts JDBC calls into database-specific calls for databases such as SQL Server, Informix, Oracle, or Sybase.
- The type 2 driver communicates directly with the database server
- It requires that some binary code be present on the client machine.

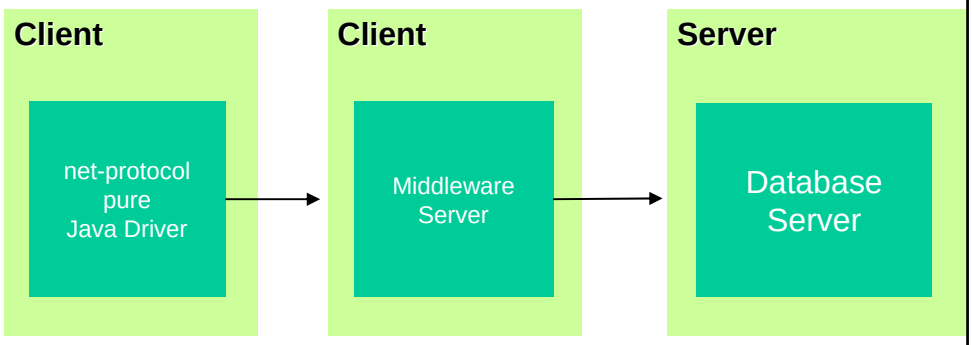
Type 2: Native-API/partly Java driver



Type 3: Net-protocol/all-Java driver

- JDBC driver type 3 follows a three-tiered approach
  - the JDBC database requests are passed through the network to the middle-tier server.
  - The middle-tier server then translates the request (directly or indirectly) to the database-specific native-connectivity interface to further the request to the database server.
  - If the middle-tier server is written in Java, it can use a type 1 or type 2 JDBC driver to do this.

Type 3: Net-protocol/all-Java driver



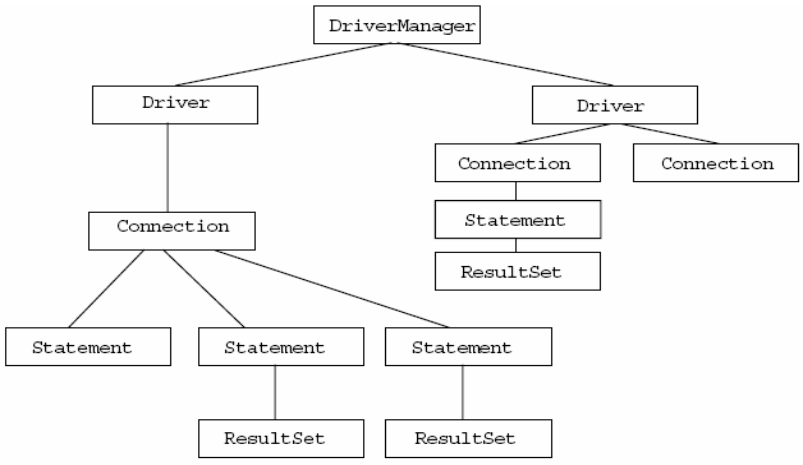
JDBC Programming Tasks

- Create an instance of a JDBC driver or load JDBC drivers through jdbc.drivers
- Register a driver
- Specify a database
- Open a database connection
- Submit a query
- Receive results

Tha java.sql Package

- java.sql.Driver
- java.sql.Connection
- java.sql.Statement
- java.sql.PreparedStatement
- java.sql.CallableStatement
- java.sql.ResultSet
- java.sql.ResultSetMetaData
- java.sql.DatabaseMetaData

JDBC Flowchart



JDBC Example

```
import java.sql.*;
import java.util.Properties;
import java.io.InputStream;

public class ConnectMe  {
  public static void main (String args[]) {
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    } catch (Exception e) {
      System.out.println("Failed to load JDBC/ODBC driver.");
      return;
    }
    try {
      Connection con = DriverManager.getConnection("jdbc:odbc:mage", "", "");
      System.out.println("Connected.");
```

Explicitly Creating an Instance of a JDBC Driver

- To communicate with a particular database engine using JDBC, you must first create an instance of the JDBC driver.
  - Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
- This driver remains behind the scenes, handling any requests for that type of database.
- You do not have to associate this driver with a variable, the driver exists after it is instantiated and successfully loaded into memory.

