Java Programming Language

SL-275

Sun Educational Services

Preface

About This Course

Course Goals

This course provides you with knowledge and skills to:

- Program and run advanced Java™ applications and applets
- Help you prepare for the Sun™ Certified Java Programmer and Developer examinations

Course Overview

This course covers the following areas:

- Syntax of the Java programming language
- Object-oriented concepts as they apply to the Java programming language
- Graphical user interface (GUI) programming
- Applet creation
- Multithreading
- Networking

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Module-by-Module Overview

- Module 1 – Getting Started
- Module 2 – Identifiers, Keywords, and Types
- Module 3 – Expressions and Flow Control
- Module 4 – Arrays
- Module 5 – Objects and Classes
- Module 6 – Advanced Language Features
- Module 7 – Exceptions
- Module 8 – Building GUIs

Module-by-Module Overview

- Module 9 – The AWT Event Model
- Module 10 – The AWT Component Library
- Module 11 – Java Foundation Classes
- Module 12 – Introduction to Java Applets
- Module 13 – Threads
- Module 14 – Stream I/O and Files
- Module 15 – Networking

Course Objectives

- Describe key language features
- Compile and run a Java application
- Understand and use the online hypertext Java technology documentation
- Describe language syntactic elements and constructs
- Understand the object-oriented paradigm and use object-oriented features of the language
- Understand and use exceptions
- Develop a graphical user interface
- Describe the Java technology platform's Abstract Window Toolkit (AWT) used to build GUIs

Course Objectives

- Program to take input from a GUI
- Understand event handling
- Describe the main features of Swing
- Develop Java applets
- Read and write to files and other data sources
- Perform input and output to all sources without the use of a GUI
- Understand the basics of multithreading
- Develop multithreaded Java applications and applets
- Develop Java client and server programs using Transmission Control Protocol/Internet Protocol (TCP/IP) and User Datagram Protocol (UDP)

Skills Gained by Module

Skills Gained	Module 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Describe key language features	■														
Compile and run a Java application	■														
Understand and use the online hypertext Java technology documentation	■														
Describe language syntactic elements and constructs		■	■	■	■	■									
Understand the object-oriented paradigm and use object-oriented features of the language		▒			■	■									
Understand and use exceptions							■								
Develop a GUI								■	■	■	■	▒			
Describe the Java technology platform’s Abstract Window Toolkit used to build GUIs								■		■	■				
Create a program to take input from a graphical user interface									■		■				
Understand event handling										■	■				
Describe the main features of Swing											■				
Develop Java applets												■			
Understand the basics of multithreading													■		
Develop multithreaded Java applications and applets													■		
Read and write to files and other data sources														■	
Perform I/O to all sources without the use of a GUI														■	

Skills Gained	Module														
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Develop Java client and server programs using TCP/IP and UDP															■

Guidelines for Module Pacing

Module	Day 1	Day 2	Day 3	Day 4	Day
About This Course	A.M.				
Module 1 – Getting Started	A.M.				
Module 2 – Identifiers, Keywords, and Types	A.M.				
Module 3 – Expressions and Flow Control	P.M.				
Module 4 – Arrays	P.M.				
Module 5 – Objects and Classes		A.M.			
Module 6 – Advanced Language Features		P.M.			
Module 7 – Exceptions			A.M.		
Module 8 – Building GUIs			A.M.		
Module 9 – The AWT Event Model			P.M.		
Module 10 – The AWT Component Library				A.M.	
Module 11 – Java Foundation Classes				A.M.	
Module 12 – Introduction to Java Applets				P.M.	
Module 13 – Threads					A.M.
Module 14 – Stream I/O and Files					P.M.
Module 15 – Networking					P.M.

Topics Not Covered

- General programming concepts. This is not a course for people who have never programmed before.
- General object-oriented concepts.

How Prepared Are You?

Before attending this course, you should have completed

- SL-110: *Java Programming For Non-Programmers*

or have

- Created compiled programs with C or C++
- Created and edited text files using a text editor
- Used a World Wide Web (WWW) browser, such as Netscape Navigator™

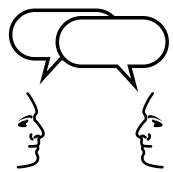
Introductions

- Name
- Company affiliation
- Title, function, and job responsibility
- Programming experience
- Reasons for enrolling in this course
- Expectations for this course

How to Use Course Materials

- Course Map
- Relevance
- Overhead Image
- Lecture
- Exercise
- Check Your Progress
- Think Beyond

Course Icons

- Reference
- Discussion 
- Exercise

Typographical Conventions

- `Courier` – Commands, files and directories, and on-screen computer output
- **`Courier bold`** – Input you type
- *`Courier italic`* – Variables and command-line placeholders
- *Palatino italics* – Book titles, new words or terms, and words that are emphasized

Module 1

Getting Started

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

- Describe key features of Java programming language
- Describe the Java virtual machine's (JVM) function
- Describe how garbage collection works
- List the three tasks performed by the Java platform that handle code security
- Define the terms *class*, *packages*, *applets*, and *applications*

Objectives

- Write, compile, and run a simple Java application
- Use the Java technology application programming interface (API) on-line documentation to identify the methods of the `java.lang` package

Relevance

- Is the Java programming language a complete language or is it just useful for writing programs for the Web?
- Why is another programming language needed?
- How does the Java technology platform improve on other language platforms?

What Is the Java Programming Language?

- The Java programming language is:
  - A programming language
  - A development environment
  - An application environment
  - A deployment environment
- Similar in syntax to C++; similar in semantics to SmallTalk
- Used for developing both *applets* and *applications*

Primary Goals of the Java Programming Language

- Provides an easy-to-use language by:
  - Avoiding the pitfalls of other languages
  - Being object-oriented
  - Enabling users to create streamlined and clear code

Primary Goals of the Java Programming Language

- Provides an interpreted environment for:
  - Improved speed of development
  - Code portability
- Enables users to run more than one thread of activity
- Supports dynamically changing programs during runtime
- Furnishes better security

Primary Goals of the Java Programming Language

The following features fulfill these goals:

- The Java virtual machine (JVM)
- Garbage collection
- Code security

The Java Virtual Machine

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

The Java Virtual Machine

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

The Java Virtual Machine

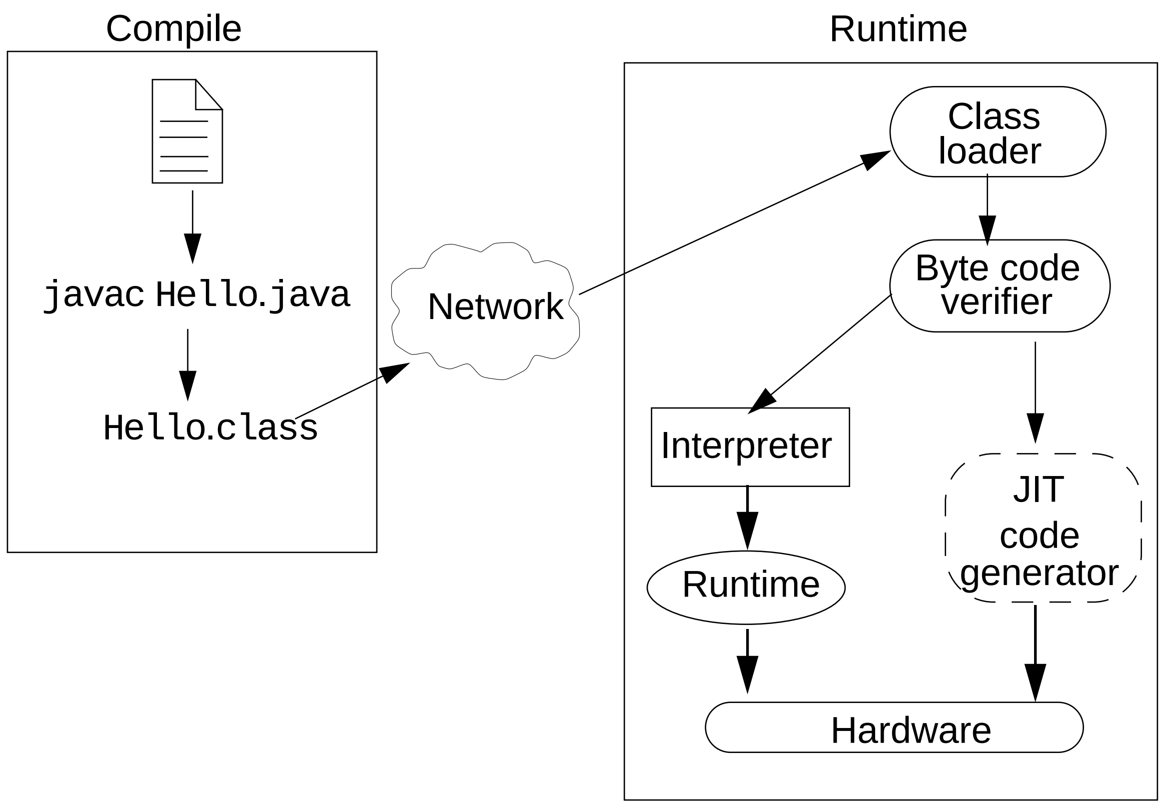
- Bytecodes that maintain proper type discipline from the code.
- The majority of type checking is done when the code is compiled.
- Every Sun approved implementation of the JVM must be able to run any compliant class file.

Garbage Collection

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

Code Security

The Java application environment performs as follows:



Java Runtime Environment

- Performs three main tasks:
  - Loads code
  - Verifies code
  - Executes code

Class Loader

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

Bytecode Verifier

Ensures that:

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

A Basic Java Application

HelloWorldApp.java

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

Compiling and Running `HelloWorldApp`

- Compiling `HelloWorldApp.java`

```
javac HelloWorldApp.java
```

- Running an application

```
java HelloWorldApp
```

- Locating common compile and runtime errors

Compile-Time Errors

- `javac: Command not found`
- `HelloWorldApp.java:6: Method printl(java.lang.String) not found in class java.io.PrintStream.`
  ```
  System.out.printl^("Hello World!");
  ```
- `In class HelloWorldApp :`
  `main must be public and static`

Runtime Errors

- `Can’t find class HelloWorldApp`
- Naming
- One public `class` per file

The Source File Layout

Contains three "top-level" elements:

- An optional package declaration
- Any number of import statements
- Class and interface declarations

Classes and Packages – An Introduction

- Classes and packages:
  - Prominent packages within the Java class library are:
    ```
    java.lang
    java.awt
    java.applet
    java.net
    java.io
    java.util
    ```

Using the Java API Documentation

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

Exercise: Performing Basic Java Tasks

- Exercise objectives:
  - Identify packages, classes, and methods in the Java API documents
  - Identify standard input and output methods
  - Write, compile, and run two simple applications using these methods
- Tasks:
  - Read the documentation
  - Create a Java application
  - Use standard input and output

Check Your Progress

- Describe key features of the Java programming language
- Describe the Java virtual machine's (JVM) function
- Describe how garbage collection works
- List the three tasks performed by the Java platform that handle code security
- Define the terms *class*, *packages*, *applets*, and *applications*
- Write, compile, and run a simple Java application

Check Your Progress

- Use the Java technology API online documentation to identify the methods of the `java.lang` package.

Think Beyond

- How can you benefit from using this programming language in your work environment?

Module 2

Identifiers, Keywords, and Types

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

- Use comments in a source program
- Distinguish between valid and invalid identifiers
- Recognize Java technology keywords
- List the eight primitive types
- Define literal values for numeric and textual types
- Define the terms *class*, *object*, *member variable*, and *reference variable*

Objectives

- Create a class definition for a simple class containing primitive member variables
- Declare variables of class type
- Construct an object using `new`
- Describe default initialization
- Access the member variables of an object using the dot notation
- Describe the significance of a reference variable
- State the consequences of assigning variables of class type

Relevance

- What is your understanding of a class?
- What is your understanding of an object?

Comments

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

```
// comment on one line

/* comment on one
or more lines */

/** documenting comment */
```

Semicolons, Blocks, and Whitespace

- A *statement* is a single line of code terminated by a semicolon(;):

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

- A *block* is a collection of statements bounded by opening and closing braces:

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

Semicolons, Blocks, and Whitespace

- You can use a *block* in a *class* definition:

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

- You can nest block statements
- Any amount of *whitespace* is allowed in a Java program

Identifiers

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

Examples:

```
identifier
username
user_name
_sys_var1
$change
```

Java Keywords

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

Primitive Types

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

Logical – `boolean`

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

```
boolean truth = true;
```

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

Textual – `char` and `String`

`char`

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

`'a'`	The letter *a*
`'\t'`	A tab
`'\u????'`	A specific Unicode character, ????, is replaced with exactly four hexadecimal digits

Textual – `char` and `String`

`String`

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

```
"The quick brown fox jumps over the lazy dog."
```

- Can be used as follows:

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

Integral – `byte`, `short`, `int`, and `long`

- Uses three forms – Decimal, octal, or hexadecimal

`2`	The decimal value is two.
`077`	The leading zero indicates an octal value.
`0xBAAC`	The leading 0x indicates a hexadecimal value.

- Has a default `int`
- Defines `long` by using the letter *L* or *l*

Integral – `byte`, `short`, `int`, and `long`

- Each of the integral data types have the following range:

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

Floating Point – `float` and `double`

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

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

Floating Point – `float` and `double`

- Floating point data types have the following ranges:

Float Length	Name or Type
32 bits	`float`
64 bits	`double`

Variables, Declarations, and Assignments

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

    int x, y; // declare int variables
    float z = 3.414f;// declare and assign float
    double w = 3.1415;// declare and assign double
    boolean truth = true;// declare and assign boolean
    char c;// declare character variable
    String str;// declare String
    String str1 = "bye";// declare and assign String variable
    c = 'A';// assign value to char variable
    str = "Hi out there!";// assign value to String variable
    x = 6;
    y = 1000;// assign values to int variables
  ...
  }
}
```

Java Coding Conventions

- Classes:

```
class AccountBook

class ComplexVariable
```

- Interfaces:

```
interface Account
```

- Methods:

```
balanceAccount()
addComplex()
```

Java Coding Conventions

- Variables:

```
currentCustomer
```

- Constants:

```
HEAD_COUNT
MAXIMUM_SIZE
```

Understanding Objects

- Reviewing the history of objects
- Creating a new type, such as `MyDate`:

```
public class MyDate {
  int day;
  int month;
  int year;
}
```

- Declaring a variable:

```
MyDate myBirth, yourBirth
```

- Accessing members:

```
myBirth.day = 26;
myBirth.month = 11;
yourBirth.year = 1960;
```

Creating an Object

- Declaration of primitive types allocates memory space
- Declaration of nonprimitive types does *not* allocate memory space
- Declared variables are not the data itself, but references (or pointers) to the data

Creating an Object – Memory Allocation and Layout

- A declaration allocates storage only for a reference:

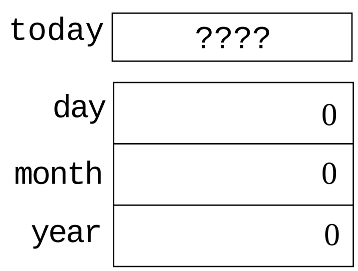
```
MyDate today;
today = new MyDate();
```

```
today [ ???? ]
```

Creating an Object – Memory Allocation and Layout

- Use the `new` operator to allocate and initialize storage:

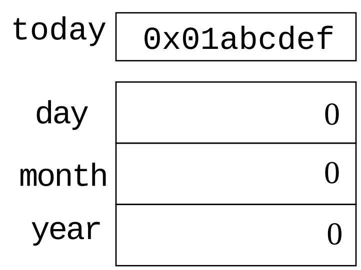
```
MyDate today;
today = new MyDate();
```



Creating an Object – Memory Allocation and Layout

- Assign newly created object to reference variable:

```
MyDate today;
today = new MyDate();
```



Assignment of Reference Variables

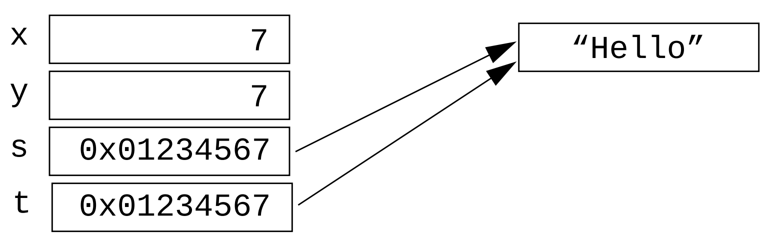
- Consider the following code fragment:

```
int x = 7;
int y = x;
String s = “Hello”;
String t = s;
```

Assignment of Reference Variables

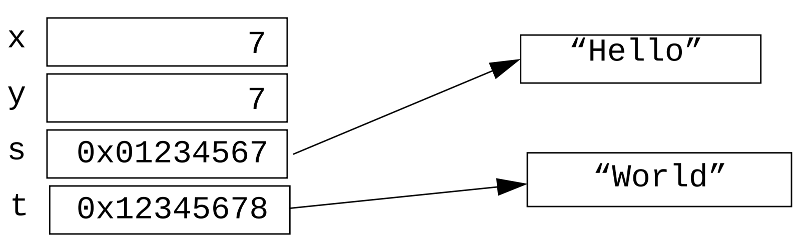
```
int x = 7;
int y = x;
String s = “Hello”;
String t = s;
```

- Two variables refer to single object




```
t = "World";
```

- Reassignment makes two variables point to two objects



Terminology Recap

- Class
- Object
- Reference type
- Member

Exercise: Using Identifiers, Keywords, and Types

- Exercise objectives:
  - Using the correct Java keywords, create a class and an object from the class
  - Compile and run the program
  - Verify that the references are assigned and manipulated as described in this module
- Tasks:
  - Create a class and corresponding objects
  - Investigate reference assignments

Check Your Progress

- Use comments in a source program
- Distinguish between valid and invalid identifiers
- Recognize Java technology keywords
- List the eight primitive types
- Define literal values for numeric and textual types
- Define the terms *class*, *object*, *member variable*, and *reference variable*

Check Your Progress

- Create a class definition for a simple class containing primitive member variables
- Declare variables of class type
- Construct an object using `new`
- Describe default initialization
- Access the member variables of an object using the dot notation
- Describe the significance of a reference variable
- State the consequences of assigning variables of class type

Think Beyond

- What classes and objects appear in your existing applications?

Module 3

Expressions and Flow Control

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

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

Objectives

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

Relevance

- What types of variables are useful to programmers?
- Can multiple classes have variables with the same name and, if so, what is their scope?
- What types of control structures are used in other languages? What methods do these languages use to control flow?

Variables and Scope

Local variables are:

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

Variable Initialization

Variable	Value
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`
`char`	`'\u0000' (NULL)`
`boolean`	`false`
All reference types	`null`

Operators

Separator	. [] () ; ,

<table>
<tr><td>R to L</td><td>++ -- + - ~ ! (data type)</td></tr>
<tr><td>L to R</td><td>* / %</td></tr>
<tr><td>L to R</td><td>+ -</td></tr>
<tr><td>L to R</td><td><< >> >>></td></tr>
<tr><td>L to R</td><td>< > <= >= instanceof</td></tr>
<tr><td>L to R</td><td>== !=</td></tr>
<tr><td>L to R</td><td>&</td></tr>
<tr><td>L to R</td><td>^</td></tr>
<tr><td>L to R</td><td>|</td></tr>
<tr><td>L to R</td><td>&&</td></tr>
<tr><td>L to R</td><td>||</td></tr>
<tr><td>R to L</td><td>?:</td></tr>
<tr><td>R to L</td><td>= *= /= %= += -= <<=
>>= >>>= &= ^= |=</td></tr>
</table>

Logical Expressions

- The Boolean operators supported are:

```
! fl– NOT              & fl– AND
| – OR                 ^ – XOR
```

- The *bitwise* operators are:

```
~ – Complement         & – AND
| – OR                 ^ – XOR
```

- The bitwise operators can work with two Boolean operands

Short -Circuit Logical Operators

- The operators are && (*AND*) and `||` (*OR*)
- Operators can be used as follows:

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

String Concatenation With +

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

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

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

Right-Shift Operators >> and >>>

- *Arithmetic* or *signed* right shift (>>) is used as follows:

```
128 >> 1 returns 128/2^1 = 64
256 >> 4 returns 256/2^4 = 16
-256 >> 4 returns -256/2^4 = -16
```

  - The sign bit is copied during the shift.
- *A logical* or *unsigned right shift* operator (>>>) is:
  - Used for bit patterns
  - Not copied during the shift

Left-Shift Operator (<<)

- Left-shift works as follows:

```
128 << 1 returns 128 * 2^1 = 256
16  << 2 returns 16 * 2^2 = 64
```

Casting

- If information is lost in an assignment, the programmer must confirm the assignment with a typecast.
- The assignment between `short` and `char` requires an explicit cast.

```
long bigValue = 99L;
int squashed = (int)(bigValue);

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

Promotion and Casting of Expressions

- Variables are automatically promoted to a longer form (such as `int` to `long`).
- Expression is *assignment compatible* if the variable type is at least as large (the same number of bits) as the expression type.

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

Branching Statements

The `if`, `else` statements:

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


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

Branching Statements

The `if`, `else` statements:

```
int count;
count = getCount(); // a method defined in the program
if (count < 0) {
   System.out.println("Error: count value is negative.");
} else {
   System.out.println("There will be " + count +
                          " people for lunch today.");
}
```

Branching Statements

The `switch` statement:

The `switch` statement syntax is:

```
switch (expr1) {
  case constant2:
    statements;
    break;
  case constant3:
    statements;
    break;
  default:
    statements;
    break;
}
```

Branching Statements

The `switch` statement:

```
int colorNum = 0;

switch (colorNum) {
  case 0:
    setBackground(Color.red);
    break;
  case 1:
    setBackground(Color.green);
    break;
  default:
    setBackground(Color.black);
    break;
}
```

Looping Statements

The `for` statement:

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

Example:

```
for (int i = 0; i < 10; i++) {
    System.out.println("Are you finished yet?");
}
System.out.println("Finally!");
```

Looping Statements

The `while` loop:

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

Example:

```
int i = 0;

while (i < 10) {
  System.out.println("Are you finished yet?");
  i++;
}
System.out.println("Done");
```

Looping Statements

The `do/while` statement:

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

Example:

```
int i = 0;

do {
  System.out.println("Are you finished yet?");
  i++;
} while (i < 10);
System.out.println("Done");
```

Special Loop Flow Control

```
• break [label];

• continue [label];

• label: statement; // Where statement should
                   // be a loop
```

Special Loop Flow Control

The `break` statement:

```
do {
   statement or block;
   if (condition is true)
     break;
} while (boolean expression);
```

Special Loop Flow Control

The `continue` statement:

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

Special Loop Flow Control

Using `break` with labels:

```
loop:
      do {
         statement;
         do {
            statement;
            statement;
            if (boolean expression)
               break loop;
         } while (boolean expression)
         statement;
      } while (boolean expression);
```

Special Loop Flow Control

Using `continue` with labels:

```
test:
     do {
        statement;
        do {
           statement;
           statement;
        if (condition is true)
           continue test;
        } while (condition is true)
          statement;
     } while (condition is true);
```

Exercise: Using Expressions

- Exercise objective:
  - Write, compile, and run two arithmetic programs that use identifiers, expressions, and control structures
- Tasks:
  - Use factorial application
  - Create a geometry program

Check Your Progress

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

Check Your Progress

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

Think Beyond

- What data types do most programming languages use to group similar data elements together?
- How do you perform the same operation on all elements of a group (for example, a matrix)?
- What data types does the Java programming language use?

Module 4

Arrays

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

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

Relevance

- What is the purpose of an array?

Declaring Arrays

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

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

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

- Create space for a reference
- Remember an array is an object not memory reserved for primitive types

Creating Arrays

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

```
s = new char[20];
p = new Point[100];



p[0] = new Point();
p[1] = new Point();
.
.
.
```

Initializing Arrays

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

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

Myclass array[] = {
  new Myclass(),
  new Myclass(),
  new Myclass()
};

Color palette[] = {
  Color.blue,
  Color.red,
  Color.white
};
```

Multi-Dimensional Arrays

- Arrays of arrays:

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

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

Multi-Dimensional Arrays

- Non-rectangular arrays of arrays:

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

- Array of four arrays of five integers each:

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

Array Bounds

All array subscripts begin at 0:

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

Array Resizing

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

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

Copying Arrays

The `System.arraycopy()` method:

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

// new larger array
int hold[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

// copy all of the elements array to the hold
// array, starting with the 0th index
System.arraycopy(elements, 0, hold, 0, elements.length);
```

Exercise: Using Arrays

- Exercise objectives:
  - Define and initialize an array
  - Write a program that defines, initializes, and uses arrays
- Tasks:
  - Use a basic array
  - Create an array of arrays
  - Create an anagram game

Check Your Progress

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

Think Beyond

- How can you create a three-dimensional array?
- What is one disadvantage of using arrays?

