Java™ Programming Language

SL-275

Sun microsystems®

THIS MANUAL IS DESIGNED TO SUPPORT AN INSTRUCTOR-LED TRAINING (ILT) COURSE AND IS INTENDED TO BE USED FOR REFERENCE PURPOSES IN CONJUNCTION WITH THE ILT COURSE. THE MANUAL IS NOT A STANDALONE TRAINING TOOL. USE OF THE MANUAL FOR SELF-STUDY WITHOUT CLASS ATTENDANCE IS NOT RECOMMENDED.

CE MANUEL DE RÉFÉRENCE DOIT ÊTRE UTILISÉ DANS LE CADRE D'UN COURS DE FORMATION DIRIGÉ PAR UN INSTRUCTEUR (ILT). IL NE S'AGIT PAS D'UN OUTIL DE FORMATION INDÉPENDANT. NOUS VOUS DÉCONSEILLONS DE L'UTILISER DANS LE CADRE D'UNE AUTO-FORMATION.

Preface

About This Course

Course Goals

This course provides you with knowledge and skills to:

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

Course Overview

This course describes the following areas:

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

Course Map

The Java Programming Language Basics

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

More Object-Oriented Programming

- Class Design
- Advanced Class Features

Building Applications

- Exceptions and Assertions
- Text-Based Applications

- Building Java GUIs
- GUI Event Handling
- GUI-Based Applications

Advanced Java Programming

- Threads
- Advanced I/O Streams
- Networking

Module-by-Module Overview

- Module 1 – "Getting Started"
- Module 2 – "Object-Oriented Programming"
- Module 3 – "Identifiers, Keywords, and Types"
- Module 4 – "Expressions and Flow Control"
- Module 5 – "Arrays"
- Module 6 – "Class Design"
- Module 7 – "Advanced Class Features"
- Module 8 – "Exceptions and Assertions"
- Module 9 – "Text-Based Applications"

Module-by-Module Overview

- Module 10 – “Building Java GUIs”
- Module 11 – “GUI Event Handling”
- Module 12 – “GUI-Based Applications”
- Module 13 – “Threads”
- Module 14 – “Advanced I/O Streams”
- Module 15 – “Networking”

Course Objectives

- Describe key language features
- Compile and run a Java technology application
- Use the online hypertext Java technology documentation
- Describe language syntactic elements and constructs
- Describe the object-oriented paradigm
- Use object-oriented features of the Java programming language
- Use exceptions
- Use the Collections API
- Read and write to files

Course Objectives

- Develop a graphical user interface (GUI)
- Describe the Java technology Abstract Window Toolkit (AWT)
- Develop a program to take input from a GUI
- Describe event handling
- Use the `java.io` package
- Describe the basics of multithreading
- Develop multi threaded Java technology applications
- Develop Java client and server programs using TCP/IP

Guidelines for Module Pacing

Module	Day 1	Day 2	Day 3	Day 4	Day 5
About This Course	A.M.				
Module 1 – “Getting Started”	A.M.				
Module 2 – “Object-Oriented Programming”	P.M.				
Module 3 – “Identifiers, Keywords, and Types”	P.M.				
Module 4 – “Expressions and Flow Control”		A.M.			
Module 5 – “Arrays”		A.M.			
Module 6 – “Class Design”		P.M.			
Module 7 – “Advanced Class Features”			A.M.		
Module 8 – “Exceptions and Assertions”			A.M.		
Module 9 – “Text-Based Applications”			P.M.		
Module 10 – “Building Java GUIs”				A.M.	
Module 11 – “GUI Event Handling”				A.M.	
Module 12 – “GUI-Based Applications”				P.M.	
Module 13 – “Threads”					A.M.
Module 14 – “Advanced I/O Streams”					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: *Fundamentals of the Java™ Programming Language*

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

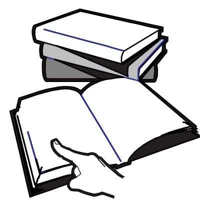
- Name
- Company affiliation
- Title, function, and job responsibility
- Programming experience
- Reasons for enrolling in this course
- Expectations for this course

How to Use Course Materials

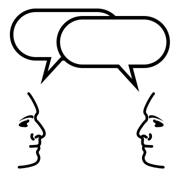
- Course Map
- Relevance
- Overhead Image
- Lecture
- Exercise
- Check Your Progress
- Think Beyond

Course Icons

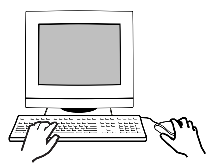
- Reference



- Discussion



- Exercise



Typographical Conventions

- `Courier` – Commands, files, 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

Objectives

- Describe key features of Java technology
- Write, compile, and run a simple Java technology application
- Describe the Java™ virtual machine's (JVM™ machine's) function
- Define garbage collection
- List the three tasks performed by the Java platform that handle code security

Relevance

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

What Is the Java Technology?

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

Primary Goals of the Java Technology

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

Primary Goals of the Java Technology

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

Primary Goals of the Java Technology

The following features fulfill these goals:

- The 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

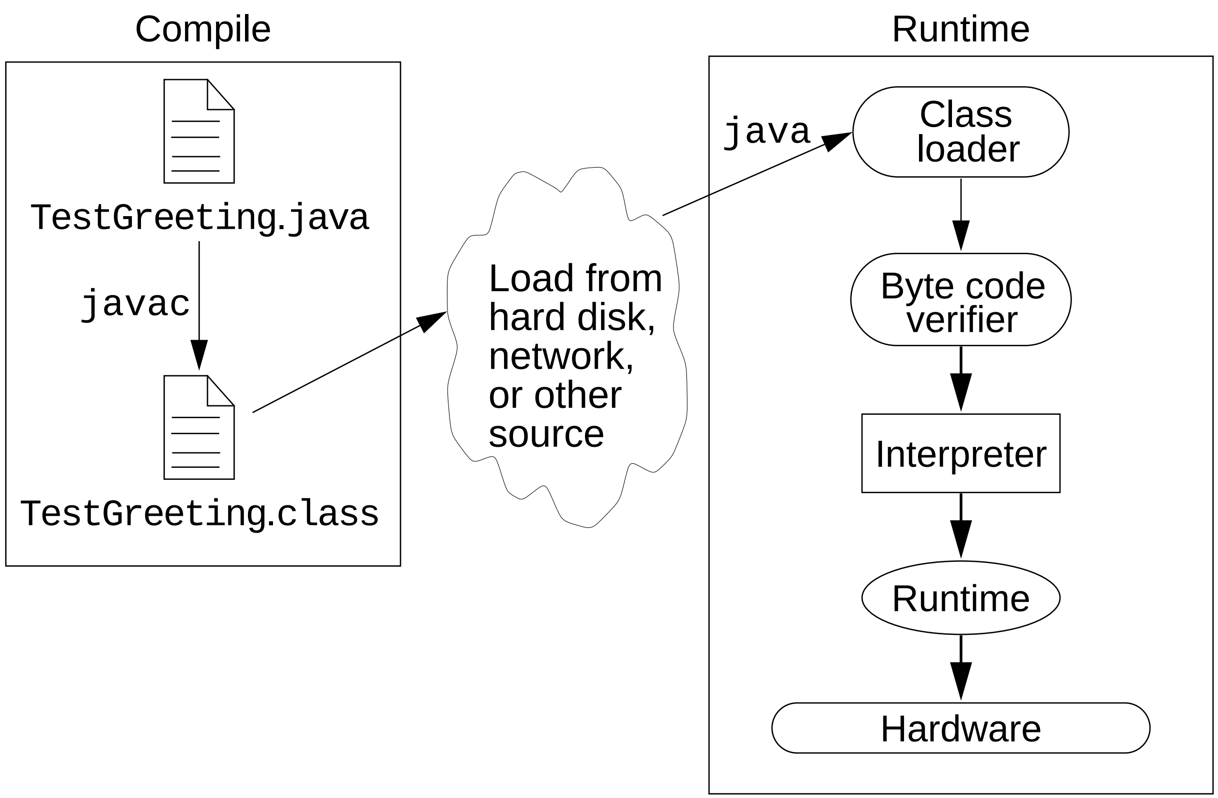
- The majority of type checking is done when the code is compiled.
- Implementation of the JVM approved by Sun Microsystems must be able to run any compliant class file.

Garbage Collection

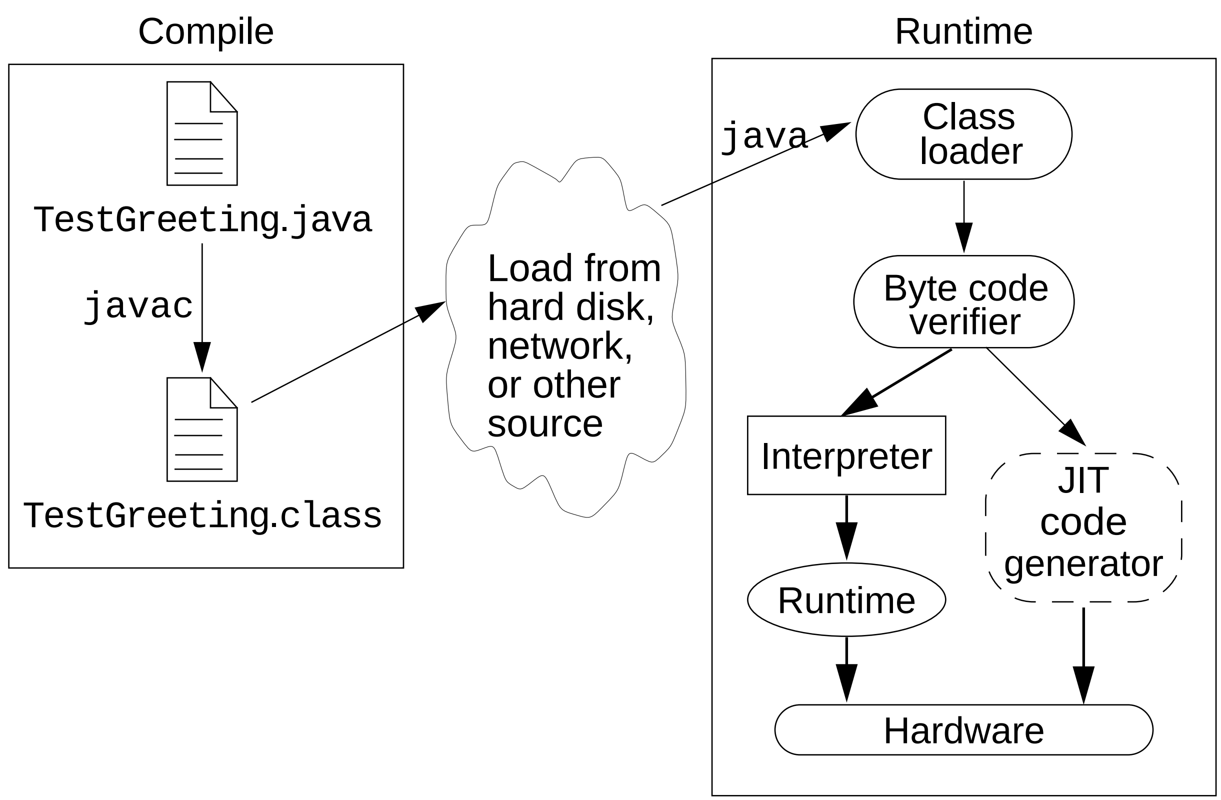
- Allocated memory that is no longer needed should be deallocated
- In other languages, deallocation is the programmer's responsibility
- The Java programming language provides a system-level thread to track memory allocation
- Garbage collection:
  - Checks for and frees memory no longer needed
  -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

Code Security

The Java application environment performs as follows:



Just-In-Time (JIT) Code Generator



The Java™ Runtime Environment

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

The Class Loader

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

The Bytecode Verifier

Ensures that:

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

A Basic Java Application

TestGreeting.java

```
//
// Sample "Hello World" application
//
public class TestGreeting{
  public static void main (String[] args) {
    Greeting hello = new Greeting();
    hello.greet();
  }
}
```

Greeting.java

```
// The Greeting class declaration.
public class Greeting {
  public void greet() {
    System.out.println(“hi”);
  }
}
```

Compiling and Running the `TestGreeting` Program

- Compiling `TestGreeting.java`

```
javac TestGreeting.java
```

- `Greeting.java` is compiled automatically
- Running an application

```
java TestGreeting
```

- Locating common compile and runtime errors

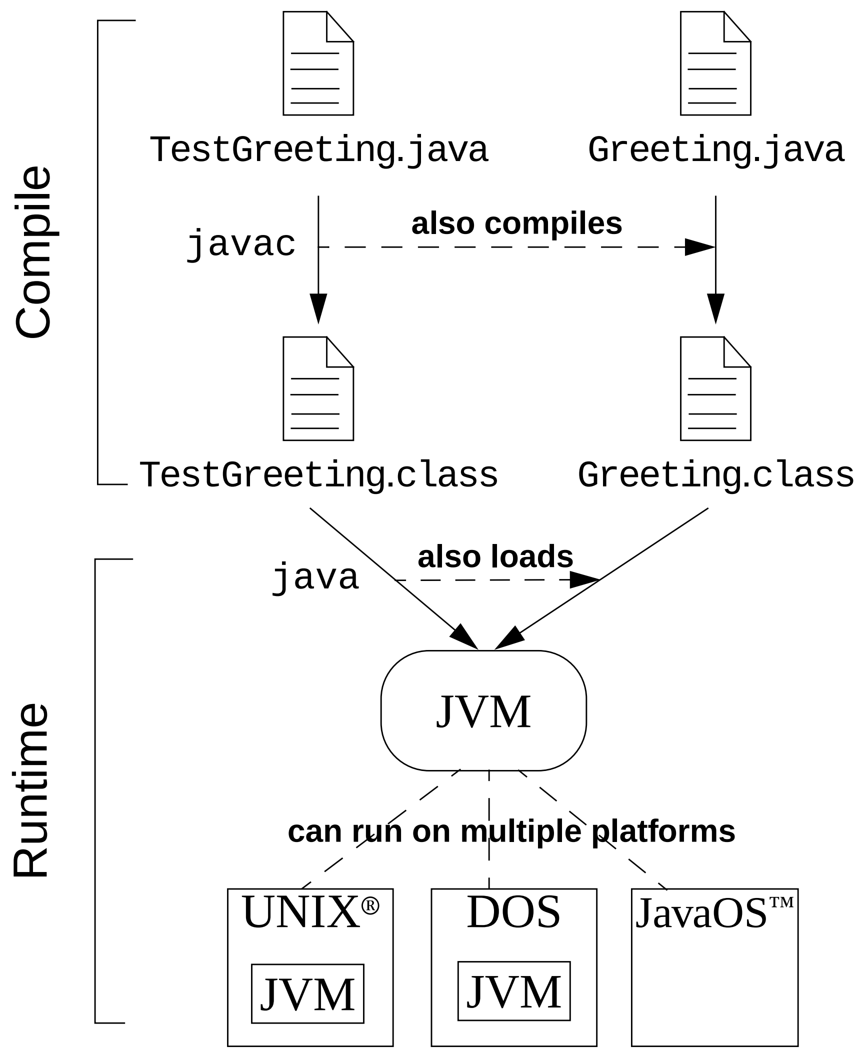
Compile-Time Errors

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

Runtime Errors

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

Java Runtime Environment



Exercise Performing Basic Tasks

- Exercise objectives:
  - Solve compilation and runtime errors in provided example programs and write a simple program
- Tasks:
  - Analyze and fix compilation and runtime errors
  - Create an application

Check Your Progress

- Describe key features of Java technology
- Write, compile, and run a simple Java application
- Describe the JVM machine’s function
- Define garbage collection
- List the three tasks performed by the Java platform that handle code security

Think Beyond

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

Module 2

Object-Oriented Programming

Objectives

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

Objectives

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

Relevance

- What is your understanding of software analysis and design?
- What is your understanding of design and code reuse?
- What features does the Java programming language possess that make it an object-oriented language?
- Define the term *object-oriented*.

Software Engineering

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

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

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

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

Machine Code (late 1940s – up)

The Analysis and Design Phase

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

Abstraction

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

Classes as Blueprints for Objects

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

Declaring Java Technology Classes

- Basic syntax of a Java class:

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

- Example:

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

Declaring Attributes

- Basic syntax of an attribute:

```
[<modifiers>] <type> <name> [ = <initial_value>];
```

- Examples:

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

Declaring Methods

- Basic syntax of a method:

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

- Examples:

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

Accessing Object Members

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

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

Information Hiding

The Problem:

MyDate
+day +month +year

Client code has direct access to internal data (`d` refers to a `MyDate` object):

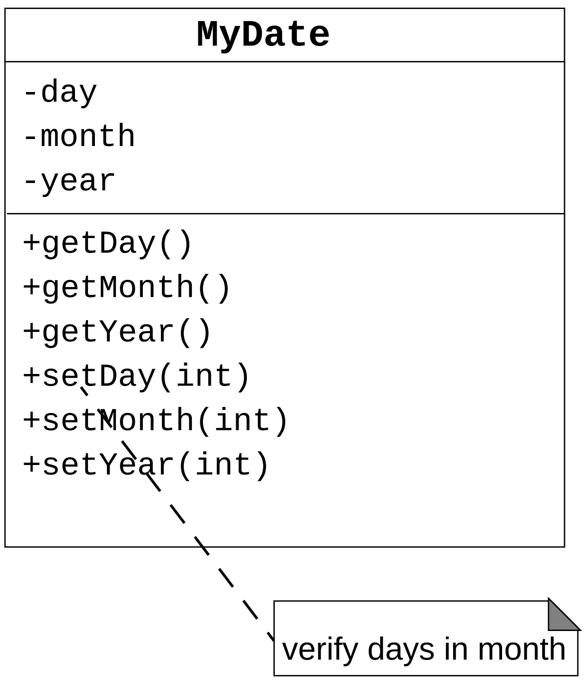
```
d.day = 32;
// invalid day

d.month = 2; d.day = 30;
// plausible but wrong

d.day = d.day + 1;
// no check for wrap around
```

Information Hiding

The Solution:



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

```
MyDate d = new MyDate();

d.setDay(32);
// invalid day, returns false

d.setMonth(2);
d.setDay(30);
// plausible but wrong, setDay returns false

d.setDay(d.getDay() + 1);
// this will return false if wrap around
// needs to occur
```

Encapsulation

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

MyDate
-date
+getDay() +getMonth() +getYear() +setDay(int) +setMonth(int) +setYear(int) -validDay(int)

Declaring Constructors

- Basic syntax of a constructor:

```
[<modifier>] <class_name> ([<argument_list>]) {
    [<statements>]
  }
```

- Example:

```
public class Dog {
  private int weight;

  public Dog() {
     weight = 42;
  }

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

The Default Constructor

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

Source File Layout

- Basic syntax of a Java source file:

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

- Example, the `VehicleCapacityReport.java` file:

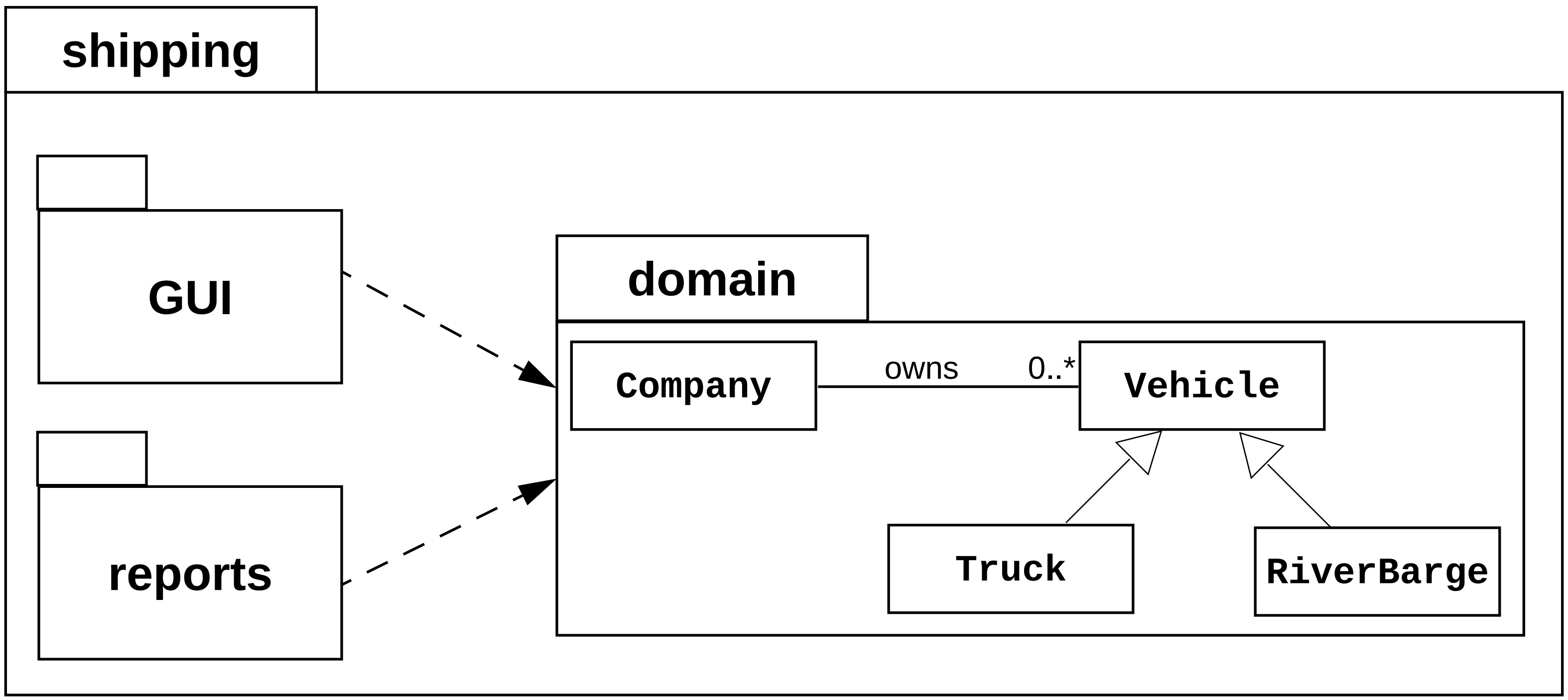
```
package shipping.reports;

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

public class VehicleCapacityReport {
  private List  vehicles;
  public void generateReport(Writer output) {...}
}
```

Software Packages

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



The package Statement

- Basic syntax of the package statement:

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

- Examples:

```
package shipping.reports;
```

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

The `import` Statement

- Basic syntax of the import statement:

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

- Examples:

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

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

Directory Layout and Packages

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

```
shipping/
   |
   +---- domain/
   |        |
   |        +---- Company.class
   |        +---- Vehicle.class
   |        +---- RiverBarge.class
   |        +---- Truck.class
   +---- GUI/
   +---- reports/
            |
            +---- VehicleCapacityReport.class
```

Development

```
JavaProjects/
    |
    |----- BankPrj/
    |        |
    |        |----- src/
    |        |        |
    |        |        |----- banking/
    |        |                  |
    |        |                  |----- domain/
    |        |                  |----- GUI/
    |        |                  |----- reports/
    |        |----- doc/
    |        |----- class/
    |                 |
    |                 |----- banking/
    |                           |
    |                           |----- domain/
    |                           |----- GUI/
    |                           |----- reports/
    |
    |----- Compiler/
             |
             |----- src/
             |----- doc/
             |----- class/
```

Compiling using `-d`

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

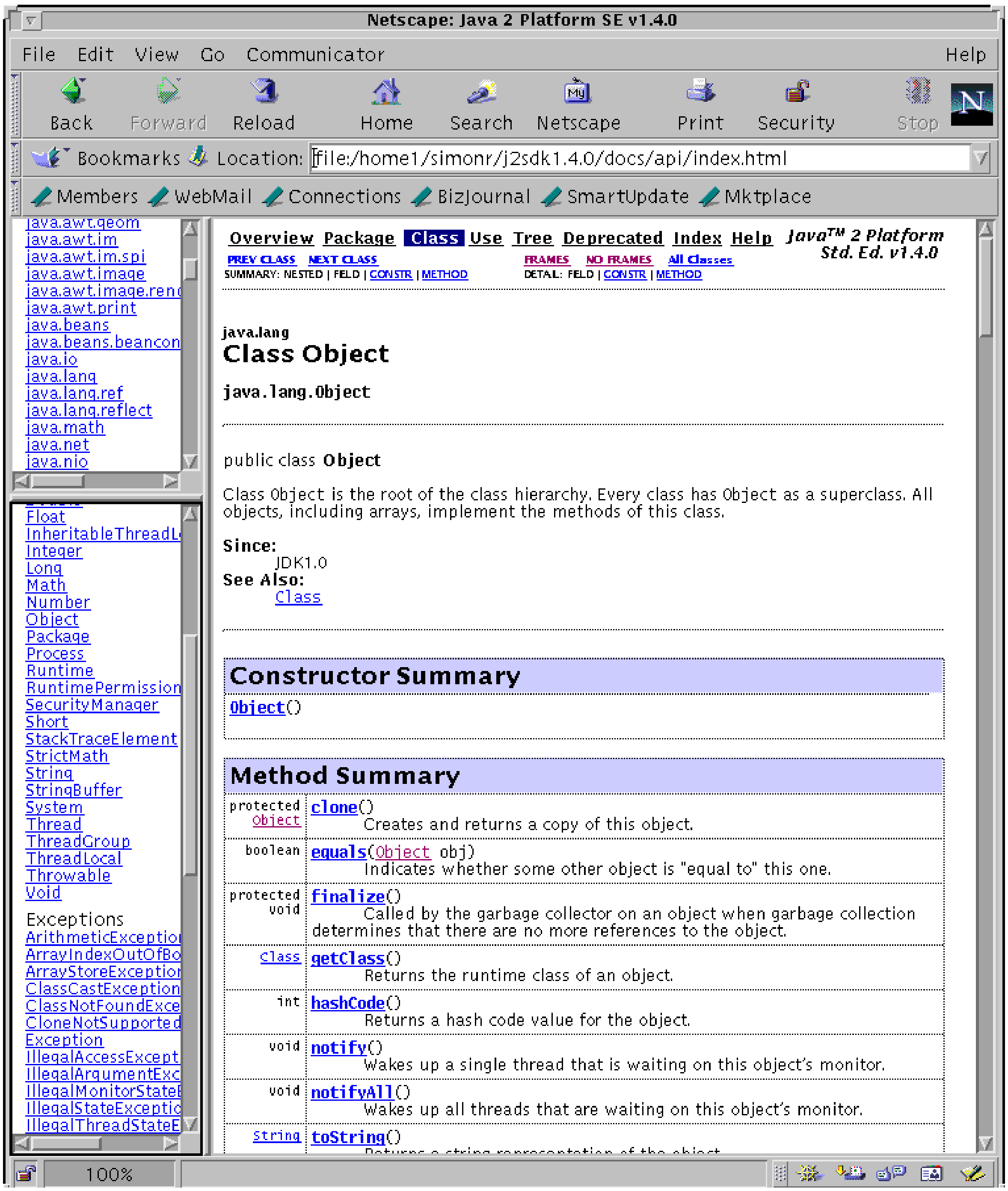
Terminology Recap

- Class – The source-code blueprint for a run-time object
- Object – An instance of a class
  Also known as: instance
- Attribute – A data element of an object
  Also known as: data member, instance variable, data field
- Method – A behavioral element of an object
  Also known as: algorithm, function, procedure
- Constructor – A "method-like" construct used to initialize a new object
- Package – A grouping of classes and/or sub-packages

Using the Java API Documentation

- A set of Hypertext Markup Language (HTML) files provides information about the API.
- A frame describes a package and contains hyperlinks to information describing each class in that package.
- A class document includes the class hierarchy, a description of the class, a list of member variables, a list of constructors, and so on.

Example API Documentation Page



Exercise: Using Objects and Classes

- Exercise objectives:
  - Implement the concepts presented in this module
- Tasks:
  - Complete the tasks specified by the instructor

Check Your Progress

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

Check Your Progress

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

Think Beyond

- What do you expect to achieve through analysis and design?
- What domain objects and relationships appear in your existing applications?

Module 3

Identifiers, Keywords, and Types

Objectives

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

Objectives

- Declare variables of class type
- Construct an object using `new`
- Describe default initialization
- Describe the significance of a reference variable
- State the consequences of assigning variables of class type

Relevance

- Do you know the primitive Java types?
- Can you describe the difference between variables holding primitive values as compared with object references?

Comments

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

