

Java Workshop

Gabriel Oteniya and Tomasz Janowski

UNU-IIST

The Course

- 1) **objectives** - what do we intend to achieve?
- 2) **outline** - what content will be taught?
- 3) **resources** - what teaching resources will be available?
- 4) **organization** - duration, major activities, daily schedule

Course Objectives

- 1) refresh and reinforce the knowledge of Java technology
- 2) review selected Java APIs (libraries)
- 3) learn best practices in developing Java applications
- 4) lay foundation for learning J2EE technology
- 5) understand why Java technology is adequate for the implementation of e-government services

Course Outline

- | | | |
|---|--|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|--|---|

Outline: Introduction

An overview of the Java language and its runtime environment.

Main points:

- 1) Familiarize participants with the language's features, goals, and documentation.
- 2) Explains how to set up the Java runtime environment to facilitate the development and execution of Java code.
- 3) Describes how to use the Java technology documentation.
- 4) Provides a simple example on how to write, compile and run a Java application.

Outline: Language

Introduces procedural aspects of the Java language.

Main points:

- 1) Basic data type and variables.
- 2) How to work with variables and arrays.
- 3) Operators: arithmetic, bitwise, relational, logical, etc.
- 4) Control structures: selection, iteration, jumps, etc.

Outline: Object-Orientation

Presents object-oriented aspects of the Java language.

Main points:

- 1) How to encapsulate data and behavior within classes.
- 2) How to build hierarchies of classes with inheritance.
- 3) How to determine the behavior of objects at run-time.
- 4) How to group classes into packages.
- 5) How to introduce abstraction thorough interfaces.

Outline: Horizontal APIs

API – Application Programming Interface

Horizontal APIs are used across the language.

Presentation of selected horizontal APIs:

- 1) string handling
- 2) event handling
- 3) Object collections

Outline: Vertical APIs

Vertical APIs are designed to perform specific functions.

Presentation of selected vertical APIs:

- 1) graphical user interface
- 2) applets
- 3) input/output
- 4) networking

Outline: Summary

Revision of the material introduced during the course.

How this course provides a foundation for the remaining courses:

- 1) distributed programming
- 2) Java XML processing
- 3) Java Web Services
- 4) J2EE web components
- 5) J2EE business components

Course Resources

1) books

- a) Java 2 The Complete Reference, Herbert Schildt, Osborne, 5th edition, 2002
- b) The Java Tutorial, Sun Microsystems,
<http://java.sun.com/docs/books/tutorial/>, 2004
- c) Thinking in Java, Bruce Ecker, 3rd edition, Prentice Hall, 2002

2) articles

Links available from the website <http://www.emacao.gov.mo>.

3) tools

- a) JDK 1.5
- b) Eclipse IDE

Course Logistics

- 1) **duration** - 36 hours
- 2) **activities** - lecture (hands-on), development
- 3) **sessions/day** - morning 09:00–13:00 and afternoon 14:30–16:30
- 4) **number of sessions** - 6 morning and 6 afternoon
- 5) **style** - interactive and tutorial

Course Prerequisite

- 1) some experience in object-oriented programming:
 - a) C++
 - b) Delphi
 - c) any other object-oriented language
- 2) basic understanding of TCP/IP networking concepts

Introduction

Course Outline

- | | | |
|---|--|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|--|---|

Overview

Topics under this module:

- 1) Java origin and history
- 2) Java technology
- 3) Java language
- 4) Java platform
- 5) simple Java program
- 6) Java documentation
- 7) setting up Java environment

Java Origins

Computer language innovation and development occurs for two fundamental reasons:

- 1) to adapt to changing environments and uses
- 2) to implement improvements in the art of programming

The development of Java was driven by both in equal measures.

Many Java features are inherited from the earlier languages:

$B \rightarrow C \rightarrow C++ \rightarrow \text{Java}$

Before Java: C

Designed by Dennis Ritchie in 1970s.

Before C, there was no language to reconcile: ease-of-use versus power, safety versus efficiency, rigidity versus extensibility.

BASIC, COBOL, FORTRAN, PASCAL optimized one set of traits, but not the other.

C- structured, efficient, high-level language that could replace assembly code when creating systems programs.

Designed, implemented and tested by programmers, not scientists.

Before Java: C++

Designed by Bjarne Stroustrup in 1979.

Response to the increased complexity of programs and respective improvements in the programming paradigms and methods:

- 1) assembler languages
- 2) high-level languages
- 3) structured programming
- 4) object-oriented programming (OOP)

OOP – methodology that helps organize complex programs through the use of inheritance, encapsulation and polymorphism.

C++ extends C by adding object-oriented features.

Java History

Designed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at Sun Microsystems in 1991.

The original motivation is not Internet: platform-independent software embedded in consumer electronics devices.

With Internet, the urgent need appeared to break the fortified positions of Intel, Macintosh and Unix programmer communities.

Java as an “Internet version of C++”? No.

Java was not designed to replace C++, but to solve a different set of problems. There are significant practical/philosophical differences.

Java Technology

There is more to Java than the language.

Java Technology consists of:

- 1) Java Programming Language
- 2) Java Virtual Machine (JVM)
- 3) Java Application Programming Interfaces (APIs)

Java Language Features

- 1) simple
- 2) object-oriented
- 3) robust
- 4) multithreaded
- 5) architecture-neutral
- 6) interpreted and high-performance
- 7) distributed
- 8) dynamic
- 9) secure

Java Language Features 1

- 1) **simple** – Java is designed to be easy for the professional programmer to learn and use.
- 2) **object-oriented** – a clean, usable, pragmatic approach to objects, not restricted by the need for compatibility with other languages.
- 3) **robust** – restricts the programmer to find the mistakes early, performs compile-time (strong typing) and run-time (exception-handling) checks, manages memory automatically.

Java Language Features 2

- 4) **multithreaded** – supports multi-threaded programming for writing program that perform concurrent computations
- 5) **architecture-neutral** – Java Virtual Machine provides a platform-independent environment for the execution of Java bytecode
- 6) **interpreted and high-performance** – Java programs are compiled into an intermediate representation – bytecode:
 - a) can be later interpreted by any JVM
 - b) can be also translated into the native machine code for efficiency.

Java Language Features 3

- 7) **distributed** – Java handles TCP/IP protocols, accessing a resource through its URL much like accessing a local file.
- 8) **dynamic** – substantial amounts of run-time type information to verify and resolve access to objects at run-time.
- 9) **secure** – programs are confined to the Java execution environment and cannot access other parts of the computer.

Execution Platform

What is an execution platform?

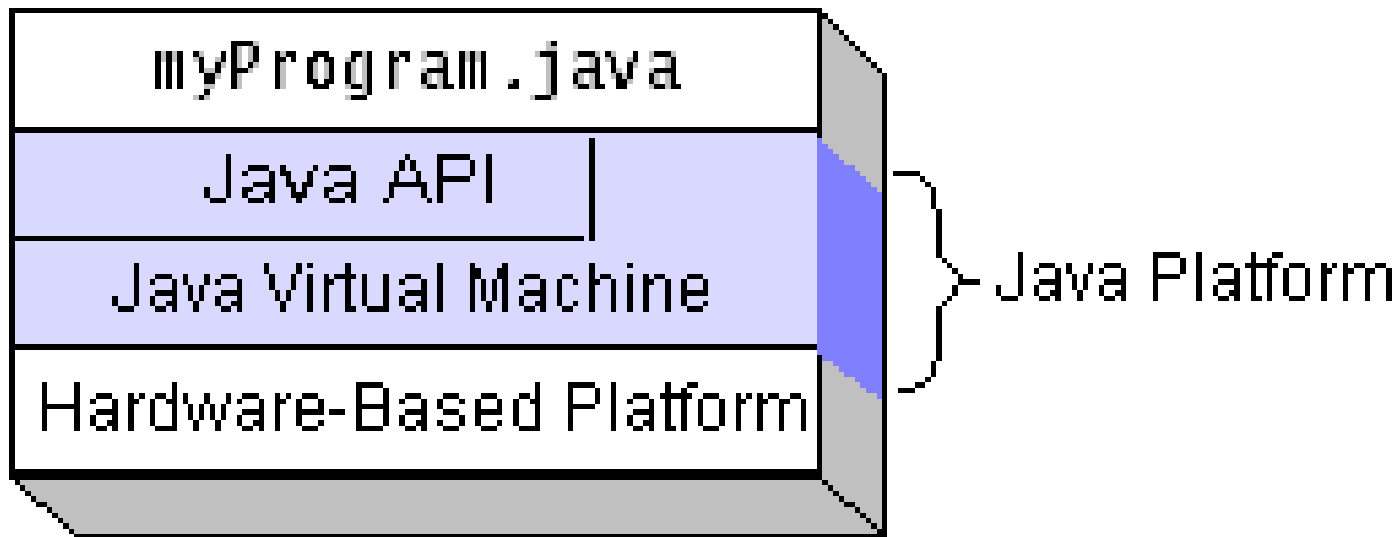
- 1) An execution platform is the hardware or software environment in which a program runs, e.g. Windows 2000, Linux, Solaris or MacOS.
- 2) Most platforms can be described as a combination of the operating system and hardware.

Java Execution Platform

What is Java Platform?

- 1) A software-only platform that runs on top of other hardware-based platforms.
- 2) Java Platform has two components:
 - a) Java Virtual Machine (JVM) – interpretation for the Java bytecode, ported onto various hardware-based platforms.
 - b) The Java Application Programming Interface (Java API)

Java Execution Platform

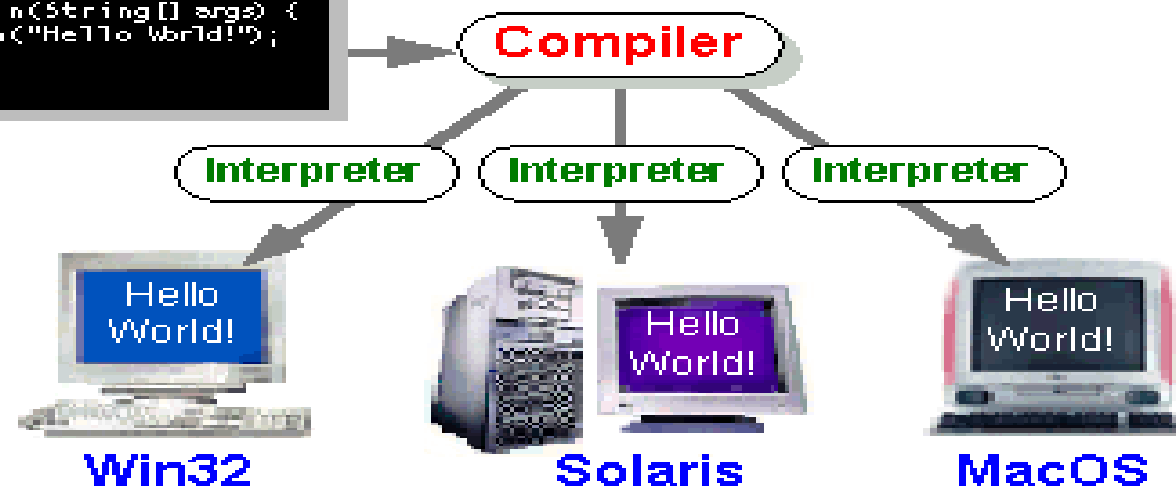


Java Platform Independence

Java Program

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

HelloWorldApp.java



Java Program Execution

Java programs are both compiled and interpreted:

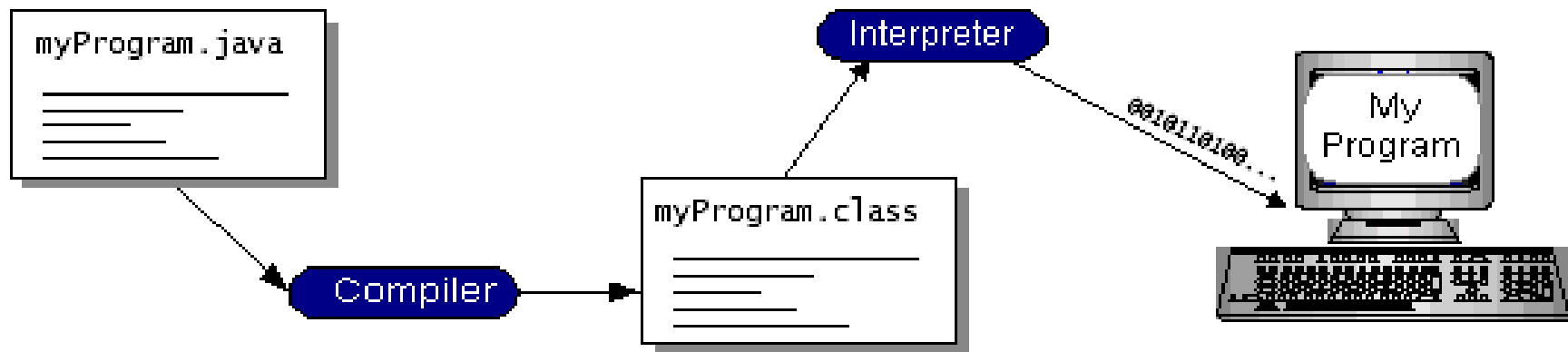
Steps:

- write the Java program
- compile the program into bytecode
- execute (interpret) the bytecode on the computer through the Java Virtual Machine

Compilation happens once.

Interpretation occurs each time the program is executed.

Java Execution Process



Java API

What is Java API?

- 1) a large collection of ready-made software components that provide many useful capabilities, e.g. graphical user interface
- 2) grouped into libraries (packages) of related classes and interfaces
- 3) together with JVM insulates Java programs from the hardware and operating system variations

Java Program Types

Types of Java programs:

- 1) applications – standalone (desktop) Java programs, executed from the command line, only need the Java Virtual Machine to run
- 2) applets – Java program that runs within a Java-enabled browser, invoked through a “applet” reference on a web page, dynamically downloaded to the client computer
- 3) servlets – Java program running on the web server, capable of responding to HTTP requests made through the network
- 4) etc.

Java Platform Features 1

- 1) **essentials** - objects, strings, threads, numbers, input/output, data structures, system properties, date and time, and others.
- 2) **networking**:
 - 1) Universal Resource Locator (URL)
 - 2) Transmission Control Protocol (TCP)
 - 3) User Datagram Protocol (UDP) sockets
 - 4) Internet Protocol (IP) addresses
- 3) **internationalization** - programs that can be localized for users worldwide, automatically adapting to specific locales and appropriate languages.

Java Platform Features 2

- 4) **security** – low-level and high-level security, including electronic signatures, public and private key management, access control, and certificates
- 5) **software components** – JavaBeans can plug into an existing component architecture
- 6) **object serialization** - lightweight persistence and communication, in particular using Remote Method Invocation (RMI)
- 7) **Java Database Connectivity** (JDBC) - provides uniform access to a wide range of relational databases

Java Technologies

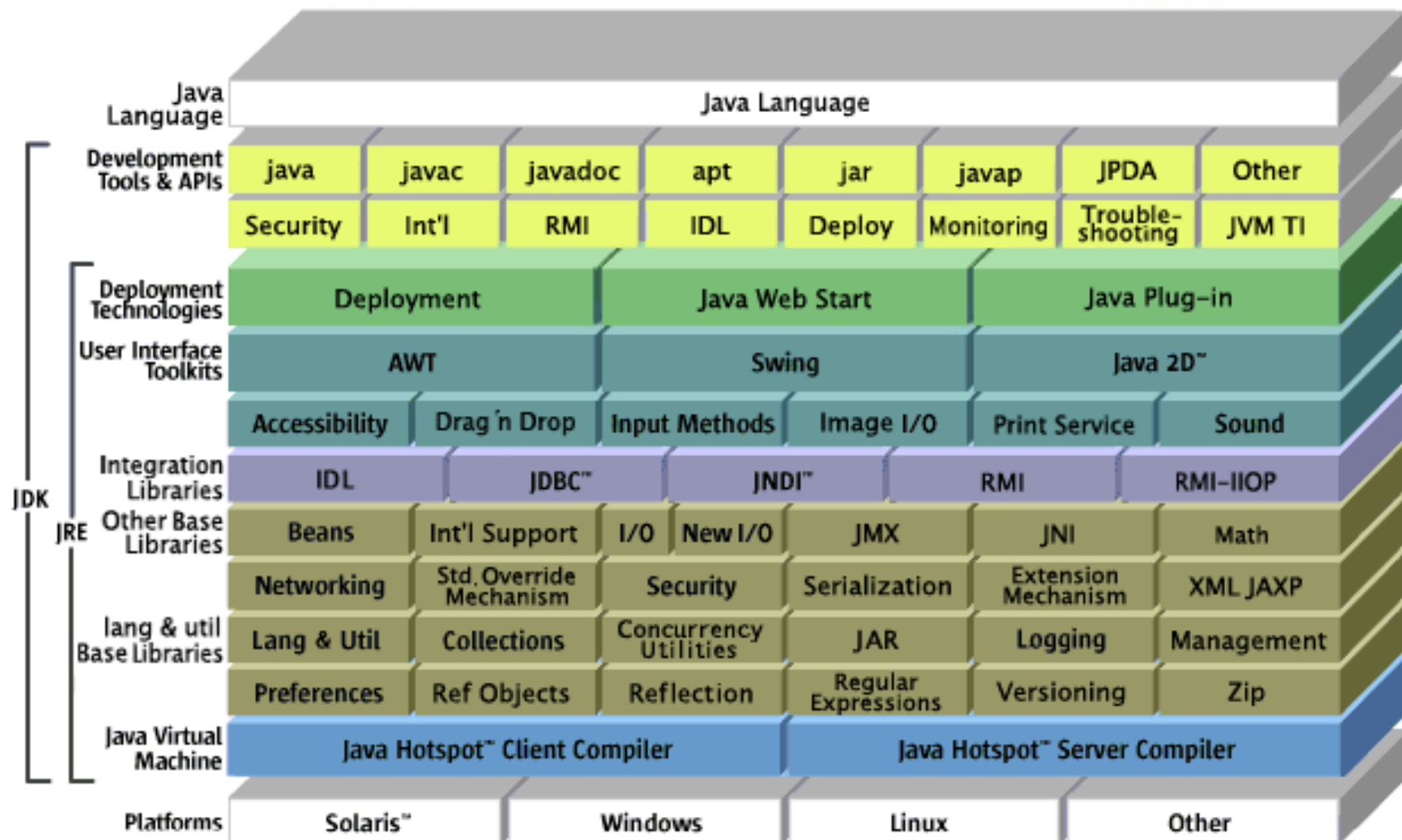
Different technologies depending on the target applications:

- 1) desktop applications - Java 2 Standard Edition (J2SE)
- 2) enterprise applications – Java 2 Enterprise Edition (J2EE)
- 3) mobile applications – Java 2 Mobile Edition (J2ME)
- 4) smart card applications – JavaCard
- 5) etc.

Each edition puts together a large collections of packages offering functionality needed and relevant to a given application.

The Java Virtual Machine remains essentially the same.

Java Technology: SDK



Exercise: Java Technology

- 1) Explain the statement “There is more to Java than the Language”.
- 2) Enumerate and explain Java design goals.
- 3) How does Java maintain a balance between Interpretation and High Performance?.
- 4) Java program is termed “Write once run everywhere”. Explain.
- 5) Why is it difficult to write viruses and malicious programs with Java?

Simple Java Program

A class to display a simple message:

```
class MyProgram {  
    public static void main(String[] args) {  
        System.out.println("First Java program.");  
    }  
}
```

Running the Program

Type the program, save as `MyProgram.java`.

In the command line, type:

```
> dir
MyProgram.java
> javac MyProgram.java
> dir
MyProgram.java, MyProgram.class
> java MyProgram
First Java program.
```


Explaining the Process

- 1) creating a source file - a source file contains text written in the Java programming language, created using any text editor on any system.
- 2) compiling the source file – Java compiler (javac) reads the source file and translates its text into instructions that the Java interpreter can understand. These instructions are called bytecode.
- 3) running the compiled program – Java interpreter (java) installed takes as input the bytecode file and carries out its instructions by translating them on the fly into instructions that your computer can understand.

Java Program Explained

`MyProgram` is the name of the class:

```
class MyProgram {
```

`main` is the method with one parameter `args` and no results:

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

`println` is a method in the standard `System` class:

```
        System.out.println("First Java program.");
    }
}
```

Classes and Objects

A class is the basic building block of Java programs.

A class encapsulates:

- a) data (attributes) and
- b) operations on this data (methods)

and permits to create objects as its instances.

Main Method

The `main` method must be present in every Java application:

- 1) `public static void main(String[] args)` where:
 - a) `public` means that the method can be called by any object
 - b) `static` means that the method is shared by all instances
 - c) `void` means that the method does not return any value
- 2) When the interpreter executes an application, it starts by calling its `main` method which in turn invokes other methods in this or other classes.
- 3) The main method accepts a single argument – a string array, which holds all command-line parameters.

Exercise: Java Program

1) Personalize the `MyProgram` program with your name so that it tells you "Hello, my name is ..."

2) Write a program that produces the following output:

```
Welcome to e-Macao Java Workshop!
```

```
I hope you will benefit from the training.
```

3) Here is a slightly modified version of `MyProgram`:

```
class MyProgram2 {  
    public void static Main(String args) {  
        system.out.println("First Java Program!");  
    }  
}
```

The program has some errors. Fix the errors so that the program successfully compiles and runs. What were the errors?

Using API Documentation 1

Ability to use Java API Specification is crucial for any practicing programmer.
Here are the steps:

- 1) download the API documentation from
<http://java.sun.com/j2se/1.5.0/download.jsp>
- 2) Extract the archive file

Using API Documentation 2

3) open `index.html`

JDK™ 5.0 Documentation

[Search](#) [General Info](#) [API & Language](#) [Guide to Features](#) [Tool Docs](#) [Demos/Tutorials](#)

J2SE™ Platform at a Glance

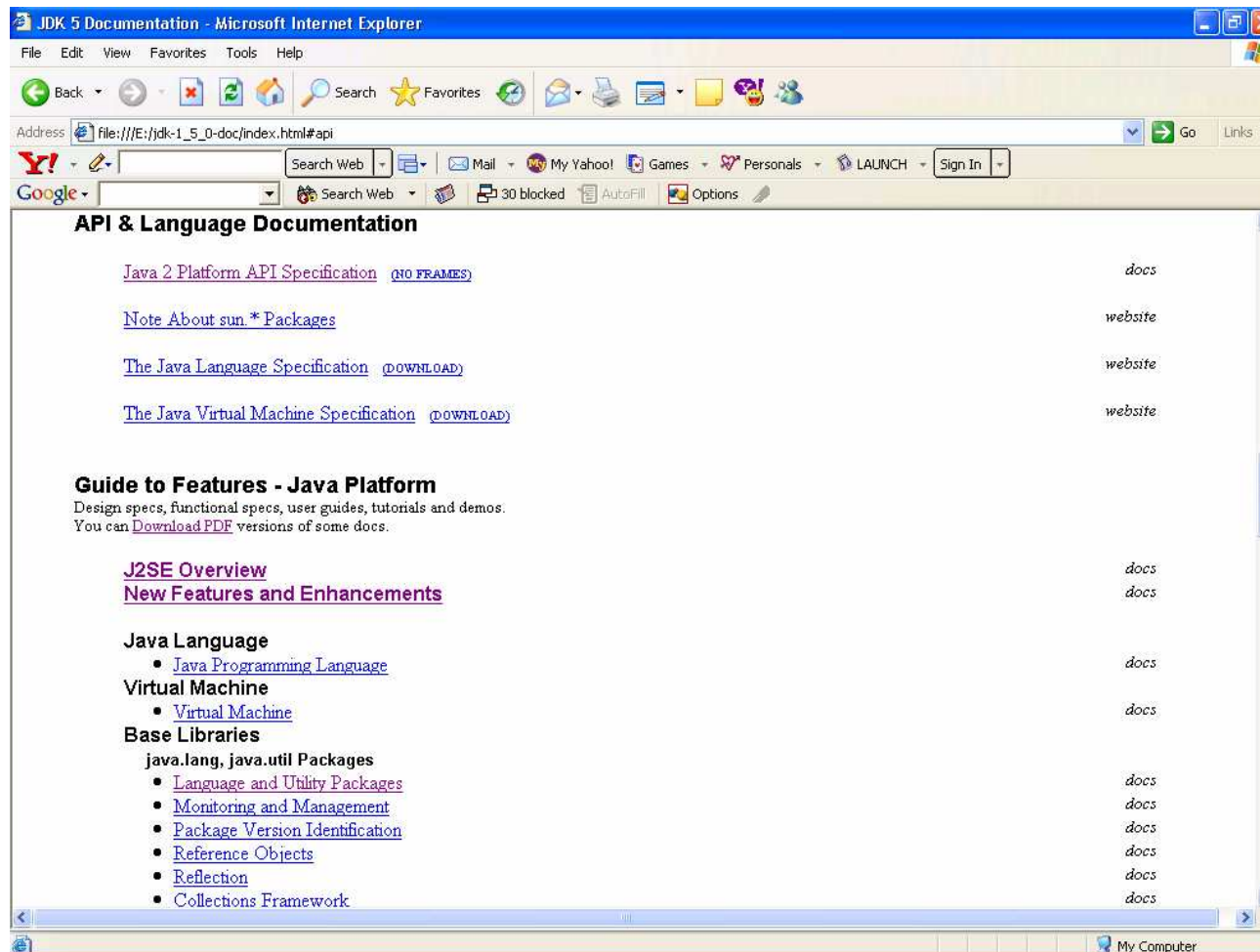
This document covers the Java™ 2 Platform Standard Edition 5.0 Development Kit (JDK 5.0). Its external version number is 5.0 and internal version number is 1.5.0. For information on a feature of the JDK, click on its component in the diagram below.

Java™ 2 Platform Standard Edition 5.0

Java Language							
Development Tools & APIs							
java	javac	javadoc	apt	jar	javap	JPDA	Other
Security	Int'l	RMI	IDL	Deploy	Monitoring	Trouble-shooting	JVM TI
Deployment		Java Web Start			Java Plug-in		
AWT		Swing			Java 2D™		
Accessibility	Drag'n Drop	Input Methods	Image I/O	Print Service	Sound		
IDL		JDBC™		JNDI™		RMI	RMI-IIOP
Beans	Int'l Support	I/O	New I/O	JMX	JNI	Math	
Networking	Std. Override	Security	Serialization	Extension	XMI IAXP		

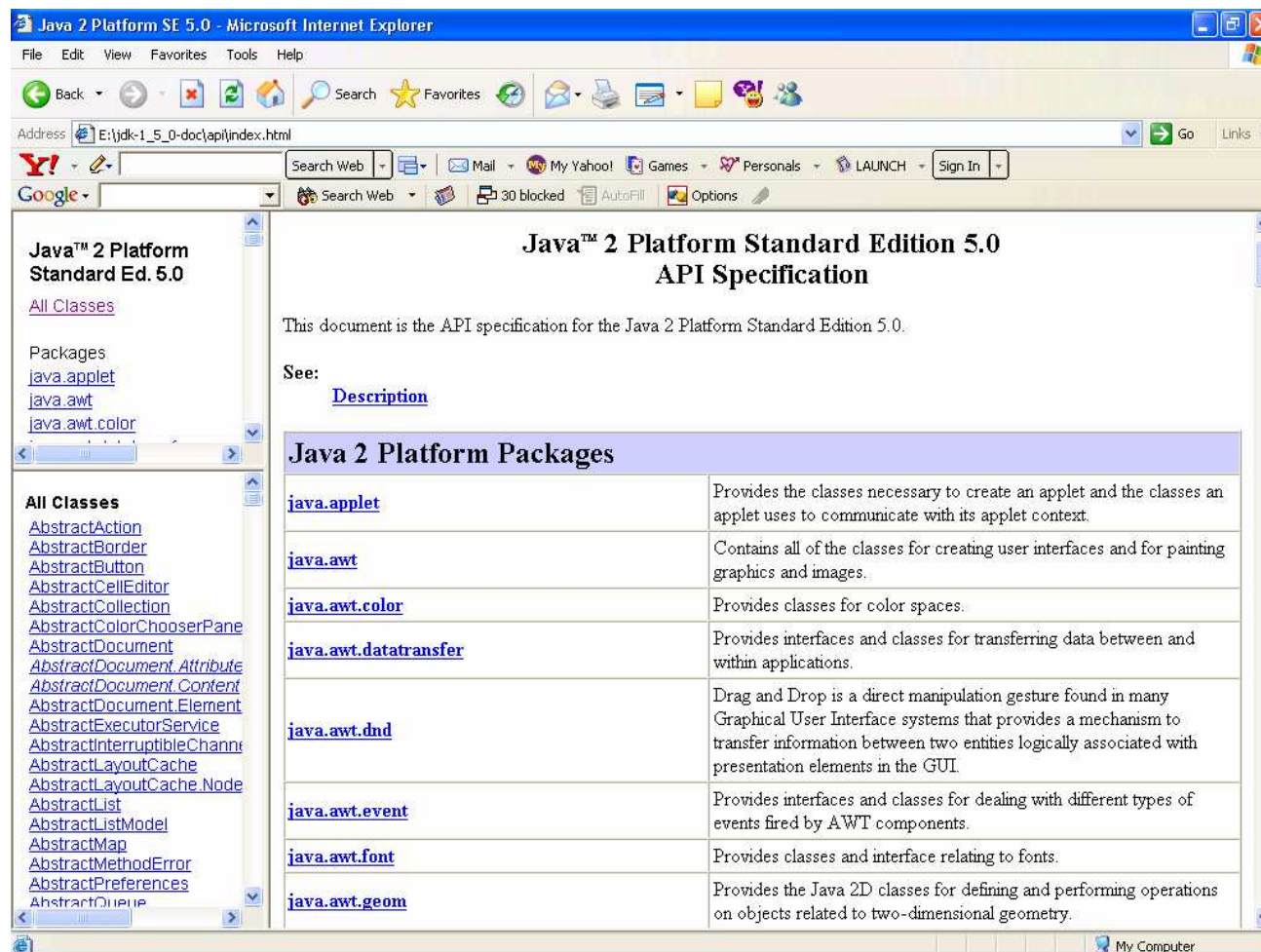
Using API Documentation 3

4) click on API & Language



Using API Documentation 4

5) click on [Java 2 Platform API Specification](#)



Java 2 Platform API Specification

The window is divided into three panes:

- 1) specification pane - click on any package, package pane will display all classes, interfaces, exceptions and errors that belong to that package.
- 2) package pane - click on any class, class pane will display all information about the class.
- 3) class pane – contains such information as:
 - 1) inheritance hierarchy
 - 2) implemented interfaces
 - 3) constructors
 - 4) attributes
 - 5) methods

Java Environment Setup 1

Setting up `JAVA_HOME` and `PATH` environment variables is necessary for the Java tools and applications to work properly from any directory.

Steps to carry out:

- 1) `JAVA_HOME` variable should point to the top installation directory of Java environment.
- 2) `PATH` should point to the `bin` directory where the Java Virtual Machine - interpreter (`java`), compiler (`javac`) and other executables are located.

Java Environment Setup 2

With more than one JVM installed, the one you wish to use must appear as the first one in the PATH variable.

Example:

- 1) Windows - add `%JAVA_HOME%\bin` to the beginning of the `PATH` variable.
- 2) Linux - include `PATH="$PATH:$JAVA_HOME/bin:."` in your `/etc/profile` or `.bashrc` files.

To test the environment, open the command window and run:

```
java -version
```

The current version number of the installed JVM should appear.

Summary: Introduction

Material covered:

- 1) What is the origin and history of Java?
- 2) What is Java Technology and its components?
- 3) What are the basic features of the Java language?
- 4) What are the basic features of the Java execution platform?
- 5) How to write, compile and execute a simple Java application?
- 6) How to use Java API documentation?
- 7) How to set up the Java execution environment?

Exercise: API Specification

- 1) Download and extract the latest JDK 5.0 Documentation.
- 2) Using the Documentation
 - a) Locate the `java.math` package
 - b) How many classes does it have?
 - c) List all the classes.
 - d) List their inheritance hierarchies.
 - e) List their implemented interfaces.
 - f) How many constructors does each have?
 - g) How many fields does each have?
 - h) How many methods does each have?

Language

Course Outline

- | | | |
|--|--|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|--|--|---|

Language

Java is intrinsically an object-oriented language.

However, for presentation reasons explain it in two parts:

- a) procedural part
- b) object-oriented part

Language Overview

- a) **syntax** – whitespaces, identifiers, comments, separators, keywords
- b) **types** – simple types byte, short, int, long, float double, char, boolean
- c) **variables** – declaration, initialization, scope, conversion and casting
- d) **arrays** – declaration, initialization, one- and multi-dimensional arrays
- e) **operators** – arithmetic, relational, conditional, shift, logical, assignment
- f) **control flow** – branching, selection, iteration, jumps and returns

Syntax

Course Outline

- | | | |
|--|--|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|--|--|---|

Java Syntax

On the most basic level, Java programs consist of:

- a) whitespaces
- b) identifiers
- c) comments
- d) literals
- e) separators
- f) keywords
- g) operators

Each of them will be described in order.

Whitespaces

A whitespace is a space, tab or new line.

Java is a form-free language that does not require special indentation.

A program could be written like this:

```
class MyProgram {  
    public static void main(String[] args) {  
        System.out.println("First Java program.");  
    }  
}
```

It could be also written like this:

```
class MyProgram { public static void main(String[] args)  
{ System.out.println("First Java program."); } }
```

Identifiers

Java identifiers:

- a) used for class names, method names, variable names
- b) an identifier is any sequence of letters, digits, “_” or “\$” characters that do not begin with a digit
- c) Java is case sensitive, so `value`, `Value` and `VALUE` are all different.

Seven identifiers in this program:

```
class MyProgram {  
    public static void main(String[] args) {  
        System.out.println("First Java program.");  
    }  
}
```

Comments 1

Three kinds of comments:

- 1) Ignore the text between `/*` and `*/`:

```
/* text */
```

- 2) Documentation comment (`javadoc` tool uses this kind of comment to automatically generate software documentation):

```
/** documentation */
```

- 3) Ignore all text from `//` to the end of the line:

```
// text
```


Comments 2

```
/**
 *   MyProgram implements application that displays
 *       a simple message on the standard output
 *   device.
 */
class MyProgram {
    /* The main method of the class.*/
    public static void main(String[] args) {
        //display string
        System.out.println("First Java program.");
    }
}
```

Literals

A literal is a constant value of certain type.

It can be used anywhere values of this type are allowed.

Examples:

a) 100

b) 98.6

c) 'X'

d) "test"

```
class MyProgram {  
    public static void main(String[] args) {  
        System.out.println("My first Java program.");  
    }  
}
```

Separators

()	parenthesis	lists of parameters in method definitions and invocations, precedence in expressions, type casts
{ }	braces	block of code, class definitions, method definitions, local scope, automatically initialized arrays
[]	brackets	declaring array types, referring to array values
;	semicolon	terminating statements, chain statements inside the “for” statement
,	comma	separating multiple identifiers in a variable declaration
.	period	separate package names from subpackages and classes, separating an object variable from its attribute or method

Keywords

Keywords are reserved words recognized by Java that cannot be used as identifiers. Java defines 49 keywords as follows:

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

Exercise: Syntax

- 1) What's the difference between a *keyword* and an *identifier*?
- 2) What's the difference between an *identifier* and a *literal*?
- 3) What's the difference between `/** text */` and `/* text */`.
- 4) Which of these are valid identifiers?

```
int, anInt, I, i1, 1, thing1, lthing, one-hundred,  
one_hundred, something2do
```

- 5) Identify the literals, separators and identifiers in the program below.

```
class MyProgram {  
    int i = 30;  int j = i;  char c = 'H';  
    public static void main(String[] args) {  
        System.out.println("i and j " + i + " " + j);  
    } }  
}
```

- 6) What's the difference between a brace `{ }` and a bracket `()`?

Types

Course Outline

- | | | |
|--|--|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|--|--|---|

Strong Typing

Java is a strongly-typed language:

- a) every variable and expression has a type
- b) every type is strictly defined
- c) all assignments are checked for type-compatibility
- d) no automatic conversion of non-compatible, conflicting types
- e) Java compiler type-checks all expressions and parameters
- f) any typing errors must be corrected for compilation to succeed

Simple Types

Java defines eight simple types:

- 1) `byte` – 8-bit integer type
- 2) `short` – 16-bit integer type
- 3) `int` – 32-bit integer type
- 4) `long` – 64-bit integer type
- 5) `float` – 32-bit floating-point type
- 6) `double` – 64-bit floating-point type
- 7) `char` – symbols in a character set
- 8) `boolean` – logical values `true` and `false`

Simple Type: byte

8-bit integer type.

Range: -128 to 127 .

Example:

```
byte b = -15;
```

Usage: particularly when working with data streams.

Simple Type: short

16-bit integer type.

Range: -32768 to 32767 .

Example:

```
short c = 1000;
```

Usage: probably the least used simple type.

Simple Type: int

32-bit integer type.

Range: `-2147483648` to `2147483647`.

Example:

```
int b = -50000;
```

Usage:

- 1) Most common integer type.
- 2) Typically used to control loops and to index arrays.
- 3) Expressions involving the `byte`, `short` and `int` values are promoted to `int` before calculation.

Simple Type: long

64-bit integer type.

Range: -9223372036854775808 to 9223372036854775807 .

Example:

```
long l = 1000000000000000000;
```

Usage:

1) useful when `int` type is not large enough to hold the desired value

Example: long

```
// compute the light travel distance
class Light {
    public static void main(String args[]) {
        int lightspeed = 186000;
        long days = 1000;
        long seconds = days * 24 * 60 * 60;
        long distance = lightspeed * seconds;
        System.out.print("In " + days);
        System.out.print(" light will travel about
");
        System.out.println(distance + " miles.");
    }
}
```

Simple Type: float

32-bit floating-point number.

Range: $1.4e-045$ to $3.4e+038$.

Example:

```
float f = 1.5;
```

Usage:

- 1) fractional part is needed
- 2) large degree of precision is not required

Simple Type: double

64-bit floating-point number.

Range: $4.9e-324$ to $1.8e+308$.

Example:

```
double pi = 3.1416;
```

Usage:

- 1) accuracy over many iterative calculations
- 2) manipulation of large-valued numbers

Example: double

```
// Compute the area of a circle.
class Area {
    public static void main(String args[]) {
        double pi = 3.1416;        // approximate pi value
        double r = 10.8;            // radius of circle
        double a = pi * r * r;      // compute area
        System.out.println("Area of circle is " + a);
    }
}
```

Simple Type: char

16-bit data type used to store characters.

Range: 0 to 65536.

Example:

```
char c = 'a';
```

Usage:

- 1) Represents both ASCII and Unicode character sets; Unicode defines a character set with characters found in (almost) all human languages.
- 2) Not the same as in C/C++ where `char` is 8-bit and represents ASCII only.

Example: char

```
// Demonstrate char data type.
class CharDemo {
    public static void main(String args[]) {
        char ch1, ch2;
        ch1 = 88;    // code for X
        ch2 = 'Y';
        System.out.print("ch1 and ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

Another Example: char

It is possible to operate on `char` values as if they were integers:

```
class CharDemo2 {  
    public static void main(String args[]) {  
        char c = 'X';  
        System.out.println("c contains " + c);  
        c++; // increment c  
        System.out.println("c is now " + c);  
    }  
}
```

Simple Type: boolean

Two-valued type of logical values.

Range: values `true` and `false`.

Example:

```
boolean b = (1<2);
```

Usage:

- 1) returned by relational operators, such as `1<2`
- 2) required by branching expressions such as `if` or `for`

Example: boolean

```
class BoolTest {  
    public static void main(String args[]) {  
        boolean b;  
        b = false;  
        System.out.println("b is " + b);  
        b = true;  
        System.out.println("b is " + b);  
        if (b) System.out.println("executed");  
        b = false;  
        if (b) System.out.println("not executed");  
        System.out.println("10 > 9 is " + (10 > 9));  
    }  
}
```

Literals Revisited

Literals express constant values.

The form of a literal depends on its type:

- 1) integer types
- 2) floating-point types
- 3) character type
- 4) boolean type
- 5) string type

Literals: Integer Types

Writing numbers with different bases:

- 1) decimal – `123`
- 2) octal – `0173`
- 3) hexadecimal – `0x7B`

Integer literals are of type `int` by default.

Integer literal written with “`L`” (e.g. `123L`) are of type `long`.

Literals: Floating-Point Types

Two notations:

- 1) standard – `2000.5`
- 2) scientific – `2.0005E3`

Floating-point literals are of type `double` by default.

Floating-point literal written with “F” (e.g. `2.0005E3F`) are of type `float`.

Literals: Boolean

Two literals are allowed only: `true` and `false`.

Those values do not convert to any numerical representation.

In particular:

- 1) `true` is not equal to `1`
- 2) `false` is not equal to `0`

Literals: Characters

Character literals belong to the Unicode character set.

Representation:

- 1) visible characters inside quotes, e.g. `'a'`
- 2) invisible characters written with escape sequences:
 - a) `\ddd` octal character `ddd`
 - b) `\uxxxx` hexadecimal Unicode character `xxxx`
 - c) `\'` single quote
 - d) `\"` double quote
 - e) `\\` backslash
 - f) `\r` carriage return
 - g) `\n` new line
 - h) `\f` form feed
 - i) `\t` tab
 - j) `\b` backspace

Literals: String

String is not a simple type.

String literals are character-sequences enclosed in double quotes.

Example:

```
"Hello World!"
```

Notes:

- 1) escape sequences can be used inside string literals
- 2) string literals must begin and end on the same line
- 3) unlike in C/C++, in Java `String` is not an array of characters

Exercise: Types

1) What is the value of `b` in the following code snippet?

```
byte b = 0;  
b += 1;
```

2) What happens when this code is executed?

```
char c = -1;
```

3) Which of the following lines will compile without warning or error. Explain each line.

```
float f=1.3;  
char c="a";  
byte b=257;  
boolean b=null;  
int i=10;
```

4) Write a java statement that will print the following line to the console:

```
To print a ` " `, we use the ` \` " `escape sequence.
```

Variables

Course Outline

- | | | |
|--|--|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|--|--|---|

Outline: Variables

- 1) declaration – how to assign a type to a variable
- 2) initialization – how to give an initial value to a variable
- 3) scope – how the variable is visible to other parts of the program
- 4) lifetime – how the variable is created, used and destroyed
- 5) type conversion – how Java handles automatic type conversion
- 6) type casting – how the type of a variable can be narrowed down
- 7) type promotion – how the type of a variable can be expanded

Variables

Java uses variables to store data.

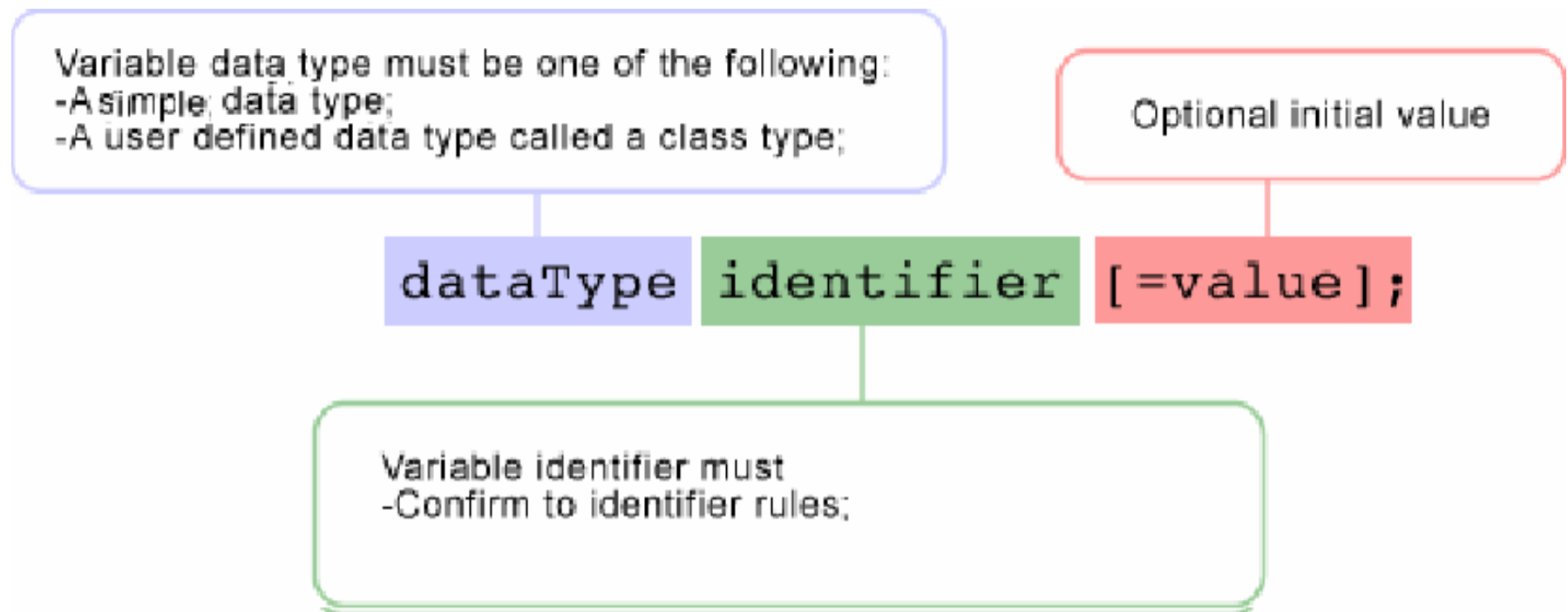
To allocate memory space for a variable JVM requires:

- 1) to specify the data type of the variable
- 2) to associate an identifier with the variable
- 3) optionally, the variable may be assigned an initial value

All done as part of variable declaration.

Basic Variable Declaration

Basic form of variable declaration:



Variable Declaration

We can declare several variables at the same time:

```
type identifier [=value][, identifier [=value] ...];
```

Examples:

```
int a, b, c;  
int d = 3, e, f = 5;  
byte hog = 22;  
double pi = 3.14159;  
char kat = 'x';
```

Constant Declaration

A variable can be declared as final:

```
final double PI = 3.14;
```

The value of the final variable cannot change after it has been initialized:

```
PI = 3.13;
```

Variable Identifiers

Identifiers are assigned to variables, methods and classes.

An identifier:

- 1) starts with a letter, underscore `_` or dollar `$`
- 2) can contain letters, digits, underscore or dollar characters
- 3) it can be of any length
- 4) it must not be a keyword (e.g. `class`)
- 5) it must be unique in its scope

Examples: `identifier`, `userName`, `_sys_var1`, `$change`

The code of Java programs is written in Unicode, rather than ASCII, so letters and digits have considerably wider definitions than just a-z and 0-9.

Naming Conventions

Conventions are not part of the language.

Naming conventions:

- 1) variable names begin with a lowercase letter
- 2) class names begin with an uppercase letter
- 3) constant names are all uppercase

If a variable name consists of more than one word, the words are joined together, and each word after the first begins with an uppercase letter.

The underscore character is used only to separate words in constants, as they are all caps and thus cannot be case-delimited.

Variable Initialization

During declaration, variables may be optionally initialized.

Initialization can be static or dynamic:

- 1) static – initialize with a literal:

```
int n = 1;
```

- 2) dynamic – initialize with an expression composed of any literals, variables or method calls available at the time of initialization:

```
int m = n + 1;
```

The types of the expression and variable must be the same.

Example: Variable Initialization

```
class DynamicInit {  
    public static void main(String args[]) {  
        double a = 3.0, b = 4.0;  
        double c = Math.sqrt(a * a + b * b);  
        System.out.println("Hypotenuse is " + c);  
    }  
}
```


Variable Scope

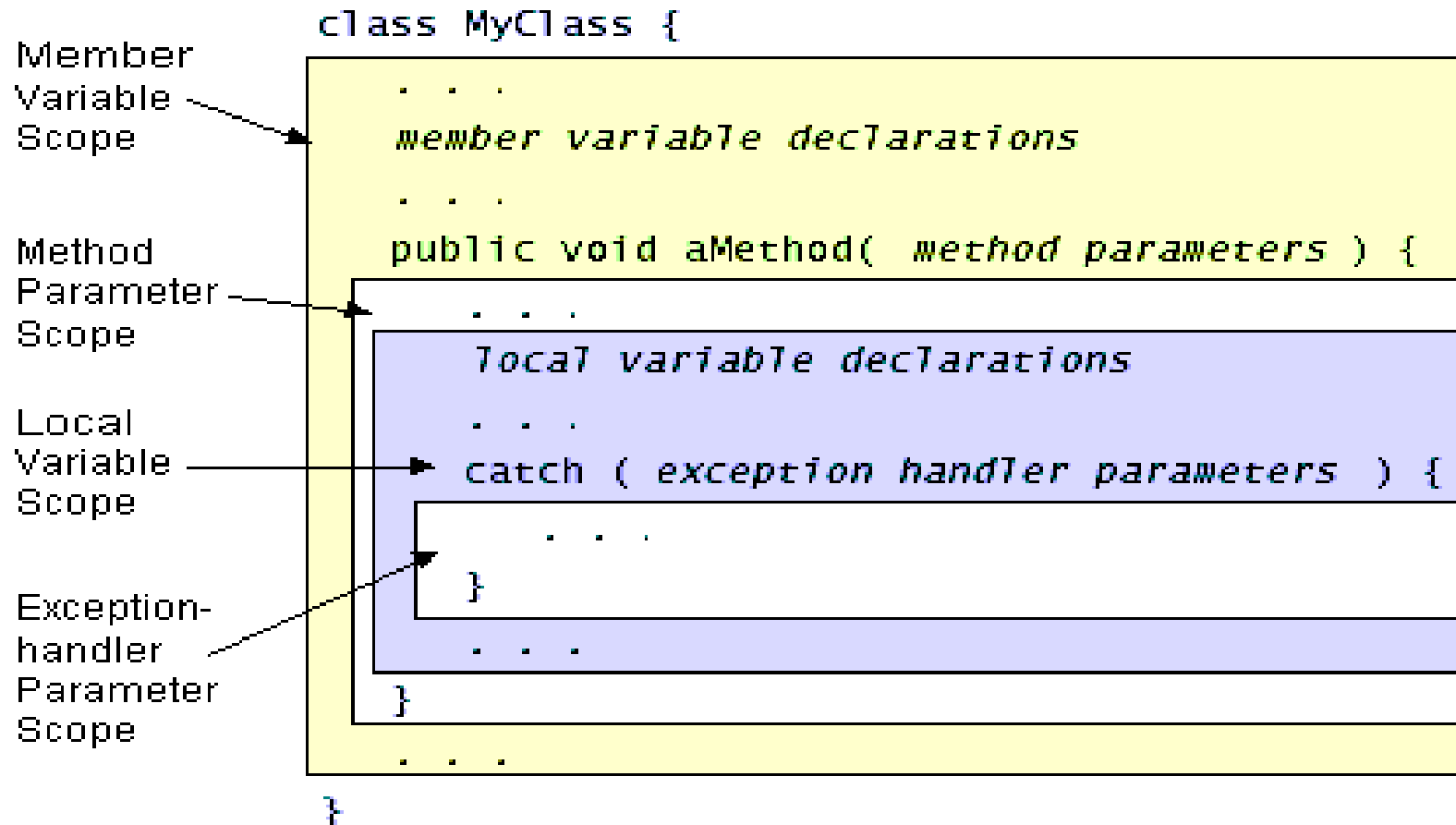
Scope determines the visibility of program elements with respect to other program elements.

In Java, scope is defined separately for classes and methods:

- 1) variables defined by a class have a “global” scope
- 2) variables defined by a method have a “local” scope

We consider the scope of method variables only; class variables will be considered later.

Variable Scope



Scope Definition

A scope is defined by a block:

```
{  
    ...  
}
```

A variable declared inside the scope is not visible outside:

```
{  
    int n;  
}  
  
n = 1;
```

Scope Nesting

Scopes can be nested:

```
{      ...      { ... }      ...      }
```

Variables declared in the outside scope are visible in the inside scope, but not the other way round:

```
{  
    int i;  
    {  
        int n = i;  
    }  
    int m = n;  
}
```

Example: Variable Scope

```
class Scope {  
    public static void main(String args[]) {  
        int x;  
        x = 10;  
        if (x == 10) {  
            int y = 20;  
            System.out.println("x and y: " + x + "  
" + y);  
            x = y * 2;  
        }  
        System.out.println("x is " + x + "y is" + y);  
    }  
}
```

Declaration Order

Method variables are only valid after their declaration:

```
{  
  int n = m;  
  int m;  
}
```

Variable Lifetime

Variables are created when their scope is entered by control flow and destroyed when their scope is left:

- 1) A variable declared in a method will not hold its value between different invocations of this method.
- 2) A variable declared in a block loses its value when the block is left.
- 3) Initialized in a block, a variable will be re-initialized with every re-entry.

Variable's lifetime is confined to its scope!

Example: Variable Lifetime

```
class LifeTime {  
    public static void main(String args[]) {  
        int x;  
        for (x = 0; x < 3; x++) {  
            int y = -1;  
            System.out.println("y is: " + y);  
            y = 100;  
            System.out.println("y is now: " + y);  
        }  
    }  
}
```


Type Differences

Suppose a value of one type is assigned to a variable of another type.

