Java™ Programming Language

SL-275-SE6

Course Contents

Preface

About This Course

Course Goals

This course provides you with knowledge and skills to:

- Create Java™ technology applications that leverage the object-oriented features of the Java language, such as encapsulation, inheritance, and polymorphism
- Execute a Java technology application from the command-line
- Use Java technology data types and expressions
- Use Java technology flow control constructs
- Use arrays and other data collections
- Implement error-handling techniques using exception handling

Course Goals

- Create an event-driven graphical user interface (GUI) by using Java technology GUI components: panels, buttons, labels, text fields, and text areas
- Implement input/output (I/O) functionality to read from and write to data and text files
- Create multithreaded programs
- Create a simple Transmission Control Protocol/Internet Protocol (TCP/IP) client that communicates through sockets

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
- 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

Exceptions, Collections, and I/O

- Exceptions and Assertions
- Collections and Generics Framework
- I/O Fundamentals

Developing Graphical User Interfaces

- Console I/O and File I/O
- GUI Event Handling
- GUI-Based Applications

Advanced Java Programming

- Threads
- Networking

Topics Not Covered

- Object-oriented analysis and design – Covered in OO-226: *Object-Oriented Application Analysis and Design Using UML*
- General programming concepts – Covered in SL-110: *Fundamentals of the Java™ Programming Language*

How Prepared Are You?

Before attending this course, you should have completed SL-110: *Fundamentals of the Java™ Programming Language*, or have:

- Created and 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
- Experience related to topics presented in this course
- Reasons for enrolling in this course
- Expectations for this course

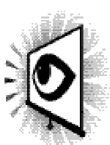
How to Use the Icons

Additional resources

Discussion

Note

Caution



Visual Aid

Typographical Conventions and Symbols

- `Courier` is used for the names of commands, files, directories, programming code, programming constructs, and on-screen computer output.
- **`Courier bold`** is used for characters and numbers that you type, and for each line of programming code that is referenced in a textual description.
- *`Courier italics`* is used for variables and command-line place holders that are replaced with a real name or value.
- ***`Courier italics bold`*** is used to represent variables whose values are to be entered by the student as part of an activity.

Typographical Conventions and Symbols

- *Palatino italics* is used for book titles, new words or terms, or words that are emphasized.

Additional Conventions

Java programming language examples use the following additional conventions:

- `Courier` is used for the class names, methods, and keywords.
- Methods are not followed by parentheses unless a formal or actual parameter list is shown.
- Line breaks occur where there are separations, conjunctions, or white space in the code.
- If a command on the Solaris™ Operating System (Solaris OS) is different from the Microsoft Windows platform, both commands are shown.

Module 1

Getting Started

Objectives

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

NOTE: The terms “Java Virtual Machine” and “JVM” mean a Virtual Machine for the Java™ platform.

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++.
- 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 changing programs dynamically during runtime by loading classes from disparate sources
- Furnishes better security

Primary Goals of the Java Technology

The following features fulfill these goals:

- The Java Virtual Machine (JVM™)[1]
- Garbage collection
- The Java Runtime Environment (JRE)
- JVM tool interface

1. The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java platform

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
- Fatal error reporting
- High-precision timing support

The Java Virtual Machine

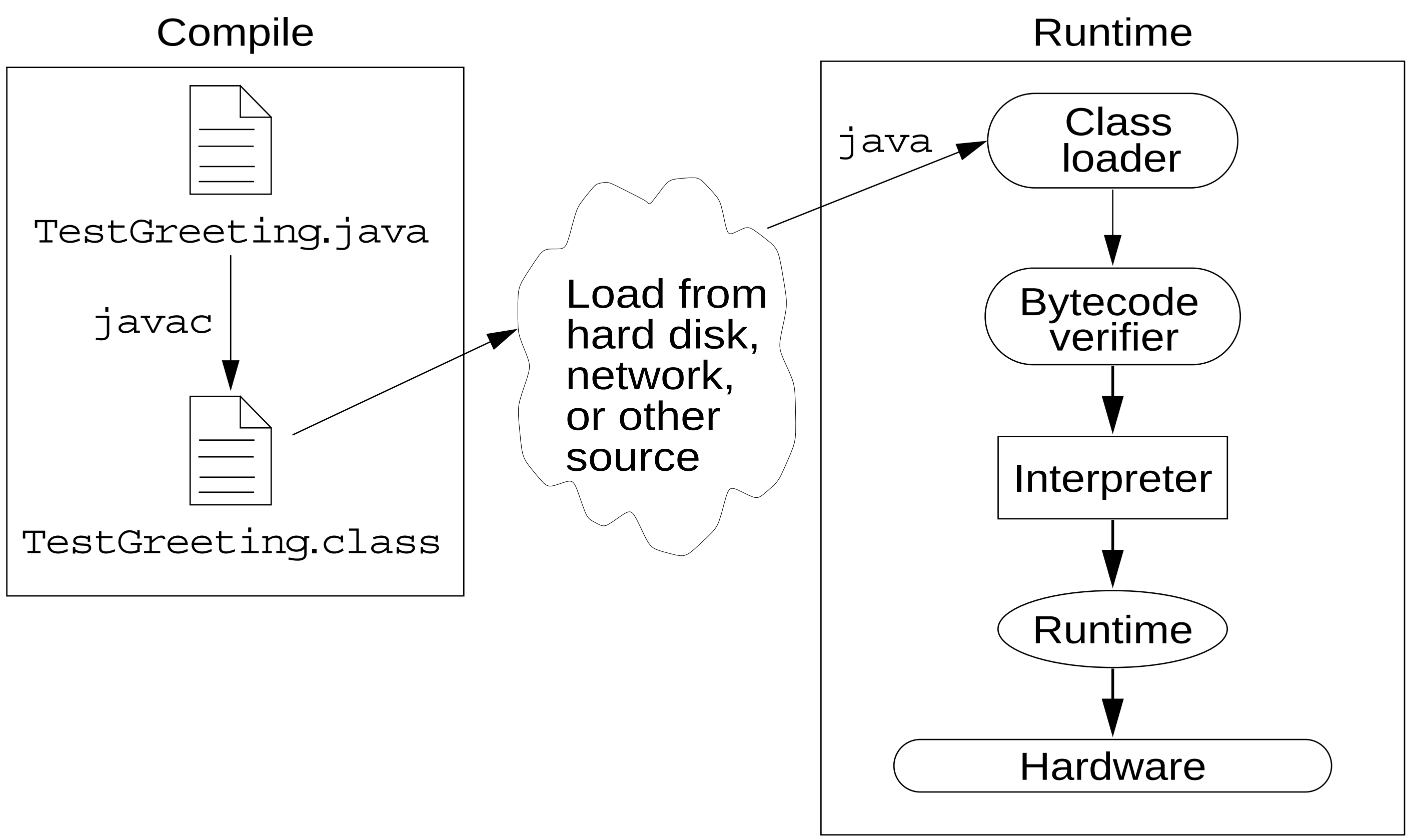
- The majority of type checking is done when the code is compiled.
- Implementation of the JVM approved by Sun Microsystems must be able to run any compliant class file.
- The JVM executes on multiple operating environments.