```
// comment on one line

/* comment on one
   or more lines */

/** documentation comment */
```

Semicolons, Blocks, and White Space

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

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

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

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

Semicolons, Blocks, and White Space

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

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

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

Identifiers

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

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

Java Keywords

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

Primitive Types

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

Logical – `boolean`

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

```
boolean truth = true;
```

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

Textual – `char` and `String`

`char`

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

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

Textual – `char` and `String`

`String`

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

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

- Can be used as follows:

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

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

- Uses three forms – Decimal, octal, or hexadecimal

`2`	The decimal form for the integer 2.
`077`	The leading `0` indicates an octal value.
`0xBAAC`	The leading `0x` indicates a hexadecimal value.

- Literals have a default type of `int`
- Literals with the suffix `L` or `l` are of type `long`

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

- Integral data types have the following ranges:

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`

- 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

- Literals have a default type of `double`

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

Java Reference Types

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

```
public class MyDate {
  private int day = 1;
  private int month = 1;
  private int year = 2000;
  public MyDate(int day, int month, int year) { ... }
  public void print() { ... }
}
```

```
public class TestMyDate {
  public static void main(String[] args) {
    MyDate today = new MyDate(22, 7, 1964);
  }
}
```

Constructing and Initializing Objects

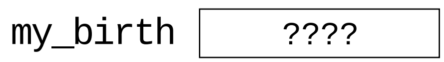
- Calling `new Xxx()` to allocate space for the new object results in:
  - Memory allocation: Space for the new object is allocated and instance variables are initialized to their default values (for example, `0`, `false`, `null`, and so on)
  - Explicit attribute initialization is performed
  - A constructor is executed
- The reference to the object is assigned to a variable
- Example:

```
MyDate my_birth = new MyDate(22, 7, 1964);
```

Memory Allocation and Layout

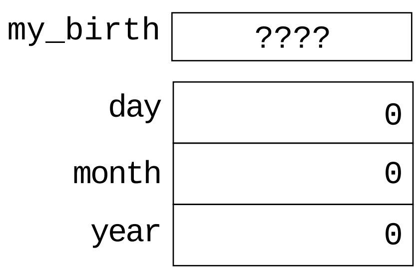
- A declaration allocates storage only for a reference:

```
MyDate my_birth = new MyDate(22, 7, 1964);
```



- Use the `new` operator to allocate space for `MyDate`:

```
MyDate my_birth = new MyDate(22, 7, 1964);
```



Explicit Attribute Initialization

- Initialize the attributes:

```
MyDate my_birth = new MyDate(22, 7, 1964);
```

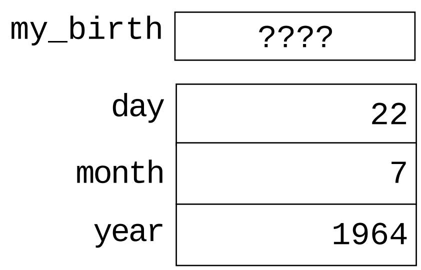
my_birth	????
day	1
month	1
year	2000

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

Executing the Constructor

- Execute the matching constructor:

```
MyDate my_birth = new MyDate(22, 7, 1964);
```

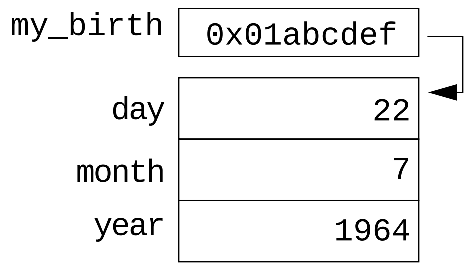


- In the case of an overloaded constructor, the first constructor may call another.

Assigning a Variable

- Assign the newly created object to the reference variable:

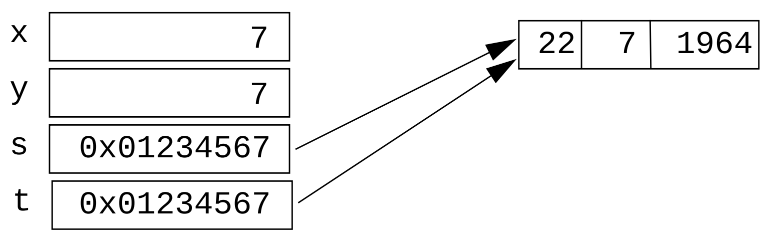
```
MyDate my_birth = new MyDate(22, 7, 1964);
```



Assigning References

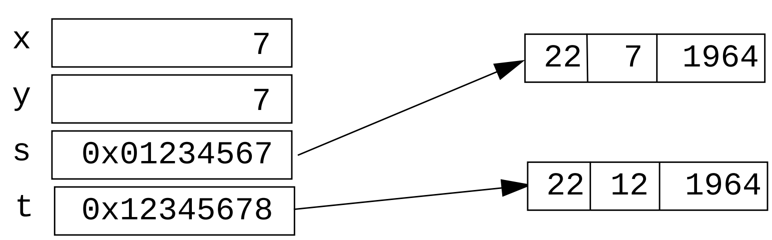
```
int x = 7;
int y = x;
MyDate s = new MyDate(22, 7, 1964);
MyDate t = s;
```

- Two variables refer to a single object:




```
t = new MyDate(22, 12, 1964);
```

- Reassignment makes two variables point to two objects:



Pass-by-Value

- In a single Virtual Machine, 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 original object reference is never changed.

Pass-by-Value

```
public class PassTest {

  // Methods to change the current values
  public static void changeInt(int value) {
    value = 55;
  }
  public static void changeObjectRef(MyDate ref) {
    ref = new MyDate(1, 1, 2000);
  }
  public static void changeObjectAttr(MyDate ref) {
    ref.setDay(4);
  }

  public static void main(String args[]) {
    MyDate date;
    int val;

    // Assign the int
    val = 11;
    // Try to change it
    changeInt(val);
    // What is the current value?
    System.out.println("Int value is: " + val);

    // Assign the date
    date = new MyDate(22, 7, 1964);
    // Try to change it
    changeObjectRef(date);
    // What is the current value?
    date.print();

    // Now change the day attribute
    // through the object reference
    changeObjectAttr(date);
    // What is the current value?
    date.print();
  }
}
```

The `this` Reference

Here are a few uses of the `this` keyword:

- Resolving ambiguity: To reference a member within code that has local variables or arguments with the same name as that member
- To pass the current object as a parameter to another method or constructor

The `this` Reference

```
public class MyDate {
  private int day = 1;
  private int month = 1;
  private int year = 2000;

  public MyDate(int day, int month, int year) {
    this.day   = day;
    this.month = month;
    this.year  = year;
  }
  public MyDate(MyDate date) {
    this.day   = date.day;
    this.month = date.month;
    this.year  = date.year;
  }

  public MyDate addDays(int more_days) {
    MyDate new_date = new MyDate(this);

    new_date.day = new_date.day + more_days;
    // Not Yet Implemented: wrap around code...

    return new_date;
  }
  public void print() {
    System.out.println("MyDate: " + day + "-" + month +
                       "-" + year);
  }
}
```

```
public class TestMyDate {
  public static void main(String[] args) {
    MyDate my_birth = new MyDate(22, 7, 1964);
    MyDate the_next_week = my_birth.addDays(7);

    the_next_week.print();
  }
}
```

Java Programming Language Coding Conventions

- Packages:

```
package banking.domain;
```

- Classes:

```
class SavingsAccount
```

- Interfaces:

```
interface Account
```

- Methods:

```
balanceAccount()
```

Java Programming Language Coding Conventions

- Variables:

```
currentCustomer
```

- Constants:

```
HEAD_COUNT
MAXIMUM_SIZE
```

Exercise: Using Objects

- Exercise objectives:
  - Implement the concepts presented in this module
- Tasks:
  - Complete the tasks specified by the instructor

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 *primitive variable* and *reference variable*

Check Your Progress

- Declare variables of class type
- Construct an object using `new`
- Describe default initialization
- Describe the significance of a reference variable
- State the consequences of assigning variables of class type

Think Beyond

- Can you think of examples of classes and objects in your existing applications?

Module 4

Expressions and Flow Control

Objectives

- Distinguish between instance and local variables
- Describe how to initialize instance variables
- Identify and correct a `Possible reference before assignment` compiler error
- Recognize, describe, and use Java software 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
- Variables that are created when the method is executed are destroyed when the method is exited
- Local variables require explicit initialization
- Member and class variables are automatically initialized

Variable Scope Example

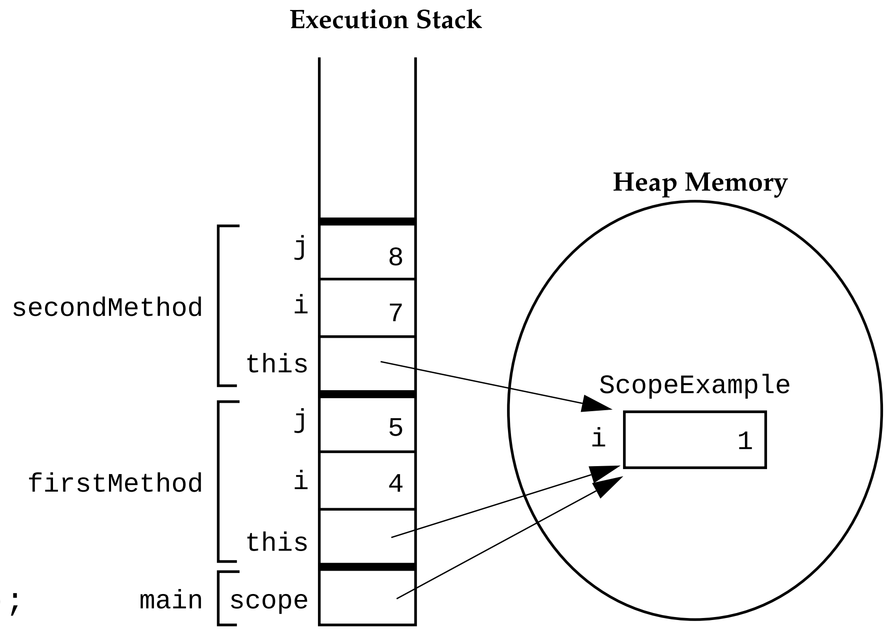
```
public class ScopeExample {
  private int i=1;

  public void firstMethod() {
    int i=4, j=5;

    this.i = i + j;
    secondMethod(7);
  }
  public void secondMethod(int i) {
    int j=8;
    this.i = i + j;
  }
}

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

    scope.firstMethod();
  }
}
```



Variable Initialization

Variable	Value
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	'\u0000'
boolean	false
All reference types	null

Operators

Associative	**Operators**
R to L	`++ -- + - ~ !` (data type)
L to R	`* / %`
L to R	`+ -`
L to R	`<< >> >>>`
L to R	`< > <= >= instanceof`
L to R	`== !=`
L to R	`&`
L to R	`^`
L to R	`|`
L to R	`&&`
L to R	`||`
R to L	`?:`
R to L	`= *= /= %= += -= <<=` `>>= >>>= &= ^= |=`

Logical Operators

- The `boolean` operators are:

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

- The short-circuit `boolean` operators are:

```
&& – AND            || – OR
```

- You can use these operators as follows:

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

Bitwise Logical Operators

- The integer *bitwise* operators are:

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

- Byte-sized examples:

```
                              0 0 1 0 1 1 0 1
~ 0 1 0 0 1 1 1 1           & 0 1 0 0 1 1 1 1
-----------------           -----------------
  1 0 1 1 0 0 0 0             0 0 0 0 1 1 0 1


  0 0 1 0 1 1 0 1             0 0 1 0 1 1 0 1
^ 0 1 0 0 1 1 1 1           | 0 1 0 0 1 1 1 1
-----------------           -----------------
  0 1 1 0 0 0 1 0             0 1 1 0 1 1 1 1
```

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.
  - The sign bit is not copied during the shift.

Left-Shift Operator (<<)

- Left-shift works as follows:

$$128 \ll 1 \text{ returns } 128 * 2^{1} = 256$$
$$16 \ll 2 \text{ returns } 16 * 2^{2} = 64$$

Shift Operator Examples

1357 =	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 1 0 1
-1357 =	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 0 0 1 1
1357 >> 5 =	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0
-1357 >> 5 =	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1
1357 >>> 5 =	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0
-1357 >>> 5 =	0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1
1357 << 5 =	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 1 0 1 0 0 0 0 0
-1357 << 5 =	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 0 0 1 1 0 0 0 0 0

String Concatenation With +

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

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

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

Casting

- If information might be lost in an assignment, the programmer must confirm the assignment with a cast.
- The assignment between `long` and `int` requires an explicit cast.

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

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

Promotion and Casting of Expressions

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

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

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

Branching Statements

The `if`, `else` statement syntax:

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

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

Branching Statements

An `if`, `else` statement example:

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

Branching Statements

The `switch` statement syntax:

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

Branching Statements

A `switch` statement example:

```
switch ( carModel ) {
  case DELUXE:
    addAirConditioning();
    addRadio();
    addWheels();
    addEngine();
    break;
  case STANDARD:
    addRadio();
    addWheels();
    addEngine();
    break;
  default:
    addWheels();
    addEngine();
}
```

Branching Statements

A `switch` statement example:

```
switch ( carModel ) {
  case THE_WORKS:
    addGoldPackage();
    add7WayAdjustableSeats();
  case DELUXE:
    addFloorMats();
    addAirConditioning();
  case STANDARD:
    addRadio();
    addDefroster();
  default:
    addWheels();
    addEngine();
}
```

Looping Statements

The `for` loop:

```
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` loop:

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

Special Loop Flow Control

The `continue` statement:

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

Special Loop Flow Control

Using `break` with labels:

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

Special Loop Flow Control

Using `continue` with labels:

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

Exercise: Using Expressions

- Exercise objectives:
  - Implement the concepts presented in this module
- Tasks:
  - Complete the tasks specified by the instructor

Check Your Progress

- Distinguish between instance and local variables
- Describe how to initialize instance variables
- Identify and correct a `Possible reference before assignment` compiler error
- Recognize, describe, and use Java software 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 5

Arrays

Objectives

- Declare and create arrays of primitive, class, or array types
- Explain why elements of an array are initialized
- Explain how to 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.
- An array is an object; it is created with `new`.

Creating Arrays

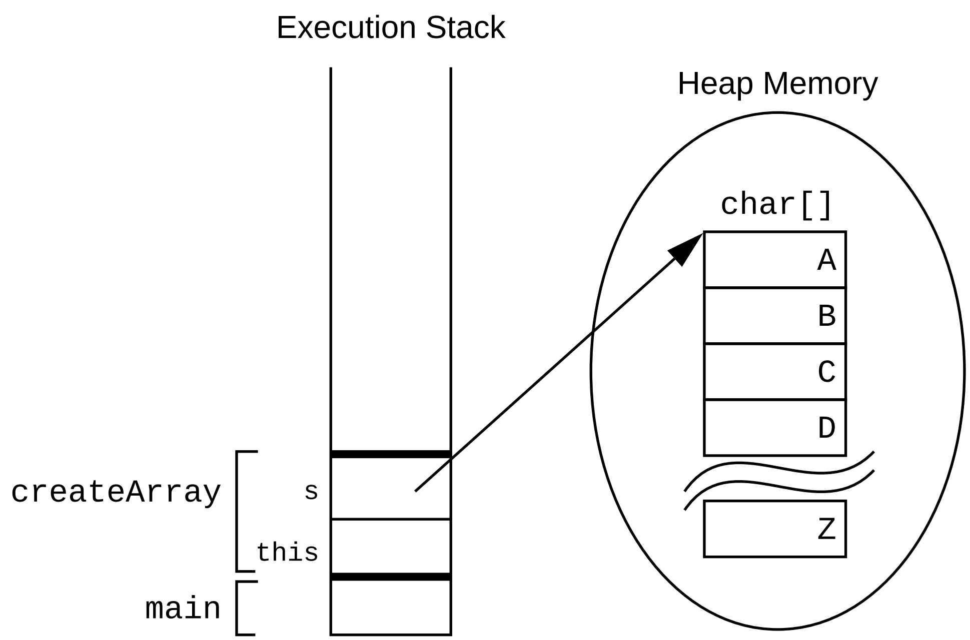
Use the `new` keyword to create an array object.

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

```
public char[] createArray() {
  char[] s;

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

  return s;
}
```



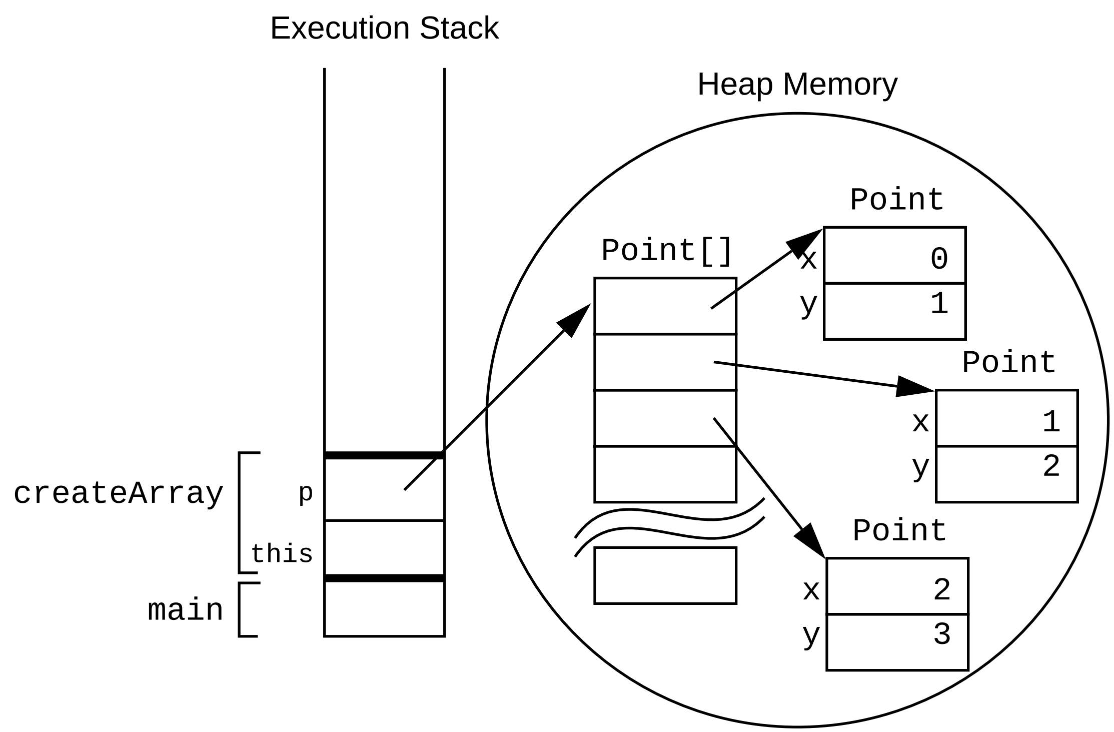
Creating Arrays

Another example, an object array:

```
public Point[] createArray() {
  Point[] p;

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

  return p;
}
```



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";
```

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

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

```
MyDate dates[] = {
    new MyDate(22, 7, 1964),
    new MyDate(1, 1, 2000),
    new MyDate(22, 12, 1964)
};
```

Multidimensional 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
```

Multidimensional Arrays

- Non-rectangular arrays of arrays:

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

- Array of four arrays of five integers each:

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

Array Bounds

All array subscripts begin at 0:

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

Array Resizing

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

```
int myArray[] = new int[6];
myArray = 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:
  - Implement the concepts presented in this module
- Tasks:
  - Complete the tasks specified by the instructor

Check Your Progress

- Declare and create arrays of primitive, class, or array types
- Explain why elements of an array are initialized
- Explain how to 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 a disadvantage of using arrays?

Module 6

Class Design

Objectives

- Define *inheritance*, *polymorphism*, *overloading*, *overriding*, and *virtual method invocation*
- Use the access modifiers `protected` and “package-friendly”
- Describe the concepts of constructor and method overloading
- Describe the complete object construction and initialization operation

Objectives

- In a Java program, identify the following:
  - Overloaded methods and constructors
  - The use of `this` to call overloaded constructors
  - Overridden methods
  - Invocation of `super` class methods
  - Parent class constructors
  - Invocation of parent class constructors

Relevance

- How does the Java programming language support object inheritance?

Subclassing

The `Employee` class:

Employee
+name : String = "" +salary : double +birthDate : Date
+getDetails() : String

```
public class Employee {
  public String name = "";
  public double salary;
  public Date birthDate;

  public String getDetails() {...}
}
```

Subclassing

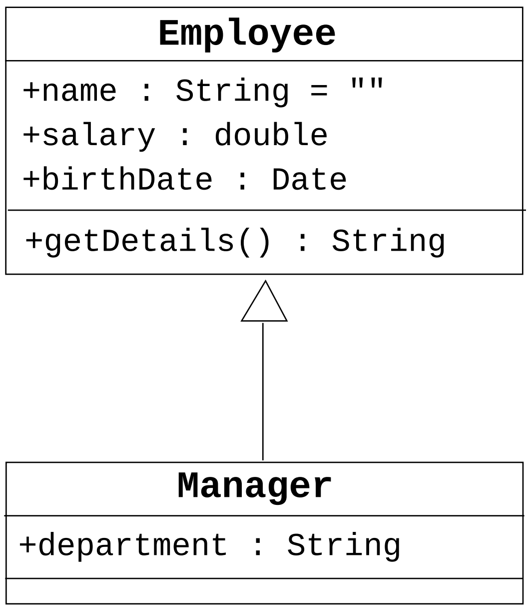
The `Manager` class:

Manager
+name : String = "" +salary : double +birthDate : Date +department : String
+getDetails() : String

```
public class Manager {
  public String name = "";
  public double salary;
  public Date birthDate;
  public String department;

  public String getDetails() {...}
}
```

Subclassing




```
public class Employee {
  public String name = "";
  public double salary;
  public Date birthDate;

  public String getDetails() {...}
}
```

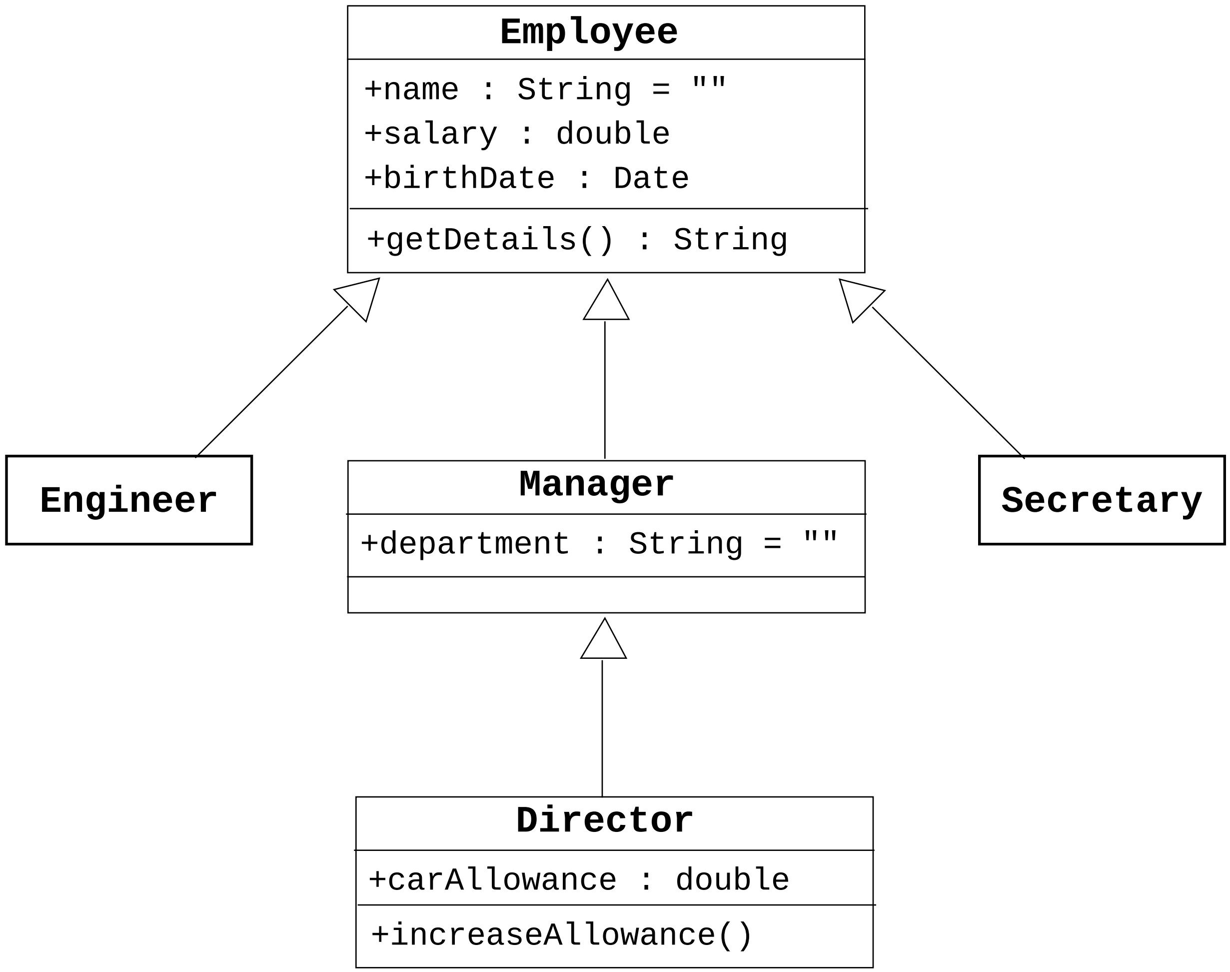
```
public class Manager extends Employee {
  public String department;
}
```

Single Inheritance

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

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

Single Inheritance



Access Control

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

Overriding Methods

- A subclass can modify behavior inherited from a parent class.
- 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 {
  protected String name;
  protected double salary;
  protected Date birthDate;

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


public class Manager extends Employee {
  protected String department;

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

The `super` Keyword

- `super` is used in a class to refer to its superclass.
- `super` is used to refer to the members of superclass, both data attributes and methods.
- 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 double salary;
  private Date birthDate;

  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;
  }
}
```

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 can refer to objects of different forms.

Polymorphism

```
Employee employee = new Manager(); //legal

// Illegal attempt to assign Manager attribute
employee.department = "Sales";
// the variable is declared as an Employee type,
// even though the Manager object has that attribute
```

Virtual Method Invocation

- 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

Rules About Overridden Methods

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

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

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

Heterogeneous Collections

- Collections of objects with the same class type are called *homogenous* collections.

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

- Collections of objects with different class types are called *heterogeneous* collections.