Module 5

Objects and Classes

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

- Define *encapsulation*, *polymorphism*, and *inheritance*
- Use the access modifiers `private` and `public`
- Develop a program segment to create and initialize an object
- Invoke a method on a particular object
- Describe constructor and method overloading
- Describe the purpose of the `this` reference

Objectives

- Discuss why Java application code is reusable.
- In a Java program, identify the following:
  - The `package` statement
  - The `import` statement
  - Classes, member functions, and variables
  - Constructors
  - Overloaded methods
  - Overriden methods
  - Parent class constructors

Relevance

- The elements of the Java programming language covered so far exist in most languages regardless of whether they are object-oriented.
- What features does the Java programming language posess that make it an object-oriented language?
- What does the term *object-oriented* mean?

Object Fundamentals

- Key features:
  - Encapsulation
  - Polymorphism
  - Inheritance
- Abstraction
- Classes and objects

Classes and Objects

- A class is a template or model.
- An object is created based on that model.
- There is one copy of a class per program, but many objects (*instantiate* using the `new` keyword).
- Methods define the operations for a class.
- Methods must belong to a class.

Classes and Objects

```
class EmpInfo {
  String name;
  String designation;
  String department;
}

// Create instance
EmpInfo employee = new EmpInfo();

// Initializes the three members
employee.name = "Robert Javaman″;   employee.designation = ″Manager″;
employee.department = ″Coffee Shop″;

System.out.println(employee.name + “ is “ +
                     employee.designation + “ at “ +
                     employee.department);
```

Classes and Objects

```
public class MyDate {
  int day, month, year;

  public void tomorrow() {
    // code to increment day
  }
}
```

```
MyDate d = new MyDate();
d.tomorrow();
int i = d.day;
```

Defining Methods

The method declaration takes the following form:

```
<modifiers> <return_type> <name> (
    [<argument_list>])
    [throws <exception>] {
  < block >
}
```

Example:

```
public int addDays(int days) {
  < block > // Method code here
}
```

Pass-by-Value

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

The `this` Reference

```
public class MyDate {
  int day, month, year;

  public void tomorrow() {
    this.day = this.day + 1;
    // wrap around code...
  }
}
```

Data Hiding

```
public class MyDate {
  private int day, month, year;

  public void tomorrow() {
    this.day = this.day + 1;
    // validate day range
  }
}


public class DateUser {
  public static void main(String args[]) {
    MyDate mydate = new MyDate();
    mydate.day = 21; // illegal!
  }
}
```

Data Hiding

```
// Part of MyDate class
public void setDay(int targetDay) {
 if (targetDay > this.daysInMonth()) {
    System.err.println("invalid day " + targetDay);
 }
 else {
    this.day = targetDay;
 }
}
```

Encapsulation

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

Overloading Method Names

- It can be used as follows:

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

- Argument lists *must* differ.
- Return types *can* be different.

Constructing and Initializing Objects

- Calling `new Xxxx()` to allocate space for the new object results in:
  - Space for the new object is allocated and initialized to 0 or null.
  - Explicit initialization is performed.
  - A *constructor* is executed.

Explicit Member Initialization

```
public class Initialized {
  private int x = 5;
  private String name = "Fred";
  private MyDate created = new MyDate();

  // Accessor methods go here
  ...
}
```

Constructors

- The method name must exactly match the classname.
- There must not be a return type declared for the method.

Constructors

```
public class Xyz {
  // member variables go here

  public Xyz() {
    // set up the object
  }

  public Xyz(int x) {
    // set up the object with a parameter
  }
}
```

Invoking Overloaded Constructors

```
public class Employee {
  private String name;
  private int salary;

  public Employee(String n, int s) {
    name = n;
    salary = s;
  }

  public Employee(String n) {
    this(n, 0);
  }

  public Employee() {
    this(“Unknown”);
  }
}
```

The Default Constructor

- Is in every class
- Enables you to create object instances with `new Xxx()`
- Is invalid if you add a constructor declaration with arguments

The `is a` Relationship

The `Employee` class:

```
public class Employee {
  String name;
  Date hireDate;
  Date dateOfBirth;
}
```

The `is a` Relationship

- The `Manager` class:

```
public class Manager {
  String name;
  Date hireDate;
  Date dateOfBirth;
  String department;
  Employee subordinates [];
}
```

- *Subclassing*

The `extends` Keyword

```
public class Employee {
  String name;
  Date hireDate;
  Date dateOfBirth;
}

public class Manager extends Employee {
  String department;
  Employee subordinates [];
}
```

Single Inheritance

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

Single Inheritance

Employee

attributes
name
address
salary
methods
up_salary
promote

← Engineer

← Secretary

← Manager ← Director

Manager:
attributes
bonus
methods
up_bonus

Director:
attributes
car allowance
methods
up_allowance
promote

Inheritance examples

Constructors Are Not Inherited

- A subclass inherits all methods and variables from the superclass (parent class).
- A subclass does not inherit the constructor from the superclass.
- Two ways to include a constructor are:
  - Use the default constructor
  - Write one or more explicit constructors

Polymorphism

- *Polymorphism* is the ability to have many different forms; for example, the Manager class has access to methods from Employee class.
- An object has only one form.
- A reference variable has many forms; it can refer to objects of different forms.

Polymorphism

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


// Illegal attempt to assign Manager member
// variable when object is a parent Employee class
e.department = "Finance";
```

Heterogeneous Collections

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

Heterogeneous Collections

- Because a `Manager` is an `Employee`:

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

- An example of heterogeneous collection is:

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

The `instanceof` Operator

```
public class Employee extends Object
public class Manager extends Employee
public class Contractor extends Employee
----------------------------------------

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

Casting Objects

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

Overriding Methods

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

Overriding Methods

```
public class Employee {
     String name;
     int salary;
     public String getDetails() {
        return “Name: “ + name + “\n” +
        “Salary: “ + salary;
     }
}

  public class Manager extends Employee {
     String department;

     public String getDetails() {
       return “Name: “ + name + “\n” +
        “Manager of “ + department;
     }
}
```

Overriding Methods

- Virtual method invocation:

```
Employee e = new Manager();
e.getDetails();
```

- Compile-time type and runtime type

Rules About Overridden Methods

- Must have a return type that is identical to the method it overrides
- Cannot be less accessible than the method it overrides
- Must throw exceptions that are same type as the method being overridden

Rules About Overridden Methods

```
public class Parent {
  public void method() {}
}

public class Child extends Parent {
  private void method() {}
}

public class UseBoth {
  public void otherMethod() {
    Parent p1 = new Parent();
    Parent p2 = new Child();
    p1.method();
    p2.method();
  }
}
```

The `super` Keyword

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

The `super` Keyword

```
public class Employee {
   private String name;
   private int salary;
   public String getDetails() {
      return Name: " + name + "\nSalary: " + salary;
   }
}
public class Manager extends Employee {
   private String department;
   public String getDetails() {
      // call parent method
      return super.getDetails() +
      "\nDepartment: " + department;
   }
}
```

Invoking Parent Class Constructors

- Initialization of objects is structured.
- When an object is initialized, the following sequence of events occur:
  - The memory space is allocated and initialized to "zero" values
  - Explicit initialization is performed for each class in the hierarchy
  - A constructor is called for each class in the hierarchy

Invoking Parent Class Constructors

- In many circumstances, the default constructor is used to initialize the parent object.

```
public class Employee {
  String name;
  public Employee(String n) {
    name = n;
  }
}
public class Manager extends Employee {
  String department;
  public Manager(String s, String d) {
    super(s);
    department = d;
  }
}
```

- If used, you must place `super` or `this` in the first line of the constructor.

Packages

- You must specify package declaration at the beginning of the source file.
- You are permitted only one package declaration per source file.

```
// Class Employee of the Finance department for the
// ABC company
package abc.financeDept;

public class Employee {
  ...
}
```

- Package names must be hierarchical and separated by dots.

The `import` Statement

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

```
import abc.financeDept.*;

public class Manager extends Employee {
  String department;
  Employee subordinates [];
}
```

Directory Layout and Packages

- Packages are stored in the directory tree containing the package name.

```
package abc.financedept

public class Employee {
  ...
}

javac -d . Employee.java
```

Exercise: Using Objects and Classes

- Exercise objective:
  - Write, compile, and run three programs that use the object-oriented concepts of inheritance, constructors, and data hiding by modeling a bank account.
- Tasks:
  - Create a bank account
  - Create several account types
  - Create an online account service

Check Your Progress

- Define *encapsulation*, *polymorphism*, and *inheritance*
- Use the access modifiers `private` and `public`
- Develop a program segment to create and initialize an object
- Invoke a method on a particular object
- Describe constructor and method overloading
- Describe the purpose of the `this` reference

Check Your Progress

- Discuss why Java application code is reusable
- In a Java program, identify the following:
  - The `package` statement
  - The `import` statement
  - Classes, member functions, and variables
  - Constructors
  - Overloaded methods
  - Overriden methods
  - Parent class constructors

Think Beyond

- Now that you understand objects and classes, how could you put this to use on a project you are working on?

Module 6

Advanced Language Features

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

- Describe `static` variables, methods, and initializers
- Describe `final` classes, methods, and variables
- List the access control levels
- Identify deprecated classes and explain how to migrate from JDK™ 1.0 to JDK 1.1 to JDK 1.2
- Describe how to apply collections and reflections

Objectives

- In a Java program, identify:
  - `static` methods and variables
  - `public`, `private`, `protected`, and *default* variables
- Use `abstract` classes and methods
- Explain how and when to use inner classes
- Explain how and when to use interfaces
- Describe the difference between `==` and `equals()`

Relevance

- How can you keep a class or method from being subclassed or overriden?
- How can you extend the use of array concepts to objects?

Class (`static`) Variables

- Are shared among all instances of a class
- Can be marked either as `public` or as `private`
- Can be accessed from outside the class if marked as `public` without an instance of the class

```
public class Count {
  private int serialNumber;
  private static int counter = 0;

  public Count() {
    counter++;
    serialNumber = counter;
  }
}
```

Class (`static`) Methods

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

```
public class GeneralFunction {
  public static int addUp(int x, int y) {
    return x + y;
  }
}

public class UseGeneral {
  public void method() {
    int a = 9;
    int b = 10;
    int c = GeneralFunction.addUp(a, b);
    System.out.println("addUp() gives " + c);
  }
}
```

Static Initializers

- A class can contain code in a *static block* that does not exist within a method body.
- Static block code executes only once, when the class is loaded.

Static Initializers

```
public class StaticInitDemo {
  static int i = 5;

  static {
    System.out.println("Static code i= "+ i++ );
  }
}

public class Test {

  public static void main(String args[]) {
    System.out.println("Main code: i=" + StaticInitDemo.i);
  }
}
```

Static Methods and Data

```
public class Car {
  String color;
  String model;

  // Specific to this instance.
  int serialNumber;

  // Accessible by all instances.
  static int nextSerialNumber = 1;

  public Car (String color, String model) {
    this.color = color;
    this.model = model;
    serialNumber = nextSerialNumber++;
  }

  public void whoAmI() {
    System.out.println(
      "I am a " + color + " " + model +
      ", serial number = " + serialNumber);
  }

  public static void main (String args[]) {
    Car JanesCar = new Car("Red", "Coupe");
    Car JoesCar = new Car("Blue", "Hatchback");

    JanesCar.whoAmI();
    JoesCar.whoAmI();
  }
}
```

Access class data:

`nextSerialNumber` or
`Car.nextSerialNumber`

Static Methods and Data

```java
public class Car2 {
  private String color;
  private String model;

  // Specific to this instance.
  private int serialNumber;

  // Accessible by all instances.
  private static int nextSerialNumber = 1;

  public Car2 (String color, String model) {
    this.color = color;
    this.model = model;
    serialNumber = nextSerialNumber++;
  }

  public void whoAmI() {
    System.out.println(
      "I am a " + color + model +
      ", serial number = " + serialNumber);
  }

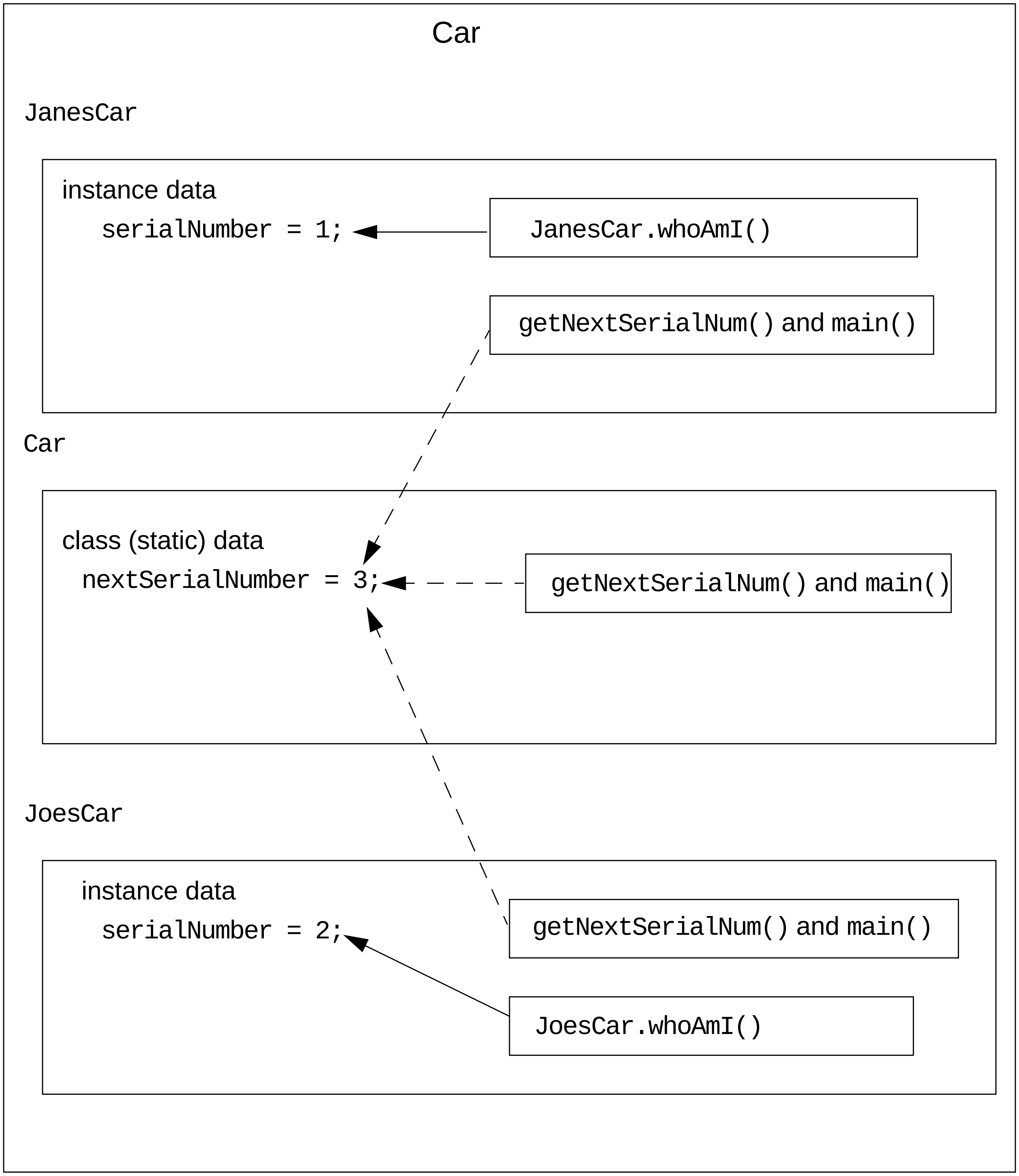
  public static void getNextSerialNum() {
    System.out.println(
        "The next available serial number is " +
        nextSerialNumber);
  }

  public static void main (String args[]) {
    Car2 JanesCar = new Car2("Red", "Coupe");
    Car2 JoesCar = new Car2("Blue", "Hatchback");

    // Use nonstatic method to get instance data
    JanesCar.whoAmI();
    JoesCar.whoAmI();

    // Use static method to get class data
    getNextSerialNum();
  }}                                   // just to fit on page
```

Static Methods and Data



The `final` Keyword

- You cannot subclass a `final` class.
- You cannot override a `final` method.
- A `final` variable is a constant.

Abstract Classes

- A class that declares the existence of methods but not the implementation is called an `abstract` class.
- You can declare a class as abstract by marking it with the `abstract` keyword.

```
public abstract class Drawing {
  public abstract void drawDot(int x, int y);
  public void drawLine(int x1, int y1,
                       int x2, int y2) {
    // draw using the drawDot() method repeatedly.
  }
}
```

- An `abstract` class can contain member variables and non-abstract methods.

Interfaces

- An `interface` is a variation on the idea of an abstract class.
- In an `interface`, all the methods are `abstract.`
- You can simulate multiple inheritance by implementing such interfaces.
- The syntax is:

```
public interface Transparency {

  public static final int OPAQUE=1;
  public static final int BITMASK=2;
  public static final int TRANSLUCENT=3;

  public int getTransparency();
}
```

Interfaces

```
public class MyApplet extends Applet
            implements Runnable, MouseListener {
  "..."
}

interface SayHello {
  void printMessage();
}

class SayHelloImpl implements SayHello {
  void printMessage() {
    System.out.println("Hello");
  }
}
```

Interfaces

- Interfaces are useful for:
  - Declaring methods that one or more classes are expected to implement
  - Determining an object's programming interface without revealing the actual body of the class
  - Capturing similarities between unrelated classes without forcing a class relationship

Advanced Access Control

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

Deprecation

- Deprecation is the obsoletion of class constructors and method calls.
- Obsolete methods and constructors are replaced by methods with a more standardized naming convention.
- When migrating code, compile the code with the `-deprecation` flag:

```
javac -deprecation MyFile.java
```

Deprecation

JDK 1.1 code, before deprecation is as follows:

```
package myutilities;

import java.util.*;
import java.text.*;

public final class DateConverter {
  private static String day_of_the_week [] =
    {"Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"};

  public static String getDayOfWeek (String theDate){
    int month, day, year;

    StringTokenizer st = new StringTokenizer (theDate, "/");

    month = Integer.parseInt(st.nextToken ());
    day = Integer.parseInt(st.nextToken());
    year = Integer.parseInt(st.nextToken());
    Date d = new Date (year, month, day);

    return (day_of_the_week[d.getDay()]);
  }
}
```

Deprecation

Compiling previous code with the `-deprecation` flag yields:

```
% javac -deprecation DateConverter.java
DateConverter.java:16: Note: The constructor java.util.Date(int,int,int) has been
deprecated.
                Date d = new Date (year, month, day);
                         ^
DateConverter.java:18: Note: The method int getDay() in class java.util.Date has
been deprecated.
                return (day_of_the_week[d.getDay()]);
                                                ^
Note: DateConverter.java uses a deprecated API.Please consult the documentation for
a better alternative.
3 warnings
```

Deprecation

A JDK 1.3 version rewritten is:

```
package myutilities;

import java.util.*;
import java.text.*;

public final class DateConverter2 {
  private static String day_Of_The_Week[] =
      {"Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"};

  public static String getDayOfWeek (String theDate) {
    Date d = null;
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yy");

    try {
      d = sdf.parse (theDate);
    } catch (ParseException e) {
      System.out.println (e);
      e.printStackTrace();
    }

    // Create a GregorianCalendar object
    Calendar c =
        new GregorianCalendar(
            TimeZone.getTimeZone("EST"),Locale.US);
    c.setTime (d);

    return(
        day_Of_The_Week[(c.get(Calendar.DAY_OF_WEEK)-1)]);
  }
}
```

The == Operator Versus `equals()` Method

- The `equals()` and == methods determine if reference values refer to the same object.
- The `equals()` method is overridden in classes to return true if the contents and type of two separate objects match.

toString() Method

- Converts an object to a `String`
- Converts a primitive type to a `String`, but uses wrapper classes that have the method
- Overrides to provide information about the object in readable format

Inner Classes

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

Properties of Inner Classes

- You can use the class name only within the defined scope, except when used in a qualified name.

  The name of the inner class must differ from the enclosing class.

- The inner class can be defined inside a method.

  Any variable, either a local variable or a formal parameter, can be accessed by methods within an inner class provided the variable is marked as `final`.

Properties of Inner Classes

- The inner class can use both class and instance variables of enclosing classes and local variables of enclosing blocks.
- The inner class can be defined as `abstract.`
- Only inner classes can be declared as `private` or `protected`.
- An inner class can act as an interface implemented by another inner class.

Properties of Inner Classes

- Inner classes that are declared `static` automatically become top-level classes.
- Inner classes cannot declare any `static` members; only top-level classes can declare `static` members.

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

Wrapper Classes

- Look at primitive data elements as objects

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

Wrapper Classes

```
int pInt = 500;
Integer wInt = new Integer(pInt);

int p2 = wInt.intValue();
```

Collection API

- A *collection* (or a container) is a single object representing a group of objects known as its elements.
- Collection classes `Vector`, `Bits`, `BitSet`, `Stack`, `Hashtable`, `LinkedList`, and so on are supported.
- The Collection API contains interfaces that maintain objects as a:
  - `Collection` – A group of objects with no specific ordering
  - `Set` – A group of objects with no duplication
  - `List` – A group of ordered objects; duplication is permitted

The `Vector` Class

The `Vector` class provides methods for working with dynamic arrays of varied element types.

```
java.lang.Object
 |
 |
 +--java.util.AbstractCollection
          |
          +--java.util.AbstractList
                |
                +--java.util.Vector
```

Synopsis

- Each vector maintains a `capacity` and `capacityIncrement`.
- As elements are added, storage for the vector increases in chunks up to the size of the `capacityIncrement` variable.