```
T1 t1;  
T2 t2 = t1;
```

What happens? Different situations:

- 1) types T1 and T2 are incompatible
- 2) types T1 and T2 are compatible:
 - a) T1 and T2 are the same
 - b) T1 is larger than T2
 - c) T2 is larger than T1

Type Compatibility

When types are compatible:

- 1) integer types and floating-point types are compatible with each other
- 2) numeric types are not compatible with `char` or `boolean`
- 3) `char` and `boolean` are not compatible with each other

Examples:

```
byte b;  
int i = b;  
char c = b;
```

Widening Type Conversion

Java performs automatic type conversion when:

- 1) two types are compatible
- 2) destination type is larger then the source type

Example:

```
int i;  
double d = i;
```

Narrowing Type Conversion

When:

- 1) two types are compatible
- 2) destination type is smaller than the source type

then Java will not carry out type-conversion:

```
int i;  
byte b = i;
```

Instead, we have to rely on manual type-casting:

```
int i;  
byte b = (byte)i;
```

Type Casting

General form: `(targetType) value`

Examples:

1) integer value will be reduced module `byte`'s range:

```
int i;  
byte b = (byte) i;
```

2) floating-point value will be truncated to integer value:

```
float f;  
int i = (int) f;
```

Example: Type Casting

```
class Conversion {  
    public static void main(String args[]) {  
        byte b;  
        int i = 257;  
        double d = 323.142;  
        System.out.println("\nConversion of int to byte.");  
        b = (byte) i;  
        System.out.println("i and b " + i + " " + b);  
        System.out.println("\ndouble to int.");  
        i = (int) d;  
        System.out.println("d and i " + d + " " + i);  
    }  
}
```

Type Promotion

In an expression, precision required to hold an intermediate value may sometimes exceed the range of either operand:

```
byte a = 40;  
byte b = 50;  
byte c = 100;  
int d = a * b / c;
```

Java promotes each `byte` operand to `int` when evaluating the expression.

Type Promotion Rules

- 1) `byte` and `short` are always promoted to `int`
- 2) if one operand is `long`, the whole expression is promoted to `long`
- 3) if one operand is `float`, the entire expression is promoted to `float`
- 4) if any operand is `double`, the result is `double`

Danger of automatic type promotion:

```
byte b = 50;  
b = b * 2;
```

What is the problem?

Example: Type Promotion

```
class Promote {  
    public static void main(String args[]) {  
        byte b = 42;  
        char c = 'a';  
        short s = 1024;  
        int i = 50000;  
        float f = 5.67f;  
        double d = .1234;  
        double result = (f * b) + (i / c) - (d * s);  
        System.out.println("result = " + result);  
    }  
}
```

Exercise: Variables

- 1) What's the difference between widening conversion and casting?
- 2) What is the output of the program below?

```
class MyProgram {  
    static final double pi = 3.15;  
    static double radius = 2.78;  
    public static void main(String[] args) {  
        pi = 3.14;  
        double area = pi * radius * radius;  
        System.out.println("The Area is " + area);  
    }  
}
```

- 3) What is the value of c[50] after this declaration:

```
char[] c = new char[100];
```

Arrays

Course Outline

- | | | |
|--|--|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|--|--|---|

Arrays

An array is a group of liked-typed variables referred to by a common name, with individual variables accessed by their index.

Arrays are:

- 1) declared
- 2) created
- 3) initialized
- 4) used

Also, arrays can have one or several dimensions.

Array Declaration

Array declaration involves:

- 1) declaring an array identifier
- 2) declaring the number of dimensions
- 3) declaring the data type of the array elements

Two styles of array declaration:

```
type array-variable[];
```

or

```
type [] array-variable;
```

Array Creation

After declaration, no array actually exists.

In order to create an array, we use the `new` operator:

```
type array-variable[];  
array-variable = new type[size];
```

This creates a new array to hold `size` elements of type `type`, which reference will be kept in the variable `array-variable`.

Array Indexing

Later we can refer to the elements of this array through their indexes:

```
array-variable[index]
```

The array index always starts with zero!

The Java run-time system makes sure that all array indexes are in the correct range, otherwise raises a run-time error.

Example: Array Use

```
class Array {  
    public static void main(String args[]) {  
        int monthDays[];  
        monthDays = new int[12];  
        monthDays[0] = 31;  
        monthDays[1] = 28;  
        monthDays[2] = 31;  
        monthDays[3] = 30;  
        monthDays[4] = 31;  
        monthDays[5] = 30;  
        ...  
        monthDays[12] = 31;  
        System.out.print("April has ");  
        System.out.println(monthDays[3] +" days.");  
    }  
}
```

Array Initialization

Arrays can be initialized when they are declared:

```
int monthDays[] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Comments:

- 1) there is no need to use the `new` operator
- 2) the array is created large enough to hold all specified elements

Example: Array Initialization

```
class Array {  
    public static void main(String args[]) {  
        int mthDys[3]=  
            {31,28,31,30,31,30,31,31,30,31,30,31};  
  
        System.out.print("April ");  
        System.out.println(mthDys[3]+ " days.");  
    }  
}
```

Multidimensional Arrays

Multidimensional arrays are arrays of arrays:

1) declaration

```
int array[][];
```

2) creation

```
int array = new int[2][3];
```

3) initialization

```
int array[][] = { {1, 2, 3}, {4, 5, 6} };
```

Example: Multidimensional Arrays

```
class Array {  
  
    public static void main(String args[]) {  
  
        int array[][] = { {1, 2, 3}, {4, 5, 6} };  
        int i, j, k = 0;  
  
        for(i=0; i<2; i++) {  
            for(j=0; j<3; j++)  
                System.out.print(array[i][j] + " ");  
            System.out.println();  
        }  
    }  
}
```

Exercise: Arrays

- 1) Write a program that creates an array of 10 integers with the initial values of 3.
- 2) Write a Java program to find the average of a sequence of nonnegative numbers entered by the user, where the user enters a negative number to terminate the input. Assume the only method in the class is the main method.
- 3) What's the index of the first and the last component of a one hundred component array?
- 4) What will happen if you try to compile and run the following code?

```
public class Q {  
    public static void main(String argv[]) {  
        int var[]=new int[5];  
        System.out.println(var[0]);  
    } }  

```

Operators

Course Outline

- | | | |
|--|--|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|--|--|---|

Operators Types

Java operators are used to build value expressions.

Java provides a rich set of operators:

- 1) assignment
- 2) arithmetic
- 3) relational
- 4) logical
- 5) bitwise
- 6) other

Operators and Operands

Each operator takes one, two or three operands:

- 1) a unary operator takes one operand

```
j++;
```

- 2) a binary operator takes two operands

```
i = j++;
```

- 3) a ternary operator requires three operands

```
i = (i>12) ? 1 : i++;
```

Assignment Operator

A binary operator:

```
variable = expression;
```

It assigns the value of the expression to the variable.

The types of the variable and expression must be compatible.

The value of the whole assignment expression is the value of the expression on the right, so it is possible to chain assignment expressions as follows:

```
int x, y, z;  
x = y = z = 2;
```

Arithmetic Operators

Java supports various arithmetic operators for:

- 1) integer numbers
- 2) floating-point numbers

There are two kinds of arithmetic operators:

- 1) basic: addition, subtraction, multiplication, division and modulo
- 2) shortcut: arithmetic assignment, increment and decrement

Table: Basic Arithmetic Operators

+	<code>op1 + op2</code>	adds <code>op1</code> and <code>op2</code>
-	<code>op1 - op2</code>	subtracts <code>op2</code> from <code>op1</code>
*	<code>op1 * op2</code>	multiplies <code>op1</code> by <code>op2</code>
/	<code>op1 / op2</code>	divides <code>op1</code> by <code>op2</code>
%	<code>op1 % op2</code>	computes the remainder of dividing <code>op1</code> by <code>op2</code>

Arithmetic Assignment Operators

Instead of writing

```
variable = variable operator expression;
```

for any arithmetic binary operator, it is possible to write shortly

```
variable operator= expression;
```

Benefits of the assignment operators:

- 1) save some typing
- 2) are implemented more efficiently by the Java run-time system

Table: Arithmetic Assignments

<code>+=</code>	<code>v += expr;</code>	<code>v = v + expr;</code>
<code>-=</code>	<code>v -= expr;</code>	<code>v = v - expr;</code>
<code>*=</code>	<code>v *= expr;</code>	<code>v = v * expr;</code>
<code>/=</code>	<code>v /= expr;</code>	<code>v = v / expr;</code>
<code>%=</code>	<code>v %= expr;</code>	<code>v = v % expr;</code>

Increment/Decrement Operators

Two unary operators:

- 1) `++` increments its operand by 1
- 2) `--` decrements its operand by 1

The operand must be a numerical variable.

Each operation can appear in two versions:

- **prefix** version evaluates the value of the operand **after** performing the increment/decrement operation
- **postfix** version evaluates the value of the operand **before** performing the increment/decrement operation

Table: Increment/Decrement

++	v++	return value of v , then increment v
++	++v	increment v , then return its value
--	v--	return value of v , then decrement v
--	--v	decrement v , then return its value

Example: Increment/Decrement

```
class IncDec {  
    public static void main(String args[]) {  
        int a = 1;  
        int b = 2;  
        int c, d;  
        c = ++b;  
        d = a++;  
        c++;  
        System.out.println("a= " + a);  
        System.out.println("b= " + b);  
        System.out.println("c= " + c);  
    }  
}
```

Relational Operators

Relational operators determine the relationship that one operand has to the other operand, specifically equality and ordering.

The outcome is always a value of type `boolean`.

They are most often used in branching and loop control statements.

Table: Relational Operators

==	equals to	apply to any type
!=	not equal to	apply to any type
>	greater than	apply to numerical types only
<	less than	apply to numerical types only
>=	greater than or equal	apply to numerical types only
<=	less than or equal	apply to numerical types only

Logical Operators

Logical operators act upon `boolean` operands only.

The outcome is always a value of type `boolean`.

In particular, “and” and “or” logical operators occur in two forms:

- 1) full – `op1 & op2` and `op1 | op2` where both `op1` and `op2` are evaluated
- 2) short-circuit - `op1 && op2` and `op1 || op2` where `op2` is only evaluated if the value of `op1` is insufficient to determine the final outcome

Table: Logical Operators

&	op1 & op2	logical AND
	op1 op2	logical OR
&&	op1 && op2	short-circuit AND
	op1 op2	short-circuit OR
!	! op	logical NOT
^	op1 ^ op2	logical XOR

Example: Logical Operators

```
class LogicalDemo {  
  
    public static void main(String[] args) {  
        int n = 2;  
        if (n != 0 && n / 0 > 10)  
            System.out.println("This is true");  
        else  
            System.out.println("This is false");  
    }  
  
}
```

Bitwise Operators

Bitwise operators apply to integer types only.

They act on individual bits of their operands.

There are three kinds of bitwise operators:

- 1) basic – bitwise `AND`, `OR`, `NOT` and `XOR`
- 2) shifts – left, right and right-zero-fill
- 3) assignments – bitwise assignment for all basic and shift operators

Table: Bitwise Operators

~	~ op	inverts all bits of its operand
&	op1 & op2	produces 1 bit if both operands are 1
	op1 op2	produces 1 bit if either operand is 1
^	op1 ^ op2	produces 1 bit if exactly one operand is 1
>>	op1 >> op2	shifts all bits in op1 right by the value of op2
<<	op1 << op2	shifts all bits in op1 left by the value of op2
>>>	op1 >>> op2	shifts op1 right by op2 value, write zero on the left

Example: Bitwise Operators

```
class BitLogic {
    public static void main(String args[]) {
        String binary[] = { "0000", "0001", "0010", ... };
        int a = 3; // 0 + 2 + 1 or 0011 in binary
        int b = 6; // 4 + 2 + 0 or 0110 in binary
        int c = a | b;
        int d = a & b;
        int e = a ^ b;
        System.out.print("a =" + binary[a]);
        System.out.println("and b =" + binary[b]);
        System.out.println("a|b = " + binary[c]);
        System.out.println("a&b = " + binary[d]);
        System.out.println("a^b = " + binary[e]);
    }
}
```

Other Operators

<code>? :</code>	shortcut if-else statement
<code>[]</code>	used to declare arrays, create arrays, access array elements
<code>.</code>	used to form qualified names
<code>(params)</code>	delimits a comma-separated list of parameters
<code>(type)</code>	casts a value to the specified type
<code>new</code>	creates a new object or a new array
<code>instanceof</code>	determines if its first operand is an instance of the second

Conditional Operator

General form:

`expr1? expr2 : expr3`

where:

- 1) `expr1` is of type boolean
- 2) `expr2` and `expr3` are of the same type

If `expr1` is true, `expr2` is evaluated, otherwise `expr3` is evaluated.

Example: Conditional Operator

```
class Ternary {  
    public static void main(String args[]) {  
        int i, k;  
  
        i = 10;  
        k = i < 0 ? -i : i;  
        System.out.print("Abs value of " + i + " is " + k);  
  
        i = -10;  
        k = i < 0 ? -i : i;  
        System.out.print("Abs value of " + i + " is " + k);  
    }  
}
```

Operator Precedence

Java operators are assigned precedence order.

Precedence determines that the expression

```
1 + 2 * 6 / 3 > 4 && 1 < 0
```

if equivalent to

```
(( (1 + ((2 * 6) / 3)) > 4) && (1 < 0))
```

When operators have the same precedence, the earlier one binds stronger.

Table: Operator Precedence

highest			
()	[]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
&&			
? :			
=	op=		
lowest			

Exercise: Operators

- 1) What operators do the code snippet below contain?

```
arrayOfInts[j] > arrayOfInts[j+1];
```

- 2) Consider the following code snippet:

```
int i = 10;
```

```
int n = i++%5;
```

- a) What are the values of *i* and *n* after the code is executed?
- b) What are the final values of *i* and *n* if instead of using the postfix increment operator (*i++*), you use the prefix version (*++i*)?
- 3) What is the value of *i* after the following code snippet executes?

```
int i = 8;
```

```
i >>=2;
```

- 4) What's the result of `System.out.println(010| 4);` ?

Control Flow

Course Outline

- | | | |
|--|--|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|--|--|---|

Control Flow

Writing a program means typing statements into a file.

Without control flow, the interpreter would execute these statements in the order they appear in the file, left-to-right, top-down.

Control flow statements, when inserted into the text of the program, determine in which order the program should be executed.

Control Flow Statements

Java control statements cause the flow of execution to advance and branch based on the changes to the state of the program.

Control statements are divided into three groups:

- 1) **selection** statements allow the program to choose different parts of the execution based on the outcome of an expression
- 2) **iteration** statements enable program execution to repeat one or more statements
- 3) **jump** statements enable your program to execute in a non-linear fashion

Selection Statements

Java selection statements allow to control the flow of program's execution based upon conditions known only during run-time.

Java provides four selection statements:

- 1) `if`
- 2) `if-else`
- 3) `if-else-if`
- 4) `switch`

if Statement

General form:

```
if (expression) statement
```

If `expression` evaluates to `true`, execute `statement`, otherwise do nothing.

The expression must be of type `boolean`.

Simple/Compound Statement

The component statement may be:

1) simple

```
if (expression) statement;
```

2) compound

```
if (expression) {  
    statement;  
}
```

if-else Statement

Suppose you want to perform two different statements depending on the outcome of a `boolean` expression. `if-else` statement can be used.

General form:

```
if (expression) statement1  
else statement2
```

Again, `statement1` and `statement2` may be simple or compound.

if-else-if Statement

General form:

```
if (expression1) statement1
else if (expression2) statement2
else if (expression3) statement3
...
else statement
```

Semantics:

- 1) statements are executed top-down
- 2) as soon as one expressions is true, its statement is executed
- 3) if none of the expressions is true, the last statement is executed

Example: if-else-if

```
class IfElse {  
    public static void main(String args[]) {  
        int month = 4;  
        String season;  
        if (month == 12 || month == 1 || month == 2)  
            season = "Winter";  
        else if(month == 3 || month == 4 || month == 5)  
            season = "Spring";  
        else if(month == 6 || month == 7 || month == 8)  
            season = "Summer";  
        else if(month == 9 || month == 10 || month == 11)  
            season = "Autumn";  
        else season = "Bogus Month";  
        System.out.println("April is in the " + season + ".");  
    }  
}
```

switch Statement

`switch` provides a better alternative than `if-else-if` when the execution follows several branches depending on the value of an expression.

General form:

```
switch (expression) {  
    case value1: statement1; break;  
    case value2: statement2; break;  
    case value3: statement3; break;  
    ...  
    default: statement;  
}
```

switch Assumptions/Semantics

Assumptions:

- 1) `expression` must be of type `byte`, `short`, `int` or `char`
- 2) each of the `case` values must be a literal of the compatible type
- 3) `case` values must be unique

Semantics:

- 1) `expression` is evaluated
- 2) its value is compared with each of the `case` values
- 3) if a match is found, the statement following the `case` is executed
- 4) if no match is found, the statement following `default` is executed

`break` makes sure that only the matching statement is executed.

Both `default` and `break` are optional.

Example: switch 1

```
class Switch {  
    public static void main(String args[]) {  
        int month = 4;  
        String season;  
        switch (month) {  
            case 12:  
            case 1:  
            case 2: season = "Winter"; break;  
            case 3:  
            case 4:  
            case 5: season = "Spring"; break;  
            case 6:  
            case 7:  
            case 8: season = "Summer"; break;
```

Example: switch 2

```
        case 9:  
        case 10:  
        case 11: season = "Autumn"; break;  
        default: season = "Bogus Month";  
    }  
    System.out.println("April is in " + season + ".");  
}  
}
```

Nested switch Statement

A switch statement can be nested within another switch statement:

```
switch(count) {  
    case 1:  
        switch(target) {  
            case 0: System.out.println("target is zero");  
                break;  
            case 1: System.out.println("target is one");  
                break;  
        }  
        break;  
    case 2: ...  
}
```

Since, every switch statement defines its own block, no conflict arises between the case constants in the inner and outer switch statements.

Comparing switch and if

Two main differences:

- 1) switch can only test for equality, while if can evaluate any kind of boolean expression
- 2) Java creates a “jump table” for switch expressions, so a switch statement is usually more efficient than a set of nested if statements

Iteration Statements

Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.

Java provides three iteration statements:

- 1) `while`
- 2) `do-while`
- 3) `for`

while Statement

General form:

```
while (expression) statement
```

where `expression` must be of type `boolean`.

Semantics:

- 1) repeat execution of `statement` until `expression` becomes false
- 2) `expression` is always evaluated before `statement`
- 3) if `expression` is false initially, `statement` will never get executed

Simple/Compound Statement

The component statement may be:

1) simple

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

2) compound

```
while (expression) {  
    statement;  
}
```

Example: while

```
class MidPoint {  
    public static void main(String args[]) {  
        int i, j;  
        i = 100;  
        j = 200;  
        while(++i < --j) {  
            System.out.println("i is " + i);  
            System.out.println("j is " + j);  
        }  
        System.out.println("The midpoint is " + i);  
    }  
}
```

do-while Statement

If a component statement has to be executed at least once, the `do-while` statement is more appropriate than the `while` statement.

General form:

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

where `expression` must be of type `boolean`.

Semantics:

- 1) repeat execution of `statement` until `expression` becomes false
- 2) `expression` is always evaluated after `statement`
- 3) even if `expression` is false initially, `statement` will be executed

Example: do-while

```
class DoWhile {  
    public static void main(String args[]) {  
        int i;  
        i = 0;  
        do  
            i++;  
        while ( 1/i < 0.001);  
        System.out.println("i is " + i);  
    }  
}
```

for Statement

When iterating over a range of values, `for` statement is more suitable to use than `while` or `do-while`.

General form:

```
for (initialization; termination; increment)
    statement
```

where:

- 1) `initialization` statement is executed once before the first iteration
- 2) `termination` expression is evaluated before each iteration to determine when the loop should terminate
- 3) `increment` statement is executed after each iteration

for Statement Semantics

This is how the for statement is executed:

- 1) `initialization` is executed once
- 2) `termination` expression is evaluated:
 - a) if false, the statement terminates
 - b) otherwise, continue to (3)
- 3) `increment` statement is executed
- 4) component `statement` is executed
- 5) control flow continues from (2)

Loop Control Variable

The `for` statement may include declaration of a loop control variable:

```
for (int i = 0; i < 1000; i++) {  
    ...  
}
```

The variable does not exist outside the `for` statement.

Example: for

```
class FindPrime {  
    public static void main(String args[]) {  
        int num = 14;  
        boolean isPrime = true;  
        for (int i=2; i < num/2; i++) {  
            if ((num % i) == 0) {  
                isPrime = false;  
                break;  
            }  
        }  
        if (isPrime) System.out.println("Prime");  
        else System.out.println("Not Prime");  
    }  
}
```

Many Initialization/Iteration Parts

The `for` statement may include several `initialization` and `iteration` parts.

Parts are separated by a comma:

```
int a, b;
```

```
for (a = 1, b = 4; a < b; a++, b--) {  
    ...  
}
```

for Statement Variations

The `for` statement need not have all components:

```
class ForVar {  
    public static void main(String args[]) {  
        int i = 0;  
        boolean done = false;  
        for( ; !done; ) {  
            System.out.println("i is " + i);  
            if(i == 10) done = true;  
            i++;  
        }  
    }  
}
```

Empty for

In fact, all three components may be omitted:

```
public class EmptyFor {  
    public static void main(String[] args) {  
        int i = 0;  
        for (; ; ) {  
            System.out.println("Infinite Loop " + i);  
        }  
    }  
}
```

Jump Statements

Java jump statements enable transfer of control to other parts of program.

Java provides three jump statements:

- 1) `break`
- 2) `continue`
- 3) `return`

In addition, Java supports **exception handling** that can also alter the control flow of a program. Exception handling will be explained in its own section.

break Statement

The break statement has three uses:

- 1) to terminate a case inside the switch statement
- 2) to exit an iterative statement
- 3) to transfer control to another statement

(1) has been described.

We continue with (2) and (3).

Loop Exit with break

When `break` is used inside a loop, the loop terminates and control is transferred to the following instruction.

```
class BreakLoop {  
    public static void main(String args[]) {  
        for (int i=0; i<100; i++) {  
            if (i == 10) break;  
            System.out.println("i: " + i);  
        }  
        System.out.println("Loop complete.");  
    }  
}
```


break in Nested Loops

Used inside nested loops, `break` will only terminate the innermost loop:

```
class NestedLoopBreak {  
    public static void main(String args[]) {  
        for (int i=0; i<3; i++) {  
            System.out.print("Pass " + i + ": ");  
            for (int j=0; j<100; j++) {  
                if (j == 10) break;  
                System.out.print(j + " ");  
            }  
            System.out.println();  
        }  
        System.out.println("Loops complete.");  
    }  
}
```

Control Transfer with break

Java does not have an unrestricted “goto” statement, which tends to produce code that is hard to understand and maintain.

However, in some places, the use of gotos is well justified. In particular, when breaking out from the deeply nested blocks of code.

`break` occurs in two versions:

- 1) unlabelled
- 2) labeled

The labeled `break` statement is a “civilized” replacement for goto.

Labeled break

General form:

```
break label;
```

where `label` is the name of a label that identifies a block of code:

```
label: { ... }
```

The effect of executing `break label;` is to transfer control immediately after the block of code identified by `label`.

Example: Labeled break

```
class Break {  
    public static void main(String args[]) {  
        boolean t = true;  
        first: {  
            second: {  
                third: {  
                    System.out.println("Before the break.");  
                    if (t) break second;  
                    System.out.println("This won't execute");  
                }  
                System.out.println("This won't execute");  
            }  
            System.out.println("After second block.");  
        }  
    }  
}
```

Example: Nested Loop break

```
class NestedLoopBreak {
    public static void main(String args[]) {
        outer: for (int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
            for (int j=0; j<100; j++) {
                if (j == 10) break outer; // exit both loops
                System.out.print(j + " ");
            }
            System.out.println("This will not print");
        }
        System.out.println("Loops complete.");
    }
}
```

break Without Label

It is not possible to break to any label which is not defined for an enclosing block. Trying to do so will result in a compiler error.

```
class BreakError {
    public static void main(String args[]) {
        one: for(int i=0; i<3; i++) {
            System.out.print("Pass " + i + ": ");
        }
        for (int j=0; j<100; j++) {
            if (j == 10) break one;
            System.out.print(j + " ");
        }
    }
}
```

continue Statement

The `break` statement terminates the block of code, in particular it terminates the execution of an iterative statement.

The `continue` statement forces the early termination of the current iteration to begin immediately the next iteration.

Like `break`, `continue` has two versions:

- 1) unlabelled – continue with the next iteration of the current loop
- 2) labeled – specifies which enclosing loop to continue

Example: Unlabeled continue

```
class Continue {  
    public static void main(String args[]) {  
        for (int i=0; i<10; i++) {  
            System.out.print(i + " ");  
            if (i%2 == 0) continue;  
            System.out.println("");  
        }  
    }  
}
```


Example: Labeled continue

```
class LabeledContinue {
    public static void main(String args[]) {
        outer: for (int i=0; i<10; i++) {
            for (int j=0; j<10; j++) {
                if (j > i) {
                    System.out.println();
                    continue outer;
                }
                System.out.print(" " + (i * j));
            }
            System.out.println();
        }
    }
}
```

Return Statement

The return statement is used to return from the current method: it causes program control to transfer back to the caller of the method.

Two forms:

1) return without value

```
return;
```

2) return with value

```
return expression;
```

Example: Return

```
class Return {  
    public static void main(String args[]) {  
        boolean t = true;  
        System.out.println("Before the return.");  
        if (t) return; // return to caller  
        System.out.println("This won't execute.");  
    }  
}
```

Exercise: Control Flow

1) Write a program that prints all the prime numbers between 1 and 49 to the console.

2) Write a program that prints the first 20 Fibonacci numbers.

The Fibonacci numbers are defined as follows:

The zeroth Fibonacci number is 1.

The first Fibonacci number is also 1.

The second Fibonacci number is $1 + 1 = 2$.

The third Fibonacci number is $1 + 2 = 3$.

In other words, except for the first two numbers each Fibonacci number is the sum of the two previous numbers.

Object-Orientation

Course Outline

- | | | |
|---|---|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|---|---|

Data versus Operations


Data (**nouns**) versus operations (**verbs**):

```
data1  
...  
dataN
```



nouns

```
operation1 { ... }  
...  
operationN { ... }
```



verbs

Procedural Programming

Procedural programming is verb-oriented:

- 1) decomposition around *operations*
- 2) operation are divided into smaller operations

Programming Languages:

- 1) C
- 2) Pascal
- 3) Fortran, etc.

Example: Procedural Program

```
program AddNums {  
    Integer a;  
    Integer b;  
  
    a = 100;  
    b = 200;  
    midPoint(a, b);  
  
    procedure midPoint(Integer a, Integer b) {  
        while(a < b) {  
            println("a is " + a); a = a+1;  
            println("b is " + b); b = b-1;  
        }  
    }  
}
```

Drawbacks

- 1) data is given a second-class status when compared with operations
- 2) difficult to relate to the real world – there are no functions in real world
- 3) difficult to create new data types – reduces extensibility
- 4) programs are difficult to debug – little restriction to data access
- 5) programs are hard to understand – many variables have global scope
- 6) programs are hard to reuse – data/functions are mutually dependent
- 7) little support for developing and comprehending really large programs
- 8) top-down development approach tends to produce monolithic programs

What is an Object?

Real world objects are things that have:

- 1) state
- 2) behavior

Example: your dog:

- 1) state – name, color, breed, sits?, barks?, wags tail?, runs?
- 2) behavior – sitting, barking, wagging tail, running

A software **object** is a bundle of variables (state) and methods (operations).

What is a Class?

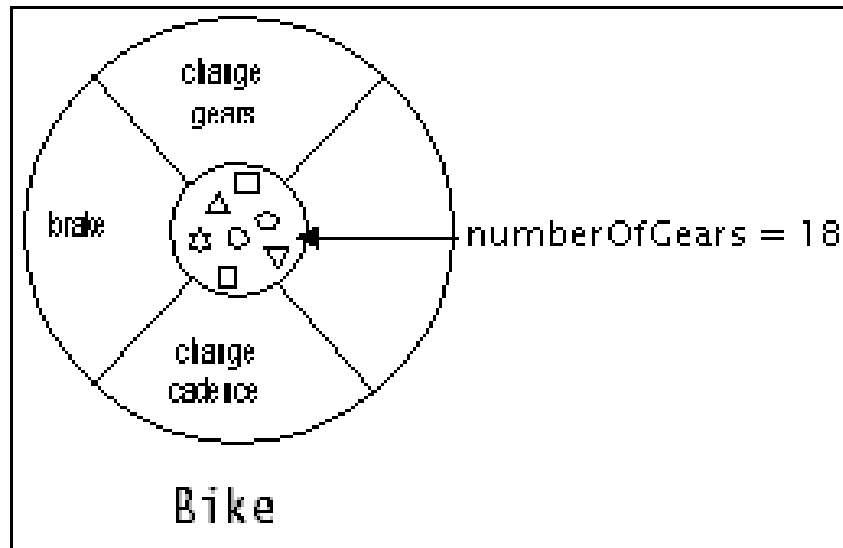
A **class** is a blueprint that defines the variables and methods common to all objects of a certain kind.

Example: 'your dog' is a object of the class Dog.

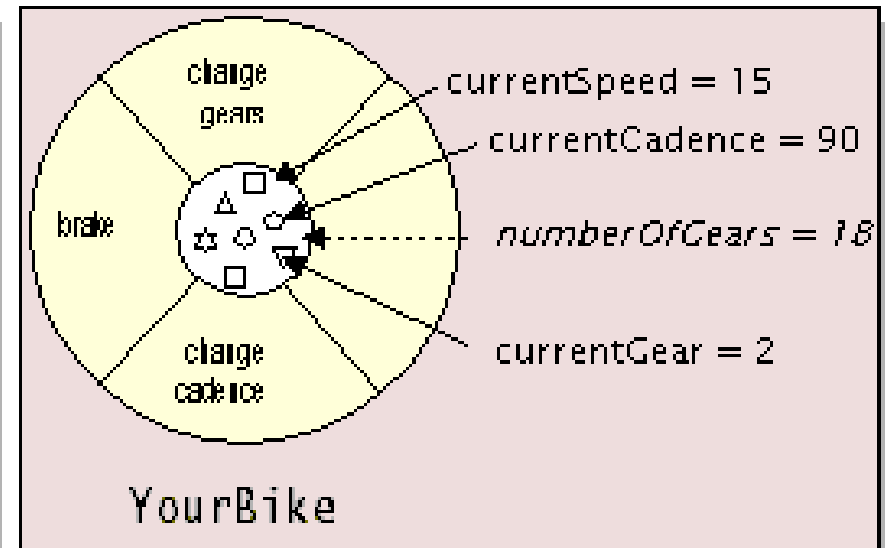
An object holds values for the variables defines in the class.

An object is called an **instance** of the Class

Objects versus Classes



Class



Instance of a Class

- 1) operations: `changeGears`, `brake`, `changeCadence`
- 2) variables: `currentSpeed`, `currentCadence`, `currentGear`
- 3) static variable: `numberOfGears`

It holds the same value for all objects of the class.

Object-Oriented Programming

Programming defined in terms:

- 1) objects (nouns) and
- 2) relationships between objects

Object-Oriented programming languages:

- 1) SmallTalk
- 2) C++
- 3) C#
- 4) Java

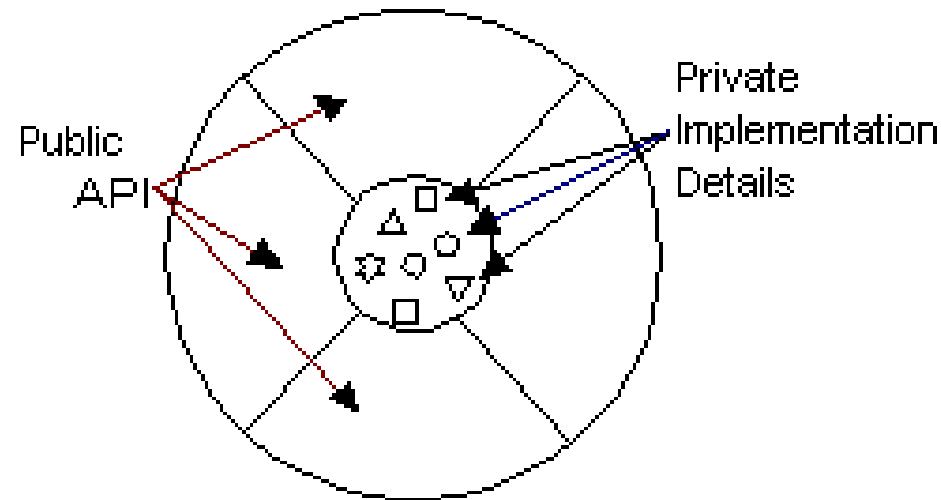
Object-Oriented Concepts

Three main concepts:

- 1) encapsulation
- 2) inheritance
- 3) polymorphism

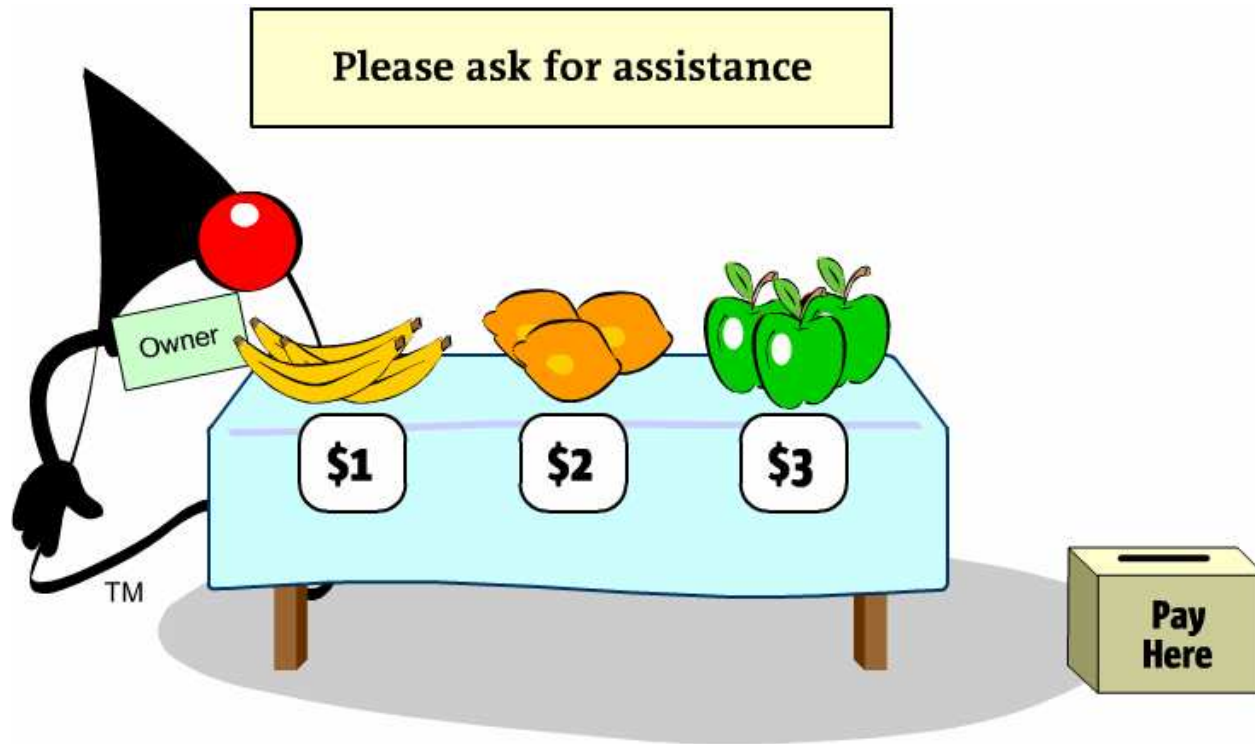
Encapsulation

Illustration:



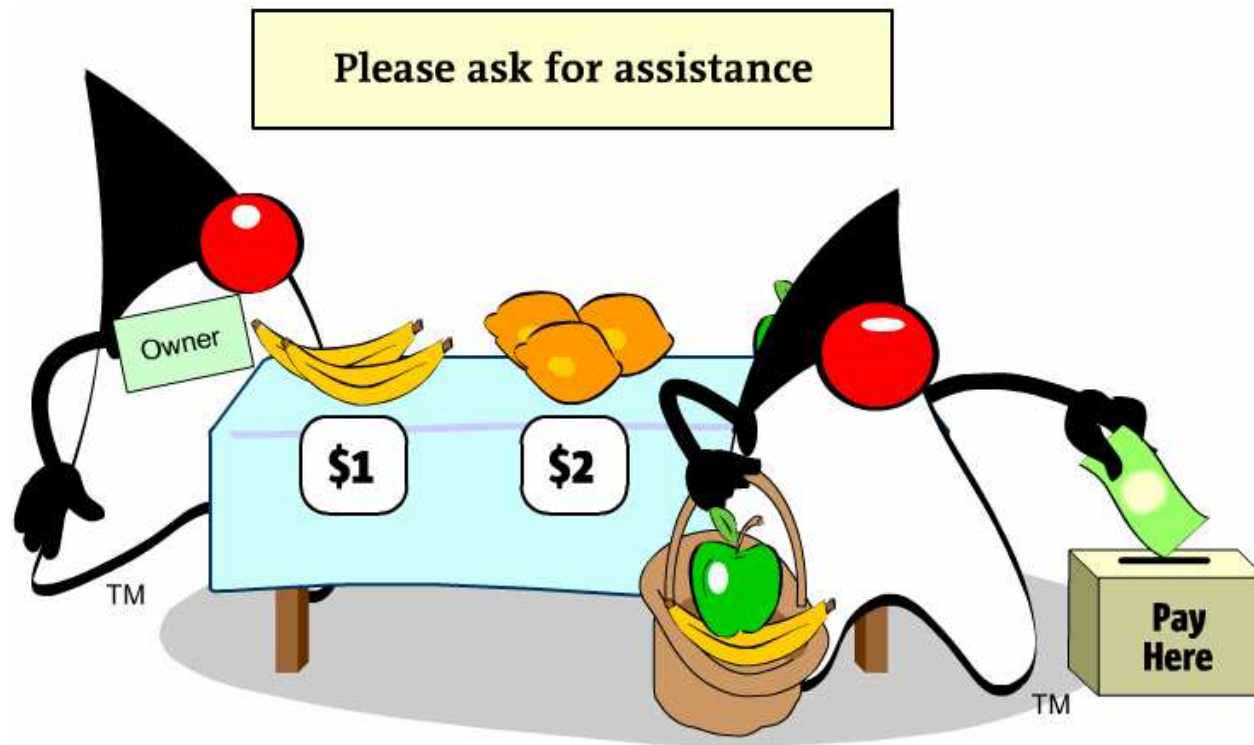
- Enclose data inside an object.
- Along with data, include operations on this data.
- Data cannot be accessed from outside except through the operations.
- Provides data security and facilitates code reuse.
- Operations provide the interface - internal state can change without affecting the user as long as the interface does not change.

Encapsulation 1



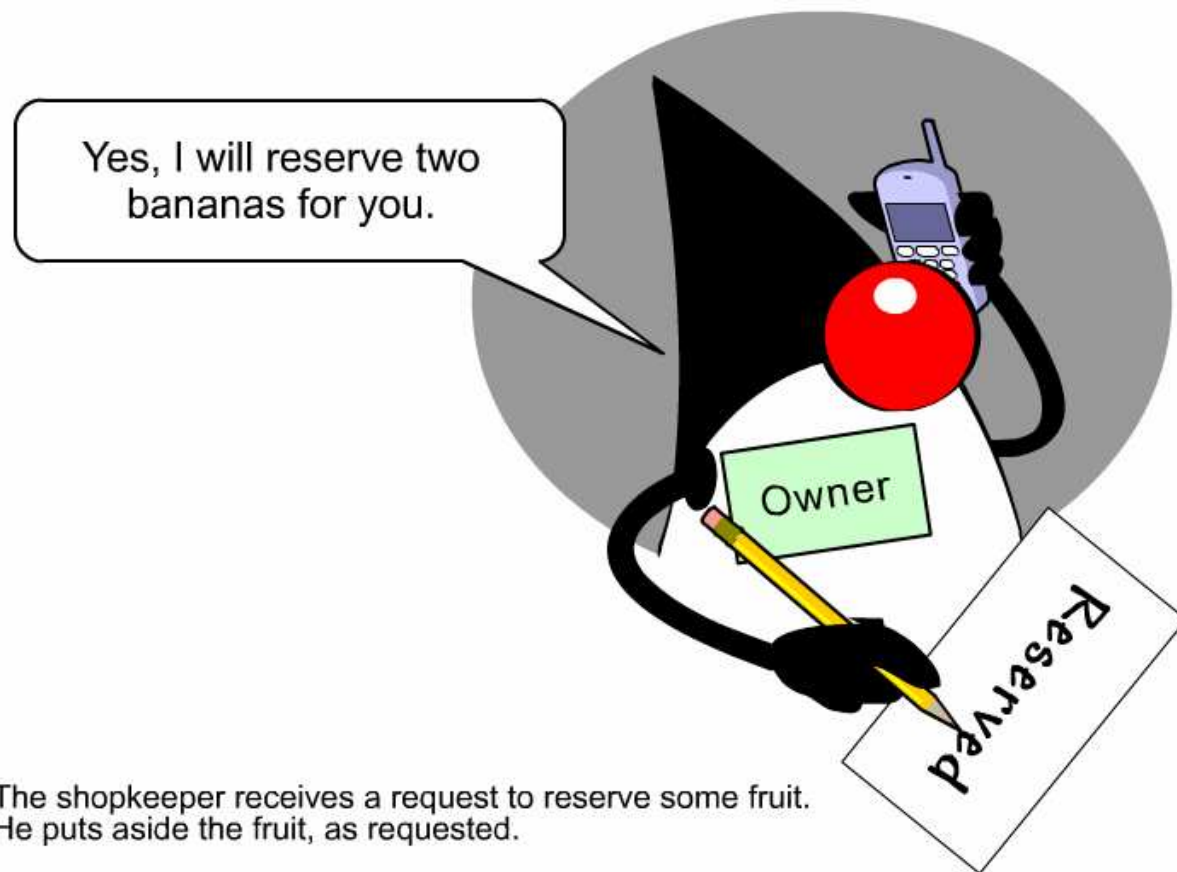
The shopkeeper's fruit stand is designed to serve all customers. However, the design of the fruit stand does not prevent self-service. The fruit is freely accessible to all customers.

Encapsulation 2

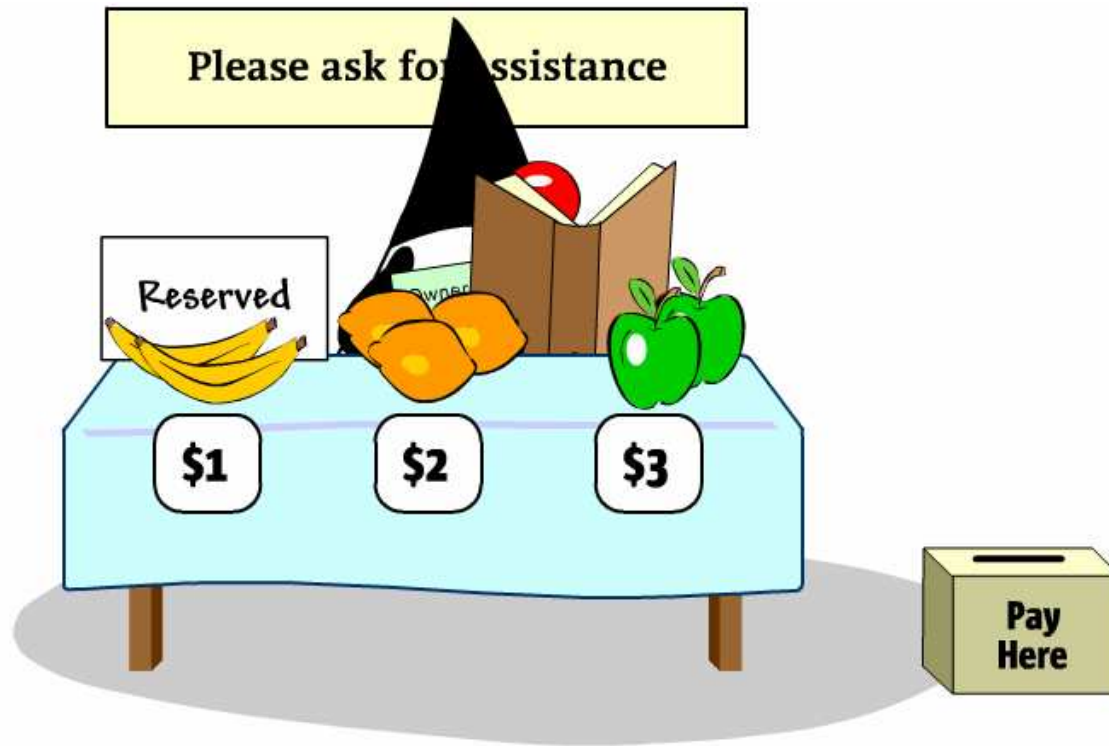


A customer can select some fruit and make payment, all without the shopkeeper's involvement.

Encapsulation 3

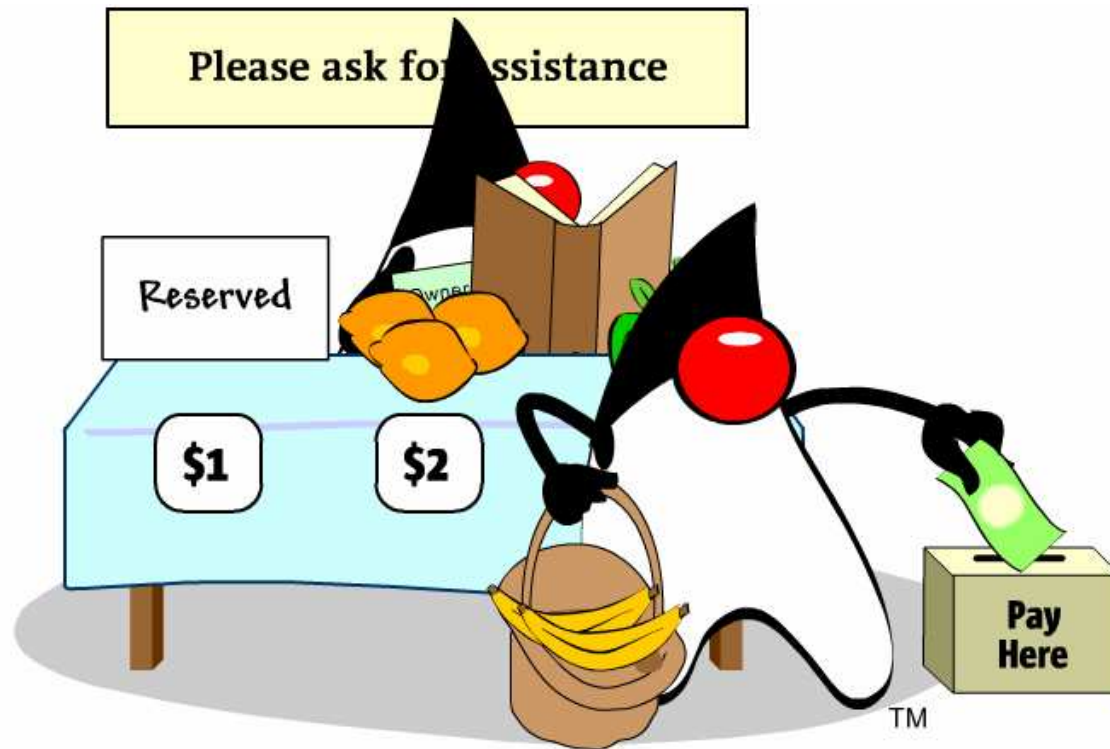


Encapsulation 4



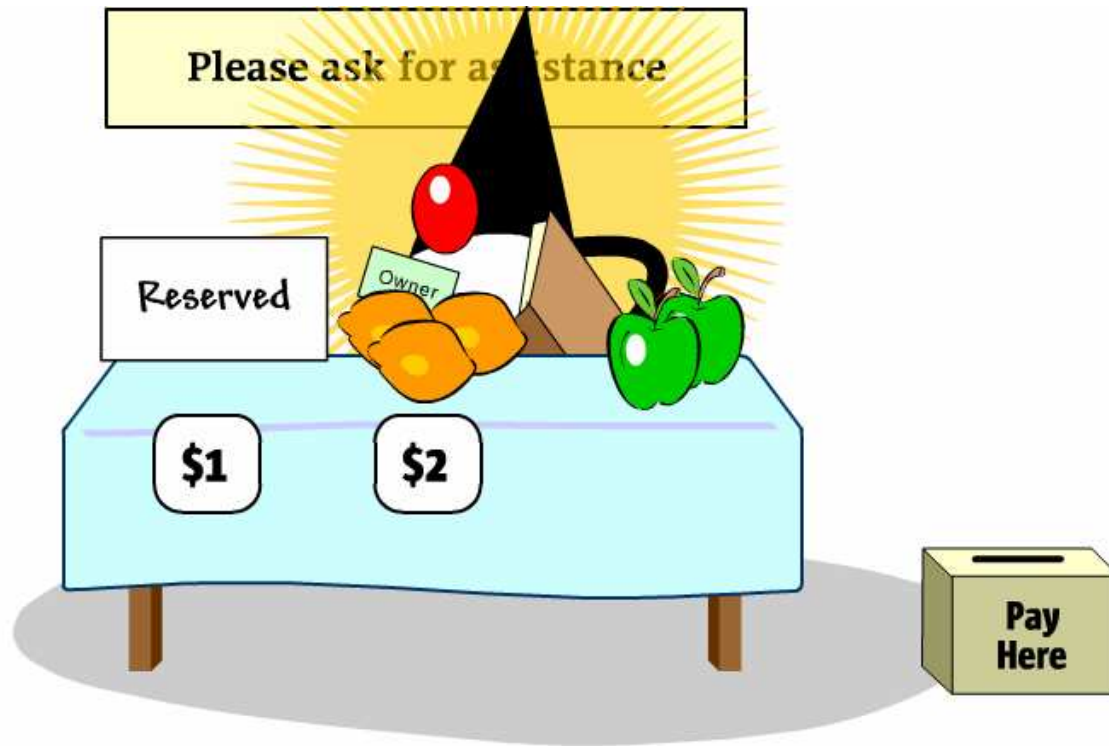
However, because of the fruit stand's design, a customer can choose the fruit that has been reserved.

Encapsulation 5



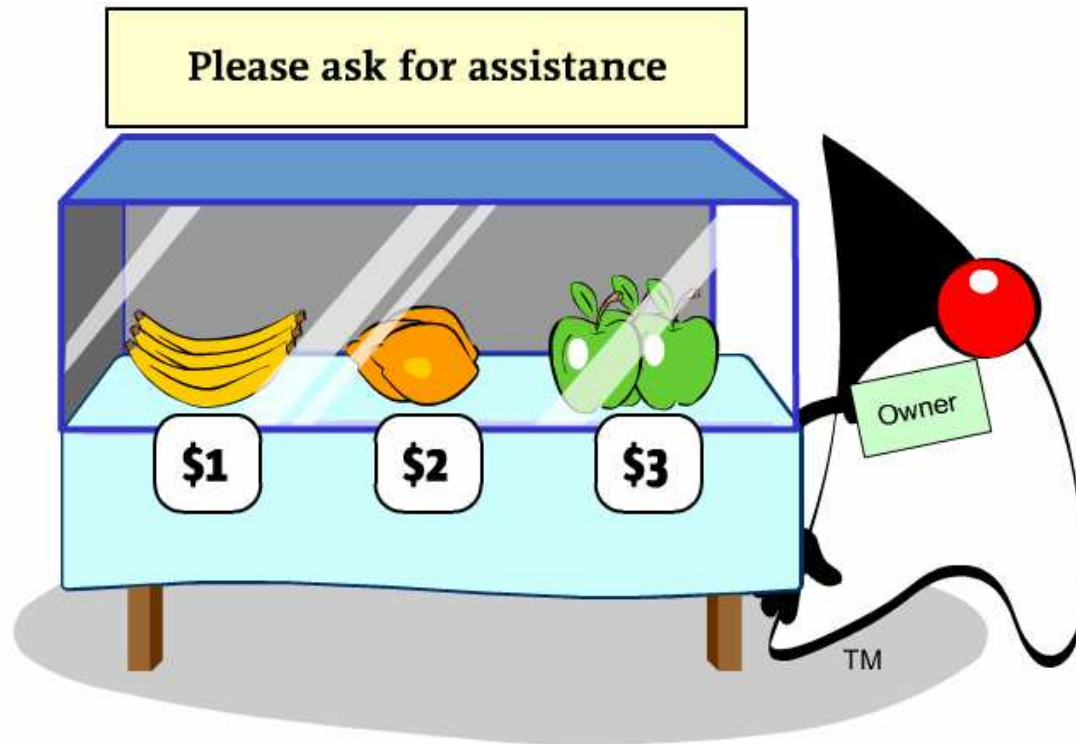
However, because of the fruit stand's design, a customer can choose the fruit that has been reserved.

Encapsulation 6



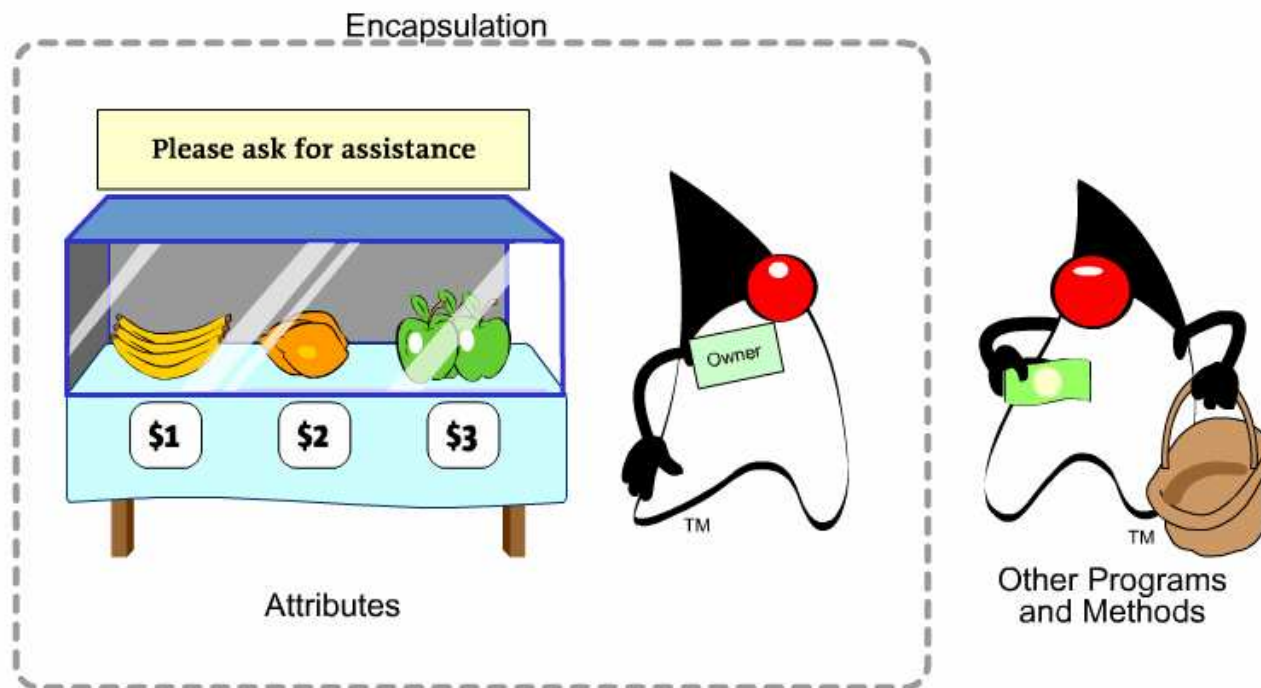
However, because of the fruit stand's design, a customer can choose the fruit that has been reserved.