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

Polymorphic Arguments

- 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);
```

The `instanceof` Operator

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

public void doSomething(Employee e) {
  if (e instanceof Manager) {
    // Process a Manager
  } else if (e instanceof Engineer) {
    // Process an Engineer
  } else {
    // Process any other type of 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.

Overloading Method Names

- Use 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.

Overloading Constructors

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

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

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

Overloading Constructors

```
public class Employee {
  private static final double BASE_SALARY = 15000.00;
  private String name;
  private double salary;
  private Date   birthDate;

  public Employee(String name, double salary, Date DoB) {
    this.name = name;
    this.salary = salary;
    this.birthDate = DoB;
  }
  public Employee(String name, double salary) {
    this(name, salary, null);
  }
  public Employee(String name, Date DoB) {
    this(name, BASE_SALARY, DoB);
  }
  public Employee(String name) {
    this(name, BASE_SALARY);
  }
  // more Employee code...
}
```

Constructors Are Not Inherited

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

Invoking Parent Class Constructors

- To invoke a parent constructor, you must place a call to `super` in the first line of the constructor
- You can call a specific parent constructor by the arguments that you use in the call to `super`
- If no `this` or `super` call is used in a constructor, then the compiler adds an implicit call to `super()` that calls the parent no argument constructor (which could be the "default" constructor)
  - If the parent class defines constructors, but does not provide a no argument constructor, then a compiler error message is issued

Invoking Parent Class Constructors

```
public class Employee {
  private static final double BASE_SALARY = 15000.00;
  private String name;
  private double salary;
  private Date   birthDate;

  public Employee(String name, double salary, Date DoB) {
    this.name = name;
    this.salary = salary;
    this.birthDate = DoB;
  }
  public Employee(String name, double salary) {
    this(name, salary, null);
  }
  public Employee(String name, Date DoB) {
    this(name, BASE_SALARY, DoB);
  }
  public Employee(String name) {
    this(name, BASE_SALARY);
  }
  // more Employee code...
}
```

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

  public Manager(String name, double salary, String dept) {
    super(name, salary);
    department = dept;
  }
  public Manager(String n, String dept) {
    super(name);
    department = dept;
  }
  public Manager(String dept) { // This code fails: no super()
    department = dept;
  }
}
```

Constructing and Initializing Objects: A Slight Reprise

- Memory is allocated and default initialization occurs
- Instance variable initialization uses these steps recursively:

1. Bind constructor parameters.
2. If explicit `this()`, call recursively, and then skip to Step 5.
3. Call recursively the implicit or explicit `super` call, except for `Object`.
4. Execute the explicit instance variable initializers.
5. Execute the body of the current constructor.

Constructor and Initialization Example

```
public class Object {
  ...
  public Object() {}
  ...
}

public class Employee extends Object {
  private String name;
  private double salary = 15000.00;
  private Date   birthDate;

  public Employee(String n, Date DoB) {
    // implicit super();
    name = n;
    birthDate = DoB;
  }
  public Employee(String n) {
    this(n, null);
  }
}

public class Manager extends Employee {
  private String department;

  public Manager(String n, String d) {
    super(n);
    department = d;
  }
}
```

Constructor and Initialization Example

- `0` Basic initialization
  - `0.1` Allocate memory for the complete `Manager` object
  - `0.2` Initialize all instance variables to their default values (`0` or `null`)
- `1` Call constructor: `Manager("Joe Smith", "Sales")`
  - `1.1` Bind constructor parameters: `n="Joe Smith", d="Sales"`
  - `1.2` No explicit `this()` call
  - `1.3` Call `super(n)` for `Employee(String)`
    - `1.3.1` Bind constructor parameters: `n="Joe Smith"`
    - `1.3.2` Call `this(n, null)` for `Employee(String, Date)`
      - `1.3.2.1` Bind constructor parameters: `n="Joe Smith", DoB=null`
      - `1.3.2.2` No explicit `this()` call
      - `1.3.2.3` Call `super()` for `Object()`
        - `1.3.2.3.1` No binding necessary
        - `1.3.2.3.2` No `this()` call
        - `1.3.2.3.3` No `super()` call (`Object` is the root)
        - `1.3.2.3.4` No explicit variable initialization for `Object`
        - `1.3.2.3.5` No method body to call
      - `1.3.2.4` Initialize explicit Employee variables: `salary=15000.00;`
      - `1.3.2.5` Execute body: `name="Joe Smith"; date=null;`
    - `1.3.3 - 1.3.4` Steps skipped
    - `1.3.5` Execute body: No body in `Employee(String)`
  - `1.4` No explicit initializers for `Manager`
  - `1.5` Execute body: `department="Sales"`

The `Object` Class

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

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

is equivalent to:

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

The `==` Operator Compared With the `equals` Method

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

`equals` Example

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

  public MyDate(int day, int month, int year) {
    this.day   = day;
    this.month = month;
    this.year  = year;
  }

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

  public int hashCode() {
    return (day ^ month ^ year);
  }
}
```

`equals` Example

```
class TestEquals {
  public static void main(String[] args) {
    MyDate  date1 = new MyDate(14, 3, 1976);
    MyDate  date2 = new MyDate(14, 3, 1976);

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

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

    System.out.println("set date2 = date1;");
    date2 = date1;

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

Generates the output:

```
date1 is not identical to date2
date1 is equal to date2
set date2 = date1;
date1 is identical to date2
```

The `toString` Method

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

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();
```

Exercise: Investing Subclasses

- Exercise objectives:
  - Implement the concepts presented in this module
- Tasks:
  - Complete the tasks specified by the instructor

Check Your Progress

- Define *inheritance*, *polymorphism*, *overloading*, *overriding*, and *virtual method invocation*
- Use the access modifiers `protected` and "package-friendly"
- Describe constructor and method overloading
- Describe the complete object construction and initialization operation

Check Your Progress

- In a Java program, identify the following:
  - Overloaded methods and constructors
  - The use of this to all overloaded constructors
  - Overridden methods
  - Invocation of super class methods
  - Parent class constructors
  - Invocation of parent class constructors

Think Beyond

- Now that you understand inheritance and polymorphism, how can you use this information on a current or future project?

Module 7

Advanced Class Features

Objectives

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

Objectives

- In a Java software program, identify:
  - `static` methods and attributes
  - `final` methods and attributes
  - Nested classes
  - `interface` and `abstract` classes
  - `abstract` methods

Relevance

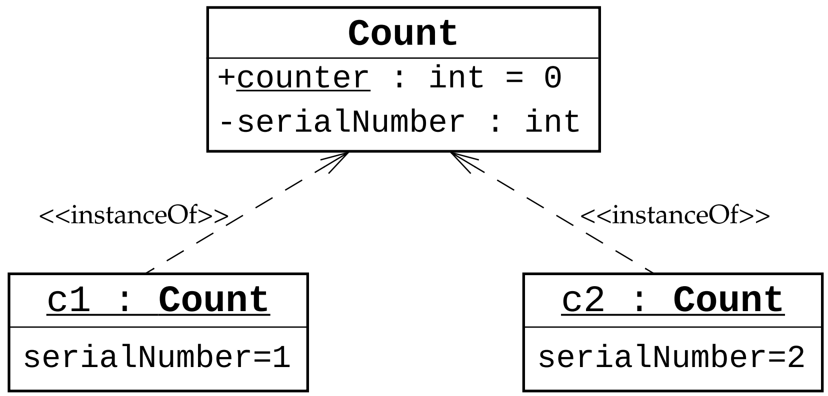
- How can you create a constant?
- How can you create an instance variable that is set once and can not be reset, even internally?
- How can you declare data that is shared by all instances of a given class?
- How can you keep a class or method from being subclassed or overridden?
- How can you create several classes that implement a common interface yet not be part of a common inheritance tree?

The `static` Keyword

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

Class Attributes

- Are shared among all instances of a class




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

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

Class Attributes

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

```
public class OtherClass {
  public void incrementNumber() {
    Count1.counter++;
  }
}
```

Class Methods

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

```
public class Count2 {
  private int serialNumber;
  private static int counter = 0;

  public static int getTotalCount() {
    return counter;
  }

  public Count2() {
    counter++;
    serialNumber = counter;
  }

```

```
public class TestCounter {
  public static void main(String[] args) {
    System.out.println("Number of counter is "
                        + Count.getTotalCount());
    Count count1 = new Count();
    System.out.println(″Number of counter is "
                        + Count.getTotalCount());
  }
}
```

The output of the `TestCounter` program is:

```
Number of counter is 0
Number of counter is 1
```

Static Initializers

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

Static Initializers

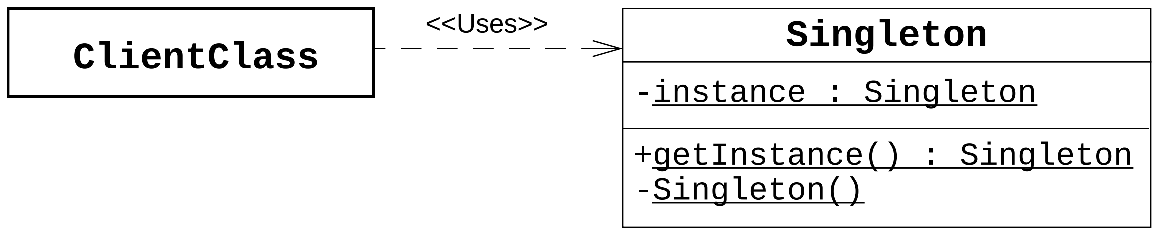
```
public class Count4 {
  public static int counter;
  static {
    counter = Integer.getInteger("myApp.Count4.counter").intValue();
  }
}
```

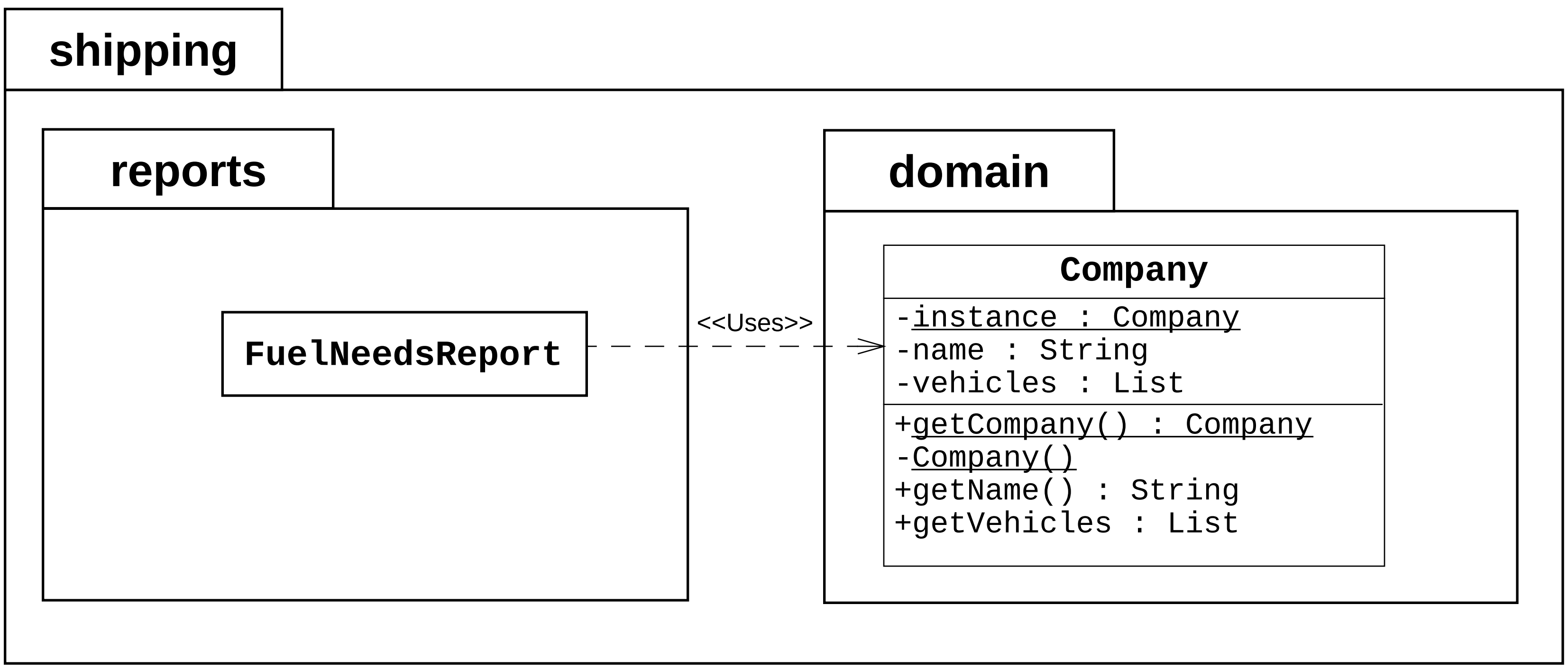
```
public class TestStaticInit {
  public static void main(String[] args) {
    System.out.println("counter = "+ Count4.counter);
  }
}
```

The output of the `TestStaticInit` program is:

```
java -DmyAppCount4.counter=47 TestStaticInit
counter = 47
```

The Singleton Design Pattern





Implementing the Singleton Design Pattern

The Singleton code:

```
package shipping.domain;

public class Company {
  private static Company instance = new Company();
  private String name;
  private Vehicle[] fleet;

  public static Company getCompany() {
    return instance;
  }

  private Company() {...}

  // more Company code ...
}
```

Usage code:

```
package shipping.reports;

import shipping.domain.*;

public class FuelNeedsReport {
  public void generateText(PrintStream output) {
    Company c = Company.getCompany();
    // use Company object to retrieve the fleet vehicles
  }
}
```

The `final` Keyword

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

Final Variables

Constants:

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

Blank Final Instance Attribute:

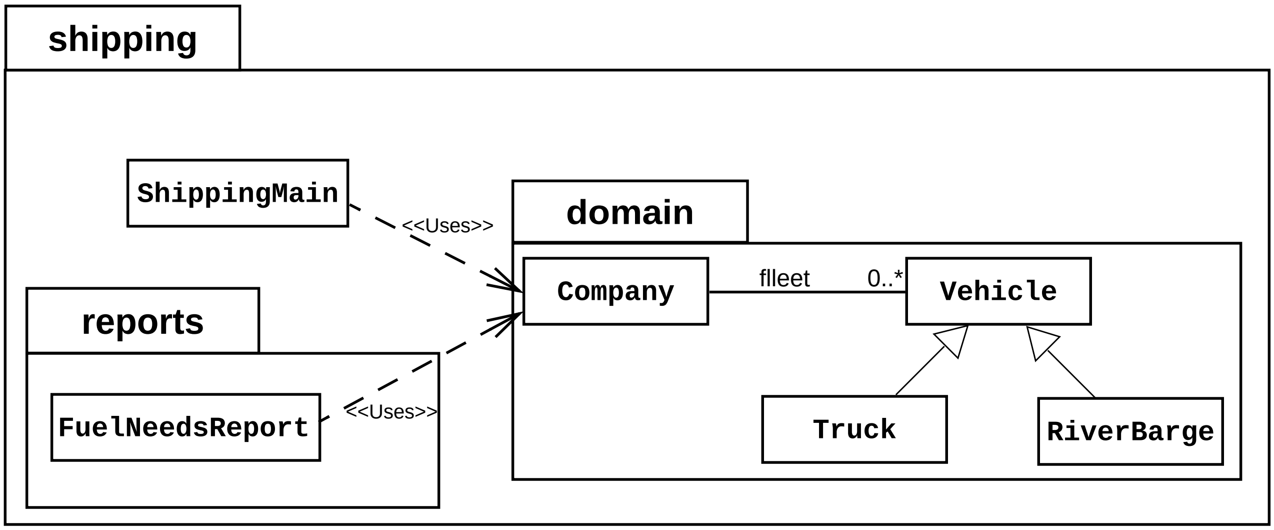
```
public class Customer {
  private final long  customerID;

  public Customer() {
    customerID = createID();
  }
  public long getID() {
    return customerID;
  }
  private long createID() {
    return ... // generate new ID
  }
  ... // more declarations
}
```

Exercise: Working With the `static` and `final` Keywords

- Preparation:
  - You must be familiar with the use of the `static` and `final` keywords.
- Exercise objective:
  - Modify the `Bank` class to implement the Singleton design pattern.
- Tasks:
  - Complete the tasks specified by the instructor.

Abstract Classes: Scenario



Fleet initialization code:

```
public class ShippingMain {
  public static void main(String[] args) {
    Company c = Company.getCompany();

    // populate the company with a fleet of vehicles
    c.addVehicle( new Truck(10000.0) );
    c.addVehicle( new Truck(15000.0) );
    c.addVehicle( new RiverBarge(500000.0) );
    c.addVehicle( new Truck(9500.0) );
    c.addVehicle( new RiverBarge(750000.0) );

    FuelNeedsReport report = new FuelNeedsReport();
    report.generateText(System.out);
  }
}
```

Abstract Classes: Scenario

`FuelNeedsReport` code:

```
public class FuelNeedsReport {
  public void generateText(PrintStream output) {
    Company c = Company.getCompany();
    Vehicle v;
    double fuel;
    double total_fuel = 0.0;

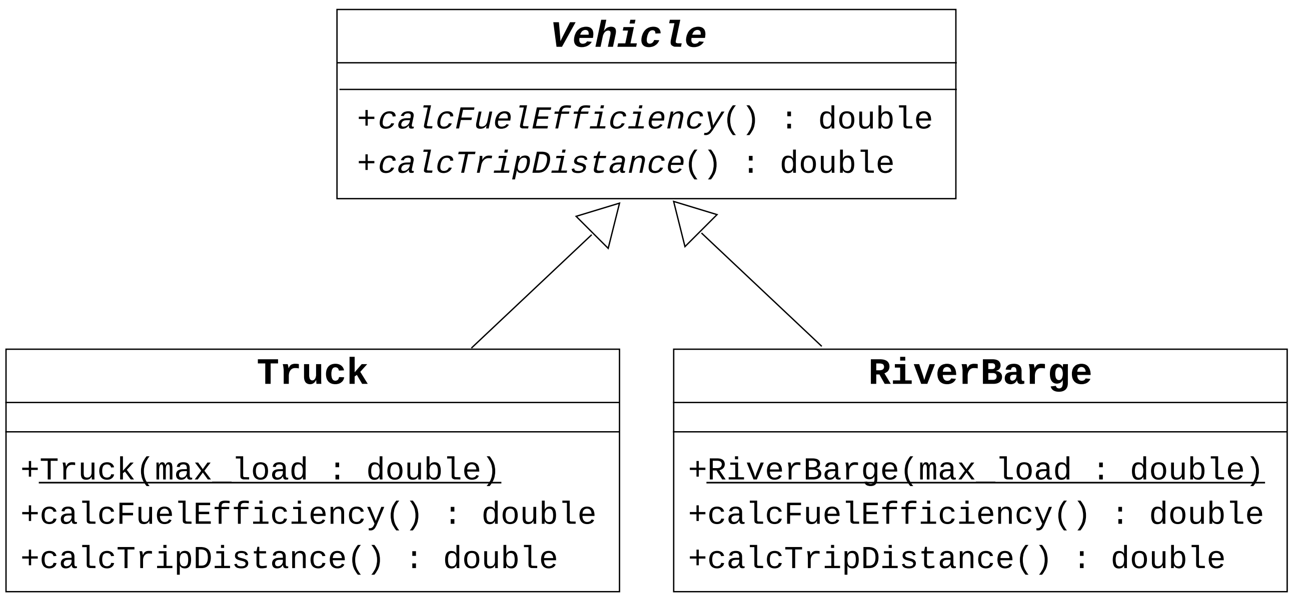
    for ( int i = 0; i < c.getFleetSize(); i++ ) {
      v = c.getVehicle(i);

      // Calculate the fuel needed for this trip
      fuel = v.calcTripDistance() / v.calcFuelEfficency();

      output.println("Vehicle " + v.getName() + " needs "
                     + fuel + " liters of fuel.");
      total_fuel += fuel;
    }
    output.println("Total fuel needs is " + total_fuel + " liters.");
  }
}
```

Abstract Classes: Solution

- An abstract class models a class of objects where the full implementation is not known but is supplied by the concrete subclasses.



Abstract Classes: Solution

```
public abstract class Vehicle {
  public abstract double calcFuelEfficiency();
  public abstract double calcTripDistance();
}
```

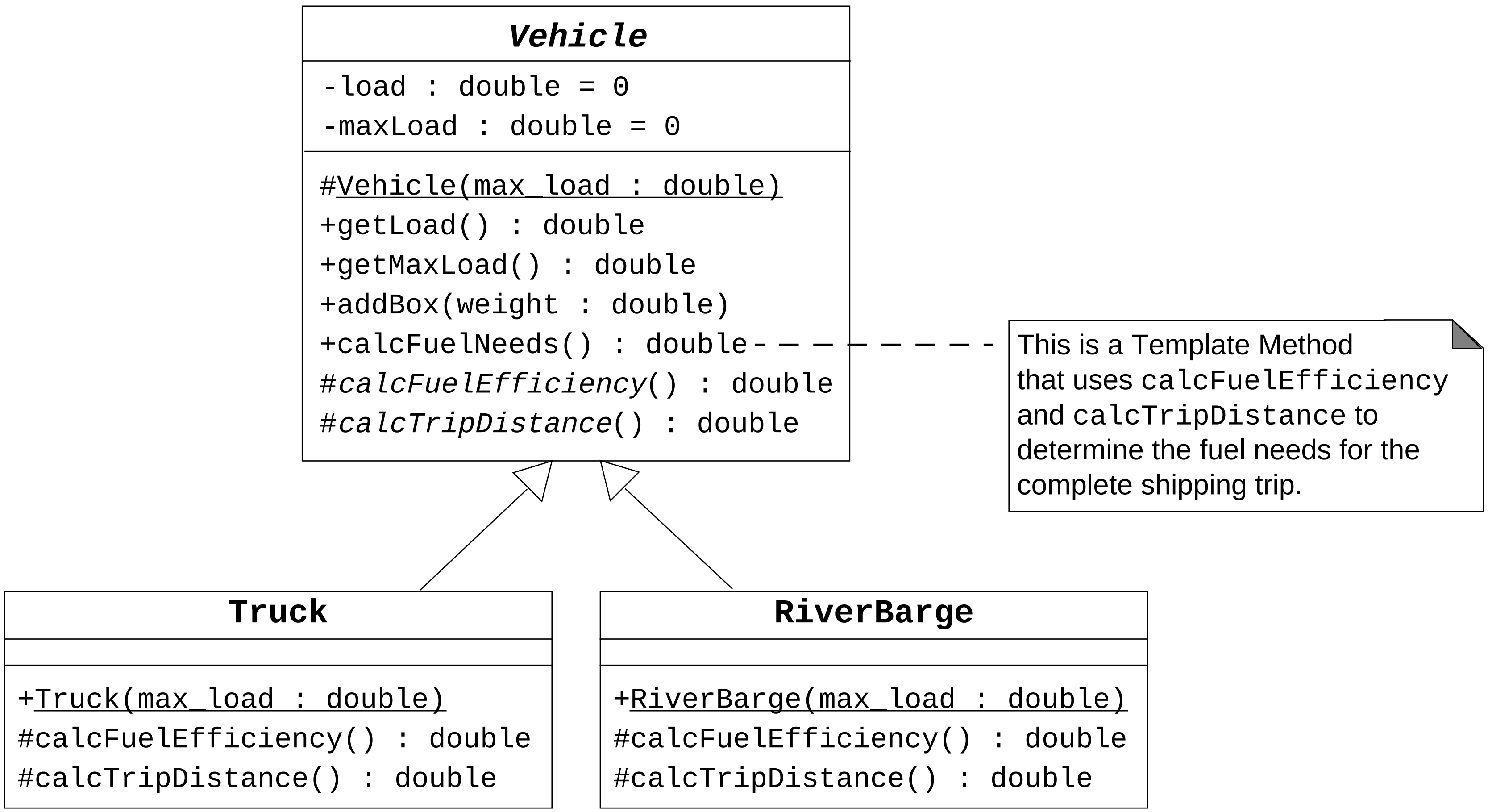
```
public class Truck extends Vehicle {
  public Truck(double max_load) {...}

  public double calcFuelEfficiency() {
    /* calculate the fuel consumption of a truck at a given load */
  }
  public double calcTripDistrance() {
    /* calculate the distance of this trip on highway */
  }
}
```

```
public class RiverBarge extends Vehicle {
  public RiverBarge(double max_load) {...}

  public double calcFuelEfficiency() {
    /* calculate the fuel efficiency of a river barge */
  }
  public double calcTripDistrance() {
    /* calculate the distance of this trip along the river-ways */
  }
}
```

Template Method Design Pattern

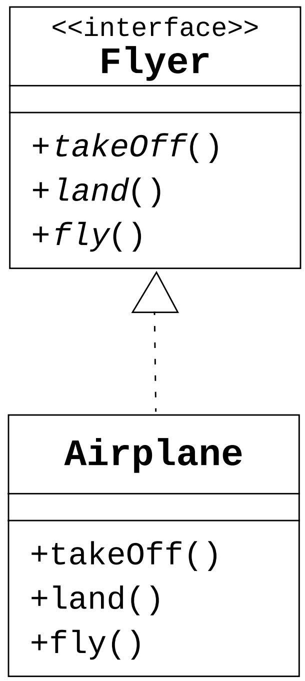


Interfaces

- A "public interface" is a contract between client code and the class that implements that interface.
- A Java *interface* is a formal declaration of such a contract in which all methods contain no implementation.
- Many unrelated classes can implement the same interface.
- A class can implement many unrelated interfaces.
- Syntax of a Java class:

```
<class_declaration> ::=
  <modifier> class <name> [extends <superclass>]
        [implements <interface> [,<interface>]* ] {
    <declarations>*
  }
```

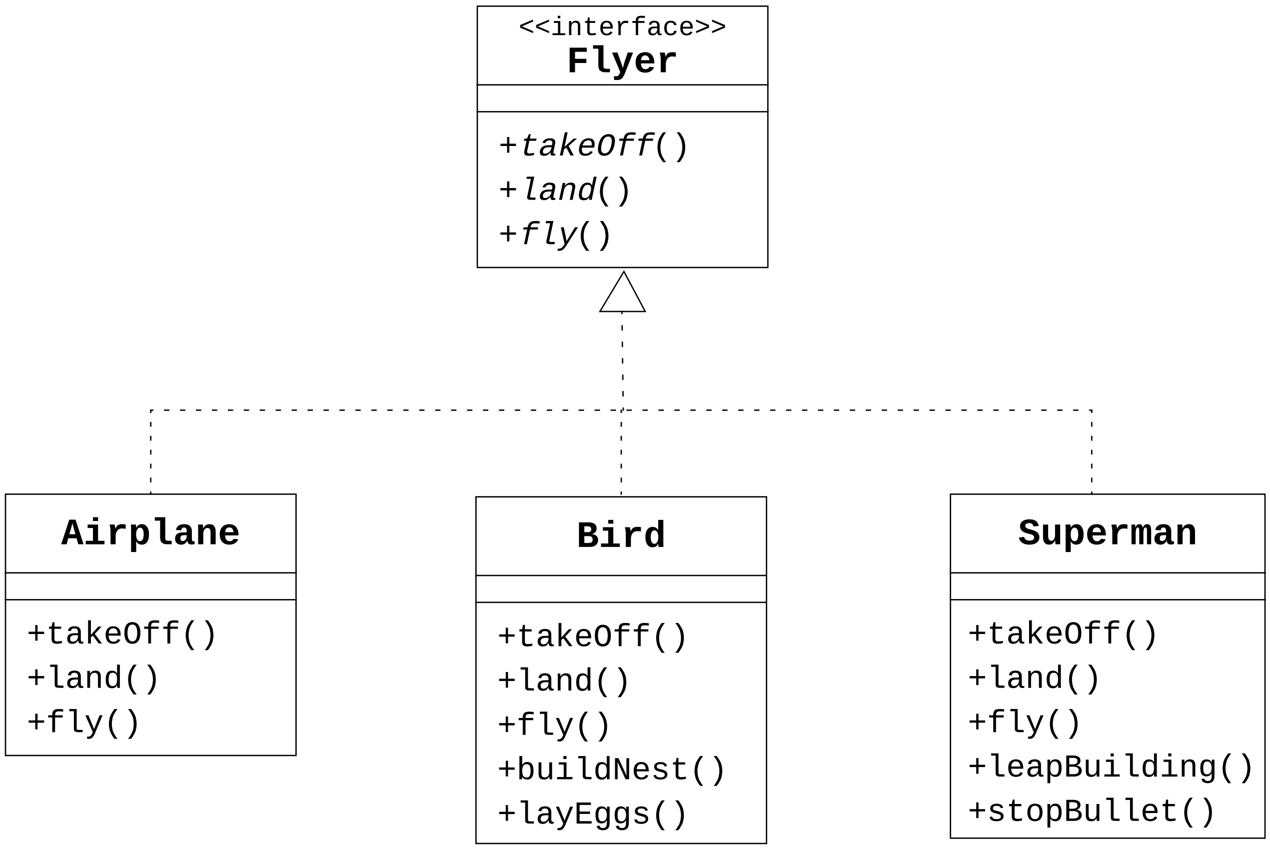
Interface Example



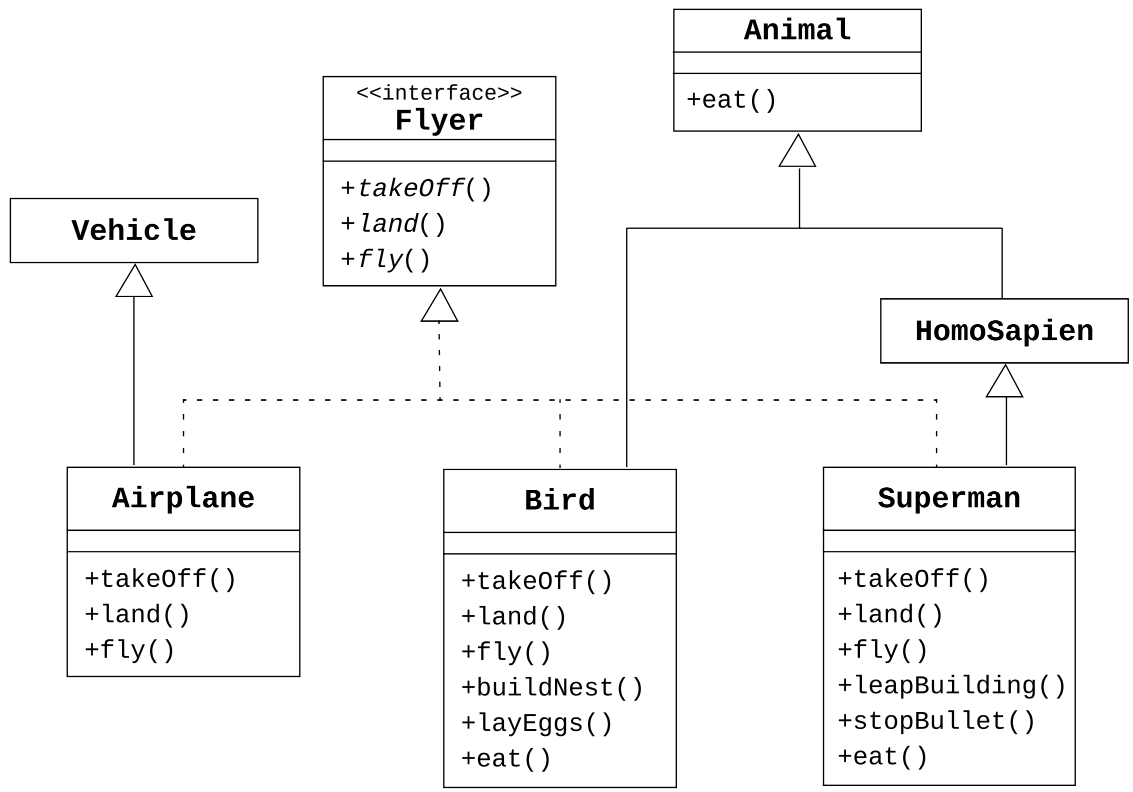

```
public interface Flyer {
  public void takeOff();
  public void land();
  public void fly();
}

public class Airplane implements Flyer {
  public void takeOff() {
    // accelerate until lift-off
    // raise landing gear
  }
  public void land() {
    // lower landing gear
    // deccelerate and lower flaps until touch-down
    // apply breaks
  }
  public void fly() {
    // keep those engines running
  }
}
```

Interface Example



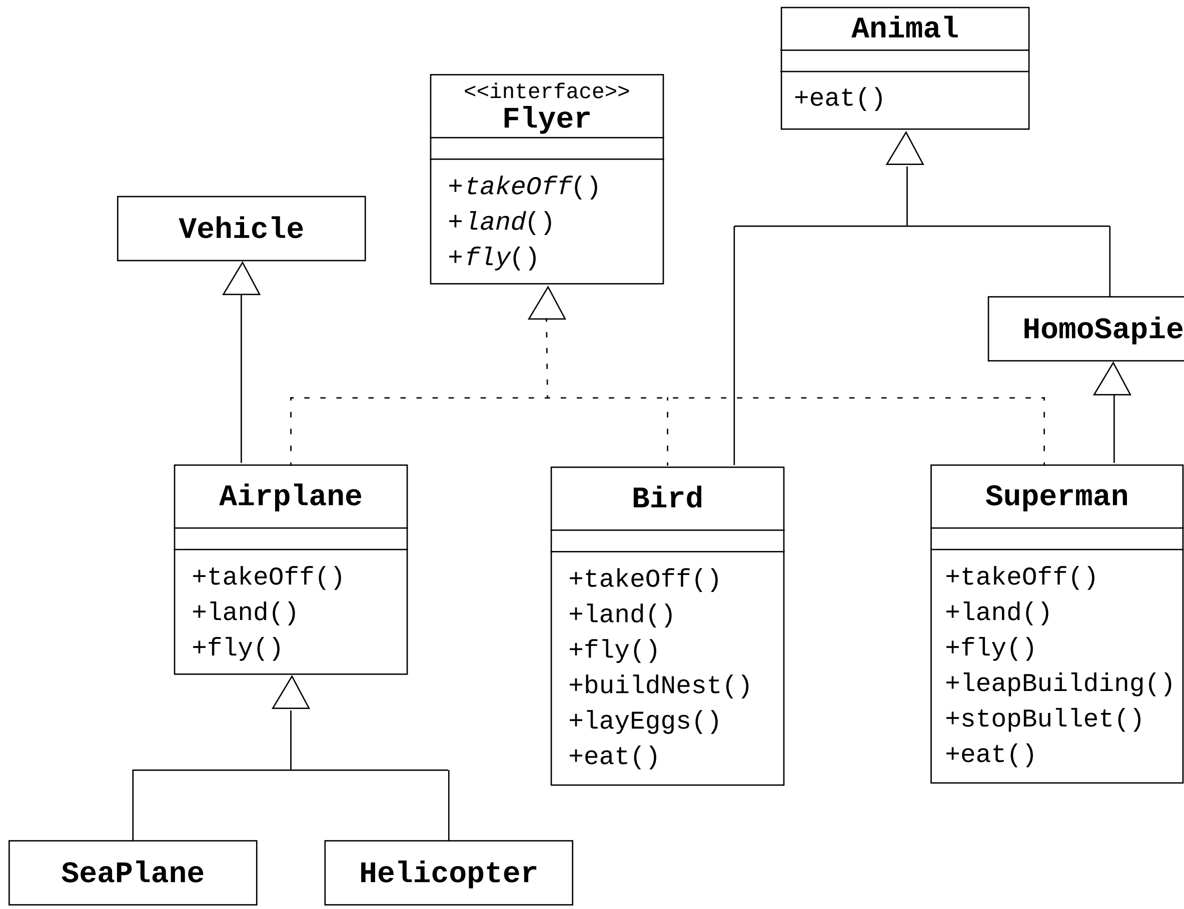
Interface Example



Interface Example

```
public class Bird extends Animal implements Flyer {
  public void takeOff()   { /* take-off implementation  */ }
  public void land()      { /* landing implementation   */ }
  public void fly()       { /* fly implementation       */ }
  public void buildNest() { /* nest building behavior   */ }
  public void layEggs()   { /* egg laying behavior      */ }
  public void eat()       { /* override eating behavior */ }
}
```

Interface Example



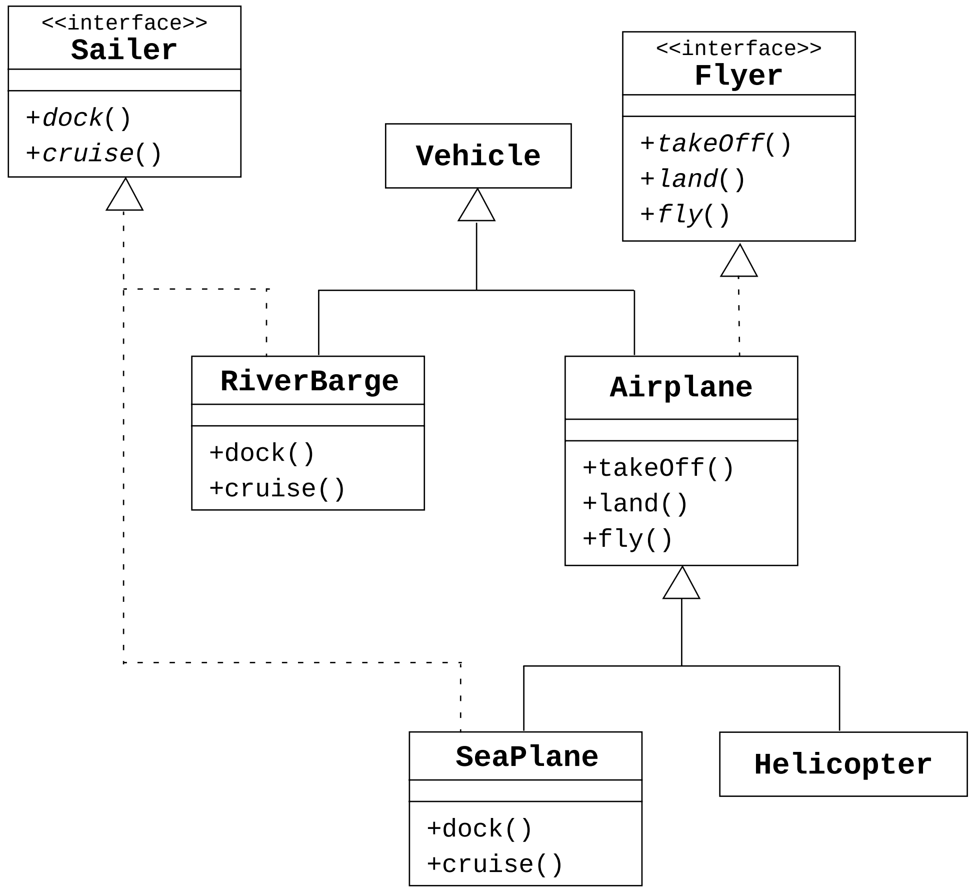
Interface Example

```
public class Airport {
  public static void main(String[] args) {
    Airport metropolisAirport = new Airport();
    Helicopter copter = new Helicopter();
    SeaPlane sPlane = new SeaPlane();
    Flyer S = Superman.getSuperman(); // Superman is a Singleton

    metropolisAirport.givePermissionToLand(copter);
    metropolisAirport.givePermissionToLand(sPlane);
    metropolisAirport.givePermissionToLand(S);
  }

  private void givePermissionToLand(Flyer f) {
    f.land();
  }
}
```

Multiple Interface Example



Multiple Interface Example

```
public class Harbor {
  public static void main(String[] args) {
    Harbor bostonHarbor = new Harbor();
    RiverBarge barge = new RiverBarge();
    SeaPlane sPlane = new SeaPlane();

    bostonHarbor.givePermissionToDock(barge);
    bostonHarbor.givePermissionToDock(sPlane);
  }

  private void givePermissionToDock(Sailer s) {
    s.dock();
  }
}
```

Uses of Interfaces

- 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
- Simulating multiple inheritance by declaring a class that implements several interfaces

Nested Classes

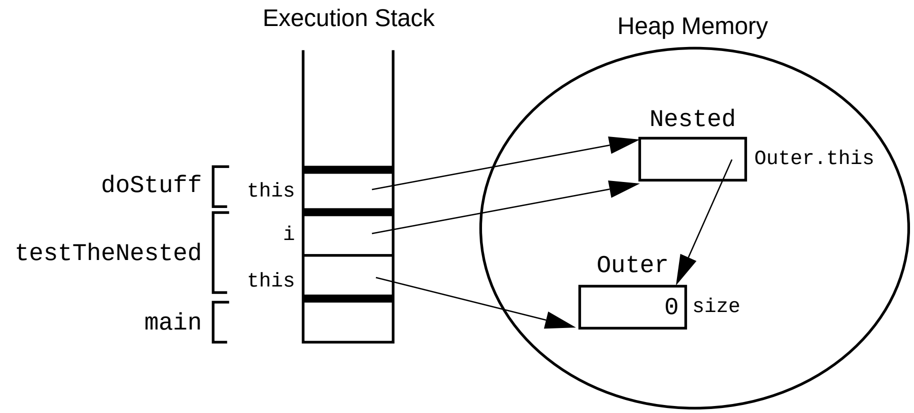
- Added to Java™ Development Kit (JDK™) 1.1
- Allow a class definition to be placed inside another class definition
- Group classes that logically belong together
- Have access to their enclosing class's scope

Nested Class Example

```
public class Outer1 {
  private int size;

  /* Declare a nested class called "Nested" */
  public class Nested {
    public void doStuff() {
      // The nested class has access to ’size’ from Outer
      size++;
    }
  }

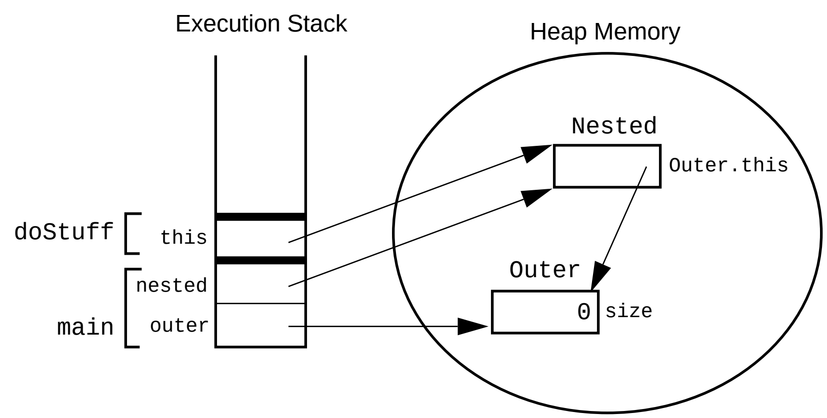
  public void testTheNested() {
    Nested i = new Nested();
    i.doStuff();
  }
}
```



Nested Class Example

```
public class Outer2 {
  private int size;

  public class Nested {
    public void doStuff() {
      size++;
    }
  }
}
```




```
public class TestNested {
  public static void main(String[] args) {
    Outer2 outer = new Outer2();

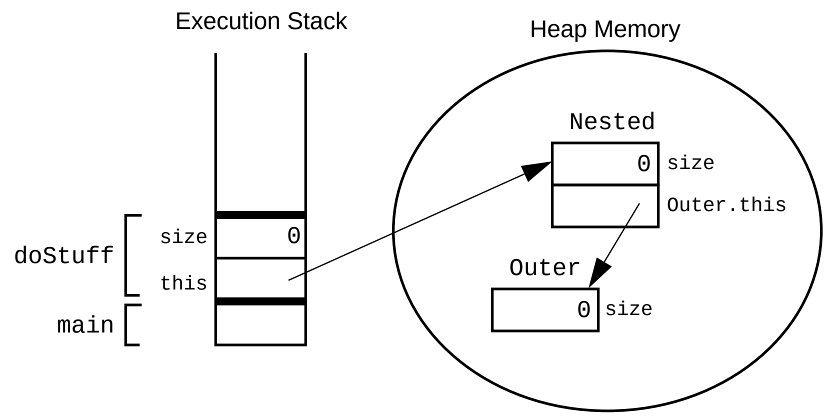
    // Must create a Nested object relative to an Outer
    Outer2.Nested nested = outer.new Nested();
    nested.doStuff();
  }
}
```

Nested Class Example

```
public class Outer3 {
  private int size;

  public class Nested {
    private int size;

    public void doStuff(int size) {
      size++;              // the local parameter
      this.size++;         // the Nested object attribute
      Outer3.this.size++;  // the Outer3 object attribute
    }
  }
}
```



Nested Class Example

```
public class Outer4 {
  private int size = 5;

  public Object makeTheNested(int localVar) {
    final int finalLocalVar = 6;

    // Declare a class within a method!?!
    class Nested {
      public String toString() {
        return ("#<Nested size=" + size +
                // " localVar=" + localVar + // ERROR: ILLEGAL
                "finalLocalVar=" + finalLocalVar + ">");
      }
    }

    return new Nested();
  }

  public static void main(String[] args) {
    Outer4 outer = new Outer4();
    Object obj = outer.makeTheNested(47);
    System.out.println("The object is " + obj);
  }
}
```

Properties of Nested Classes

- Nested class names must be adequately qualified.
- Nested classes defined in a method are called *local*.
- Local classes can access `final` local variables.
- Nested classes can be `abstract`.
- Interfaces can be nested.
- Nested classes can access `static` members of enclosing scopes.
- Non-local classes can have any access protection.
- Nested and enclosing classes are compiled together.

Nested and Inner Classes

- Nested classes can be declared `static`.
- Non-`static` nested classes are called *inner* classes.
- Inner classes can access members of their enclosing instance using the `this` reference.
- Inner classes cannot declare `static` members except compile time constants.

Exercise: Working With Interfaces and Abstract Classes

- Exercise objective:
  - Create a hierarchy of animals that is rooted in an abstract class `Animal`. Several of the animal classes will implement an interface called `Pet`. You will experiment with variations of these animals, their methods, and polymorphism.
- Tasks:
  - Complete the tasks specified by the instructor.

Check Your Progress

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

Check Your Progress

- In a Java software program, identify:
  - `static` methods and attributes
  - `final` methods and attributes
  - Nested classes
  - `interface` and `abstract` classes
  - `abstract` methods

Think Beyond

- What features of the Java programming language do you use to handle runtime error conditions?

Module 8

Exceptions and Assertions

Objectives

- Define exceptions
- Use `try`, `catch`, and `finally` statements
- Describe exception categories
- Identify common exceptions
- Develop programs to handle your own exceptions
- Use assertions
- Distinguish appropriate and inappropriate uses of assertions
- Disable assertions at runtime

Relevance

- In most programming languages, how do you resolve runtime errors?
- If you make assumptions about the way your code works, and those assumptions are wrong, what might happen?
- Is it always necessary or desirable to expend CPU power testing assertions in production programs?

Exceptions and Assertions

- Exceptions handle unexpected situations:
  - File not found, network failure, illegal argument
- Assertions document and test programming assumptions
  - "This can never be negative here"
- Assertion tests can be removed entirely from code at runtime, so the code is not slowed down at all

Exceptions

- Conditions that can readily occur in a correct program are *checked exceptions*
  - Represented by the `Exception` class
- Severe problems that are normally treated as fatal or situations that probably reflect program bugs are *unchecked exceptions*
  - Fatal situations are represented by the `Error` class
  - Probable bugs are represented by the `RuntimeException` class
- The API lists checked exceptions from a method

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++;
    }
  }
}
```

The `try` and `catch` Statements

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

The `finally` Statement

```
try {
  startFaucet();
  waterLawn();
}  catch (BrokenPipeException e) {
  logProblem(e);
} finally {
  stopFaucet();
}
```

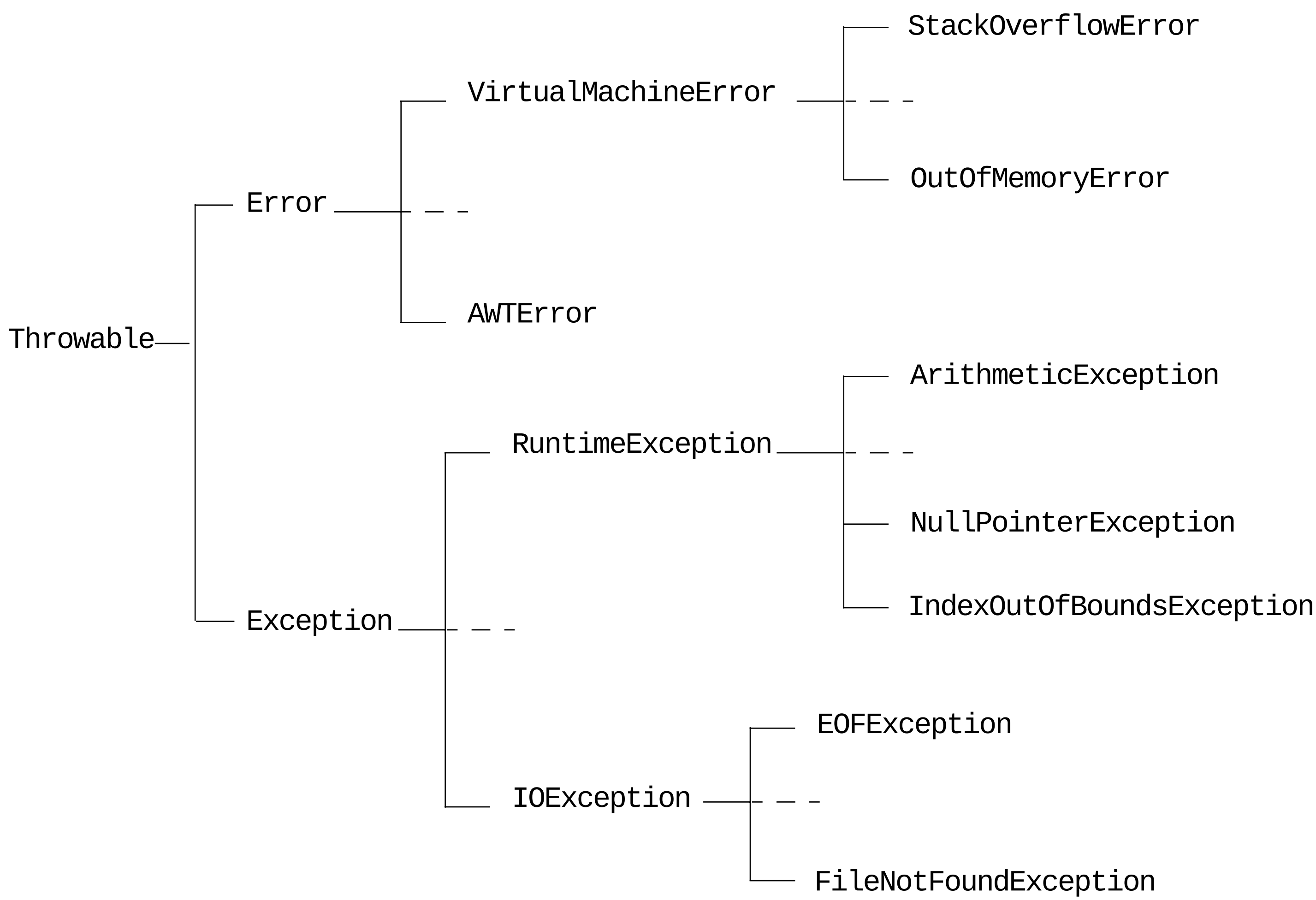
Exception Example Revisited

```
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) {
      try {
        System.out.println(greetings[i]);
        i++;
      } catch (ArrayIndexOutOfBoundsException e){
        System.out.println("Re-setting Index Value");
        i = 0;
      } finally {
        System.out.println("This is always printed");
      }
    }
  }
}
```

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.
- Declare exception or exceptions a method can throw:

```
void trouble() throws IOException {
void trouble() throws IOException, OtherException {
```

- You do not need to handle or declare runtime exceptions or errors.