Constructors

- `public Vector()`
- `public Vector(int initialCapacity)`
- ```
  public Vector(int initialCapacity,
                int capacityIncrement)
  ```
  ```

Variables

- `protected int capacityIncrement`
- `protected int elementCount`
- `protected Object elementData[]`

Methods

```
public final int size()

public final boolean contains(Object elem)

public final int indexOf(Object elem)

public final synchronized
    Object elementAt(int index)

public final synchronized void
    setElementAt(Object obj, int index)

public final synchronized void
    removeElementAt(int index)

public final synchronized void
    addElement(Object obj)

public final synchronized void
    insertElementAt(Object obj, int index)
```

The Vector Class

```
import java.util.*;

public class MyVector extends Vector {
  public MyVector() {

    // storage capacity & capacityIncrement
    super(1,1);
  }

  public void addInt(int i) {

    // addElement requires Object arg
    addElement(new Integer(i));
  }

  public void addFloat(float f) {
    addElement(new Float(f));
  }

  public void addString(String s) {
    addElement(s);
  }

  public void addCharArray(char a[]) {
    addElement(a);
  }
```

The `Vector` Class

```
  public void printVector() {
    Object o;

    // compare with capacity()
    int length = size();
    System.out.println("Number of vector elements is " +
        length + " and they are:");

    for (int i = 0; i < length; i++) {
      o = elementAt(i);

      if (o instanceof char[]) {

        // An array's toString() method does not print
        // what we want.
        System.out.println(String.copyValueOf((char[]) o));
      } else {
        System.out.println(o.toString());
      }
    }
  }

  public static void main(String args[]) {
    MyVector v = new MyVector();
    int digit = 5;
    float real = 3.14F;
    char letters[] = { 'a', 'b', 'c', 'd' };
    String s = new String("High there!");

    v.addInt(digit);
    v.addFloat(real);
    v.addString(s);
    v.addCharArray(letters);

    v.printVector();
  }
}
```

Reflection API

Can be used to:

- Construct new class instances and new arrays
- Access and modify fields of objects and classes
- Invoke methods on objects and classes
- Access and modify elements of arrays

Reflection API Features

```
java.lang.Class
java.lang.reflect.Field
java.lang.reflect.Method
java.lang.reflect.Array
java.lang.reflect.Constructor
```

Reflection API Security Model

- The Java Security Manager controls access to the core Reflection API on a class-by-class basis.
- Standard Java programming language access control is enforced when:
  - A `Field` is used to get or set a field value
  - A `Method` is used to invoke a method
  - A `Constructor` is used to create and initialize a new instance of a class

Exercise: Working With Advanced Language Features

- Exercise objective:
  - Rewrite, compile, and run three programs that use the bank account model and employ advanced object-oriented features, such as inner classes, vector classes, and interfaces
- Tasks:
  - Modify the bank account
  - Use inner classes
  - Add `find` and `delete` methods to `MyVector` class

Check Your Progress

- Describe `static` variables, methods, and initializers
- Describe `final` classes, methods, and variables
- List the access control levels
- Identify deprecated classes and explain how to migrate from JDK 1.0 to JDK 1.1 to JDK 1.2
- Describe how to apply collections and reflections

Check Your Progress

- In a Java program, identify:
  - `static` methods and variables
  - `public`, `private`, `protected`, and default variables
- Use `abstract` classes and methods
- Explain how and when inner classes are used
- Explain how and when interfaces are used
- Describe the difference between `==` and `equals()`

Think Beyond

- What features of the Java programming language are used to deal with runtime error conditions?

Module 7

Exceptions

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

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

Relevance

- In most programming languages, how do you resolve runtime errors?

Exceptions

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

Exception Example

```
public class HelloWorld {
  public static void main (String args[]) {
    int i = 0;

    String greetings [] = {
      "Hello world!",
      "No, I mean it!",
      "HELLO WORLD!!"
    };

    while (i < 4) {
      System.out.println (greetings[i]);
      i++;
    }
  }
}
```

`try` and `catch` Statements

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

`Call Stack` Mechanism

- If an exception is not handled in the current try/catch block, it is thrown to the caller of that method.
- If the exception gets back to the main method and is not handled there, the program is terminated abnormally.

`finally` Statement

```
try {
  startFaucet();
  waterLawn();
} finally {
  stopFaucet();
}
```

Exception Example Revisited

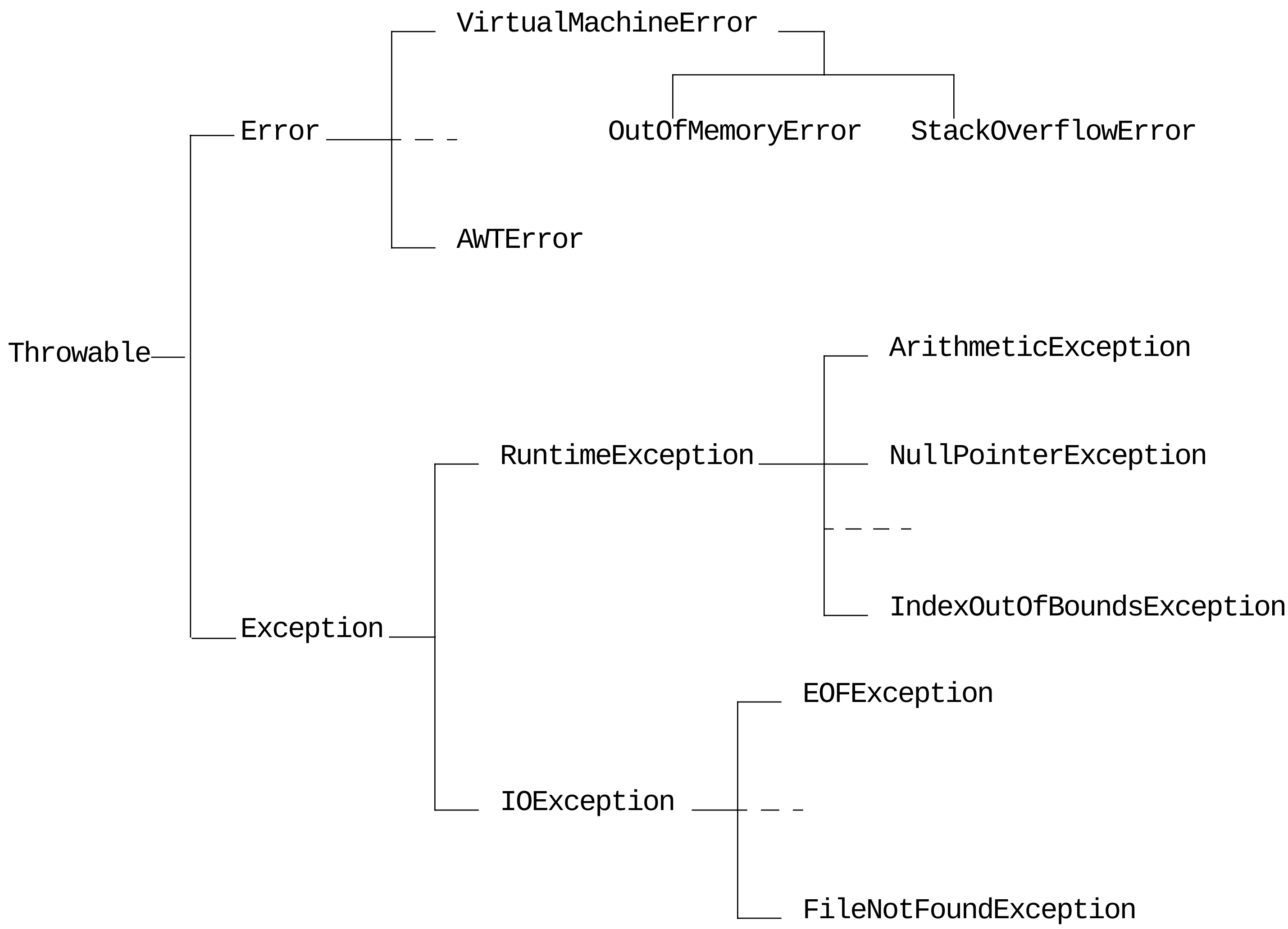
```
public class HelloWorld2 {
  public static void main (String args[]) {
    int i = 0;

    String greetings [] = {
      "Hello world!",
      "No, I mean it!",
      "HELLO WORLD!!"
    };

    while (i < 4) {
      try {
        System.out.println (greetings[i]);
      } catch (ArrayIndexOutOfBoundsException e){
        System.out.println("Re-stting Index Value");
        i = -1;
      } finally {
        System.out.println("This is always printed");
      }

      i++;
    }
  }
}
```

Exception Categories



Common Exceptions

- `ArithmeticException`
- `NullPointerException`
- `NegativeArraySizeException`
- `ArrayIndexOutOfBoundsException`
- `SecurityException`

The Handle or Declare Rule

- Handle the exception by using the `try-catch-finally` block.
- Declare that the code causes an exception by using the `throws` clause.

Creating Your Own Exceptions

```
public class ServerTimedOutException extends Exception {
  private int port;

  public ServerTimedOutException(String reason, int port) {
    super(reason);
    this.port = port;
  }

  // Use Exception class`s getMessage() to get the
  // reason the exception was madE

  public int getPort() {
    return port;
  }
}
```

Handling User-Defined Exceptions

```
public void connectMe(String serverName) throws ServerTimedOutException {
  int success;
  int portToConnect = 80;
  success = open(serverName, portToConnect);
  if (success == -1) {
    throw new ServerTimedOutException("Could not connect", 80);
  }
}
```

```
public void findServer() {
    try {
      connectMe(defaultServer);
    } catch (ServerTimedOutException e) {
      System.out.println(
          "Server timed out, trying alternative");
      try {
        connectMe(alternativeServer);
      } catch (ServerTimedOutException e1) {
        System.out.println(
            "Error: " + e1.getReason() +
            " connecting to port " + e1.getPort());
      }
    }
  }
```

Check Your Progress

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

Think Beyond

- What features does the Java application environment have that support user interface development?

Module 8

Bulding GUIs

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

- Describe the AWT package and its components
- Define the terms *containers*, *components*, and *layout managers*, and how they work together to build a graphical user interface (GUI)
- Use layout managers
- Use the `FlowLayout`, `BorderLayout`, `GridLayout`, and `CardLayout` managers to achieve a desired dynamic layout
- Add components to a container
- Use the `Frame` and `Panel` containers appropriately

Objectives

- Describe how complex layouts with nested containers work
- In a Java program, identify the following:
  - Containers
  - The associated layout managers
  - The layout hierarchy of all components

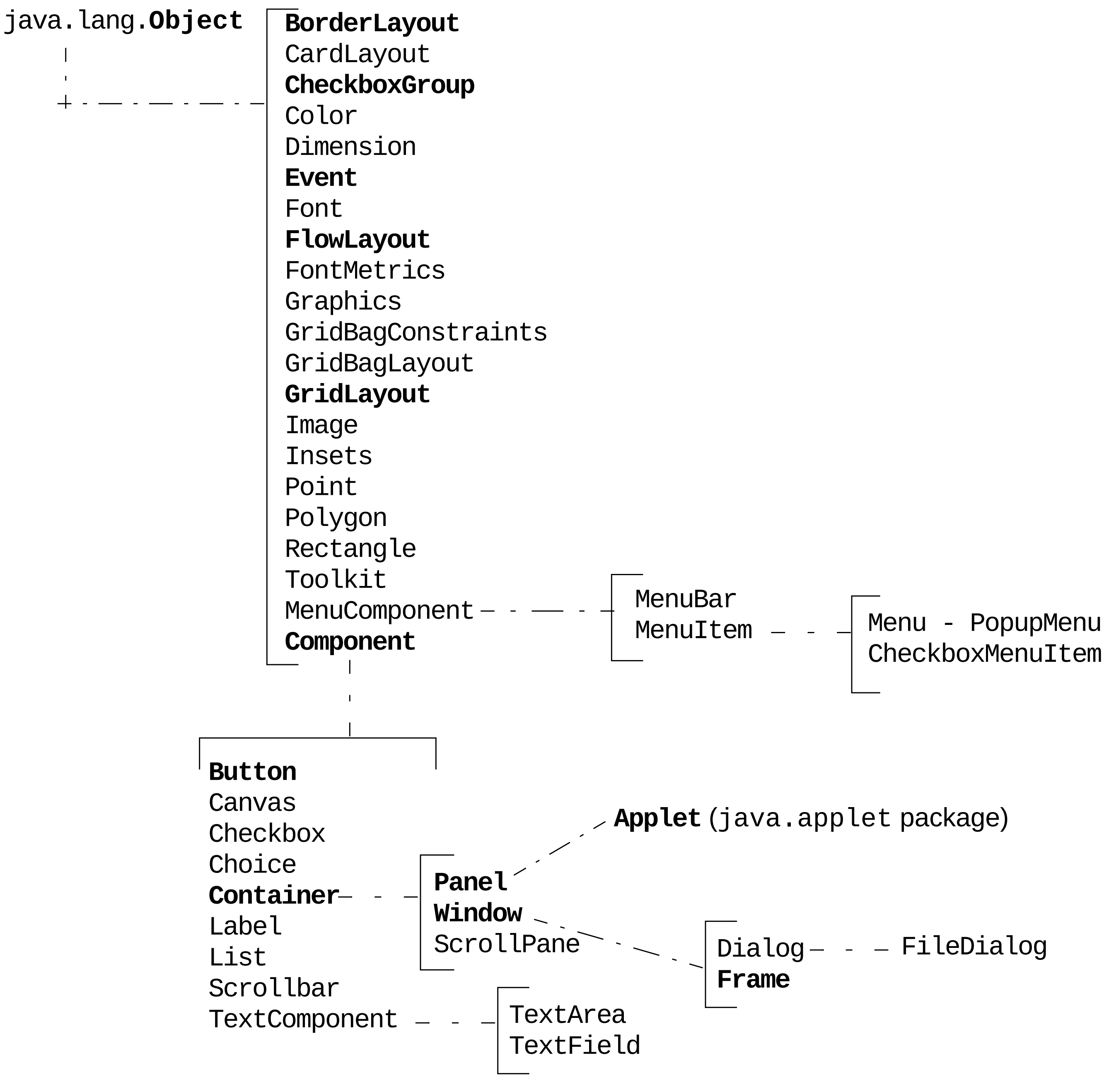
Relevance

- As a platform-independent programming language, how is Java technology used to make the GUI platform independent?

The AWT

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

The `java.awt` Package



Exceptions – `AWTException`   Errors – `AWTError`

Containers

- The two main types of containers are `Window` and `Panel`.
  - `Windows` are objects of `java.awt.Window`.
  - `Panels` are objects of `java.awt.Panel`.

Building Graphical User Interfaces

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

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

Frame

- Is a subclass of `Window`
- Has title and resizing corners
- Inherits from `Container` and adds components with the `add()` method
- Can be used to create invisible `Frame` objects with a title specified by a string.
- Has `BorderLayout` as the default layout manager
- Uses the `setLayout` method to change the default layout manager

MyFrame.java

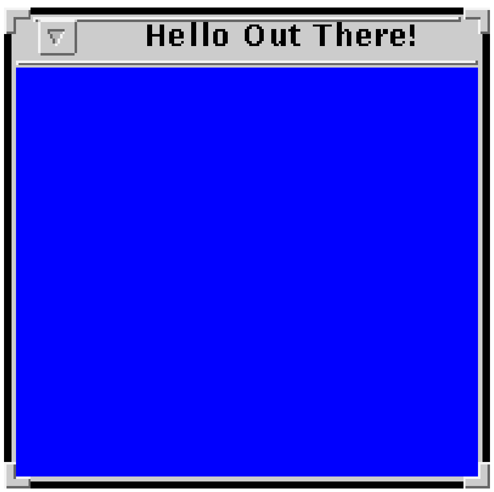
```
import java.awt.*;

public class MyFrame extends Frame {

  public MyFrame (String str) {
    super(str);
  }

  public static void main (String args[]) {
    MyFrame fr = new MyFrame("Hello Out There!");
    fr.setSize(500,500);
    fr.setBackground(Color.blue);
    fr.setVisible(true);
  }
}
```

MyFrame.java



`Panel`

- Provides a space for components
- Allows subpanels to have their own layout manager
- Adds components with the `add()` method

FrameWithPanel.java

```
import java.awt.*;

public class FrameWithPanel extends Frame {

  // Constructor
  public FrameWithPanel (String str) {
    super(str);
  }

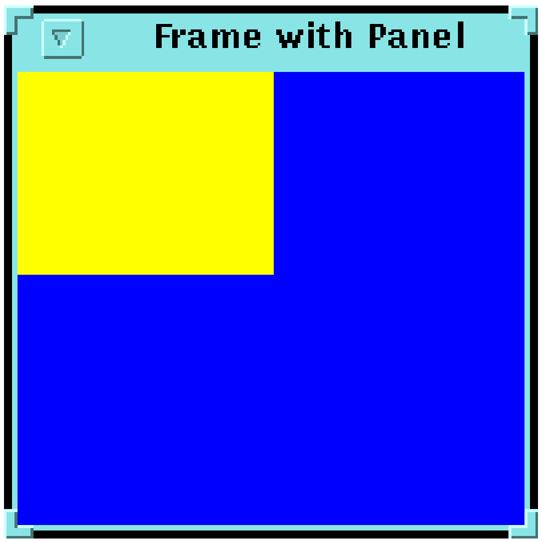
  public static void main (String args[]) {
    FrameWithPanel fr =
          new FrameWithPanel("Frame with Panel");
    Panel pan = new Panel();

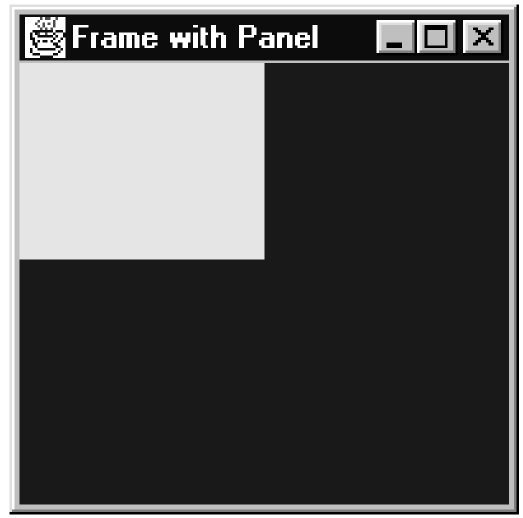
    fr.setSize(200,200);
    fr.setBackground(Color.blue);
    fr.setLayout(null);  // Override default layout mgr

    pan.setSize(100,100);
    pan.setBackground(Color.yellow);

    fr.add(pan);
    fr.setVisible(true);
  }
}
```

FrameWithPanel.java

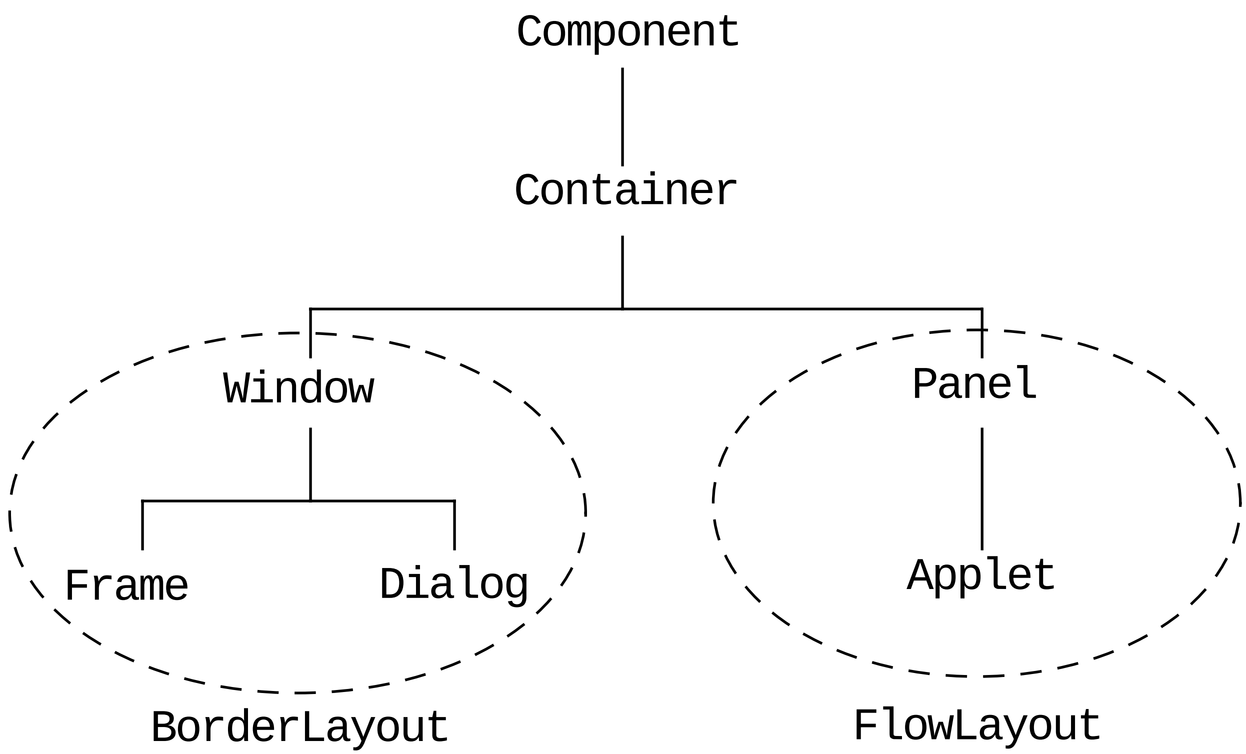




Container Layouts

- FlowLayout
- BorderLayout
- GridLayout
- CardLayout
- GridBagLayout

Default Layout Managers



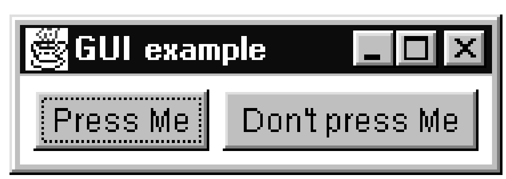
A Simple FlowLayout Example

```
import java.awt.*;

public class ExGui {
  private Frame f;
  private Button b1;
  private Button b2;

  public void go() {
    f = new Frame("GUI example");
    f.setLayout(new FlowLayout());
    b1 = new Button("Press Me");
    b2 = new Button("Don't press Me");
    f.add(b1);
    f.add(b2);
    f.pack();
    f.setVisible(true);
  }

  public static void main(String args[]) {
    ExGui guiWindow = new ExGui();
    guiWindow.go();
  }
}
```



`FlowLayout` Manager

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

MyFlow.java

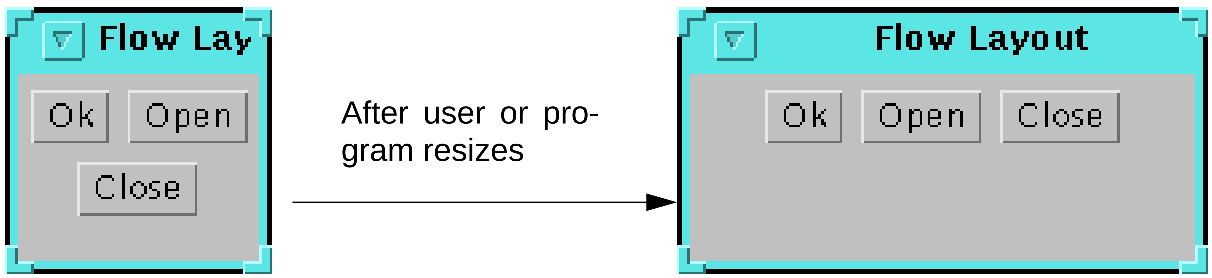
```
import java.awt.*;

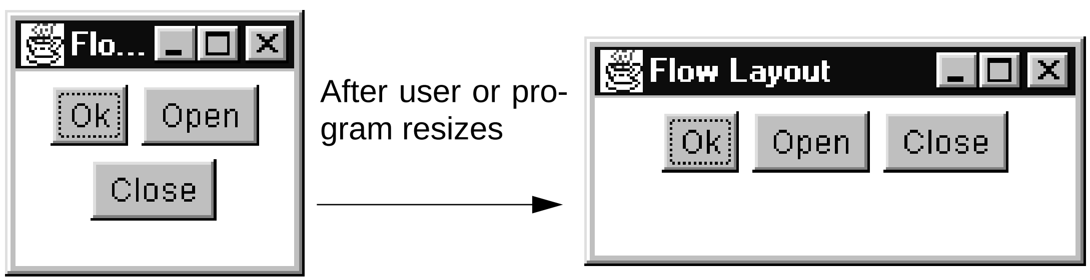
public class MyFlow {
  private Frame f;
  private Button button1, button2, button3;

  public void go() {
    f = new Frame("Flow Layout");
    f.setLayout(new FlowLayout());
    button1 = new Button("Ok");
    button2 = new Button("Open");
    button3 = new Button("Close");
    f.add(button1);
    f.add(button2);
    f.add(button3);
    f.setSize(100,100);
    f.setVisible(true);
  }

  public static void main(String args[]) {
    MyFlow mflow = new MyFlow();
    mflow.go();
  }
}
```

MyFlow.java


Flow Lay
Ok
Open
Close
After user or program resizes
Flow Layout
Ok
Open
Close


Flo...
Ok
Open
Close
After user or program resizes
Flow Layout
Ok
Open
Close

ExGui2.java

```
import java.awt.*;

public class ExGui2 {
  private Frame f;
  private Button bn, bs, bw, be, bc;

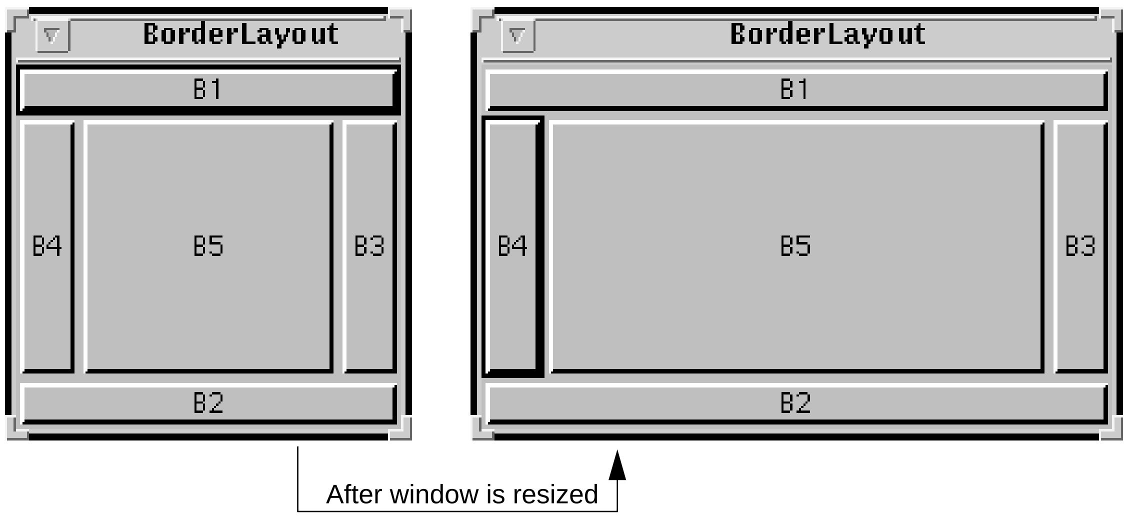
  public void go() {
    f = new Frame("Border Layout");
    bn = new Button("B1");
    bs = new Button("B2");
    bw = new Button("B3");
    be = new Button("B4");
    bc = new Button("B5");