Encapsulation 7



By enclosing the fruit behind glass, and controlling all access by customers, the shopkeeper has encapsulated the process of buying fruit.

Encapsulation 8

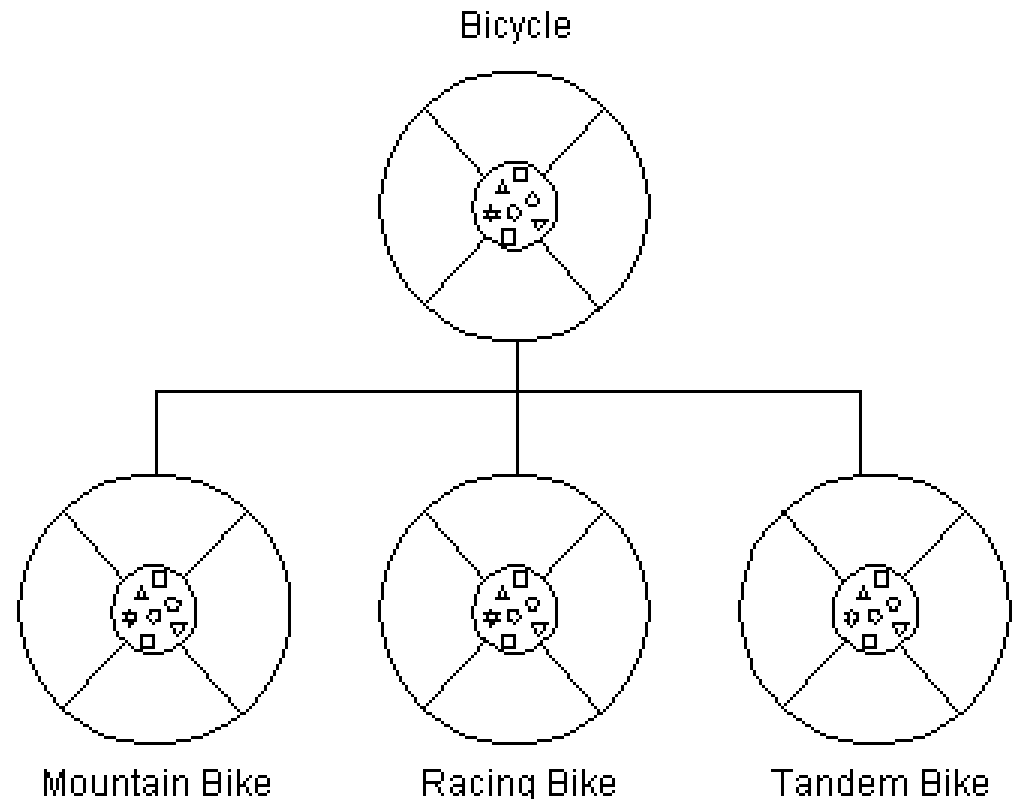


Encapsulation in the Java programming language is protection of the attributes from arbitrary access by other programs and methods. You control how access to attributes is accomplished.

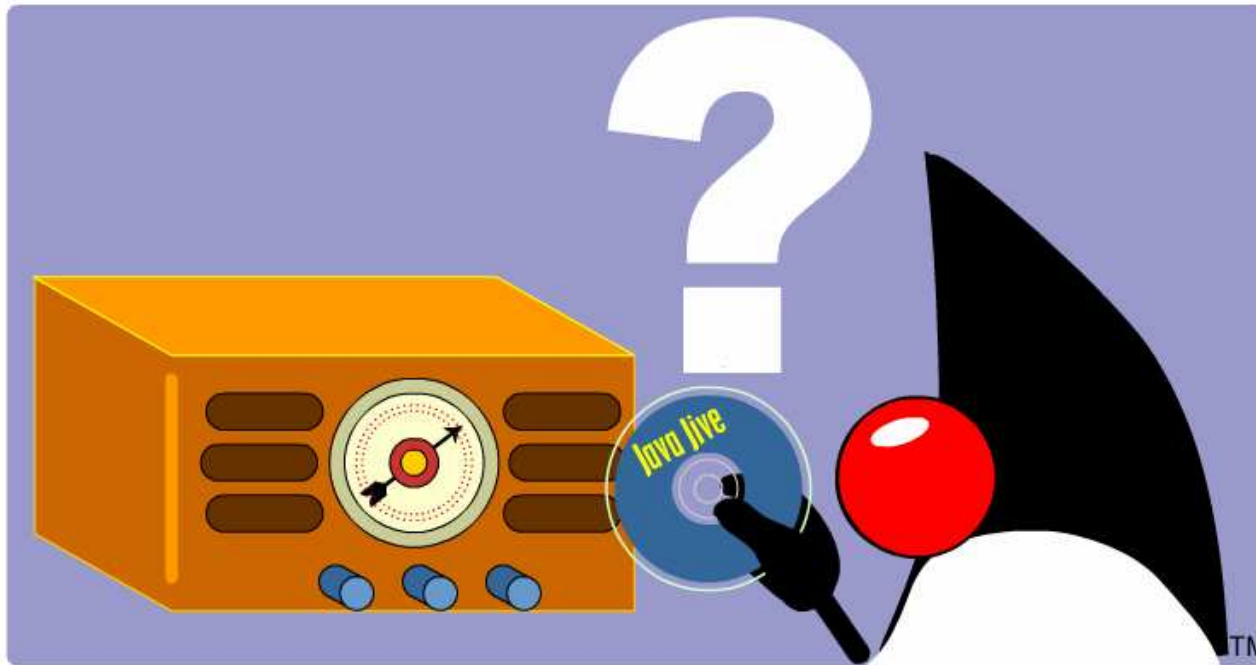
Inheritance

Features:

- 1) a class obtains variables and methods from another class
- 2) the former is called sub-class, the latter super-class
- 3) a sub-class provides a specialized behavior with respect to its super-class
- 4) inheritance facilitates code reuse and avoids duplication of data

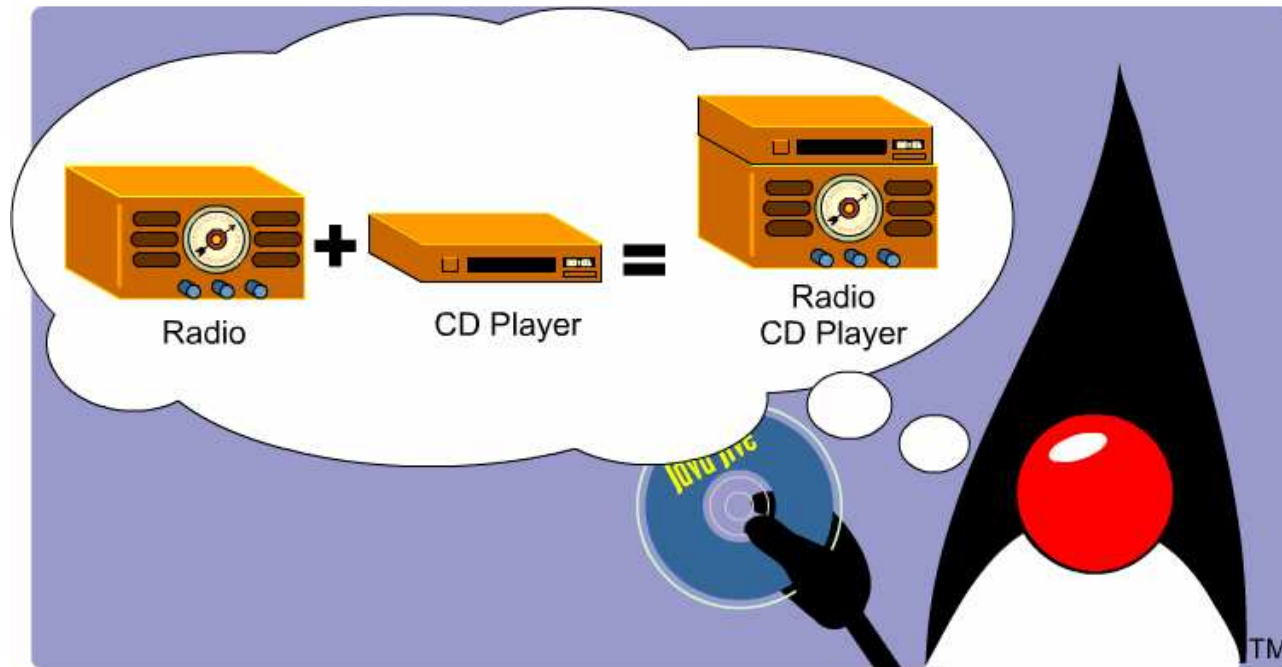


Inheritance 1



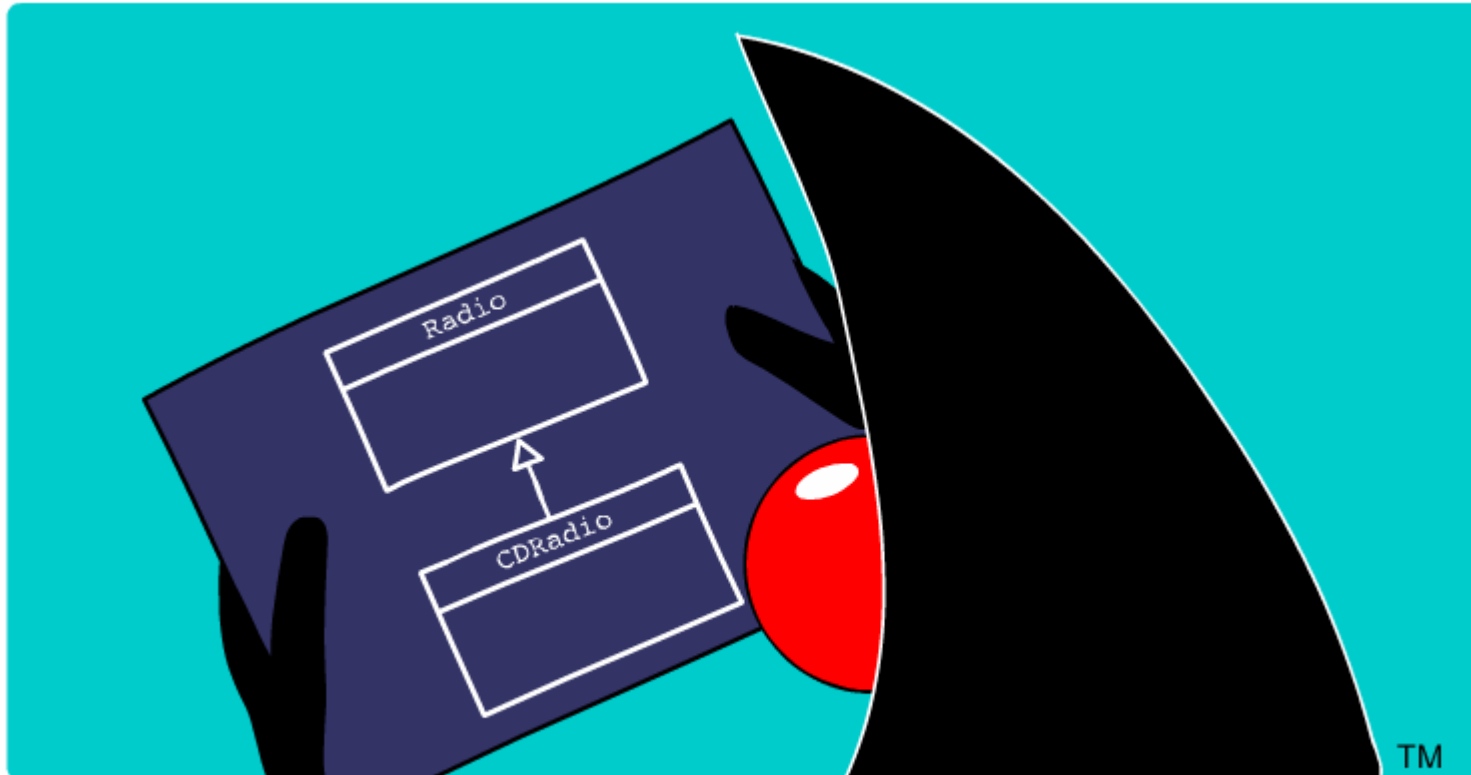
Duke would like to listen to his music CD, but his radio is not equipped with a CD player.

Inheritance 2



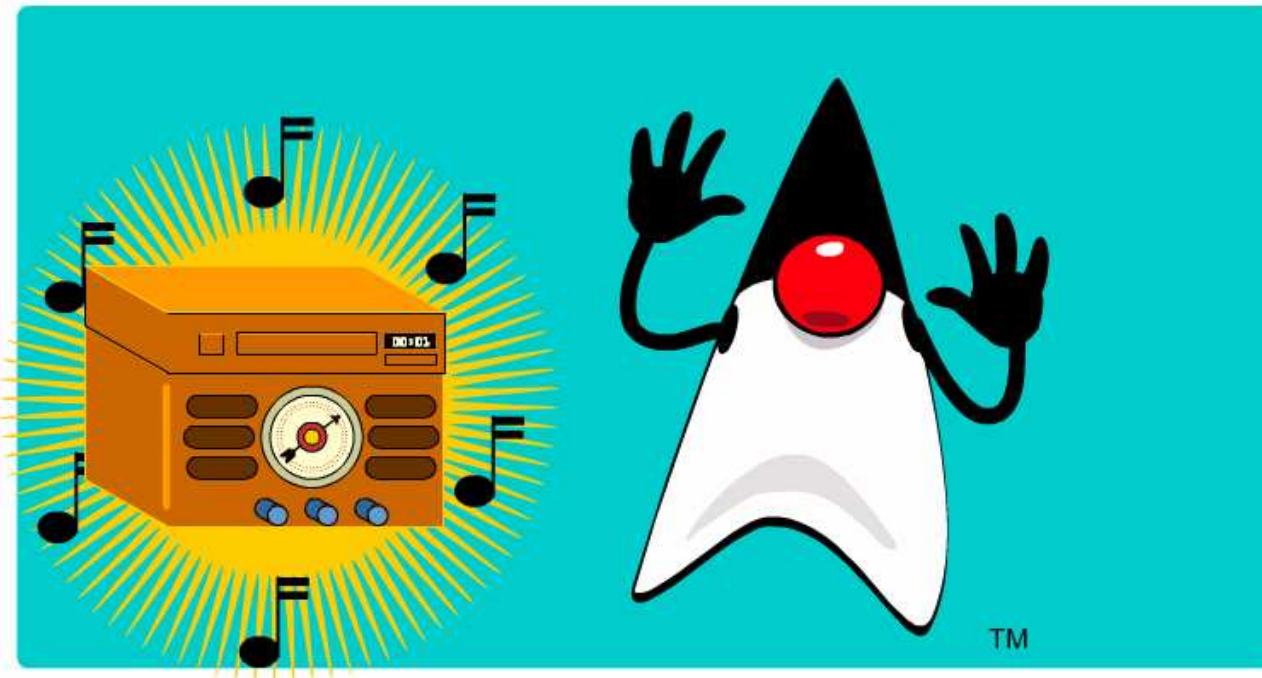
Duke decides he will create a specialized radio, one that plays CDs.

Inheritance 3



Duke creates a blueprint for a radio CD player.

Inheritance 4



Now Duke can enjoy listening to CDs on his radio CD player.

Polymorphism

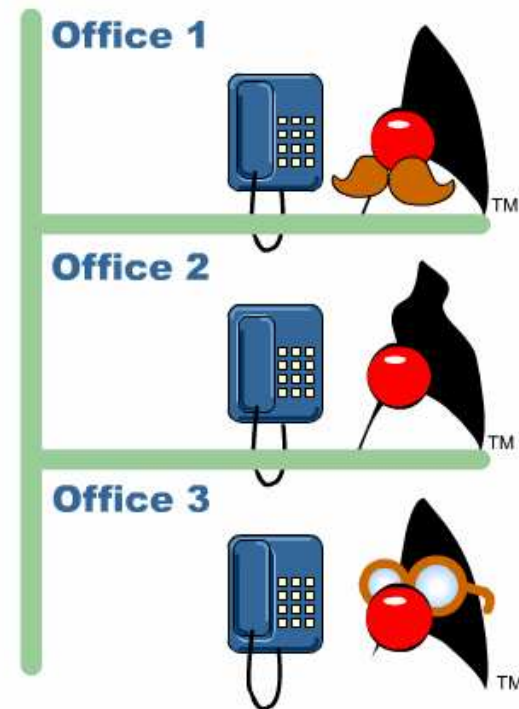
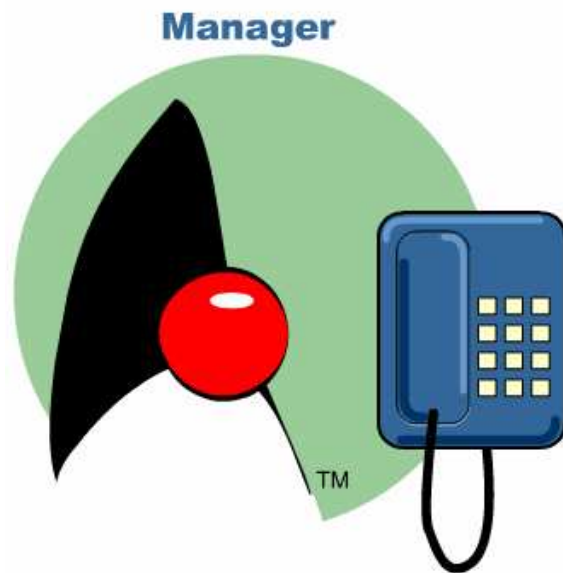
Polymorphism = many different (poly) forms of objects that share a common interface respond differently when a method of that interface is invoked.

Polymorphism is enabled by inheritance:

- 1) a super-class defines an interface that all sub-classes must follow
- 2) it is up to the sub-classes how this interface is implemented; a sub-class may override methods of its super-class

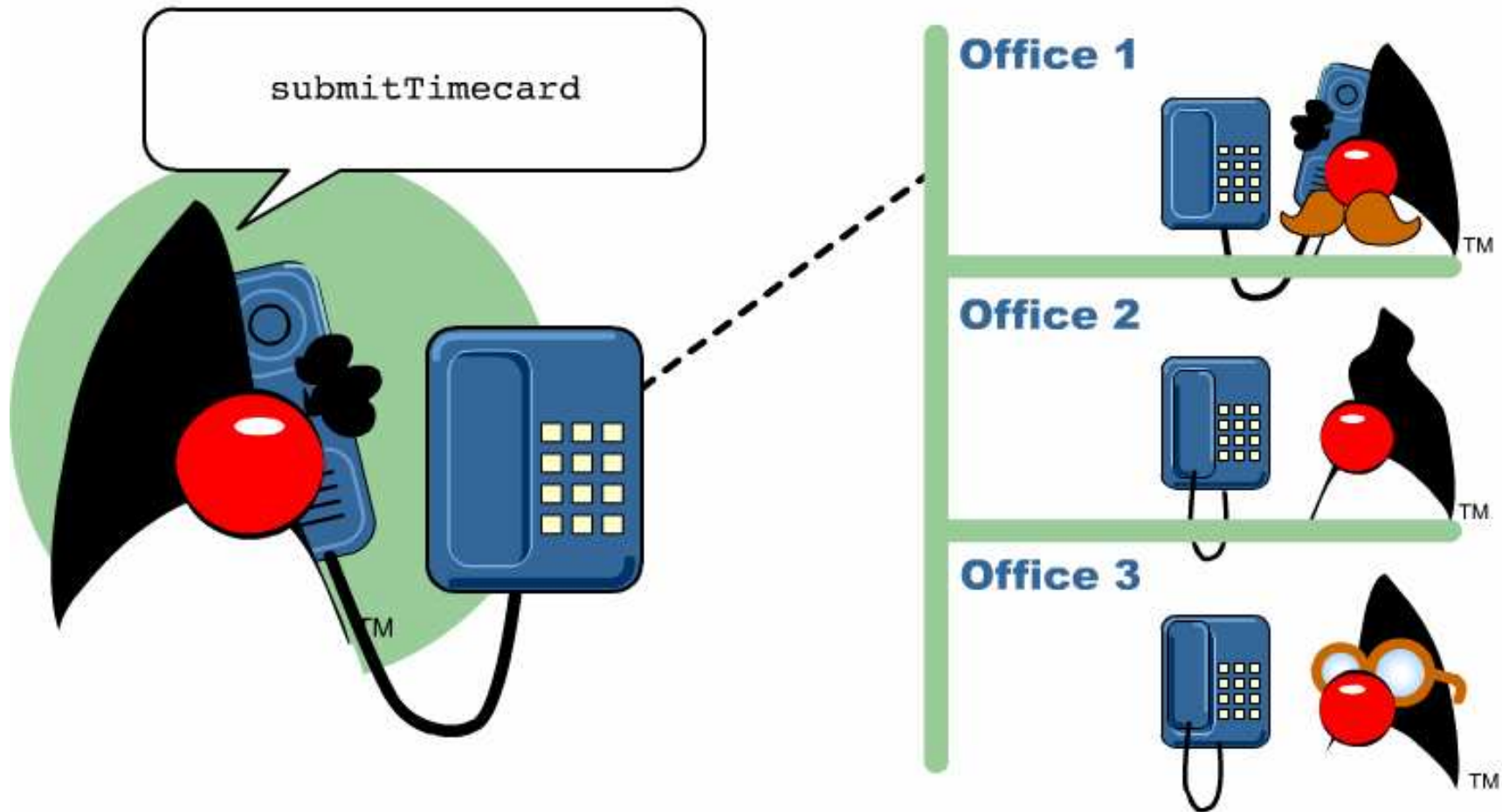
Therefore, objects from the classes related by inheritance may receive the same requests but respond to such requests in their own ways.

Polymorphism 1



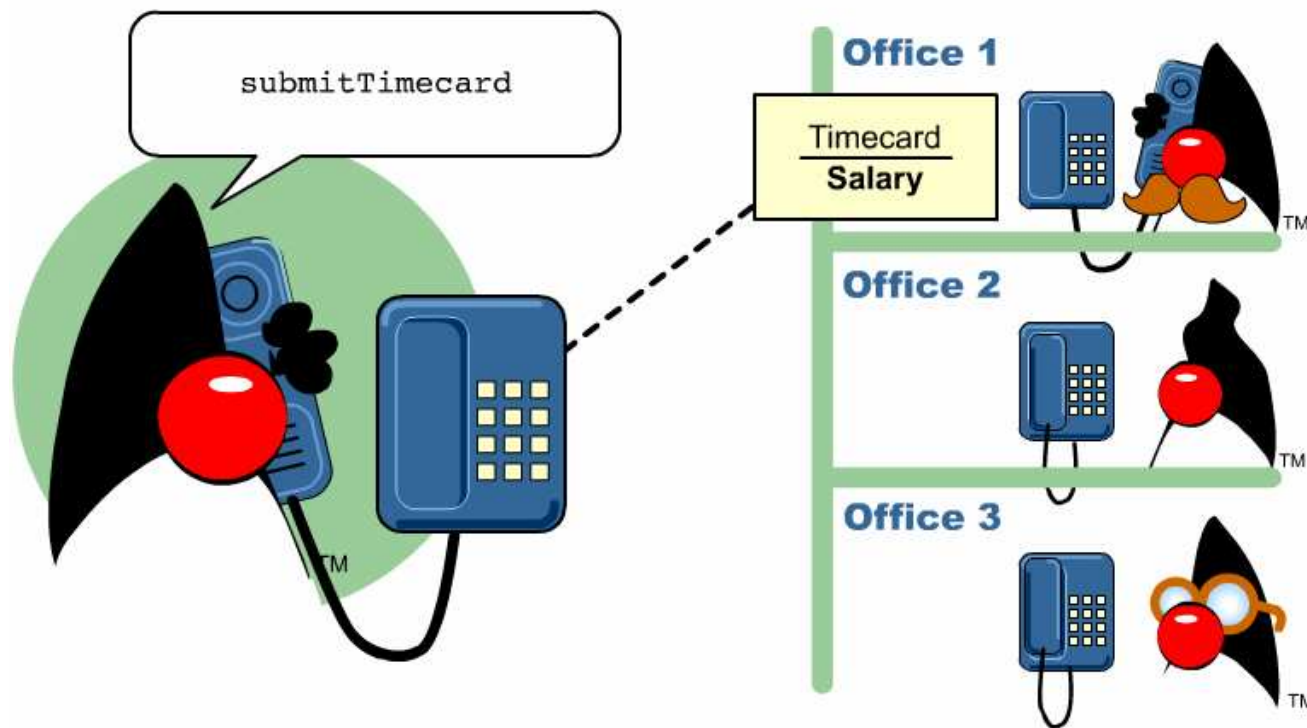
To illustrate polymorphism, imagine a manager with a phone connected to several employee offices.

Polymorphism 2



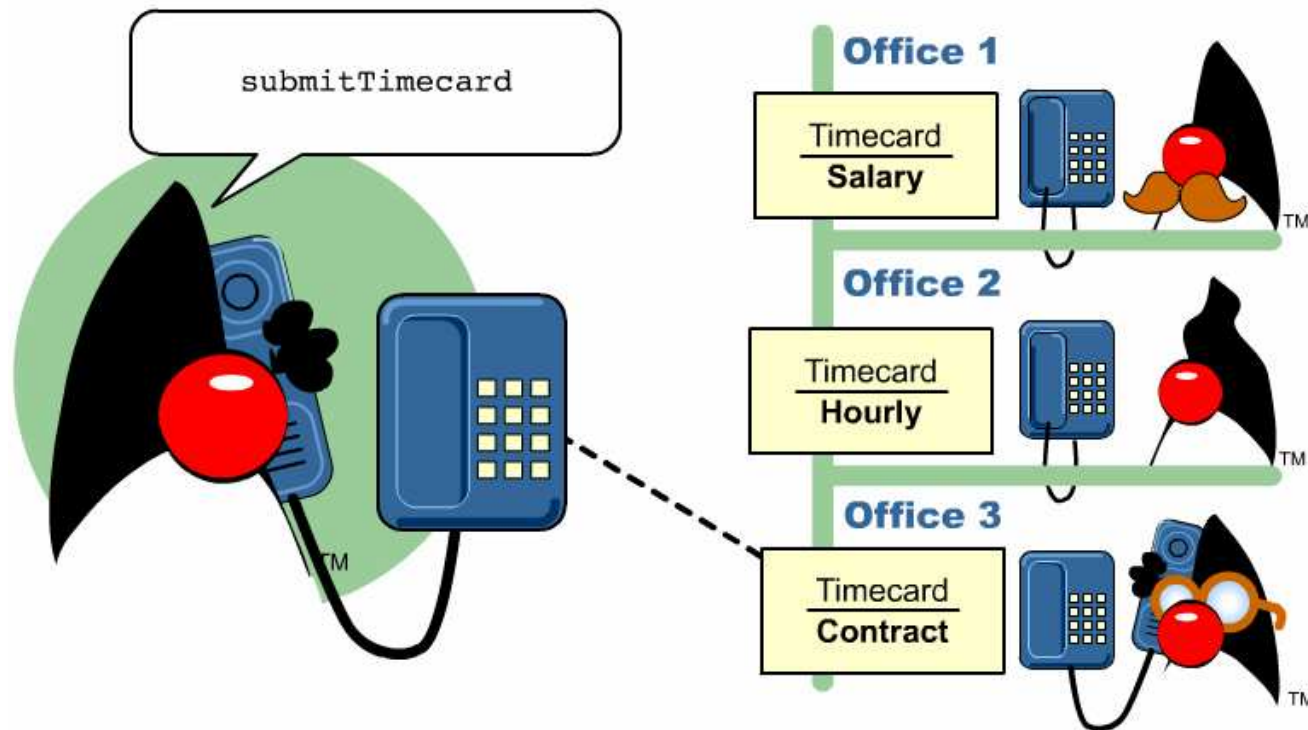
The manager places a call to the first office, and sends the `submitTimecard` method.

Polymorphism 3



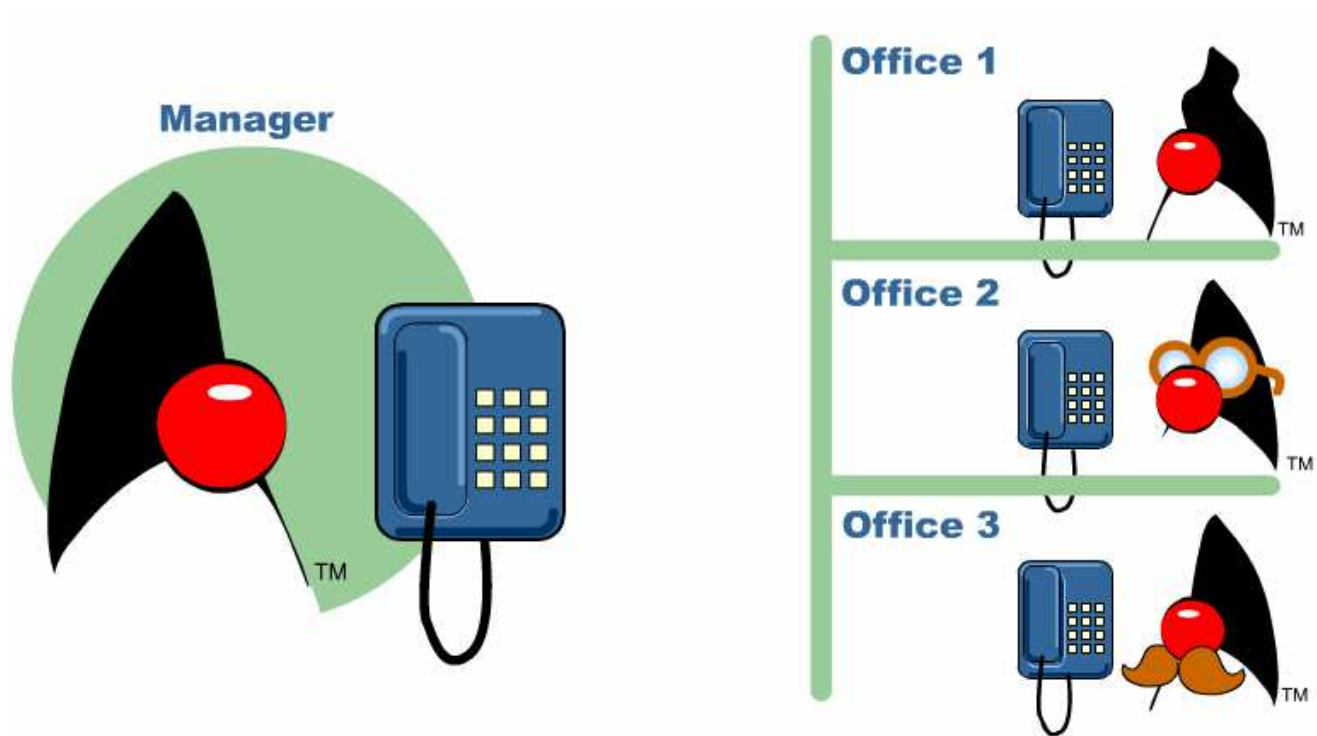
The employee in the first office is on salary. Salaried employees have their own method for submitting a timecard, which this employee follows.

Polymorphism 4



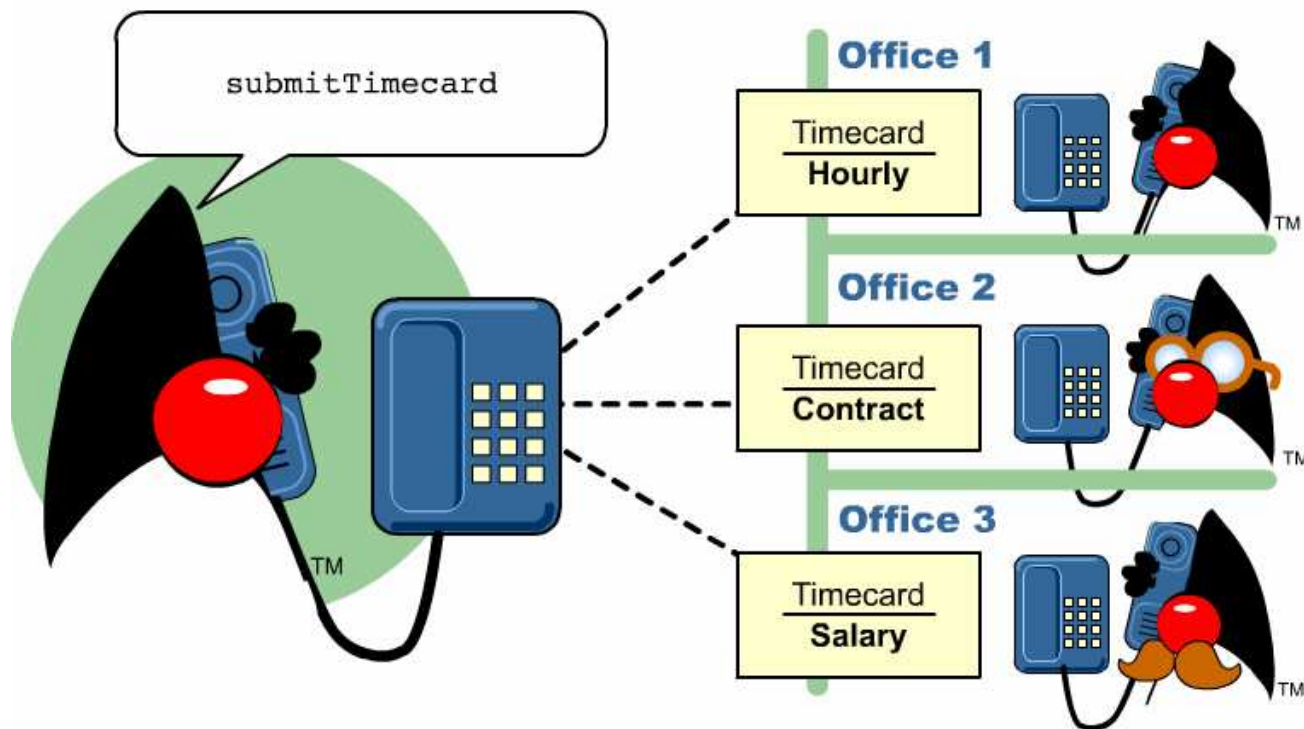
The manager places calls to each of the other offices. The employees in each office follow their own `submitTimecard` method.

Polymorphism 5



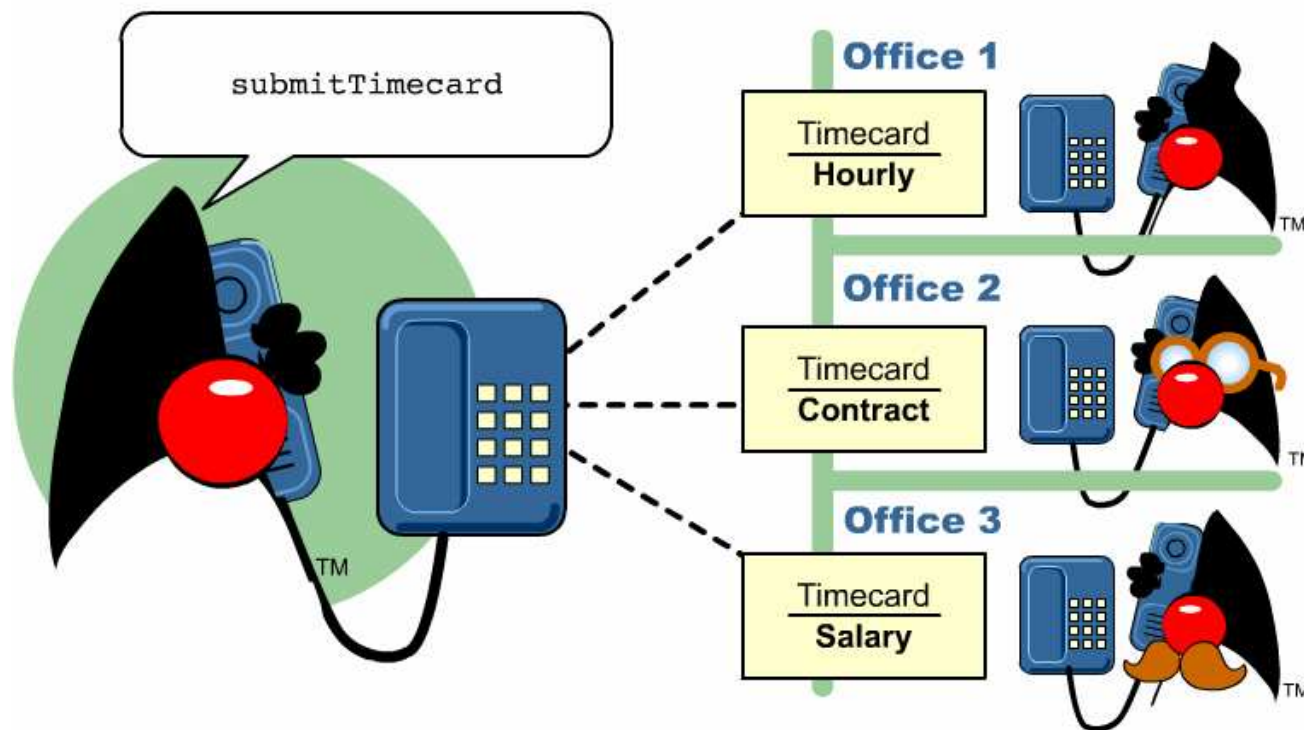
During the next pay period, the employees have switched offices.

Polymorphism 6



The manager can still place the same call to all the offices, knowing that the employees will follow their own `submitTimecard` method.

Polymorphism 7



Polymorphism is the ability for objects from separate, yet related classes to receive the same message, but act on it in their own way.

Exercise: OOP Concepts

- 1) Why Object-oriented programming?
- 2) Explain the three main concepts of Object-orientation.
- 3) How does Object-orientation enables data security?
- 4) Explain how to achieve Polymorphism through Inheritance.

Objects

Course Outline

- | | | |
|---|---|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|---|---|

Objects

Everything in Java is an object.

Well ... almost.

Object lifecycle:

- 1) creation
- 2) usage
- 3) destruction

Object Creation

A variable `s` is declared to refer to the objects of type/class `String`:

```
String s;
```

The value of `s` is `null`; it does not yet refer to any object.

A new `String` object is created in memory with initial `"abc"` value:

```
String s = new String("abc");
```

Now `s` contains the address of this new object.

Object Usage

Objects are used mostly through variables.

Four usage scenarios:

- 1) one variable, one object
- 2) two variables, one object
- 3) two variables, two objects
- 4) one variable, two objects

One Variable, One Object

One variable, one object:

```
String s = new String("abc");
```

What can you do with the object addressed by `s`?

- 1) Check the length: `s.length() == 3`
- 2) Return the substring: `s.substring(2)`
- 3) etc.

Depending what is allowed by the definition of `String`.

Two Variables, One Object

Two variables, one object:

```
String s1 = new String("abc");  
String s2;
```

Assignment copies the address, not value:

```
s2 = s1;
```

Now `s1` and `s2` both refer to one object. After

```
s1 = null;
```

`s2` still points to this object.

Two Variables, Two Objects

Two variables, two objects:

```
String s1 = new String("abc");  
String s2 = new String("abc");
```

`s1` and `s2` objects have initially the same values:

```
s1.equals(s2) == true
```

But they are not the same objects:

```
(s1 == s2) == false
```

They can be changed independently of each other.

One Variable, Two Objects

One variable, two objects:

```
String s = new String("abc");  
s = new String("cba");
```

The "abc" object is no longer accessible through any variable.

Object Destruction

A program accumulates memory through its execution.

Two mechanism to free memory that is no longer need by the program:

- 1) manual – done in C/C++
- 2) automatic – done in Java

In Java, when an object is no longer accessible through any variable, it is eventually removed from the memory by the garbage collector.

Garbage collector is parts of the Java Run-Time Environment.

Exercise: Objects

- 1) Explain in detail the lifecycle of an object, making reference to it's
 - a) creation
 - b) usage and
 - c) destruction
- 2) What is garbage collection?

Classes

Course Outline

- | | | |
|---|---|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|---|---|

Class Outline

- 1) class definition
 - a) attributes
 - b) methods
 - c) constructors
- 2) method overloading
- 3) method communication
 - a) passing arguments
 - b) returning results
- 4) recursive methods
- 5) arrays as object
- 6) static components
- 7) nested and internal classes

Class

A basis for the Java language.

Each concept we wish to describe in Java must be included inside a class.

A class defines a new data type, whose values are objects:

- 1) a class is a template for objects
- 2) an object is an instance of a class

Class Definition

A class contains a **name**, several **variable** declarations (instance variables) and several **method** declarations. All are called **members** of the class.

General form of a class:

```
class classname {  
    type instance-variable-1;  
    ...  
    type instance-variable-n;  
  
    type method-name-1 (parameter-list) { ... }  
    type method-name-2 (parameter-list) { ... }  
    ...  
    type method-name-m (parameter-list) { ... }  
}
```

Example: Class

A class with three variable members:

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

A new `Box` object is created and a new value assigned to its width variable:

```
Box myBox = new Box();  
myBox.width = 100;
```

Example: Class Usage

```
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox = new Box();  
        double vol;  
  
        mybox.width = 10;  
        mybox.height = 20;  
        mybox.depth = 15;  
  
        vol = mybox.width * mybox.height * mybox.depth;  
        System.out.println("Volume is " + vol);  
    }  
}
```


Compilation and Execution

Place the `Box` class definitions in file `Box.java`:

```
class Box { ... }
```

Place the `BoxDemo` class definitions in file `BoxDemo.java`:

```
class BoxDemo {  
    public static void main(...) { ... }  
}
```

Compilation and execution:

```
> javac BoxDemo.java  
> java BoxDemo
```

Variable Independence 1

Each object has its own copy of the instance variables: changing the variables of one object has no effect on the variables of another object.

Consider this example:

```
class BoxDemo2 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
  
        mybox1.width = 10;  
        mybox1.height = 20;  
        mybox1.depth = 15;
```

Variable Independence 2

```
mybox2.width = 3;  
mybox2.height = 6;  
mybox2.depth = 9;
```

```
vol = mybox1.width * mybox1.height * mybox1.depth;  
System.out.println("Volume is " + vol);
```

```
vol = mybox2.width * mybox2.height * mybox2.depth;  
System.out.println("Volume is " + vol);
```

```
}
```

```
}
```

What are the printed volumes of both boxes?

Declaring Objects

Obtaining objects of a class is a two-stage process:

- 1) Declare a variable of the class type:

```
Box myBox;
```

The value of `myBox` is a reference to an object, if one exists, or `null`.

At this moment, the value of `myBox` is `null`.

- 2) Acquire an actual, physical copy of an object and assign its address to the variable. How to do this?

Operator new

Allocates memory for a `Box` object and returns its address:

```
Box myBox = new Box();
```

The address is then stored in the `myBox` reference variable.

`Box()` is a class constructor - a class may declare its own constructor or rely on the default constructor provided by the Java environment.

Memory Allocation

Memory is allocated for objects dynamically.

This has both advantages and disadvantages:

- 1) as many objects are created as needed
- 2) allocation is uncertain – memory may be insufficient

Variables of simple types do not require `new`:

```
int n = 1;
```

In the interest of efficiency, Java does not implement simple types as objects. Variables of simple types hold values, not references.

Assigning Reference Variables

Assignment copies address, not the actual value:

```
Box b1 = new Box();  
Box b2 = b1;
```

Both variables point to the same object.

Variables are not in any way connected. After

```
b1 = null;
```

`b2` still refers to the original object.

Methods

General form of a method definition:

```
type name(parameter-list) {  
    ... return value; ...  
}
```

Components:

- 1) `type` - type of values returned by the method. If a method does not return any value, its return type must be `void`.
- 2) `name` is the name of the method
- 3) `parameter-list` is a sequence of type-identifier lists separated by commas
- 4) `return value` indicates what value is returned by the method.

Example: Method 1

Classes declare methods to hide their internal data structures, as well as for their own internal use:

Within a class, we can refer directly to its member variables:

```
class Box {  
    double width, height, depth;  
    void volume() {  
        System.out.print("Volume is ");  
        System.out.println(width * height * depth);  
    }  
}
```

Example: Method 2

When an instance variable is accessed by code that is not part of the class in which that variable is defined, access must be done through an object:

```
class BoxDemo3 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        mybox1.width = 10;    mybox2.width = 3;  
        mybox1.height = 20;   mybox2.height = 6;  
        mybox1.depth = 15;    mybox2.depth = 9;  
  
        mybox1.volume();  
        mybox2.volume();  
    }  
}
```

Value-Returning Method 1

The type of an expression returning value from a method must agree with the return type of this method:

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    double volume() {  
        return width * height * depth;  
    }  
}
```

Value-Returning Method 2

```
class BoxDemo4 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
        mybox1.width = 10;  
        mybox2.width = 3;  
        mybox1.height = 20;  
        mybox2.height = 6;  
        mybox1.depth = 15;  
        mybox2.depth = 9;  
    }  
}
```

Value-Returning Method 3

The type of a variable assigned the value returned by a method must agree with the return type of this method:

```
    vol = mybox1.volume();  
    System.out.println("Volume is " + vol);  
    vol = mybox2.volume();  
    System.out.println("Volume is " + vol);  
}  
}
```

Parameterized Method

Parameters increase generality and applicability of a method:

- 1) method without parameters

```
int square() { return 10*10; }
```

- 2) method with parameters

```
int square(int i) { return i*i; }
```

Parameter: a variable receiving value at the time the method is invoked.

Argument: a value passed to the method when it is invoked.

Example: Parameterized Method 1

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    double volume() {  
        return width * height * depth;  
    }  
  
    void setDim(double w, double h, double d) {  
        width = w; height = h; depth = d;  
    }  
}
```

Example: Parameterized Method 2

```
class BoxDemo5 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
  
        mybox1.setDim(10, 20, 15);  
        mybox2.setDim(3, 6, 9);  
  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```


Constructor

A constructor initializes the instance variables of an object.

It is called immediately after the object is created but before the `new` operator completes.

- 1) it is syntactically similar to a method:
- 2) it has the same name as the name of its class
- 3) it is written without return type; the default return type of a class constructor is the same class

When the class has no constructor, the default constructor automatically initializes all its instance variables with zero.

Example: Constructor 1

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    Box() {  
        System.out.println("Constructing Box");  
        width = 10; height = 10; depth = 10;  
    }  
  
    double volume() {  
        return width * height * depth;  
    }  
}
```

Example: Constructor 2

```
class BoxDemo6 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```

Parameterized Constructor 1

So far, all boxes have the same dimensions.

We need a constructor able to create boxes with different dimensions:

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    Box(double w, double h, double d) {  
        width = w; height = h; depth = d;  
    }  
  
    double volume() { return width * height * depth; }  
}
```

Parameterized Constructor 2

```
class BoxDemo7 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box(10, 20, 15);  
        Box mybox2 = new Box(3, 6, 9);  
        double vol;  
  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```

finalize() Method

A constructor helps to initialize an object just after it has been created.

In contrast, the `finalize` method is invoked just before the object is destroyed:

- 1) implemented inside a class as:

```
protected void finalize() { ... }
```

- 2) implemented when the usual way of removing objects from memory is insufficient, and some special actions has to be carried out

How is the `finalize` method invoked?

Garbage Collection

Garbage collection is a mechanism to remove objects from memory when they are no longer needed.

Garbage collection is carried out by the garbage collector:

- 1) The garbage collector keeps track of how many references an object has.
- 2) It removes an object from memory when it has no longer any references.
- 3) Thereafter, the memory occupied by the object can be allocated again.
- 4) The garbage collector invokes the `finalize` method.

Keyword this

Keyword `this` allows a method to refer to the object that invoked it.

It can be used inside any method to refer to the current object:

```
Box(double width, double height, double depth) {  
    this.width = width;  
    this.height = height;  
    this.depth = depth;  
}
```

The above use of `this` is redundant but correct.

When is `this` really needed?

Instance Variable Hiding

Variables with the same names:

- 1) it is illegal to declare two local variables with the same name inside the same or enclosing scopes
- 2) it is legal to declare local variables or parameters with the same name as the instance variables of the class.

As the same-named local variables/parameters will hide the instance variables, using `this` is necessary to regain access to them:

```
Box(double width, double height, double depth) {  
    this.width = width;  
    this.height = height;  
    this.depth = depth;  
}
```

Example: Stack 1

A stack hold data in the first-in-last-out order.

Here is the implementation of a 10-element stack:

```
class Stack {  
    int stck[] = new int[10];  
    int tos;  
  
    Stack() {  
        tos = -1;  
    }  
}
```

Example: Stack 2

```
void push(int item) {
    if (tos==9) System.out.println("Stack is full.");
    else stck[++tos] = item;
}

int pop() {
    if (tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
    }
    else return stck[tos--];
}
}
```

Example: Stack 3

```
class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();
        for (int i=0; i<10; i++) mystack1.push(i);
        for (int i=10; i<20; i++) mystack2.push(i);

        System.out.println("Stack in mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());
    }
}
```

Method Overloading

It is legal for a class to have two or more methods with the same name.

However, Java has to be able to uniquely associate the invocation of a method with its definition relying on the number and types of arguments.

Therefore the same-named methods must be distinguished:

- 1) by the number of arguments, or
- 2) by the types of arguments

Overloading and inheritance are two ways to implement polymorphism.

Example: Overloading 1

```
class OverloadDemo {  
    void test() {  
        System.out.println("No parameters");  
    }  
    void test(int a) {  
        System.out.println("a: " + a);  
    }  
    void test(int a, int b) {  
        System.out.println("a and b: " + a + " " + b);  
    }  
    double test(double a) {  
        System.out.println("double a: " + a); return a*a;  
    }  
}
```

Example: Overloading 2

```
class Overload {  
    public static void main(String args[]) {  
        OverloadDemo ob = new OverloadDemo();  
        double result;  
        ob.test();  
        ob.test(10);  
        ob.test(10, 20);  
        result = ob.test(123.2);  
        System.out.println("ob.test(123.2): " + result);  
    }  
}
```

Different Result Types

Different result types are insufficient.

The following will not compile:

```
double test(double a) {  
    System.out.println("double a: " + a);  
    return a*a;  
}
```

```
int test(double a) {  
    System.out.println("double a: " + a);  
    return (int) a*a;  
}
```


Overloading and Conversion 1

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.

When no exact match can be found, Java's automatic type conversion can aid overload resolution:

```
class OverloadDemo {  
    void test() {  
        System.out.println("No parameters");  
    }  
    void test(int a, int b) {  
        System.out.println("a and b: " + a + " " + b);  
    }  
}
```

Overloading and Conversion 2

```
void test(double a) {  
    System.out.println("Inside test(double) a: " + a);  
}  
}
```

```
class Overload {  
    public static void main(String args[]) {  
        OverloadDemo ob = new OverloadDemo();  
        int i = 88;  
        ob.test();  
        ob.test(10, 20);  
        ob.test(i);  
        ob.test(123.2);  
    }  
}
```

Overloading and Polymorphism

In the languages without overloading, methods must have a unique names:

```
int abs(int i)
long labs(int i)
float fabs(int i)
```

Java enables logically-related methods to occur under the same name:

```
static int abs(int i)
static long abs(long i)
static float abs(float i)
```

Constructor Overloading

Why overload constructors? Consider this:

```
class Box {  
    double width, height, depth;  
  
    Box(double w, double h, double d) {  
        width = w; height = h; depth = d;  
    }  
    double volume() {  
        return width * height * depth;  
    }  
}
```

All `Box` objects can be created in one way: passing all three dimensions.

Example: Overloading 1

Three constructors: 3-parameter, 1-parameter, parameter-less.