Method Overriding and Exceptions

The overriding method:

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

Method Overriding Examples

```
public class TestA {
  public void methodA() throws IOException {
    // do some number crunching
  }
}
```

```
public class TestB1 extends TestA {
  public void methodA() throws EOFException {
    // do some number crunching
  }
}
```

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

Creating Your Own Exceptions

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

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

// Use getMessage method 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",
                                      portToConnect);
  }
}
```

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

Assertions

- assert <<boolean_expression> ;
- assert <<boolean_expression>> : <<detail_expression>> ;
- If <<boolean_expression>> evaluates false, then an AssertionError is thrown.
- The second argument is converted to a string and used as descriptive text in the AssertionError message.
- Compile using javac -source 1.4 MyClass.java

Recommended Uses of Assertions

- Documentation and verification of assumptions and internal logic in a single method
  - Internal invariants
  - Control flow invariants
  - Postconditions and class invariants
- Not recommended for precondition checking on public methods

Controlling Runtime Evaluation of Assertions

- If assertion checking is disabled, the code runs as fast as if the check was never there.
- Assertion checks are enabled by default, disable with:
  `java -disableassertions`
  or
  `java -da`
- Assertion checking can be controlled on class, package, and package hierarchy bases, see:
  `docs/guide/lang/assert.html#enable-disable`

Exercise: Handling and Creating Exceptions

- Exercise objectives:
  - Write, compile, and run a program that catches an exception. Write, compile, and run a program that uses a user-defined exception.
- Tasks:
  - Complete the tasks identified by the instructor.

Check Your Progress

- Define exceptions
- Use `try`, `catch`, and `finally` statements
- Describe exception categories
- Identify common exceptions
- Develop programs to handle your own exceptions
- Use assertions
- Distinguish appropriate and inappropriate uses of assertions
- Disable assertions at runtime

Think Beyond

- How many situations can you think of that would require you to create new classes of exceptions?
- Can you think of situations where a constructor would throw an exception?

Module 9

Text-Based Applications

Objectives

- Write a program that uses command-line arguments and system properties
- Write a program that reads from *standard input*
- Write a program that can create, read, and write files
- Describe the basic hierarchy of collections in the Java™ 2 Software Development Kit (Java™ 2 SDK)
- Write a program that uses sets and lists
- Write a program to iterate over a collection

Objectives

- Describe the collection classes that existed before Java 2 SDK
- Identify deprecated classes and explain how to migrate from JDK 1.0 to JDK 1.1 to Java 2 SDK

Relevance

- It is often the case that certain elements of a program should not be hard-coded, such as file names or the name of a database. How can a program be coded to supply these elements at runtime?
- Simple arrays are far too static for most collections (that is, a fixed number of elements). What Java technology features exist to support more flexible collections?
- Besides computation, what are key elements of any text-based application?

Command-Line Arguments

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

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

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

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

Command-Line Arguments

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

`java TestArgs arg1 arg2 "another arg"`

Here is an excerpt of the output:

```
args[0] is ’arg1’
args[1] is ’arg2’
args[2] is ’another arg’
```

System Properties

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

The `Properties` Class

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

System Properties

```
import java.util.Properties;
import java.util.Enumeration;

public class TestProperties {
  public static void main(String[] args) {
    Properties props = System.getProperties();
    Enumeration prop_names = props.propertyNames();

    while ( prop_names.hasMoreElements() ) {
      String prop_name = (String) prop_names.nextElement();
      String property = props.getProperty(prop_name);
      System.out.println("property ’" + prop_name
         + "’ is ’" + property + "’");
    }
  }
}
```

`java -DmyProp=theValue TestProperties`

Here is an excerpt of the output:

```
property ’java.vm.version’ is ’1.4.0’
property ’java.compiler’ is ’NONE’
property ’path.separator’ is ’:’
property ’file.separator’ is ’/’
property ’user.home’ is ’/home/basham’
property ’java.specification.vendor’ is ’Sun Microsystems Inc.’
property ’user.language’ is ’en’
property ’user.name’ is ’basham’
property ’myProp’ is ’theValue’
```

Console I/O

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

Writing to Standard Output

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

Reading From Standard Input

```
import java.io.*;

public class KeyboardInput {
  public static void main (String args[]) {
    String s;
    // Create a buffered reader to read
    // each line from the keyboard.
    InputStreamReader ir = new InputStreamReader(System.in);
    BufferedReader in = new BufferedReader(ir);

    System.out.println("Unix: Type ctrl-d or ctrl-c to exit." +
        "\nWindows: Type ctrl-z to exit");
    try {
      // Read each input line and echo it to the screen.
      s = in.readLine();
      while ( s != null ) {
        System.out.println("Read: " + s);
        s = in.readLine();
      }

      // Close the buffered reader.
      in.close();
    } catch (IOException e) { // Catch any IO exceptions.
      e.printStackTrace();
    }
  }
}
```

Files and File I/O

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

Creating a New `File` Object

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

`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()
```

File Stream I/O

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

File Input Example

```
import java.io.*;
public class ReadFile {
  public static void main (String[] args) {
    // Create file
    File file = new File(args[0]);

    try {
      // Create a buffered reader to read each line from a file.
      BufferedReader in = new BufferedReader(new FileReader(file));
      String s;

      // Read each line from the file and echo it to the screen.
      s = in.readLine();
      while ( s != null ) {
        System.out.println("Read: " + s);
        s = in.readLine();
      }
      // Close the buffered reader, which also closes the file reader.
      in.close();

    } catch (FileNotFoundException e1) {
    // If this file does not exist
      System.err.println("File not found: " + file);

    } catch (IOException e2) {
    // Catch any other IO exceptions.
      e2.printStackTrace();
    }
  }
}
```

File Output Example

```
import java.io.*;

public class WriteFile {
  public static void main (String[] args) {
    // Create file
    File file = new File(args[0]);

    try {
   // Create a buffered reader to read each line from standard in.
      BufferedReader in
        = new BufferedReader(new InputStreamReader(System.in));
      // Create a print writer on this file.
      PrintWriter out
        = new PrintWriter(new FileWriter(file));
      String s;

      System.out.print("Enter file text.  ");
      System.out.println("[Type ctrl-d (or ctrl-z) to stop.]");

      // Read each input line and echo it to the screen.
      while ((s = in.readLine()) != null) {
        out.println(s);
      }

      // Close the buffered reader and the file print writer.
      in.close();
      out.close();

    } catch (IOException e) {
    // Catch any IO exceptions.
      e.printStackTrace();
    }
  }
}
```

Exercise: Writing User Input to a File

- Exercise objectives:
  - Create a program to read text from standard input and write it to a file with each line prefixed with a line-number count. This file is specified by a command-line argument.
- Tasks:
  - Complete the tasks specified by the instructor

The `Math` Class

The `Math` class contains a group of static math functions:

- Truncation: `ceil`, `floor`, and `round`
- Variations on `max`, `min`, and `abs` (absolute value)
- Trigonometry: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `toDegrees`, and `toRadians`
- Logarithms: `log` and `exp`
- Others: `sqrt`, `pow`, and `random`
- Constants: `PI` and `E`

The `String` Class

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

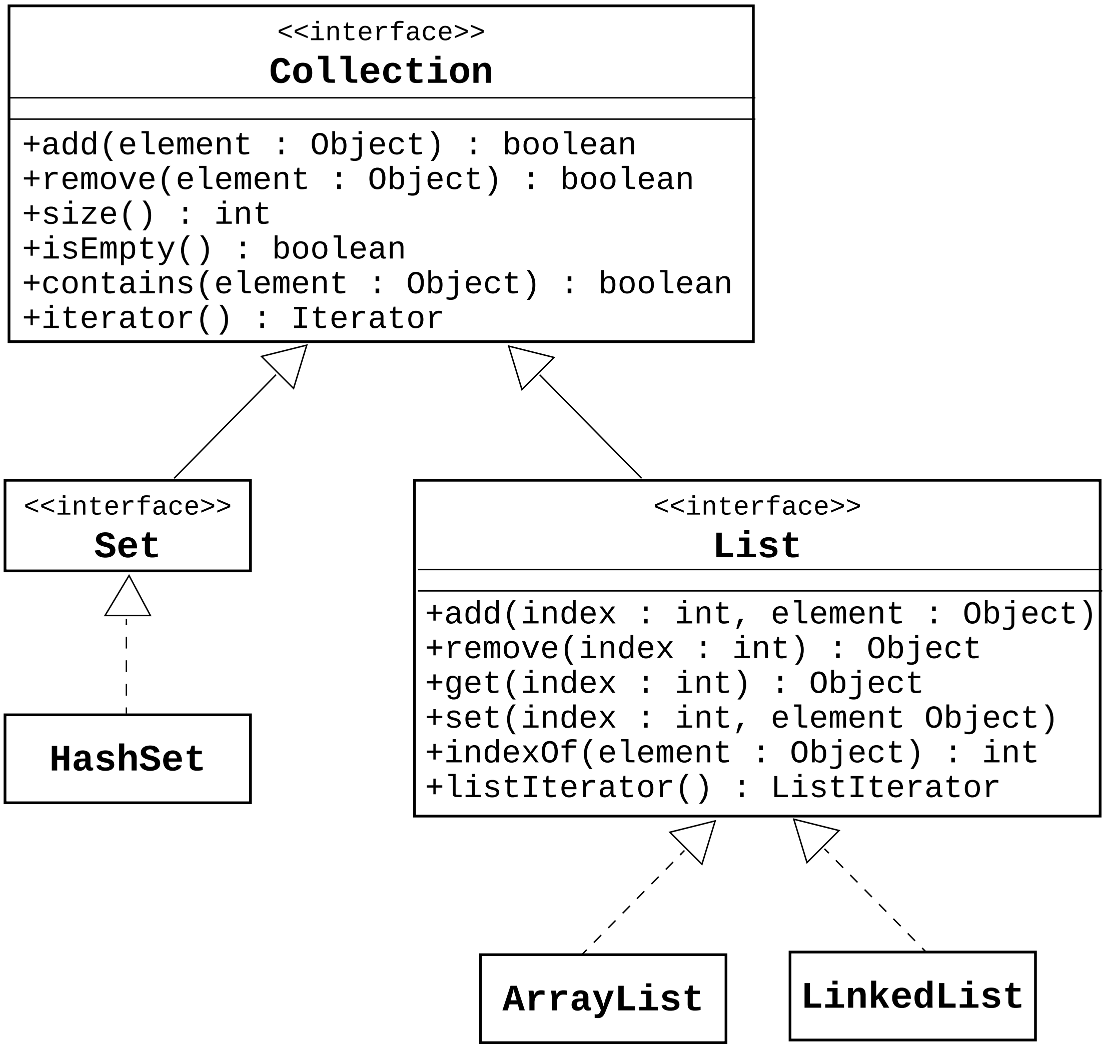
The `StringBuffer` Class

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

The Collections API

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

The Collections API



A Set Example

```
import java.util.*;

public class SetExample {
  public static void main(String[] args) {
    Set set = new HashSet();
    set.add("one");
    set.add("second");
    set.add("3rd");
    set.add(new Integer(4));
    set.add(new Float(5.0F));
    set.add("second");         // duplicate, not added
    set.add(new Integer(4));  // duplicate, not added
    System.out.println(set);
  }
}
```

The output generated from this program is:

```
[one, second, 5.0, 3rd, 4]
```

A `List` Example

```
import java.util.*

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

The output generated from this program is:

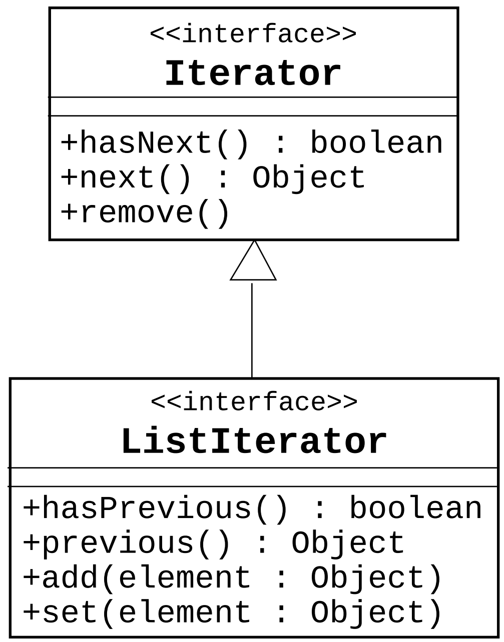
```
[one, second, 3rd, 4, 5.0, second, 4]
```

Iterators

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

```
List list = new ArrayList();
// add some elements
Iterator elements = list.iterator();
while ( elements.hasNext() ) {
  System.out.println(elements.next());
}
```

The Iterator Interface Hierarchy



Collections in JDK 1.1

- `Vector` implements the `List` interface.
- `Stack` is a subclass of `Vector` and supports the `push`, `pop`, and `peek` methods.
- `Hashtable` implements the `Map` interface.
- `Enumeration` is a variation on the `Iterator` interface:
  - An enumeration is returned by the `elements` method in `Vector`, `Stack`, and `Hashtable`
- These classes are thread-safe, and therefore, "heavyweight."

Exercise: Using Collections to Represent Aggregation

- Exercise objectives:
  - Replace the arrays code that you used to implement multiplicity in the relationships between bank and customer, and customer and their accounts.
- Tasks:
  - Complete the tasks specified by the instructor

Deprecation

- Deprecation makes classes, attributes, methods, constructors, and so on, obsolete.
- Obsolete declarations 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 final 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:19: warning: Date(int,int,int) in java.util.Date has been
deprecated
    Date d = new Date (year, month, day);
             ^
DateConverter.java:21: warning: getDay() in java.util.Date has been deprecated
    return (DAY_OF_THE_WEEK[d.getDay()]);
                             ^
2 warnings
```

Deprecation

A Java 2 SDK version rewritten is:

```java
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) {
    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)]);
  }
}
```

Check Your Progress

- Write a program that uses command-line arguments and system properties
- Write a program that reads from *standard input*
- Write a program that can create, read, and write files
- Describe the basic hierarchy of collections in Java 2 SDK
- Write a program that uses sets and lists
- Write a program to iterate over a collection

Check Your Progress

- Describe the collection classes that existed before Java 2 SDK
- Identify deprecated classes and explain how to migrate from JDK 1.0 to JDK 1.1 to Java 2 SDK

Think Beyond

- Many applications are text-based. What other styles of programs exist?
- What features does the Java application environment have that support user interface development?
- How were interfaces used in this module? Could they have been replaced by some other mechanism, such as abstract classes?

Module 10

Building Java GUIs

Objectives

- Describe the Abstract Windowing Toolkit (AWT) package and its components
- Define the terms *containers*, *components*, and *layout managers*, and describe how they work together to build a graphical user interface (GUI)
- Use layout managers
- Use the `FlowLayout`, `BorderLayout`, and `GridLayout` 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 technology program, identify the following:
  - Containers
  - The associated layout managers
  - The layout hierarchy of all components

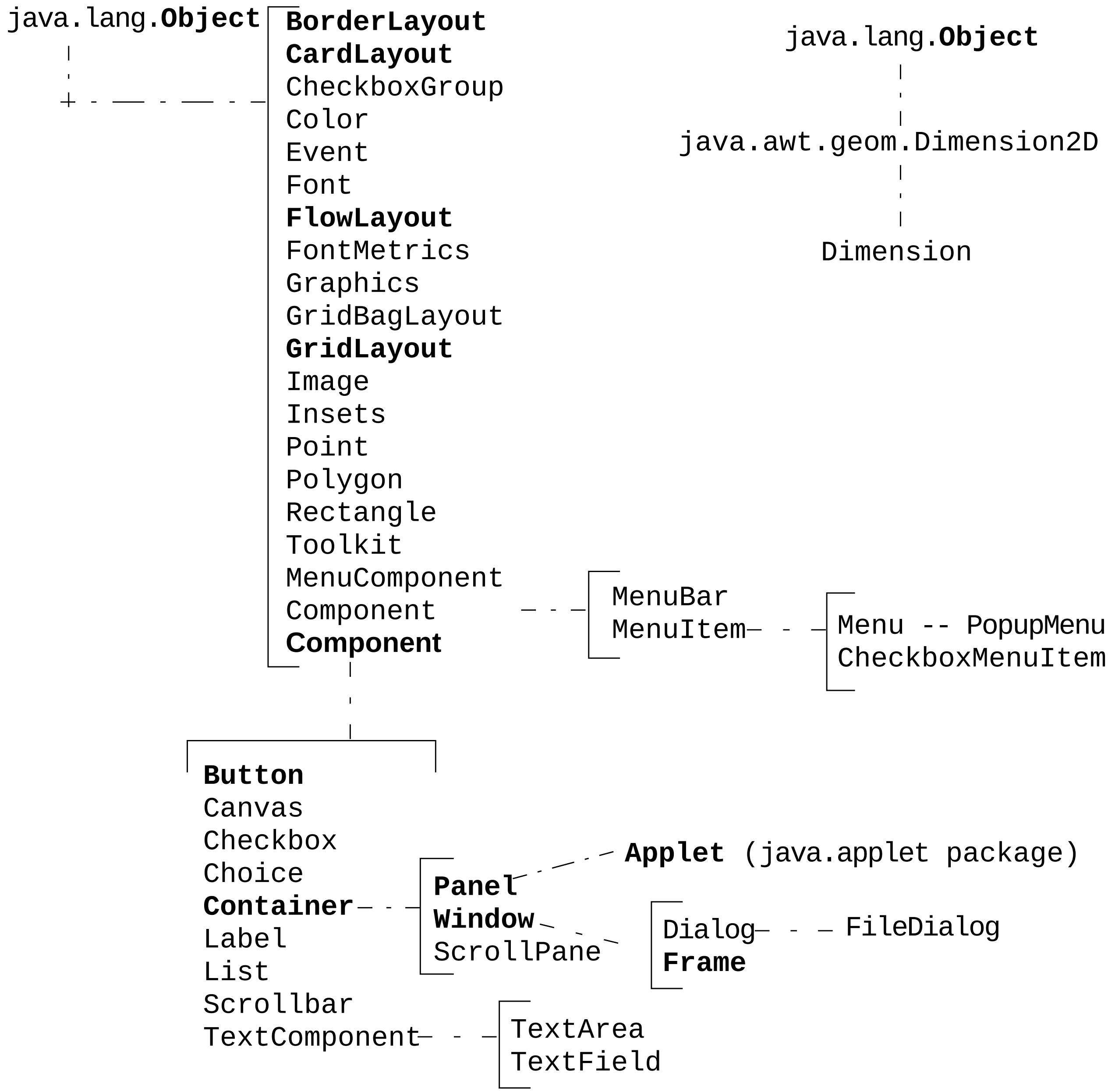
Relevance

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

Abstract Window Toolkit (AWT)

- Provides graphical user interface (GUI) components that are used in all Java applets and applications
- Contains classes that can be composed or extended. Classes can also be abstract
- Ensures that every GUI component that is displayed on the screen is a subclass of the abstract class `Component` or `MenuComponent`
- Has `Container`, which is an abstract subclass of `Component` and includes two subclasses:
  - `Panel`
  - `Window`

The `java.awt` Package



Exceptions – `AWTException` Errors – `AWTError`

Containers

- Add components with the `add()` method.
- The two main types of containers are `Window` and `Panel`.
- A `Window` is a free floating window on the display.
- A `Panel` is a container of GUI components that must exist in the context of some other container, such as a window or applet.

Positioning Components

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

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

Frames

- Are a subclass of `Window`
- Have title and resizing corners
- Are initially invisible, use `setVisible(true)` to expose the frame
- Have `BorderLayout` as the default layout manager
- Use the `setLayout` method to change the default layout manager

FrameExample.java

```
import java.awt.*;

public class FrameExample {
  private Frame f;

  public FrameExample() {
    f = new Frame("Hello Out There!");
  }

  public void launchFrame() {
    f.setSize(170,170);
    f.setBackground(Color.blue);
    f.setVisible(true);
  }

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

FrameExample.java

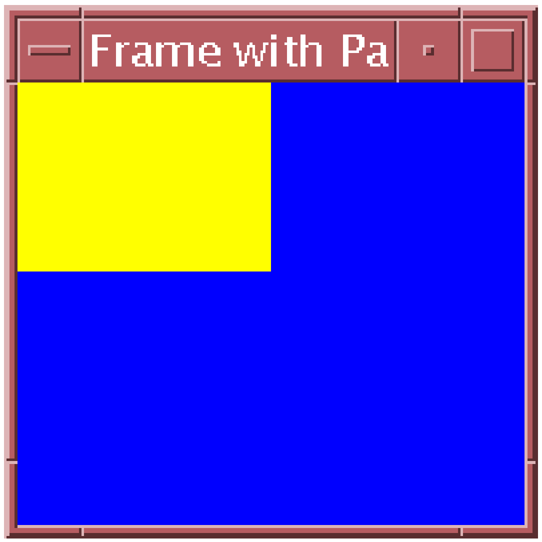
Panels

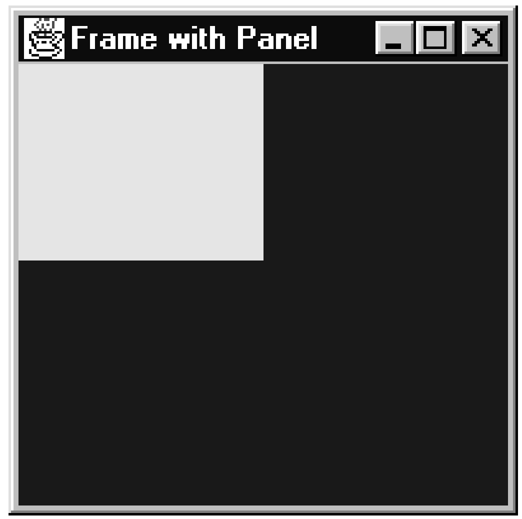
- Provide a space for components
- Allow subpanels to have their own layout manager

FrameWithPanel.java

```
import java.awt.*;

public class FrameWithPanel {
  private Frame f;
  private Panel pan;

  public FrameWithPanel(String title) {
    f = new Frame(title);
    pan = new Panel();
}

  public void launchFrame() {
    f.setSize(200,200);
    f.setBackground(Color.blue);
    f.setLayout(null);  // Override default layout mgr

    pan.setSize(100,100);
    pan.setBackground(Color.yellow);
    f.add(pan);
    f.setVisible(true);
  }

  public static void main(String args[]) {
    FrameWithPanel guiWindow =
        new FrameWithPanel("Frame with Panel");
    guiWindow.launchFrame();
  }
}
```

FrameWithPanel.java

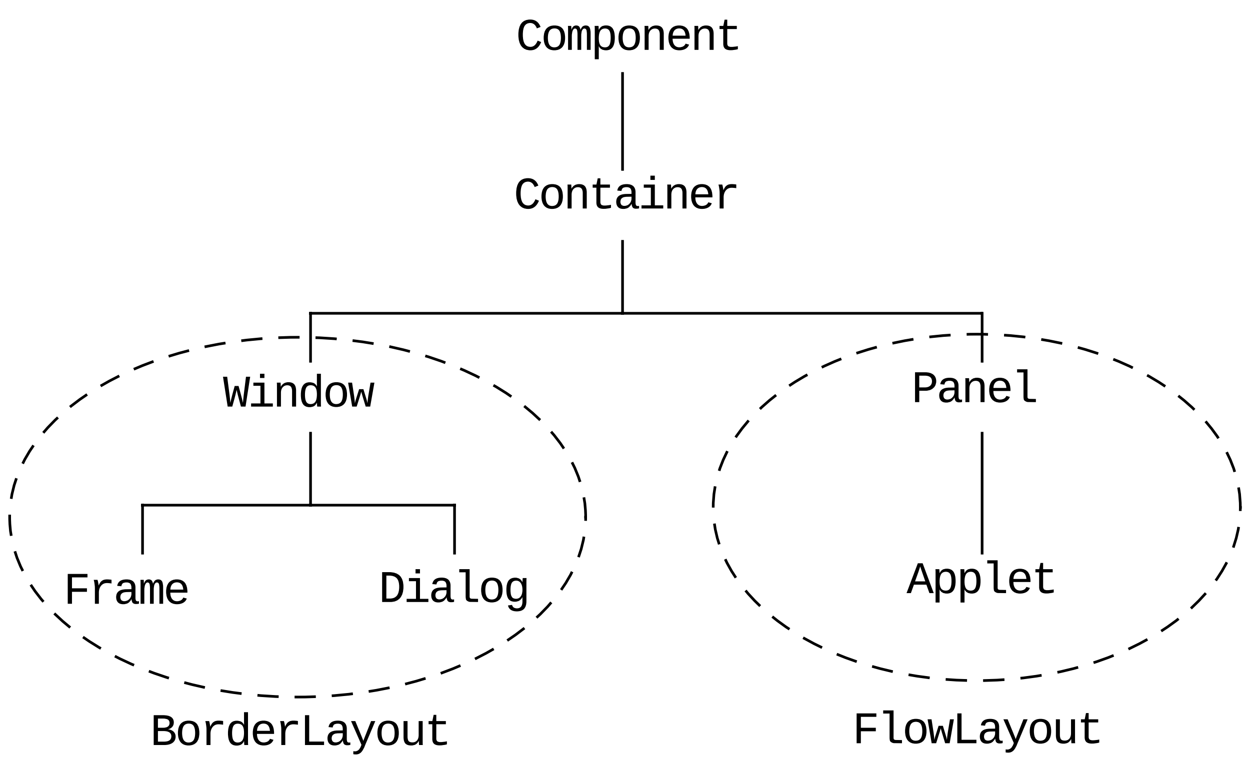




Container Layouts

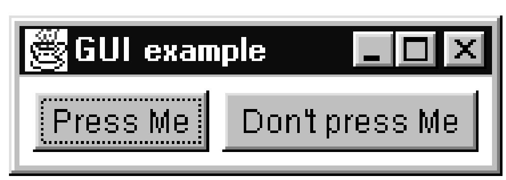
- `FlowLayout`
- `BorderLayout`
- `GridLayout`
- `CardLayout`
- `GridBagLayout`

Default Layout Managers



A Simple FlowLayout Example

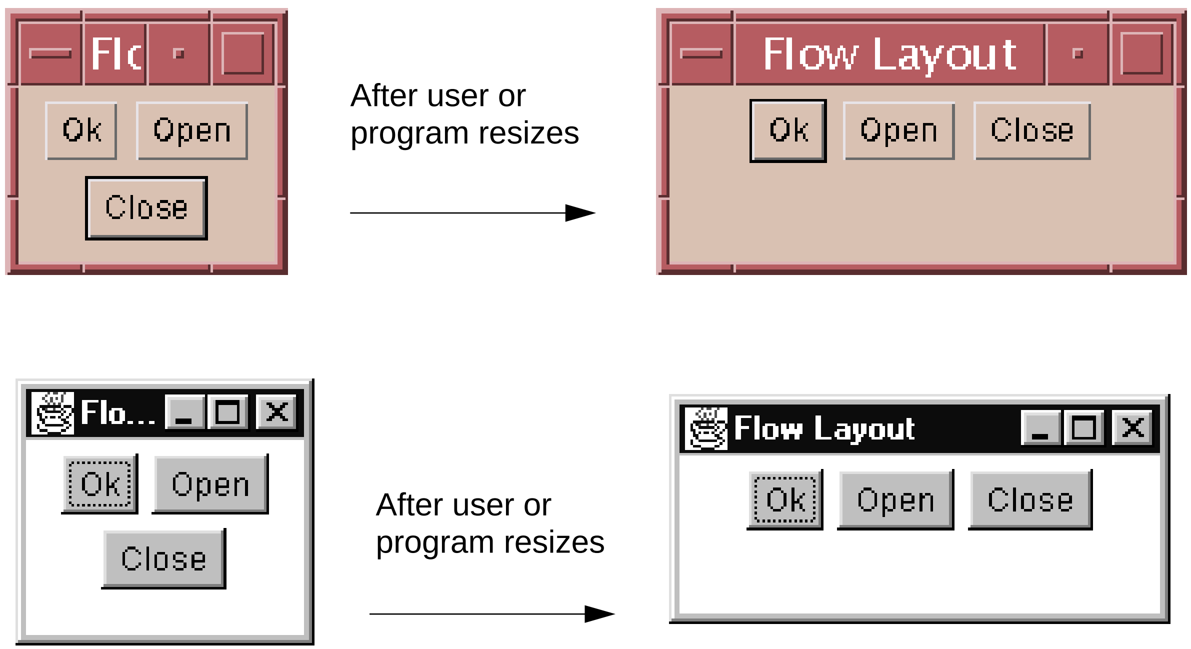
```
import java.awt.*;

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

  public LayoutExample() {
    f = new Frame("GUI example");
    b1 = new Button("Press Me");
    b2 = new Button("Don’t press Me");
  }

  public void launchFrame() {
    f.setLayout(new FlowLayout());
    f.add(b1);
    f.add(b2);
    f.pack();
    f.setVisible(true);
  }

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



The FlowLayout Manager

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

FlowExample.java

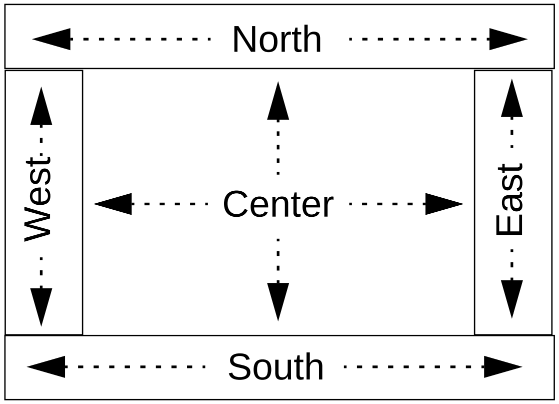


FlowExample.java

```
import java.awt.*;

public class FlowExample {
  private Frame f;
  private Button button1;
  private Button button2;
  private Button button3;

  public FlowExample() {
    f = new Frame("Flow Layout");
    button1 = new Button("Ok");
    button2 = new Button("Open");
    button3 = new Button("Close");
  }

  public void launchFrame() {
    f.setLayout(new FlowLayout());
    f.add(button1);
    f.add(button2);
    f.add(button3);
    f.setSize(100,100);
    f.setVisible(true);
  }

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

The `BorderLayout` Manager

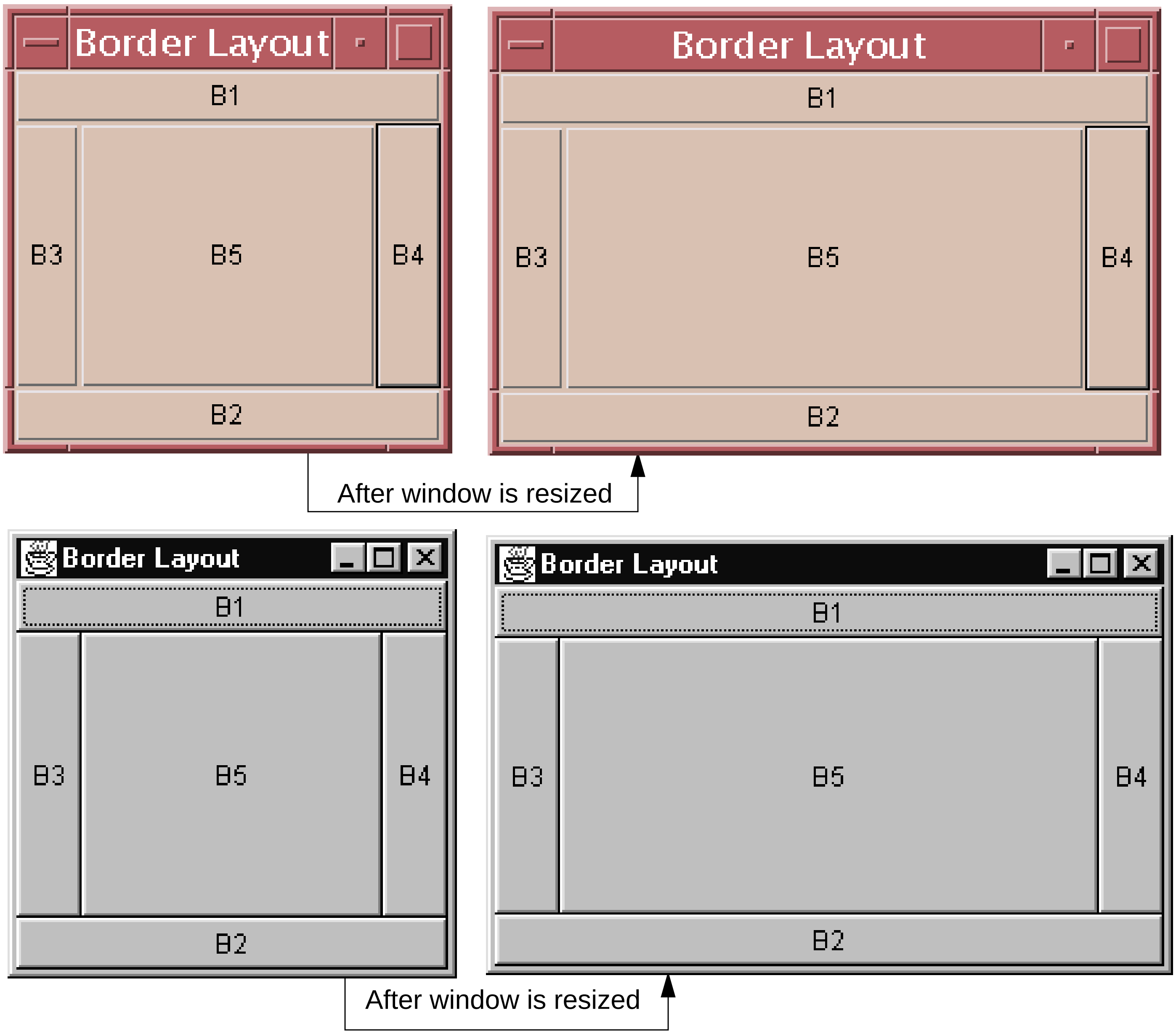
- Default layout for the `Frame` class
- Components added to specific regions
- The resizing behavior:
  - North, South, and Center regions adjust horizontally
  - East, West, and Center regions adjust vertically



BorderExample.java