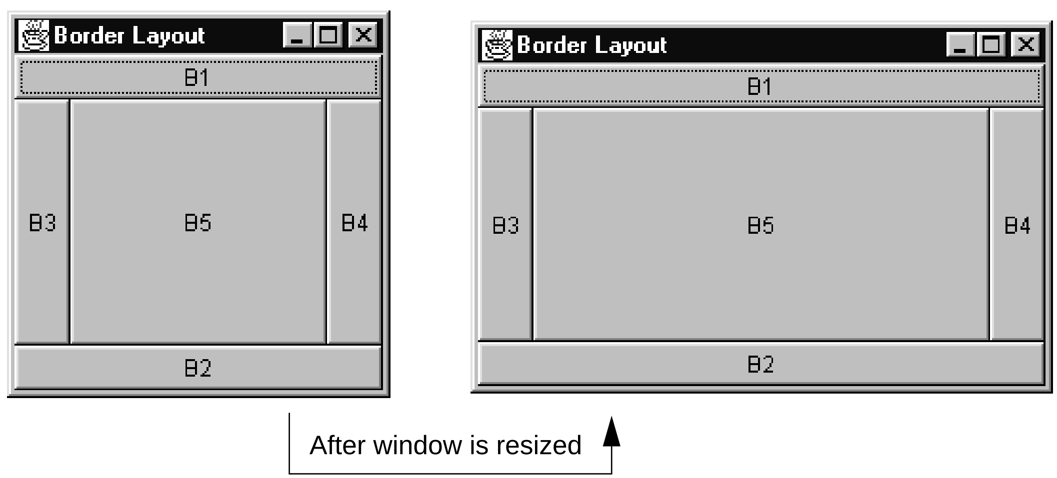
    f.add(bn, BorderLayout.NORTH);
    f.add(bs, BorderLayout.SOUTH);
    f.add(bw, BorderLayout.WEST);
    f.add(be, BorderLayout.EAST);
    f.add(bc, BorderLayout.CENTER);

    f.setSize(200,200);
    f.setVisible(true);
  }

  public static void main(String args[]) {
    ExGui2 guiWindow2 = new ExGui2();
    guiWindow2.go();
  }
}
```

ExGui2.java


BorderLayout
B1
B4
B5
B3
B2
BorderLayout
B1
B4
B5
B3
B2
After window is resized


Border Layout
B1
B3
B5
B4
B2
Border Layout
B1
B3
B5
B4
B2
After window is resized

`GridLayout` Manager

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

GridEx.java

```
import java.awt.*;

public class GridEx {
  private Frame f;
  private Button b1, b2, b3, b4, b5, b6;

  public void go() {
    f = new Frame("Grid Example");
    f.setLayout (new GridLayout(3,2));

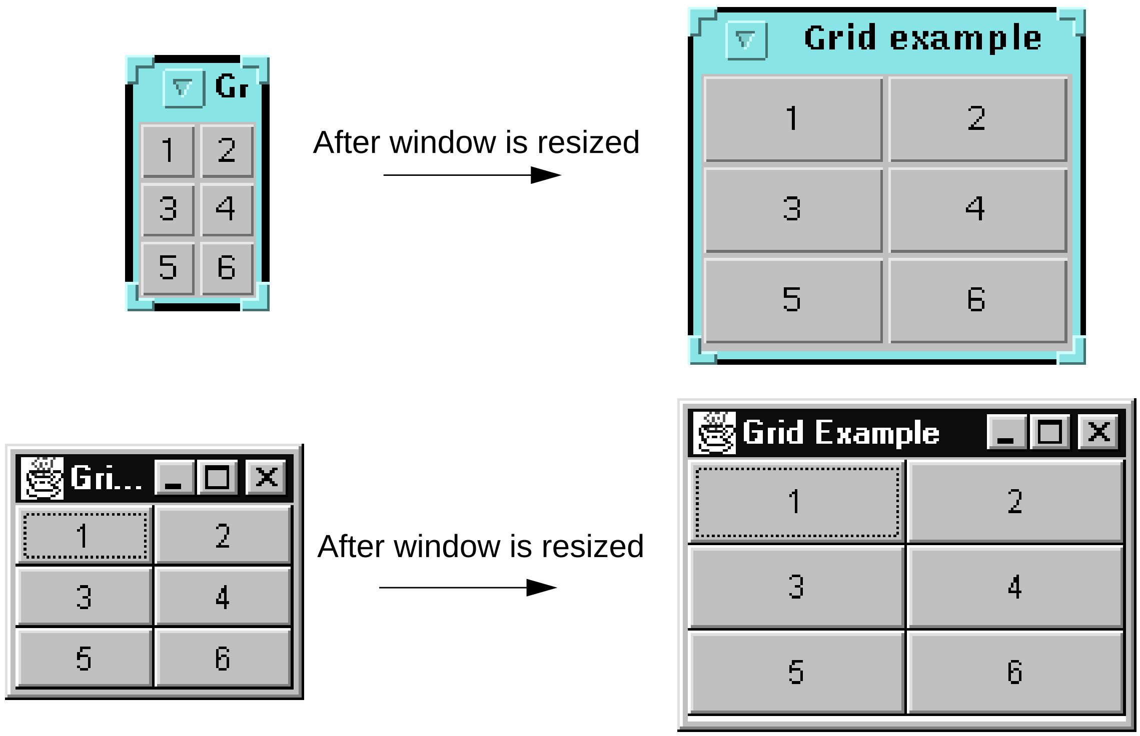
    b1 = new Button("1");
    b2 = new Button("2");
    b3 = new Button("3");
    b4 = new Button("4");
    b5 = new Button("5");
    b6 = new Button("6");

    f.add(b1);
    f.add(b2);
    f.add(b3);
    f.add(b4);
    f.add(b5);
    f.add(b6);

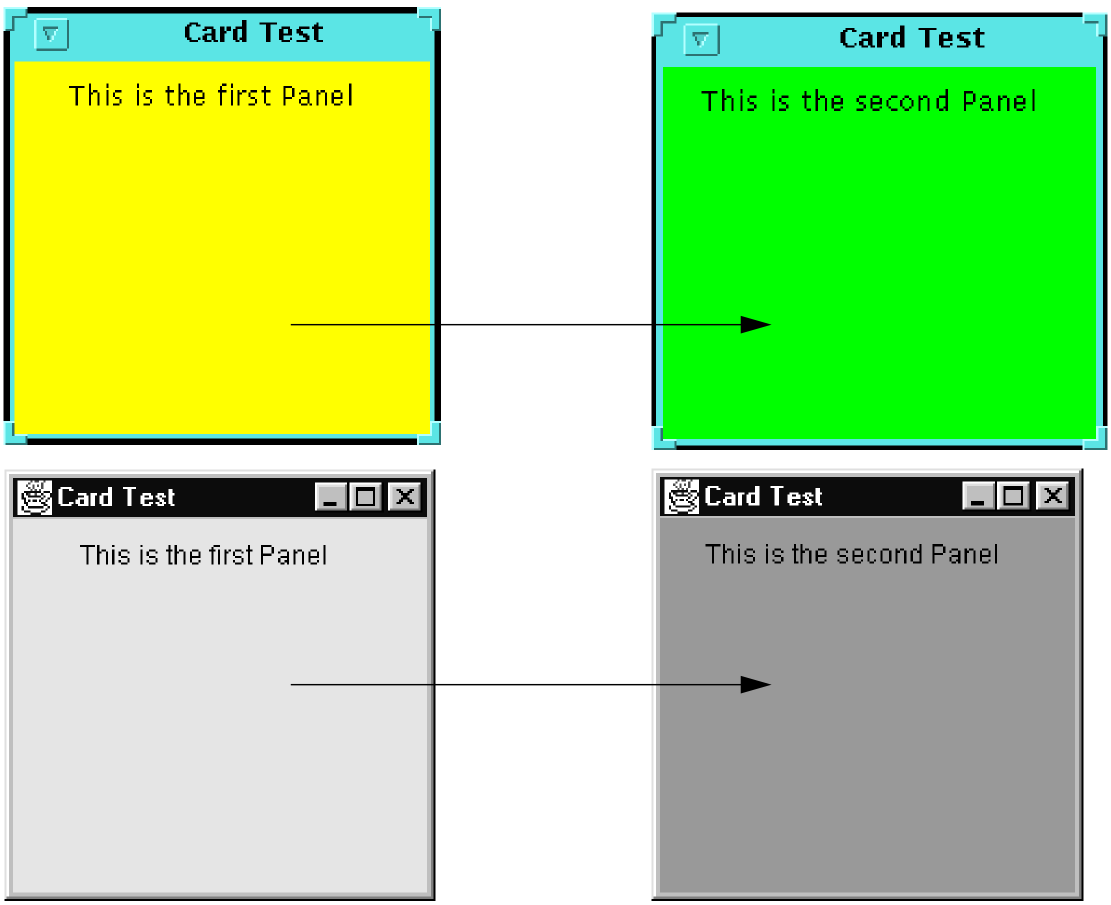
    f.pack();
    f.setVisible(true);
  }

  public static void main(String args[]) {
    GridEx grid = new GridEx();
    grid.go();
  }
}
```

GridEx.java


Gr
1
2
3
4
5
6
After window is resized
Grid example
1
2
3
4
5
6
Gri...
1
2
3
4
5
6
After window is resized
Grid Example
1
2
3
4
5
6

`CardLayout` Manager



CardLayout Manager

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

public class CardTest implements MouseListener {
  private Panel p1, p2, p3, p4, p5;
  private Label lb1, lb2, lb3, lb4, lb5;

  // Declare a CardLayout object to call its methods.
  private CardLayout myCard;
  private Frame f;

  public void go() {
    f = new Frame ("Card Test");
    myCard = new CardLayout();
    f.setLayout(myCard);

    // Create the panels that I want
    // to use as cards.
    p1 = new Panel();
    p2 = new Panel();
    p3 = new Panel();
    p4 = new Panel();
    p5 = new Panel();

    // Create a label to attach to each panel, and
    // change the color of each panel, so they are
    // easily distinguishable

    lb1 = new Label("This is the first Panel");
    p1.setBackground(Color.yellow);
    p1.add(lb1);

    lb2 = new Label("This is the second Panel");
    p2.setBackground(Color.green);
    p2.add(lb2);

    lb3 = new Label("This is the third Panel");
    p3.setBackground(Color.magenta);
    p3.add(lb3);
```

CardLayout Manager

```

    lb4 = new Label("This is the fourth Panel");
    p4.setBackground(Color.white);
    p4.add(lb4);

    lb5 = new Label("This is the fifth Panel");
    p5.setBackground(Color.cyan);
    p5.add(lb5);

    // Set up the event handling here.
    p1.addMouseListener(this);
    p2.addMouseListener(this);
    p3.addMouseListener(this);
    p4.addMouseListener(this);
    p5.addMouseListener(this);

    // Add each panel to my CardLayout
    f.add(p1, "First");
    f.add(p2, "Second");
    f.add(p3, "Third");
    f.add(p4, "Fourth");
    f.add(p5, "Fifth");

    // Display the first panel.
    myCard.show(f, "First");

    f.setSize(200,200);
    f.setVisible(true);
  }

  public void mousePressed(MouseEvent e) {
    myCard.next(f);
  }

  public void mouseReleased(MouseEvent e) { }
  public void mouseClicked(MouseEvent e) { }
  public void mouseEntered(MouseEvent e) { }
  public void mouseExited(MouseEvent e) { }

```

CardLayout Manager

```
 public static void main (String args[]) {
    CardTest ct = new CardTest();
    ct.go();
  }
}
```

`GridBagLayout` Manager

- Complex layout facilities can be placed in a grid.
- A single component can take its preferred size.
- A component can extend over more than one cell.

ExGui3.java

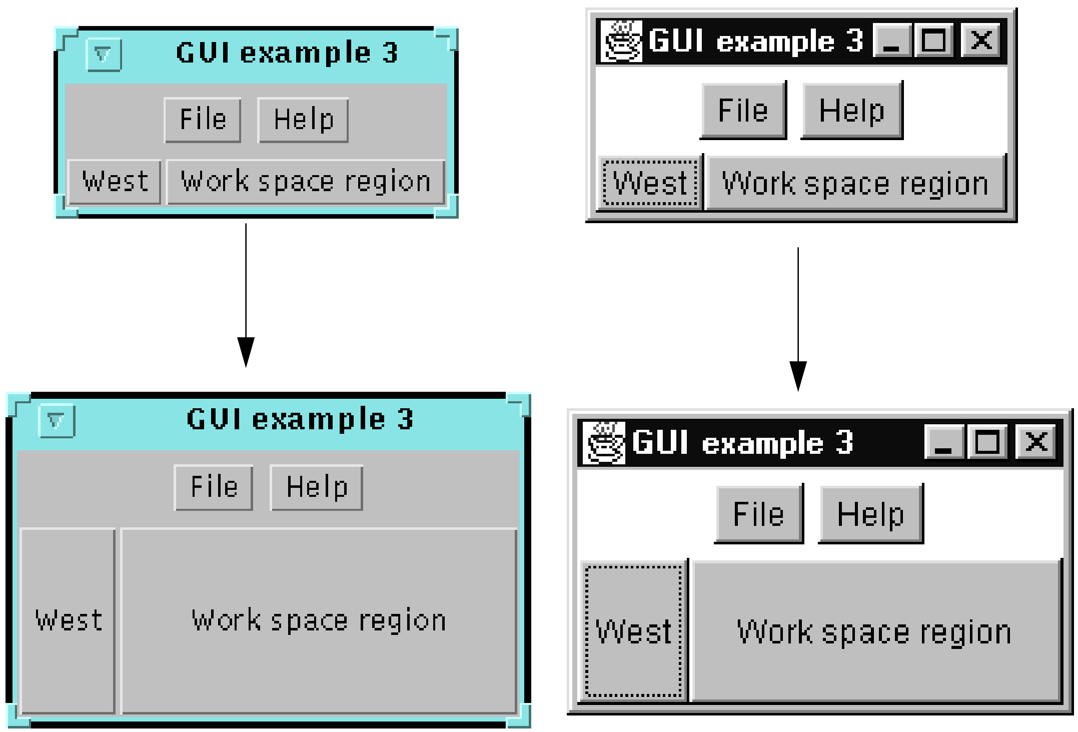
```
import java.awt.*;

public class ExGui3 {
  private Frame f;
  private Panel p;
  private Button bw, bc;
  private Button bfile, bhelp;

  public void go() {
    f = new Frame("GUI example 3");
    bw = new Button("West");
    bc = new Button("Work space region");
    f.add(bw, BorderLayout.WEST);
    f.add(bc, BorderLayout.CENTER);
    p = new Panel();
    bfile = new Button("File");
    bhelp = new Button("Help");
    p.add(bfile);
    p.add(bhelp);
    f.add(p, BorderLayout.NORTH);
    f.pack();
    f.setVisible(true);
  }

  public static void main(String args[]) {
    ExGui3 gui = new ExGui3();
    gui.go();
  }
}
```

Output of `ExGui3.java`



Exercise: Building GUIs

- Exercise objective:
  - Develop two graphical user interfaces using the AWT
- Tasks:
  - Create a calculator GUI
  - Create an account GUI

Check Your Progress

- Describe the AWT package and its components
- Define the terms *containers*, *components*, and *layout managers*, and how they work together to build a graphical user interface (GUI)
- Use layout managers
- Use the `FlowLayout`, `BorderLayout`, `GridLayout`, and `CardLayout` managers to achieve a desired dynamic layout
- Add components to a container
- Use the `Frame` and `Panel` containers appropriately

Check Your Progress

- Describe how complex layouts with nested containers work
- In a Java program, identify the following:
  - Containers
  - The associated layout managers
  - The layout hierarchy of all components

Think Beyond

- You now know how to display a GUI on the computer screen. What do you need to make the GUI useful?

Module 9

The AWT Event Model

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

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

Relevance

- What parts are required for a GUI to make it useful?
- How does a graphical progam handle a mouse click or any other type of user interaction?

What Is an Event?

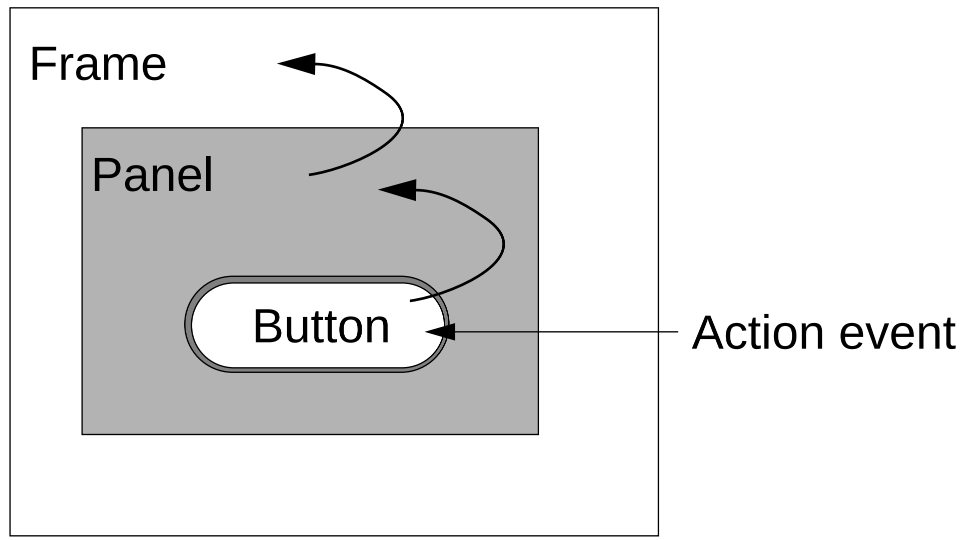
- Events – Objects that describe what happened
- Event sources – The generator of an event
- Event handlers – A method that receives an event object, deciphers it, and processes the user's interaction

JDK 1.0 Event Model Versus Java 2 SDK Event Model

- Hierarchical model (JDK 1.0)
- Delegation model (JDK 1.1 and beyond)

Hierarchical Model (JDK 1.0)

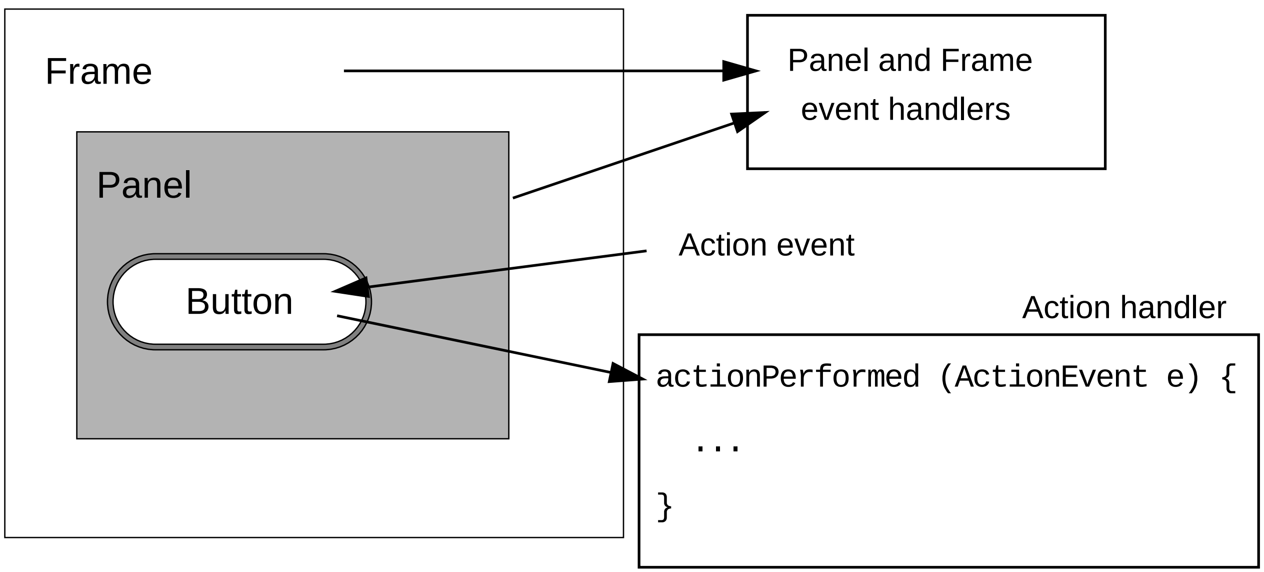
- Is based on containment



Hierarchical Model (JDK 1.0)

- Advantages:
  - Uses object-oriented principles
- Disadvantages:
  - An event can be handled only by the component from which it originated or by one of the containers of the originating component
  - To handle events, you must either subclass the component that receives the event or create a `handleEvent()` method at the base container

Delegation Model



Delegation Model

```
import java.awt.*;

public class TestButton {
  public static void main(String args[]) {
    Frame f = new Frame("Test");
    Button b = new Button("Press Me!");
    b.addActionListener(new ButtonHandler());
    f.add(b,BorderLayout.CENTER);
    f.pack();
    f.setVisible(true);
  }
}
```

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

public class ButtonHandler implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("Action occurred");
    System.out.println(
          "Button's label is :"  + e.getActionCommand());
  }
}
```

Delegation Model (JDK 1.1 and Beyond)

- Advantages:
  - Events are not accidentally handled
  - You can create and use filter (adapter) classes to classify event actions
  - There is better distribution of work among the classes
- Disadvantage:
  - You should not combine two event models

Frame With a Single Button

```
import java.awt.*;

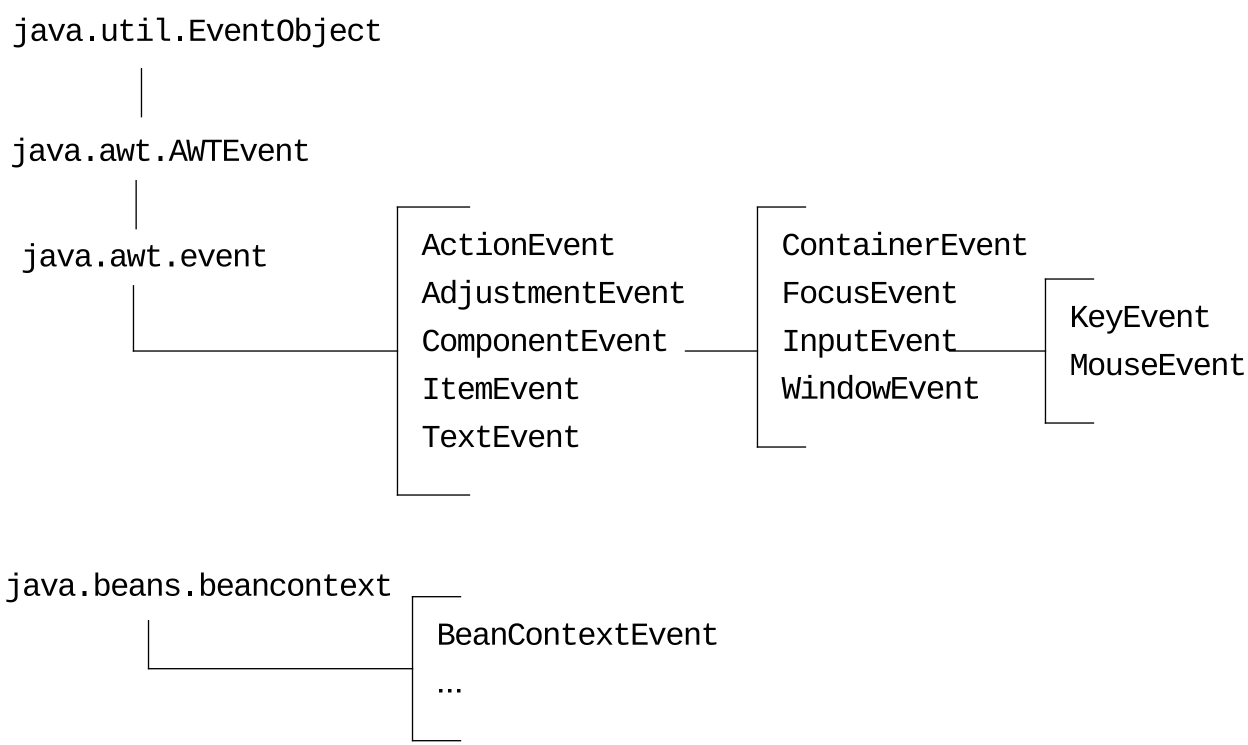
public class TestButton {
  public static void main(String args[]) {
    Frame f = new Frame("Test");
    Button b = new Button("Press Me!");
    b.addActionListener(new ButtonHandler());
    f.add(b,BorderLayout.CENTER);
    f.pack();
    f.setVisible(true);
  }
}
```

The `ButtonHandler` Class

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

public class ButtonHandler implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("Action occurred");
    System.out.println(
          "Button's label is :"  + e.getActionCommand());
  }
}
```