```
class Box {  
    double width, height, depth;  
    Box(double w, double h, double d) {  
        width = w; height = h; depth = d;  
    }  
    Box() {  
        width = -1; height = -1; depth = -1;  
    }  
    Box(double len) {  
        width = height = depth = len;  
    }  
    double volume() { return width * height * depth; }  
}
```

Example: Overloading 2

```
class OverloadCons {
    public static void main(String args[]) {
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        double vol;

        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        vol = mycube.volume();
        System.out.println("Volume of mycube is " + vol);
    }
}
```

Object Argument 1

So far, all method received arguments of simple types.

They may also receive an object as an argument. Here is a method to check if a parameter object is equal to the invoking object:

```
class Test {  
    int a, b;  
    Test(int i, int j) {  
        a = i; b = j;  
    }  
    boolean equals(Test o) {  
        if (o.a == a && o.b == b) return true;  
        else return false;  
    }  
}
```

Object Argument 2

```
class PassOb {  
    public static void main(String args[]) {  
        Test ob1 = new Test(100, 22);  
        Test ob2 = new Test(100, 22);  
        Test ob3 = new Test(-1, -1);  
        System.out.println("ob1==ob2: " + ob1.equals(ob2));  
        System.out.println("ob1==ob3: " + ob1.equals(ob3));  
    }  
}
```


Passing Object to Constructor 1

A special case of object-passing is passing an object to the constructor.

This is to initialize one object with another object:

```
class Box {  
    double width, height, depth;  
  
    Box(Box ob) {  
        width = ob.width;  
        height = ob.height;  
        depth = ob.depth;  
    }  
}
```

Passing Object to Constructor 2

```
Box(double w, double h, double d) {  
    width = w;  
    height = h;  
    depth = d;  
}
```

```
double volume() {  
    return width * height * depth;  
}  
}
```

Passing Object to Constructor 3

```
class OverloadCons2 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box(10, 20, 15);  
        Box mybox2 = new Box(mybox1);  
        double vol;  
  
        vol = mybox1.volume();  
        System.out.println("Volume of mybox1 is " + vol);  
  
        vol = mybox2.volume();  
        System.out.println("Volume of mybox2 is " + vol);  
    }  
}
```

Argument-Passing

Two types of variables:

- 1) simple types
- 2) class types

Two corresponding ways of how the arguments are passed to methods:

- 1) **by value** – a method receives a copy of the original value; parameters of simple types
- 2) **by reference** – a method receives the memory address of the original value, not the value itself; parameters of class types

Simple Type Argument-Passing 1

Passing arguments of simple types takes place by value:

```
class Test {  
    void meth(int i, int j) {  
        i *= 2;  
        j /= 2;  
    }  
}
```

Simple Type Argument-Passing 2

With by-value argument-passing what occurs to the parameter that receives the argument has no effect outside the method:

```
class CallByValue {  
    public static void main(String args[]) {  
        Test ob = new Test();  
        int a = 15, b = 20;  
        System.out.print("a and b before call: ");  
        System.out.println(a + " " + b);  
        ob.meth(a, b);  
        System.out.print("a and b after call: ");  
        System.out.println(a + " " + b);  
    }  
}
```

Class Type Argument-Passing 1

Objects are passed to the methods by reference: a parameter obtains the same address as the corresponding argument:

```
class Test {  
    int a, b;  
  
    Test(int i, int j) {  
        a = i; b = j;  
    }  
  
    void meth(Test o) {  
        o.a *= 2; o.b /= 2;  
    }  
}
```

Class Type Argument-Passing 2

As the parameter hold the same address as the argument, changes to the object inside the method do affect the object used by the argument:

```
class CallByRef {  
    public static void main(String args[]) {  
        Test ob = new Test(15, 20);  
        System.out.print("ob.a and ob.b before call: ");  
        System.out.println(ob.a + " " + ob.b);  
        ob.meth(ob);  
        System.out.print("ob.a and ob.b after call: ");  
        System.out.println(ob.a + " " + ob.b);  
    }  
}
```


Returning Objects 1

So far, all methods returned no values or values of simple types.

Methods may also return objects:

```
class Test {  
    int a;  
    Test(int i) {  
        a = i;  
    }  
    Test incrByTen() {  
        Test temp = new Test(a+10);  
        return temp;  
    }  
}
```

Returning Objects 2

Each time a method `incrByTen` is invoked a new object is created and a reference to it is returned:

```
class RetOb {  
    public static void main(String args[]) {  
        Test ob1 = new Test(2);  
        Test ob2;  
        ob2 = ob1.incrByTen();  
        System.out.println("ob1.a: " + ob1.a);  
        System.out.println("ob2.a: " + ob2.a);  
        ob2 = ob2.incrByTen();  
        System.out.print("ob2.a after second increase: ");  
        System.out.println(ob2.a);  
    }  
}
```

Recursion

A recursive method is a method that calls itself:

What happens then?

- 1) all method parameters and local variables are allocated on the stack
- 2) arguments are prepared in the corresponding parameter positions
- 3) the method code is executed for the new arguments
- 4) upon return, all parameters and variables are removed from the stack
- 5) the execution continues immediately after the invocation point

Example: Recursion

```
class Factorial {  
    int fact(int n) {  
        if (n==1) return 1;  
        return fact(n-1) * n;  
    }  
}
```

```
class Recursion {  
    public static void main(String args[]) {  
        Factorial f = new Factorial();  
        System.out.print("Factorial of 5 is ");  
        System.out.println(f.fact(5));  
    }  
}
```

Example: Recursion and Arrays 1

A method that recursively prints out a given number of elements in a table:

```
class RecTest {
    int values[];

    RecTest(int i) {
        values = new int[i];
    }

    void printArray(int i) {
        if (i==0) return;
        else printArray(i-1);
        System.out.print "[" + (i-1) + " ] ";
        System.out.println(values[i-1]);
    }
}
```

Example: Recursion and Arrays 2

```
class Recursion2 {  
    public static void main(String args[]) {  
        RecTest ob = new RecTest(10);  
        int i;  
        for(i=0; i<10; i++)  
            ob.values[i] = i;  
        ob.printArray(10);  
    }  
}
```

Arrays Revisited

All arrays are implemented as objects.

In particular, all arrays have the `length` attribute to return the number of elements that an array may hold:

```
class Length {  
    public static void main(String args[]) {  
        int a1[] = new int[10];  
        int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};  
        int a3[] = {4, 3, 2, 1};  
        System.out.println("length of a1 is " + a1.length);  
        System.out.println("length of a2 is " + a2.length);  
        System.out.println("length of a3 is " + a3.length);  
    }  
}
```

Example: Arrays as Objects 1

An improved `Stack` example, able to handle stacks of any size:

```
class Stack {
    private int stck[];
    private int tos;

    Stack(int size) {
        stck = new int[size]; tos = -1;
    }

    void push(int item) {
        if (tos==stck.length-1)
            System.out.println("Stack is full.");
        else stck[++tos] = item;
    }
}
```


Example: Arrays as Objects 2

```
int pop() {  
    if (tos < 0) {  
        System.out.println("Stack underflow.");  
        return 0;  
    }  
    else  
        return stck[tos--];  
}
```

Example: Arrays as Objects 3

```
class TestStack2 {
    public static void main(String args[]) {
        Stack mystack1 = new Stack(5);
        Stack mystack2 = new Stack(8);
        for (int i=0; i<5; i++) mystack1.push(i);
        for (int i=0; i<8; i++) mystack2.push(i);

        System.out.println("Stack in mystack1:");
        for (int i=0; i<5; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for (int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}
```

Static Class Members

Normally, the members of a class (its variables and methods) may be only used through the objects of this class.

Static members are independent of the objects:

- 1) variables
- 2) methods
- 3) initialization block

All declared with the `static` keyword.

Static Variables

Static variable:

```
static int a;
```

Essentially, it a global variable shared by all instances of the class.

It cannot be used within a non-static method.

Static Methods

Static method:

```
static void meth() { ... }
```

Several restrictions apply:

- can only call static methods
- must only access static variables
- cannot refer to `this` or `super` in any way

Static Block

Static block:

```
static { ... }
```

This is where the static variables are initialized.

The block is executed exactly once, when the class is first loaded.

Example: Static 1

```
class UseStatic {
    static int a = 3;
    static int b;
    static void meth(int x) {
        System.out.print("x = " + x + " a = " + a);
        System.out.println(" b = " + b);
    }
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }
    public static void main(String args[]) {
        meth(42);
    }
}
```

Static Member Usage 1

How to use static members outside their class?

Consider this class:

```
class StaticDemo {  
    static int a = 42;  
    static int b = 99;  
    static void callme() {  
        System.out.println("a = " + a);  
    }  
}
```


Static Member Usage 2

Static variables/method are used through the class name:

```
StaticDemo.a  
StaticDemo.callme()
```

Example:

```
class StaticByName {  
    public static void main(String args[]) {  
        StaticDemo.callme();  
        System.out.println("b = " + StaticDemo.b);  
    }  
}
```

Nested Classes

It is possible to define a class within a class – nested class.

The scope of the nested class is its enclosing class: if class `B` is defined within class `A` then `B` is known to `A` but not outside.

Access rights:

- 1) a nested class has access to all members of its enclosing class, including its private members
- 2) the enclosing class does not have access to the members of the nested class

Types of Nested Classes

There are two types of nested classes:

- 1) **static** – cannot access the members of its enclosing class directly, but through an object; defined with the `static` keyword
- 2) non-static – has direct access to all members of the enclosing class in the same way as other non-static member of this class so

A static nested class is seldom used.

A non-static nested class is also called an **inner class**.

Example: Inner Classes 1

`Outer` has a variable `out_x`, an inner class `Inner` and a method `test` which creates an object of the `Inner` class and calls its `display` method:

```
class Outer {
    int out_x = 100;
    void test() {
        Inner inner = new Inner();
        inner.display();
    }
    class Inner {
        void display() {
            System.out.println("out_x = " + out_x);
        }
    }
}
```

Example: Inner Classes 2

A demonstration class to create an object of the `Outer` class and invoke the `test` method on this object:

```
class InnerClassDemo {  
    public static void main(String args[]) {  
        Outer outer = new Outer();  
        outer.test();  
    }  
}
```

The `Inner` class is only known within the `Outer` class. Any reference to `Inner` outside `Outer` will create a compile-time error.

Inner Members Visibility 1

Inner class has access to all member of the outer class.

The reverse is not true: members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

This is the `Outer` class with a variable, two methods and `Inner` class.

The first method refers to the `Inner` class correctly through an object:

```
class Outer {  
    int outer_x = 100;  
    void test() {  
        Inner inner = new Inner();  
        inner.display();  
    }  
}
```

Inner Members Visibility 2

Inner class declares variable `y` and refers to the `Outer` class variable:

```
class Inner {  
    int y = 10;  
    void display() {  
        System.out.println("outer_x = " + outer_x);  
    }  
}
```

`showy` method refers incorrectly to the `Inner` class's `y` variable:

```
void showy() {  
    System.out.println(y);  
}  
}
```

Inner Members Visibility 3

As a result, this program will not compile:

```
class InnerClassDemo {  
    public static void main(String args[]) {  
        Outer outer = new Outer();  
        outer.test();  
    }  
}
```


Inner Class Declaration

So far, all inner classes were defined within the outer class scope.

In fact, an inner class may be defined within any block scope.

The following is an example of an inner class define within a for loop.

Example: Inner Class Declaration 1

```
class Outer {  
    int outer_x = 100;  
    void test() {  
        for (int i=0; i<10; i++) {  
            class Inner {  
                void display() {  
                    System.out.println("outer_x= " + outer_x);  
                }  
            }  
            Inner inner = new Inner();  
            inner.display();  
        }  
    }  
}
```

Example: Inner Class Declaration 2

A demonstration creates an `Outer` object and invokes a `test` method on it:

```
class InnerClassDemo {  
    public static void main(String args[]) {  
        Outer outer = new Outer();  
        outer.test();  
    }  
}
```

Exercise: Classes 1

- 1) What's the difference between an object and a class?
- 2) Explain why constructors don't have return types.
- 3) Can *this* keyword be used within the context of a static method?. Why?.
- 4) What's Overloading?
- 5) Explain the mechanism of Argument passing in Java. Is it by value or by reference?
- 6) Explain the mechanism that Java uses to remove objects from memory when they are no longer needed.
- 7) An object has a handle to some non-Java resources such as file or window character font. How would you ensure that these resources are freed before the object is destroyed?

Exercise: Classes 2

- 1) Create a class with a default constructor (no arguments) that prints a message. Create an object of this class.
- 2) Add an overloaded constructor to the class in 1 which takes a String argument and prints it along with your message.
- 3) Make a two dimensional array of objects of the class you created in 2.
- 4) Create a class Counter with an adequate data member. Write four methods, one to get the value of the counter, one to increment the counter, one to decrement the counter, and one to reset the counter to zero. Make sure that the value of the counter never gets less than zero.
- 5) Create an object of your class Counter and assign it to two different variables. Change the state of the object, calling your method on one variable. See what happens if you call the get-value-method on the other variable.

Inheritance

Course Outline

- | | | |
|---|--|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|--|---|

Inheritance

One of the pillars of object-orientation.

A new class is derived from an existing class:

- 1) existing class is called **super-class**
- 2) derived class is called **sub-class**

A sub-class is a specialized version of its super-class:

- 1) has all non-private members of its super-class
- 2) may provide its own implementation of super-class methods

Objects of a sub-class are a special kind of objects of a super-class.

Inheritance Syntax

Syntax:

```
class sub-class extends super-class {  
    ...  
}
```

Each class has at most one super-class; no multi-inheritance in Java.

No class is a sub-class of itself.

Example: Super-Class

```
class A {  
    int i;  
  
    void showi() {  
        System.out.println("i: " + i);  
    }  
  
}
```

Example: Sub-Class

```
class B extends A {  
  
    int j;  
  
    void showj() {  
        System.out.println("j: " + j);  
    }  
  
    void sum() {  
        System.out.println("i+j: " + (i+j));  
    }  
  
}
```

Example: Testing Class

```
class SimpleInheritance {  
    public static void main(String args[]) {  
        A a = new A();  
        B b = new B();  
        a.i = 10;  
        System.out.println("Contents of a: ");  
        a.showi();  
        b.i = 7; b.j = 8;  
        System.out.println("Contents of b: ");  
        subOb.showi(); subOb.showj();  
        System.out.println("Sum of I and j in b:");  
        b.sum();  
    }  
}
```

Inheritance and Private Members 1

A class may declare some of its members **private**.

A sub-class has no access to the private members of its super-class:

```
class A {  
    int i;  
    private int j;  
    void setij(int x, int y) {  
        i = x; j = y;  
    }  
}
```

Inheritance and Private Members 2

Class B has no access to the A's private variable j.

This program will not compile:

```
class B extends A {  
    int total;  
    void sum() {  
        total = i + j;  
    }  
}
```

Example: Box Class

The basic `Box` class with width, height and depth:

```
class Box {  
    double width, height, depth;  
    Box(double w, double h, double d) {  
        width = w; height = h; depth = d;  
    }  
    Box(Box b) {  
        width = b.width;  
        height = b.height; depth = b.depth;  
    }  
    double volume() {  
        return width * height * depth;  
    }  
}
```

Example: BoxWeight Sub-Class

BoxWeight class extends Box with the new weight variable:

```
class BoxWeight extends Box {  
    double weight;  
    BoxWeight(double w, double h, double d, double m) {  
        width = w; height = h; depth = d; weight = m;  
    }  
    BoxWeight(Box b, double w) {  
        Box(b); weight = w;  
    }  
}
```

Box is a super-class, BoxWeight is a sub-class.

Example: BoxWeight Demo

```
class DemoBoxWeight {  
    public static void main(String args[]) {  
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);  
        BoxWeight mybox2 = new BoxWeight(mybox1);  
        double vol;  
        vol = mybox1.volume();  
        System.out.println("Volume of mybox1 is " + vol);  
        System.out.print("Weight of mybox1 is ");  
        System.out.println(mybox1.weight);  
        vol = mybox2.volume();  
        System.out.println("Volume of mybox2 is " + vol);  
        System.out.print("Weight of mybox2 is ");  
        System.out.println(mybox2.weight);  
    }  
}
```

Another Sub-Class

Once a super-class exists that defines the attributes common to a set of objects, it can be used to create any number of more specific sub-classes.

The following sub-class of `Box` adds the `color` attribute instead of `weight`:

```
class ColorBox extends Box {  
    int color;  
  
    ColorBox(double w, double h, double d, int c) {  
        width = w; height = h; depth = d;  
        color = c;  
    }  
}
```

Referencing Sub-Class Objects

A variable of a super-class type may refer to any of its sub-class objects:

```
class SuperClass { ... }  
class SubClass extends SuperClass { ... }
```

```
SuperClass o1;  
SubClass o2 = new SubClass();
```

```
o1 = o2;
```

However, the inverse is illegal:

```
o2 = o1
```

Example: Sub-Class Objects 1

```
class RefDemo {  
    public static void main(String args[]) {  
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);  
        Box plainbox = new Box(5, 5, 5);  
        double vol;  
        vol = weightbox.volume();  
        System.out.print("Volume of weightbox is ");  
        System.out.println(vol);  
        System.out.print("Weight of weightbox is ");  
        System.out.println(weightbox.weight);  
        plainbox = weightbox;  
        vol = plainbox.volume();  
        System.out.println("Volume of plainbox is " + vol);  
    }  
}
```

Super-Class Variable Access

`plainbox` variable now refers to the `WeightBox` object.

Can we then access this object's `weight` variable through `plainbox`?

No. The type of a variable, not the object this variable refers to, determines which members we can access!

This is illegal:

```
System.out.print("Weight of plainbox is ");  
System.out.println(plainbox.weight);
```

Super as a Constructor

Calling a constructor of a super-class from the constructor of a sub-class:

```
super (parameter-list) ;
```

Must occur as the very first instructor in the sub-class constructor:

```
class SuperClass { ... }  
  
class SubClass extends SuperClass {  
    SubClass (...) {  
        super (...);  
        ...  
    }  
    ...  
}
```

Example: Super Constructor 1

`BoxWeight` need not initialize the variable for the `Box` super-class, only the added `weight` variable:

```
class BoxWeight extends Box {  
    double weight;  
  
    BoxWeight(double w, double h, double d, double m) {  
        super(w, h, d); weight = m;  
    }  
  
    BoxWeight(Box b, double w) {  
        super(b); weight = w;  
    }  
}
```

Example: Super Constructor 2

```
class DemoSuper {
    public static void main(String args[]) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(mybox1, 10.5);
        double vol;
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.print("Weight of mybox1 is ");
        System.out.println(mybox1.weight);
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.print("Weight of mybox2 is ");
        System.out.println(mybox2.weight);
    }
}
```


Referencing Sub-Class Objects

Sending a sub-class object:

```
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);  
BoxWeight mybox2 = new BoxWeight(mybox1, 10.5);
```

to the constructor expecting a super-class object:

```
BoxWeight(Box b, double w) {  
    super(b); weight = w;  
}
```

Uses of Super

Two uses of `super`:

- 1) to invoke the super-class constructor

```
super();
```

- 2) to access super-class members

```
super.variable;  
super.method(...);
```

(1) was discussed, consider (2).

Super and Hiding

Why is `super` needed to access super-class members?

When a sub-class declares the variables or methods with the same names and types as its super-class:

```
class A {  
    int i = 1;  
}
```

```
class B extends A {  
    int i = 2;  
    System.out.println("i is " + i);  
}
```

The re-declared variables/methods hide those of the super-class.

Example: Super and Hiding 1

```
class A {  
    int i;  
}  
  
class B extends A {  
    int i;  
  
    B(int a, int b) {  
        super.i = a; i = b;  
    }  
    void show() {  
        System.out.println("i in superclass: " + super.i);  
        System.out.println("i in subclass: " + i);  
    }  
}
```

Example: Super and Hiding 2

Although the `i` variable in `B` hides the `i` variable in `A`, `super` allows access to the hidden variable of the super-class:

```
class UseSuper {  
    public static void main(String args[]) {  
        B subOb = new B(1, 2);  
        subOb.show();  
    }  
}
```

Multi-Level Class Hierarchy 1

The basic `Box` class:

```
class Box {  
    private double width, height, depth;  
    Box(double w, double h, double d) {  
        width = w; height = h; depth = d;  
    }  
    Box(Box ob) {  
        width = ob.width;  
        height = ob.height; depth = ob.depth;  
    }  
    double volume() {  
        return width * height * depth;  
    }  
}
```

Multi-Level Class Hierarchy 2

Adding the `weight` variable to the `Box` class:

```
class BoxWeight extends Box {  
    double weight;  
  
    BoxWeight(BoxWeight ob) {  
        super(ob); weight = ob.weight;  
    }  
  
    BoxWeight(double w, double h, double d, double m) {  
        super(w, h, d); weight = m;  
    }  
}
```

Multi-Level Class Hierarchy 3

Adding the `cost` variable to the `BoxWeight` class:

```
class Ship extends BoxWeight {  
    double cost;  
  
    Ship(Ship ob) {  
        super(ob); cost = ob.cost;  
    }  
  
    Ship(double w, double h,  
        double d, double m, double c) {  
        super(w, h, d, m); cost = c;  
    }  
}
```


Multi-Level Class Hierarchy 4

```
class DemoShip {  
    public static void main(String args[]) {  
        Ship ship1 = new Ship(10, 20, 15, 10, 3.41);  
        Ship ship2 = new Ship(2, 3, 4, 0.76, 1.28);  
        double vol;  
  
        vol = ship1.volume();  
        System.out.println("Volume of ship1 is " + vol);  
        System.out.print("Weight of ship1 is");  
        System.out.println(ship1.weight);  
        System.out.print("Shipping cost: $");  
        System.out.println(ship1.cost);  
    }  
}
```

Multi-Level Class Hierarchy 5

```
        vol = ship2.volume();  
        System.out.println("Volume of ship2 is " + vol);  
        System.out.print("Weight of ship2 is ");  
        System.out.println(ship2.weight);  
        System.out.print("Shipping cost: $");  
        System.out.println(ship2.cost);  
    }  
}
```

Constructor Call-Order

Constructor call-order:

- 1) first call super-class constructors
- 2) then call sub-class constructors

In the sub-class constructor, if `super (...)` is not used explicitly, Java calls the default, parameter-less super-class constructor.

Example: Constructor Call-Order 1

A is the super-class:

```
class A {  
    A() {  
        System.out.println("Inside A's constructor.");  
    }  
}
```

B and C are sub-classes of A:

```
class B extends A {  
    B() {  
        System.out.println("Inside B's constructor.");  
    }  
}
```

Example: Constructor Call-Order 2

```
class C extends B {  
    C() {  
        System.out.println("Inside C's constructor.");  
    }  
}
```

CallingCons **creates** a single object of the class C:

```
class CallingCons {  
    public static void main(String args[]) {  
        C c = new C();  
    }  
}
```

Exercise: Inheritance 1

- 1) Define a Class `Building` for building objects. Each building has a door as one of its components.
 - a) In the class `Door`, model the fact that a door has a color and three states, "open", "closed", "locked" and "unlocked". To avoid illegal state changes, make the state private, write a method (`getState`) that inspects the state and four methods (open, close, lock and unlock) that change the state. Initialize the state to "closed" in the constructor. Look for an alternative place for this initialization.
 - b) Write a method `enter` that visualizes the process of entering the building (unlock door, open door, enter, ...) by printing adequate messages, e.g. to show the state of the door.
 - c) Write a corresponding method `quit` that visualizes the process of leaving the house. Don't forget to close and lock the door.
 - d) Test your class by defining an object of type `Building` and visualizing the state changes when entering and leaving the building.

Exercise: Inheritance 2

- 3) Extend question 1 by introducing a subclass `HighBuilding` that contains an elevator and the height of the building in addition to the components of `Building`. Override the method `enter` to reflect the use of the elevator. Define a constructor that takes the height of the building as a parameter.
- 4) Define a subclass `Skyscraper` of `HighBuilding`, where the number of floors is stored with each skyscraper.

What happens, if you don't define a constructor for class `Skyscraper` (Try it)?

Write a constructor that takes the number of floors and the height as a parameter. Test the class by creating a skyscraper with 40 floors and using the inherited method `enter`.

Polymorphism

Course Outline

- | | | |
|---|---|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|---|---|

Inheritance and Reuse

Reuse of code: every time a new sub-class is defined, programmers are reusing the code in a super-class.

All non-private members of a super-class are inherited by its sub-class:

- 1) an attribute in a super-class is inherited as-such in a sub-class
- 2) a method in a super-class is inherited in a sub-class:
 - a) as-such, or
 - b) is substituted with the method which has the same name and parameters (overriding) but a different implementation

Polymorphism: Definition

Polymorphism is one of three pillars of object-orientation.

Polymorphism: many different (poly) forms of objects that share a common interface respond differently when a method of that interface is invoked:

- 1) a super-class defines the common interface
- 2) sub-classes have to follow this interface (inheritance), but are also permitted to provide their own implementations (overriding)

A sub-class provides a specialized behaviors relying on the common elements defined by its super-class.

Polymorphism: Behavior

Suppose we have a hierarchy of classes:

- 1) The top class in the hierarchy represents a common interface to all classes below. This class is the base class.
- 2) All classes below the base represent a number of forms of objects, all referred to by a variable of the base class type.

What happens when a method is invoked using the base class reference?
The object responds in accordance with its true type.

What if the user pointed to a different form of object using the same reference? The user would observe different behavior.

This is polymorphism.

Method Overriding

When a method of a sub-class has the same name and type as a method of the super-class, we say that this method is **overridden**.

When an overridden method is called from within the sub-class:

- 1) it will always refer to the sub-class method
- 2) super-class method is hidden

Example: Hiding with Overriding 1

```
class A {  
    int i, j;  
  
    A(int a, int b) {  
        i = a; j = b;  
    }  
  
    void show() {  
        System.out.println("i and j: " + i + " " + j);  
    }  
}
```

Example: Hiding with Overriding 2

```
class B extends A {  
    int k;  
  
    B(int a, int b, int c) {  
        super(a, b);  
        k = c;  
    }  
  
    void show() {  
        System.out.println("k: " + k);  
    }  
}
```

Example: Hiding with Overriding 3

When `show()` is invoked on an object of type `B`, the version of `show()` defined in `B` is used:

```
class Override {  
    public static void main(String args[]) {  
        B subOb = new B(1, 2, 3);  
        subOb.show();  
    }  
}
```

The version of `show()` in `A` is hidden through overriding.

Super and Method Overriding

The hidden super-class method may be invoked using `super`:

```
class B extends A {  
    int k;  
    B(int a, int b, int c) {  
        super(a, b);  
        k = c;  
    }  
    void show() {  
        super.show();  
        System.out.println("k: " + k);  
    }  
}
```

The super-class version of `show()` is called within the sub-class's version.

Overriding versus Overloading 1

Method overriding occurs only when the names and types of the two methods (super-class and sub-class methods) are identical.

If not identical, the two methods are simply overloaded:

```
class A {  
    int i, j;  
  
    A(int a, int b) {  
        i = a; j = b;  
    }  
    void show() {  
        System.out.println("i and j: " + i + " " + j);  
    }  
}
```

Overriding versus Overloading 2

The `show()` method in `B` takes a `String` parameter, while the `show()` method in `A` takes no parameters:

```
class B extends A {  
    int k;  
  
    B(int a, int b, int c) {  
        super(a, b); k = c;  
    }  
  
    void show(String msg) {  
        System.out.println(msg + k);  
    }  
}
```

Overriding versus Overloading 3

The two invocations of `show()` are resolved through the number of arguments (zero versus one):

```
class Override {  
    public static void main(String args[]) {  
        B subOb = new B(1, 2, 3);  
  
        subOb.show("This is k: ");  
        subOb.show();  
    }  
}
```

Dynamic Method Invocation

Overriding is a lot more than the namespace convention.

Overriding is the basis for dynamic method dispatch – a call to an overridden method is resolved at run-time, rather than compile-time.

Method overriding allows for dynamic method invocation:

- 1) an overridden method is called through the super-class variable
- 2) Java determines which version of that method to execute based on the type of the referred object at the time the call occurs
- 3) when different types of objects are referred, different versions of the overridden method will be called.

Example: Dynamic Invocation 1

A super-class A:

```
class A {  
    void callme() {  
        System.out.println("Inside A's callme method");  
    }  
}
```

Example: Dynamic Invocation 2

Two sub-classes B and C:

```
class B extends A {  
    void callme() {  
        System.out.println("Inside B's callme method");  
    }  
}  
class C extends A {  
    void callme() {  
        System.out.println("Inside C's callme method");  
    }  
}
```

B and C override the A's callme() method.

Example: Dynamic Invocation 3

Overridden method is invoked through the variable of the super-class type. Each time, the version of the `callme()` method executed depends on the type of the object being referred to at the time of the call:

```
class Dispatch {  
    public static void main(String args[]) {  
        A a = new A();  
        B b = new B();  
        C c = new C();  
        A r;  
        r = a; r.callme();  
        r = b; r.callme();  
        r = c; r.callme();  
    }  
}
```


Polymorphism Again

One interface, many behaviors:

- 1) super-class defines common methods for sub-classes
- 2) sub-class provides specific implementations for some of the methods of the super-class

A combination of inheritance and overriding – sub-classes retain flexibility to define their own methods, yet they still have to follow a consistent interface.

Example: Polymorphism 1

A class that stores the dimensions of various 2-dimensional objects:

```
class Figure {  
    double dim1;  
    double dim2;  
    Figure(double a, double b) {  
        dim1 = a; dim2 = b;  
    }  
    double area() {  
        System.out.println("Area is undefined.");  
        return 0;  
    }  
}
```

Example: Polymorphism 2

Rectangle is a sub-class of Figure:

```
class Rectangle extends Figure {  
  
    Rectangle(double a, double b) {  
        super(a, b);  
    }  
  
    double area() {  
        System.out.println("Inside Area for Rectangle.");  
        return dim1 * dim2;  
    }  
}
```

Example: Polymorphism 3

Triangle is a sub-class of Figure:

```
class Triangle extends Figure {  
  
    Triangle(double a, double b) {  
        super(a, b);  
    }  
  
    double area() {  
        System.out.println("Inside Area for Triangle.");  
        return dim1 * dim2 / 2;  
    }  
}
```

Example: Polymorphism 4

Invoked through the `Figure` variable and overridden in their respective subclasses, the `area()` method returns the area of the invoking object:

```
class FindAreas {  
    public static void main(String args[]) {  
        Figure f = new Figure(10, 10);  
        Rectangle r = new Rectangle(9, 5);  
        Triangle t = new Triangle(10, 8);  
        Figure figref;  
        figref = r; System.out.println(figref.area());  
        figref = t; System.out.println(figref.area());  
        figref = f; System.out.println(figref.area());  
    }  
}
```

Abstract Method

Inheritance allows a sub-class to override the methods of its super-class.

In fact, a super-class may altogether leave the implementation details of a method and declare such a method **abstract**:

```
abstract type name(parameter-list);
```

Two kinds of methods:

- 1) concrete – may be overridden by sub-classes
- 2) abstract – must be overridden by sub-classes

It is illegal to define abstract constructors or static methods.

Example: Abstract Method

The `area` method cannot compute the area of an arbitrary figure:

```
double area() {  
    System.out.println("Area is undefined.");  
    return 0;  
}
```

Instead, `area` should be defined abstract in `Figure`:

```
abstract double area() ;
```

Abstract Class

A class that contains an abstract method must be itself declared **abstract**:

```
abstract class abstractClassName {  
    abstract type methodName(parameter-list) {  
        ...  
    }  
    ...  
}
```

An abstract class has no instances - it is illegal to use the `new` operator:

```
abstractClassName a = new abstractClassName () ;
```

It is legal to define variables of the abstract class type.

Abstract Sub-Class

A sub-class of an abstract class:

- 1) implements all abstract methods of its super-class, or
- 2) is also declared as an abstract class

```
abstract class A {  
    abstract void callMe();  
}
```

```
abstract class B extends A {  
    int checkMe;  
}
```

Abstract and Concrete Classes 1

Abstract super-class, concrete sub-class:

```
abstract class A {  
    abstract void callme();  
    void callmetoo() {  
        System.out.println("This is a concrete method.");  
    }  
}  
  
class B extends A {  
    void callme() {  
        System.out.println("B's implementation.");  
    }  
}
```

Abstract and Concrete Classes 2

Calling concrete and overridden abstract methods:

```
class AbstractDemo {  
    public static void main(String args[]) {  
        B b = new B();  
  
        b.callme();  
        b.callmetoo();  
    }  
}
```

Example: Abstract Class 1

Figure is an abstract class; it contains an abstract `area` method:

```
abstract class Figure {  
    double dim1;  
    double dim2;  
  
    Figure(double a, double b) {  
        dim1 = a; dim2 = b;  
    }  
  
    abstract double area();  
}
```

Example: Abstract Class 2

Rectangle is concrete – it provides a concrete implementation for `area`:

```
class Rectangle extends Figure {  
    Rectangle(double a, double b) {  
        super(a, b);  
    }  
  
    double area() {  
        System.out.println("Inside Area for Rectangle.");  
        return dim1 * dim2;  
    }  
}
```

Example: Abstract Class 3

Triangle is concrete – it provides a concrete implementation for `area`:

```
class Triangle extends Figure {  
    Triangle(double a, double b) {  
        super(a, b);  
    }  
  
    double area() {  
        System.out.println("Inside Area for Triangle.");  
        return dim1 * dim2 / 2;  
    }  
}
```

Example: Abstract Class 4

Invoked through the `Figure` variable and overridden in their respective subclasses, the `area()` method returns the area of the invoking object:

```
class AbstractAreas {  
    public static void main(String args[]) {  
        Rectangle r = new Rectangle(9, 5);  
        Triangle t = new Triangle(10, 8);  
  
        Figure figref;  
  
        figref = r; System.out.println(figref.area());  
        figref = t; System.out.println(figref.area());  
    }  
}
```

Abstract Class References

It is illegal to create objects of the abstract class:

```
Figure f = new Figure(10, 10);
```

It is legal to create a variable with the abstract class type:

```
Figure figref;
```

Later, `figref` may be used to assign references to any object of a concrete sub-class of `Figure` (e.g. `Rectangle`) and to invoke methods of this class:

```
Rectangle r = new Rectangle(9, 5);  
figref = r; System.out.println(figref.area());
```


Uses of final

The final keyword has three uses:

- 1) declare a variable which value cannot change after initialization
- 2) declare a method which cannot be overridden in sub-classes
- 3) declare a class which cannot have any sub-classes

(1) has been discussed before.

Now is time for (2) and (3).

Preventing Overriding with final

A method declared `final` cannot be overridden in any sub-class:

```
class A {  
    final void meth() {  
        System.out.println("This is a final method.");  
    }  
}
```

This class declaration is illegal:

```
class B extends A {  
    void meth() {  
        System.out.println("Illegal!");  
    }  
}
```

final and Early Binding

Two types of method invocation:

- 1) **early binding** – method call is decided at compile-time
- 2) **late binding** – method call is decided at run-time

By default, method calls are resolved at run-time.

As a final method cannot be overridden, their invocations are resolved at compile-time. This is one way to improve performance of a method call.

Preventing Inheritance with final

A class declared `final` cannot be inherited – has no sub-classes.

```
final class A { ... }
```

This class declaration is considered illegal:

```
class B extends A { ... }
```

Declaring a class `final` implicitly declares all its methods `final`.

It is illegal to declare a class as both `abstract` and `final`.

Object Class

`Object` class is a super-class of all Java classes:

- 1) `Object` is the root of the Java inheritance hierarchy.
- 2) A variable of the `Object` type may refer to objects of any class.
- 3) As arrays are implemented as objects, it may also refer to any array.

Object Class Methods 1

Methods declared in the `Object` class:

- 1) `Object clone()` - creates an object which is an ideal copy of the invoking object.
- 2) `boolean equals(Object object)` - determines if the invoking object and the argument object are the same.
- 3) `void finalize()` – called before an unused object is recycled
- 4) `Class getClass()` – obtains the class description of an object at run-time
- 5) `int hashCode()` – returns the hash code for the invoking object

Object Class Methods 2

- 6) `void notify()` – resumes execution of a thread waiting on the invoking object
- 7) `void notifyAll()` – resumes execution of all threads waiting on the invoking object
- 8) `String toString()` – returns the string that describes the invoking object
- 9) three methods to wait on another thread of execution:
 - a) `void wait()`
 - b) `void wait(long milliseconds)`
 - c) `void wait(long milliseconds, int nanoseconds)`

Overriding Object Class Methods

All methods except `getClass`, `notify`, `notifyAll` and `wait` can be overridden.

Two methods are frequently overridden:

1) `equals()`

2) `toString()`

This way, classes can tailor the equality and the textual description of objects to their own specific structure and needs.

Exercise: Polymorphism

- 1) Define a relationship among the following `Building`, `HighBuilding` and `Skyscraper` classes.
- 2) Define a class `Visits` that stores an array of 10 buildings (representing a street).
- 3) Define a method that enters all the buildings in the street using the method `enter`, one after another.
- 4) Fill the array with mixed objects from the classes `Building`, `HighBuilding` and `Skyscraper`.

Make sure, that the output of your program visualizes the fact that different method implementations are used depending on the type of the actual object.

Access

Course Outline

- | | | |
|---|---|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|---|---|

Name Space Management

Classes written so far all belong to a single name space: a unique name has to be chosen for each class to avoid name collision.

Some way to manage the name space is needed to:

- 1) ensure that the names are unique
- 2) provide a continuous supply of convenient, descriptive names
- 3) ensure that the names chosen by one programmer will not collide with those chosen by another programmers

Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is a **package**.

Package

A package is both a naming and a visibility control mechanism:

- 1) divides the name space into disjoint subsets

It is possible to define classes within a package that are not accessible by code outside the package.

- 2) controls the visibility of classes and their members

It is possible to define class members that are only exposed to other members of the same package.

Same-package classes may have an intimate knowledge of each other, but not expose that knowledge to other packages.

Package Definition

A package statement inserted as the first line of the source file:

```
package myPackage;  
class MyClass1 { ... }  
class MyClass2 { ... }
```

means that all classes in this file belong to the `myPackage` package. The package statement creates a name space where such classes are stored.

When the package statement is omitted, class names are put into the default package which has no name.

Multiple Source Files

Other files may include the same package instruction:

```
package myPackage;  
class MyClass1 { ... }  
class MyClass2 { ... }
```

```
package myPackage;  
class MyClass3{ ... }
```

A package may be distributed through several source files.

Packages and Directories

Java uses file system directories to store packages.

Consider the Java source file:

```
package myPackage;  
class MyClass1 { ... }  
class MyClass2 { ... }
```

The bytecode files `MyClass1.class` and `MyClass2.class` must be stored in a directory `myPackage`.

Case is significant! Directory names must match package names exactly.

Package Hierarchy

To create a package hierarchy, separate each package name with a dot:

```
package myPackage1.myPackage2.myPackage3;
```

A package hierarchy must be stored accordingly in the file system:

- | | |
|--------------|---|
| 1) Unix | <code>myPackage1/myPackage2/myPackage3</code> |
| 2) Windows | <code>myPackage1\myPackage2\myPackage3</code> |
| 3) Macintosh | <code>myPackage1:myPackage2:myPackage3</code> |

You cannot rename a package without renaming its directory!

Finding Packages

As packages are stored in directories, how does the Java run-time system know where to look for packages?

Two ways:

- 1) The current directory is the default start point - if packages are stored in the current directory or sub-directories, they will be found.
- 2) Specify a directory path or paths by setting the `CLASSPATH` environment variable.

CLASSPATH Variable

`CLASSPATH` - environment variable that points to the root directory of the system's package hierarchy.

Several root directories may be specified in `CLASSPATH`, e.g. the current directory and the `C:\myJava` directory:

```
.;C:\myJava
```

Java will search for the required packages by looking up subsequent directories described in the `CLASSPATH` variable.

Finding Packages

Consider this package statement:

```
package myPackage;
```

In order for a program to find `myPackage`, one of the following must be true:

- 1) program is executed from the directory immediately above `myPackage` (the parent of `myPackage` directory)
- 2) `CLASSPATH` must be set to include the path to `myPackage`

Example: Package 1

```
package MyPack;

class Balance {
    String name;
    double bal;
    Balance(String n, double b) {
        name = n; bal = b;
    }
    void show() {
        if (bal<0) System.out.print("-->> ");
        System.out.println(name + ": $" + bal);
    }
}
```

Example: Package 2

```
class AccountBalance {  
    public static void main(String args[]) {  
        Balance current[] = new Balance[3];  
  
        current[0] = new Balance("K. J. Fielding", 123.23);  
        current[1] = new Balance("Will Tell", 157.02);  
        current[2] = new Balance("Tom Jackson", -12.33);  
  
        for (int i=0; i<3; i++) current[i].show();  
    }  
}
```

Example: Package 3

Save, compile and execute:

- 1) call the file `AccountBalance.java`
- 2) save the file in the directory `MyPack`
- 3) compile; `AccountBalance.class` should be also in `MyPack`
- 4) set access to `MyPack` in `CLASSPATH` variable, or make the parent of `MyPack` your current directory
- 5) run:

```
java MyPack.AccountBalance
```

Make sure to use the package-qualified class name.

Importing of Packages

Since classes within packages must be fully-qualified with their package names, it would be tedious to always type long dot-separated names.

The import statement allows to use classes or whole packages directly.

Importing of a concrete class:

```
import myPackage1.myPackage2.myClass;
```

Importing of all classes within a package:

```
import myPackage1.myPackage2.*;
```


Access Control

Classes and packages are both means of encapsulating and containing the name space and scope of classes, variables and methods:

- 1) packages act as a container for classes and other packages
- 2) classes act as a container for data and code

Access control is set separately for classes and class members.

Access Control: Classes

Two levels of access:

- 1) A class available in the whole program:

```
public class MyClass { ... }
```

- 2) A class available within the same package only:

```
class MyClass { ... }
```

Access Control: Members

Four levels of access:

- 1) a member is available in the whole program:

```
public int variable;  
public int method(...) { ... }
```

- 2) a member is only available within the same class:

```
private int variable;  
private int method(...) { ... }
```

Access Control: Members

- 3) a member is available within the same package (default access):

```
int variable;  
int method(...) { ... }
```

- 4) a member is available within the same package as the current class, or within its sub-classes:

```
protected int variable;  
protected int method(...) { ... }
```

The sub-class may be located inside or outside the current package.

Access Control Summary

Complicated?

Any member declared `public` can be accessed from anywhere.

Any member declared `private` cannot be seen outside its class.

When a member does not have any access specification (default access), it is visible to all classes within the same package.

To make a member visible outside the current package, but only to sub-classes of the current class, declare this member `protected`.

Table: Access Control

	private	default	protected	public
same class	yes	yes	yes	yes
same package subclass	no	yes	yes	yes
same package non-sub-class	no	yes	yes	yes
different package sub-class	no	no	yes	yes
different package non-sub-class	no	no	no	yes

Example: Access 1

Access example with two packages `p1` and `p2` and five classes.

A public `Protection` class is in the package `p1`.

It has four variables with four possible access rights:

```
package p1;
```

```
public class Protection {  
    int n = 1;  
    private int n_pri = 2;  
    protected int n_pro = 3;  
    public int n_pub = 4;  
}
```

Example: Access 2

```
public Protection() {  
    System.out.println("base constructor");  
    System.out.println("n = " + n);  
    System.out.println("n_pri = " + n_pri);  
    System.out.println("n_pro = " + n_pro);  
    System.out.println("n_pub = " + n_pub);  
}  
}
```

The rest of the example tests the access to those variables.

Example: Access 3

Derived class is in the same `p1` package and is the sub-class of `Protection`.

It has access to all variables of `Protection` except the private `n_pri`:

```
package p1;
```

```
class Derived extends Protection {  
    Derived() {  
        System.out.println("derived constructor");  
        System.out.println("n = " + n);  
        System.out.println("n_pro = " + n_pro);  
        System.out.println("n_pub = " + n_pub);  
    }  
}
```

Example: Access 4

SamePackage is in the `p1` package but is not a sub-class of `Protection`.

It has access to all variables of `Protection` except the private `n_pri`:

```
package p1;

class SamePackage {
    SamePackage() {
        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);
        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

Example: Access 5

`Protection2` is a sub-class of `p1.Protection`, but is located in a different package – package `p2`.

`Protection2` has access to the public and protected variables of `Protection`. It has no access to its private and default-access variables:

```
package p2;
```

```
class Protection2 extends p1.Protection {  
    Protection2() {  
        System.out.println("derived other package");  
        System.out.println("n_pro = " + n_pro);  
        System.out.println("n_pub = " + n_pub);  
    }  
}
```

Example: Access 6

`OtherPackage` is in the `p2` package and is not a sub-class of `p1.Protection`.

`OtherPackage` has access to the public variable of `Protection` only. It has no access to its private, protected or default-access variables:

```
class OtherPackage {
    OtherPackage() {
        p1.Protection p = new p1.Protection();
        System.out.println("other package constructor");
        System.out.println("n_pub = " + p.n_pub);
    }
}
```

Example: Access 7

A demonstration to use classes of the `p1` package:

```
package p1;

public class Demo {
    public static void main(String args[]) {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
    }
}
```

Example: Access 8

A demonstration to use classes of the `p2` package:

```
package p2;

public class Demo {
    public static void main(String args[]) {
        Protection2 ob1 = new Protection2();
        OtherPackage ob2 = new OtherPackage();
    }
}
```

Import Statement

The import statement occurs immediately after the package statement and before the class statement:

```
package myPackage;  
import otherPackage1;otherPackage2.otherClass;  
class myClass { ... }
```

The Java system accepts this import statement by default:

```
import java.lang.*;
```

This package includes the basic language functions. Without such functions, Java is of no much use.

Name Conflict 1

Suppose a same-named class occurs in two different imported packages:

```
import otherPackage1.*;  
import otherPackage2.*;  
class myClass { ... otherClass ... }
```

```
package otherPackage1;  
class otherClass { ... }
```

```
package otherPackage2;  
class otherClass { ... }
```


Name Conflict 2

Compiler will remain silent, unless we try to use `otherClass`.
Then it will display an error message.

In this situation we should use the full name:

```
import otherPackage1.*;
import otherPackage2.*;
class myClass {
    ...
    otherPackage1.otherClass
    ...
    otherPackage2.otherClass
    ...
}
```

Short versus Full References

Short reference:

```
import java.util.*;  
class MyClass extends Date { ... }
```

Full reference:

```
class MyClass extends java.util.Date { ... }
```

Only the `public` components in imported package are accessible for non-sub-classes in the importing code!

Example: Packages 1

A package `MyPack` with one `public` class `Balance`. The class has two same-package variables: `public` constructor and a `public` `show` method.

```
package MyPack;
public class Balance {
    String name;
    double bal;
    public Balance(String n, double b) {
        name = n; bal = b;
    }
    public void show() {
        if (bal<0) System.out.print("-->> ");
        System.out.println(name + ": $" + bal);
    }
}
```

Example: Packages 2

The importing code has access to the `public` class `Balance` of the `MyPack` package and its two public members:

```
import MyPack.*;

class TestBalance {
    public static void main(String args[]) {
        Balance test = new Balance("J. J. Jaspers", 99.88);
        test.show();
    }
}
```

Java Source File

Finally, a Java source file consists of:

- 1) a single package instruction (optional)
- 2) several import statements (optional)
- 3) a single public class declaration (required)
- 4) several classes private to the package (optional)

At the minimum, a file contains a single public class declaration.

Exercise: Access

- 1) Create a package `emacao`. Don't forget to insert your package into a directory of the same name. Insert a class `AccessTest` into this package. Define `public`, `default` and `private` data members and methods in your class `AccessTest`.
- 2) Define a second class `Accessor1` in your package that accesses the different kinds of data members of methods (`private`, `public`, `default`).
See what compiler messages you get.
- 3) Define class `Accessor2` outside the package. Again try to access all methods and data members of the class `AccessTest`.
See what compiler messages you get.
- 4) Where are the differences between `Accessor1` and `Accessor2` ?

Interfaces

Course Outline

- | | | |
|---|---|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|---|---|

Interface

Using interface, we specify what a class must do, but not how it does this.

An interface is syntactically similar to a class, but it lacks instance variables and its methods are declared without any body.

An interface is defined with an `interface` keyword.

Interface Format

General format:

```
access interface name {  
    type method-name1 (parameter-list);  
    type method-name2 (parameter-list);  
    ...  
    type var-name1 = value1;  
    type var-nameM = valueM;  
    ...  
}
```

Interface Comments

Two types of access:

- 1) `public` – interface may be used anywhere in a program
- 2) `default` – interface may be used in the current package only

Interface methods have no bodies – they end with the semicolon after the parameter list. They are essentially abstract methods.

An interface may include variables, but they must be `final`, `static` and initialized with a constant value.

In a `public` interface, all members are implicitly `public`.

Interface Implementation

A class implements an interface if it provides a complete set of methods defined by this interface.

- 1) any number of classes may implement an interface
- 2) one class may implement any number of interfaces

Each class is free to determine the details of its implementation.

Implementation relation is written with the `implements` keyword.

Implementation Format

General format of a class that includes the `implements` clause:

```
access class name
    extends super-class
    implements interface1, interface2, ..., interfaceN {
    ...
}
```

Access is `public` or default.

Implementation Comments

If a class implements several interfaces, they are separated with a comma.

If a class implements two interfaces that declare the same method, the same method will be used by the clients of either interface.

The methods that implement an interface must be declared `public`.

The type signature of the implementing method must match exactly the type signature specified in the interface definition.

Example: Interface

Declaration of the `Callback` interface:

```
interface Callback {  
    void callback(int param);  
}
```

`Client` class implements the `Callback` interface:

```
class Client implements Callback {  
    public void callback(int p) {  
        System.out.println("callback called with " + p);  
    }  
}
```

More Methods in Implementation

An implementing class may also declare its own methods:

```
class Client implements Callback {  
    public void callback(int p) {  
        System.out.println("callback called with " + p);  
    }  
  
    void nonIfaceMeth() {  
        System.out.println("Classes that implement " +  
            "interfaces may also define " +  
            "other members, too.");  
    }  
}
```


Interface as a Type

Variable may be declared with interface as its type:

```
interface MyInterface { ... }  
  
...  
MyInterface mi;
```

The variable of an interface type may reference an object of any class that implements this interface.

```
class MyClass implements MyInterface { ... }  
  
MyInterface mi = new MyClass();
```

Call Through Interface Variable

Using the interface type variable, we can call any method in the interface:

```
interface MyInterface {  
    void myMethod(...) ;  
    ...  
}  
class MyClass implements MyInterface { ... }  
...  
MyInterface mi = new MyClass();  
...  
mi.myMethod();
```

The correct version of the method will be called based on the actual instance of the interface being referred to.

Example: Call Through Interface 1

Declaration of the `Callback` interface:

```
interface Callback {  
    void callback(int param);  
}
```

`Client` class implements the `Callback` interface:

```
class Client implements Callback {  
    public void callback(int p) {  
        System.out.println("callback called with " + p);  
    }  
}
```

Example: Call Through Interface 2

`TestIface` declares the `Callback` interface variable, initializes it with the new `Client` object, and calls the `callback` method through this variable:

```
class TestIface {  
    public static void main(String args[]) {  
        Callback c = new Client();  
        c.callback(42);  
    }  
}
```

Call Through Interface Variable 2

Call through an interface variable is one of the key features of interfaces:

- 1) the method to be executed is looked up dynamically at run-time
- 2) the calling code can dispatch through an interface without having to know anything about the callee

Allows classes to be created later than the code that calls methods on them.

Example: Interface Call 1

Another implementation of the `Callback` interface:

```
class AnotherClient implements Callback {  
    public void callback(int p) {  
        System.out.println("Another version of callback");  
        System.out.println("p squared is " + (p*p));  
    }  
}
```

Example: Interface Call 2

Callback variable `c` is assigned `Client` and later `AnotherClient` objects and the corresponding `callback` is invoked depending on its value:

```
class TestIface2 {  
    public static void main(String args[]) {  
        Callback c = new Client();  
        c.callback(42);  
        AnotherClient ob = new AnotherClient();  
        c = ob;  
        c.callback(42);  
    }  
}
```

Compile-Time Method Binding

Normally, in order for a method to be called from one class to another, both classes must be present at compile time.

This implies:

- 1) a static, non-extensible classing environment
- 2) functionality gets pushed higher and higher in the class hierarchy to make them available to more sub-classes

Run-Time Method Binding

Interfaces support dynamic method binding.

Interface disconnects the method definition from the inheritance hierarchy:

- 1) interfaces are in a different hierarchy from classes
- 2) it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface

Interface and Abstract Class

A class that claims to implement an interface but does not implement all its methods must be declared abstract.

`Incomplete` class implements the `Callback` interface but not its `callback` method, so the class is declared `abstract`:

```
abstract class Incomplete implements Callback {  
    int a, b;  
    void show() {  
        System.out.println(a + " " + b);  
    }  
}
```

Example: Stack Interface

Many ways to implement a stack but one interface:

```
interface IntStack {  
    void push(int item);  
    int pop();  
}
```

Lets look at two implementations of this interface:

- 1) `FixedStack` – a fixed-length version of the integer stack
- 2) `DynStack` – a dynamic-length version of the integer stack

Example: FixedStack 1

A fixed-length stack implements the `IntStack` interface with two private variables, a constructor and two public methods:

```
class FixedStack implements IntStack {  
    private int stck[];  
    private int tos;  
  
    FixedStack(int size) {  
        stck = new int[size]; tos = -1;  
    }  
}
```

Example: FixedStack 2

```
public void push(int item) {
    if (tos==stck.length-1)
        System.out.println("Stack is full.");
    else stck[++tos] = item;
}

public int pop() {
    if (tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
    }
    else return stck[tos--];
}
}
```

Example: FixedStack 3

A testing class creates two stacks:

```
class IFTest {  
    public static void main(String args[]) {  
        FixedStack mystack1 = new FixedStack(5);  
        FixedStack mystack2 = new FixedStack(8);  
    }  
}
```

Example: FixedStack 4

It pushes and then pops off some values from those stacks:

```
for (int i=0; i<5; i++) mystack1.push(i);
for (int i=0; i<8; i++) mystack2.push(i);

System.out.println("Stack in mystack1:");
for (int i=0; i<5; i++)
    System.out.println(mystack1.pop());

System.out.println("Stack in mystack2:");
for (int i=0; i<8; i++)
    System.out.println(mystack2.pop());
}
}
```

Example: DynStack 1

Another implementation of an integer stack.

A dynamic-length stack is first created with an initial length. The stack is doubled in size every time this initial length is exceeded.

```
class DynStack implements IntStack {  
    private int stck[];  
    private int tos;  
  
    DynStack(int size) {  
        stck = new int[size];  
        tos = -1;  
    }  
}
```


Example: DynStack 2

If stack is full, `push` creates a new stack with double the size of the old stack:

```
public void push(int item) {
    if (tos==stck.length-1) {
        int temp[] = new int[stck.length * 2];
        for (int i=0; i<stck.length; i++)
            temp[i] = stck[i];
        stck = temp;
        stck[++tos] = item;
    }
    else stck[++tos] = item;
}
```

Example: DynStack 3

If the stack is empty, `pop` returns the zero value:

```
public int pop() {  
    if(tos < 0) {  
        System.out.println("Stack underflow.");  
        return 0;  
    }  
    else return stck[tos--];  
}
```

Example: DynStack 4

The testing class creates two dynamic-length stacks:

```
class IFTest2 {  
    public static void main(String args[]) {  
        DynStack mystack1 = new DynStack(5);  
        DynStack mystack2 = new DynStack(8);  
    }  
}
```

Example: DynStack 5

It then pushes some numbers onto those stacks, dynamically increasing their size, then pops those numbers off:

```
for (int i=0; i<12; i++) mystack1.push(i);  
for (int i=0; i<20; i++) mystack2.push(i);
```

```
System.out.println("Stack in mystack1:");  
for (int i=0; i<12; i++)  
    System.out.println(mystack1.pop());
```

```
System.out.println("Stack in mystack2:");  
for (int i=0; i<20; i++)  
    System.out.println(mystack2.pop());
```

```
}
```

```
}
```

Example: Two Stacks 1

Testing two stack implementations through an interface variable.

First, some numbers are pushed onto both stacks:

```
class IFTest3 {  
    public static void main(String args[]) {  
        IntStack mystack;  
        DynStack ds = new DynStack(5);  
        FixedStack fs = new FixedStack(8);  
  
        mystack = ds;  
        for (int i=0; i<12; i++) mystack.push(i);  
        mystack = fs;  
        for (int i=0; i<8; i++) mystack.push(i);  
    }  
}
```

Example: Two Stacks 2

Then, those numbers are popped off:

```
mystack = ds;
System.out.println("Values in dynamic stack:");
for (int i=0; i<12; i++)
    System.out.println(mystack.pop());
mystack = fs;
System.out.println("Values in fixed stack:");
for (int i=0; i<8; i++)
    System.out.println(mystack.pop());
}
}
```

Which stack implementation is the value of the `mystack` variable, therefore which version of `push` and `pop` are used, is determined at run-time.