```
import java.awt.*;

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

  public BorderExample() {
    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");
  }

  public void launchFrame() {
    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[]) {
    BorderExample guiWindow2 = new BorderExample();
    guiWindow2.launchFrame();
  }
}
```

BorderExample.java



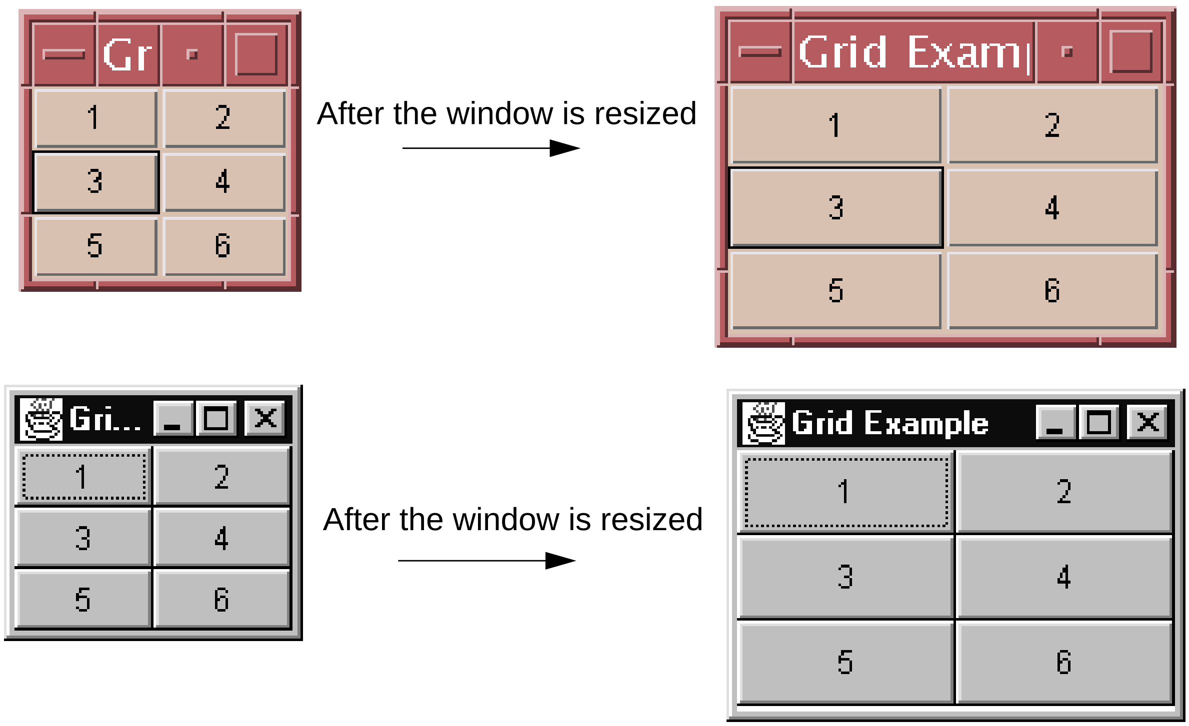
The `GridLayout` Manager

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

GridExample.java

```
import java.awt.*;

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

  public GridExample() {
    f = new Frame("Grid Example");
    b1 = new Button("1");
    b2 = new Button("2");
    b3 = new Button("3");
    b4 = new Button("4");
    b5 = new Button("5");
    b6 = new Button("6");
  }

  public void launchFrame() {
    f.setLayout (new GridLayout(3,2));

    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[]) {
    GridExample grid = new GridExample();
    grid.launchFrame();
  }
}
```

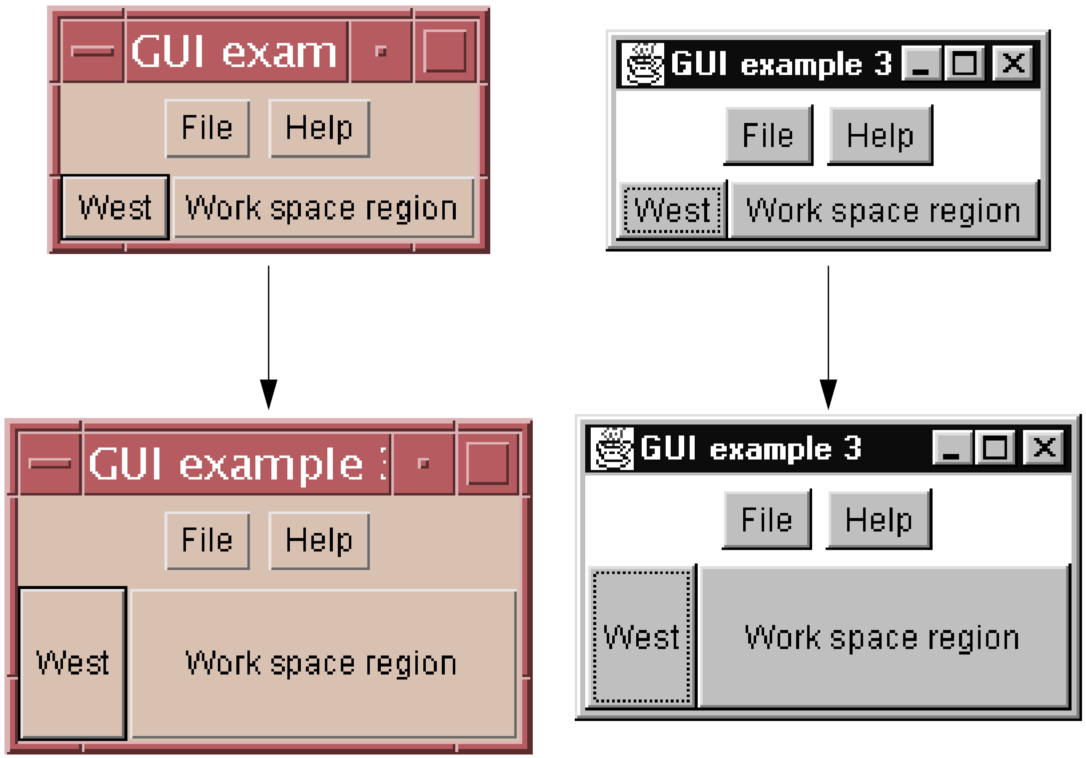
GridEx.java


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

ComplexLayoutExample.java

```
import java.awt.*;

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

  public ComplexLayoutExample() {
    f = new Frame("GUI example 3");
    bw = new Button("West");
    bc = new Button("Work space region");
    bfile = new Button("File");
    bhelp = new Button("Help");
  }

  public void launchFrame() {
    // Add bw and bc buttons in the frame border
    f.add(bw, BorderLayout.WEST);
    f.add(bc, BorderLayout.CENTER);
    // Create panel for the buttons in the north border
    p = new Panel();
    p.add(bfile);
    p.add(bhelp);
    f.add(p, BorderLayout.NORTH);
    // Pack the frame and make it visible
    f.pack();
    f.setVisible(true);
  }

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

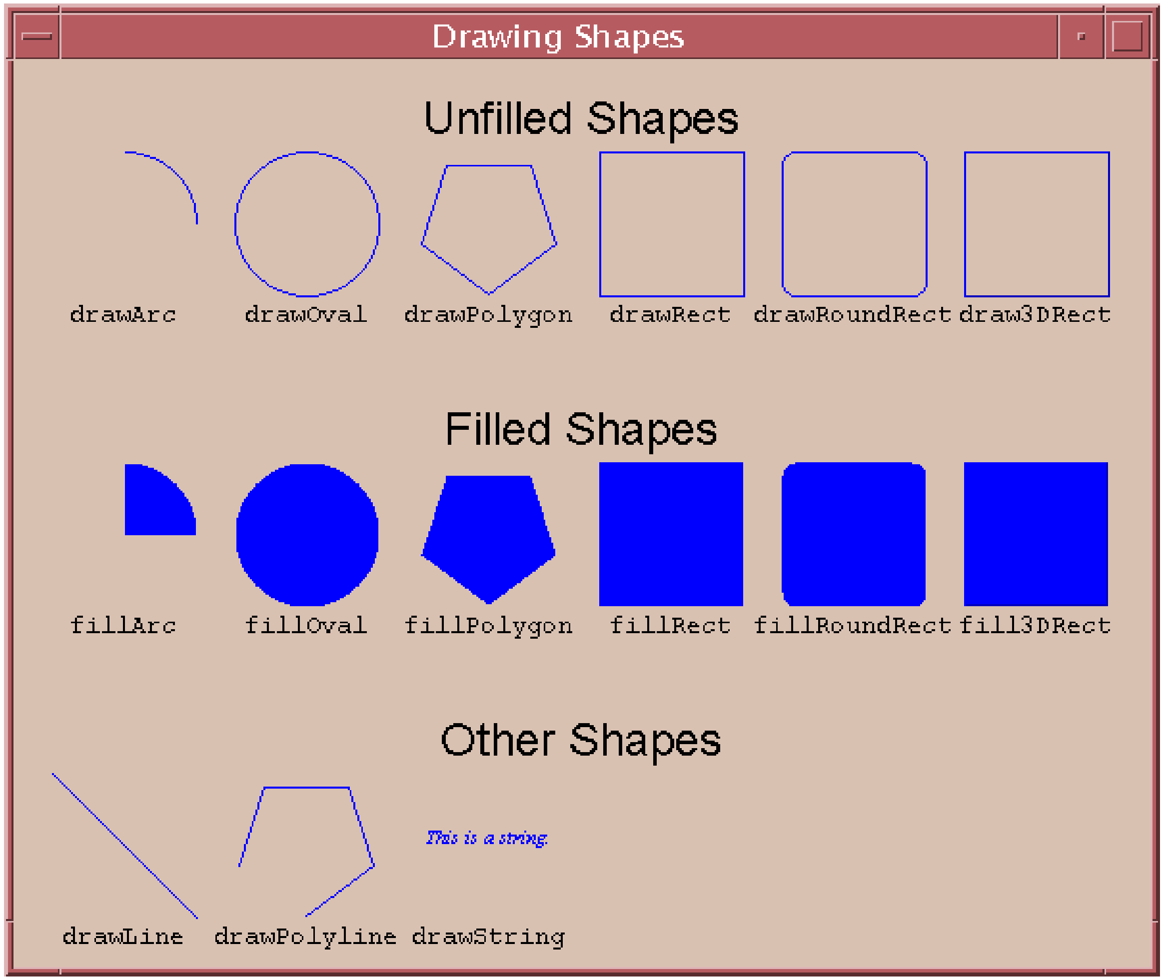
Output of `ComplexLayoutExample.java`



Drawing in AWT

- You can draw in any `Component` (although AWT provides the `Canvas` and `Panel` classes just for this purpose).
- Typically, you would create a subclass of `Canvas` or `Panel` and override the `paint` method.
- The `paint` method is called every time the component is shown (for example, if another window was overlapping the component and then removed).
- Every component has a `Graphics` object.
- The `Graphics` class implements many drawing methods.

Drawing With the Graphics Object



Exercise: Building Java GUIs

- Exercise objective:
  - Develop a GUI for a “chat room” application and a “calculator” application using the AWT.
- Tasks:
  - Complete the tasks specified by the instructor.

Check Your Progress

- Describe the AWT package and its components
- Define the terms *containers*, *components*, and *layout managers*, and describe how they work together to build a GUI
- Use layout managers
- Use the `FlowLayout`, `BorderLayout`, and `GridLayout` 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 technology 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 11

GUI Event Handling

Objectives

- Define events and event handling
- 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

Objectives

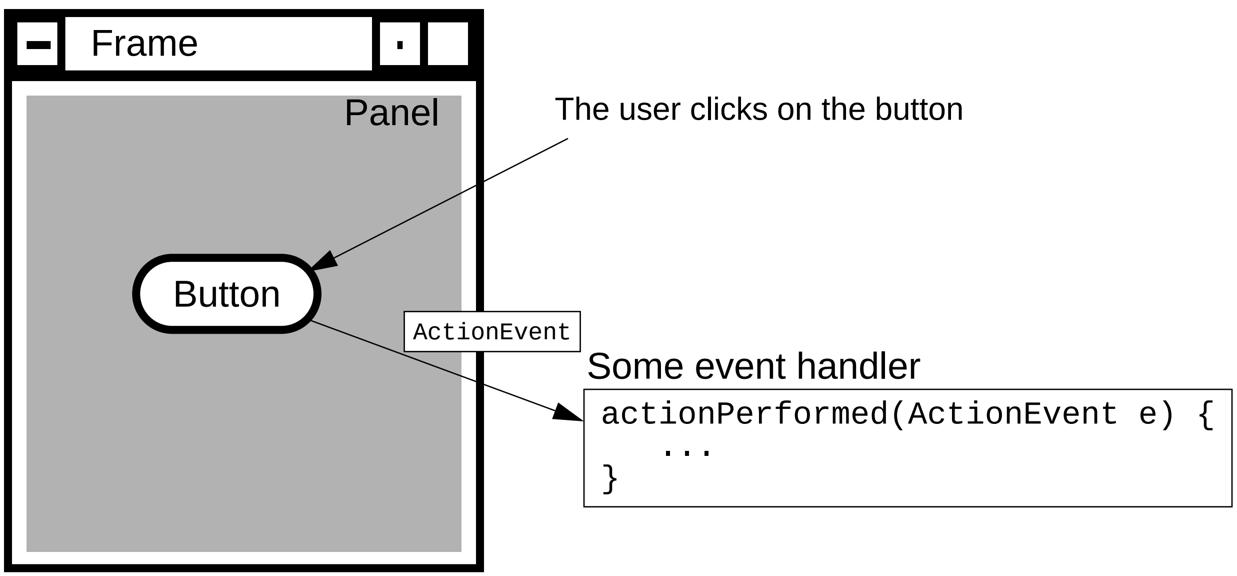
- Identify the appropriate interface for a variety of event types
- Create the appropriate event handler methods for a variety of event types
- Understand the use of inner classes and anonymous classes in event handling

Relevance

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

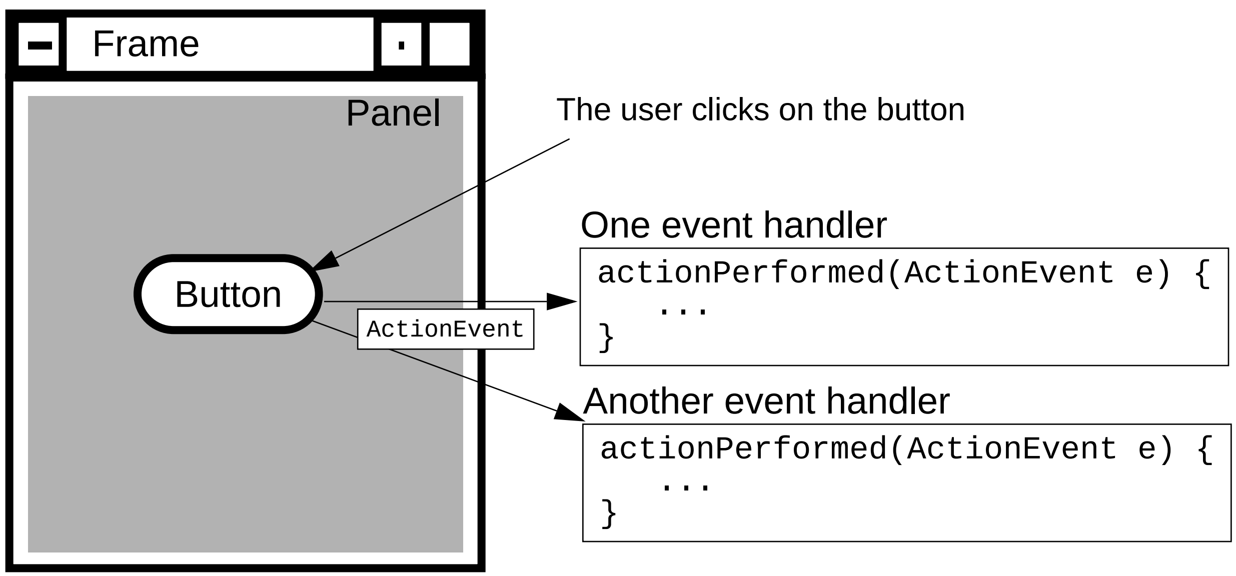
What Is an Event?

- 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



Delegation Model

- An event can be sent to many event handlers.



- Event handlers register with components when they are interested in events generated by that component.

Delegation Model

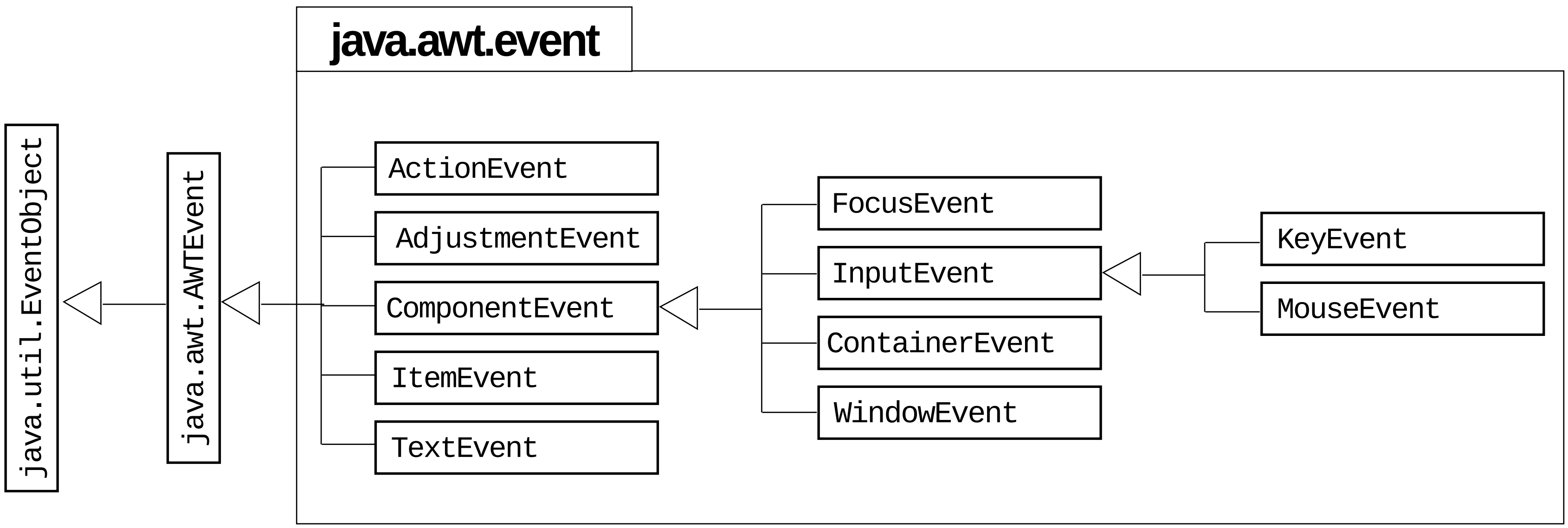
```
import java.awt.*;

public class TestButton {
  private Frame f;
  private Button b;

  public TestButton() {
    f = new Frame("Test");
    b = new Button("Press Me!");
    b.setActionCommand("ButtonPressed");
  }

  public void launchFrame() {
    b.addActionListener(new ButtonHandler());
    f.add(b,BorderLayout.CENTER);
    f.pack();
    f.setVisible(true);
  }

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

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

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

Delegation Model

- Client objects (handlers) register with a GUI component they want to observe.
- GUI components only trigger the handlers for the type of event that has occurred.
  - Most components can trigger more than one type of event.
- Distributes the work among multiple classes.

Event Categories



Java GUI Behavior

Category	**Interface Name**	**Methods**
Action	`ActionListener`	`actionPerformed(ActionEvent)`
Item	`ItemListener`	`itemStateChanged(ItemEvent)`

Mouse	`MouseListener`	`mousePressed(MouseEvent)`
		`mouseReleased(MouseEvent)`
		`mouseEntered(MouseEvent)`
		`mouseExited(MouseEvent)`
		`mouseClicked(MouseEvent)`

Mouse Motion	`MouseMotionListener`	`mouseDragged(MouseEvent)` `mouseMoved(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)`

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 TwoListener
     implements MouseMotionListener,
                MouseListener {
  private Frame f;
  private TextField tf;

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

  public void launchFrame() {
    Label label = new Label("Click and drag the mouse");
    // Add components to the frame
    f.add(label, BorderLayout.NORTH);
    f.add(tf, BorderLayout.SOUTH);
    // Add this object as a listener
    f.addMouseMotionListener(this);
    f.addMouseListener(this);
    // Size the frame and make it visible
    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);
  }
```

Complex Example

```

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

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

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

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

Multiple Listeners

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

Event Adapters

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

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

public class MouseClickHandler extends MouseAdapter {

  // 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...
  }
}
```

Inner Classes

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

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

  public TestInner() {
    f = new Frame("Inner classes example");
    tf = new TextField(30);
  }

  public void launchFrame() {
    Label label = new Label("Click and drag the mouse");
    // Add components to the frame
    f.add(label, BorderLayout.NORTH);
    f.add(tf, BorderLayout.SOUTH);
    // Add a listener that uses an Inner class
    f.addMouseMotionListener(new MyMouseMotionListener());
    f.addMouseListener(new MouseClickHandler());
    // Size the frame and make it visible
    f.setSize(300, 200);
    f.setVisible(true);
  }

  class MyMouseMotionListener extends MouseMotionAdapter {
      public void mouseDragged(MouseEvent e) {
        String s = "Mouse dragging:  X = "+ e.getX()
                    + " Y = " + e.getY();
        tf.setText(s);
      }
    }

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

Anonymous Classes

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

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

  public TestAnonymous() {
    f = new Frame("Anonymous classes example");
    tf = new TextField(30);
  }

  public void launchFrame() {
    Label label = new Label("Click and drag the mouse");
    // Add components to the frame
    f.add(label, BorderLayout.NORTH);
    f.add(tf, BorderLayout.SOUTH);
    // Add a listener that uses an anonymous class
    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()); // Not shown
    // Size the frame and make it visible
    f.setSize(300, 200);
    f.setVisible(true);
  }

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

Exercise: Working With Events

- Exercise objective:
  - Implement basic event handlers for the "chat room" GUI and the "calculator" GUI.
- Tasks:
  - Complete the tasks specified by the instructor.

Check Your Progress

- Define events and event handling
- 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

Check Your Progress

- Identify the appropriate interface for a variety of event types
- Create the appropriate event handler methods for a variety of event types
- Understand the use of inner classes and anonymous classes in event handling

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 12

GUI-Based Applications

Objectives

- Identify the key AWT components and the events that they trigger
- Describe how to construct a menu bar, menu, and menu items in a Java GUI
- Understand how to change the color and font of a component

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?
- How can you create a menu for your GUI frame?

AWT Components

Component Type	Description
`Button`	A named rectangular box used for receiving mouse clicks
`Canvas`	A panel used for drawing
`Checkbox`	A component allowing the user to select an item
`CheckboxMenuItem`	A checkbox within a menu
`Choice`	A pull-down static list of items
`Component`	The parent of all AWT components, except menu components
`Container`	The parent of all AWT containers
`Dialog`	A top-level window with a title and a border; dialogs can be modeless or modal.
`Frame`	The base class of all GUI windows with window manager controls
`Label`	A text string component
`List`	A component that contains a dynamic set of items
`Menu`	An element under the menu bar, which contains a set of menu items
`MenuItem`	An item within a menu
`Panel`	A basic container class used most often to create complex layouts
`Scrollbar`	A component that allows a user to “select from a range of values”
`ScrollPane`	A container class that implements automatic horizontal and vertical scrolling for a single child component
`TextArea`	A component that allows the user to enter a block of text
`TextField`	A component that allows the user to enter a single line of text
`Window`	The base class of all GUI windows, without window manager controls

Component Events

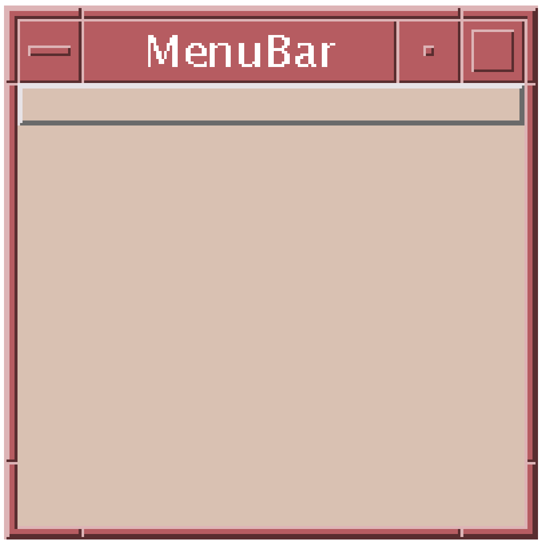
Component Type	Act	Adj	Cmp	Cnt	Foc	Itm	Key	Mou	MM	Text	Win
Button	✓		✓		✓		✓	✓	✓		
Canvas			✓		✓		✓	✓	✓		
Checkbox			✓		✓	✓	✓	✓	✓		
CheckboxMenuItem						✓					
Choice			✓		✓	✓	✓	✓	✓		
Component			✓		✓		✓	✓	✓		
Container			✓	✓	✓		✓	✓	✓		
Dialog			✓	✓	✓		✓	✓	✓		✓
Frame			✓	✓	✓		✓	✓	✓		✓
Label			✓		✓		✓	✓	✓		
List	✓		✓		✓	✓	✓	✓	✓		
MenuItem	✓										
Panel			✓	✓	✓		✓	✓	✓		
Scrollbar		✓	✓		✓		✓	✓	✓		
ScrollPane			✓	✓	✓		✓	✓	✓		
TextArea			✓		✓		✓	✓	✓	✓	
TextField	✓		✓		✓		✓	✓	✓	✓	
Window			✓	✓	✓		✓	✓	✓		✓

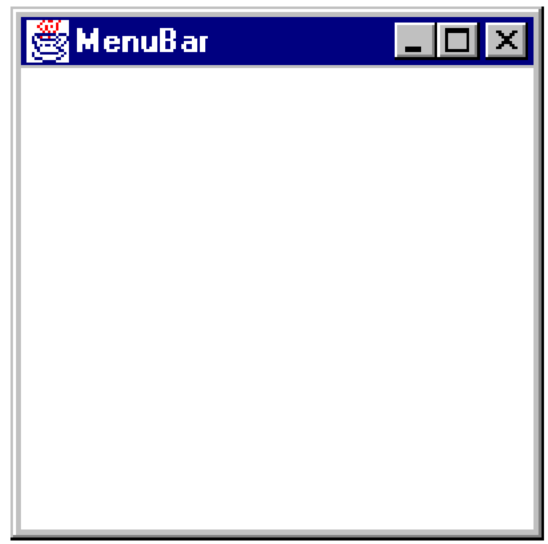
How to Create a Menu

1. Create a `MenuBar` object, and set it into a menu container, such as a `Frame`.
2. Create one or more `Menu` objects, and add them to the menu bar object.
3. Create one or more `MenuItem` objects, and add them to the menu object.