Event Categories



Java GUI Behavior

Category	Interface Name	Methods
Action	`ActionListener`	`actionPerformed(ActionEvent)`
Item	`ItemListener`	`itemStateChanged(ItemEvent)`
Mouse motion	`MouseMotionListener`	`mouseDragged(MouseEvent)`
		`mouseMoved(MouseEvent)`
Mouse button	`MouseListener`	`mousePressed(MouseEvent)`
		`mouseReleased(MouseEvent)`
		`mouseEntered(MouseEvent)`
		`mouseExited(MouseEvent)`
		`mouseClicked(MouseEvent)`
Key	`KeyListener`	`keyPressed(KeyEvent)`
		`keyReleased(KeyEvent)`
		`keyTyped(KeyEvent)`
Focus	`FocusListener`	`focusGained(FocusEvent)`
		`focusLost(FocusEvent)`
Adjustment	`AdjustmentListener`	`adjustmentValueChanged (AdjustmentEvent)`
Component	`ComponentListener`	`componentMoved(ComponentEvent)`
		`componentHidden(ComponentEvent)`
		`componentResized(ComponentEvent)`
		`componentShown(ComponentEvent)`

Java GUI Behavior

Category	Interface Name	Methods
Window	`WindowListener`	`windowClosing(WindowEvent)`
		`windowOpened(WindowEvent)`
		`windowIconified(WindowEvent)`
		`windowDeiconified(WindowEvent)`
		`windowClosed(WindowEvent)`
		`windowActivated(WindowEvent)`
		`windowDeactivated(WindowEvent)`
Container	`ContainerListener`	`componentAdded(ContainerEvent)`
		`componentRemoved(ContainerEvent)`
Text	`TextListener`	`textValueChanged(TextEvent)`

Complex Example

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

public class TwoListen
      implements MouseMotionListener,
      MouseListener {
  private Frame f;
  private TextField tf;

  public void go() {
    f = new Frame("Two listeners example");
    f.add(new Label ("Click and drag the mouse"),
                      BorderLayout.NORTH);
    tf = new TextField (30);
    f.add (tf, BorderLayout.SOUTH);

    f.addMouseMotionListener(this);
    f.addMouseListener (this);
    f.setSize(300, 200);
    f.setVisible(true);
  }

  // These are MouseMotionListener events
  public void mouseDragged (MouseEvent e) {
    String s =
      "Mouse dragging:  X = " + e.getX() +
                      " Y = " + e.getY();
    tf.setText (s);
  }

  public void mouseEntered (MouseEvent e) {
    String s = "The mouse entered";
    tf.setText (s);
  }

  public void mouseExited (MouseEvent e) {
    String s = "The mouse has left the building";
    tf.setText (s);
  }
```

Complex Example

```

  // Unused MouseMotionListener method.
  // All methods of a listener must be present in the
  // class even if they are not used.
  public void mouseMoved (MouseEvent e) { }

  // Unused MouseListener methods.
  public void mousePressed (MouseEvent e) { }
  public void mouseClicked (MouseEvent e) { }
  public void mouseReleased (MouseEvent e) { }

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

Multiple Listeners

- Multiple listeners cause unrelated parts of a program to react to the same event
- All registered listeners call their handlers when the event occurs

Event Adapters

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

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

public class MouseClickHandler extends MouseAdapter {

  // We just need the mouseClick handler, so we use
  // the an adapter to avoid having to write all the
  // event handler methods

  public void mouseClicked (MouseEvent e) {
    // Do stuff with the mouse click...
  }
}
```

Anonymous Classes

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

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

  public void go() {
    f = new Frame("Anonymous classes example");
    f.add(new Label("Click and drag the mouse"),
            BorderLayout.NORTH);
    tf = new TextField (30);
    f.add (tf, BorderLayout.SOUTH);

    f.addMouseMotionListener( new MouseMotionAdapter() {
      public void mouseDragged (MouseEvent e) {
        String s = "Mouse dragging:  X = "+ e.getX() +
                    " Y = " + e.getY();
        tf.setText (s);
      }
    }); // <- note the closing parenthesis

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

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

Exercise: Working With Events

- Exercise objective:
  - Write, compile, and run the revised Calculator GUI and Account GUI codes to include event handlers.
- Tasks:
  - Re-create the calculator GUI
  - Re-create the account GUI

Check Your Progress

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

Think Beyond

You now know how to set up a Java GUI for both graphic output and interactive user input. However, only a few of the components from which GUIs can be built have been described. What other components would be useful in a GUI?

Module 10

The AWT Component Library

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

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

Relevance

- You now know how to set up a Java GUI for both graphic output and interactive user input. However, only a few of the components from which GUIs can be built have been described. What other components would be useful in a GUI?

Features of the AWT

- AWT components provide mechanisms for controlling the interface appearance, including color and font.
- The AWT also supports printing. (It was added in the JDK 1.1 release.)

Creating a `Button`

```
f = new Frame("Sample Button");
Button b = new Button("Sample");
b.addActionListener(this);
f.add(b);
```


```
public void actionPerformed(ActionEvent ae) {
  System.out.println("Button press received.");
  System.out.println("Button’s action command is: "+
      ae.getActionCommand());
}

```

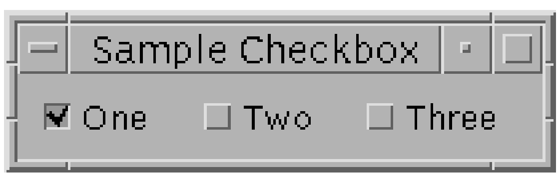
```
    b = new Button("Sample");
    b.setActionCommand("Action Command Was Here!");
    b.addActionListener(this);
    f.add(b);
```

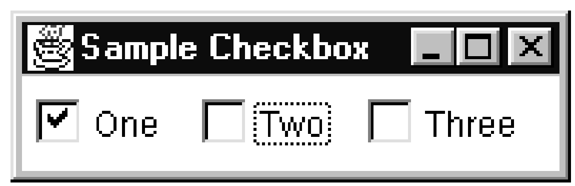
Creating a Checkbox

```
f = new Frame("Sample Checkbox");
      one = new Checkbox("One", true);
      two = new Checkbox("Two", false);
      three = new Checkbox("Three", false);

      one.addItemListener(this);
      two.addItemListener(this);
      three.addItemListener(this);

      f.setLayout(new FlowLayout());
      f.add(one);
      f.add(two);
      f.add(three);
```





Creating the `ItemListener` Interface

```
class Handler implements ItemListener {
  public void itemStateChanged(ItemEvent ev) {
    String state = "deselected";
    if (ev.getStateChange() == ItemEvent.SELECTED){
      state = "selected";
    }
    System.out.println(ev.getItem() + " " + state);
  }
}
```

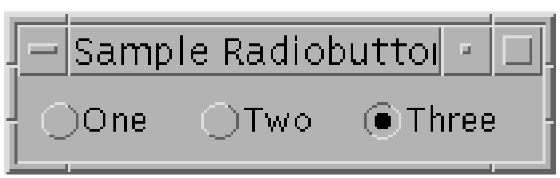
Creating the `CheckboxGroup` – Radio Buttons

```
f = new Frame("CheckBoxGroup");
cbg = new CheckboxGroup();
one = new Checkbox("One", cbg, false);
two = new Checkbox("Two", cbg, false);
three = new Checkbox("Three", cbg, true);

f.setLayout(new FlowLayout());

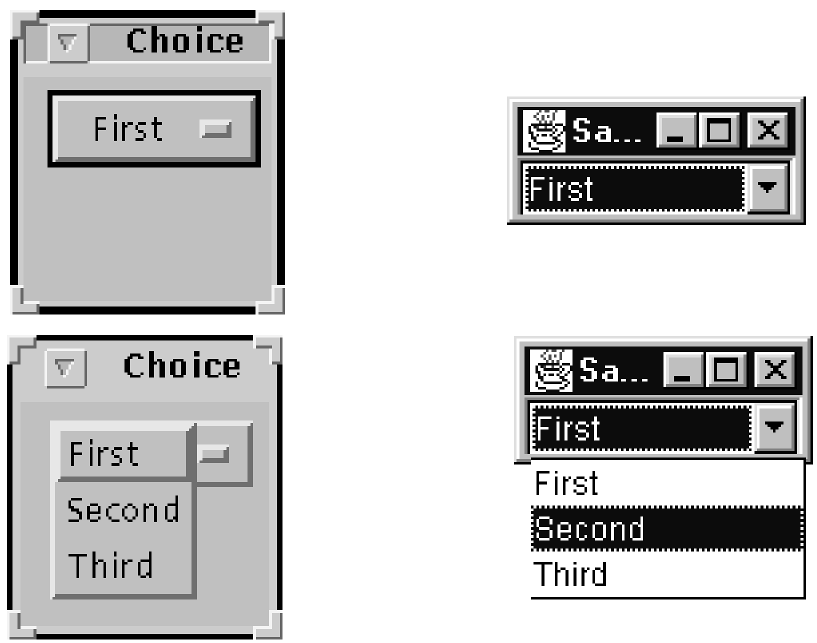
one.addItemListener(this);
two.addItemListener(this);
three.addItemListener(this);

f.add(one);
f.add(two);
f.add(three);
```



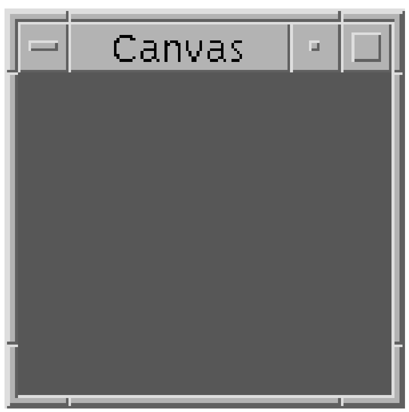
Creating a Choice

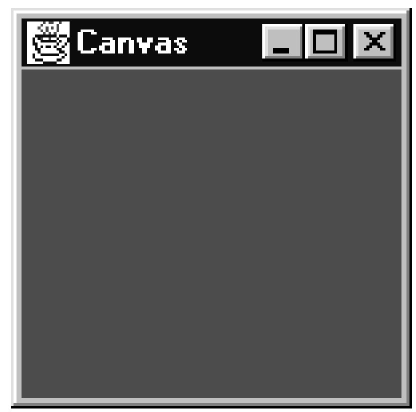
```
f = new Frame("Sample Choice");
choice = new Choice();
choice.addItem("First");
choice.addItem("Second");
choice.addItem("Third");
choice.addItemListener(this);
f.add(choice, BorderLayout.CENTER);
```



Canvas

- Provides a blank space to draw, write text, or receive keyboard or mouse input





Creating a Canvas

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

public class MyCanvas extends Canvas
      implements KeyListener{
  private int index;
  Color colors[] = { Color.red, Color.green, Color.blue };

  public void paint(Graphics g) {
    g.setColor(colors[ index ]);
    g.fillRect(0, 0, getSize().width, getSize().height);
  }

  public void keyTyped(KeyEvent ev) {
    index++;
    if ( index == colors.length ) {
      index = 0;
    }
    repaint();
  }

  // Unused KeyListener methods
  public void keyPressed(KeyEvent ev) { }
  public void keyReleased(KeyEvent ev) { }

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

Creating a `Label`

```
Frame f = new Frame("Label");
Label lb = new Label("Hello");
f.add(lb);
```

Creating a `TextField`

```
Frame f = new Frame("TextField");
TextField tf = new TextField("Single line"”, 30);
tf.addActionListener(this);
f.add(tf);
```

Creating a `TextArea`

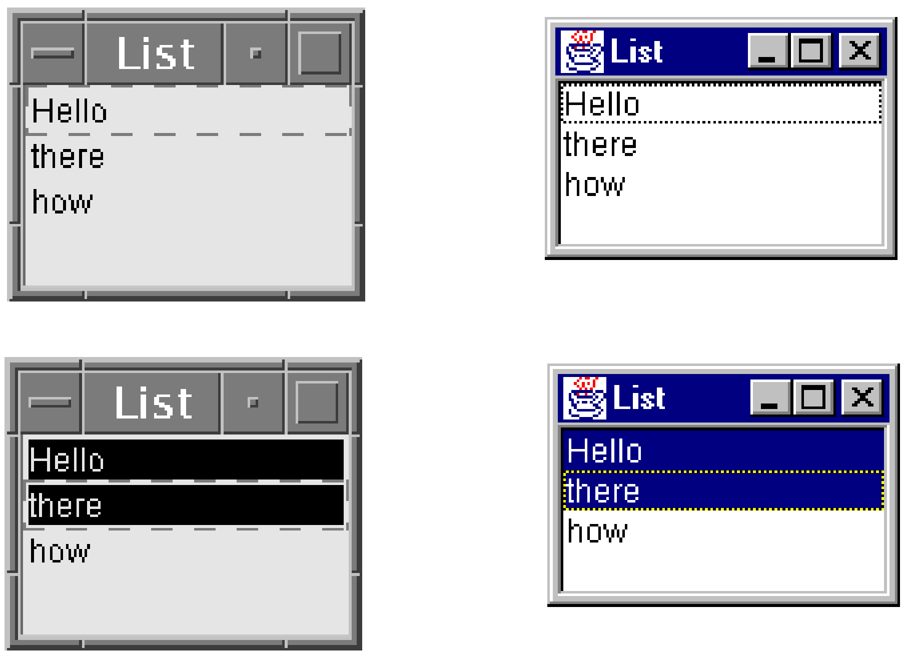
```
f = new Frame("TextArea");
ta = new TextArea("Hello!", 4, 30);
f.add(ta, BorderLayout.CENTER);
```

Text Components

- `TextArea` and `TextField` are subclasses
- `TextArea` and `TextField` inherit the default behavior for keystrokes from `TextComponent`

Creating a `List` Component

```
List lst = new List(4, true);
lst.add("Hello");
lst.add("there");
lst.add("how");
```



Creating a `Dialog`

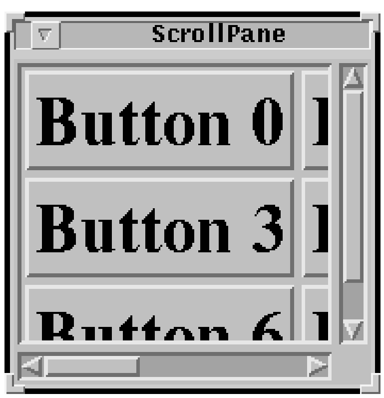
```
d = new Dialog(f, "Dialog", true);
d.setLayout(new GridLayout(2,1));
dl = new Label("Hello, I'm a Dialog");
db1 = new Button("OK");
d.add(dl);
d.add(db1);
d.pack();
```

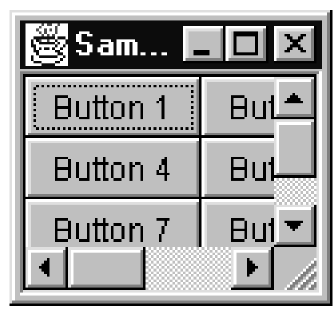
Creating a `FileDialog`

```
FileDialog d = new FileDialog(parentFrame, "FileDialog");
d.setVisible(true);  // block here until OK selected
String fname = d.getDirectory() + d.getFile();
```

Creating a ScrollPane

```
Frame f = new Frame("ScrollPane");
Panel p = new Panel();
ScrollPane sp = new ScrollPane();
p.setLayout(new GridLayout(3, 4));
.
.
.
sp.add(p);
f.add(sp, BorderLayout.CENTER");
f.setSize(100, 100);
f.setVisible(true);
```



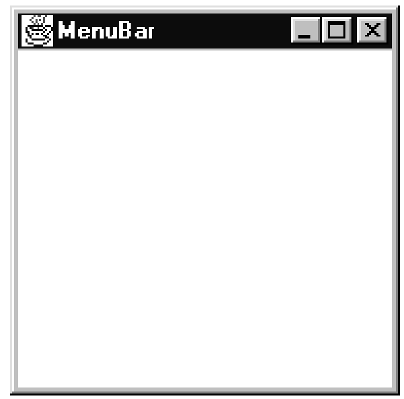


Menu

- Must be added to a menu container
- Includes a help menu:
  - `setHelpMenu(Menu)`

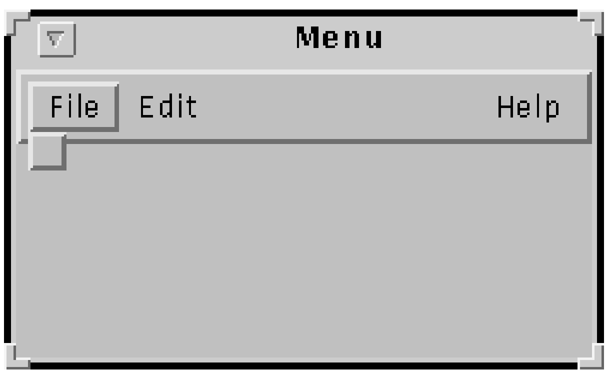
Creating a MenuBar

```
Frame f = new Frame("MenuBar");
MenuBar mb = new MenuBar();
f.setMenuBar(mb);
```



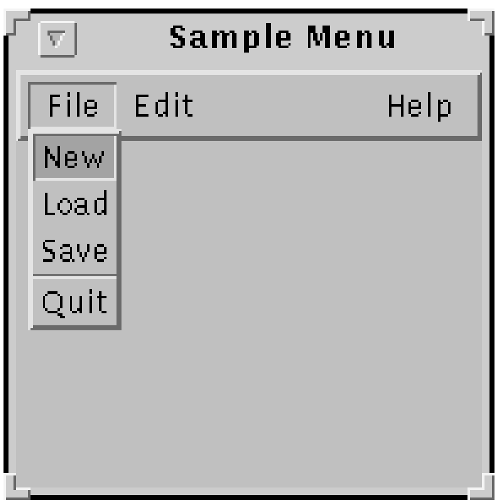
Creating a Menu

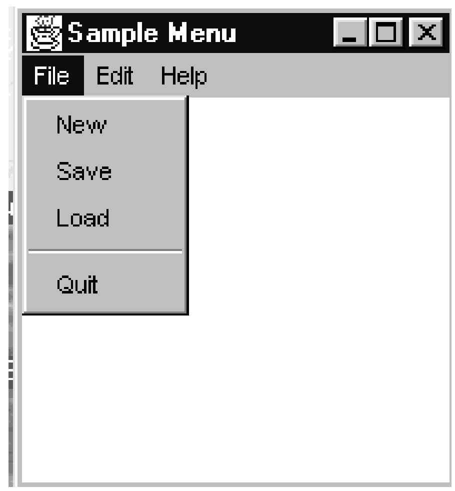
```
f = new Frame("Menu");
mb = new MenuBar();
m1 = new Menu("File");
m2 = new Menu("Edit");
m3 = new Menu("Help");
mb.add(m1);
mb.add(m2);
mb.setHelpMenu(m3);
f.setMenuBar(mb);
```



Creating a `MenuItem`

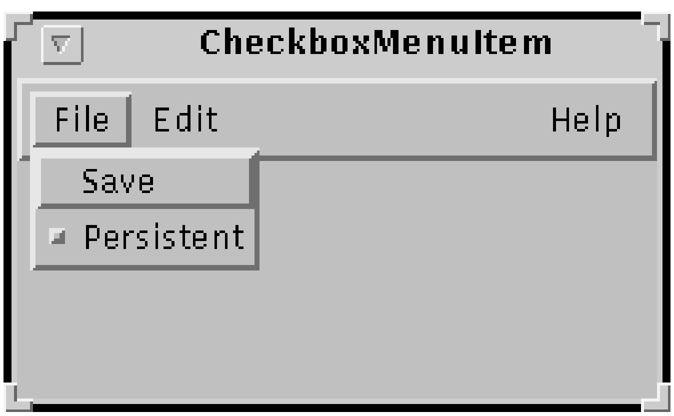
```
mi1 = new MenuItem("New");
mi2 = new MenuItem("Save");
mi3 = new MenuItem("Load");
mi4 = new MenuItem("Quit");
mi1.addActionListener(this);
mi2.addActionListener(this);
mi3.addActionListener(this);
mi4.addActionListener(this);
m1.add(mi1);
m1.add(mi2);
m1.add(mi3);
m1.addSeparator();
m1.add(mi4);
```

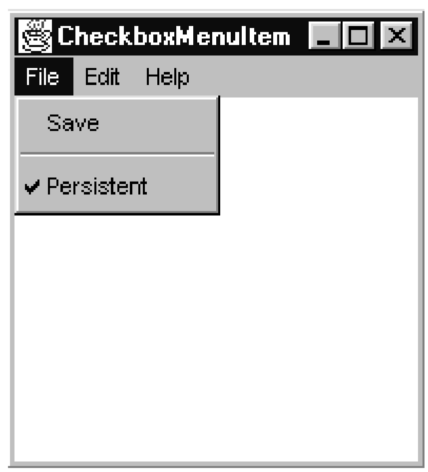




Creating a `CheckBoxMenuItem`

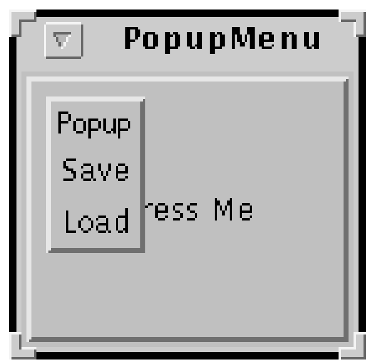
```
mb = new MenuBar();
m1 = new Menu("File");
m2 = new Menu("Edit");
m3 = new Menu("Help");
mb.add(m1);
mb.add(m2);
mb.setHelpMenu(m3);
f.setMenuBar(mb);
.....
mi2 = new MenuItem("Save");
mi2.addActionListener(this);
m1.add(mi2);
......
mi5 = new CheckboxMenuItem("Persistent");
mi5.addItemListener(this);
m1.add(mi5);
```





Creating a PopupMenu

```
Frame f = new Frame("PopupMenu");
Button b = new Button("Press Me");
PopupMenu p = new PopupMenu("Popup");
MenuItem s = new MenuItem("Save");
MenuItem ld = new MenuItem("Load");
b.addActionListener(this);
f.add(b,BorderLayout.CENTER);
p.add(s);
p.add(ld);
f.add(p);
```




```
public void actionPerformed(ActionEvent ev) {

  // display popup at (10,10) relative to b
  p.show(b, 10, 10);
}
```

Controlling Visual Aspects

- Colors:
  - `setForeground()`
  - `setBackground()`

Example:

```
int r = 255;
Color c = new Color(r, 0, 0);
```

Controlling Visual Aspects

- Fonts:
  - You can use the `setFont()` method to specify the font used for displaying text.
  - `Dialog`, `DialogInput`, `Serif`, and `SansSerif` are valid font names.

Printing

- Allow the use of local printer conventions:

```
Frame f = new Frame("Print test");
Toolkit t = f.getToolkit();
PrintJob job = t.getPrintJob(f, "MyPrintJob", null);
Graphics g = job.getGraphics();
```

- Draw on the graphics object
- Send the graphics object to printer
- End the print job
- Obtain a new graphic for each page use:

```
f.printComponents(g);
```

Exercise: Creating a Paint Program Layout

- Exercise objective:
  - Practice creating a more sophisticated GUI application that uses many components
- Tasks:
  - Create a Java application to use classes and objects
  - Investigate reference assignments

Check Your Progress

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

Think Beyond

- What would make the AWT work better?

Module 11

Java Foundation Classes

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

- Identify the key features of Java Foundation Classes
- Describe the key features of `com.sun.java.swing` package
- Identify Swing components
- Define *containers* and *components*, and explain how they work together to build a Swing GUI
- Write, compile, and run a basic Swing application
- Use top-level containers, such as `JFrame` and `JApplet` effectively

Relevance

- While the AWT by itself is useful, it is a part of a new set of classes, called Java Foundation Classes (JFC), that, as a whole, take GUIs to a new level. What exactly is JFC and, in particular, what is Swing? What can Swing do that AWT cannot?

Introduction

- Java Foundation Classes (JFC) consists of five APIs:
  - AWT
  - Java 2D
  - Accessibility
  - Drag and Drop
  - Swing

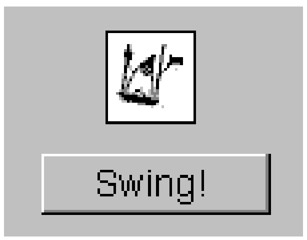
Swing Introduction

- Pluggable look and feel:
  - Application appears to be platform specific
  - There are custom Swing components
- Swing architecture:
  - Built around APIs that implement various parts of AWT
  - Most components do not use platform-specific implementations like AWT

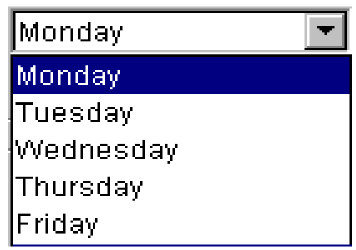
Swing Hierarchy

- java.awt.Container
  - com.sun.java.swing.JComponent
    - JTextComponent
      - JTextArea
      - JTextField
        - JPasswordField
      - HtmlEditorKit
    - AbstractButton
      - JToggleButton
        - JCheckBox
        - JRadioButton
      - JButton
      - JMenuItem
        - JRadioButtonMenuItem
        - JCheckBoxMenuItem
        - JMenu
    - JPanel
    - JComboBox
    - JLabel
    - JLayeredPane
    - JList
    - JToolBar
    - JMenuBar
    - JPopupMenu
    - JPanel
    - JScrollBar
    - JScrollPane
    - JSlider
    - JTable
    - JSeparator
    - JTree
    - JProgressBar
    - JRootPane

Swing Components



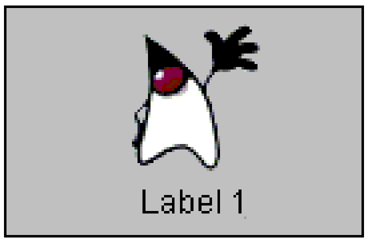
JApplet



JComboBox

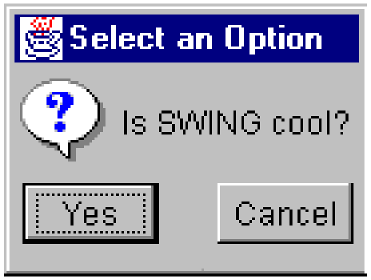
JList

Swing Components



`JLabel`

`JButton`
`JToggleButton`



`JOptionPane`

A Basic Swing Application

HelloSwing

```
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.accessibility.*;

public class HelloSwing implements ActionListener {
  private JFrame jFrame;
  private JLabel jLabel;
  private JPanel jPanel;
  private JButton jButton;
  private AccessibleContext accContext;

  private String labelPrefix = "Number of button clicks: ";
  private int numClicks = 0;

  public void go() {
    // Here is how you can set up a particular
    // lookAndFeel.  Not necessary for default.
    //
    // try {
    //     UIManager.setLookAndFeel(
    //      UIManager.getLookAndFeel());
    // } catch (UnsupportedLookAndFeelException e) {
    //     System.err.println("Couldn't use the " +
    //       "default look and feel " + e);
    // }

    jFrame = new JFrame("HelloSwing");
    jLabel = new JLabel(labelPrefix + "0");

    jButton = new JButton("I am a Swing button!");

    // Create a shortcut: make ALT-A be equivalent
    // to pressing mouse over button.
    jButton.setMnemonic('i');

    jButton.addActionListener(this);

```

HelloSwing

```
    // Add support for accessibility.
    accContext = jButton.getAccessibleContext();
    accContext.setAccessibleDescription(
      "Pressing this button increments " +
      "the number of button clicks");

    // Set up pane.
    // Give it a border around the edges.
    jPanel = new JPanel();
    jPanel.setBorder(
      BorderFactory.createEmptyBorder(30,30,10,30));

    // Arrange for compts to be in a single column.
    jPanel.setLayout(new GridLayout(0, 1));

      // Put compts in pane, not in JFrame directly.
    jPanel.add(jButton);
    jPanel.add(jLabel);
    jFrame.setContentPane(jPanel);

    // Set up a WindowListener inner class to handle
    // window's quit button.
    WindowListener wl = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };

    jFrame.addWindowListener(wl);

    jFrame.pack();
    jFrame.setVisible(true);
  }

  // Button handling.
  public void actionPerformed(ActionEvent e) {
    numClicks++;
    jLabel.setText(labelPrefix + numClicks);
  }
```

HelloSwing

```

  public static void main(String[] args) {
    HelloSwing helloSwing = new HelloSwing();
    helloSwing.go();
  }
}
```

Basic Swing Application

- Importing Swing packages
- Choosing the look and feel:
  - `getLookAndFeel()`
- Setting up a `Window` container
  - `JFrame` is similar to `Frame`
  - You cannot add components directly to `JFrame`
  - *A content pane* contains all of the `Frame`'s visible components except menu bar

Basic Swing Application

- Setting up Swing components:
  - `HelloSwing.java` example instantiates four Swing components: `JFrame`, `JButton`, `JLabel`, and `JPanel`
- Supporting assistive technologies:
  - `HelloSwing.java` example code supports assistive technologies

```
accContext = jButton.getAccessibleContext();
accContext.setAccessibleDescription(
    "Pressing this button increments " + "the number of button clicks.");
```

Building a Swing GUI

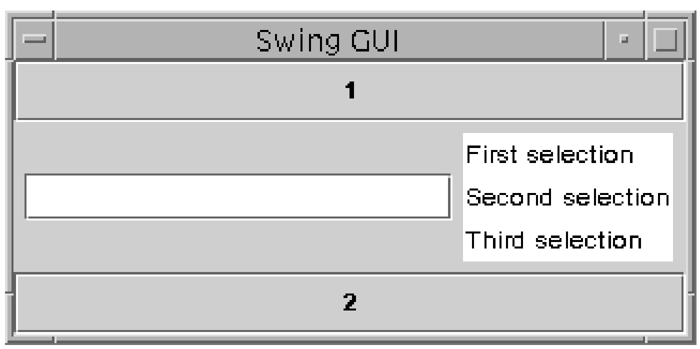
- Top-level containers (`JFrame`, `JApplet`, `JDialog`, and `JWindow`)
- Lightweight components (such as `JButton`, `JPanel`, and `JMenu`)
- Swing components are added to a content pane associated with a top-level container

Building a Swing GUI

```
import java.awt.*;
import com.sun.java.swing.*;

public class SwingGUI {
  private JFrame topLevel;
  private JPanel jPanel;
  private JTextField jTextField;
  private JList jList;

  private JButton b1;
  private JButton b2;
  private Container contentPane;

  private Object listData[] = {
    new String("First selection"),
    new String("Second selection"),
    new String("Third selection")
  };

  public void go() {
    topLevel = new JFrame("Swing GUI");

    // Set up the JPanel, which contains the text field
    // and list.
    jPanel = new JPanel();
    jTextField = new JTextField(20);
    jList = new JList(listData);

    contentPane = topLevel.getContentPane();
    contentPane.setLayout(new BorderLayout());

    b1 = new JButton("1");
    b2 = new JButton("2");
    contentPane.add(b1, BorderLayout.NORTH);
    contentPane.add(b2, BorderLayout.SOUTH);
```

Building a Swing GUI

```

    jPanel.setLayout(new FlowLayout());
    jPanel.add(jTextField);
    jPanel.add(jList);
    contentPane.add(jPanel, BorderLayout.CENTER);

    topLevel.pack();
    topLevel.setVisible(true);
  }

  public static void main (String args[]) {
    SwingGUI swingGUI = new SwingGUI();
    swingGUI.go();
  }
}
```



The `JComponent` Class

- Swing components that are subclasses of `JComponent`
- Borders
- Double buffering
- Tool tips
- Keyboard navigation
- Application-wide pluggable look and feel

Exercise: Creating Swing Applications

- Exercise objective:
  - Write, compile, and run a simple and an advanced Swing GUI program using Swing components
- Tasks:
  - Create a basic Swing application
  - Create a text editor using Swing

Check Your Progress

- Identify the key features of Java Foundation Classes
- Describe the key features of `com.sun.java.swing` package
- Identify Swing components
- Define *containers* and *components*, and explain how they work together to build a Swing GUI
- Write, compile, and run a basic Swing application
- Use top-level containers, such as `JFrame` and `JApplet` effectively

Think Beyond

- You now know how to program GUI applications. Suppose you want to run a GUI application using a Web browser. How is this done?

Module 12

Introduction to Java Applets

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

- Differentiate between a standalone application and an applet
- Write an HTML tag to call a Java applet
- Describe the class hierarchy of the applet and AWT classes
- Create the `HelloWorld.Java` applet
- List the major methods of an applet
- Describe and use the painting model of AWT

Objectives

- Use applet methods to read images and files from URLs
- Use `<param>` tags to configure applets

Relevance

- What advantages do applets provide?

What Is an Applet?

A Java class that can be:

- Embedded within an HTML page and downloaded and executed by a Web browser
- Loaded using the browser as follows:
  - Load URL
  - Load the HTML document
  - Load applet classes
  - Run the applet

Applet Security Restrictions

- Most browsers prevent the following:
  - Runtime execution of another program
  - File I/O (input/output)
  - Calls to any native methods
  - Attempts to open a socket to any system except the host that provided the applet

Applet Class Hierarchy

- java.lang.Object
  - java.awt.Component
    - java.awt.Container
      - java.awt.Window
        - java.awt.Frame
      - java.awt.Panel
        - java.applet.Applet

Key Applet Methods

- `init()`
- `start()`
- `stop()`
- `destroy()`
- `paint()`

Applet Display

- Applets are graphical in nature
- The browser environment calls the `paint()` method

```
import java.awt.*;
import java.applet.*;

public class HelloWorld extends Applet {
  public void paint(Graphics g){
    g.drawString("Hello World!", 25, 25);
  }
}
```

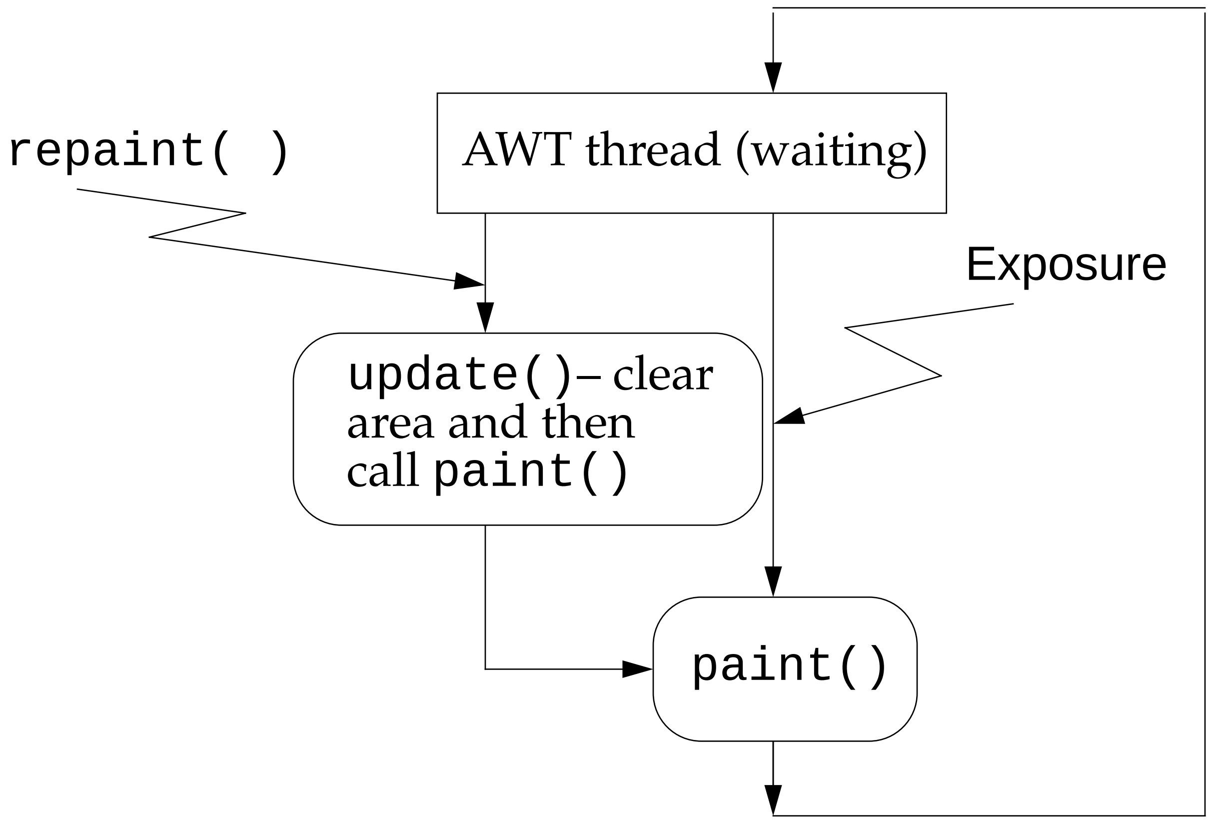
Applet Methods and the Applet Life Cycle

- `init()`
  - Called when the applet is created
  - Can be used to initialize data values
- `start()`
  - Called when the applet becomes visible
- `stop()`
  - Called when the applet becomes invisible

AWT Painting

- `paint(Graphics g)`
- `repaint()`
- `update(Graphics g)`

AWT Painting



Applet Display Strategies

- Maintain a model of the display
- Use `paint()` to render the display based only on the model
- Update the model and call `repaint()` to change the display

What Is the `appletviewer`?

A Java application that:

- Enables you to run applets without using a Web browser
- Loads the HTML file supplied as an argument

```
appletviewer HelloWorld.html
```

- Needs at least the following HTML code:

```
<applet code=HelloWorld.class width=100 height=100>
</applet>
```

The `applet` Tag

```
<applet
[archive=archiveList]
code=appletFile.class
width=pixels height=pixels
[codebase=codebaseURL]
[alt=alternateText]
[name=appletInstanceName]
[align=alignment]
[vspace=pixels] [hspace=pixels]
>
[<param name=appletAttribute1 value=value>]
[<param name=appletAttribute2 value=value>]
    . . .
[alternateHTML]
</applet>
```

Additional Applet Facilities

- `getDocumentBase()` – Returns a `URL` object that describes the directory of the current browser page
- `getCodeBase()` – Returns a `URL` object that describes the source directory of the applet class
- `getImage(URL base, String target)` and `getAudioClip(URL base, String target)` – Use the `URL` object as a starting point

A Simple Image Test

```
// Applet which shows an image of Duke in surfing mode

import java.awt.*;
import java.applet.Applet;

public class HwImage extends Applet {
  Image duke;

  public void init() {
    duke = getImage(getDocumentBase(),
        "graphics/surferDuke.gif");
  }

  public void paint(Graphics g) {
    g.drawImage(duke, 25, 25, this);
  }
}
```

AudioClip

- Playing a clip

```
play(URL soundDirectory, String soundFile);

play(URL soundURL);
```

A Simple Audio Test

```
// Applet which plays a sound once

import java.awt.Graphics;
import java.applet.Applet;

public class HwAudio extends Applet {
  public void paint(Graphics g) {
    g.drawString("Audio Test", 25, 25);
    play(getCodeBase(), "sounds/cuckoo.au");
  }
}
```

Looping an `AudioClip`

- Loading an `AudioClip`
- Playing an `AudioClip`
- Stopping an `AudioClip`

A Simple Looping Test

```
// Applet which continuously repeats a sound

import java.awt.Graphics;
import java.applet.*;

public class HwLoop extends Applet {
  AudioClip sound;

  public void init() {
    sound = getAudioClip(getCodeBase(), "sounds/cuckoo.au");
  }

  public void paint(Graphics g) {
    g.drawString("Audio Test", 25, 25);
  }

  public void start() {
    sound.loop();
  }

  public void stop() {
    sound.stop();
  }
}
```

Mouse Input

- `mouseClicked` – The mouse has been clicked (mouse button pressed and then released in one motion)
- `mouseEntered` – The mouse cursor enters a component
- `mouseExited` – The mouse cursor leaves a component
- `mousePressed` – The mouse button is pressed down
- `mouseReleased` – The mouse button is later released

A Simple Mouse Test

```
// This applet is HelloWorld extended to watch for mouse
// input. "Hello World!" is reprinted at the location of
// the mouse press.

import java.awt.Graphics;
import java.awt.event.*;
import java.applet.Applet;

public class HwMouse
    extends Applet
    implements MouseListener {

  private int mouseX = 25;
  private int mouseY = 25;

  // Register this applet instance to catch
  // MouseListener events.
  public void init() {
    addMouseListener(this);
  }

  public void paint(Graphics g) {
    g.drawString("Hello World!", mouseX, mouseY);
  }

  // Process the mousePressed MouseListener event
  public void mousePressed(MouseEvent evt) {
    mouseX = evt.getX();
    mouseY = evt.getY();
    repaint();
  }

  // We are not using the other mouse events.
  public void mouseClicked(MouseEvent e) { }
  public void mouseEntered(MouseEvent e) { }
  public void mouseExited(MouseEvent e) { }
  public void mouseReleased(MouseEvent e) { }
}
```

Reading Parameters

- Applet code

```
<html>
<applet code=DrawAny.class width=200 height=200>
<param name=image value="graphics/duke.gif">
</applet>
</html>
```

- Program code

```
import java.awt.*;
import java.applet.*;

public class DrawAny extends Applet {
  Image im;

  public void init() {
    String imageName = getParameter("image");

    if ( imageName == null ) {
      System.out.println(
          "Error: Cannot find image");
      System.exit(0);
    }

    im = getImage(getDocumentBase(), imageName);
  }

  public void paint(Graphics g) {
    g.drawImage(im, 0, 0, this);
  }
}

```

Dual Purpose Code Sample

```
// Applet/Application which shows an image of
// Duke in surfing mode

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class AppletApp extends Applet {
  Date date;

  public void init() {
    date = new Date();
  }

  public void paint (Graphics g) {
    g.drawString("This Java program started at", 25, 25);
    g.drawString(date.toString(), 25, 60);
  }

  // An application will require a main()
  public static void main (String args[]) {

    // Create a Frame to house the applet
    Frame frame = new Frame("Application");

    // Create an instance of the class (applet)
    AppletApp app = new AppletApp();

    // Add it to the center of the frame
    frame.add(app, BorderLayout.CENTER);
    frame.setSize (250, 150);

```

Dual Purpose Code Sample

```
    // Register the AppletApp class as the
    // listener for a Window Destroy event
    frame.addWindowListener (new WindowAdapter() {
      public void windowClosing (WindowEvent e) {
        System.exit(0);
      }
    });

    // Call the applet methods
    app.init();
    app.start();
    frame.setVisible(true);  // Invokes paint()
  }
}
```

Exercise: Creating Applets

- Exercise objective:
  - Become familiar with programming Java applets
- Tasks:
  - Write an applet
  - Create concentric squares
  - Create a rollover applet

Check Your Progress

- Differentiate between a standalone application and an applet
- Write an HTML tag to call a Java applet
- Describe the class hierarchy of the applet and AWT classes
- Create the `HelloWorld.Java` applet
- List the major methods of an applet
- Describe and use the painting model of AWT
- Use applet methods to read images and files from URLs

Check Your Progress

- Use `<param>` tags to configure applets
- Use the URL object to fetch sounds and images into your applet
- Handle various mouse events within the applet
- Pass parameters to an applet from an HTML file using the `<param>` tags

Think Beyond

- How can you use applets on your company's Web page to improve the overall presentation?

Module 13

Threads

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

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

Objectives

- Use `synchronized` to protect data from corruption
- Explain why `suspend()`, `resume()`, and `stop()` methods have been deprecated in JDK 1.2

Relevance

- How do you get programs to perform multiple tasks?

Threads

- What are threads?
  - Virtual CPU

Three Parts of a Thread

- CPU
- Code
- Data

Creating the Thread

```
public class ThreadTest {
  public static void main(String args[]) {
    Xyz r = new Xyz();
    Thread t = new Thread(r);
    t.start();
  }
}

class Xyz implements Runnable {
  int i;

  public void run() {
    i = 0;

    while (true) {
      System.out.println("Hello " + i++);
      if ( i == 50 ) {
        break;
      }
    }
  }
}
```

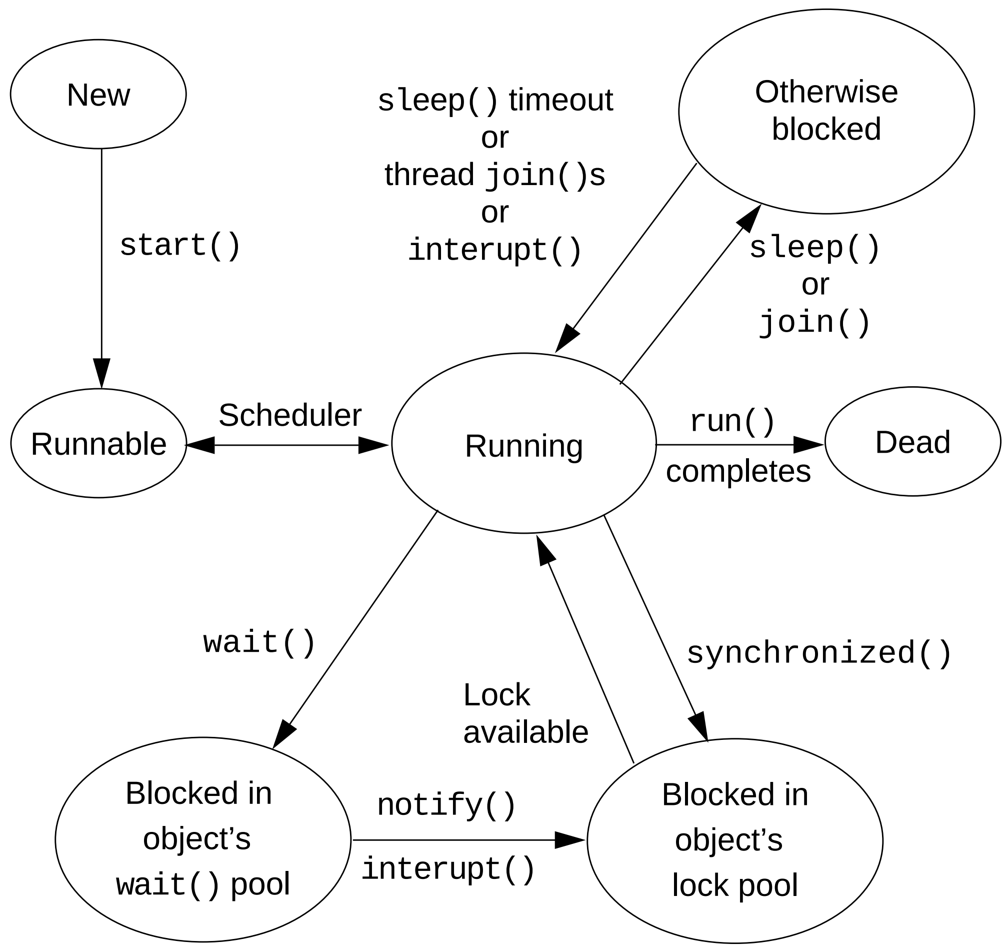
Creating the Thread

- Multithreaded programming:
  - Multiple threads from the same `Runnable` instance
  - Threads share the same data and code

Starting the Thread

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

Thread Scheduling



Thread Scheduling

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

Terminating a Thread

```
public class Xyz implements Runnable {
  private boolean timeToQuit=false;

  public void run() {
    while(! timeToQuit) {
      ...
    }
    // clean up before run() ends
  }

  public void stopRunning() {
    timeToQuit=true;
  }
}

public class ControlThread {
  private Runnable r = new Xyz();
  private Thread t = new Thread(r);

  public void startThread() {
    t.start();
  }

  public void stopThread() {
    // use specific instance of Xyz
    r.stopRunning();
  }
}
```

Basic Control of Threads

- Testing threads:
  - `isAlive()`
- Putting threads on hold:
  - `sleep()`
  - `join()`

Putting Threads on Hold

```
public class Xyz implements Runnable {
...
  public void run() {
    while (running) {
      // do your task
      try {
        Thread.sleep((int)(Math.rando() * 100));
      } catch (InterruptedException e) {
        // somebody woke me up
      }
      ...
    }
  }
}

public class TTest {
  public static void main(String args[]) {
    Runnable r = new Xyz();
    Thread t1 = new Thread(r);
    t1.start();
  }
}
```

Putting Threads on Hold

```
public void doTask() {
  TimerThread tt = new TimerThread (100);
  tt.start ();
  ...
  // Do stuff in parallel with the other thread for
  // a while
  ...
  // Wait here for the timer thread to finish
  try {
    tt.join ();
  } catch (InterruptedException e) {
    // tt came back early
  }
  ...
  // Now continue in this thread
  ...
}
```

Extending the Thread Class