Interface Variables

Variables declared in an interface must be constants.

A technique to import shared constants into multiple classes:

- 1) declare an interface with variables initialized to the desired values
- 2) include that interface in a class through implementation

As no methods are included in the interface, the class does not implement anything except importing the variables as constants.

Example: Interface Variables 1

An interface with constant values:

```
import java.util.Random;

interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}
```


Example: Interface Variables 2

`Question` implements `SharedConstants`, including all its constants.

Which constant is returned depends on the generated random number:

```
class Question implements SharedConstants {
    Random rand = new Random();
    int ask() {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 30)      return NO;
        else if (prob < 60) return YES;
        else if (prob < 75) return LATER;
        else if (prob < 98) return SOON;
        else return NEVER;
    }
}
```

Example: Interface Variables 3

`AskMe` includes all shared constants in the same way, using them to display the result, depending on the value received:

```
class AskMe implements SharedConstants {  
    static void answer(int result) {  
        switch(result) {  
            case NO:      System.out.println("No"); break;  
            case YES:     System.out.println("Yes"); break;  
            case MAYBE:   System.out.println("Maybe"); break;  
            case LATER:   System.out.println("Later"); break;  
            case SOON:    System.out.println("Soon"); break;  
            case NEVER:   System.out.println("Never"); break;  
        }  
    }  
}
```

Example: Interface Variables 4

The testing function relies on the fact that both `ask` and `answer` methods, defined in different classes, rely on the same constants:

```
public static void main(String args[]) {  
    Question q = new Question();  
    answer(q.ask());  
    answer(q.ask());  
    answer(q.ask());  
    answer(q.ask());  
}  
}
```

Interface Inheritance

One interface may inherit another interface.

The inheritance syntax is the same for classes and interfaces.

```
interface MyInterface1 {  
    void myMethod1 (...) ;  
}  
interface MyInterface2 extends MyInterface1 {  
    void myMethod2 (...) ;  
}
```

When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Inheritance and Implementation

When a class implements an interface that inherits another interface, it must provide implementations for all inherited methods:

```
class MyClass implements MyInterface2 {  
    void myMethod1(...) { ... }  
    void myMethod1(...) { ... }  
    ...  
}
```

Example: Interface Inheritance 1

Consider interfaces `A` and `B`.

```
interface A {  
    void meth1();  
    void meth2();  
}
```

`B` extends `A`:

```
interface B extends A {  
    void meth3();  
}
```

Example: Interface Inheritance 2

MyClass must implement all of A and B methods:

```
class MyClass implements B {  
    public void meth1() {  
        System.out.println("Implement meth1().");  
    }  
    public void meth2() {  
        System.out.println("Implement meth2().");  
    }  
    public void meth3() {  
        System.out.println("Implement meth3().");  
    }  
}
```

Example: Interface Inheritance 3

Create a new `MyClass` object, then invoke all interface methods on it:

```
class IFExtend {  
    public static void main(String arg[]) {  
        MyClass ob = new MyClass();  
        ob.meth1();  
        ob.meth2();  
        ob.meth3();  
    }  
}
```


Exercise: Interface

- 1) Define two interfaces:
 - a) an interface `CardUse` for card use with methods `read` to read the state and `reduceBy` with a parameter amount to change the state of the card. The user has to identify himself by a PIN for this operation;
 - b) an interface `CardChange` for the administration of the card by authorized people, that need a method `reset` to reset the card state, a method `fill` to fill the card with an amount of money and a method `changePIN` to change the PIN for the card.
- 2) Define a third interface `CardAll` that includes all card operation (Use inheritance).
- 3) Change the interface `CardAll` into an abstract class that implements the balance of the card and a basic solution for the methods `fill` and `reduceBy` leaving the rest of the methods abstract. Choose the correct access specifier to make the balance accessible to subclasses but not to the public; Check this;

Exception-Handling

Course Outline

- | | | |
|---|---|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|---|---|

Exceptions

Exception is an abnormal condition that arises when executing a program.

In the languages that do not support exception handling, errors must be checked and handled manually, usually through the use of error codes.

In contrast, Java:

- 1) provides syntactic mechanisms to signal, detect and handle errors
- 2) ensures a clean separation between the code executed in the absence of errors and the code to handle various kinds of errors
- 3) brings run-time error management into object-oriented programming

Exception Handling

An exception is an object that describes an exceptional condition (error) that has occurred when executing a program.

Exception handling involves the following:

- 1) when an error occurs, an object (exception) representing this error is created and **thrown** in the method that caused it
- 2) that method may choose to **handle** the exception itself or **pass** it on
- 3) either way, at some point, the exception is **caught** and processed

Exception Sources

Exceptions can be:

- 1) generated by the Java run-time system

Fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.

- 2) manually generated by programmer's code

Such exceptions are typically used to report some error conditions to the caller of a method.

Exception Constructs

Five constructs are used in exception handling:

- 1) `try` – a block surrounding program statements to monitor for exceptions
- 2) `catch` – together with `try`, catches specific kinds of exceptions and handles them in some way
- 3) `finally` – specifies any code that absolutely must be executed whether or not an exception occurs
- 4) `throw` – used to throw a specific exception from the program
- 5) `throws` – specifies which exceptions a given method can throw

Exception-Handling Block

General form:

```
try { ... }  
catch (Exception1 ex1) { ... }  
catch (Exception2 ex2) { ... }  
...  
finally { ... }
```

where:

- 1) `try { ... }` is the block of code to monitor for exceptions
- 2) `catch (Exception ex) { ... }` is exception handler for the exception `Exception`
- 3) `finally { ... }` is the block of code to execute before the `try` block ends

Exception Hierarchy

All exceptions are sub-classes of the build-in class `Throwable`.

`Throwable` contains two immediate sub-classes:

- 1) `Exception` – exceptional conditions that programs should catch

The class includes:

- a) `RuntimeException` – defined automatically for user programs to include: division by zero, invalid array indexing, etc.
- b) use-defined exception classes

- 2) `Error` – exceptions used by Java to indicate errors with the run-time environment; user programs are not supposed to catch them

Uncaught Exception

What happens when exceptions are not handled?

```
class Exc0 {  
    public static void main(String args[]) {  
        int d = 0;  
        int a = 42 / d;  
    }  
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and throws this object.

This will cause the execution of `Exc0` to stop – once an exception has been thrown it must be caught by an exception handler and dealt with.

Default Exception Handler

As we have not provided any exception handler, the exception is caught by the default handler provided by the Java run-time system.

This default handler:

- 1) displays a string describing the exception,
- 2) prints the stack trace from the point where the exception occurred
- 3) terminates the program

```
java.lang.ArithmeticException: / by zero  
at Exc0.main(Exc0.java:4)
```

Any exception not caught by the user program is ultimately processed by the default handler.

Stack Trace Display

The stack trace displayed by the default error handler shows the sequence of method invocations that led up to the error.

Here the exception is raised in `subroutine()` which is called by `main()`:

```
class Excl {  
    static void subroutine() {  
        int d = 0;  
        int a = 10 / d;  
    }  
    public static void main(String args[]) {  
        Excl.subroutine();  
    }  
}
```

Own Exception Handling

Default exception handling is basically useful for debugging.

Normally, we want to handle exceptions ourselves because:

- 1) if we detected the error, we can try to fix it
- 2) we prevent the program from automatically terminating

Exception handling is done through the **try and catch** block.

Try and Catch 1

Try and catch:

- 1) `try` surrounds any code we want to monitor for exceptions
- 2) `catch` specifies which exception we want to handle and how.

When an exception is thrown in the try block:

```
try {  
    d = 0;  
    a = 42 / d;  
    System.out.println("This will not be printed.");  
}
```

Try and Catch 2

control moves immediately to the catch block:

```
catch (ArithmeticException e) {  
    System.out.println("Division by zero.");  
}
```

The exception is handled and the execution resumes.

The scope of catch is restricted to the immediately preceding try statement - it cannot catch exceptions thrown by another try statements.

Try and Catch 3

Resumption occurs with the next statement after the try/catch block:

```
try { ... }  
catch (ArithmeticException e) { ... }  
System.out.println("After catch statement.");
```

Not with the next statement after `a = 42/d;` which caused the exception!

```
a = 42 / d;  
System.out.println("This will not be printed.");
```


Catch and Continue 1

The purpose of catch should be to resolve the exception and then continue as if the error had never happened.

Try/catch block inside a loop:

```
import java.util.Random;

class HandleError {
    public static void main(String args[]) {
        int a=0, b=0, c=0;
        Random r = new Random();
```

Catch and Continue 2

After exception-handling, the program continues with the next iteration:

```
for (int i=0; i<32000; i++) {  
    try {  
        b = r.nextInt();  
        c = r.nextInt();  
        a = 12345 / (b/c);  
    } catch (ArithmeticException e) {  
        System.out.println("Division by zero.");  
        a = 0; // set a to zero and continue  
    }  
    System.out.println("a: " + a);  
}  
}
```

Exception Display

All exception classes inherit from the `Throwable` class.

`Throwable` overrides `toString()` to describe the exception textually:

```
try { ... }  
catch (ArithmeticException e) {  
    System.out.println("Exception: " + e);  
}
```

The following text will be displayed:

```
Exception: java.lang.ArithmeticException: / by zero
```

Multiple Catch Clauses

When more than one exception can be raised by a single piece of code, several `catch` clauses can be used with one `try` block:

- 1) each `catch` catches a different kind of exception
- 2) when an exception is thrown, the first one whose type matches that of the exception is executed
- 3) after one `catch` executes, the other are bypassed and the execution continues after the try/catch block

Example: Multiple Catch 1

Two different exception types are possible in the following code: division by zero and array index out of bound:

```
class MultiCatch {  
    public static void main(String args[]) {  
        try {  
            int a = args.length;  
            System.out.println("a = " + a);  
            int b = 42 / a;  
            int c[] = { 1 };  
            c[42] = 99;  
        }  
    }  
}
```

Example: Multiple Catch 2

Both exceptions can be caught by the following catch clauses:

```
        catch (ArithmeticException e) {  
            System.out.println("Divide by 0: " + e);  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Array index oob: " + e);  
        }  
        System.out.println("After try/catch blocks.");  
    }  
}
```

Order of Multiple Catch Clauses

Order is important:

- 1) catch clauses are inspected top-down
- 2) a clause using a super-class will catch all sub-class exceptions

Therefore, specific exceptions should appear before more general ones.

In particular, exception sub-classes must appear before super-classes.

Example: Multiple Catch Order 1

A try block with two catch clauses:

```
class SuperSubCatch {  
    public static void main(String args[]) {  
        try {  
            int a = 0;  
            int b = 42 / a;  

```

This exception is more general but occurs first:

```
        } catch (Exception e) {  
            System.out.println("Generic Exception catch.");  
        }
```


Example: Multiple Catch Order 2

This exception is more specific but occurs last:

```
        catch (ArithmeticException e) {  
            System.out.println("This is never reached.");  
        }  
    }  
}
```

The second clause will never get executed. A compile-time error (unreachable code) will be raised.

Nested try Statements

The try statements can be nested:

- 1) if an inner try does not catch a particular exception
- 2) the exception is inspected by the outer try block
- 3) this continues until:
 - a) one of the catch statements succeeds or
 - b) all the nested try statements are exhausted
- 4) in the latter case, the Java run-time system will handle the exception

Example: Nested try 1

An example with two nested try statements:

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

Outer try statement:

```
        try {  
            int a = args.length;
```

Division by zero when no command-line argument is present:

```
            int b = 42 / a;  
            System.out.println("a = " + a);
```

Example: Nested try 2

Inner try statement:

```
try {
```

Division by zero when one command-line argument is present:

```
if (a==1) a = a / (a-a);
```

Array index out of bound when two command-line arguments are present:

```
if (a==2) {  
    int c[] = { 1 };  
    c[42] = 99;  
}
```

Example: Nested try 3

Catch statement for the inner try statement, catches the array index out of bound exception:

```
    } catch (ArrayIndexOutOfBoundsException e) {  
        System.out.println(e);  
    }
```

Catch statement for the outer try statement, catches both division-by-zero exceptions for the inner and outer try statements:

```
    } catch (ArithmeticException e) {  
        System.out.println("Divide by 0: " + e);  
    }  
}  
}
```

Method Calls and Nested try 1

Nesting of try statements with method calls:

- 1) method call is enclosed within one try statement
- 2) the method includes another try statement

Still, the try blocks are considered nested within each other.

```
class MethNestTry {
```

Method Calls and Nested try 2

Method `nesttry` contains the try statement:

```
static void nesttry(int a) {  
    try {  
        if (a==1) a = a/(a-a);  
        if (a==2) {  
            int c[] = { 1 }; c[42] = 99;  
        }  
    } catch (ArrayIndexOutOfBoundsException e) {  
        System.out.println(e);  
    }  
}
```

Method Calls and Nested try 3

Method `main` contains another try block which includes the call to `nesttry`:

```
public static void main(String args[]) {  
    try {  
        int a = args.length;  
        int b = 42 / a;  
        System.out.println("a = " + a);  
        nesttry(a);  
    }  
    catch (ArithmeticException e) {  
        System.out.println("Divide by 0: " + e);  
    }  
}
```


Throwing Exceptions

So far, we were only catching the exceptions thrown by the Java system.

In fact, a user program may throw an exception explicitly:

```
throw ThrowableInstance;
```

`ThrowableInstance` must be an object of type `Throwable` or its subclass.

throw Follow-up

Once an exception is thrown by:

```
throw ThrowableInstance;
```

- 1) the flow of control stops immediately
- 2) the nearest enclosing `try` statement is inspected if it has a `catch` statement that matches the type of exception:
 - 1) if one exists, control is transferred to that statement
 - 2) otherwise, the next enclosing `try` statement is examined
- 3) if no enclosing `try` statement has a corresponding `catch` clause, the default exception handler halts the program and prints the stack

Creating Exceptions

Two ways to obtain a `Throwable` instance:

- 1) creating one with the `new` operator

All Java built-in exceptions have at least two constructors: one without parameters and another with one `String` parameter:

```
throw new NullPointerException("demo");
```

- 2) using a parameter of the `catch` clause

```
try { ... } catch(Throwable e) { ... e ... }
```

Example: throw 1

```
class ThrowDemo {
```

The method `demoproc` throws a `NullPointerException` exception which is immediately caught in the try block and re-thrown:

```
    static void demoproc() {  
        try {  
            throw new NullPointerException("demo");  
        } catch (NullPointerException e) {  
            System.out.println("Caught inside demoproc.");  
            throw e;  
        }  
    }
```

Example: throw 2

The main method calls `demoproc` within the try block which catches and handles the `NullPointerException` exception:

```
public static void main(String args[]) {  
    try {  
        demoproc();  
    } catch (NullPointerException e) {  
        System.out.println("Recought: " + e);  
    }  
}
```

throws Declaration

If a method is capable of causing an exception that it does not handle, it must specify this behavior by the `throws` clause in its declaration:

```
type name(parameter-list) throws exception-list {  
    ...  
}
```

where `exception-list` is a comma-separated list of all types of exceptions that a method might throw.

All exceptions must be listed except `Error` and `RuntimeException` or any of their subclasses, otherwise a compile-time error occurs.

Example: throws 1

The `throwOne` method throws an exception that it does not catch, nor declares it within the `throws` clause.

```
class ThrowsDemo {  
    static void throwOne() {  
        System.out.println("Inside throwOne.");  
        throw new IllegalAccessException("demo");  
    }  
    public static void main(String args[]) {  
        throwOne();  
    }  
}
```

Therefore this program does not compile.

Example: throws 2

Corrected program: `throwOne` lists exception, `main` catches it:

```
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
        }
    }
}
```


Motivating finally

When an exception is thrown:

- 1) the execution of a method is changed
- 2) the method may even return prematurely.

This may be a problem in many situations.

For instance, if a method opens a file on entry and closes on exit; exception handling should not bypass the proper closure of the file.

The `finally` block is used to address this problem.

finally Clause

The `try/catch` statement requires at least one `catch` or `finally` clause, although both are optional:

```
try { ... }  
catch (Exception1 ex1) { ... } ...  
finally { ... }
```

Executed after `try/catch` whether or not the exception is thrown.

Any time a method is to return to a caller from inside the `try/catch` block via:

- 1) uncaught exception or
- 2) explicit return

the `finally` clause is executed just before the method returns.

Example: finally 1

Three methods to exit in various ways.

```
class FinallyDemo {
```

`procA` prematurely breaks out of the `try` by throwing an exception, the `finally` clause is executed on the way out:

```
    static void procA() {  
        try {  
            System.out.println("inside procA");  
            throw new RuntimeException("demo");  
        } finally {  
            System.out.println("procA's finally");  
        }  
    }  
}
```

Example: finally 2

`procB`'s `try` statement is exited via a `return` statement, the `finally` clause is executed before `procB` returns:

```
static void procB() {  
    try {  
        System.out.println("inside procB");  
        return;  
    } finally {  
        System.out.println("procB's finally");  
    }  
}
```

Example: finally 3

In `procC`, the `try` statement executes normally without error, however the `finally` clause is still executed:

```
static void procC() {  
    try {  
        System.out.println("inside procC");  
    } finally {  
        System.out.println("procC's finally");  
    }  
}
```

Example: finally 4

Demonstration of the three methods:

```
public static void main(String args[]) {  
    try {  
        procA();  
    } catch (Exception e) {  
        System.out.println("Exception caught");  
    }  
    procB();  
    procC();  
}  
}
```

Java Built-In Exceptions

The default `java.lang` package provides several exception classes, all sub-classing the `RuntimeException` class.

Two sets of build-in exception classes:

- 1) **unchecked exceptions** – the compiler does not check if a method handles or throws there exceptions
- 2) **checked exceptions** – must be included in the method's `throws` clause if the method generates but does not handle them

Unchecked Built-In Exceptions 1

Methods that generate but do not handle those exceptions need not declare them in the `throws` clause:

<code>ArithmeticException</code>	arithmetic error such as divide-by-zero
<code>ArrayIndexOutOfBoundsException</code>	array index out of bounds
<code>ArrayStoreException</code>	assignment to an array element of the wrong type
<code>ClassCastException</code>	invalid cast
<code>IllegalArgumentException</code>	illegal argument used to invoke a method
<code>IllegalMonitorStateException</code>	illegal monitor behavior, e.g. waiting on an unlocked thread
<code>IllegalStateException</code>	environment of application is in incorrect state

Unchecked Built-In Exceptions 2

<code>IllegalThreadStateException</code>	requested operation not compatible with current thread state
<code>IndexOutOfBoundsException</code>	some type of index is out-of-bounds
<code>NegativeArraySizeException</code>	array created with a negative size
<code>NullPointerException</code>	invalid use of null reference
<code>NumberFormatException</code>	invalid conversion of a string to a numeric format
<code>SecurityException</code>	attempt to violate security
<code>StringIndexOutOfBoundsException</code>	attempt to index outside the the bounds of a string
<code>UnsupportedOperationException</code>	an unsupported operation was encountered

Checked Built-In Exceptions

Methods that generate but do not handle those exceptions must declare them in the `throws` clause:

<code>ClassNotFoundException</code>	class not found
<code>CloneNotSupportedException</code>	attempt to clone an object that does not implement the <code>Cloneable</code> interface
<code>IllegalAccessException</code>	access to a class is denied
<code>InstantiationException</code>	attempt to create an object of an abstract class or interface
<code>InterruptedException</code>	one thread has been interrupted by another thread
<code>NoSuchFieldException</code>	a requested field does not exist
<code>NoSuchMethodException</code>	a requested method does not exist

Creating Own Exception Classes

Build-in exception classes handle some generic errors.

For application-specific errors define your own exception classes.

How? Define a subclass of `Exception`:

```
class MyException extends Exception { ... }
```

`MyException` need not implement anything – its mere existence in the type system allows to use its objects as exceptions.

Throwable Class 1

`Exception` itself is a sub-class of `Throwable`.

All user exceptions have the methods defined by the `Throwable` class:

- 1) `Throwable fillInStackTrace()` – returns a `Throwable` object that contains a completed stack trace; the object can be rethrown.
- 2) `Throwable getCause()` – returns the exception that underlies the current exception. If no underlying exception exists, null is returned.
- 3) `String getLocalizedMessage()` – returns a localized description of the exception.
- 4) `String getMessage()` – returns a description of the exception
- 5) `StackTraceElement[] getStackTrace()` – returns an array that contains the stack trace; the method at the top is the last one called before exception.

Throwable Class 2

More methods defined by the `Throwable` class:

- 6) `Throwable initCause(Throwable causeExc)` – associates `causeExc` with the invoking exception as its cause, returns the exception reference
- 7) `void printStackTrace()` – displays the stack trace
- 8) `void printStackTrace(PrintStream stream)` – sends the stack trace to the specified stream
- 9) `void setStackTrace(StackTraceElement elements[])` – sets the stack trace to the elements passed in `elements`; for specialized applications only
- 10) `String toString()` – returns a `String` object containing a description of the exception; called by `print()` when displaying a `Throwable` object.

Example: Own Exceptions 1

A new exception class is defined, with a private `detail` variable, a one-parameter constructor and an overridden `toString` method:

```
class MyException extends Exception {  
    private int detail;  
  
    MyException(int a) {  
        detail = a;  
    }  
  
    public String toString() {  
        return "MyException[" + detail + "];"  
    }  
}
```

Example: Own Exceptions 2

```
class ExceptionDemo {
```

The static `compute` method throws the `MyException` exception whenever its `a` argument is greater than 10:

```
    static void compute(int a) throws MyException {  
        System.out.println("Called compute(" + a + ")");  
        if (a > 10) throw new MyException(a);  
        System.out.println("Normal exit");  
    }
```

Example: Own Exceptions 3

The `main` method calls `compute` with two arguments within a `try` block that catches the `MyException` exception:

```
public static void main(String args[]) {  
    try {  
        compute(1);  
        compute(20);  
    } catch (MyException e) {  
        System.out.println("Caught " + e);  
    }  
}
```


Chained Exceptions 1

The chained exception allows to associate with a given exception another exception that describes its cause.

`Throwable` class includes two constructors to handle chained exceptions:

```
Throwable(Throwable causeExc)
```

```
Throwable(String msg, Throwable causeExc)
```

They both create an exception with `causeExc` being its underlying reason and optional `msg` providing the textual description.

Chained Exceptions 2

`Throwable` class also includes two methods to handle chained exceptions:

- 1) `Throwable getCause()` – returns an exception that is the cause of the current exception, or null if there is no underlying exception.
- 2) `Throwable initCause(Throwable causeExc)` – associates `causeExc` with the invoking exception and returns a reference to the exception:
 - a) `initCause` allows to associate a cause with an existing exception
 - b) the cause exception can be set only once
 - c) if the cause exception was set by a constructor, it is not possible to set it again with `initCause`

Example: Chained Exceptions 1

```
class ChainExcDemo {
```

The `demoproc` method creates a new `NullPointerException` exception `e`, associates `ArithmeticException` as its cause, then throws `e`:

```
    static void demoproc() {  
        NullPointerException e =  
            new NullPointerException("top layer");  
        e.initCause(new ArithmeticException("cause"));  
        throw e;  
    }
```

Example: Chained Exceptions 2

The `main` method calls `demoproc` within the `try` block that catches `NullPointerException`, then displays the exception and its cause:

```
public static void main(String args[]) {  
    try {  
        demoproc();  
    } catch (NullPointerException e) {  
        System.out.println("Caught: " + e);  
        System.out.println("Cause: " + e.getCause());  
    }  
}
```

A cause exception may itself have a cause. In fact, the cause-chain of exceptions may be arbitrarily long.

Exceptions Usage

New Java programmers should break the habit of returning error codes to signal abnormal exit from a method.

Java provides a clean and powerful way to handle errors and unusual boundary conditions through its:

- 1) `try`
- 2) `catch`
- 3) `finally`
- 4) `throw` **and**
- 5) `throws` statements.

However, Java's exception-handling should not be considered a general mechanism for non-local branching!

Exercise: Exception-Handling

- 1) Create exception classes called `Even` and `Odd`
- 2) Generate numbers within an endless loop. Print the generated numbers.
- 3) If the number is even, throw the `Even` exception with the message “`The number thrown an even number`” along with the number.
- 4) If the number is odd, throw the `Odd` exception with the message “`The number thrown an odd number`” along with the number.
- 5) Catch the `Even` exception within the endless loop and print the message.
- 6) Catch the `Odd` exception outside of the loop and print the message.

Multi-Threading

Course Outline

- | | | |
|---|--|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|--|---|

Multi-Tasking

Two kinds of multi-tasking:

- 1) process-based multi-tasking
- 2) thread-based multi-tasking

Process-based multi-tasking is about allowing several programs to execute concurrently, e.g. Java compiler and a text editor.

Processes are heavyweight tasks:

- 1) that require their own address space
- 2) inter-process communication is expensive and limited
- 3) context-switching from one process to another is expensive and limited

Thread-Based Multi-Tasking

Thread-based multi-tasking is about a single program executing concurrently several tasks e.g. a text editor printing and spell-checking text.

Threads are lightweight tasks:

- 1) they share the same address space
- 2) they cooperatively share the same process
- 3) inter-thread communication is inexpensive
- 4) context-switching from one thread to another is low-cost

Java multi-tasking is thread-based.

Reasons for Multi-Threading

Multi-threading enables to write efficient programs that make the maximum use of the CPU, keeping the idle time to a minimum.

There is plenty of idle time for interactive, networked applications:

- 1) the transmission rate of data over a network is much slower than the rate at which the computer can process it
- 2) local file system resources can be read and written at a much slower rate than can be processed by the CPU
- 3) of course, user input is much slower than the computer

Single-Threading

In a single-threaded environment, the program has to wait for each of these tasks to finish before it can proceed to the next.

Single-threaded systems use event loop with pooling:

- 1) a single thread of control runs in an infinite loop
- 2) the loop pools a single event queue to decide what to do next
- 3) the pooling mechanism returns an event
- 4) control is dispatched to the appropriate event handler
- 5) until this event handler returns, nothing else can happen

Threads: Model

Thread exist in several states:

- 1) **ready** to run
- 2) **running**
- 3) a running thread can be **suspended**
- 4) a suspended thread can be **resumed**
- 5) a thread can be **blocked** when waiting for a resource
- 6) a thread can be **terminated**

Once terminated, a thread cannot be resumed.

Threads: Priorities

Every thread is assigned priority – an integer number to decide when to switch from one running thread to the next (context-switching).

Rules for context switching:

- 1) a thread can **voluntarily relinquish control** (sleeping, blocking on I/O, etc.), then the highest-priority ready to run thread is given the CPU.
- 2) a thread can be **preempted by a higher-priority thread** – a lower-priority thread is suspended

When two equal-priority threads are competing for CPU time, which one is chosen depends on the operating system.

Threads: Synchronization

Multi-threading introduces asynchronous behavior to a program. How to ensure synchronous behavior when we need it?

For instance, how to prevent two threads from simultaneously writing and reading the same object?

Java implementation of monitors:

- 1) classes can define so-called **synchronized methods**
- 2) each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called
- 3) once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object

Thread Class

To create a new thread a program will:

- 1) extend the `Thread` class, or
- 2) implement the `Runnable` interface

`Thread` class encapsulates a thread of execution.

The whole Java multithreading environment is based on the `Thread` class.

Thread Methods

<code>getName</code>	obtain a thread's name
<code>getPriority</code>	obtain a thread's priority
<code>isAlive</code>	determine if a thread is still running
<code>join</code>	wait for a thread to terminate
<code>run</code>	entry-point for a thread
<code>sleep</code>	suspend a thread for a period of time
<code>start</code>	start a thread by calling its run method

The Main Thread

The main thread is a thread that begins as soon as a program starts.

The main thread:

- 1) is invoked automatically
- 2) is the first to start and the last to finish
- 3) is the thread from which other “child” threads will be spawned

It can be obtained through the

```
public static Thread currentThread()
```

method of `Thread`.

Example: Main Thread 1

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

The main thread is obtained, displayed, its name changed and re-displayed:

```
        Thread t = Thread.currentThread();  
        System.out.println("Current thread: " + t);  
        t.setName("My Thread");  
        System.out.println("After name change: " + t);
```

Example: Main Thread 2

A loop performs five iterations pausing for a second between the iterations.

It is performed within the `try/catch` block – the `sleep` method may throw `InterruptedException` if some other thread wanted to interrupt:

```
try {
    for (int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
    }
} catch (InterruptedException e) {
    System.out.println("Main thread interrupted");
}
}
```

Example: Thread Methods

Thread methods used by the example:

- 1) `static void sleep(long milliseconds)`
 throws `InterruptedException`

Causes the thread from which it is executed to suspend execution for the specified number of milliseconds.

- 2) `final String getName()`

Allows to obtain the name of the current thread.

- 3) `final void setName(String threadName)`

Sets the name of the current thread.

Creating a Thread

Two methods to create a new thread:

- 1) by implementing the `Runnable` interface
- 2) by extending the `Thread` class

We look at each method in order.

New Thread: Runnable

To create a new thread by implementing the `Runnable` interface:

- 1) create a class that implements the `run` method (inside this method, we define the code that constitutes the new thread):

```
public void run()
```

- 2) instantiate a `Thread` object within that class, a possible constructor is:

```
Thread(Runnable threadOb, String threadName)
```

- 3) call the `start` method on this object (`start` calls `run`):

```
void start()
```

Example: New Thread 1

A class `NewThread` that implements `Runnable`:

```
class NewThread implements Runnable {  
    Thread t;
```

Creating and starting a new thread. Passing `this` to the `Thread` constructor – the new thread will call this object's `run` method:

```
NewThread() {  
    t = new Thread(this, "Demo Thread");  
    System.out.println("Child thread: " + t);  
    t.start();  
}
```


Example: New Thread 2

This is the entry point for the newly created thread – a five-iterations loop with a half-second pause between the iterations all within try/catch:

```
public void run() {  
    try {  
        for (int i = 5; i > 0; i--) {  
            System.out.println("Child Thread: " + i);  
            Thread.sleep(500);  
        }  
    } catch (InterruptedException e) {  
        System.out.println("Child interrupted.");  
    }  
    System.out.println("Exiting child thread.");  
}
```

Example: New Thread 3

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

A new thread is created as an object of `NewThread`:

```
        new NewThread();
```

After calling the `NewThread start` method, control returns here.

Example: New Thread 4

Both threads (new and main) continue concurrently.

Here is the loop for the main thread:

```
try {
    for (int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
    }
} catch (InterruptedException e) {
    System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
```

New Thread: Extend Thread

The second way to create a new thread:

- 1) create a new class that extends `Thread`
- 2) create an instance of that class

Thread provides both `run` and `start` methods:

- 1) the extending class must override `run`
- 2) it must also call the `start` method

Example: New Thread 1

The new thread class extends `Thread`:

```
class NewThread extends Thread {
```

Create a new thread by calling the `Thread`'s constructor and `start` method:

```
    NewThread() {  
        super("Demo Thread");  
        System.out.println("Child thread: " + this);  
        start();  
    }
```

Example: New Thread 2

NewThread **overrides** the Thread's run method:

```
public void run() {  
    try {  
        for (int i = 5; i > 0; i--) {  
            System.out.println("Child Thread: " + i);  
            Thread.sleep(500);  
        }  
    } catch (InterruptedException e) {  
        System.out.println("Child interrupted.");  
    }  
    System.out.println("Exiting child thread.");  
}
```

Example: New Thread 3

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

After a new thread is created:

```
        new NewThread();
```

the new and main threads continue concurrently...

Example: New Thread 4

This is the loop of the main thread:

```
try {
    for (int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
    }
} catch (InterruptedException e) {
    System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
```


New Thread: Which Approach?

The `Thread` class defines several methods that can be overridden.

Of these methods, only `run` must be overridden.

Creating a new thread:

- 1) implement `Runnable` if only `run` is overridden
- 2) extend `Thread` if other methods are also overridden

Example: Multiple Threads 1

So far, we were using only two threads - main and new, but in fact a program may spawn as many threads as it needs.

`NewThread` class implements the `Runnable` interface:

```
class NewThread implements Runnable {  
    String name;  
    Thread t;  
  
    NewThread(String threadname) {  
        name = threadname;  
        t = new Thread(this, name);  
        System.out.println("New thread: " + t);  
        t.start();  
    }  
}
```

Example: Multiple Threads 2

Here is the implementation of the `run` method:

```
public void run() {  
    try {  
        for (int i = 5; i > 0; i--) {  
            System.out.println(name + ": " + i);  
            Thread.sleep(1000);  
        }  
    } catch (InterruptedException e) {  
        System.out.println(name + "Interrupted");  
    }  
    System.out.println(name + " exiting.");  
}
```

Example: Multiple Threads 3

The demonstration class creates three threads then waits until they all finish:

```
class MultiThreadDemo {  
    public static void main(String args[]) {  
        new NewThread("One");  
        new NewThread("Two");  
        new NewThread("Three");  
        try {  
            Thread.sleep(10000);  
        } catch (InterruptedException e) {  
            System.out.println("Main thread Interrupted");  
        }  
        System.out.println("Main thread exiting.");  
    }  
}
```

Using isAlive and join Methods

How can one thread know when another thread has ended?

Two methods are useful:

- 1) `final boolean isAlive()` - returns `true` if the thread upon which it is called is still running and `false` otherwise
- 2) `final void join()` throws `InterruptedException` – waits until the thread on which it is called terminates

Example: isAlive and join 1

Previous example improved to use `isAlive` and `join` methods.

New thread implements the `Runnable` interface:

```
class NewThread implements Runnable {
    String name;
    Thread t;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start();
    }
}
```

Example: isAlive and join 2

Here is the new thread's run method:

```
public void run() {  
    try {  
        for (int i = 5; i > 0; i--) {  
            System.out.println(name + ": " + i);  
            Thread.sleep(1000);  
        }  
    } catch (InterruptedException e) {  
        System.out.println(name + " interrupted.");  
    }  
    System.out.println(name + " exiting.");  
}
```

Example: isAlive and join 3

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

Creating three new threads:

```
        NewThread ob1 = new NewThread("One");  
        NewThread ob2 = new NewThread("Two");  
        NewThread ob3 = new NewThread("Three");
```

Checking if those threads are still alive:

```
        System.out.println(ob1.t.isAlive());  
        System.out.println(ob2.t.isAlive());  
        System.out.println(ob3.t.isAlive());
```


Example: isAlive and join 4

Waiting until all three threads have finished:

```
try {  
    System.out.println("Waiting to finish.");  
    ob1.t.join();  
    ob2.t.join();  
    ob3.t.join();  
} catch (InterruptedException e) {  
    System.out.println("Main thread Interrupted");  
}
```

Example: isAlive and join 5

Testing again if the new threads are still alive:

```
System.out.println(ob1.t.isAlive());  
System.out.println(ob2.t.isAlive());  
System.out.println(ob3.t.isAlive());  
  
System.out.println("Main thread exiting.");  
}  
}
```

Thread Priorities

Priority is used by the scheduler to decide when each thread should run.

In theory, higher-priority thread gets more CPU than lower-priority thread and threads of equal priority should get equal access to the CPU.

In practice, the amount of CPU time that a thread gets depends on several factors besides its priority.

Setting and Checking Priorities

Setting thread's priority:

```
final void setPriority(int level)
```

where level specifies the new priority setting between:

- 1) `MIN_PRIORITY` (1)
- 2) `MAX_PRIORITY` (10)
- 3) `NORM_PRIORITY` (5)

Obtain the current priority setting:

```
final int getPriority()
```

Example: Priorities 1

A new thread class with `click` and `running` variables:

```
class Clicker implements Runnable {  
    int click = 0;  
    Thread t;  
    private volatile boolean running = true;
```

A new thread is created, its priority initialised:

```
public Clicker(int p) {  
    t = new Thread(this);  
    t.setPriority(p);  
}
```

Example: Priorities 2

When running, `click` is incremented. When stopped, `running` is `false`:

```
public void run() {  
    while (running) {  
        click++;  
    }  
}  
public void stop() {  
    running = false;  
}  
  
public void start() {  
    t.start();  
}  
}
```

Example: Priorities 3

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

The main thread is set at the highest priority, the new threads at two above and two below the normal priority:

```
        Thread.currentThread().  
            setPriority(Thread.MAX_PRIORITY);  
        clicker hi = new clicker(Thread.NORM_PRIORITY + 2);  
        clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
```

Example: Priorities 4

The threads are started and allowed to run for 10 seconds:

```
lo.start();  
hi.start();  
try {  
    Thread.sleep(10000);  
} catch (InterruptedException e) {  
    System.out.println("Main thread interrupted.");  
}
```


Example: Priorities 5

After 10 seconds, both threads are stopped and click variables printed:

```
lo.stop();
hi.stop();
try {
    hi.t.join();
    lo.t.join();
} catch (InterruptedException e) {
    System.out.println("InterruptedException");
}

System.out.println("Low-priority: " + lo.click);
System.out.println("High-priority: " + hi.click);
}
}
```

Volatile Variable

The `volatile` keyword is used to declare the `running` variable:

```
private volatile boolean running = true;
```

This is to ensure that the value of `running` is examined at each iteration of:

```
while (running) {  
    click++;  
}
```

Otherwise, Java is free to optimize the loop in such a way that a local copy of `running` is created. The use of `volatile` prevents this optimization.

Synchronization

When several threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.

This way is called **synchronization**.

Synchronization uses the concept of **monitors**:

- 1) only one thread can enter a monitor at any one time
- 2) other threads have to wait until the thread exits the monitor

Java implements synchronization in two ways: through the **synchronized methods** and through the **synchronized statement**.

Synchronized Method

All objects have their own implicit monitor associated with them.

To enter an object's monitor, call this object's `synchronized` method.

While a thread is inside a monitor, all threads that try to call this or any other `synchronized` method on this object have to wait.

To exit the monitor, it is enough to return from the `synchronized` method.

Consider first an example without synchronization...

Example: No Synchronization 1

The `call` method tries to print the message string inside brackets, pausing the current thread for one second in the middle:

```
class Callme {  
    void call(String msg) {  
        System.out.print "[" + msg);  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
        System.out.println("]");  
    }  
}
```

Example: No Synchronization 2

Caller constructor obtains references to the `Callme` object and `String`, stores them in the `target` and `msg` variables, then creates a new thread:

```
class Caller implements Runnable {  
    String msg;  
    Callme target;  
    Thread t;  
  
    public Caller(Callme targ, String s) {  
        target = targ;  
        msg = s;  
        t = new Thread(this);  
        t.start();  
    }  
}
```

Example: No Synchronization 3

The Caller's `run` method calls the `call` method on the `target` instance of `Calllme`, passing in the `msg` string:

```
public void run() {  
    target.call(msg);  
}  
}
```

Example: No Synchronization 4

`Synch` class creates a single instance of `Callme` and three of `Caller`, each with a message. The `Callme` instance is passed to each `Caller`:

```
class Synch {  
    public static void main(String args[]) {  
        Callme target = new Callme();  
        Caller ob1 = new Caller(target, "Hello");  
        Caller ob2 = new Caller(target, "Synchronized");  
        Caller ob3 = new Caller(target, "World");  
    }  
}
```


Example: No Synchronization 5

Waiting for all three threads to finish:

```
try {
    ob1.t.join();
    ob2.t.join();
    ob3.t.join();
} catch (InterruptedException e) {
    System.out.println("Interrupted");
}
}
```

No Synchronization

Output from the earlier program:

```
[Hello [Synchronized [World]  
]  
]
```

By pausing for one second, the call method allows execution to switch to another thread. A mix-up of the outputs from of the three message strings.

In this program, nothing exists to stop all three threads from calling the same method on the same object at the same time.

Synchronized Method

To fix the earlier program, we must serialize the access to `call`:

```
class Callme {  
    synchronized void call(String msg) {  
        ...  
    }  
}
```

This prevents other threads from entering `call` while another thread is using it. The output result of the program is now as follows:

```
[Hello]  
[Synchronized]  
[World]
```

Synchronized Statement

How to synchronize access to instances of a class that was not designed for multithreading and we have no access to its source code?

Put calls to the methods of this class inside the `synchronized` block:

```
synchronized(object) {  
    ...  
}
```

This ensures that a call to a method that is a member of the `object` occurs only after the current thread has successfully entered the `object`'s monitor.

Example: Synchronized 1

Now the `call` method is not modified by `synchronized`:

```
class Callme {  
    void call(String msg) {  
        System.out.print "[" + msg);  
        try {  
            Thread.sleep(1000);  
        } catch (InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
        System.out.println("]");  
    }  
}
```

Example: Synchronized 2

```
class Caller implements Runnable {  
    String msg;  
    Callme targ;  
    Thread t;  
  
    public Caller(Callme targ, String s) {  
        targ = targ;  
        msg = s;  
        t = new Thread(this);  
        t.start();  
    }  
}
```

Example: Synchronized 3

The `Caller's run` method uses the `synchronized` statement to include the call the `target's call` method:

```
public void run() {  
    synchronized(target) {  
        target.call(msg);  
    }  
}
```

Example: Synchronized 4

```
class Synch1 {  
    public static void main(String args[]) {  
        Callme target = new Callme();  
        Caller ob1 = new Caller(target, "Hello");  
        Caller ob2 = new Caller(target, "Synchronized");  
        Caller ob3 = new Caller(target, "World");  
        try {  
            ob1.t.join();  
            ob2.t.join();  
            ob3.t.join();  
        } catch (InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
    }  
}
```


Inter-Thread Communication

Inter-thread communication relies on three methods in the `Object` class:

- 1) `final void wait()` throws `InterruptedException`
tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls `notify()`.
- 2) `final void notify()`
wakes up the first thread that called `wait()` on the same object
- 3) `final void notifyAll()`
wakes up all the threads that called `wait()` on the same object; the highest-priority thread will run first.

All three must be called from within a `synchronized` context.

Queuing Problem

Consider the classic queuing problem where one thread (producer) is producing some data and another (consumer) is consuming this data:

- 1) producer should not overrun the consumer with data
- 2) consumer should not consume the same data many times

We consider two solutions:

- 1) incorrect with `synchronized` only
- 2) correct with `synchronized` and `wait/notify`

Example: Incorrect Queue 1

The one-place queue class `Q`, with the variable `n` and methods `get` and `put`. Both methods are `synchronized`:

```
class Q {  
    int n;  
  
    synchronized int get() {  
        System.out.println("Got: " + n);  
        return n;  
    }  
    synchronized void put(int n) {  
        this.n = n;  
        System.out.println("Put: " + n);  
    }  
}
```

Example: Incorrect Queue 2

`Producer` creates a thread that keeps producing entries for the queue:

```
class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}
```

Example: Incorrect Queue 3

Consumer creates a thread that keeps consuming entries in the queue:

```
class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}
```

Example: Incorrect Queue 4

The `PC` class first creates a single `Queue` instance `q`, then creates a `Producer` and `Consumer` that share this `q`:

```
class PC {  
    public static void main(String args[]) {  
        Q q = new Q();  
        new Producer(q);  
        new Consumer(q);  
        System.out.println("Press Control-C to stop.");  
    }  
}
```

Why Incorrect?

Here is the output:

```
Put: 1  
Got: 1  
Got: 1  
Put: 2  
Put: 3  
Get: 3  
...
```

Nothing stops the producer from overrunning the consumer, nor the consumer from consuming the same data twice.

Example: Corrected Queue 1

The correct producer-consumer system uses `wait` and `notify` to synchronize the behavior of the producer and consumer.

The queue class introduces the additional `boolean` variable `valueSet` used by the `get` and `put` methods:

```
class Q {  
    int n;  
    boolean valueSet = false;
```


Example: Corrected Queue 2

Inside `get`, `wait` is called to suspend the execution of `Consumer` until `Producer` notifies that some data is ready:

```
synchronized int get() {  
    if (!valueSet)  
        try {  
            wait();  
        }  
    catch (InterruptedException e) {  
        System.out.println("InterruptedException");  
    }  
}
```

Example: Corrected Queue 3

After the data has been obtained, `get` calls `notify` to tell `Producer` that it can put more data on the queue:

```
    System.out.println("Got: " + n);  
    valueSet = false;  
    notify();  
    return n;  
}
```

Example: Corrected Queue 4

Inside `put`, `wait` is called to suspend the execution of `Producer` until `Consumer` has removed the item from the queue:

```
synchronized void put(int n) {  
    if (valueSet)  
        try {  
            wait();  
        } catch (InterruptedException e) {  
            System.out.println("InterruptedException");  
        }  
}
```

Example: Corrected Queue 5

After the next item of data is put in the queue, `put` calls `notify` to tell `Consumer` that it can remove this item:

```
        this.n = n;  
        valueSet = true;  
        System.out.println("Put: " + n);  
        notify();  
    }  
}
```

Example: Corrected Queue 6

`Producer` creates a thread that keeps producing entries for the queue:

```
class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}
```

Example: Corrected Queue 7

`Consumer` creates a thread that keeps consuming entries in the queue:

```
class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}
```

Example: Corrected Queue 8

The `PCFixed` class first creates a single `Queue` instance `q`, then creates a `Producer` and `Consumer` that share this `q`:

```
class PCFixed {  
    public static void main(String args[]) {  
        Q q = new Q();  
        new Producer(q);  
        new Consumer(q);  
        System.out.println("Press Control-C to stop.");  
    }  
}
```

Deadlock

Multi-threading and synchronization create the danger of deadlock.

Deadlock: a circular dependency on a pair of synchronized objects.

Suppose that:

- 1) one thread enters the monitor on object X
- 2) another thread enters the monitor on object Y
- 3) the first thread tries to call a synchronized method on object Y
- 4) the second thread tries to call a synchronized method on object X

The result: the threads wait forever – deadlock.

Example: Deadlock 1

Class `A` contains the `foo` method which takes an instance `b` of class `B` as a parameter. It pauses briefly before trying to call the `b`'s `last` method:

```
class A {  
    synchronized void foo(B b) {  
        String name = Thread.currentThread().getName();  
        System.out.println(name + " entered A.foo");  
        try {  
            Thread.sleep(1000);  
        } catch (Exception e) {  
            System.out.println("A Interrupted");  
        }  
        System.out.println(name + " trying B.last()");  
        b.last();  
    }  
}
```

Example: Deadlock 2

Class `A` also contains the `synchronized` method `last`:

```
synchronized void last() {  
    System.out.println("Inside A.last");  
}  
}
```

Example: Deadlock 3

Class `B` contains the `bar` method which takes an instance `a` of class `A` as a parameter. It pauses briefly before trying to call the `a`'s `last` method:

```
class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");
        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            System.out.println("B Interrupted");
        }
        System.out.println(name + " trying A.last()");
        a.last();
    }
}
```

Example: Deadlock 4

Class B also contains the `synchronized` method `last`:

```
synchronized void last() {  
    System.out.println("Inside A.last");  
}  
}
```

Example: Deadlock 5

The main `Deadlock` class creates the instances `a` of `A` and `b` of `B`:

```
class Deadlock implements Runnable {  
  
    A a = new A();  
    B b = new B();
```

Example: Deadlock 6

The constructor creates and starts a new thread, and creates a lock on the `a` object in the `main` thread (running `foo` on `a`) with `b` passed as a parameter:

```
Deadlock() {  
    Thread.currentThread().setName("MainThread");  
    Thread t = new Thread(this, "RacingThread");  
    t.start();  
    a.foo(b);  
    System.out.println("Back in main thread");  
}
```

Example: Deadlock 7

The run method creates a lock on the `b` object in the new thread (running `bar` on `b`) with `a` passed as a parameter:

```
public void run() {  
    b.bar(a);  
    System.out.println("Back in other thread");  
}
```

Create a new `Deadlock` instance:

```
public static void main(String args[]) {  
    new Deadlock();  
}  
}
```

Deadlock Reached

Program output:

```
MainThread entered A.foo
```

```
RacingThread entered B.bar
```

```
MainThread trying to call B.last()
```

```
RacingThread trying to call A.last()
```

`RacingThread` owns the monitor on `b` while waiting for the monitor on `a`.

`MainThread` owns the monitor on `a` while it is waiting for the monitor on `b`.

The program deadlocks!

Suspending/Resuming Threads

Thread management should use the `run` method to check periodically whether the thread should suspend, resume or stop its own execution.

This is usually accomplished through a flag variable that indicates the execution state of a thread, e.g.

- 1) `running` – the thread should continue executing
- 2) `suspend` – the thread must pause
- 3) `stop` – the thread must terminate

Example: Suspending/Resuming 1

`NewThread` class contains the `boolean` variable `suspendFlag` to control the execution of a thread, initialized to `false`:

```
class NewThread implements Runnable {
    String name;
    Thread t;
    boolean suspendFlag;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        suspendFlag = false;
        t.start();
    }
}
```

Example: Suspending/Resuming 2

The `run` method contains the `synchronized` statement that checks `suspendFlag`. If `true`, the `wait` method is called.

```
public void run() {  
    try {  
        for (int i = 15; i > 0; i--) {  
            System.out.println(name + ": " + i);  
            Thread.sleep(200);  
            synchronized(this) {  
                while(suspendFlag)    wait();  
            }  
        }  
    }  
}
```

Example: Suspending/Resuming 3

```
        catch (InterruptedException e) {  
            System.out.println(name + " interrupted.");  
        }  
        System.out.println(name + " exiting.");  
    }
```

Example: Suspending/Resuming 4

The `mysuspend` method sets `suspendFlag` to true:

```
void mysuspend() {  
    suspendFlag = true;  
}
```

The `myresume` method sets `suspendFlag` to false and invokes `notify` to wake up the thread:

```
synchronized void myresume() {  
    suspendFlag = false;  
    notify();  
}  
}
```

Example: Suspending/Resuming 5

SuspendResume class creates two instances `ob1` and `ob2` of `NewThread`, therefore two new threads, through its `main` method:

```
class SuspendResume {  
  
    public static void main(String args[]) {  
        NewThread ob1 = new NewThread("One");  
        NewThread ob2 = new NewThread("Two");  
    }  
}
```

The two threads are kept running, then suspended, then resumed from the main thread:

Example: Suspending/Resuming 6

```
try {
    Thread.sleep(1000);
    ob1.mysuspend();
    System.out.println("Suspending thread One");
    Thread.sleep(1000);
    ob1.myresume();
    System.out.println("Resuming thread One");
    ob2.mysuspend();
    System.out.println("Suspending thread Two");
    Thread.sleep(1000);
    ob2.myresume();
    System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
}
```

Example: Suspending/Resuming 7

The main thread waits for the two child threads to finish, then finishes itself:

```
try {
    System.out.println("Waiting to finish.");
    ob1.t.join();
    ob2.t.join();
} catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
```


The Last Word on Multi-Threading

Multi-threading is a powerful tool to writing efficient programs.

When you have two subsystems within a program that can execute concurrently, make them individual threads.

However, creating too many threads can actually degrade the performance of your program because of the cost of context switching.

Exercise: Multi-Threading 1

- 1) Create a new main class called `MultiThread`
- 2) Create a new class called `MemoryThread`. This class should implement the interface `Runnable` so that it can be run as a thread. This Thread will monitor the memory usage on the system.
 - a) Add `void start()`, `stop()` and `run()` methods.
 - b) The `run()` method should print to the screen every 5 seconds the amount of memory presently being used. This can be done using these two lines of code:

```
Runtime r = Runtime.getRuntime();  
long memoryUsed = r.totalMemory() - r.freeMemory();
```
- 3) Create a new class called `ClockThread`. This class should implement the interface `Runnable` so that it can be run as a thread. This Thread will monitor what the time is.
 - a) The constructor for `ClockThread` should take an argument that sets the time (in seconds) that this thread will run for.
 - b) Add `void start()`, `stop()` and `run()` methods.

Exercise: Multi-Threading 2

- c) The `run()` method should print the current time to the screen every 5 seconds. This can be done using the built in `Date` class.

```
import java.util.Date;  
Date timeNow = new Date();  
System.out.println(timeNow.toString());
```

The `run()` method should stop when the elapsed time set in the constructor has been reached.

- 4) In the `main()` method of `MultiThread` create the two thread objects and start them; using the `start()` method.
- 5) Add a `while` loop, that will print the number of active Threads every one second. A one second pause can be implemented as follows:

```
while (true) {  
    Thread.sleep(1000);  
}
```

- 6) The number of threads can be monitored using the `ThreadGroup` class as follows:

```
int numThreads =  
Thread.currentThread().getThreadGroup().activeCount();
```

Exercise: Multi-Threading 3

- 7) This while loop should terminate after one minute and the two threads stopped by invoking their `stop()` methods.
- 8) When printing to the screen - the Threads names should be appended so that it is clear from which process the data is originating.
- 9) The `MemoryThread` thread should be assigned a maximum priority and the `ClockThread` thread a minimum priority.

Horizontal Libraries

Course Outline

- | | | |
|---|--|--|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|--|--|

Horizontal Libraries

Horizontal libraries are APIs that are used across the language.

Java provides a rich set of horizontal libraries:

- a) **String handling** – for handling sequence of characters
- b) **event handling** – helps to handle how programs respond to actions generated by the user.
- c) **Object collection** – handling a group of objects

String Handling

Course Outline

- | | | |
|---|--|--|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|--|--|

String Object

String is a sequence of characters.

Unlike many other programming languages that implements string as character arrays, Java implements strings as object of type `String`.

This provides a full compliment of features that make string handling convenient. For example, Java String has methods to:

- 1) compare two strings
- 2) Search for a substring
- 3) Concatenate two strings and
- 4) Change the case of letters within a string
- 5) Can be constructed a number of ways making it easy to obtain a string when needed

String is Immutable

Once a String Object has been created, you cannot change the characters that comprise that string.

This is not a restriction. It means each time you need an altered version of an existing string, a new string object is created that contains the modification.

It is more efficient to implement immutable strings than changeable ones.

To solve this, Java provides a companion class to `String` called `StringBuffer`.

`StringBuffer` objects can be modified after they are created.

String Constructors 1

String supports several constructors:

- 1) to create an empty String

```
String s = new String();
```

- 2) to create a string that have initial values

```
String(char chars[])
```

Example:

```
char chars[] = { 'a', 'b', 'c' };  
String s = new String(chars);
```

String Constructors 2

- 3) to create a string as a subrange of a character array

```
String(char chars[], int startindex, int numchars)
```

Here, `startindex` specifies the index at which the subrange begins, and `numChars` specifies the number of characters to use.

Example:

```
char chars[] = {'a','b','c','d','e','f'};  
String s = new String(chars,2,3);
```

This initializes `s` with the characters `cde`.

String Constructors 3

- 4) to construct a String object that contains the same character sequence as another String object

```
String(String obj)
```

Example

```
class MakeString {  
    public static void main(String args[]) {  
        char c[] = {'J', 'a', 'v', 'a'};  
        String s1 = new String(c);  
        String s2 = new String(s1);  
        System.out.println(s1);  
        System.out.println(s2);  
    }  
}
```

String Length

The length of a string is the number of characters that it contains.

To obtain this value call the `length()` method:

```
int length()
```

The following fragment prints “3”, since there are three characters in the string `s`.