Creating a MenuBar

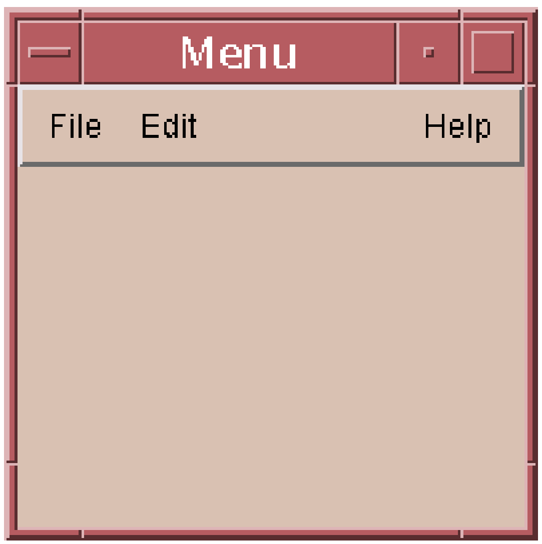
```
Frame f = new Frame("MenuBar");
MenuBar mb = new MenuBar();
f.setMenuBar(mb);
```

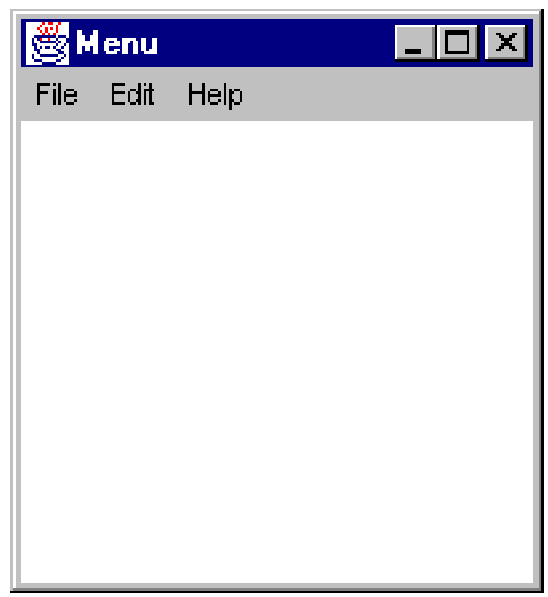




Creating a Menu

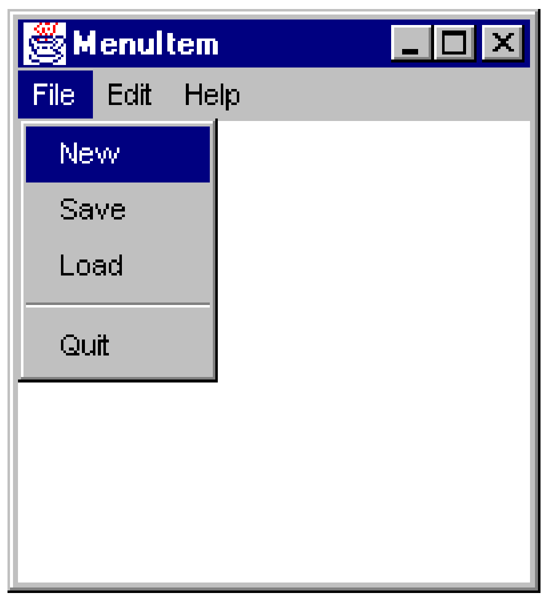
```
Frame f = new Frame("Menu");
MenuBar mb = new MenuBar();
Menu m1 = new Menu("File");
Menu m2 = new Menu("Edit");
Menu m3 = new Menu("Help");
mb.add(m1);
mb.add(m2);
mb.setHelpMenu(m3);
f.setMenuBar(mb);
```





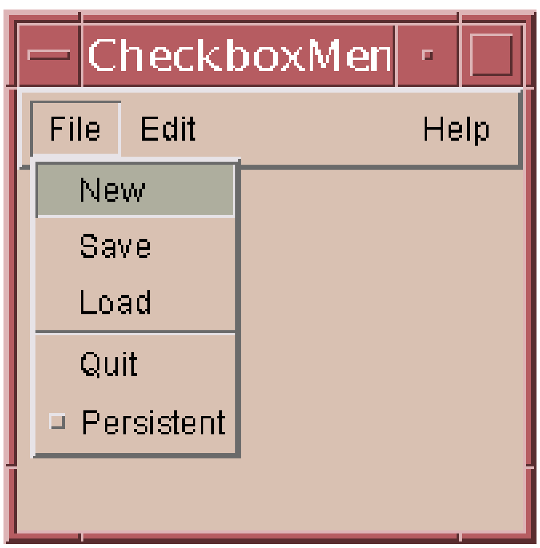
Creating a MenuItem

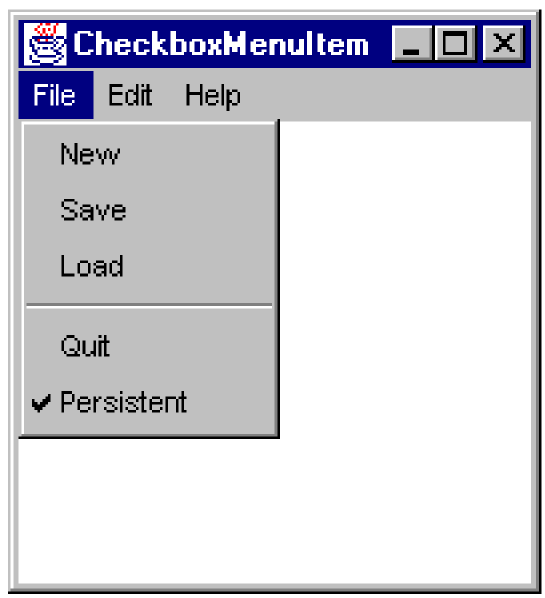
```
MenuItem mi1 = new MenuItem("New");
MenuItem mi2 = new MenuItem("Save");
MenuItem mi3 = new MenuItem("Load");
MenuItem mi4 = new MenuItem("Quit");
mi1.addActionListener(this);
mi2.addActionListener(this);
mi3.addActionListener(this);
mi4.addActionListener(this);
m1.add(mi1);
m1.add(mi2);
m1.add(mi3);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



Creating a `CheckBoxMenuItem`

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





Controlling Visual Aspects

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

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

Swing

- Swing is a second-generation GUI toolkit.
- It builds on top of AWT, but supplants the components with “lightweight” versions.
- There are many more components, and much more complex components: `JTable`, `JTree`, and `JComboBox`.

Exercise: Building GUI-Based Applications

- Exercise objective:
  - Finish the GUI for a "chat room" application. Add menus to it and use a dialog box.
- Tasks:
  - Complete the tasks specified by the instructor.

Check Your Progress

- Identify the key AWT components and the events that they trigger
- Describe how to construct a menu bar, menu, and menu items in a Java GUI
- Understand how to change the color and font of a component

Think Beyond

- What problems occur when your GUI code must wait for the application logic to perform its job?
- What are the limitations of AWT?

Module 13

Threads

Objectives

- Define a thread
- Create separate threads in a Java technology 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

Objectives

- Use `wait` and `notify` to communicate between threads
- Use `synchronized` to protect data from corruption

Relevance

- How do you get programs to perform multiple tasks concurrently?

Threads

- What are threads?
  - Virtual CPU

Three Parts of a Thread

- CPU
- Code
- Data

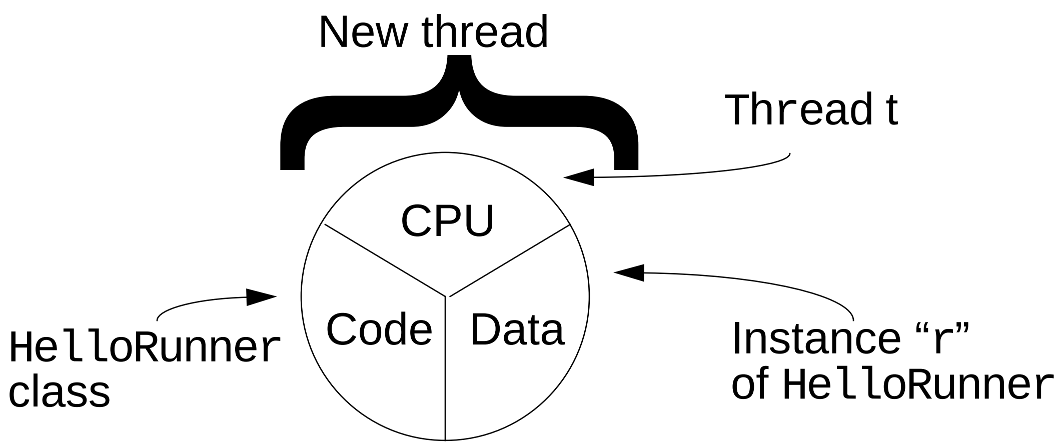
Creating the Thread

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

class HelloRunner implements Runnable {
  int i;

  public void run() {
    i = 0;

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

Creating the Thread

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

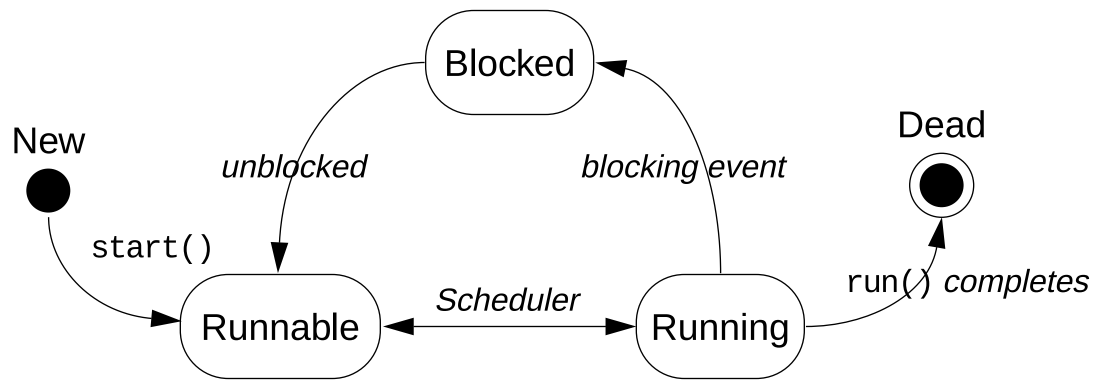
```
Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
```


New thread
Thread t
CPU
Code
Data
HelloRunner class
Instance "r" of HelloRunner

Starting the Thread

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

Thread Scheduling




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

Terminating a Thread

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

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

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

```
public class ThreadController {
  private Runner r = new Runner();
  private Thread t = new Thread(r);

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

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

Basic Control of Threads

- Testing threads:
  - `isAlive()`
- Accessing thread priority:
  - `getPriority()`
  - `setPriority()`
- Putting threads on hold:
  - `Thread.sleep()`
  - `join()`
  - `Thread.yield()`

The `join` Method

```
public static void main(String[] args) {
  Thread t = new Thread(new Runner());
  t.start();
  ...
  // Do stuff in parallel with the other thread for a while
  ...
  // Wait here for the timer thread to finish
  try {
    t.join();
  } catch (InterruptedException e) {
    // t came back early
  }
  ...
  // Now continue in this thread
  ...
}
```

Other Ways to Create Threads

```
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

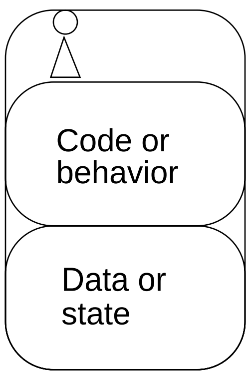
Using the `synchronized` Keyword

```
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`

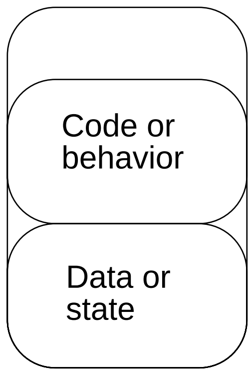


Thread before `synchronized(this)`

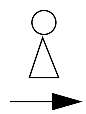
```
public void push(char c) {
  synchronized (this) {
    data[idx] = c;
    idx++;
  }
}
```

The Object Lock Flag

Object `this`



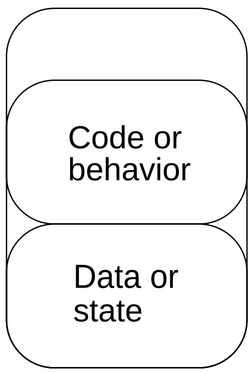
Thread after `synchronized(this)`



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

The Object Lock Flag

Object `this`
Lock flag missing



Thread, trying to execute
`synchronized(this)`

Waiting for
object lock

```
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, return, 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`.

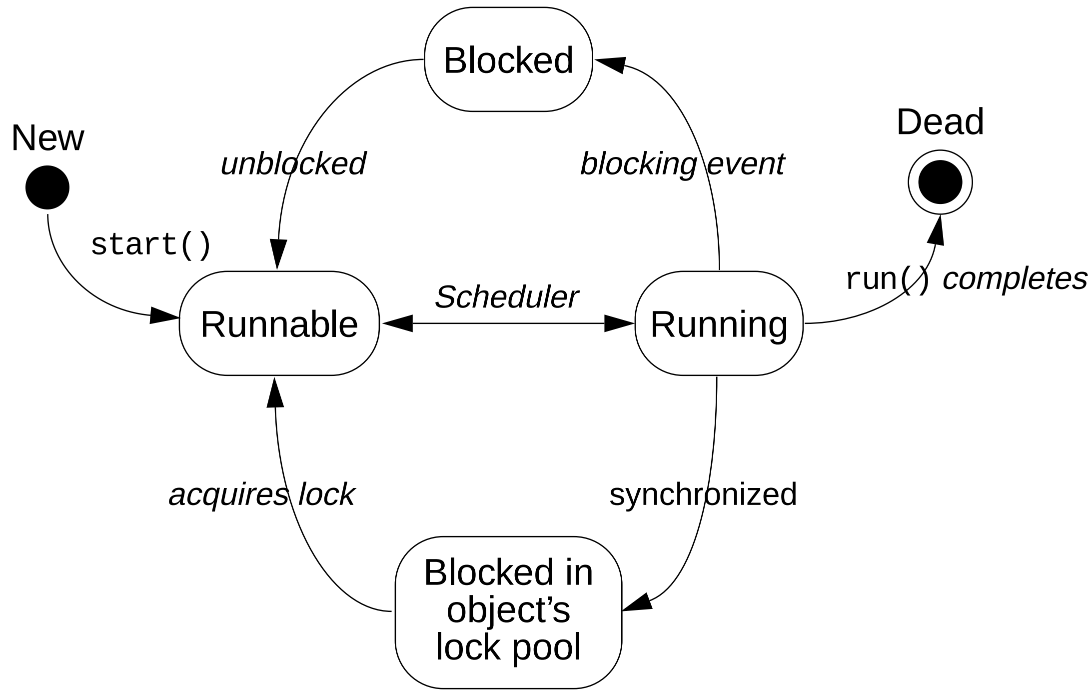
synchronized – Putting It Together

- The following two code segments are equivalent:

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

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

Threads State Diagram With Synchronization



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

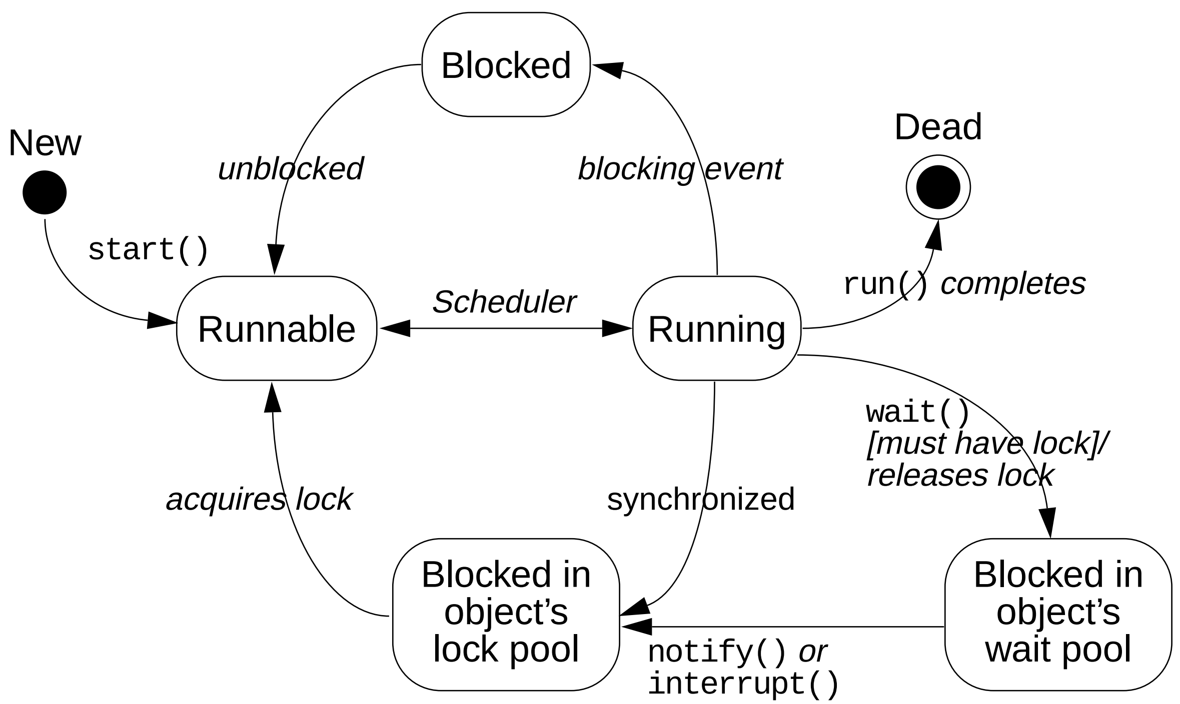
Thread Interaction – `wait` and `notify`

- 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
  - The cabbie drives and notifies you upon arrival at your destination

Thread Interaction

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

Threads State Diagram With `wait` and `notify`



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) { }

  }
}
```

The SyncStack Class

```
public class SyncStack {

  private List buffer = new ArrayList(400);

  public synchronized char pop() {
  }

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

The 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;
}
```

The push Method

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

SyncTest.java

```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

```
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.*;

public class SyncStack {
  private List buffer = new ArrayList(400);

  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.add(charObj);
  }
}
```

SyncStack Example

```
Producer2: F
Consumer1: F
Producer2: K
Consumer2: K
Producer2: T
Producer1: N
Producer1: V
Consumer2: V
Consumer1: N
Producer2: V
Producer2: U
Consumer2: U
Consumer2: V
Producer1: F
Consumer1: F
Producer2: M
Consumer2: M
Consumer2: T
```

Exercise: Using Multithreaded Programming

- Exercise objectives:
  - Become familiar with the concepts of multithreading by writing a multi-threaded program.
- Tasks:
  - Complete the tasks specified by the instructor

Check Your Progress

- Define a thread
- Create separate threads in a Java technology 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

Check Your Progress

- Use `wait` and `notify` to communicate between threads
- Use `synchronized` to protect data from corruption

Think Beyond

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

Module 14

Advanced I/O Streams

Objectives

- Describe the main features of the `java.io` package
- Construct node and processing streams, and use them appropriately
- Distinguish readers and writers from streams, and select appropriately between them

Relevance

- What mechanisms are in place within the Java programming language to read and write from sources (or sinks) other than files?
- How are international character sets supported in I/O operations?
- What are the possible sources and sinks of character and byte streams?

I/O Fundamentals

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

Fundamental Stream Classes

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

Data Within Streams

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

`InputStream` Methods

- The three basic `read` methods:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

- The other methods:

```
void close()
int available()
skip(long n)
boolean markSupported()
void mark(int readlimit)
void reset()
```

`OutputStream` Methods

- The three basic `write` methods:

```
void write(int c)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

- The other methods:

```
void close()
void flush()
```

`Reader` Methods

- The three basic `read` methods:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

- The other methods:

```
void close()
boolean ready()
skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```

`Writer` Methods

- The basic `write` methods:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

- The other methods:

```
void close()
void flush()
```

Node Streams

Type	Character Streams	Byte Streams
File	FileReader FileWriter	FileInputStream FileOutputStream
Memory: Array	CharArrayReader CharArrayWriter	ByteArrayInputStream ByteArrayOutputStream
Memory: String	StringReader StringWriter	
Pipe	PipedReader PipedWriter	PipedInputStream PipedOutputStream

A Simple Example

- This program performs a copy file operation:

`java TestNodeStreams file1 file2`

```
import java.io.*;

public class TestNodeStreams {
  public static void main(String[] args) {
    try {
      FileReader input = new FileReader(args[0]);
      FileWriter output = new FileWriter(args[1]);
      char[]     buffer = new char[128];
      int        charsRead;

      // read the first buffer
      charsRead = input.read(buffer);

      while ( charsRead != -1 ) {
        // write the buffer out to the output file
        output.write(buffer, 0, charsRead);