Opening a Database Connection


```
Connection con = DriverManager.getConnection("jdbc:odbc:mage",
    "", "");
```

- This method takes a URL string as an argument
- If a connection is established, a Connection object is returned
- The Connection object represents a session with a specific database

Submitting a Query

- To submit a standard query, get a Statement object from the Connection.createStatement method.

```
// Create a Statement object
try {
  stmt = con.createStatement();
} catch (SQLException e) {
  System.out.println (e.getMessage());
}
```


Submitting a Query (Cont'd)

- Use the Statement.executeUpdate method to submit an INSERT, UPDATE, or DELETE statement to the database.
- JDBC passes the SQL statement to the underlying database connection unaltered, it does not attempt to interpret queries.

```
// Pass a query via the Statement object
int count = stmt.executeUpdate("DELETE from
Customer WHERE ssn='999-55-6666'");
```

- The Statement.executeUpdate method returns an int, representing the number of rows affected by the INSERT, UPDATE, or DELETE statements.

Submitting a Query (Cont'd)

- Use the Statement.executeQuery method to submit a SELECT statement to the database.

```
// Pass a query via the Statement object
ResultSet rs = stmt.executeQuery("SELECT * from Customer
    order by ssn");
```

- The Statement.executeQuery method returns a ResultSet object for processing.

Receiving Results

- The result of executing a query statement is a set of rows that are accessible using a java.sql.ResultSet object.
- The rows are received in order.
- A ResultSet object keeps a cursor pointing to the current row of data and is initially positioned before its first row.
- Use the ResultSet.next() method to move between the rows of the ResultSet object.
- The first call to next makes the first row the current row, the second call makes the second row the current row, and so on.

Receiving Results (Cont'd)

- The ResultSet object provides a set of get methods that enable access to the various columns of the current row.

```
while (rs.next()) {
  System.out.println ("Customer: "+rs.getString(2));
  System.out.println ("Id: "+rs.getString(1));
  System.out.println ("");
}
```


get-xxx Methods

Method	Java Type Returned
getASCIIStream	java.io.InputStream
getBigDecimal	java.math.BigDecimal
getBinaryStream	java.io.InputStream
getBoolean	boolean
getByte	byte
getBytes	byte[]
getDate	java.sql.Date
getDouble	double
getFloat	float
getInt	int
getLong	long
getObject	Object
getShort	short
getString	java.lang.String
getTime	java.sql.Time
getTimestamp	java.sql.Timestamp
getUnicodeStream	java.io.InputStream of Unicode characters

get-xxx Methods

- The various getXXX methods can take either a column name or an index as their argument.
- It is a good idea to use an index when referencing a column.
- Column indexes start at 1.
- When using a name to reference a column, more than one column can have the same name, thus causing a conflict.

Working With Prepared Statements

- If the same SQL statements are going to be executed multiple times, it is advantageous to use a PreparedStatement object.
- A prepared statement is a precompiled SQL statement that is more efficient than calling the same SQL statement over and over.
- The PreparedStatement class extends the Statement class to add the capability of setting parameters inside of a statement.
- When declaring the PreparedStatement object, use the question mark (?) character as a placeholder for the incoming parameter.
- When passing the parameter to the statement, indicate which placeholder you are referencing by its sequential position in the statement.

Working With Prepared Statements (Cont'd)

```
Connection conn = DriverManager.getConnection(url);
.
.
java.sql.PreparedStatement stmt =
  conn.prepareStatement
  ("UPDATE table3 set m = ? WHERE x = ?");

stmt.setString(1, "Hi");
for (int i = 0; i < 10; i++) {
    stmt.setInt(2, i);
    int j = stmt.executeUpdate();
    System.out.println(j +" rows affected when i="+i);
}
```


Method	SQL Type(s)
setASCIIStream	LONGVARCHAR produced by an ASCII stream
setBigDecimal	NUMERIC
setBinaryStream	LONGVARBINARY
setBoolean	BIT
setByte	TINYINT
setBytes	VARBINARY or LONGVARBINARY (depending on the size relative to the limits on VARBINARY)
setDate	DATE
setDouble	DOUBLE
setFloat	FLOAT
setInt	INTEGER
setLong	BIGINT
setNull	NULL
setObject	The given object that is converted to the target SQL type before being sent
setShort	SMALLINT
setString	VARCHAR or LONGVARCHAR (depending on the size relative to the driver's limits on VARCHAR)
setTime	TIME
setTimestamp	TIMESTAMP
setUnicodeStream	UNICODE