```
char chars[] = {'a','b','c'};  
String s = new String(chars);  
System.out.println(s.length());
```

String Operations

Strings are a common and important part of programming.

Java provides several string operations within the syntax of the language.

These operations include:

- 1) automatic creation of new `String` instances from literals
- 2) concatenation of multiple `String` objects using the `+` operator
- 3) conversion of other data types to a string representation

There are explicit methods to perform all these functions, but Java does them automatically for the convenience of the programmer and to add clarity.

String Literals

Using String literals is an easier way of creating Strings Objects.

For each String literal, Java automatically constructs a `String` object.

You can use String literal to initialize a `String` object.

Example:

```
char chars[] = {'a','b','c'};  
String s1 = new String(chars);
```

Using String literals

```
String s2 = "abc";
```

String Concatenation

Java does not allow operations to be applied to a `String` object.

The one exception to this rule is the `+` operator, which concatenates two strings producing a string object as a result.

With this you can chain together a series of `+` operations.

Example:

```
String age = "9";  
String s = "He is " + age + " years old.";  
System.out.println(s);
```

Concatenation Usage

One practical use is found when you are creating very long strings.

Instead of letting long strings wrap around your source code, you can break them into smaller pieces, using the `+` to concatenate them.

Example:

```
class ConCat {  
    public static void main(String args[]) {  
        String longStr = "This could have been " +  
            "a very long line that would have " +  
            "wrapped around.  But string concatenation "  
            + "prevents this.";   
        System.out.println(longStr);  
    }  
}
```

Concatenation & Other Data Type

You can concatenate Strings with other data types.

Example:

```
int age = 9;  
String s = "He is " + age + " years old.";  
System.out.println(s);
```

The compiler will convert an operand to its string equivalent whenever the other operand of the + is an instance of `String`.

Be careful:

```
String s = "Four:" + 2 + 2;  
System.out.println(s);
```

Prints `Four:22` rather than `Four: 4`.

To achieve the desired result, bracket has to be used.

```
String s = "Four:" + (2 + 2);
```

Conversion and toString() Method

When Java converts data into its string representation during concatenation, it does so by calling one of its overloaded `valueOf()` method defined by `String`.

`valueOf()` is overloaded for

- 1) **simple types** – which returns a string that contains the human – readable equivalent of the value with which it is called.
- 2) **object types** – which calls the `toString()` method of the object.

Example: toString() Method 1

Override toString() for Box class

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
}
```

Example: toString() Method 2

```
public String toString() {  
    return "Dimensions are " + width + " by " +  
        depth + " by " + height + ".";  
}  
}
```

Example: toString() Method 3

```
class toStringDemo {  
    public static void main(String args[]) {  
        Box b = new Box(10, 12, 14);  
        String s = "Box b: " + b; // concatenate Box object  
  
        System.out.println(b); // convert Box to string  
        System.out.println(s);  
    }  
}
```

Box's `toString()` method is automatically invoked when a `Box` object is used in a concatenation expression or in a Call to `println()`.

Character Extraction

String class provides a number of ways in which characters can be extracted from a String object.

String index begin at zero.

These extraction methods are:

- 1) `charAt()`
- 2) `getChars()`
- 3) `getBytes()`
- 4) `toCharArray()`

Each will considered.

charAt()

To extract a single character from a String.

General form:

```
char charAt(int where)
```

`where` is the index of the character you want to obtain. The value of `where` must be nonnegative and specify a location within the string.

Example:

```
char ch;  
ch = "abc".charAt(1);
```

Assigns a value of "b" to `ch`.

getChars()

Used to extract more than one character at a time.

General form:

```
void getChars(int sourceStart, int sourceEnd, char[]  
              target, int targetStart)
```

`sourceStart` – specifies the index of the beginning of the substring

`sourceEnd` – specifies an index that is one past the end of the desired substring

`target` – is the array that will receive the characters

`targetStart` – is the index within target at which the substring will be copied is passed in this parameter

getChars()

```
class getCharsDemo {  
    public static void main(String args[]) {  
        String s = "This is a demo of the getChars method.";  
        int start = 10;  
        int end = 14;  
        char buf[] = new char[end - start];  
        s.getChars(start, end, buf, 0);  
        System.out.println(buf);  
    }  
}
```

getBytes()

Alternative to getChars() that stores the characters in an array of bytes. It uses the default character-to-byte conversions provided by the platform.

General form:

```
byte[] getBytes()
```

Usage:

Most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters.

For example, most internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray()

To convert all the characters in a String object into character array.

It returns an array of characters for the entire string.

General form:

```
char[] toCharArray()
```

It is provided as a convenience, since it is possible to use `getChars()` to achieve the same result.

String Comparison

The String class includes several methods that compare strings or substrings within strings.

They are:

- 1) `equals()` and `equalsIgnoreCase()`
- 2) `regionMatches()`
- 3) `startsWith()` and `endsWith()`
- 4) `equals()` Versus `==`
- 5) `compareTo()`

Each will be considered.

equals()

To compare two Strings for equality, use equals()

General form:

```
boolean equals(Object str)
```

`str` is the `String` object being compared with the invoking String object.

It returns true if the string contain the same character in the same order, and false otherwise.

The comparison is case-sensitive.

equalsIgnoreCase()

To perform operations that ignores case differences.

When it compares two strings, it considers `A-Z` as the same as `a-z`.

General form:

```
boolean equalsIgnoreCase(Object str)
```

`str` is the `String` object being compared with the invoking `String` object.

It returns `true` if the string contain the same character in the same order, and `false` otherwise.

The comparison is case-sensitive.

equals and equalsIgnoreCase() 2

```
System.out.println(s1 + " equals " + s4 + " -> " +  
                    s1.equals(s4));  
  
System.out.println(s1 + " equalsIgnoreCase " + s4 +  
                    " -> " + s1.equalsIgnoreCase(s4));  
  
}  
  
}
```

regionMatches() 1

Compares a specific region inside a string with another specific region in another string.

There is an overloaded form that allows you to ignore case in such comparison.

General form:

```
boolean regionMatches(int startindex, String str2,  
                      int str2StartIndex, int numChars)
```

```
boolean regionMatches(boolean ignoreCase, int  
                      startindex, String str2, int str2StartIndex,  
                      int numChars)
```

regionMatches() 2

In both versions, `startIndex` specifies the index at which the region begins within the invoking String object.

The string object being compared is specified as `str`.

The index at which the comparison will start within `str2` is specified by `str2StartIndex`.

The length of the substring being compared is passed in `numChars`.

In the second version, if the `ignoreCase` is `true`, the case of the characters is ignored. Otherwise case is significant.

startsWith() and endsWith() 1

`String` defines two routines that are more or less the specialised forms of `regionMatches()`.

The `startsWith()` method determines whether a given string begins with a specified string.

Conversely, `endsWith()` method determines whether the string in question ends with a specified string.

General form:

```
boolean startsWith(String str)
```

```
boolean endsWith(String str)
```

`str` is the `String` being tested. If the string matches, `true` is returned, otherwise `false` is returned.

startsWith() and endsWith() 2

Example:

```
"Foobar".endsWith("bar");
```

and

```
"Foobar".startsWith("Foo");
```

are both `true`.

startsWith() and endsWith() 3

A second form of `startsWith()`, let you specify a starting point:

General form:

```
boolean startWith(String str, int startIndex)
```

Where `startIndex` specifies the index into the invoking string at which point the search will begin.

Example:

```
"Foobar".startsWith("bar", 3);
```

returns `true`.

equals() Versus ==

It is important to understand that the two methods perform different functions.

- 1) `equals()` method compares the characters inside a `String` object.
- 2) `==` operator compares two object references to see whether they refer to the same instance.

Example: equals() Versus ==

```
class EqualsNotEqualTo {  
    public static void main(String args[]) {  
        String s1 = "Hello";  
        String s2 = new String(s1);  
        System.out.print(s1 + " equals " + s2 + " -> ");  
        System.out.println(s1.equals(s2));  
        System.out.print(s1 + " == " + s2 + " -> ")  
        System.out.println((s1 == s2));  
    }  
}
```

compareTo() 1

It is not enough to know that two Strings are identical. You need to know which is **less than**, **equal to**, or **greater than** the next.

A string is less than the another if it comes before the other in the dictionary order.

A string is greater than the another if it comes after the other in the dictionary order.

The `String` method `compareTo()` serves this purpose.

compareTo() 2

General form:

```
int compareTo(String str)
```

`str` is the string that is being compared with the invoking String. The result of the comparison is returned and is interpreted as shown here:

Less than zero	The invoking string is less than <code>str</code>
Greater than zero	The invoking string is greater than <code>str</code>
Zero	The two strings are equal

Example: compareTo()

```
class SortString {  
    static String arr[] =  
        {"Now", "is", "the", "time", "for", "all", "good,"  
"men", "to", "come", "to", "the", "aid", "of", "their",  
"country"};  
  
    public static void main(String args[]) {  
        for(int j = 0; j < arr.length; j++) {  
            for(int i = j + 1; i < arr.length;  
i++) {  
  
                if(arr[i].compareTo(arr[j]) < 0)  
  
                {  
  
                    String t = arr[j];  
                    arr[j] = arr[i];  
                    arr[i] = t;  
                }  
            }  
        }  
    }  
}
```

Searching String 1

String class provides two methods that allows you search a string for a specified character or substring:

- 1) `indexOf()` – Searches for the first occurrence of a character or substring.
- 2) `lastIndexOf()` – Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or `-1` on failure.

Searching String 2

To search for the first occurrence of a character, use

```
int indexOf(int ch)
```

To search for the last occurrence of a character, use

```
int lastIndexOf(int ch)
```

To search for the first and the last occurrence of a substring, use

```
int indexOf(String str)
```

```
int lastIndexOf(String str)
```

Here `str` specifies the substring.

Searching String 3

You can specify a starting point for the search using these forms:

```
int indexOf(int ch, int startIndex)
```

```
int lastIndexOf(int ch, int startIndex)
```

```
int indexOf(String str, int startIndex)
```

```
int lastIndexOf(String str, int startIndex)
```

`startIndex` – specifies the index at which point the search begins.

For `indexOf()`, the search runs from `startIndex` to the end of the string.

For `lastIndexOf()`, the search runs from `startIndex` to zero.

Example: Searching String 2

```
System.out.println("indexOf(t, 10) = " +
                   s.indexOf('t', 10));
System.out.println("lastIndexOf(t, 60) = " +
                   s.lastIndexOf('t', 60));
System.out.println("indexOf(the, 10) = " +
                   s.indexOf("the", 10));
System.out.println("lastIndexOf(the, 60) = " +
                   s.lastIndexOf("the", 60));
}
}
```

Modifying a String

String object are immutable.

Whenever you want to modify a String, you must either copy it into a StringBuffer or use the following String methods,, which will construct a new copy of the string with your modification complete.

They are:

- 1) `subString()`
- 2) `concat()`
- 3) `replace()`
- 4) `trim()`

Each will be discussed.

substring() 1

You can extract a substring using `substring()`.

It has two forms:

```
String substring(int startIndex)
```

`startIndex` specifies the index at which the substring will begin. This form returns a copy of the substring that begins at `startIndex` and runs to the end of the invoking string.

substring() 2

The second form allows you to specify both the beginning and ending index of the substring.

```
String substring(int startIndex, int endIndex)
```

`startIndex` specifies the index beginning index, and

`endIndex` specifies the stopping point.

The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

Example: substring()

```
class StringReplace {  
    public static void main(String args[]) {  
        String org = "This is a test. This is, too.";  
        String search = "is";  
        String sub = "was";  
        String result = "";  
        int i;  
        do { // replace all matching substrings  
            System.out.println(org);  
            i = org.indexOf(search);  
            if(i != -1) {  
                result = org.substring(0, i);
```

Example: substring()

```
        result = result + sub;
        result = result + org.substring(i +
                                         search.length());

    org = result;
}
} while(i != -1);
}
}
```

concat()

You can concatenate two string using concat()

General form:

```
String concat (String str)
```

This method creates a new object that contains the invoking string with the contents of `str` appended to the end.

`concat()` performs the same function as `+`.

Example:

```
String s1 ="one";  
String s2 = s1.concat("two");
```

Or

```
String s2 = s1 + "two";
```


replace()

Replaces all occurrences of one character in the invoking string with another character.

General form:

```
String replace(char original, char replacement)
```

`original` – specifies the character to be replaced by the character specified by `replacement`. The resulting string is returned.

Example:

```
String s = "Hello".replace('l','w');
```

Puts the string "Hewwo" into `s`.

trim()

Returns a copy of the involving string from which any leading and trailing whitespace has been removed.

General form:

```
String trim();
```

Example:

```
String s = "    Hello world    ".trim();
```

This puts the string `"Hello world"` into `s`.

It is quite useful when you process user commands.

Example: trim() 1

```
import java.io.*;

class UseTrim {

    public static void main(String args[]) throws

                                IOException{

        BufferedReader br = new BufferedReader(new

            InputStreamReader(System.in));

        String str;

        System.out.println("Enter 'stop' to quit.");

        System.out.println("Enter State: ");

        do {

            str = br.readLine();
```

Example: trim() 2

```
str = str.trim(); // remove whitespace
if(str.equals("Illinois"))
    System.out.println("Capital is pringfield.");
else if(str.equals("Missouri"))
    System.out.println("Capital is Jefferson
                        City.");
else if(str.equals("California"))
    System.out.println("Capital is Sacramento.");
else if(str.equals("Washington"))
    System.out.println("Capital is Olympia.");
} while(!str.equals("stop"));
}
}
```

Data Conversion Using `valueOf()`

Converts data from its internal format into human-readable form.

It is a static method that is overloaded within `String` for all of Java's built-in types, so that each of the type can be converted properly into a `String`.

`valueOf()` can be overloaded for type `Object` so an object of any class type you create can also be used as an argument.

General forms:

```
static String valueOf(double num)
```

```
static String valueOf(long num)
```

```
static String valueOf(Object obj)
```

```
static String valueOf(char chars[])
```

Case of Characters

The method `toLowerCase()` converts all the characters in a string from **uppercase** to **lowercase**.

The `toUpperCase()` method converts all the characters in a string from lowercase to uppercase.

Non-alphabetical characters, such as **digits** are **unaffected**.

General form:

```
String toLowerCase()
```

```
String toUpperCase()
```

Example: Case of Characters

```
class ChangeCase {  
    public static void main(String args[]) {  
        String s = "This is a test.";  
        System.out.println("Original: " + s);  
        String upper = s.toUpperCase();  
        String lower = s.toLowerCase();  
        System.out.println("Uppercase: " + upper);  
        System.out.println("Lowercase: " + lower);  
    }  
}
```

StringBuffer

`StringBuffer` is a peer class of `String` that provides much of the functionality of `String`s.

`String` is immutable. `StringBuffer` represents growable and writable character sequence.

`StringBuffer` may have characters and substring inserted in the middle or appended to the end.

`StringBuffer` will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

Defines three constructors:

- 1) `StringBuffer()` – default and reserves room for 16 characters without reallocation
- 2) `StringBuffer(int size)` – accepts an integer argument that explicitly sets the size of the buffer
- 3) `StringBuffer(String str)` – accepts a `String` argument that initially sets the content of the `StringBuffer` Object and reserves room for more 16 characters without reallocation.

Length() and capacity()

Current length of a `StringBuffer` can be found via the `length()` method, while the total allocated capacity can be found through the `capacity()` method.

General form:

```
int length()
```

```
Int capacity()
```

Example: Length() and capacity()

```
class StringBufferDemo {  
    public static void main(String args[]) {  
        StringBuffer sb = new StringBuffer("Hello");  
  
        System.out.println("buffer = " + sb);  
        System.out.println("length = " + sb.length());  
        System.out.println("capacity = " + sb.capacity());  
    }  
}
```

ensureCapacity()

Use `ensureCapacity()` to set the size of the buffer in order to preallocate room for a certain number of characters after a `StringBuffer` has been constructed.

General form:

```
void ensureCapacity(int capacity)
```

Here, `capacity` specifies the size of the buffer.

Usage:

Useful if you know in advance that you will be appending a large number of small strings to a `StringBuffer`.

setLength()

To set the length of the buffer within a `StringBuffer` object.

General form:

```
void setLength(int len)
```

Here, `len` specifies the length of the buffer.

Usage:

When you increase the length of the buffer, null characters are added to the end of the existing buffer. If you call `setLength()` with a value less than the current value returned by `length()`, then the characters stored beyond the new length will be lost.

charAt() and setCharAt()

To obtain the value of a single character, use `charAt()`.

To set the value of a character within `StringBuffer`, use `setCharAt()`.

General form:

```
char charAt(int where)
```

```
void setCharAt(int where, char ch)
```

For `charAt()`, `where` specifies the index of the characters being obtained.

For `setCharAt()`, `where` specifies the index of the characters being set, and `ch` specifies the new value of that character.

`where` must be non negative and must not specify a location beyond the end of the buffer.

Example:charAt() and setCharAt()

```
class setCharAtDemo {  
    public static void main(String args[]) {  
        StringBuffer sb = new StringBuffer("Hello");  
        System.out.println("buffer before = " + sb);  
        System.out.println("charAt(1) before = " +  
                               sb.charAt(1));  
  
        sb.setCharAt(1, 'i');  
        sb.setLength(2);  
        System.out.println("buffer after = " + sb);  
        System.out.println("charAt(1) after = " +  
sb.charAt(1));  
    }  
}
```

getChars()

To copy a substring of a StringBuffer into an array.

General form:

```
void getChars(int srcBegin, int srcEnd, char[] dst,  
              int dstBegin)
```

Where :

`srcBegin` - start copying at this offset.

`srcEnd` - stop copying at this offset.

`dst` - the array to copy the data into.

`dstBegin` - offset into `dst`.

append()

Concatenates the string representation of any other type of data to the end of the invoking `StringBuffer` object.

General form:

```
StringBuffer append(Object obj)
```

```
StringBuffer append(String str)
```

```
StringBuffer append(int num)
```

`String.valueOf()` is called for each parameter to obtain its string representation. The result is appended to the current `StringBuffer` object.

Example: append()

```
class appendDemo {  
    public static void main(String args[]) {  
        String s;  
        int a = 42;  
        StringBuffer sb = new StringBuffer(40);  
        s = sb.append("a =  
  
        ").append(a).append("!").toString();  
        System.out.println(s);  
    }  
}
```

insert()

Inserts one string into another. It is overloaded to accept values of all the simple types, plus String and Objects.

General form:

```
StringBuffer insert(int index, String str)
```

```
StringBuffer insert(int index, char ch)
```

```
StringBuffer insert(int index, Object obj)
```

Here, `index` specifies the index at which point the String will be inserted into the invoking StringBuffer object.

Example: insert()

```
class insertDemo {  
    public static void main(String args[]) {  
        StringBuffer sb = new StringBuffer("I Java!");  
  
        sb.insert(2, "like ");  
        System.out.println(sb);  
    }  
}
```

reverse()

To reverse the character within a StringBuffer object.

General form:

```
StringBuffer reverse()
```

This method returns the reversed on which it was called.

For example:

```
class ReverseDemo {  
    public static void main(String args[]) {  
        StringBuffer s = new StringBuffer("abcdef");  
        System.out.println(s);  
        s.reverse();  
        System.out.println(s);  
    }  
}
```

replace()

Replaces one set of characters with another set inside a `StringBuffer` object.

General form:

```
StringBuffer replace(int startIndex, String endIndex,  
                    String str)
```

The substring being replaced is specified by the indexes `startIndex` and `endIndex`. Thus, the substring at `startIndex` through `endIndex-1` is replaced. The replacement string is passed in `str`. The resulting `StringBuffer` object is returned.

Example: replace()

```
class replaceDemo {  
    public static void main(String args[]) {  
        StringBuffer sb = new StringBuffer("This is a  
  
        test.");  
        sb.replace(5, 7, "was");  
        System.out.println("After replace: " + sb);  
    }  
}
```

substring()

Returns a portion of a StringBuffer.

General form:

```
String substring(int startIndex)
```

```
String substring(int startIndex, int endIndex)
```

The first form returns the substring that starts at `startIndex` and runs to the end of the invoking `StringBuffer` object.

The second form returns the substring that starts at `startIndex` and runs through `endIndex-1`.

These methods work just like those defined for `String` that were described earlier.

Exercise: String Handling

- 1.) Write a program that computes your initials from your full name and displays them.
- 2.) Write a program to test if a word is a palindrome.
- 3.) Write a program to read English text to end-of-data, and print a count of word lengths, i.e. the total number of words of length 1 which occurred, the number of length 2, and so on.

Type in question 3 as input to test your program.

- 4.) An anagram is a word or a phrase made by transposing the letters of another word or phrase; for example, "parliament" is an anagram of "partial men," and "software" is an anagram of "swear oft." Write a program that figures out whether one string is an anagram of another string. The program should ignore white space and punctuation.

Event Handling

Course Outline

- | | | |
|---|--|--|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|--|--|

Event Handling

For the user to interact with a GUI, the underlying operating system must support event handling.

- 1) operating systems constantly monitor events such as keystrokes, mouse clicks, ink input, voice command, etc.
- 2) operating systems sort out these events and report them to the appropriate application programs
- 3) each application program then decides what to do in response to these events

Complexity vs. Power

Many programming languages have their ways of implementing events:

1) Visual Basic

- a) Each component responds to a fixed set of events

2) C

- a) A giant loop with a massive switch statement

3) Java

- a) Event delegation model – events are transmitted from **event sources** to **event listeners**
- b) You can designate any object to be an **event listener**

Event Handling Components 1

Event handling involves three components:

- 1) **listener object** is an instance of a class that implements a special interface called a listener interface.
- 2) **event source** is an object that can **register** listener objects and send them event objects
- 3) **event source** sends out **event objects** to all registered listeners when that event occurs.

The listener objects reacts based on the information in the event.

Event Handling Process 1

Register listener object with the event source object

```
eventSourceObject.addEventLisTener (eventListenerObject) ;
```

For example

```
ActionListener listener = ...;  
JButton button = new JButton("OK");  
button.addActionListener(listener);
```

The listener object is notified whenever an action event occurs in the button (i.e., a button click)

Event Handling Process 2

The class to which the listener object belongs should implement the appropriate interface (in this case, the `ActionListener` interface)

Listener class must have a method `actionPerformed` that receives an `ActionEvent` object as a parameter

For Example:

```
class MyListener implements ActionListener {  
    ...  
    public void actionPerformed(ActionEvent event)  
{  
        // reaction to button click goes here  
        ...  
    }  
}
```


Event Handling Process 3

Whenever the user clicks the button,

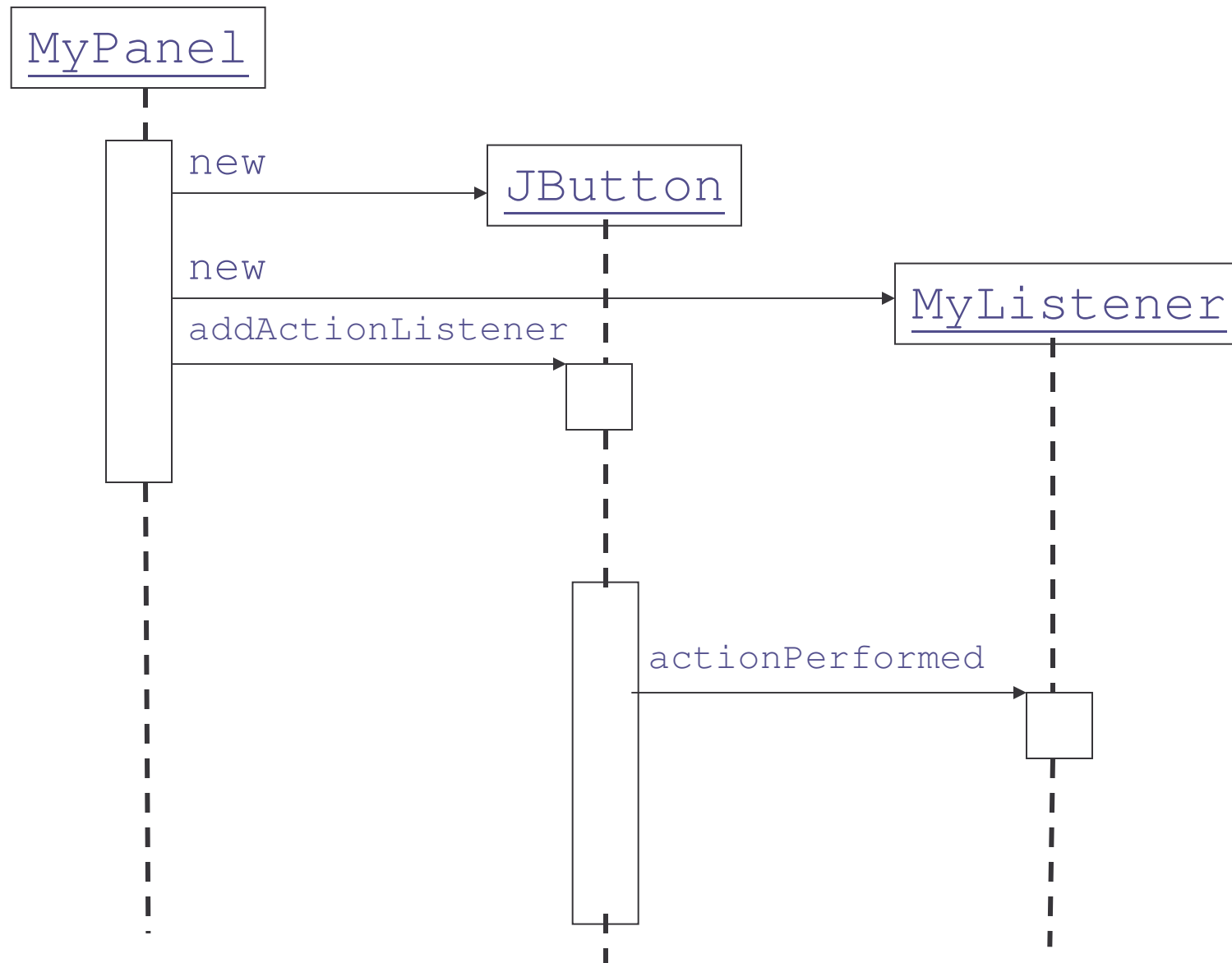
the `JButton` object creates an `ActionEvent` object and

calls `listener.actionPerformed(event)`, passing that event object

It is possible for multiple objects to be added as listeners to an event source by using the `addActionListener(listener)`.

The `actionPerformed` methods of all listeners will be called.

ActionEvent Handling Example



Event and Listener Objects

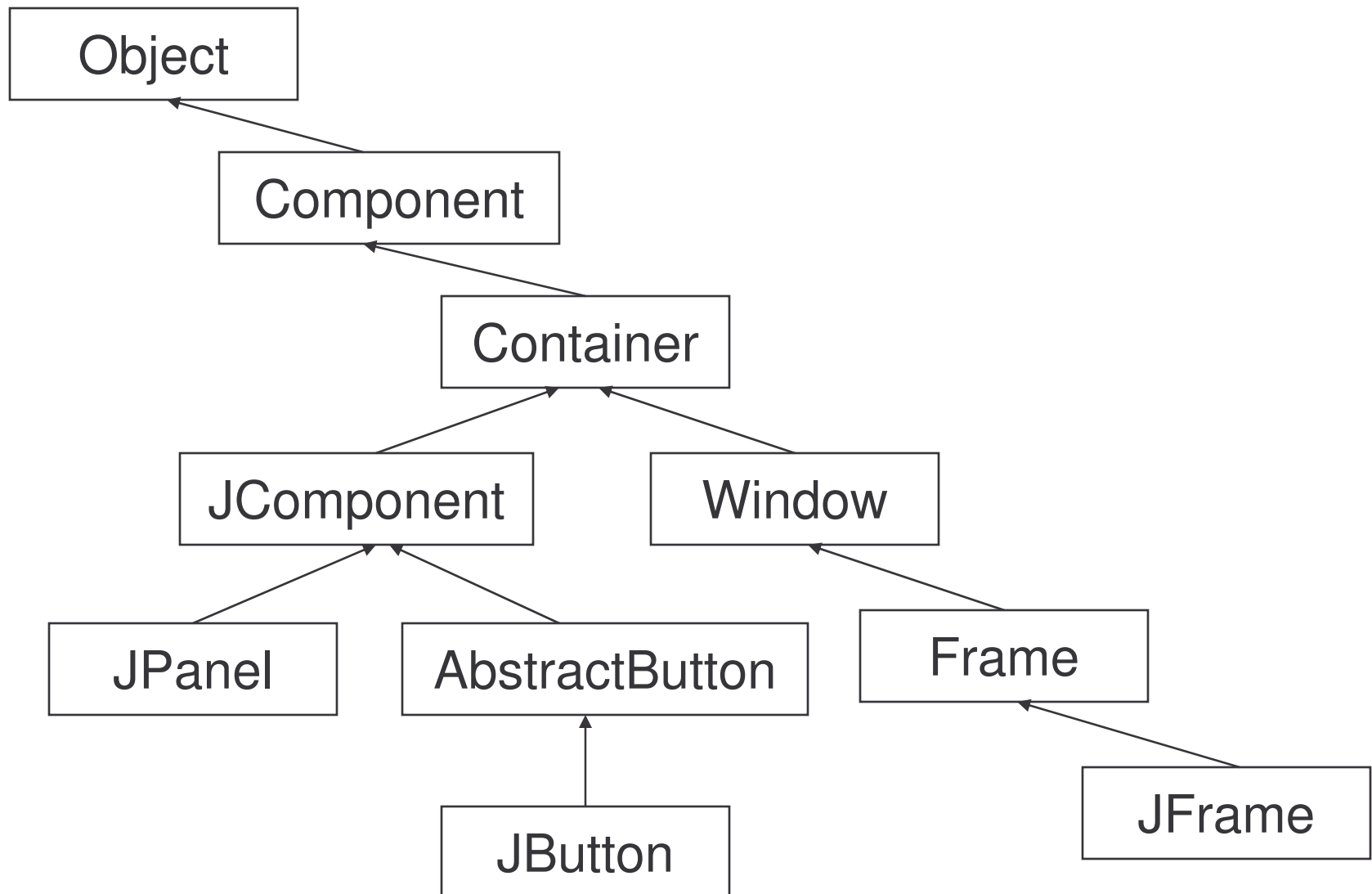
Different event sources can produce different kinds of events

- a) `ActionEvent` – sent by `JButton` click and other components
- b) `WindowEvent` – sent by `JFrame`

Different listener objects implement different required methods from interfaces

- 1) `ActionListener` – action performed
- 2) `WindowListener` – activated, closed, closing, deactivated, deiconified, iconified, opened

Example: JButton



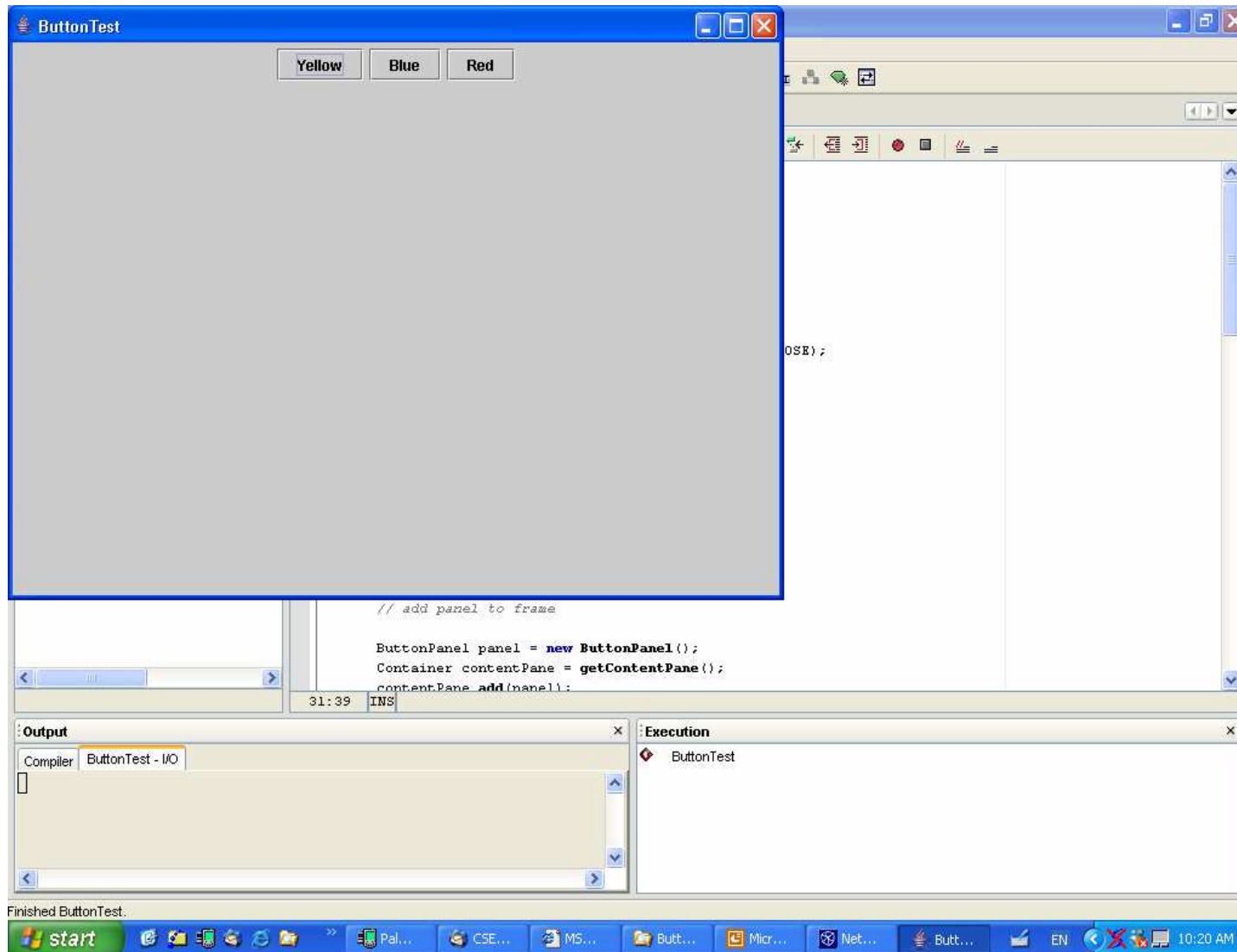
Example: JButton Event 1

```
class ButtonPanel extends JPanel {  
    public ButtonPanel( ) {  
        JButton yellowButton = new JButton("Yellow");  
        JButton blueButton = new JButton("Blue");  
        JButton redButton = new JButton("Red");  
        add(yellowButton);  
        add(blueButton);  
        add(redButton);  
    }  
}
```

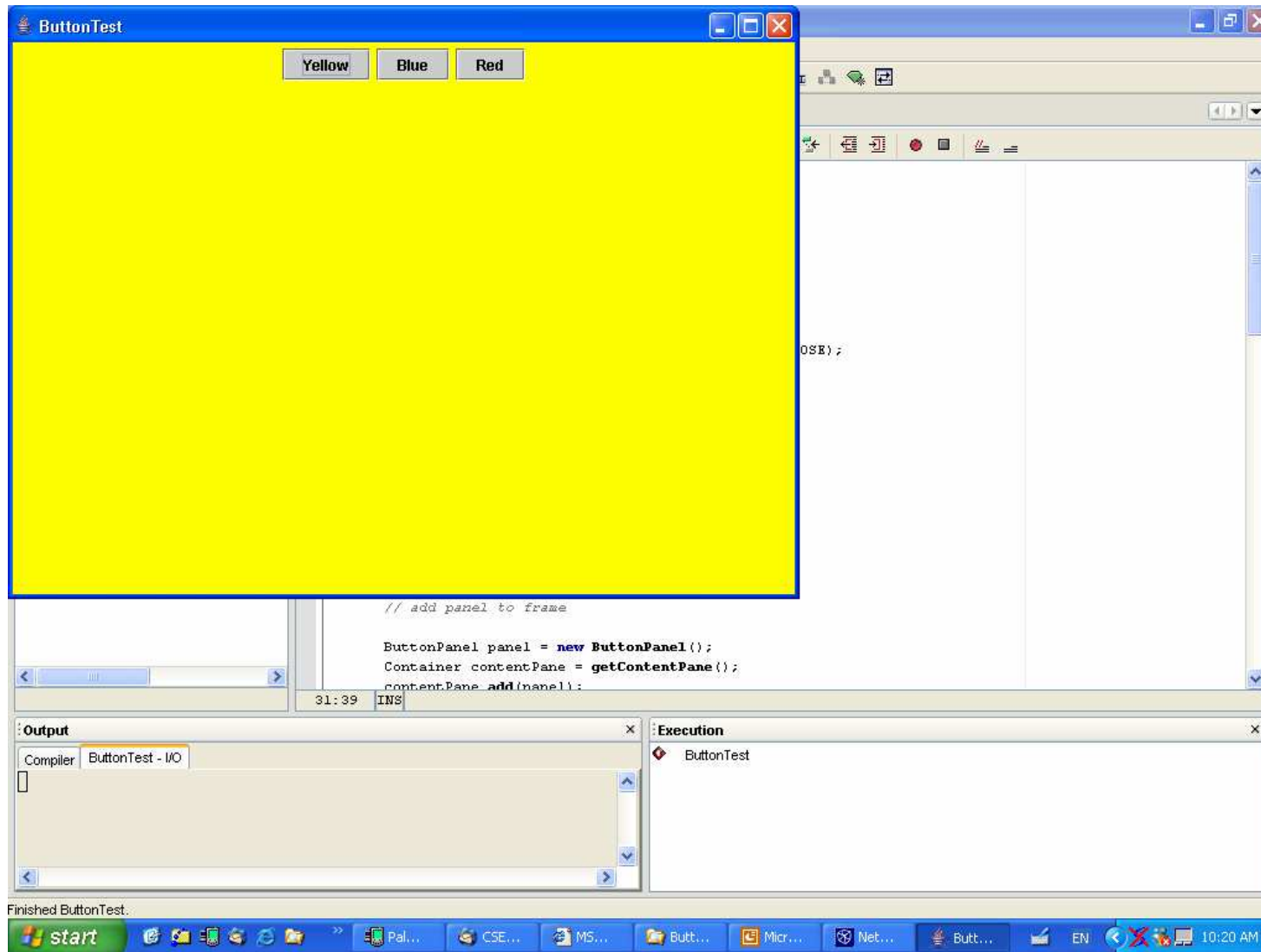
Constructors – button with a label string, an icon, or both a label string and an icon

```
JButton blueButton = new JButton("Blue");  
JButton blueButton = new JButton(new ImageIcon("blue-  
ball.gif"));
```

Example: JButton Event 2



Example: JButton Event 3



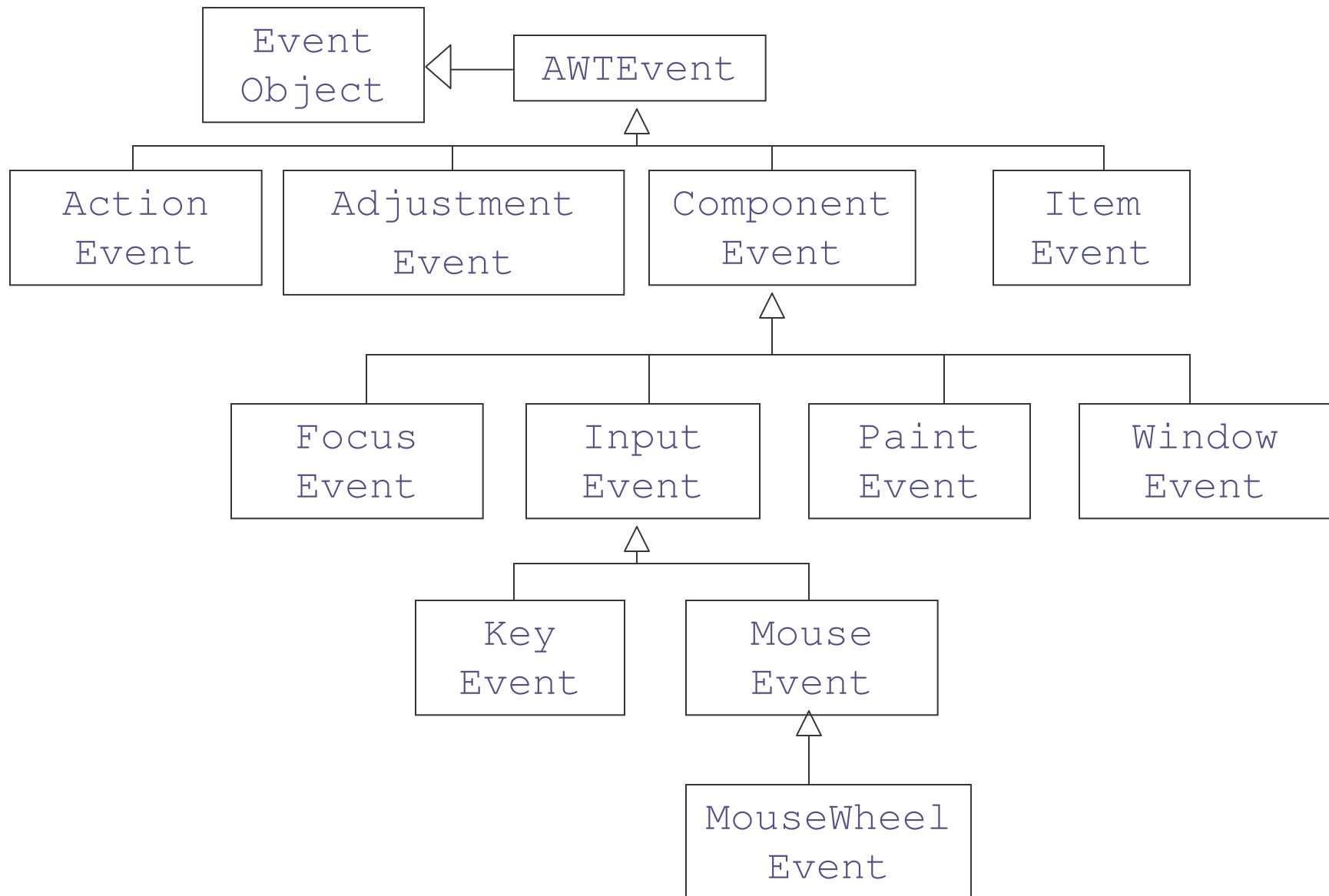
Event Hierarchy 1

All events in Java descend from the `EventObject` class in the `java.util` package

- a) The `EventObject` class has a subclass `AWTEvent`, which is the parent of all AWT event classes
- b) Some of the Swing components generate event objects that directly extend `EventObject`, not `AWTEvent`
- c) You can add your own custom events by subclassing `EventObject` or any of the subclasses

The event objects encapsulate information about the event that the event source communicates to its listeners.

Event Hierarchy 2



Example: AWT Event Objects

Commonly used AWT event types

- 1) `ActionEvent`
- 2) `AdjustmentEvent`
- 3) `FocusEvent`
- 4) `ItemEvent`
- 5) `KeyEvent`
- 6) `MouseEvent`
- 7) `MouseWheelEvent`
- 8) `WindowEvent`

Example: AWT Listener Interfaces

The following interfaces listen to these events

- 1) `ActionListener`
- 2) `AdjustmentListener`
- 3) `FocusListener`
- 4) `ItemListener`
- 5) `KeyListener`
- 6) `MouseListener`
- 7) `MouseMotionListener`
- 8) `MouseWheelListener`
- 9) `WindowListener`
- 10) `WindowFocusListener`
- 11) `WindowStateListener`

Adapter Classes

Adapter Class exists as convenience for creating a listener object.

Extend this class to create a listener for a particular listener interface and override the methods for the events of interest.

It defines `null` methods for all of the methods in the listener interface, so you can only have to define methods for events you care about.

Commonly used adapter classes:

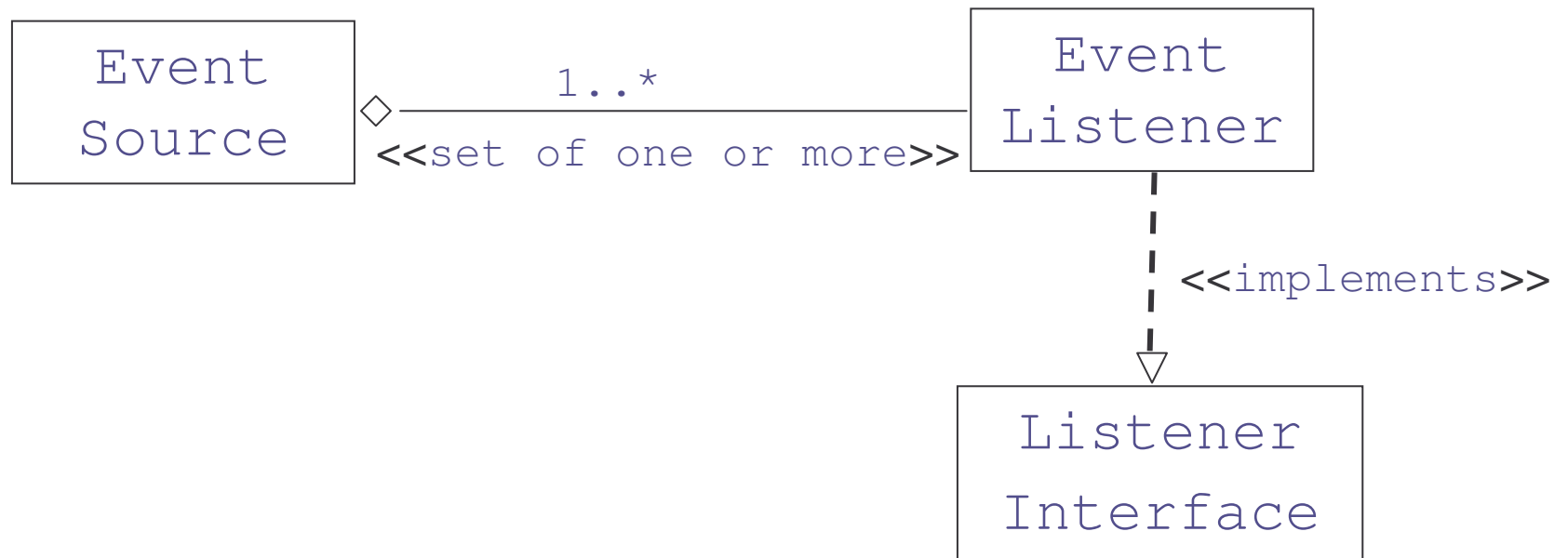
- 1) `FocusAdapter`
- 2) `KeyAdapter`
- 3) `MouseAdapter`
- 4) `MouseMotionAdapter`
- 5) `WindowAdapter`

Event Types

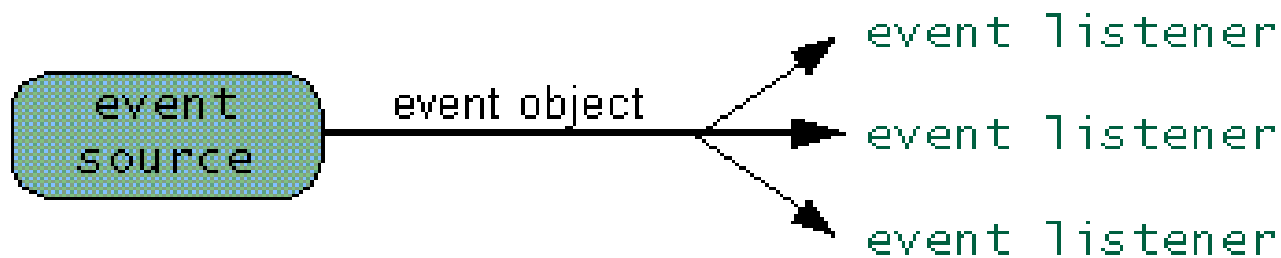
There are two types of events:

- 1) **Semantic event** – expresses what the user is doing
 - a) `ActionEvent` – button click, menu selection, list item selection, press ENTER in a text field
 - b) `AdjustmentEvent` – scrollbar adjustment
 - c) `ItemEvent` – selection from a set of checkbox or list items
- 2) **Low-level event** – makes user interactions possible
 - a) `FocusEvent` – a component got focus or lost focus
 - b) `KeyEvent` – key was pressed or released
 - c) `MouseEvent` – mouse button pressed, released, moved, dragged
 - d) `MouseEvent` – mouse wheel rotated
 - e) `WindowEvent` – window state changed

Event Handling Summary



Implementation:



Exercise: Event Handling

- 1) What listener would you implement to be notified when a particular component has appeared on screen? What method tells you this information?
- 2) What listener would you implement to be notified when the user has finished editing a text field by pressing Enter?
- 3) What listener would you implement to be notified as each character is typed into a text field? Note that you should not implement a general-purpose key listener, but a listener specific to text.
- 4) What listener would you implement to be notified when a spinner's value has changed? How would you get the spinner's new value?
- 5) The default behavior for the focus subsystem is to consume the focus traversal keys, such as Tab and Shift Tab. Say you want to prevent this from happening in one of your application's components. How would you accomplish this?

Object Collections

Course Outline

- | | | |
|---|--|--|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|--|--|

Overview 1

- 1) **Introduction** - tells you what collections are, and how they will make your job easier and your programs better.
- 2) **Interfaces** - describes the core collection interfaces, which are the heart and soul of the Java Collections Framework. You will learn:
 - a) general guidelines for effective use of these interfaces, including when to use which interface
 - b) idioms for each interface that will help you get the most out of the interfaces.

Overview 2

- 3) **Implementations** - describes the JDK's general-purpose collection implementations and tells you when to use which implementation.
- 4) **Algorithms** - describes the polymorphic algorithms provided by the JDK to operate on collections. With any luck you will never have to write your own sort routine again!

What Is a Collection?

Collection (sometimes called a container) is simply an object that groups multiple elements into a single unit.

Usage:

- 1) to store and retrieve data
- 2) to manipulate data
- 3) to transmit data from one method to another

Collections typically represent data items that form a natural group like:

- 1) a poker hand (a collection of cards)
- 2) a mail folder (a collection of letters)
- 3) a telephone directory (a collection of name-to-phone-number mappings)

Collection Framework 1

A **collections framework** is a unified architecture for representing and manipulating collections.

All collections frameworks contain three things:

1) **Interfaces**

- a) abstract data types representing collections.
- b) Interfaces allow collections to be manipulated independently of the details of their representation.

Collection Framework 2

2) Implementations

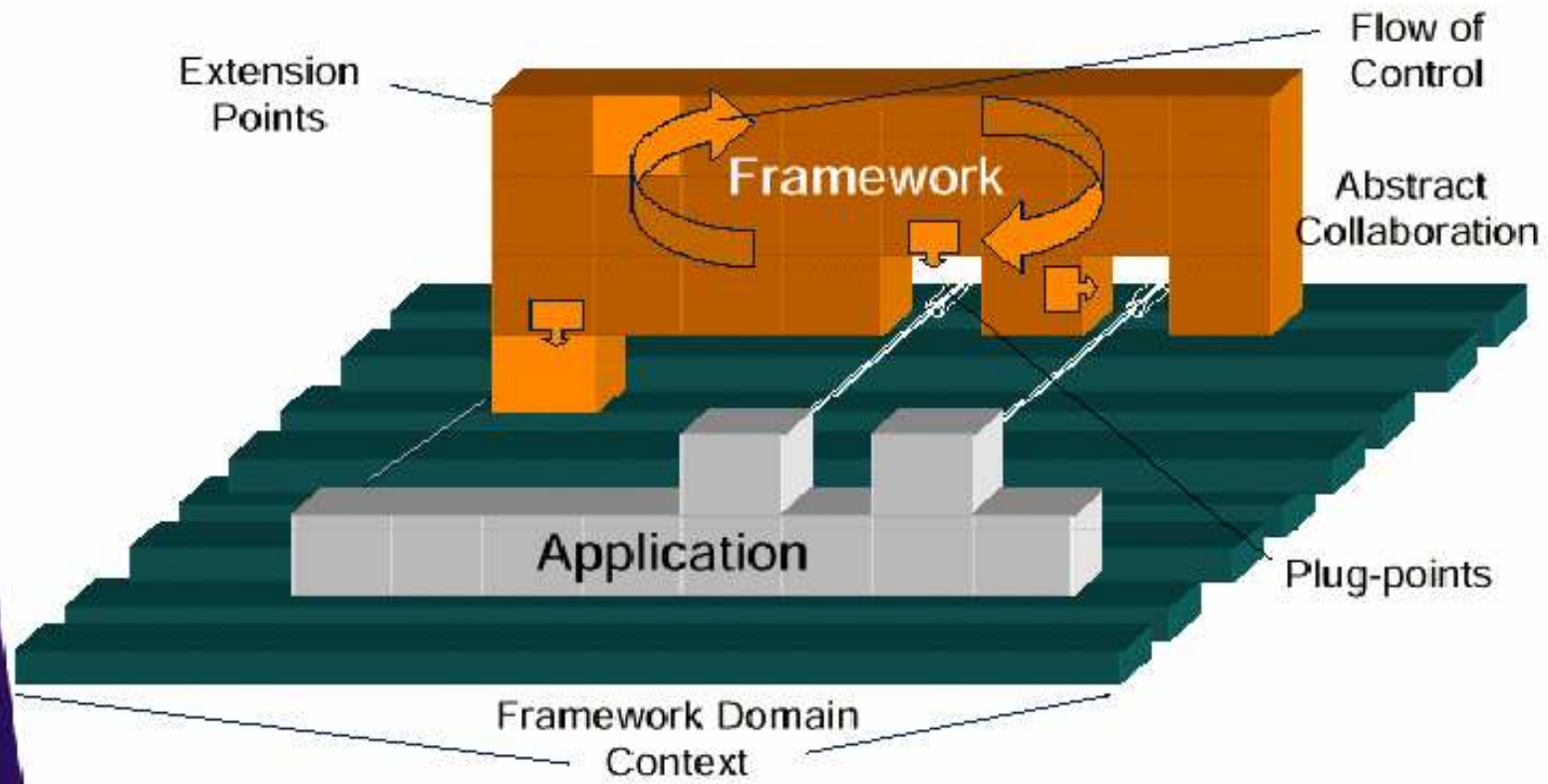
- a. concrete implementations of the collection interfaces.
- b. In essence, these are **reusable data structures**.