        // read the next buffer
        charsRead = input.read(buffer);
      }

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

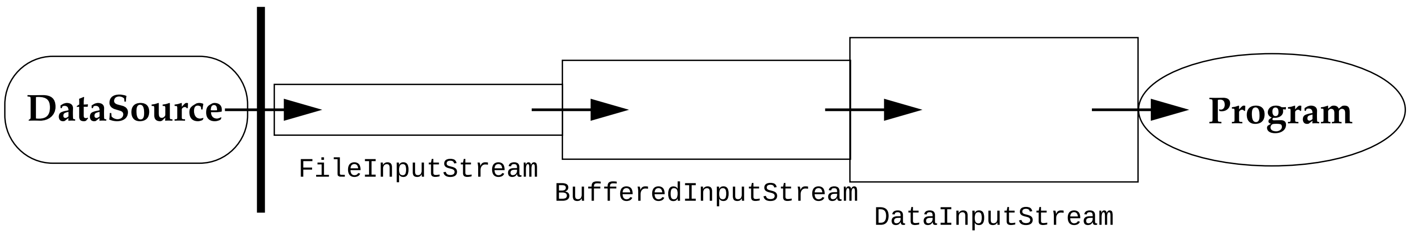
Buffered Streams

`java TestBufferedStreams file1 file2`

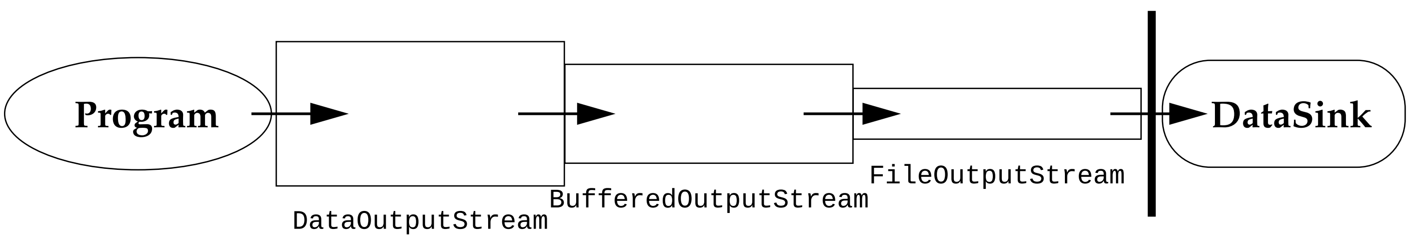
```
import java.io.*;

public class TestBufferedStreams {
  public static void main(String[] args) {
    try {
      FileReader     input     = new FileReader(args[0]);
      BufferedReader bufInput  = new BufferedReader(input);
      FileWriter     output    = new FileWriter(args[1]);
      BufferedWriter bufOutput = new BufferedWriter(output);
      String line;

      // read the first line
      line = bufInput.readLine();

      while ( line != null ) {
        // write the line out to the output file
        bufOutput.write(line, 0, line.length());
        bufOutput.newLine();

        // read the next line
        line = bufInput.readLine();
      }

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

I/O Stream Chaining

Input Stream Chain



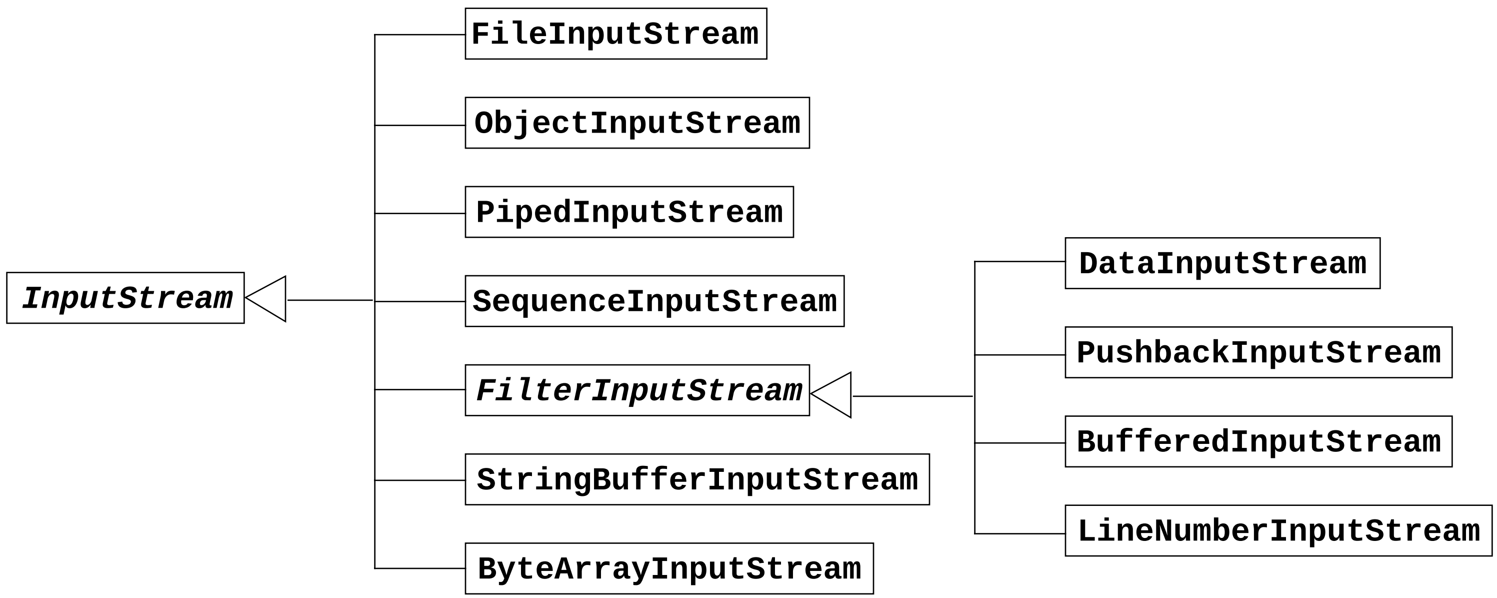
Output Stream Chain



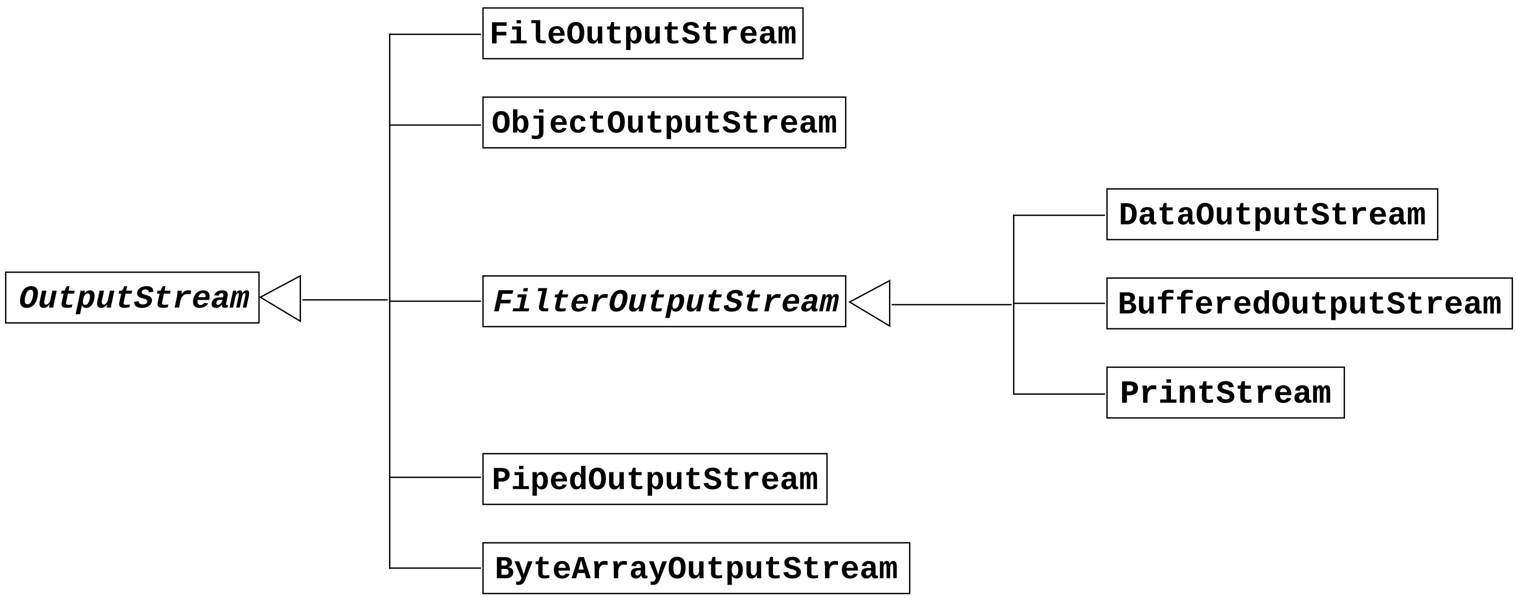
Processing Streams

Type	Character Streams	Byte Streams
Buffering	`BufferedReader` `BufferedWriter`	`BufferedInputStream` `BufferedOutputStream`
Filtering	*`FilterReader`* *`FilterWriter`*	*`FilterInputStream`* *`FilterOutputStream`*
Converting between bytes and character	`InputStreamReader` `OuptutStreamWriter`	
Object serialization		`ObjectInputStream` `ObjectOutputStream`
Data conversion		`DataInputStream` `DataOutputStream`
Counting	`LineNumberReader`	`LineNumberInputStream`
Peeking ahead	`PushbackReader`	`PushbackInputStream`
Printing	`PrintWriter`	`PrintStream`

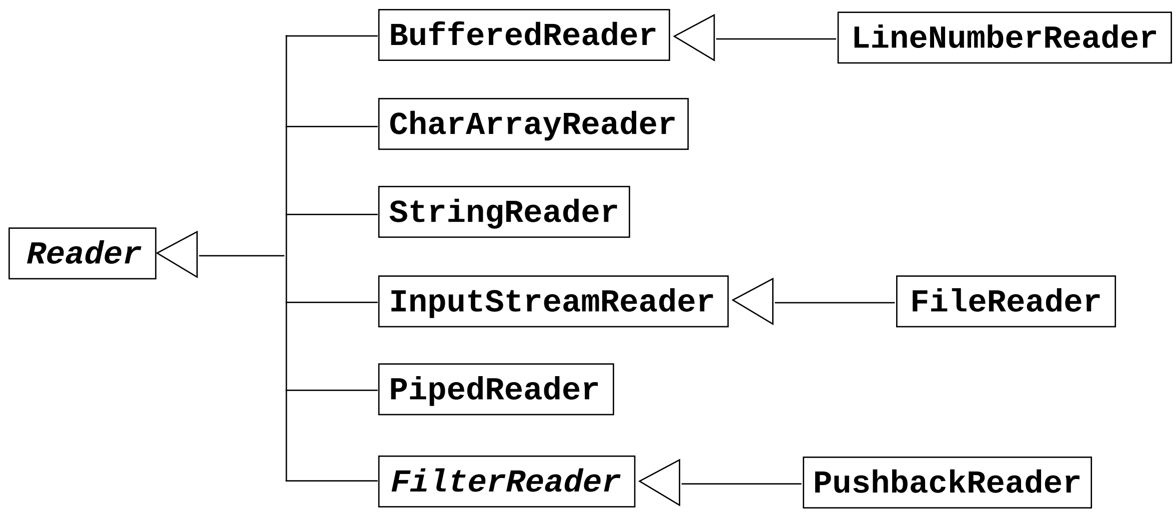
InputStream Class Hierarchy



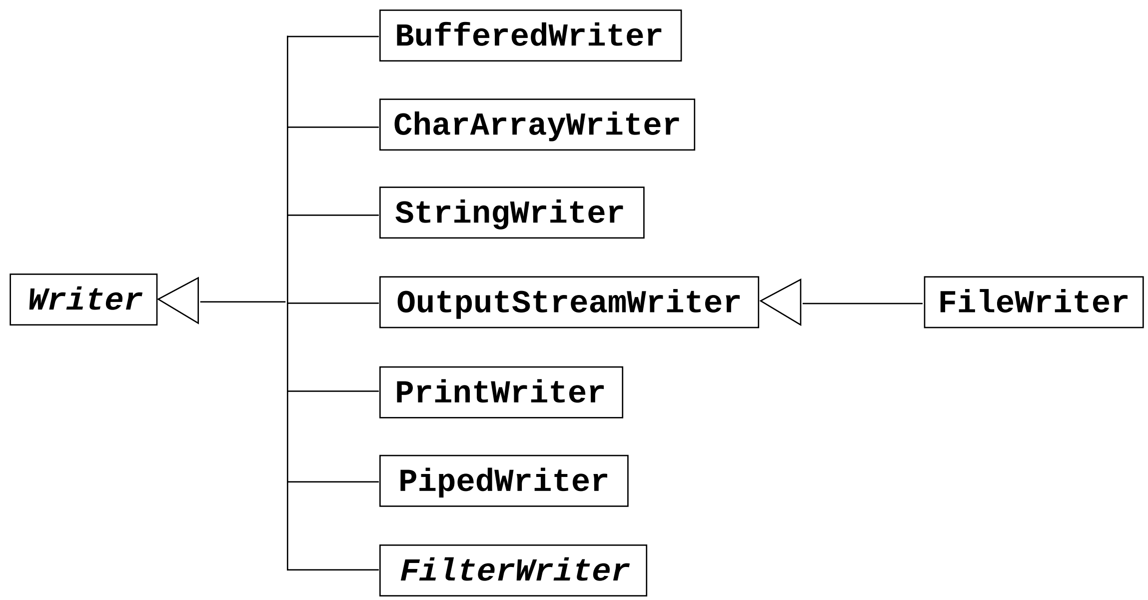
OutputStream Class Hierarchy



Reader Class Hierarchy



Writer Class Hierarchy



Check Your Progress

- Describe the main features of the `java.io` package
- Construct node and processing streams, and use them appropriately
- Distinguish readers and writers from streams, and select appropriately between them

Think Beyond

- Do you have applications that could benefit from creating specialized stream or character filters?

Module 15

Networking

Objectives

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

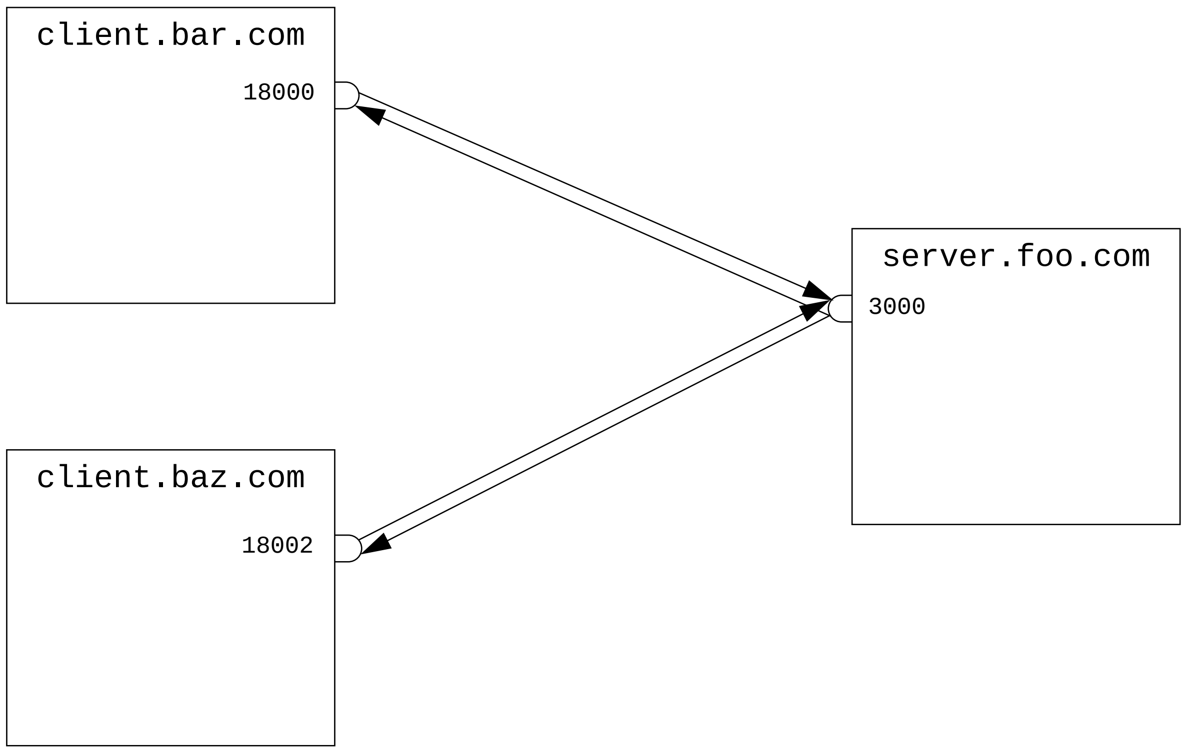
Relevance

- 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

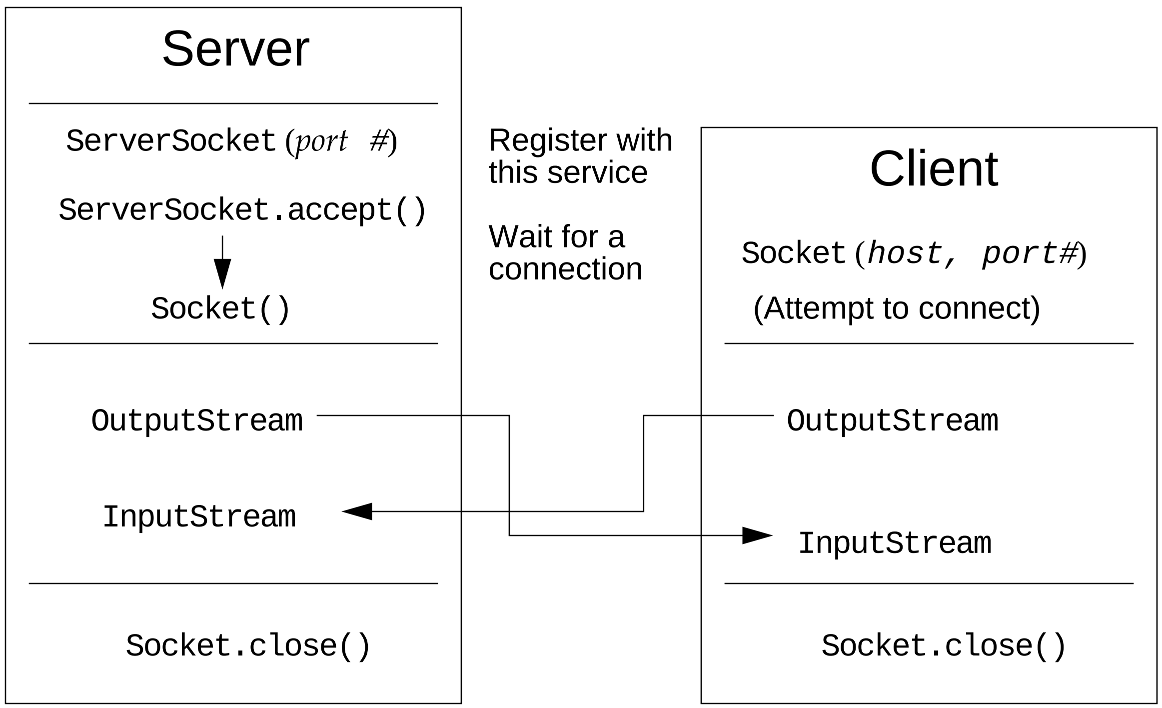
Networking


client.bar.com
18000
server.foo.com
3000
client.baz.com
18002

Networking With Java Technology

- Addressing the connection:
  - The address or name of remote machine
  - Port number to identify purpose at the server
- Port numbers:
  - Range from 0 to 65535

Java Networking Model



Minimal TCP/IP Server

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

public class SimpleServer {
  public static void main(String args[]) {
    ServerSocket s = null;

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

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

        // Get output stream associated with the socket
        OutputStream s1out = s1.getOutputStream();
        BufferedWriter bw = new BufferedWriter(
          new OutputStreamWriter(s1out));

        // Send your string!
        bw.write("Hello Net World!\n");

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

Minimal TCP/IP Client

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

public class SimpleClient {
  public static void main(String args[]) {
    try {
      // Open your connection to a server, at port 5432
      // localhost used here
      Socket s1 = new Socket("127.0.0.1", 5432);

      // Get an input stream from the socket
      InputStream is = s1.getInputStream();
      // Decorate it with a "data" input stream
      DataInputStream dis = new DataInputStream(is);

      // Read the input and print it to the screen
      System.out.println(dis.readUTF());

      // When done, just close the steam and connection
      dis.close();
      s1.close();
    } catch (ConnectException connExc) {
      System.err.println("Could not connect to the server.");
    } catch (IOException e) {
      // ignore
    }
  }
}
```

Exercise: Using Socket Programming

- Exercise objective:
  - Finish the "chat room" client program. Your client will connect to a "chat server" using sockets so that you can chat with other students in the class
- Tasks:
  - Complete the tasks specified by the instructor

Check Your Progress

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

Think Beyond

- How can you create a distributed object system using object serialization and these network protocols? Have you heard of Remote Method Invocation (RMI)?
- There are several advanced Java platform topics, many of which are addressed in other Sun Educational Services courses.

Course Contents

About This Course .... ***Preface-iv***

- Course Goals .... Preface-v
- Course Overview .... Preface-vi
- Course Map .... Preface-vii
- Module-by-Module Overview .... Preface-viii
- Course Objectives .... Preface-x
- Guidelines for Module Pacing .... Preface-xii
- Topics Not Covered .... Preface-xiii
- How Prepared Are You? .... Preface-xiv
- Introductions .... Preface-xv
- How to Use Course Materials .... Preface-xvi
- Course Icons .... Preface-xvii
- Typographical Conventions .... Preface-xviii

Getting Started ***1-1***

- Objectives 1-2
- Relevance 1-3
- What Is the Java Technology? 1-4
- Primary Goals of the Java Technology 1-5
- The Java Virtual Machine 1-8
- Garbage Collection 1-11
- Code Security 1-12
- Just-In-Time (JIT) Code Generator 1-13
- The Java™ Runtime Environment 1-14
- The Class Loader 1-15
- The Bytecode Verifier 1-16

A Basic Java Application 1-17
Compiling and Running the `TestGreeting` Program 1-18
Compile-Time Errors 1-19
Runtime Errors 1-20
Java Runtime Environment 1-21
Exercise Performing Basic Tasks 1-22
Check Your Progress 1-23
Think Beyond 1-24

Object-Oriented Programming 2-1
Objectives 2-2
Relevance 2-4
Software Engineering 2-5
The Analysis and Design Phase 2-6
Abstraction 2-7
Classes as Blueprints for Objects 2-8
Declaring Java Technology Classes 2-9
Declaring Attributes 2-10
Declaring Methods 2-11
Accessing Object Members 2-12
Information Hiding 2-13
Encapsulation 2-15
Declaring Constructors 2-16
The Default Constructor 2-17
Source File Layout 2-18
Software Packages 2-19
The `package` Statement 2-20
The `import` Statement 2-21
Directory Layout and Packages 2-22
Development 2-23

Terminology Recap 2-24
Using the Java API Documentation 2-25
Example API Documentation Page 2-26
Exercise: Using Objects and Classes 2-27
Check Your Progress 2-28
Think Beyond 2-30

Identifiers, Keywords, and Types 3-1
Objectives 3-2
Relevance 3-4
Comments 3-5
Semicolons, Blocks, and White Space 3-6
Identifiers 3-8
Java Keywords 3-9
Primitive Types 3-10
Logical – `boolean` 3-11
Textual – `char` and `String` 3-12
Integral – `byte`, `short`, `int`, and `long` 3-14
Floating Point – `float` and `double` 3-16
Variables, Declarations, and Assignments 3-18
Java Reference Types 3-19
Constructing and Initializing Objects 3-20
Memory Allocation and Layout 3-21
Explicit Attribute Initialization 3-22
Executing the Constructor 3-23
Assigning a Variable 3-24
Pass-by-Value 3-26
The `this` Reference 3-28
Java Programming Language Coding Conventions 3-30
Exercise: Using Objects 3-32

Check Your Progress 3-33
Think Beyond 3-35

Expressions and Flow Control* *4-1
Objectives 4-2
Relevance 4-4
Variables and Scope 4-5
Variable Scope Example 4-6
Variable Initialization 4-7
Operators 4-8
Logical Operators 4-9
Bitwise Logical Operators 4-10
Right-Shift Operators >> and >>> 4-11
Left-Shift Operator (<<) 4-12
Shift Operator Examples 4-13
String Concatenation With + ... 4-14
Casting 4-15
Promotion and Casting of Expressions 4-16
Branching Statements 4-17
Looping Statements 4-22
Special Loop Flow Control 4-25
Exercise: Using Expressions 4-30
Check Your Progress 4-31
Think Beyond 4-33

Arrays* *5-1
Objectives 5-2
Relevance 5-3
Declaring Arrays 5-4
Creating Arrays 5-5

Initializing Arrays 5-7
Multidimensional Arrays 5-8
Array Bounds 5-10
Array Resizing 5-11
Exercise: Using Arrays 5-13
Check Your Progress 5-14
Think Beyond 5-15

Class Design 6-1
Objectives 6-2
Relevance 6-4
Subclassing 6-5
Single Inheritance 6-8
Access Control 6-10
Overriding Methods 6-11
The `super` Keyword 6-13
Polymorphism 6-15
Rules About Overridden Methods 6-18
Heterogeneous Collections 6-20
Polymorphic Arguments 6-21
The `instanceof` Operator 6-22
Casting Objects 6-23
Overloading Method Names 6-24
Overloading Constructors 6-25
Constructors Are Not Inherited 6-27
Invoking Parent Class Constructors 6-28
Constructing and Initializing Objects: A Slight Reprise 6-30
Constructor and Initialization Example 6-31
The `Object` Class 6-33
The `==` Operator Compared With the `equals` Method 6-34

`equals` Example 6-35
The `toString` Method 6-37
Wrapper Classes 6-38
Exercise: Investing Subclasses 6-40
Check Your Progress 6-41
Think Beyond 6-43

Advanced Class Features 7-1
Objectives 7-2
Relevance 7-4
The `static` Keyword 7-5
Class Attributes 7-6
Class Methods 7-8
Static Initializers 7-9
The Singleton Design Pattern 7-11
Implementing the Singleton Design Pattern 7-12
The `final` Keyword 7-13
Final Variables 7-14
Exercise: Working With the `static` and `final` Keywords 7-15
Abstract Classes: Scenario 7-16
Abstract Classes: Solution 7-18
Template Method Design Pattern 7-20
Interfaces 7-21
Interface Example 7-22
Multiple Interface Example 7-28
Uses of Interfaces 7-30
Nested Classes 7-31
Nested Class Example 7-32
Properties of Nested Classes 7-36
Exercise: Working With Interfaces and Abstract Classes 7-38

Check Your Progress 7-39
Think Beyond 7-41

***Exceptions and Assertions* 8-1**
Objectives 8-2
Relevance 8-3
Exceptions and Assertions 8-4
Exceptions 8-5
Exception Example 8-6
The `try` and `catch` Statements 8-7
Call Stack Mechanism 8-8
The `finally` Statement 8-9
Exception Example Revisited 8-10
Exception Categories 8-11
Common Exceptions 8-12
The Handle or Declare Rule 8-13
Method Overriding and Exceptions 8-14
Creating Your Own Exceptions 8-16
Handling User-Defined Exceptions 8-17
Assertions 8-18
Recommended Uses of Assertions 8-19
Controlling Runtime Evaluation of Assertions 8-20
Exercise: Handling and Creating Exceptions 8-21
Check Your Progress 8-22
Think Beyond 8-23

***Text-Based Applications* 9-1**
Objectives 9-2
Relevance 9-4
Command-Line Arguments 9-5

System Properties 9-7
The `Properties` Class 9-8
System Properties 9-9
Console I/O 9-10
Writing to Standard Output 9-11
Reading From Standard Input 9-12
Files and File I/O 9-13
Creating a New `File` Object 9-14
`File` Tests and Utilities 9-15
File Stream I/O 9-17
File Input Example 9-18
File Output Example 9-19
Exercise: Writing User Input to a File 9-20
The `Math` Class 9-21
The `String` Class 9-22
The `StringBuffer` Class 9-23
A `Set` Example 9-26
A `List` Example 9-27
Iterators 9-28
The Iterator Interface Hierarchy 9-29
Collections in JDK 1.1 9-30
Exercise: Using Collections to Represent Aggregation 9-31
Deprecation 9-32
Check Your Progress 9-36
Think Beyond 9-38

Building Java GUIs 10-1
Objectives 10-2
Relevance 10-4
Abstract Window Toolkit (AWT) 10-5

The `java.awt` Package 10-6
Containers 10-7
Positioning Components 10-8
Frames 10-9
`FrameExample.java` 10-10
Panels 10-12
`FrameWithPanel.java` 10-13
Container Layouts 10-15
Default Layout Managers 10-16
A Simple FlowLayout Example 10-17
The `FlowLayout` Manager 10-18
`FlowExample.java` 10-20
The `BorderLayout` Manager 10-21
`BorderExample.java` 10-22
The `GridLayout` Manager 10-24
`GridExample.java` 10-25
`ComplexLayoutExample.java` 10-27
Output of `ComplexLayoutExample.java` 10-28
Drawing in AWT 10-29
Drawing With the Graphics Object 10-30
Exercise: Building Java GUIs 10-31
Check Your Progress 10-32
Think Beyond 10-34

GUI Event Handling* *11-1
Objectives 11-2
Relevance 11-4
What Is an Event? 11-5
Delegation Model 11-6
Event Categories 11-9

Java GUI Behavior 11-10
Complex Example 11-11
Multiple Listeners 11-13
Event Adapters 11-14
Inner Classes 11-15
Anonymous Classes 11-16
Exercise: Working With Events 11-17
Check Your Progress 11-18
Think Beyond 11-20

GUI-Based Applications 12-1
Objectives 12-2
Relevance 12-3
AWT Components 12-4
Component Events 12-5
How to Create a Menu 12-6
Creating a `MenuBar` 12-7
Creating a `Menu` 12-8
Creating a `MenuItem` 12-9
Creating a `CheckBoxMenuItem` 12-10
Controlling Visual Aspects 12-11
Swing 12-12
Exercise: Building GUI-Based Applications 12-13
Check Your Progress 12-14
Think Beyond 12-15

Threads 13-1
Objectives 13-2
Relevance 13-4
Threads 13-5

Three Parts of a Thread 13-6
Creating the Thread 13-7
Starting the Thread 13-10
Thread Scheduling 13-11
Terminating a Thread 13-12
Basic Control of Threads 13-13
Other Ways to Create Threads 13-15
Selecting a Way to Create Threads 13-16
Using the `synchronized` Keyword 13-17
The Object Lock Flag 13-18
Releasing the Lock Flag 13-21
`synchronized` – Putting It Together 13-22
Threads State Diagram With Synchronization 13-24
Deadlock 13-25
Thread Interaction – `wait` and `notify` 13-26
Thread Interaction 13-27
Threads State Diagram With `wait` and `notify` 13-28
Monitor Model for Synchronization 13-29
`Producer` 13-30
`Consumer` 13-31
The `SyncStack` Class 13-32
The `pop` Method 13-33
The `push` Method 13-34
`SyncTest.java` 13-35
`Producer.java` 13-36
`Consumer.java` 13-37
`SyncStack.java` 13-38
`SyncStack` Example 13-39
Exercise: Using Multithreaded Programming 13-40
Check Your Progress 13-41
Think Beyond 13-43

Advanced I/O Streams 14-1

Objectives 14-2
Relevance 14-3
I/O Fundamentals 14-4
Fundamental Stream Classes 14-5
Data Within Streams 14-6
`InputStream` Methods 14-7
`OutputStream` Methods 14-8
Reader Methods 14-9
Writer Methods 14-10
Node Streams 14-11
A Simple Example 14-12
Buffered Streams 14-13
I/O Stream Chaining 14-14
Processing Streams 14-15
`InputStream` Class Hierarchy 14-16
`OutputStream` Class Hierarchy 14-17
`Reader` Class Hierarchy 14-18
`Writer` Class Hierarchy 14-19
Check Your Progress 14-20
Think Beyond 14-21

Networking 15-1

Objectives 15-2
Relevance 15-3
Networking 15-4
Networking With Java Technology 15-6
Java Networking Model 15-7
Minimal TCP/IP Server 15-8
Minimal TCP/IP Client 15-9

Exercise: Using Socket Programming 15-10
Check Your Progress 15-11
Think Beyond 15-12