```
public class MyThread extends Thread {
  public void run() {
    while (running) {
      // do lots of interesting stuff
      try {
        sleep(100);
      } catch (InterruptedException e) {
        // sleep interrupted
      }
    }
  }

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

Selecting a Way to Create Threads

- Implementing `Runnable`:
  - Better object-oriented design
  - Single inheritance
  - Consistency
- Extending `Thread`:
  - Simpler code

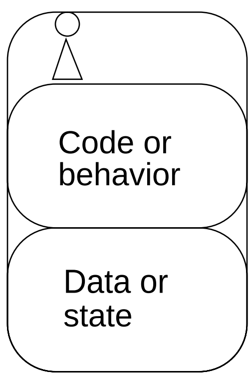
Using the `synchronized` Keyword

```
public class MyStack {
  int idx = 0;
  char [] data = new char[6];

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

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

The Object Lock Flag

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

Object `this`

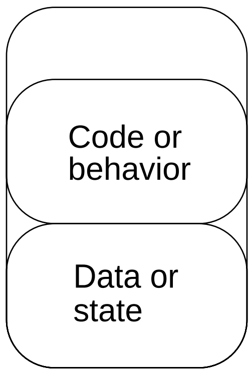


Thread before `synchronized(this)`

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

The Object Lock Flag

Object `this`
Lock flag missing



Waiting for object lock

Thread, trying to execute `synchronized(this)`

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

Releasing the Lock Flag

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

`synchronized` – Putting It Together

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

`synchronized` – Putting It Together

- The following two code segments are equivalent:

```
public void push(char c) {
  synchronized(this) {
    :
    :
  }
}
```

```
public synchronized void push(char c) {
  :
  :
}
```

Deadlock

- Is two threads, each waiting for a lock from the other
- Is not detected or avoided
- Can be avoided by:
  - Deciding on the order to obtain locks
  - Adhering to this order throughout
  - Releasing locks in reverse order

Thread Interaction – `wait()` and `notify()`

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

Thread Interaction

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

Monitor Model for Synchronization

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

Producer

```
public void run() {
  char c;

  for (int i = 0; i < 200; i++) {
    c = (char)(Math.random() * 26 +'A');
    theStack.push(c);
    System.out.println("Producer" + num + ": " + c);
    try {
      Thread.sleep((int)(Math.random() * 300));
    } catch (InterruptedException e) {
      // ignore it
    }
  }
}
```

Consumer

```
public void run() {
  char c;
  for (int i = 0; i < 200; i++) {
    c = theStack.pop();
    System.out.println("Consumer" + num + ": " + c);

    try {
      Thread.sleep((int)(Math.random() * 300));
    } catch (InterruptedException e) { }

  }
}
```

SyncStack Class

```
public class SyncStack {
  private Vector buffer = new Vector(400, 200);

  public synchronized char pop() {
  }

  public synchronized void push(char c) {
  }
```

pop( ) Method

```
public synchronized char pop() {
  char c;
  while (buffer.size() == 0) {
    try {
      this.wait();
    } catch (InterruptedException e) {
      // ignore it...
    }
  }
  c = ((Character)buffer.remove(buffer.size()-1)).
      charValue();
  return c;
}
```

push() Method

```
public synchronized void push(char c) {
   this.notify();
   Character charObj = new Character(c);
   buffer.addElement(charObj);
}
```

SyncTest.java

```
package mod13;

public class SyncTest {

  public static void main(String[] args) {

    SyncStack stack = new SyncStack();

    Producer p1 = new Producer(stack);
    Thread prodT1 = new Thread (p1);
    prodT1.start();

    Producer p2 = new Producer(stack);
    Thread prodT2 = new Thread (p2);
    prodT2.start();

    Consumer c1 = new Consumer(stack);
    Thread consT1 = new Thread (c1);
    consT1.start();

    Consumer c2 = new Consumer(stack);
    Thread consT2 = new Thread (c2);
    consT2.start();
  }
}
```

Producer.java

```java
package mod13;

public class Producer implements Runnable {
  private SyncStack theStack;
  private int num;
  private static int counter = 1;

  public Producer (SyncStack s) {
    theStack = s;
    num = counter++;
  }

  public void run() {
    char c;
    for (int i = 0; i < 200; i++) {
      c = (char)(Math.random() * 26 +'A');
      theStack.push(c);
      System.out.println("Producer" + num + ": " + c);
      try {
        Thread.sleep((int)(Math.random() * 300));
      } catch (InterruptedException e) {
        // ignore it
      }
    }
  }
}
```

Consumer.java

```
package mod13;

public class Consumer implements Runnable {
  private SyncStack theStack;
  private int num;
  private static int counter = 1;

  public Consumer (SyncStack s) {
    theStack = s;
    num = counter++;
  }

  public void run() {
    char c;
    for (int i = 0; i < 200; i++) {
      c = theStack.pop();
      System.out.println("Consumer" + num + ": " + c);

      try {
        Thread.sleep((int)(Math.random() * 300));
      } catch (InterruptedException e) { }

    }
  }
}
```

SyncStack.java

```
package mod13;

import java.util.Vector;

public class SyncStack {
  private Vector buffer = new Vector(400, 200);

  public synchronized char pop() {
    char c;
    while (buffer.size() == 0) {
      try {
        this.wait();
      } catch (InterruptedException e) {
        // ignore it...
      }
    }
    c = ((Character)buffer.remove(buffer.size()-1)).
        charValue();
    return c;
  }

  public synchronized void push(char c) {
    this.notify();
    Character charObj = new Character(c);
    buffer.addElement(charObj);
  }
}
```

The `suspend()` and `resume()` Methods

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

The `stop()` Method

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

Proper Thread Control

```
public class ControlledThread extends Thread {
  static final int SUSP = 1;
  static final int STOP = 2;
  static final int RUN = 0;
  private int state = RUN;

  public synchronized void setState(int s) {
    state = s;
    if ( s == RUN ) {
      notify();
    }
  }

  public synchronized boolean checkState() {
    while ( state == SUSP ) {
      try {
        wait();
      } catch (InterruptedException e) {
        // ignore
      }
    }
    if ( state == STOP ) {
      return false;
    }
    return true;
  }

  public void run() {
    while ( true ) {
      // doSomething();

      // Be sure shared data is in consistent state in
      // case the thread is waited or marked for exiting
      // from run()
      if ( !checkState() ) {
        break;
      }
    }
  }} // just to fit it on this page
```

Exercise: Using Multithreaded Programming

- Exercise objectives:
  - Become familiar with the concepts of multithreading by writing some multithreaded programs
  - Create a multithreaded applet
- Tasks:
  - Create three threads
  - Incorporate animation

Check Your Progress

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

Check Your Progress

- Use `synchronized` to protect data from corruption
- Explain why `suspend()`, `resume()`, and `stop()` methods have been deprecated in JDK 1.2

Think Beyond

- Do you have applications that could benefit from being multithreaded?

Module 14

Stream I/O and Files

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

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

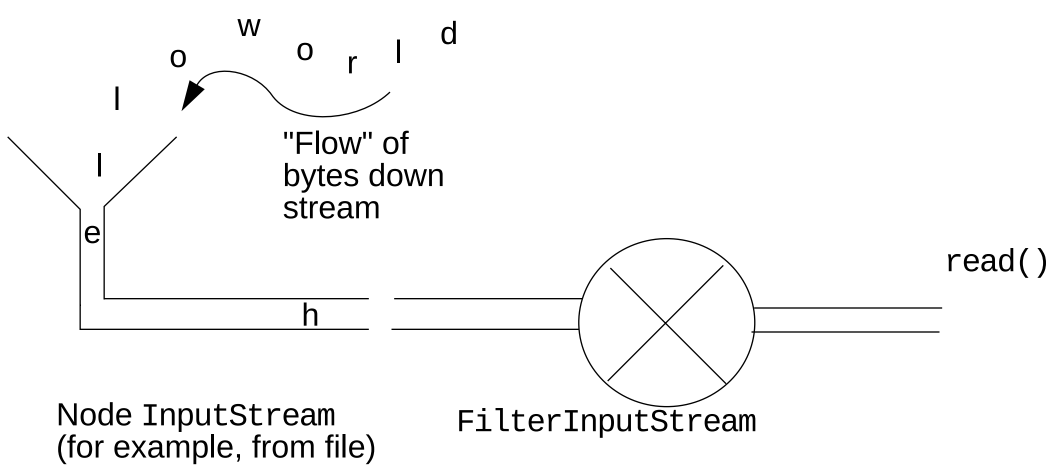
Relevance

- What mechanisms are in place within the Java programming language to read and write from files?

Stream I/O

- A *stream* is either a source of bytes or a destination for bytes.
- The two basic types of streams are:
  - Input stream
  - Output stream
- *Node* streams read from or write to a specific place.
- *Filter* streams use *node* streams as input or output.

Stream Fundamentals



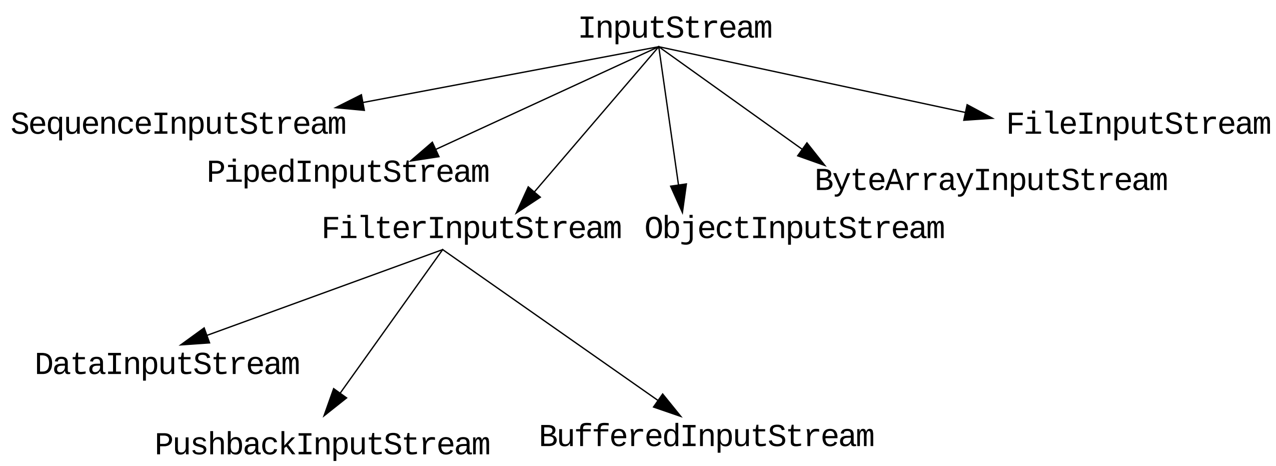
`InputStream` Methods

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

`OutputStream` Methods

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

Basic Stream Classes



Basic Stream Classes

- `FileInputStream` and `FileOutputStream`
- `BufferedInputStream` and `BufferOutputStream`
- `DataInputStream` and `DataOutputStream`
- `PipedInputStream` and `PipedOutputStream`

URL Input Streams

```
java.net.URL imageSource;

try {
  imageSource = new URL("http://mysite.com/~info");
} catch (MalformedURLException e) {
  // ignore
}

images[0] = getImage(imageSource, "Duke/T1.gif");
```

Opening an Input Stream

```
InputStream is = null;
String datafile = new String("Data/data.1-96");
byte buffer[] = new byte[24];
try {
  // new URL throws a MalformedURLException
  // URL.openStream() throws an IOException
  is = (new URL(getDocumentBase(),datafile)).openStream();
} catch (Exception e) {
  // ignore
}
```

Now you can use `it` to read information, just as with a `FileInputStream` object:

```
try {
  is.read(buffer, 0, buffer.length);
} catch (IOException e1) {
  // ignore
}
```

Readers and Writers

- The Java programming language uses *Unicode* to represent strings and characters.
- `InputStreamReader` and `OutputStreamWriter` convert Unicode to platform-specific code.
- Chain `BufferedReader` and `BufferedWriter` to `InputStreamReader` and `OutputStreamWriter` for efficiency.

Reading String Input

```
import java.io.*;

public class CharInput {

  public static void main (String args[])
      throws java.io.IOException {
    String s;
    InputStreamReader ir;
    BufferedReader in;

    ir = new InputStreamReader(System.in);
    in = new BufferedReader(ir);

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

Creating a New `File` Object

- `File myFile;`
- `myFile = new File("mymotd");`
- `myFile = new File("/", "mymotd");`
- ```
  // more useful if the directory or filename
  // is a variable
  File myDir = new File("/");
  myFile = new File(myDir, "mymotd");
  ```
  ```

`File` Tests and Utilities

- `File` names:

```
String getName()
String getPath()
String getAbsolutePath()
String getParent()
boolean renameTo(File newName)
```

- `File` tests:

```
boolean exists()
boolean canWrite()
boolean canRead()
boolean isFile()
boolean isDirectory()
boolean isAbsolute();
```

`File` Tests and Utilities

- General file information and utilities:

```
long lastModified()
long length()
boolean delete()
```

- Directory utilities:

```
boolean mkdir()
String[] list()
```

Creating a Random Access File

- With the file name:

```
myRAFile = new RandomAccessFile(
                  String name, String mode);
```

- With a `File` object:

```
myRAFile = new RandomAccessFile(
                  File file, String mode);
```

Random Access Files

- `long getFilePointer()`
- `void seek(long pos)`
- `long length()`

Serialization

- Saving an object to permanent storage is called *persistence.*
- Only the object's data are serialized.
- Data marked with the `transient` keyword are not serialized.

```
public class MyClass implements Serializable {
  public transient Thread myThread;
    private String customerID;
    private int total;
}
```

```
public class MyClass implements Serializable {
  public transient Thread myThread;
  private transient String customerID;
  private int total;
}
```

Writing an Object to a File Stream

```
import java.io.*;
import java.util.Date;

public class SerializeDate {

  SerializeDate() {
    Date d = new Date ();

    try {
      FileOutputStream f =
          new FileOutputStream ("date.ser");
      ObjectOutputStream s =
          new ObjectOutputStream (f);
      s.writeObject (d);
      s.close ();
    } catch (IOException e) {
      e.printStackTrace ();
    }
  }

  public static void main (String args[]) {
    new SerializeDate();
  }
}
```

Reading an Object From a File Stream

```
import java.io.*;
import java.util.Date;

public class UnSerializeDate {

  UnSerializeDate () {
    Date d = null;

    try {
      FileInputStream f =
          new FileInputStream ("date.ser");
      ObjectInputStream s =
          new ObjectInputStream (f);
      d = (Date) s.readObject ();
      s.close ();
    } catch (Exception e) {
      e.printStackTrace ();
    }

    System.out.println(
      "Unserialized Date object from date.ser");
    System.out.println("Date: "+d);
  }

  public static void main (String args[]) {
    new UnSerializeDate();
  }
}
```

Exercise: Getting Acquainted With I/O

- Exercise objective:
  - Become familiar with stream I/O by writing programs that perform I/O to files
- Tasks:
  - Open a file
  - Create a simple database program
  - Use persistence

Check Your Progress

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

Think Beyond

- Do you have applications that require file I/O?

Module 15

Networking

Course Map

The Java Programming Language Basics

- Getting Started
- Identifiers, Keywords, and Types
- Expressions and Flow Control
- Arrays

Object-Oriented Programming

- Objects and Classes
- Advanced Language Features

Exception Handling

- Exceptions

Developing Graphical User Interfaces

- Building GUIs
- The AWT Event Model
- The AWT Component Library
- Java Foundation Classes

Applets

- Introduction to Java Applets

Multithreading

- Threads

Communications

- Stream I/O and Files
- Networking

Objectives

- Create a minimal Transmission Control Protocol/ Internet Protocol (TCP/IP) server and a minimal TCP/ IP client:
  - `ServerSocket`
  - `Socket`
- Create a minimal User Datagram Protocol (UDP) server and a minimal UDP client:
  - `DatagramSocket`
  - `DatagramPacket`

Relevance

- How can a communication link between a client machine and a server on the network be established?

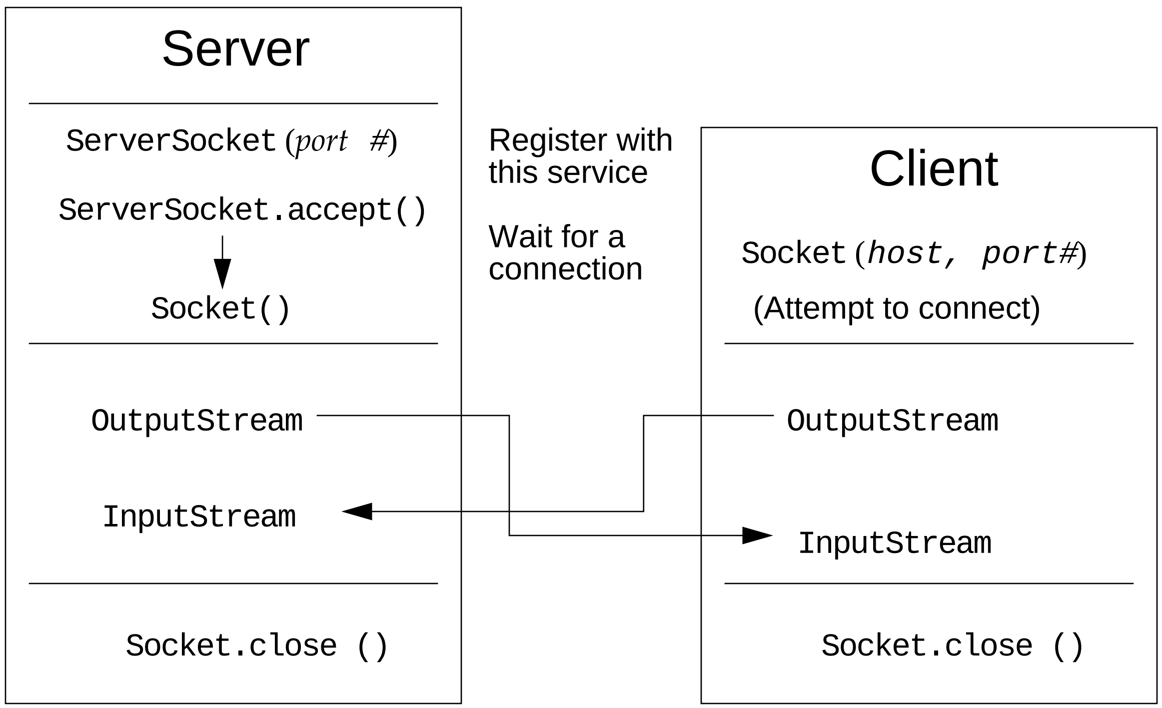
Networking

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

Networking With Java Technology

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

Java Networking Model



Minimal TCP/IP Server

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

public class SimpleServer {
  public static void main(String args[]) {
    ServerSocket s = null;
    Socket s1;
    String sendString = "Hello Net World!";
    int slength = sendString.length();
    OutputStream s1out;
    DataOutputStream dos;

    // Register your service on port 5432
    try {
      s = new ServerSocket(5432);
    } catch (IOException e) {
      // ignore
    }

```

Minimal TCP/IP Server

```
    // Run the listen/accept loop forever
    while (true) {
      try {
        // Wait here and listen for a connection
        s1=s.accept();

        // Get a communication stream associated with
        // the socket
        s1out = s1.getOutputStream();
        dos = new DataOutputStream (s1out);

        // Send your string!
        // (UTF provides machine independence)
        dos.writeUTF(sendString);

        // Close the connection, but not the server socket
        dos.close();
        s1out.close();
        s1.close();
      } catch (IOException e) {
        // ignore
      }
    }
  }
}
```

Minimal TCP/IP Client

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

public class SimpleClient {

  public static void main(String args[])
      throws IOException {
    int c;
    Socket s1;
    InputStream s1In;
    DataInputStream dis;

    // Open your connection to a server, at port 5432
    // localhost used here
    s1 = new Socket("127.0.0.1",5432);

    // Get an input file handle from the socket and
    // read the input
    s1In = s1.getInputStream();
    dis = new DataInputStream(s1In);

    String st = new String (dis.readUTF());
    System.out.println(st);

    // When done, just close the connection and exit
    dis.close();
    s1In.close();
    s1.close();
  }
}
```

UDP Sockets

- Are used for connection-less protocol
- Messages are not guaranteed
- Are supported in Java technology through the `DatagramSocket` and `DatagramPacket` classes

The `DatagramPacket`

`DatagramPacket` has two constructors: one for receiving data and one for sending data.

- ```
  DatagramPacket(
          byte [] recvBuf, int readLength)
  ```

- ```
  DatagramPacket(
          byte [] sendBuf, int sendLength,
              InetAddress iaddr, int iport)
  ```

The `DatagramSocket`

`DatagramSocket` has three constructors:

- `DatagramSocket()`
- `DatagramSocket(int port)`
- `DatagramSocket(int port, InetAddress iaddr)`

Minimal UDP Server

```
import java.io.*;
import java.net.*;
import java.util.*;

public class UdpServer{

  //This method retrieves the current time on the server
  public byte[] getTime(){
    Date d= new Date();
    return d.toString().getBytes();
  }

  // Main server loop.
  public void go() throws IOException {

    DatagramSocket datagramSocket;
    // Datagram packet from the client
    DatagramPacket inDataPacket;
    // Datagram packet to the client
    DatagramPacket outDataPacket;
    // Client return address
    InetAddress clientAddress;
    // Client return port
    int clientPort;
    // Incoming data buffer.  Ignored.
    byte[] msg= new byte[10];
    // Stores retrieved time
    byte[] time;

    // Allocate a socket to man port 8000 for requests.
    datagramSocket = new DatagramSocket(8000);
    System.out.println("UDP server active on port 8000");

```

Minimal UDP Server

```
      // Loop forever
      while(true) {

        // Set up receiver packet.  Data will be ignored.
        inDataPacket = new DatagramPacket(msg, msg.length);

        // Get the message.
        datagramSocket.receive(inDataPacket);

        // Retrieve return address information, including
        // InetAddress and port from the datagram packet
        // just recieved.

        clientAddress = inDataPacket.getAddress();
        clientPort = inDataPacket.getPort();

        // Get the current time.
        time = getTime();

        //set up a datagram to be sent to the client using the
        //current time, the client address and port
        outDataPacket =
            new DatagramPacket(
                time, time.length, clientAddress, clientPort);

        //finally send the packet
        datagramSocket.send(outDataPacket);
      }
    }

```

Minimal UDP Server

```
  public static void main(String args[]) {
    UdpServer udpServer = new UdpServer();

    try {
      udpServer.go();
    } catch (IOException e) {
      System.out.println(
          "IOException occured with socket.");
      System.out.println (e);
      System.exit(1);
    }
  }
}
```

Minimal UDP Client

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

public class UdpClient {

  public void go()
      throws IOException,
      UnknownHostException {

    DatagramSocket datagramSocket;
    // Datagram packet to the server
    DatagramPacket outDataPacket;
    // Datagram packet from the server
    DatagramPacket inDataPacket;
    // Server host address
    InetAddress serverAddress;
    // Buffer space.
    byte[] msg = new byte[100];
    // Received message in String form.
    String receivedMsg;

    // Allocate a socket by which messages are sent
    // and received.
    datagramSocket = new DatagramSocket();

    // Server is running on this same machine for this
    // example.
    // This method can throw an UnknownHostException.
    serverAddress = InetAddress.getLocalHost();

    // Set up a datagram request to be sent to the server.
    // Send to port 8000.
    outDataPacket =
        new DatagramPacket(msg, 1, serverAddress, 8000);

    // Make the request to the server.
    datagramSocket.send(outDataPacket);

```

Minimal UDP Client

```
      // Set up a datagram packet to receive
      // server's response.
      inDataPacket = new DatagramPacket(msg, msg.length);

      // Receive the time data from the server
      datagramSocket.receive(inDataPacket);

      // Print the data received from the server
      receivedMsg = new String(
        inDataPacket.getData(), 0, inDataPacket.getLength());
      System.out.println(receivedMsg);

      //close the socket
      datagramSocket.close();
    }

    public static void main(String args[]) {
      UdpClient udpClient = new UdpClient();

      try {
        udpClient.go();
      } catch (Exception e) {
        System.out.println ("Exception occured with socket.");
        System.out.println (e);
        System.exit(1);
      }
    }
  }
```

Exercise: Using Socket Programming

- Exercise objective:
  - Gain experience using sockets by implementing a client and server which communicate using sockets
- Tasks:
  - Create sockets
  - Use a multithreaded server

Check Your Progress

- Develop code to set up network connection
- Understand TCP/IP and UDP protocol
- Use `ServerSocket` and `Socket` classes for implementing TCP/IP client and servers
- Use `DatagramPacket` and `DatagramSocket` for effecting a UDP-based network communication

Think Beyond

- There are several advanced Java platform topics, many of which are addressed in other SunEd courses. See Appendix A for a brief discourse on some of them. Be sure and check out the JavaSoft web site (`www.javasoft.com`) as well.