3) Algorithms

- a) methods that perform useful computations like searching and sorting, on objects that implement collection interfaces.
- b) they are polymorphic because the same method can be used on many different implementations of the appropriate collections interface.
- c) In essence, they are **reusable functionality**.

Collection Framework 3

Frameworks



Benefits 1

Collection Framework offers the following benefits:

- 1) It reduces programming effort
 - a) Powerful data structures and algorithms
- 2) It increases program speed and quality
 - a) High quality implementations
 - b) Fine tuning by switching implementations
- 3) It allows interoperability among unrelated APIs
 - a) Passing objects around from one API to another

Benefits 2

- 4) It reduces the effort to learn and use new APIs
 - a) Uniformity of the framework
 - b) APIs of applications
- 5) It reduces effort to design new APIs
- 6) It fosters software reuse
 - a) New data structures and algorithms

Core Collection Interfaces 1

These are interfaces used to manipulate collections, and pass them from one method to another.

Purpose:

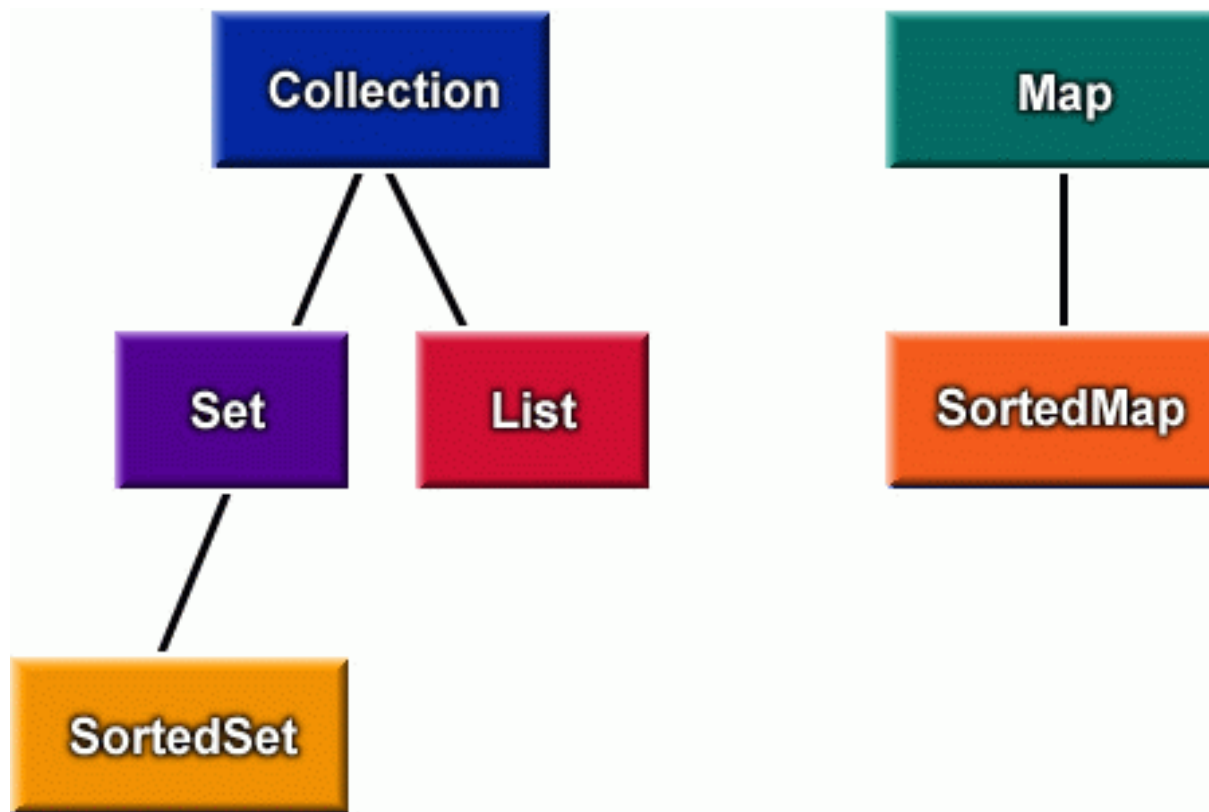
- 1) To allow collections to be manipulated independently of the details of their representation.

Note:

- 1) They are the heart and soul of the collections framework.
- 2) When you understand how to use these interfaces, you know most of what there is to know about the framework.

Core Collection Interfaces 2

The core collections interfaces are shown below:



Core Collection Interfaces 3

There are four basic core collection interfaces:

- 1) `Collection`
- 2) `Set`
- 3) `List`
- 4) `Map`

Collection

The `Collection_` interface is the root of the collection hierarchy.

Usage:

To pass around collections of objects where maximum generality is desired.

Behaviors:

- 1) Basic Operations
- 2) Bulk Operations
- 3) Array Operations

Collection Methods 1

```
public interface Collection {  
    // Basic Operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(Object element);    // Optional  
    boolean remove(Object element); // Optional  
    Iterator iterator();  
  
    // Bulk Operations  
    boolean containsAll(Collection c);  
    boolean addAll(Collection c);    // Optional  
    boolean removeAll(Collection c); // Optional  
    boolean retainAll(Collection c); // Optional  
    void clear();                    // Optional
```

Collection Methods 2

```
// Array Operations  
Object[] toArray();  
Object[] toArray(Object a[]);  
}
```

It has methods to tell you:

- 1) how many elements are in the collection (size, isEmpty)
- 2) to check if a given object is in the collection (contains)
- 3) to add and remove an element from the collection (add, remove),
- 4) and to provide an iterator over the collection (iterator)

Set 1

A `Set` is a `Collection` that cannot contain duplicate elements.

`Set` models the mathematical **set** abstraction.

Example:

- 1) Set of Cars - {BMW, Ford, Jeep, Chevrolet, Nissan, Toyota, VW}
- 2) Nationalities in the class - {Chinese, American, Canadian, Indian}

It extends `Collection` and contains **no** methods other than those inherited from `Collection`.

Set 2

`Set` extends `Collection` to add the following functionality:

- a) stronger contract on the behavior of the `equals` and `hashCode` operations,
- b) allowing `Set` objects with different implementation types to be compared meaningfully.

Two `Set` objects are equal if they contain the same elements.

Set Methods 1

The `Set` interface is shown below:

```
public interface Set {  
    // Basic Operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(Object element);    // Optional  
    boolean remove(Object element); // Optional  
    Iterator iterator();  
  
    // Bulk Operations  
    boolean containsAll(Collection c);  
    boolean addAll(Collection c);    // Optional  
    boolean removeAll(Collection c); // Optional
```

Set Methods 2

```
boolean retainAll(Collection c); // Optional
void clear();                    // Optional

// Array Operations
Object[] toArray();
Object[] toArray(Object a[]);
}
```

Provides two general purpose implementations:

- 1) `HashSet` – which stores its elements in a hash table, is the best-performing
- 2) `TreeSet` – which stores its elements in a red-black tree, guarantees the order of iteration.

Example: Set

```
import java.util.*;

public class FindDups {
    public static void main(String args[]) {
        Set s = new HashSet();
        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.print("Duplicate")
                System.out.println("detected: "+args[i]);
                System.out.println(s.size()+" distinct")
                System.out.println("words detected: "+s);
    }
}
```

List 1

An ordered collection (sometimes called a sequence)

Lists may contain duplicate elements

Collection operations:

- 1) remove operation removes the first occurrence of the specified element
- 2) add and addAll operations always appends new elements to the end of the list.
- 3) Two List objects are equal if they contain the same elements in the same order.

List 2

- 1) New List operations
 - a) Positional access
 - b) Search
 - c) Iteration (ordered, backward)
 - d) Range-view operations

- 2) General Purpose Implementation
 - a) ArrayList
 - b) ListIterator

Example: List

```
private static void swap(List a, int i, int j)
{
    Object tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}

for (ListIterator i=l.listIterator(l.size());
     i.hasPrevious(); ) {
    Foo f = (Foo)i.previous();
    ...
}
```

Example: List

```
public static void replace(List l, Object val, List
    newVals) {
    for (ListIterator i = l.listIterator(); i.hasNext() ;
        ) {
        if (val==null ? i.next()==null :
            val.equals(i.next())) {
            i.remove();
            for (Iterator j = newVals.iterator(); j.hasNext();
                )
                i.add(j.next());
        }
    }
}
```


Iterator

A mechanism for iterating over a collection's elements.

Represented by the Iterator interface.

Iterator allows the caller to remove elements from the underlying collection during the iteration with well-defined semantics

The only safe way to modify a collection during iteration

Location: `java.util.Iterator`

Example: Iterator

```
static void filter(Collection c)
{
    for (Iterator it = c.iterator() ; it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

```
static void filter(Collection c)
{
    Iterator it = c.iterator();
    while (it.hasNext())
        if (!cond(it.next()))
            it.remove();
}
```

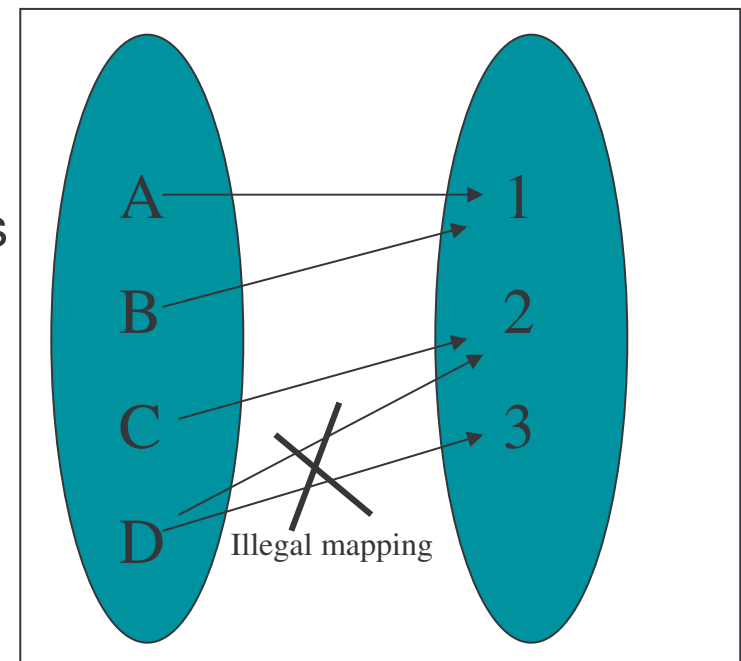
Map 1

A Map is an object that maps keys to values. Maps cannot contain duplicate keys.

- 1) Each key can map to at most one value
- 2) A map cannot contain duplicate keys.
- 3) Two Map objects are equal if they represent the same key-value mappings

Examples:

- a) Think of a dictionary:
word \leftrightarrow description
- b) address book
name \leftrightarrow phone number



Map

Map 2

- 1) Collection-view methods allow a Map to be viewed as a Collection
 - a) `keySet` – The Set of keys contained in the Map
 - b) `values` – The Collection of values contained in the Map
 - c) `entrySet` – The Set of key-value pairs contained in the Map
- 2) With all three Collection-views, calling an Iterator's `remove` operation removes the associated entry from the backing Map.
 - a) This is the only safe way to modify a Map during iteration.
- 3) Every object can be used as a hash key
- 4) Two Map objects are equal if they represent the same key-value mappings

Example: Map 1

```
import java.util.*;

public class Freq {
    private static final Integer ONE = new Integer(1);
    public static void main(String args[]){
        Map m = new HashMap();
        for (int i=0; i<args.length; i++) {
            Integer freq = (Integer)m.get(args[i]);
            m.put(args[i], (freq==null ?
                ONE : new Integer(freq.intValue() + 1)));
        }
        System.out.println(m.size() + " distinct words
detected:");
        System.out.println(m);
    } }
```

Example: Map 2

```
for (Iterator i = m.keySet().iterator() ;  
     i.hasNext() ; )
```

```
    System.out.println(i.next());
```

```
for (Iterator i = m.values().iterator();  
     i.hasNext() ; )
```

```
    System.out.println(i.next());
```

```
for (Iterator i = m.entrySet().iterator();  
     i.hasNext() ; ) {
```

```
    Map.Entry e = (Map.Entry)i.next();
```

```
    System.out.println(e.getKey() + ": " +  
        e.getValue());
```

```
}
```

SortedSet Interface

- 1) A Set that maintains its elements in ascending order
 - a) according to elements' natural order
 - b) according to a Comparator provided at SortedSet creation time
- 2) Set operations
 - a) Iterator traverses the sorted set in order
- 3) Additional operations
 - a) Range view – range operations
 - b) Endpoints – return the first or last element
 - c) Comparator access
- 4) Location: `java.util.SortedSet`


SortedMap Interface

- 1) A Map that maintains its entries in ascending order
 - a) According to keys' natural order
 - b) According to a Comparator provided at creation time.
- 2) Map operations
 - a) Iterator traverses elements in any of the sorted map's collection-views in key-order.
- 3) Additional Operations
 - a) Range view
 - b) End points
 - c) Comparator access
- 4) Location: `java.util.SortedMap`

Implementations

- 1) Implementations are the actual data objects used to store elements
 - a) Implement the core collection interfaces
- 2) There are three kinds of implementations
 - a) General purpose implementations
 - b) Wrapper implementations
 - c) Convenience implementations

General Purpose Implementations

		Implementations			
		Hash Table	Resizable Array	Balanced Tree	Linked List
Interfaces	Set	HashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap	

Older Implementations

- 1) The collections framework was introduced in JDK1.2.
- 2) Earlier JDK versions included collection implementations that were not part of any framework
 - a) `java.util.Vector`
 - b) `java.util.Hashtable`
- 3) These implementations were extended to implement the core interfaces but still have all their legacy operations
 - a) Be careful to always manipulate them only through the core interfaces.

Algorithms 1

Algorithms are pieces of reusable functionality provided by the JDK.

All of them come from the `Collections` class.

All take the form of static methods whose first argument is the collection on which the operation is to be performed.

The great majority of the algorithms provided by the Java platform operate on `List` objects,

A couple of them (min and max) operate on arbitrary `Collection` objects.

Algorithms 2

The algorithms are described below:

1) Sorting

- a) reorders a List so that its elements are ascending order according to some ordering relation.
- b) The important things to know about this algorithm are that it is:
 - Fast: This algorithm is guaranteed to run in $n \log(n)$ time, and runs substantially faster on nearly sorted lists.
 - Stable: That is to say, it doesn't reorder equal elements.

2) Shuffling

- a) does the opposite of what sort does - it destroys any trace of order that may have been present in a List.

Algorithms 3

3) Routine Data Manipulation

- a) The Collections class provides three algorithms for doing routine data manipulation on List objects: `Reverse`, `Fill` and `Copy`

4) Searching

- a) The `binarySearch` algorithm searches for a specified element in a sorted List using the `binary search` algorithm.

5) Finding Extreme Values

- a) The `min` and `max` algorithms return, respectively, the minimum and maximum element contained in a specified Collection.

Exercise: Object Collection

- 1) Write a class that stores an object of `Building`, `HighBuilding`, and `Skyscraper` into a Map (either a `HashMap` or `TreeMap`)
- 2) Store each of these elements into an `ArrayList`.
- 3) Create another class that takes a variable `ArrayList` as a parameter.
- 4) Call a method on the class to iterate through the `ArrayList` and call `enter` method of each object. Direct your output to the console.
- 5) Write a program to count the number of different words occurring in a text. Whenever we inspect the next word in the text we need to know if it has already occurred or not, so we need to store the words that have already occurred in *some* data structure in our program.

For this program we need a data structure that stores the words we have already encountered in the text in order to look following words up and decide whether to count one more different word or not.

Test your program with the text of exercise 5. Print out the content of the data structure.

Vertical Libraries

Course Outline

- | | | |
|---|--|--|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|--|--|

Graphical Interface

Course Outline

- | | | |
|---|--|---|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|--|---|

Overview

- 1) A Quick Start Guide
- 2) Swing Features and Concepts
- 3) Summary

Overview

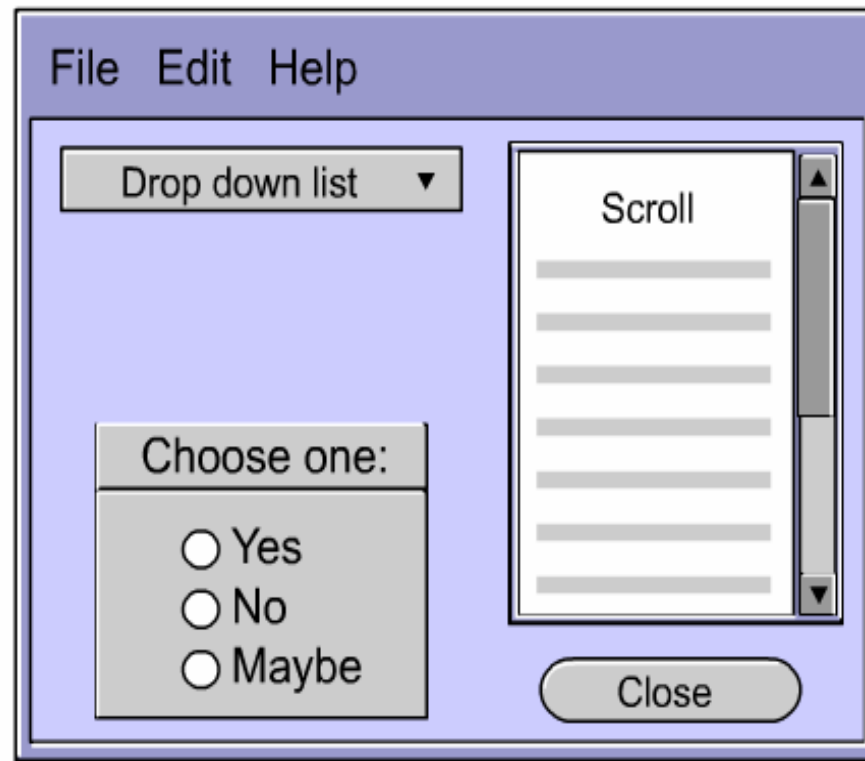
- 1) **A Quick Start Guide**
- 2) Swing Features and Concepts
- 3) Summary

A Quick Start Guide

- 1) Overview of Swing API
- 2) First Swing Application
- 3) Swing Application (SA)
 - a) Look and Feel
 - b) Setting up Button, Label
 - c) Handling Events
 - d) Border and Component
- 4) CelsiusConverter
 - a) Adding HTML
 - b) Adding an Icon
- 5) LunarPhases
 - a) Compound Border
 - b) Combo Boxes
 - c) Multiple Images

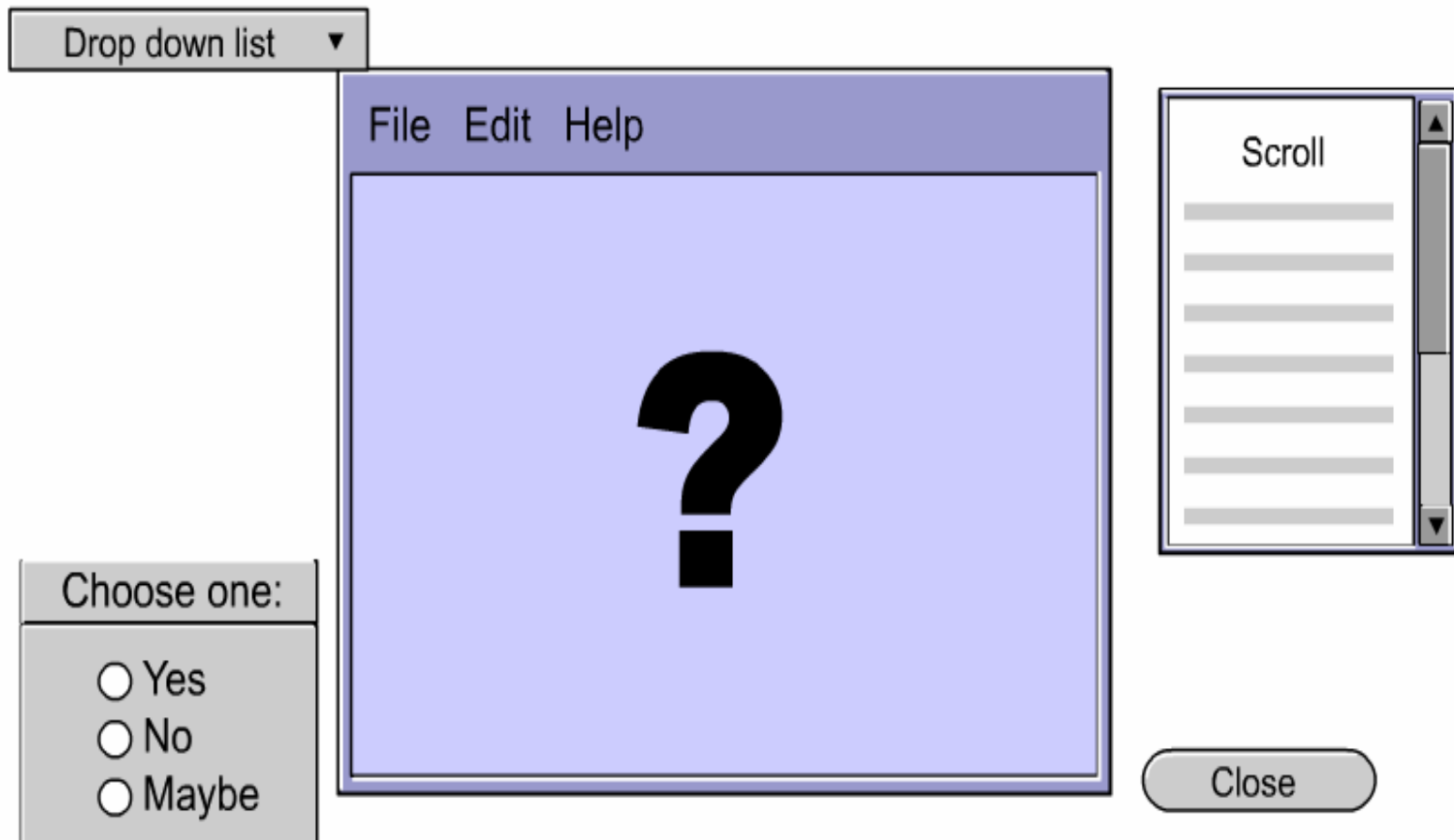
6. Layout Management

Overview of Swing API 1



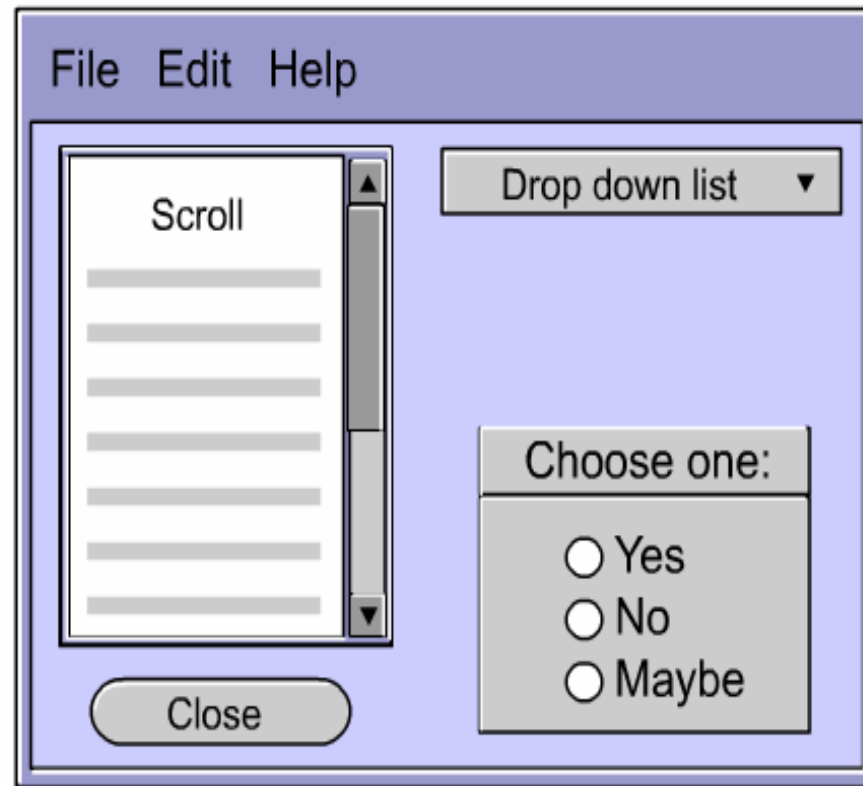
Modern computer applications usually use a graphical user interface, or GUI, to interact with users.

Overview of Swing API 2



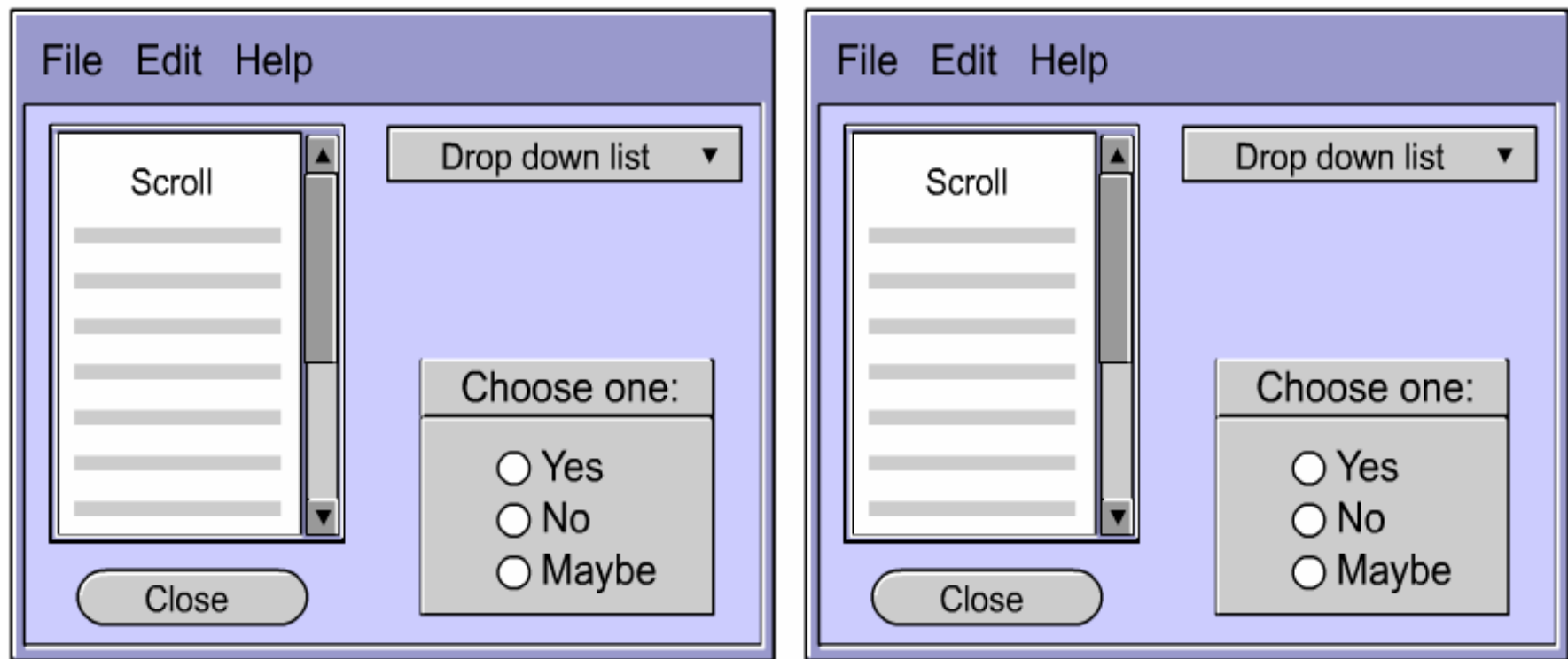
One item to consider is how to position each component on the GUI.

Overview of Swing API 3



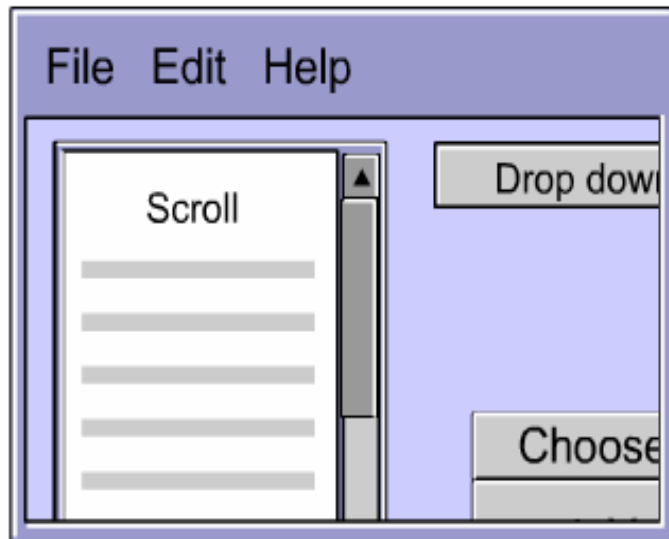
One item to consider is how to position each component on the GUI.

Overview of Swing API 4



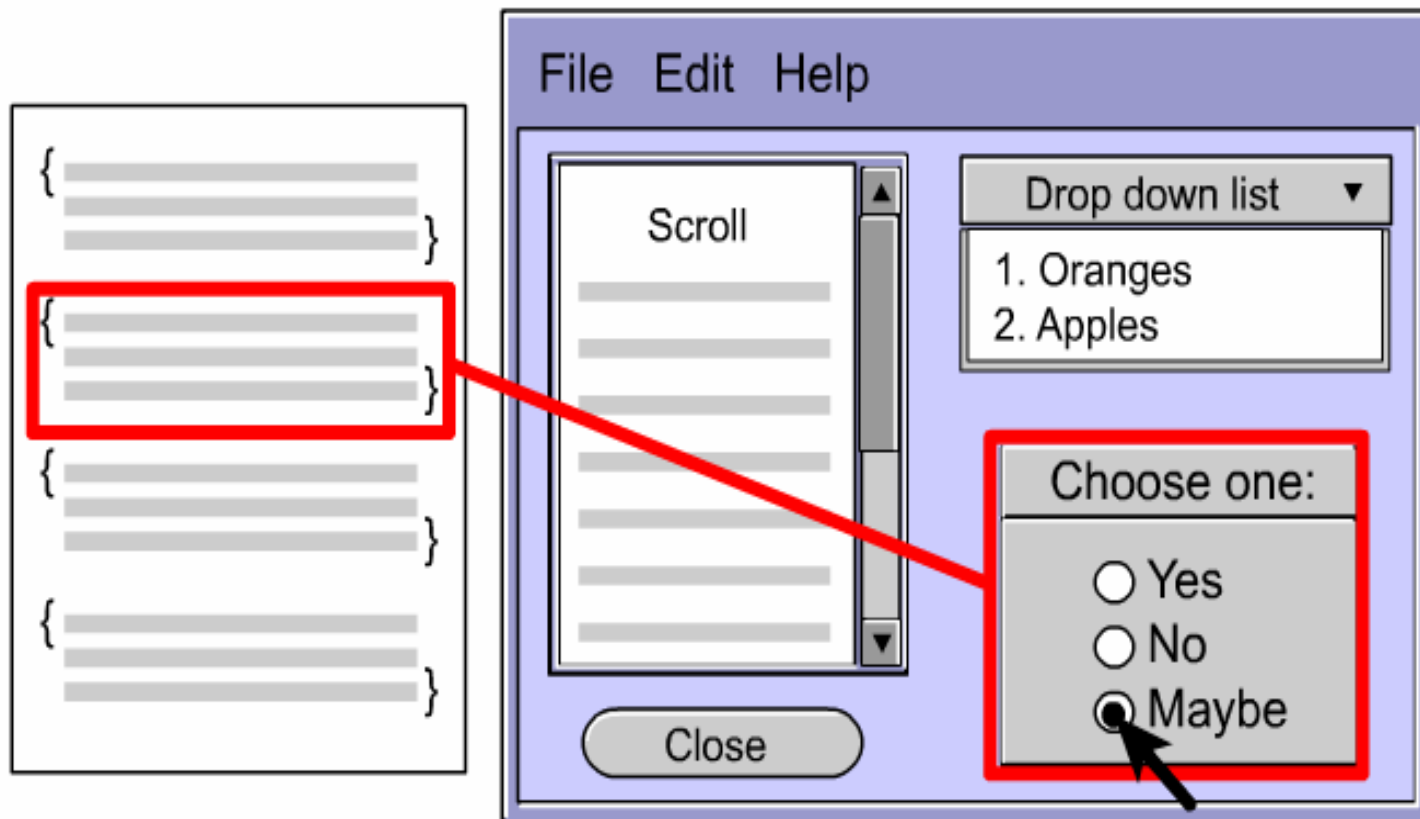
Another thing to consider is what happens when the window is resized.

Overview of Swing API 5



Another thing to consider is what happens when the window is resized.

Overview of Swing API 6



Finally, the GUI needs to be functional. Specific behaviors need to be associated with each component.

Overview of Swing API 7

What are the JFC and Swing?

Definition

JFC is short for **Java Foundation Classes**, which encompass a group of features to help people build graphical user interfaces (GUIs).

First announced at the 1997 **JavaOne developer conference** and is defined as containing the following features:

- 1) **Swing Components** - from buttons to split panes to tables
- 2) **Pluggable Look and Feel Support** - looks and feels
- 3) **Accessibility API** - enables assistive technologies
- 4) **Java 2D API** - high-quality 2D graphics, text, images
- 5) **Drag and Drop Support** - ability to drag and drop between a Java application and a native application

Overview of Swing API 7

The first three JFC features were implemented without any native code, relying only on the API defined in JDK 1.1

This extension was released as JFC 1.1, which is sometimes called "**the Swing release.**"

The API in JFC 1.1 is often called "**the Swing API.**"

Note: "**Swing**" was the codename of the project that developed the new components.

Overview of Swing API 8

Prior to the introduction of the Swing, AWT (Abstract Window Toolkit) provided all the UI components in the JDK 1.0 and 1.1 platforms.

Java 2 Platform still supports the AWT components

AWT is in `java.awt` while Swing is in `javax.swing`

Swing Components starts with "J"

AWT button – Button

Swing Button - JButton

First Swing Application 1

This section examines the code for a simple program, **HelloWorldSwing**. This introduces some basic features of Swing.

```
import javax.swing.*;

public class HelloWorldSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("HelloWorldSwing");
        final JLabel label = new JLabel("Hello World");
        frame.getContentPane().add(label);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.pack();
        frame.setVisible(true);
    }
}
```


First Swing Application 2

The code demonstrates the basic code in every Swing program:

- 1) Import the pertinent packages
- 2) Set up a top-level container

The first line imports the main Swing package:

```
import javax.swing.*;
```

However, most Swing programs also need to import two AWT packages because Swing components use the AWT infrastructure, including the **AWT event model**.

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

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

First Swing Application 3

Every program with a Swing GUI must contain at least one **top-level Swing container**.

Definition

A **top-level Swing container** provides the support that Swing components need to perform their painting and event handling.

There are **three top-level** containers:

- 1) `JFrame`- implements a single main window
- 2) `JDialog`- implements a secondary window (a window that's dependent on another window)
- 3) `JApplet`- implements an applet's display area within a browser window

First Swing Application 4

Here is the code that sets up and shows the frame:

```
JFrame frame = new JFrame("HelloWorldSwing");  
...  
frame.pack();  
frame.setVisible(true);
```

These two lines of code construct a label and then add the component to the frame.

```
final JLabel label = new JLabel("Hello World");  
frame.getContentPane().add(label);
```

This code closes the window when  is clicked

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

First Swing Application 5

JFrame provides the `setDefaultCloseOperation` method to configure the default action for when the user clicks the close button.

The `EXIT_ON_CLOSE` constant lets you specify this, as of version 1.3 of the Java 2 Platform.

To implement this in earlier version, you have to write a window event listener.

```
frame.addWindowListener(new WindowAdapter() {  
    public void windowClosing(WindowEvent e) {  
        System.exit(0);  
    }  
}) ;
```

Basic Features

We use the following applications to illustrate the basic features of swing:

- 1) `SwingApplication.java`
- 2) `CelsiusConverter.java`
- 3) `LunarPhases.java`

Example: SwingApplication 1

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class SwingApplication {
    private static String labelPrefix = "Number of
                                         button clicks: ";
    private int numClicks = 0;
    public Component createComponents() {
        final JLabel label = new JLabel(labelPrefix +
                                         "0");
        JButton button = new JButton("I'm a Swing
                                     button!");
        button.setMnemonic(KeyEvent.VK_I);
    }
}
```

Example: SwingApplication 2

```
button.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e) {
        numClicks++;
        label.setText(labelPrefix + numClicks);
    }
});
label.setLabelFor(button);
JPanel pane = new JPanel();
```

Example: SwingApplication 3

```
pane.setBorder(BorderFactory.createEmptyBorder(  
                                                    30, //top  
                                                    30, //left  
                                                    10, //bottom  
                                                    30) //right  
                                                    );  
  
pane.setLayout(new GridLayout(0, 1));  
pane.add(button);  
pane.add(label);  
  
return pane;  
}
```


Example: SwingApplication 4

```
public static void main(String[] args) {  
    try {  
        UIManager.setLookAndFeel(  
            UIManager.getCrossPlatformLookAndFeelClassName(  
                ));  
    } catch (Exception e) {}  
    JFrame frame = new JFrame("SwingApplication");  
    SwingApplication app = new SwingApplication();  
    Component contents = app.createComponents();  
    frame.getContentPane().add(contents,  
                                BorderLayout.CENTER);  
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_C  
                                LOSE);  
    frame.pack(); frame.setVisible(true);}}
```

Example: SwingApplication 5

This program illustrates the following features of swing:

- 1) Look and Feel
- 2) Setting Up Buttons and Labels
- 3) Handling Events
- 4) Adding Borders Around Components

Example: SwingApplication 5



Look and Feel 1

Swing allows you to specify which look and feel your program uses

- 1) Java look and feel
- 2) CDE/Motif look and feel
- 3) Windows look and feel, and so on

Example: Look and Feel

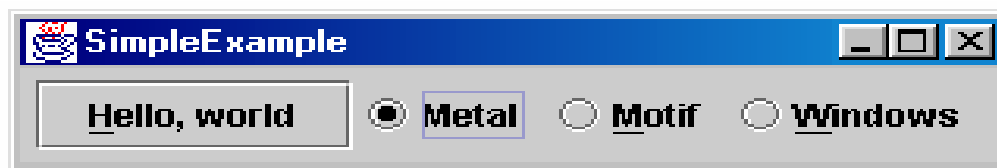
The code snippet shows you how `SwingApplication` specifies its look and feel:

```
public static void main(String[] args) {  
    try {  
        UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());  
    } catch (Exception e) { }  
    ...// Create and show the GUI...  
}
```

Use cross-platform look and feel.

Example: Look and Feel

The following figure shows three views of a GUI that uses Swing components.



Java look and feel



CDE/Motif look and feel



Windows look and feel

Buttons and Labels

Here's the code that initializes the button:

```
JBUTTON button = new JBUTTON("I'm a Swing button!");  
button.setMnemonic('i');  
button.addActionListener(...an actionlistener...);
```

```
private static String labelPrefix = "Number of button  
clicks: ";
```

```
private int numClicks = 0;
```

```
...// in GUI initialization code:
```

```
final JLabel label = new JLabel(labelPrefix + "0  
"); ...
```

```
label.setLabelFor(button);
```

```
...// in the event handler for button clicks:
```

```
label.setText(labelPrefix + numClicks);
```

Example: CelsiusConverter 1

This program illustrates two features of swing:

- 1) Adding HTML to a Label
- 2) Adding Icon to a swing Button

Example: CelsiusConverter 1

This program illustrates two features of swing:

- 1) Adding HTML to a Label
- 2) Adding Icon to a swing Button

Example: CelsiusConverter 3

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

public class CelsiusConverter implements
    ActionListener {
    JFrame converterFrame;
    JPanel converterPanel;
    JTextField tempCelsius;
    JLabel celsiusLabel, fahrenheitLabel;
    JButton convertTemp;

    public CelsiusConverter() {
        converterFrame = new JFrame("Convert Celsius to
                                    Fahrenheit");
```

Example: CelsiusConverter 4

```
converterPanel = new JPanel();  
converterPanel.setLayout(new GridLayout(2, 2));  
addWidgets();  
converterFrame.getContentPane().add(  
    converterPanel, BorderLayout.CENTER);  
converterFrame.setDefaultCloseOperation(JFrame.  
    EXIT_ON_CLOSE);  
converterFrame.pack();  
converterFrame.setVisible(true);  
}
```

Example: CelsiusConverter 5

```
private void addWidgets() {  
tempCelsius = new JTextField(2);  
celsiusLabel = new JLabel("Celsius",  
                           SwingConstants.LEFT);  
convertTemp = new JButton("Convert...");  
fahrenheitLabel = new JLabel("Fahrenheit",  
                              SwingConstants.LEFT);  
convertTemp.addActionListener(this);  
converterPanel.add(tempCelsius);  
converterPanel.add(celsiusLabel);  
converterPanel.add(convertTemp);  
converterPanel.add(fahrenheitLabel);  
}
```

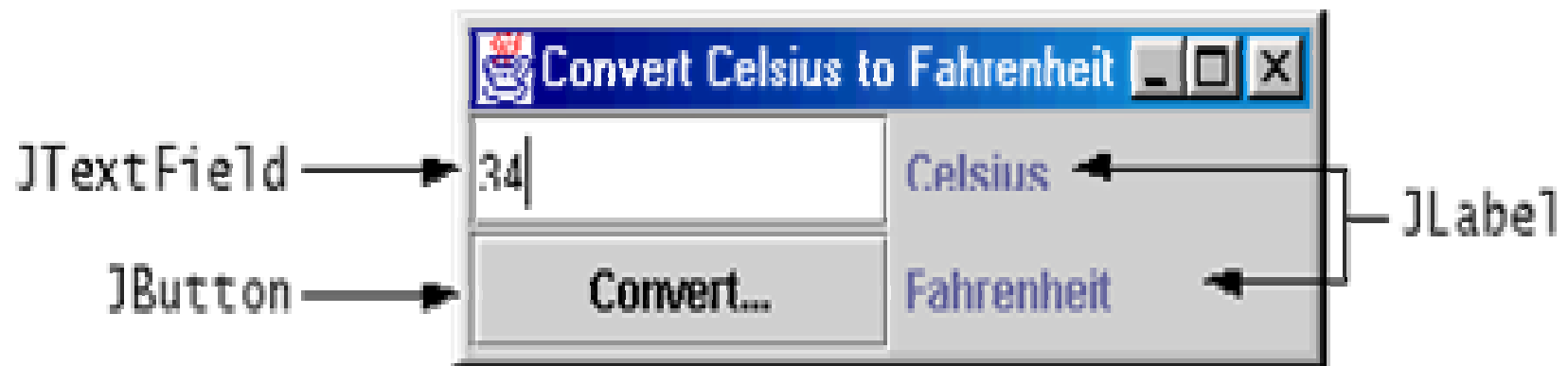
Example: CelsiusConverter 6

```
celsiusLabel.setBorder(BorderFactory.  
createEmptyBorder(5, 5, 5, 5));  
  
fahrenheitLabel.setBorder(BorderFactory.  
createEmptyBorder(5, 5, 5, 5));  
  
}  
  
public void actionPerformed(ActionEvent event) {  
    int tempFahr =  
(int) ((Double.parseDouble(tempCelsius.getText()))  
        * 1.8 + 32);  
    fahrenheitLabel.setText(tempFahr + " Fahrenheit");  
}
```

Example: CelsiusConverter 7

```
    public static void main(String[] args) {  
    try {  
        UIManager.setLookAndFeel(  
            UIManager.getCrossPlatformLookAndFeelClassName(  
                )));  
    } catch (Exception e) {}  
    CelsiusConverter converter = new  
        CelsiusConverter();  
    }  
}
```

Example: CelsiusConverter 2



Adding HTML

You can use HTML to specify the text on some Swing components, such as buttons and labels.

To do this we modify the actionPerformed event as follows

```
if (tempFahr <= 32) {  
    fahrenheitLabel.setText("<html><font color=blue>" +  
        tempFahr + "&#176 Fahrenheit </font></html>");  
} else if (tempFahr <= 80) {  
    fahrenheitLabel.setText("<html><font color=green>" +  
        tempFahr + "&#176 Fahrenheit </font></html>");  
} else {  
    fahrenheitLabel.setText("<html><font color=red>" +  
        tempFahr + "&#176 Fahrenheit </font></html>");  
}
```


Adding Icon

Some Swing components can be decorated with an icon--a fixed size

A Swing icon is an object that adheres to the Icon interface.

Swing provides a particularly useful implementation of the Icon interface: ImageIcon.

`ImageIcon` paints an icon from a GIF or a JPEG image.

Here's the code that adds the arrow graphic to the `convertTemp` button:

```
ImageIcon icon = new  
ImageIcon("images/convert.gif", "Convert  
temperature");  
  
...  
  
convertTemp = new JButton(icon);
```

Example: LunarPhrases 1

This program illustrates two features of swing

- 1) Compound Borders
- 2) Combo Boxes
- 3) Loading Multiple Images

Example: Lunar Phrases



Compound Border

Both subpanels, selectPanel and displayPanel, have a compound border.

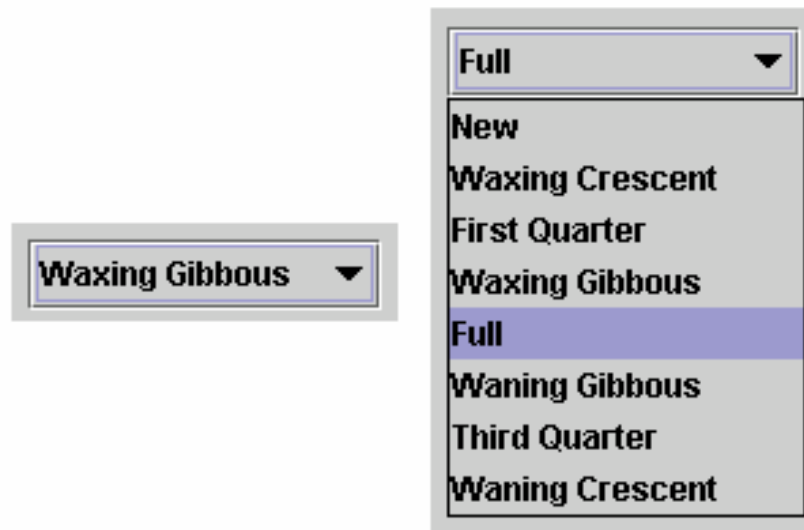
This compound border consists of a titled border (an outlined border with a title) and an empty border (to add extra space),

The code for the selectPanel border follows.

```
selectPanel.setBorder(BorderFactory.createCompoundBorder(
    BorderFactory.createTitledBorder("Select
Phase"), BorderFactory.createEmptyBorder(5, 5, 5, 5)));
```

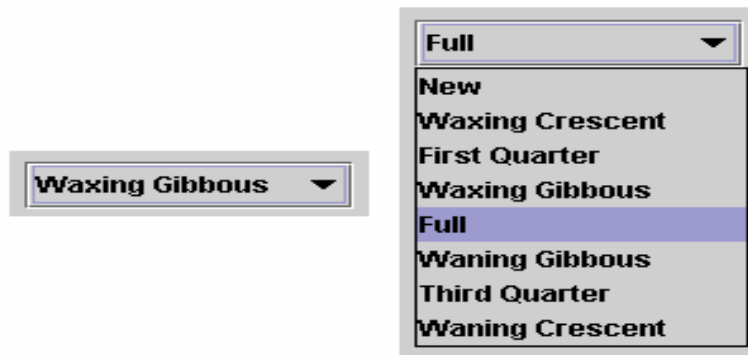
Combo Boxes 1

- 1) A combo box enables user choice.
- 2) A combo box can be either **editable** or **uneditable**, but by default it is **uneditable**
- 3) An **uneditable** combo box looks like a button until the user interacts with it. When the user clicks it, the combo box displays a menu of items.



Combo Boxes 2

- 1) A combo box enables user choice.
- 2) A combo box can be either **editable** or **uneditable**, but by default it is **uneditable**
- 3) An **uneditable** combo box looks like a button until the user interacts with it. When the user clicks it, the combo box displays a menu of items.



- 4) When to use **uneditable combo box**
 - a) when space is limited,
 - b) when the number of choices is large, or when the menu items are computed at runtime.

Combo Boxes 2

To populate a combo box:

1. initialise with an array of Strings or Vector
2. add the array of Strings or Vector to the constructor

Example:

```
JComboBox phaseChoices = null;  
...  
// Create combo box with lunar phase choices  
String[] phases = { "New", "Waxing Crescent", "First  
Quarter", "Waxing Gibbous", "Full", "Waning Gibbous",  
                    "Third Quarter", "Waning Crescent" };  
phaseChoices = new JComboBox(phases);  
phaseChoices.setSelectedIndex(START_INDEX);
```

Combo Box Event

The combo box fires an **action** event when the user selects an item from the combo box's menu.

```
phaseChoices.addActionListener(this);  
...  
public void actionPerformed(ActionEvent event) {  
    if  
    ("comboBoxChanged".equals(event.getActionCommand  
    ())) {  
        // update the icon to display the new  
        phase  
        phaseIconLabel.setIcon(images[phaseChoices.getSe  
        lectedIndex()]);  
    }  
}
```


Multiple Images 1

- 1) In previous program, we saw how to add a single ImageIcon to a button.
- 2) LP eight images all loaded at a time.

```
final static int NUM_IMAGES = 8;
final static int START_INDEX = 3;

ImageIcon[] images = new ImageIcon[NUM_IMAGES];
...
private void addWidgets() {
    // Get the images and put them into an array of
    ImageIcon.
    for (int i = 0; i < NUM_IMAGES; i++) {
```

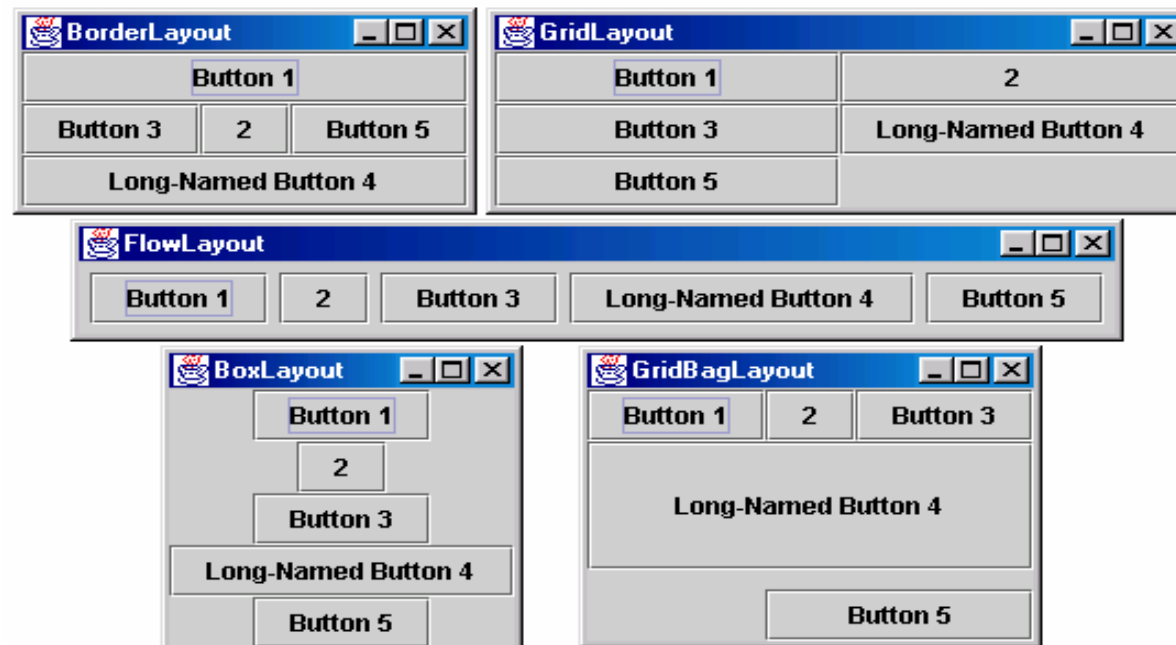
Multiple Images 2

```
String imageName = "images/image" + i + ".jpg";
    URL iconURL =
        ClassLoader.getResource(imageName);

    ImageIcon icon = new ImageIcon(iconURL);
    images[i] = icon;
}
}
```

Note: the use of `getResource`, a method in `ClassLoader` that searches the `classpath` to find the image file names so that we don't have to specify the fully qualified path name.

Layout Management



- 1) The figure above shows the GUI of five different programs.
- 2) The buttons are identical and the code are almost identical.
So why do the GUIs look so different?
 - a) Because they use different layout managers to control the size and the position of the buttons.

Using Layout Managers 1

The Java platform supplies five commonly used layout managers:

- 1) BorderLayout
- 2) BoxLayout
- 3) FlowLayout
- 4) GridBagLayout, and
- 5) GridLayout

Using Layout Managers 2

- 1) By default, every container has a layout manager
- 2) JPanel objects use a **FlowLayout** by default
- 3) main containers in **JApplet**, **JDialog**, and **JFrame** objects use **BorderLayout** by default
- 4) As a rule, the only time you have to think about layout managers is when you **create a JPanel** or **add components** to a content pane.

Using Layout Managers 3

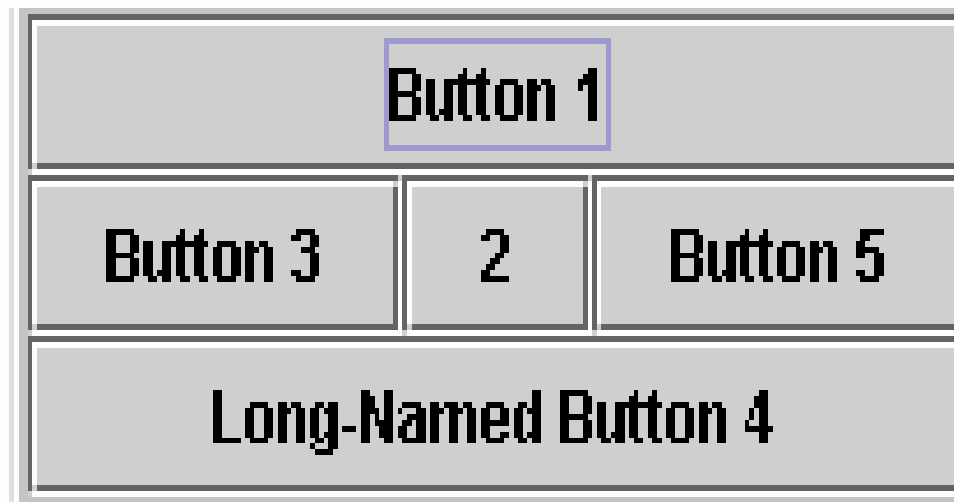
If you don't like the default layout manager that a panel or content pane uses, you can change it to a different one by setting the `setLayout` method of the content pane.

```
JPanel pane = new JPanel();  
pane.setLayout(new BorderLayout());
```

BorderLayout 1

BorderLayout

1. the default layout manager for every content pane.
2. the main container in all **frames**, **applets**, and **dialogs**
3. has five areas available to hold components
north, south, east, west, and center



Example: BorderLayout 2

How to use BorderLayout

```
...  
//Container pane = aFrame.getContentPane()  
...  
JButton button = new JButton("Button 1  
    (PAGE_START)");  
pane.add(button, BorderLayout.PAGE_START);  
  
//Make the center component big, since that's the  
//typical usage of BorderLayout.  
button = new JButton("Button 2 (CENTER)");  
button.setPreferredSize(new Dimension(200, 100));  
pane.add(button, BorderLayout.CENTER);
```


BorderLayout 3

```
button = new JButton("Button 3 (LINE_START)");  
pane.add(button, BorderLayout.LINE_START);
```

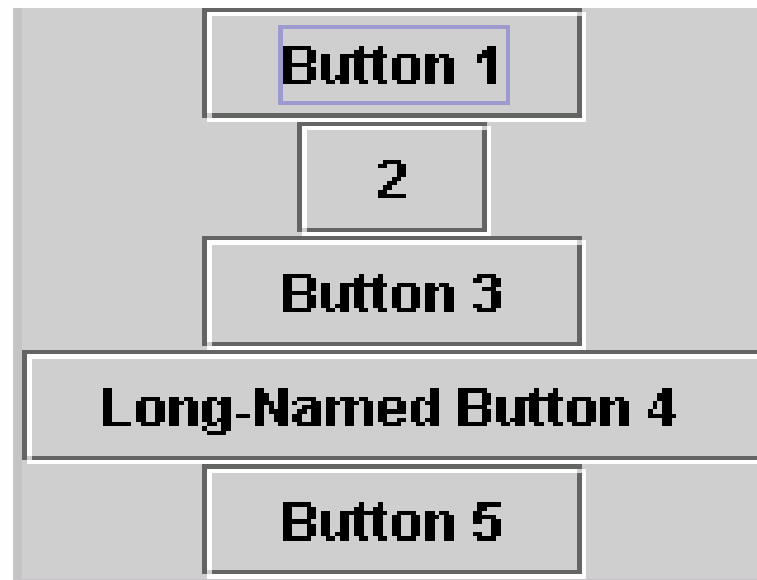
```
button = new JButton("Long-Named Button 4PAGE_END");  
pane.add(button, BorderLayout.PAGE_END);
```

```
button = new JButton("5 (LINE_END)");  
pane.add(button, BorderLayout.LINE_END);
```

BoxLayout 1

BoxLayout

- 1) puts components in a single row or column
- 2) respects the components' requested maximum sizes
- 3) also lets you align components



Example: BoxLayout 2

```
public static void addComponentsToPane(Container
pane) {
    pane.setLayout(new BoxLayout(pane,
                                BoxLayout.Y_AXIS));
    addAButton("Button 1", pane);
    addAButton("Button 2", pane);
    addAButton("Button 3", pane);
    addAButton("Long-Named Button 4", pane);
    addAButton("5", pane);
}
```

FlowLayout

FlowLayout

- 1) default layout manager for every JPanel
- 2) lays out components from left to right, starting new rows, if necessary
- 3) respects the components' requested maximum sizes
- 4) also lets you align components



Example: FlowLayout

```
contentPane.setLayout(new FlowLayout());  
contentPane.add(new JButton("Button 1"));  
contentPane.add(new JButton("Button 2"));  
contentPane.add(new JButton("Button 3"));  
contentPane.add(new JButton("Long-Named Button 4"));  
contentPane.add(new JButton("5"));
```

FlowLayout

FlowLayout API

The FlowLayout class has three constructors:

```
public FlowLayout()  
public FlowLayout(int alignment)  
public FlowLayout(int alignment, int  
                  horizontalGap, int verticalGap)
```

The **alignment** argument can be

```
FlowLayout.LEADING,  
FlowLayout.CENTER, or  
FlowLayout.TRAILING
```

GridLayout

Makes a bunch of components equal in size and displays them in the requested number of rows and columns

Button 1	2
Button 3	Long-Named Button 4
Button 5	

Example: GridLayout

```
pane.setLayout(new GridLayout(0,2));  
  
pane.add(new JButton("Button 1"));  
pane.add(new JButton("Button 2"));  
pane.add(new JButton("Button 3"));  
pane.add(new JButton("Long-Named Button 4"));  
pane.add(new JButton("5"));
```


GridLayout

The GridLayout class has two constructors:

```
public GridLayout(int rows, int columns)
public GridLayout(int rows, int columns,
                  int horizontalGap, int verticalGap)
```

- 1) At least one of the **rows** and **columns** arguments must be nonzero
- 2) the **rows** argument has precedence over the **columns** argument
- 3) The **horizontalGap** and **verticalGap** arguments to the second constructor allow you to specify the number of pixels between cells. If you don't specify gaps, their values default to zero.

GridBagLayout

- 1) the most **sophisticated** and **flexible** layout manager
- 2) aligns components by placing them within a grid of cells, allowing some components to span more than one cell
- 3) The rows in the grid aren't necessarily all the same height; similarly, grid columns can have different widths.



Example: GrigBagLayout 1

```
JButton button;  
pane.setLayout(new GridBagLayout());  
GridBagConstraints c = new GridBagConstraints();  
c.fill = GridBagConstraints.HORIZONTAL;  
  
button = new JButton("Button 1");  
c.weightx = 0.5;  
c.gridx = 0;  
c.gridy = 0;  
pane.add(button, c);
```

Example: GrigBagLayout 2

```
button = new JButton("Button 2");
c.gridx = 1;
c.gridy = 0;
pane.add(button, c);
button = new JButton("Button 3");
c.gridx = 2;
c.gridy = 0;
pane.add(button, c);
button = new JButton("Long-Named Button 4");
c.ipady = 40;           //make this component tall
c.weightx = 0.0;
c.gridwidth = 3;
c.gridx = 0;
c.gridy = 1;
pane.add(button, c);
```

Example: GrigBagLayout 3

```
button = new JButton("5");
c.ipady = 0;           //reset to default
c.weighty = 1.0;       //request any extra vertical space
c.anchor = GridBagConstraints.PAGE_END; //bottom of
                                     space
c.insets = new Insets(10,0,0,0); //top padding
c.gridx = 1;           //aligned with button 2
c.gridwidth = 2;       //2 columns wide
c.gridy = 2;           //third row
pane.add(button, c);
```

Overview

- 1) A Quick Start Guide
- 2) **Swing Features and Concepts**
- 3) Summary

Swing Features and Concepts

- 1) Components and the Containment Hierarchy
- 2) Event Handling

Components 1

- 1) **SwingApplication.java** creates four commonly used Swing components
 - a) a frame, or main window (JFrame)
 - b) a panel, sometimes called a pane (JPanel)
 - c) a button (JButton)
 - d) a label (JLabel)

- 2) The **frame** is a **top-level container**
 - a) exists mainly to provide a place for other Swing components to paint themselves
 - b) The other commonly used top-level containers are dialogs (**JDialog**) and applets (**JApplet**).



Components 2

- 1) The panel is an **intermediate container**
 - a) Its only purpose is to simplify the positioning of the button and label
 - b) Other intermediate Swing containers, such as **scroll panes (JScrollPane)** and **tabbed panes (JTabbedPane)**, typically play a more visible, interactive role in a program's GUI.

- 2) The **button** and **label** are **atomic components**
 - a) self-sufficient entities that present bits of information to the user
 - b) atomic components also get input from the user
 - 1) Combo boxes (**JComboBox**)
 - 2) Text fields (**JTextField**)
 - 3) Tables (**JTable**)

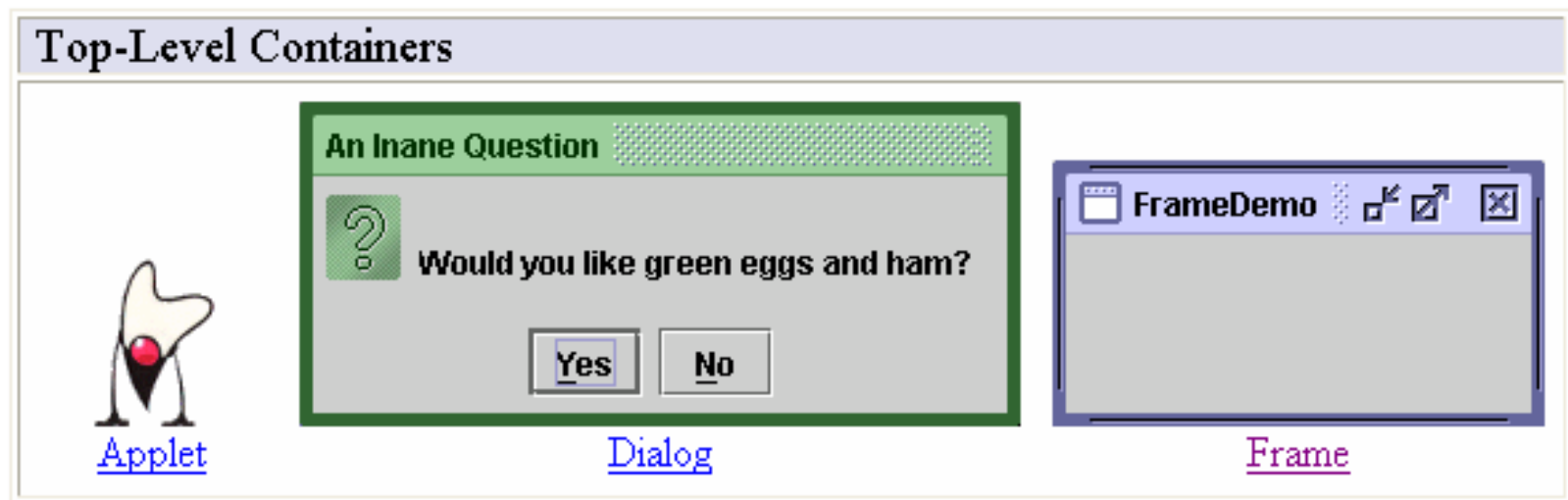
Components 3

Swing Components are divided into the following categories

- 1) Top-level Containers
- 2) General Purpose Containers
- 3) Special Purpose Containers
- 4) Basic Controls
- 5) Uneditable Information Display
- 6) Interactive Display of Highly Formatted Information (IDHFI)

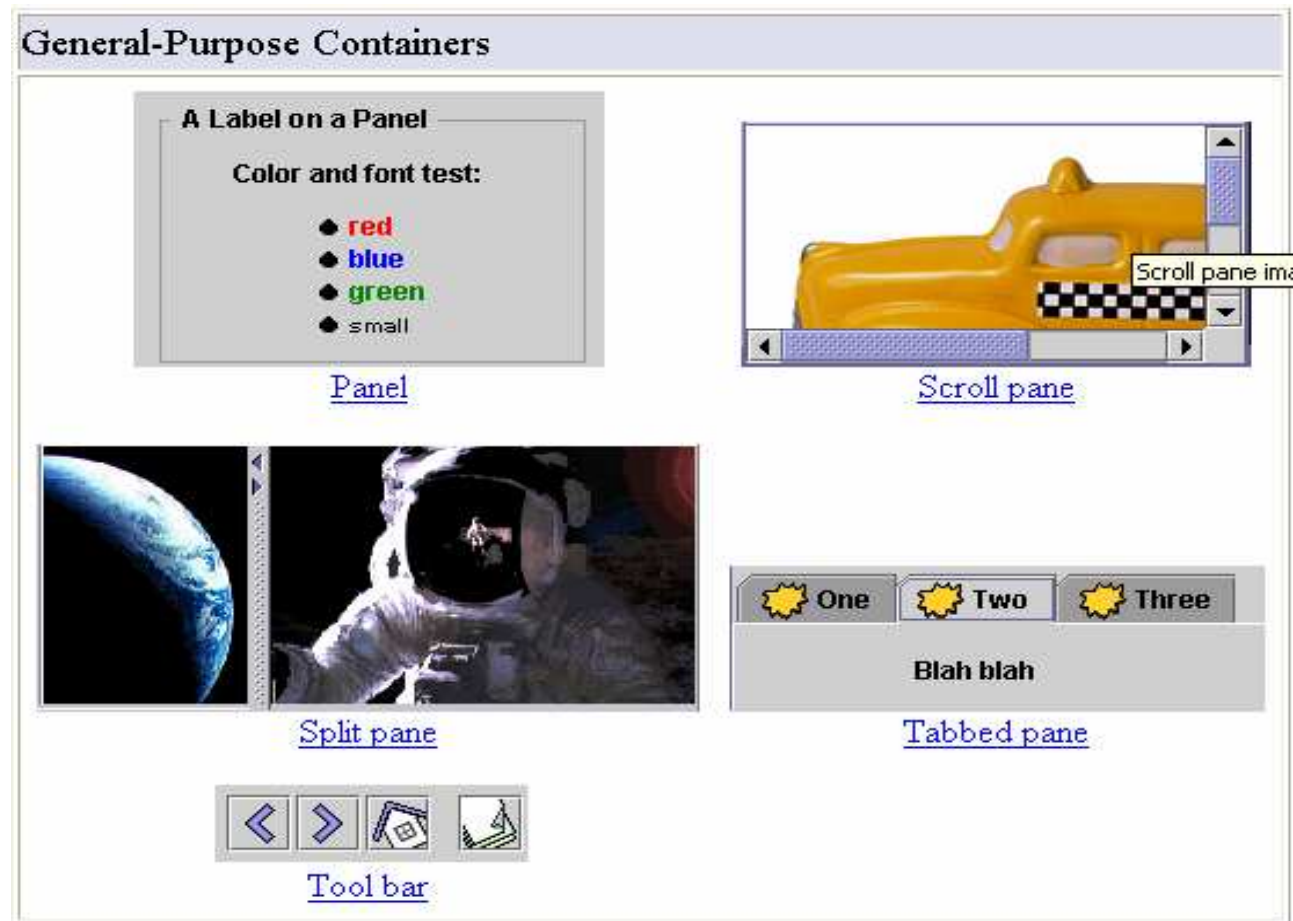
Top-Level Containers

The components at the top of any Swing containment hierarchy



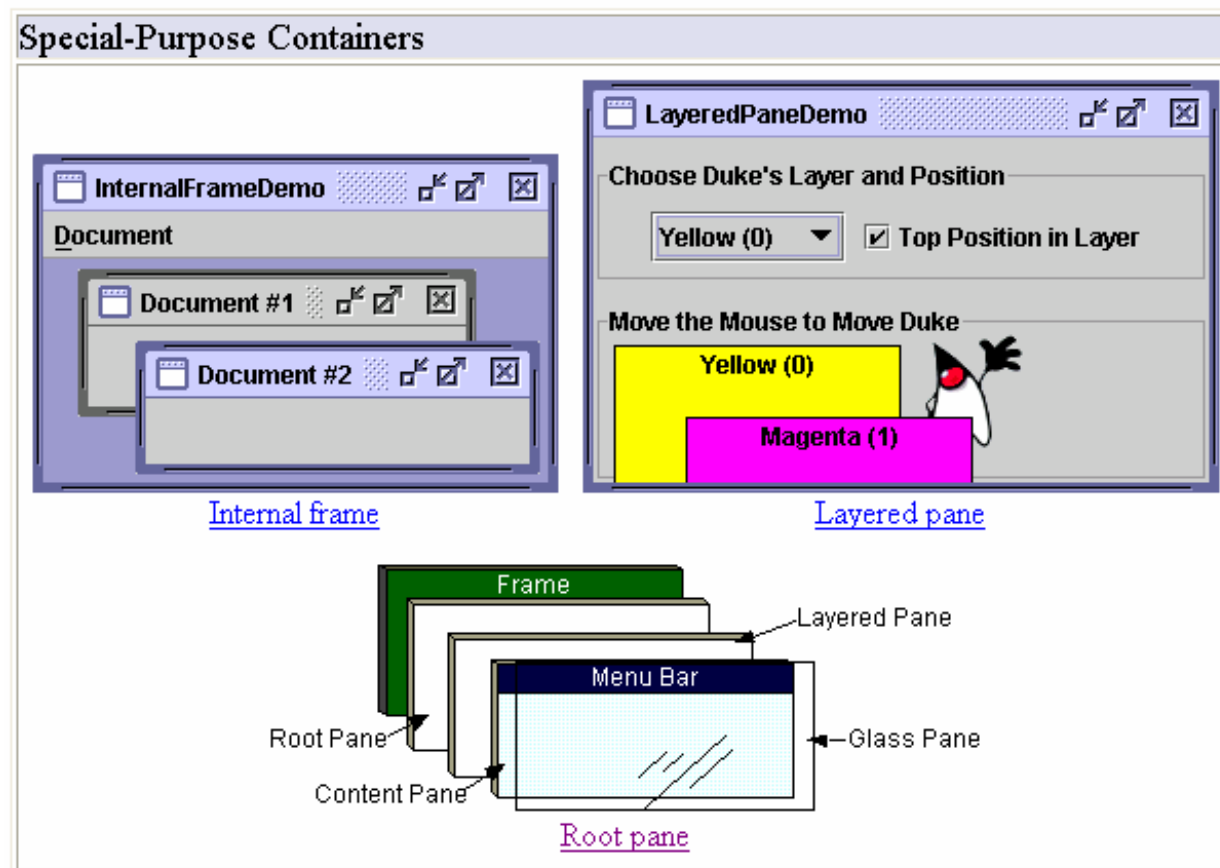
General Purpose Containers

Intermediate containers that can be used under many different circumstances.



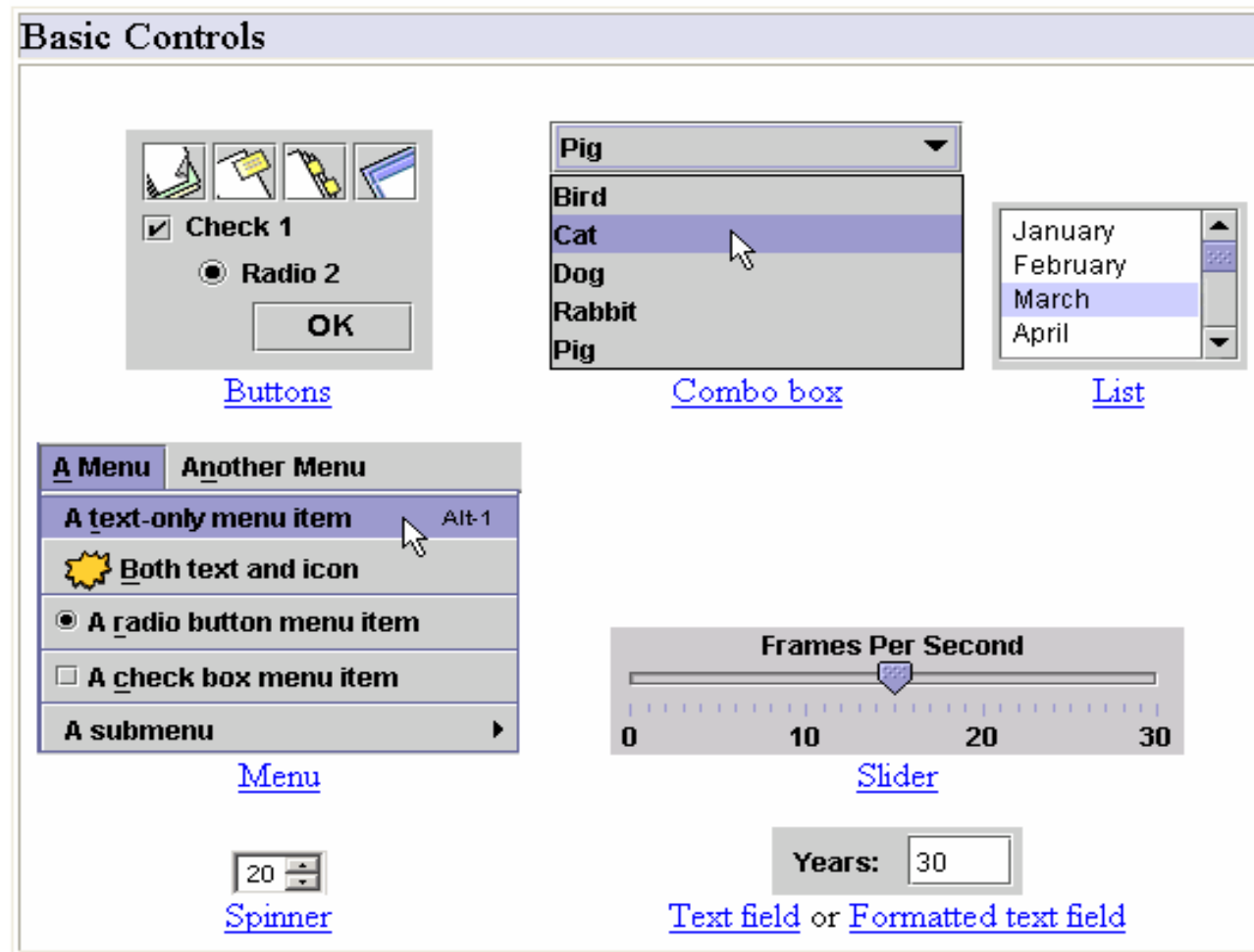
Special Purpose Container

Intermediate containers that play specific roles in the UI.



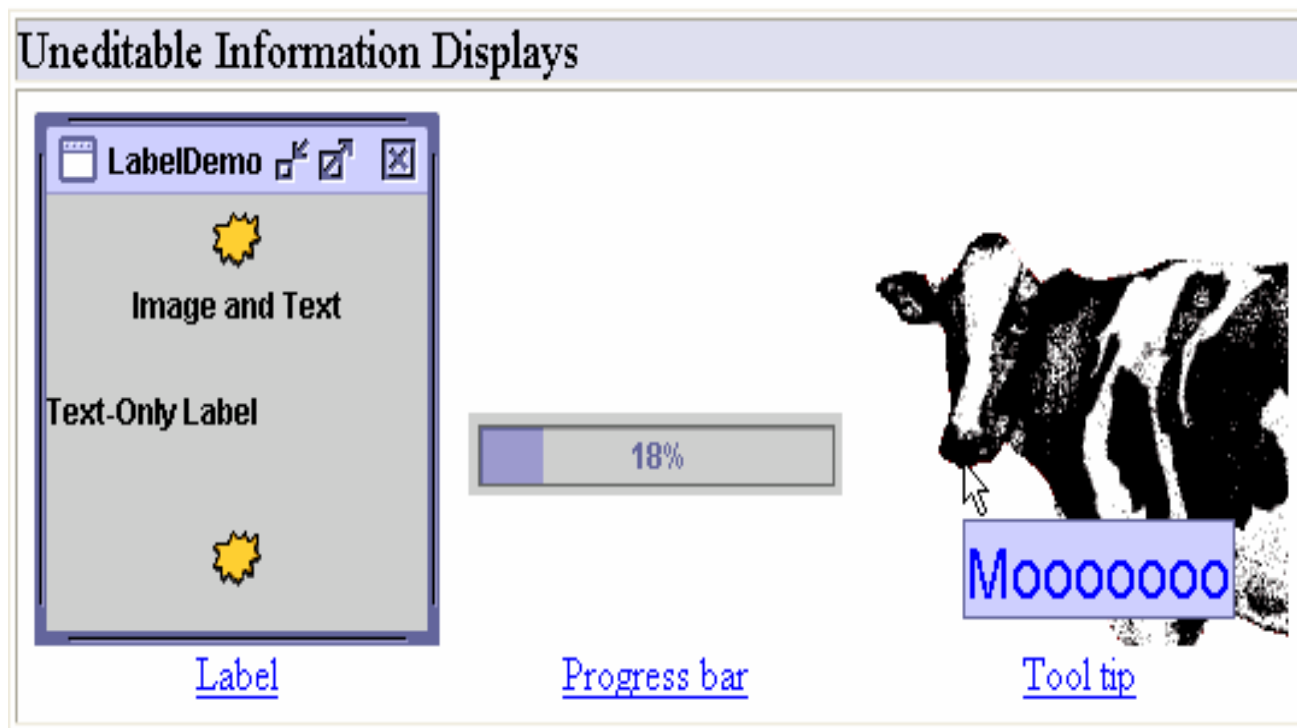
Basic Controls

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



Uneditable Information Display

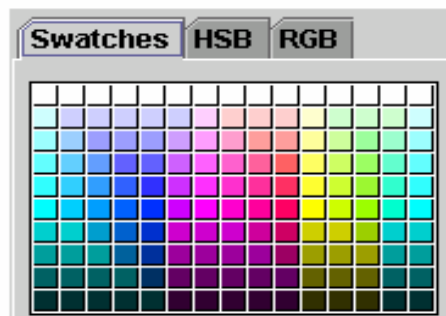
Atomic components that exist solely to give the user information.



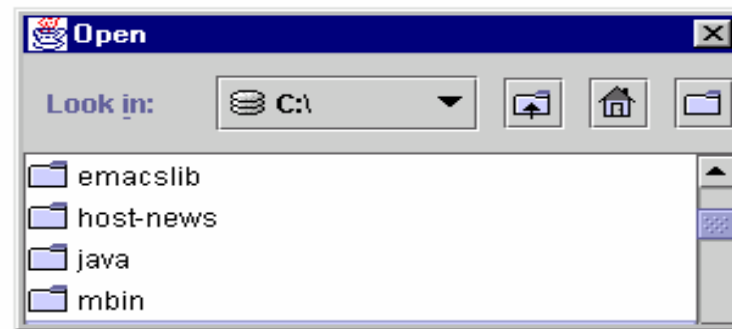
Interactive Displays

Atomic components that display highly formatted information that (if you choose) can be modified by the user.

Interactive Displays of Highly Formatted Information



[Color chooser](#)



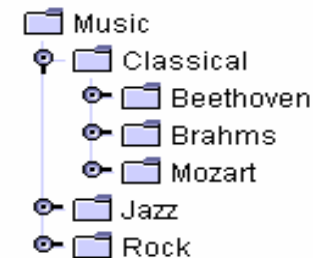
[File chooser](#)

First Name	Last Name	Favorite Food
Jeff	Dinkins	
Ewan	Dinkins	
Amy	Fowler	
Hania	Gajewska	
David	Geary	

[Table](#)



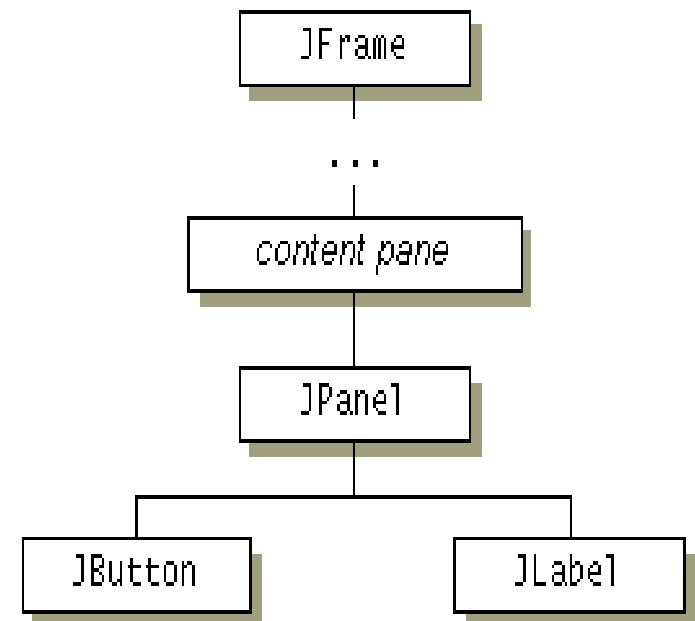
[Text](#)



[Tree](#)

Containment Hierarchy 1

- 1) The diagram shows the **containment hierarchy** for the window shown by **SwingApplication**.
- 2) The root of the containment hierarchy is always a top-level container.
- 3) The top-level container provides a place for its descendent Swing components to paint themselves.



Containment Hierarchy 2

- 1) Every **top-level container** indirectly contains an intermediate container known as a **content pane**.
- 2) the content pane contains, directly or indirectly, all of the visible components in the window's GUI.
- 3) The big exception to the rule is that if the top-level container has a **menu bar**, then by convention the menu bar goes in a special place outside of the **content pane**.

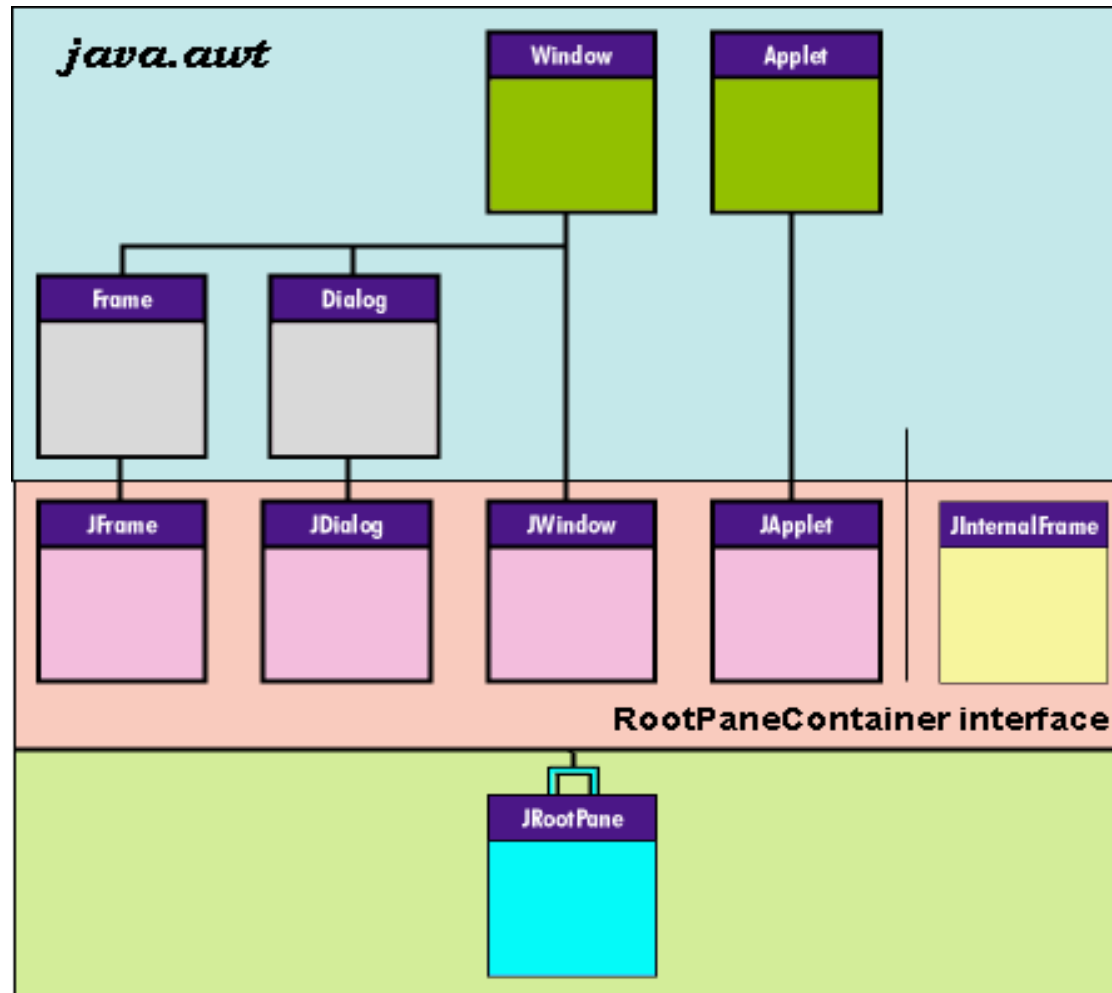
Containment Hierarchy 3

Here is the code that adds the button and label to the panel, and the panel to the content pane

```
frame = new JFrame(...);  
button = new JButton(...);  
label = new JLabel(...);  
pane = new JPanel();  
pane.add(button);  
pane.add(label);  
frame.getContentPane().add(pane, BorderLayout.CENTER);
```

Containment Hierarchy 4

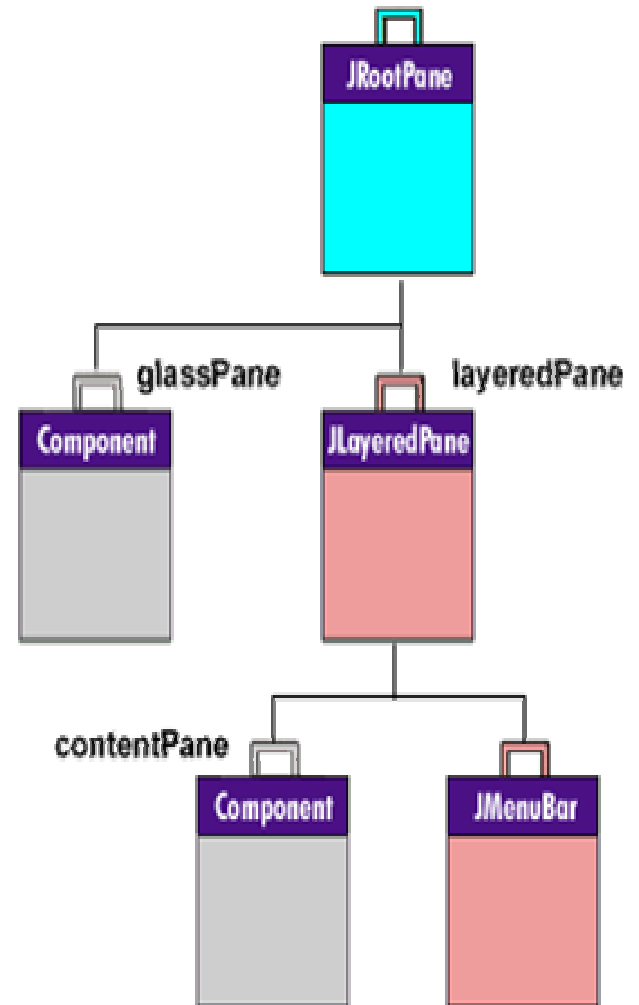
Basic Swing Container Architecture



Containment Hierarchy 5

Architectural Underpinnings

- 1) A **JRootPane** object is made up of
 - a) a glassPane,
 - b) a contentPane, and
 - c) an optional menu bar
- 2) the layeredPane, contentPane, and glassPane shown in this diagram always exist
- 3) What happens when you invoke `getContentPane()` method of a top-level container?



Containment Hierarchy 6

- 1) Take for example a **JFrame**.
- 2) **JFrame** delegates the operation to it's container. But how?
- 3) Consider the inheritance and implementation hierarchy

`javax.swing`

Class JFrame

```
java.lang.Object
└─ java.awt.Component
    └─ java.awt.Container
        └─ java.awt.Window
            └─ java.awt.Frame
                └─ javax.swing.JFrame
```

All Implemented Interfaces:

Accessible, ImageObserver, MenuContainer, RootPaneContainer, Serializable, WindowConstants

Containment Hierarchy 7

public interface **RootPaneContainer**

Method Summary

Container	getContentPane () Returns the contentPane.
Component	getGlassPane () Returns the glassPane.
JLayeredPane	getLayeredPane () Returns the layeredPane.
JRootPane	getRootPane () Return this component's single JRootPane child.
void	setContentPane (Container contentPane) The "contentPane" is the primary container for application specific components.
void	setGlassPane (Component glassPane) The glassPane is always the first child of the rootPane and the rootPanes layout manager ensures always as big as the rootPane.
void	setLayeredPane (JLayeredPane layeredPane) A Container that manages the contentPane and in some cases a menu bar.

Containment Hierarchy 8

Implementation of `getContentPane` method

```
public Container getContentPane() {  
    return getRootPane().getContentPane();  
}
```

`JRootPane` objects, in turn, delegates its `getContentPane()` methods to their `JLayeredPane` instances.

Applet

Course Outline

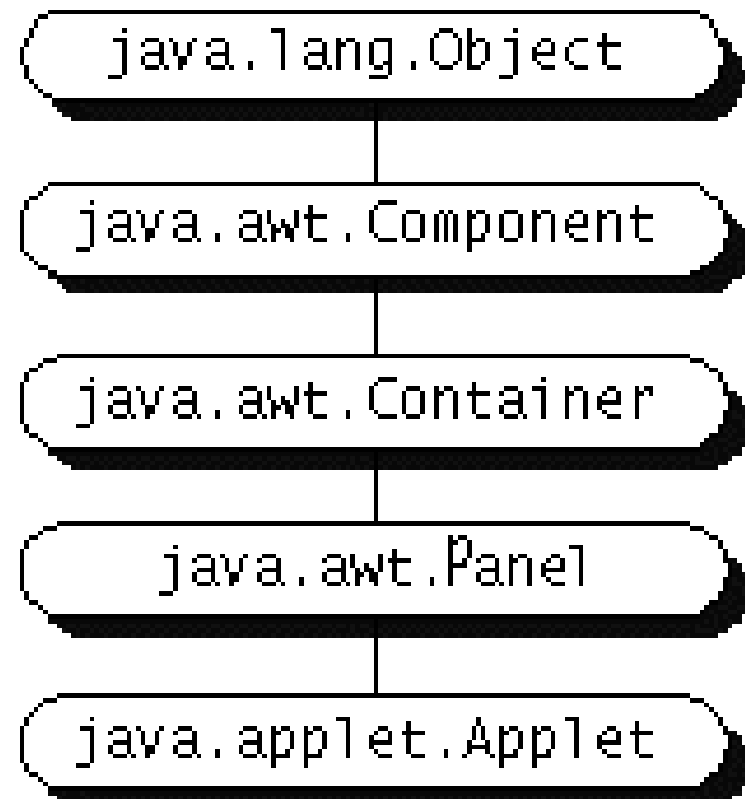
- | | | |
|---|--|--|
| <ul style="list-style-type: none">1) introduction2) language<ul style="list-style-type: none">a) syntaxb) typesc) variablesd) arrayse) operatorsf) control flow | <ul style="list-style-type: none">3) object-orientation<ul style="list-style-type: none">a) objectsb) classesc) inheritanced) polymorphisme) accessf) interfacesg) exception handlingh) multi-threading | <ul style="list-style-type: none">4) horizontal libraries<ul style="list-style-type: none">a) string handlingb) event handlingc) object collections5) vertical libraries<ul style="list-style-type: none">a) graphical interfaceb) appletsc) input/outputd) networking6) summary |
|---|--|--|

What is an Applet?

- 1) Program embedded in another application, generally a Web browser that provides a JVM.
 - a) An applet's host program provides an *applet context* in which the applet executes.
 - b) Launched from an HTML document with an `APPLET` tag that specifies the URL for the applet
- 2) A class that extends Applet or a descendant of the Applet class
 - a) `javax.swing.JApplet` extends Applet

The Hierarchy

- 1) An applet is an object whose class descends ultimately from `Applet` or `JApplet`.
- 2) An applet is an embeddable `Panel`, which is a simple `Container` window.
- 3) An applet's class **must be public**.
- 4) An applet typically overrides the inherited `init`, `start`, `paint`, `stop`, and `destroy` methods.



What does an applet do?

- 1) An applet can react to major events in the following ways:
 - a) It can *initialize* itself
 - `init()` method called when an applet is first loaded
 - b) It can *start* running
 - `start()` method called when browser receives focus
 - c) It can *stop* running.
 - `stop()` method called when the browser minimized or user leaves page
 - d) It can perform a *final cleanup*, in preparation for being unloaded.
 - `destroy()` method called when browser closed

What to override?

- 1) *If you don't override one of the following methods, your applet will not do anything!*
 - a) `init()`
 - Called once. Place “constructor code” here.
 - b) `start()`
 - Usually performs applet's work. Called many times.
 - c) `paint(Graphics g)`
 - Can override to simply draw oneself
 - d) `stop()`
 - Should stop execution of applet (ie: pause a timer thread)
 - e) `destroy()`
 - Performs cleanup of resources (ie: release DB connection)

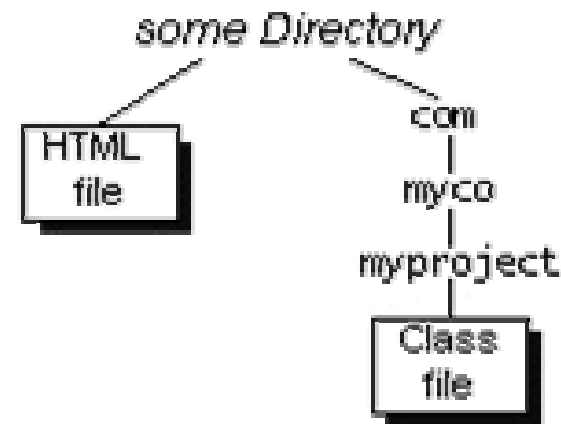
Where to put stuff

- 1) Where the applet class file must be, relative to the HTML document that contains the `<APPLET>` tag.

No Package Statements in Applet Code



`package com.myco.myproject;`



Applet Lifecycle 1

- 1) When a Java-enabled browser encounters an `<APPLET>` tag...
 - a) it reserves a display area of the specified width and height for the applet
 - b) loads the bytecode for the specified Applet subclass,
 - c) creates an instance of the subclass
 - d) invokes the `init` method to enable once-only initialization (e.g., setting colors, fonts, and the like)

Applet Lifecycle 2

a) invokes the `start` method.

- If the applet is multithreaded, other threads can be constructed and started in this method.
- This results in the `paint` method being called. If the page that launched the applet is exited, the browser typically invokes the `stop` method, which then can perform appropriate cleanup operations.

b) If the launching page is entered again, the browser again invokes `start`.

c) `destroy` method

- Ends the lifecycle when the browser window is closed

JApplet Methods

Table 13-3 Some Members of the `class` JApplet (`package` javax.swing)

```
public void init()  
    //Called by the browser or applet viewer to inform this applet  
    //that it has been loaded into the system.  
  
public void start()  
    //Called by the browser or applet viewer to inform this applet  
    //that it should start its execution. It is called after the init  
    //method and each time the applet is revisited in a Web page.  
  
public void stop()  
    //Called by the browser or applet viewer to inform this applet  
    //that it should stop its execution. It is called before the  
    //method destroy.  
  
public void destroy()  
    //Called by the browser or applet viewer. Informs this applet that  
    //it is being reclaimed and that it should destroy any resources  
    //that it has allocated. The method stop is called before destroy.  
  
public void showStatus(String msg)  
    //Displays the string msg in the status bar.  
  
public Container getContentPane()  
    //Returns the ContentPane object for this applet.
```

JApplet Methods

Table 13-3 Some Members of the `class` JApplet (`package` javax.swing) (continued)

```
public JMenuBar getJMenuBar()  
    //Returns the JMenuBar object for this applet.  
  
public URL getDocumentBase()  
    //Returns the URL of the document that contains this applet.  
  
public URL getCodeBase()  
    //Returns the URL of this applet.  
  
public void update(Graphics g)  
    //Calls the paint() method.  
  
protected String paramString()  
    //Returns a string representation of this JApplet; mainly used  
    //for debugging.
```

Applet Methods to Override

- 1) No main method
- 2) Methods init, start, and paint guaranteed to be invoked in sequence
- 3) To develop an applet
 - a) Override any/all of the methods above
 - b) Override stop and destroy to stop threads or free resources (eg: close database connections)

Init and Paint Methods

1) init Method

- a) Initializes variables
- b) Gets data from user
- c) Places various GUI components

2) paint Method

- a) Performs output

Skeleton of a Java Applet

```
import java.awt.Graphics;  
import javax.swing.JApplet;  
  
public class WelcomeApplet extends JApplet  
{  
  
}
```

Displaying Welcome Message

```
//Welcome Applet

import java.awt.Graphics;
import javax.swing.JApplet;

public class WelcomeApplet extends JApplet {

    public void paint(Graphics g) {
        super.paint(g); // <-- make sure you always call this!
        g.drawString("Welcome to Java Programming",30, 30);
    }
}
```

HTML to Run Applet

```
<html>
  <head>
    <title>This title shows at the top of the browser
    window</title>
  </head>
  <body>
    <applet code="WelcomeApplet" height="100" width="400"/>
  </body>
</html>
```


class Font

- 1) Shows text in different fonts
- 2) Contained in package java.awt
- 3) Available fonts
 - a) Serif/SanSerif
 - b) Monospaced
 - c) Dialog/DialogInput
- 4) Arguments for constructor
 - a) String specifying the Font face name
 - b) int value specifying Font style
 - c) int value specifying Font size
 - Expressed in points (72 points = 1 inch)

Methods of the class Font

Table 13-4 Some Constructors and Methods of the `class` Font

Method
<pre>public Font(String name, int style, int size) //Constructor //Creates a new Font from the specified name, style, and point //size.</pre>
<pre>public String getFamily() //Returns the family name of this Font.</pre>
<pre>public String getFontName() //Returns the font face name of this Font.</pre>

class Color

- 1) Shows text in different colors
- 2) Changes background color of component
- 3) Contained in package java.awt

Constructors of the class Color

Table 13-5 Some Constructors and Methods of the `class Color`

```
Color(int r, int g, int b)
    //Constructor
    //Creates an instance of Color with red value r, green value g,
    //and blue value b. In this case, r, g, and b can be
    //between 0 and 255.
    //Example: new Color(0,255,0)
    //      creates a color with no red or blue component.

Color(int rgb)
    //Constructor
    //Creates an instance of Color with red value r, green
    //value g, and blue value b, choose rgb as
    //r * 65536 + g * 256 + b. In this case, r, g, and b
    //can be between 0 and 255.
    //Example: new Color(255)
    //      creates a color with no red or green component.
```

Methods of the class Color

```
Color(float r, float g, float b)
    //Constructor
    //Creates an instance of Color with red value r, green value g,
    //and blue value b. In this case, r, g, and b can be between 0
    //and 1.0.
    //Example: new Color(1.0,0,0)
    //         creates a color with no green or blue component.

public Color brighter()
    //Returns a Color that is brighter.

public Color darker()
    //Returns a Color that is darker.

public boolean equals(Object o)
    //Returns true if the color of this object is the same as the
    //color of the object o; false otherwise.
```

Methods of the class Color

```
public int getBlue()  
    //Returns the value of the blue component.
```

```
public int getGreen()  
    //Returns the value of the green component.
```

```
public int getRGB()  
    //Returns the RGB value. RGB value is  $r * 65536 + g * 256 + b$ .
```

```
public int getRed()  
    //Returns the value of the red component.
```

```
public String toString()  
    //Returns a String with information about the color.
```

Color Constants

Table 13-6 Constants Defined in the `class Color`

<code>Color.black:</code> (0,0,0)	<code>Color.magenta:</code> (255,0,255)
<code>Color.blue:</code> (0,0,255)	<code>Color.orange:</code> (255,200,0)
<code>Color.cyan:</code> (0,255,255)	<code>Color.pink:</code> (255,175,175)
<code>Color.darkGray:</code> (64,64,64)	<code>Color.red:</code> (255,0,0)
<code>Color.gray:</code> (128,128,128)	<code>Color.white:</code> (255,255,255)
<code>Color.green:</code> (0,255,0)	<code>Color.yellow:</code> (255,255,0)
<code>Color.lightGray:</code> (192,192,192)	

class Graphics

- 1) Provides methods for drawing items such as lines, ovals, and rectangles on the screen
- 2) Contains methods to set the properties of graphic elements including clipping area, fonts, and colors
- 3) Contained in the package java.awt

Graphics class

```
protected Graphics()  
    //Constructs a Graphics object that defines a context in which the  
    //user can draw. This constructor cannot be called directly.  
  
public void draw3Rect(int x, int y, int w, int h, boolean t)  
    //Draws a 3D rectangle at (x, y) with width w, height h. If t is  
    //true, rectangle will appear raised.  
  
public abstract void drawArc(int x, int y, int w, int h,  
                             int sangle, int aangle)  
    //Draws an arc in the rectangle at position (x, y) of width w and  
    //height h. The arc starts at angle sangle with an arc angle aangle.  
    //Both angles are measured in degrees.  
  
public abstract boolean drawImage(Image img, int xs1, int ys1,  
                                   int xs2, int ys2, int xd1, int yd1,  
                                   int xd2, int yd2, Color c, ImageObserver ob)  
    //Draws the image specified by img from the area defined by  
    //bounding rectangle, (xs1, ys1) to (xs2, ys2), in the area defined  
    //by the rectangle (xd1, yd1) to (xd2, yd2). Any transparent color  
    //pixels are drawn in color c. The ob monitors the progress of  
    //the image.
```

Graphics class

```
public abstract void drawLine(int xs, int ys, int xd, int yd)
    //Draws a line from (xs, ys) to (xd, yd).
```

```
public abstract void drawOval(int x, int y, int w, int h)
    //Draws an oval at position (x, y) of width w and height h.
```

```
public abstract void drawPolygon(int[] x, int[] y, int num)
    //Draws a polygon with points (x[0], y[0]), ...,
    //(x[num-1], y[num-1]). Here num is the number of points in
    //the polygon.
```

```
public abstract void drawPolygon(Polygon poly)
    //Draws a polygon as defined by the object poly.
```

```
public abstract void drawRect(int x, int y, int w, int h)
    //Draws a rectangle at position (x, y) having a width w and
    //height h.
```

```
public abstract void drawRoundRect(int x, int y, int w, int h,
                                   int arcw, int arch)
    //Draws a round-cornered rectangle at position (x, y) having a
    //width w and height h. The shape of the rounded corners is
    //determined by arc with width arcw and height arch.
```

Graphics class

```
public abstract void drawString(String s, int x, int y)
    //Draws the string s at (x, y).

public void fill3Rect(int x, int y, int w, int h, boolean t)
    //Draws a 3D filled rectangle at (x, y) with width w, height h.
    //If t is true, rectangle will appear raised. The rectangle is
    //filled with current color.

public abstract void fillArc(int x, int y, int w, int h,
                             int sangle, int aangle)
    //Draws a filled arc in the rectangle at position (x, y) of width
    //w and height h starting at angle sangle with an arc
    //angle aangle. Both angles are measured in degrees. The arc is
    //filled with current color.

public abstract void fillOval(int x, int y, int w, int h)
    //Draws a filled oval at position (x, y) having a width w and
    //height h. The oval is filled with current color.

public abstract void fillPolygon(int[] x, int[] y, int num)
    //Draws a filled polygon with points (x[0], y[0]), ...,
    //(x[num-1], y[num-1]). Here num is the number of points in
    //the polygon. The polygon is filled with current color.

public abstract void fillPolygon(Polygon poly)
    //Draws a filled polygon as defined by the object poly. The polygon
    //is filled with current color.

public abstract void fillRect(int x, int y, int w, int h)
    //Draws a filled rectangle at position (x, y) having a width w
    //and height h. The rectangle is filled with current color.
```

Graphics class

```
public abstract void fillRoundRect(int x, int y, int w, int h,  
                                  int arcw, int arch)  
    //Draws a filled, round-cornered rectangle at position (x, y)  
    //having a width w and height h. The shape of the rounded corners  
    //is determined by the arc with width arcw and height arch. The  
    //rectangle is filled with current color.  
  
public abstract Color getColor()  
    //Returns the current color for this graphics context.  
  
public abstract void setColor(Color c)  
    //Sets the current color for this graphics context to c.  
  
public abstract Font getFont()  
    //Returns the current font for this graphics context.  
  
public abstract void setFont(Font f)  
    //Sets the current font for this graphics context to f.  
  
public FontMetrics getFontMetrics()  
    //Returns the font metrics associated with this graphics context.  
  
public FontMetrics getFontMetrics(Font f)  
    //Returns the font metrics associated with Font f.  
  
public void toString()  
    //Returns a string representation of this graphics context.
```

GUI Components and Applets

- 1) Applets inherit from Panel
- 2) Can add UI components
- 3) Call `setLayout(LayoutManager lm)` to change to a different layout
 - a) Default is `BorderLayout`
- 4) Place all adding of UI components in `init()` method
 - a) No longer need to override `paint()` as UI components will automatically update themselves
 - b) Call `validate()` when done adding components

GUI Application vs. Applet

- 1) Applications extend JFrame
- 2) GUI components setup in constructor
- 3) Things start in main method
- 4) Typically call `setTitle`, `setSize`, `pack`, `setVisible`, `setDefaultCloseOperation`

- 1) Applets extend JApplet
- 2) GUI components setup in `init()` method
- 3) No main method, browser creates instance automatically
- 4) Not called in applet, as it doesn't have it's own window

System.out.print and Applets

- 1) When you use System.out to print text, it does not show up in an applet
- 2) Treated as debugging information
- 3) Most browsers have a way to view the output
 - a) IE has a Java console under “Tools” → “Sun Java Console”

Applets and Security

- 1) Tight security called *sandbox security*
- 2) Every browser implements security policies to keep applets from compromising system security.
- 3) The implementation of the security policies differs from browser to browser.
- 4) Security policies are subject to change.
 - a) For example, if a browser is developed for use only in trusted environments, then its security policies will likely be much more lax than those described here.

Sandbox Security?

- 1) In general, applets loaded over the net are prevented from:
 - a) Reading and writing files on the client file system
 - b) Making network connections
 - Except to the originating host
 - c) Starting other programs on the client
 - d) Loading libraries
 - e) Define native method calls
 - If an applet could define native method calls, that would give the applet direct access to the underlying computer.

The SecurityManager

- 1) Each browser has a `SecurityManager` object that implements its security policies.
- 2) When a `SecurityManager` detects a violation, it throws a `SecurityException`.
- 3) Your applet can catch this `SecurityException` and react appropriately.

What can applets do?

- 1) Make network connections to the host they came from.
- 2) Cause HTML documents to be displayed.
- 3) Invoke public methods of other applets on the same page.
- 4) (if loaded from the local file system, i.e. from a directory in the user's CLASSPATH), and Applet does not have the restrictions that applets loaded over the network do
- 5) Play audio clips