Appendix B

Using the `GridBagLayout`

Layout Managers

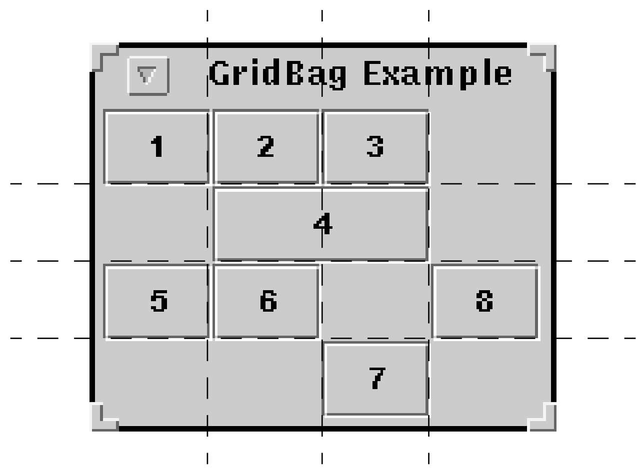
- Position and size components in a `Container`
- Adhere to a policy
- Make absolute coordinates platform dependent
- Determine limitations of:
  - `FlowLayout`
  - `GridLayout`
  - `BorderLayout`

The `GridBagLayout`

- Divides the region into rows and columns
- Sizes components to fit width, height, both, or neither of their *regions* (one or more contiguous rows and one or more contiguous columns)

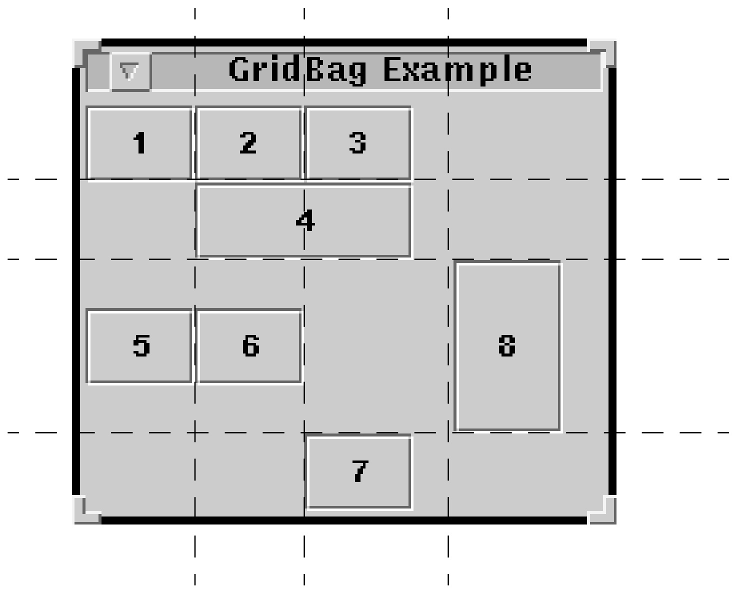
The `GridBagLayout`

- Row/column count determined by cell usage
- Row/column basic size determined by contents



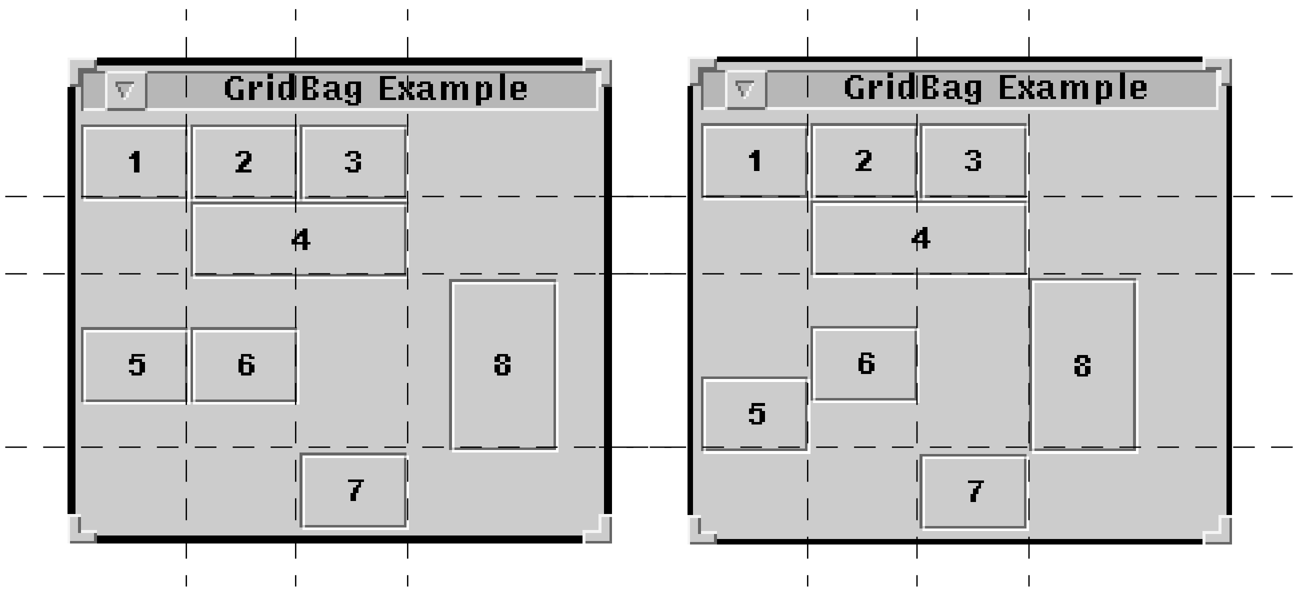
The `GridBagLayout`

- Use of "spare" space is determined by weight.
- Components can fit width, height, or both of the region.



The `GridBagLayout`

- Components are located within a region by an anchor.
- Fill can make the anchor ineffective.



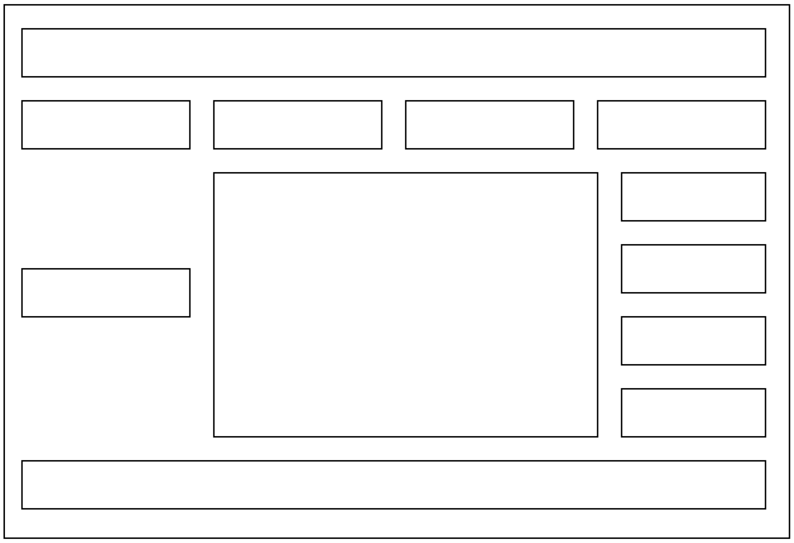
The `GridBagConstraints` Class

- For each component, specify:
  - Top left corner of cell with `gridx` and `gridy`
  - Cell size with `gridwidth` and `gridheight`
  - Capacity with `fill`
  - `anchor`
- For each row and column, specify:
  - Capacity with `weightx` and `weighty`

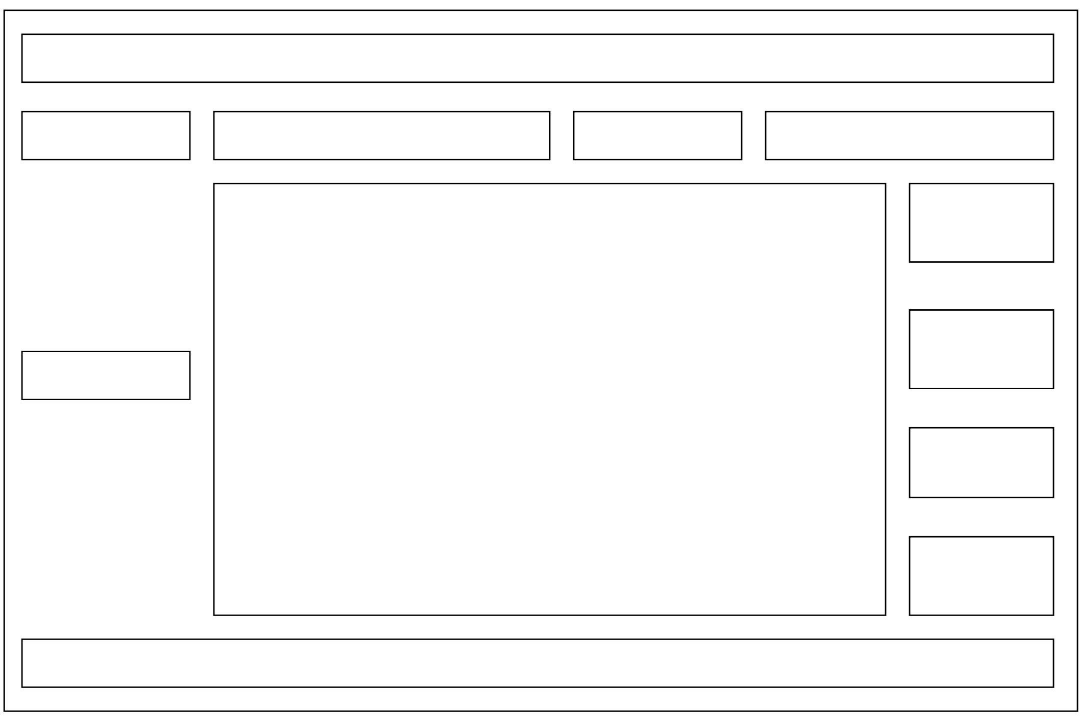
Designing with `GridBagLayout`

- Sketch all components
- Sketch all components on resized container
- Identify all gridlines and rowhence/column counts
- Identify stretchy rows/columns and allocate weights
- Identify starting row/column for each component
- Identify width/height for each component
- Identify `fill` for each component
- Identify `anchor` for each component
- Define row/column weights for each component

Example

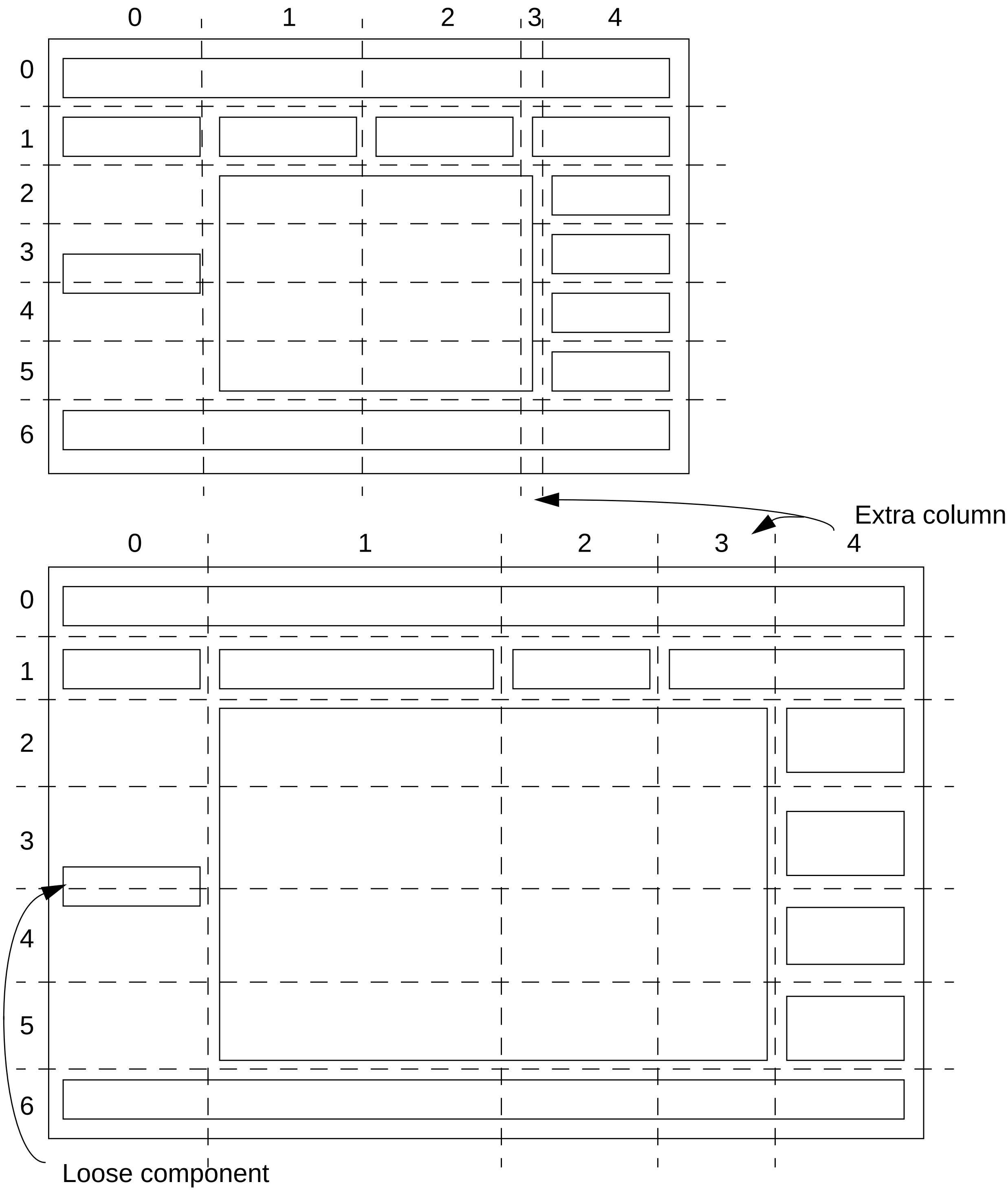


Basic, unexpanded layout proposal

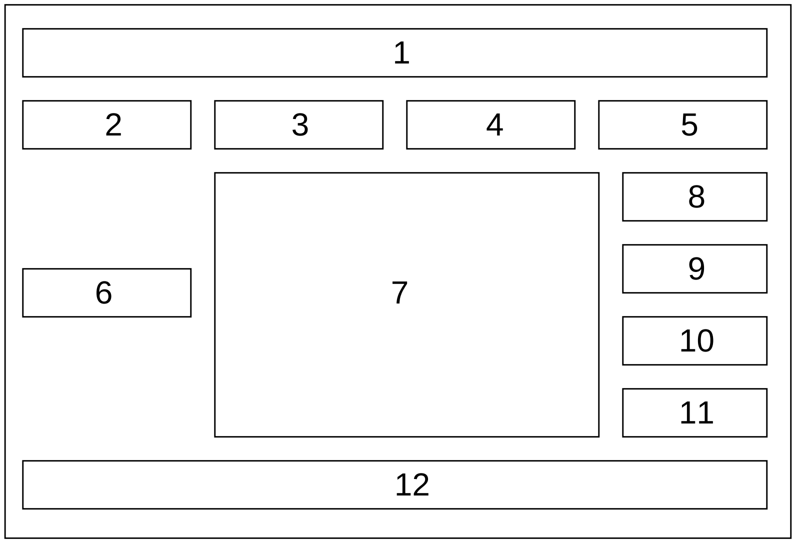


Basic, expanded layout proposal

Example

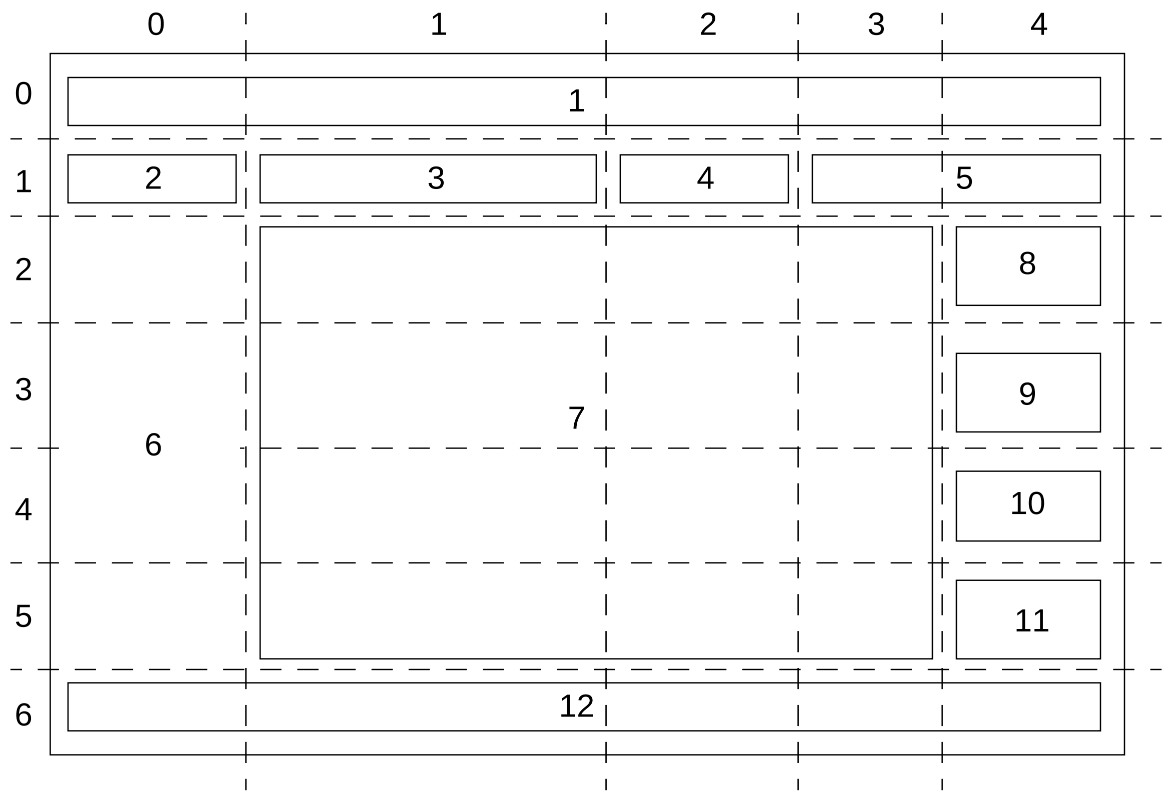


Example

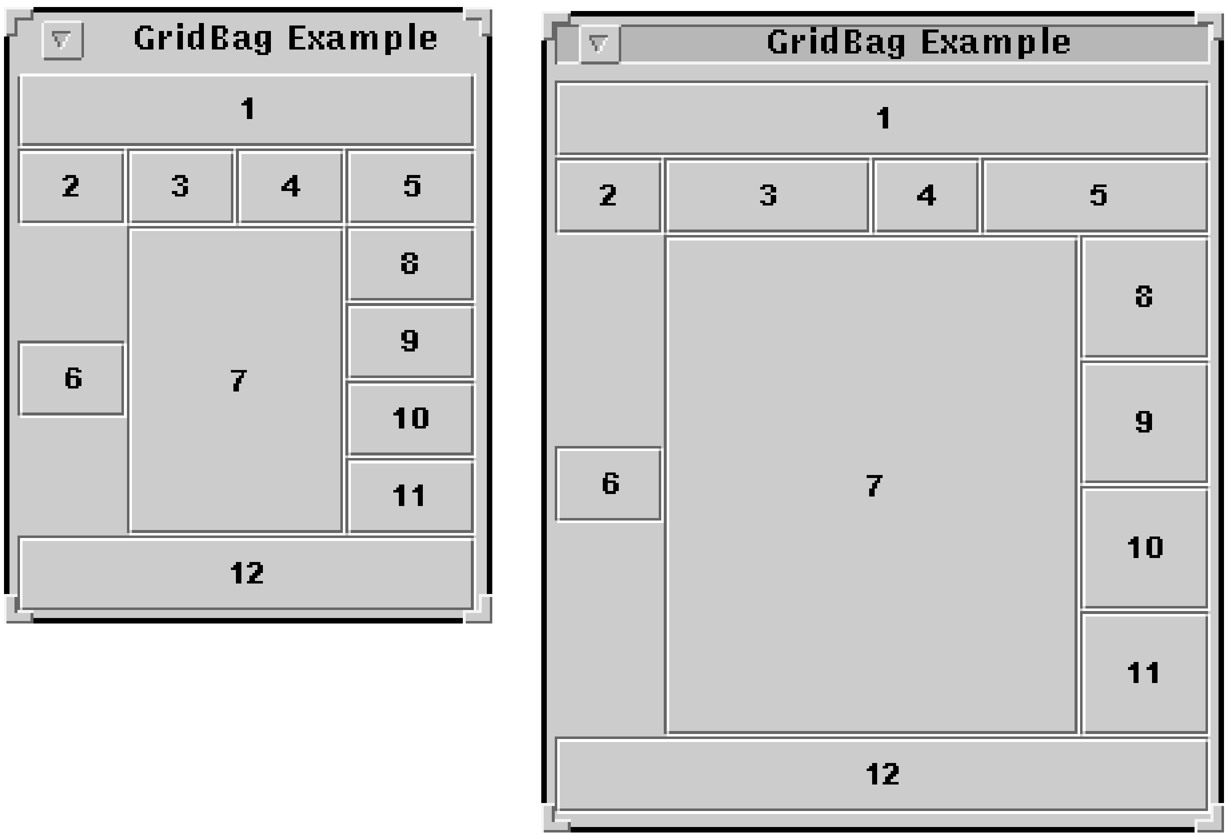


Component	`gridx`	`gridy`	`gridwidth`	`gridheight`
1	0	0	5	1
2	0	1	1	1
3	1	1	1	1
4	2	1	1	1
5	3	1	2	1
6	0	2	1	4
7	1	2	3	4
8	4	2	1	1
9	4	3	1	1
10	4	4	1	1
11	4	5	1	1
12	0	6	5	1

Example



Example


GridBag Example
1
2
3
4
5
6
7
8
9
10
11
12
GridBag Example
1
2
3
4
5
6
7
8
9
10
11
12

Example

```
import java.awt.*;
import com.sun.java.swing.*;

public class ExampleGB {
  public static void main(String args[]) {
    JFrame f = new JFrame(“GridBag Example”);
    Container c = f.getContentPane();
    c.setLayout(new GridBagLayout());
    GridBagAdder.add(c, new Canvas(), 3, 2, 1, 1, 1, 0,
      GridBagConstraints.NONE, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton(“1”), 0, 0, 5, 1, 0, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton(“2”), 0, 1, 1, 1, 0, 0,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton(“3”), 1, 1, 1, 1, 1, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton(“4”), 2, 1, 1, 1, 0, 0,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton(“5”), 3, 1, 2, 1, 0, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton(“6”), 0, 2, 1, 4, 0, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton(“7”), 1, 2, 3, 4, 0, 0,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton(“8”), 4, 2, 1, 1, 0, 1,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton(“9”), 4, 3, 1, 1, 0, 1,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton(“10”), 4, 4, 1, 1, 0, 1,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton(“11”), 4, 5, 1, 1, 0, 1,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton(“12”), 0, 6, 5, 1, 0, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    f.pack();
    f.setVisible(true);
  }
```

Example

```
static class GridBagAdder {
  // OK to reuse this as we overwrite all elements every time
  // Note that this is not threadsafe however!
  static GridBagConstraints cons = new GridBagConstraints();
  public static void add(Container cont, Component comp, int x, int y,
   int width, int height, int weightx, int weighty, int fill, int anchor) {
   cons.gridx = x;
   cons.gridy = y;
   cons.gridwidth = width;
   cons.gridheight = height;
   cons.weightx = weightx;
   cons.weighty = weighty;
   cons.fill = fill;
   cons.anchor = anchor;
   cont.add(comp, cons);
  }
 }
}
```

`RELATIVE` and `REMAINDER`

- Shorthand for position, size, or both
- For `gridx/gridy`:
  `RELATIVE` => extends to the next position
- For `gridwidth/gridheight`:
  `RELATIVE` => extends to last one
- For `gridwidth/gridheight`:
  `REMAINDER` => extends to last one
- Careful use of these helps maintenance, but it:
  - Makes adding order significant
  - Might decrease readability of code

Think Beyond

- Are there any layout effects that you cannot handle using the layout managers you now understand?