Garbage Collection

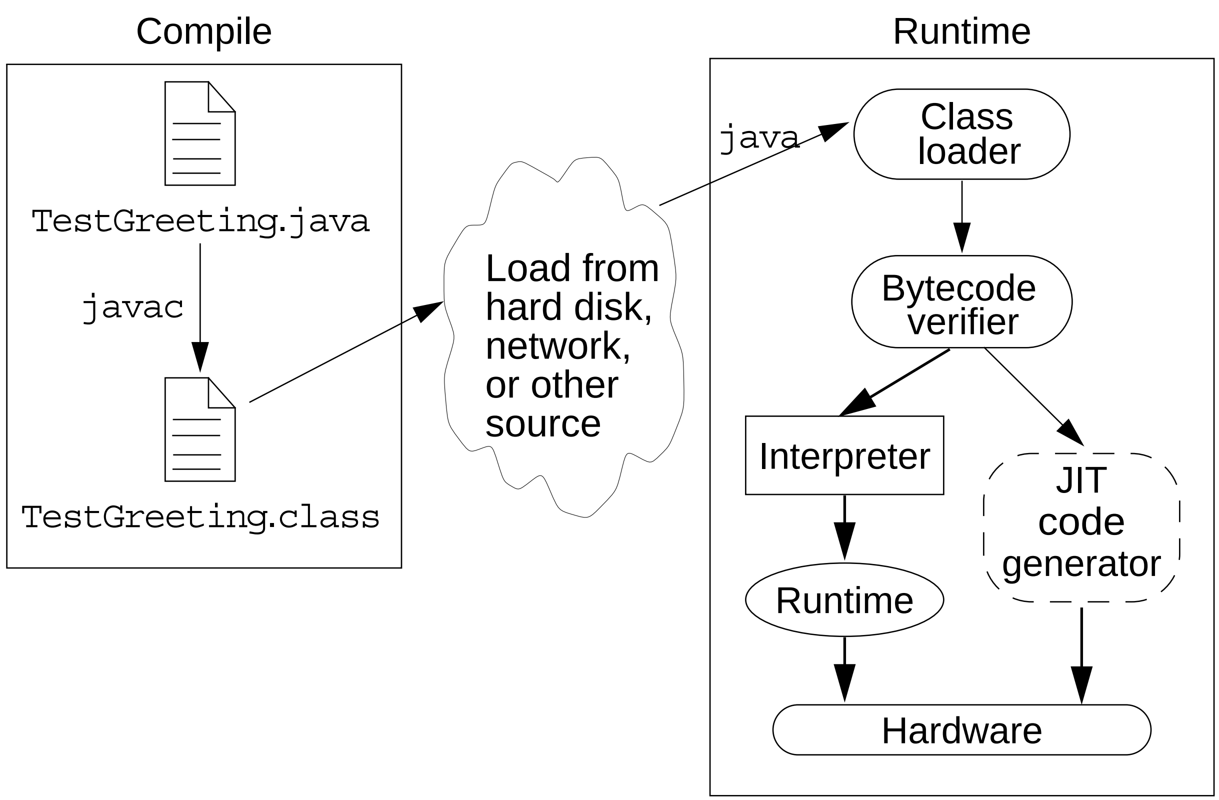
- Allocated memory that is no longer needed should be deallocated.
- In other languages, deallocation is the programmer’s responsibility.
- The Java programming language provides a system-level thread to track memory allocation.
- Garbage collection has the following characteristics:
  - Checks for and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

The Java Runtime Environment

The Java application environment performs as follows:



Operation of the JRE With a Just-In-Time (JIT) Compiler



JVM™ Tasks

The JVM 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 Simple Java Application

The `TestGreeting.java` Application

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

The `Greeting.java` Class

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

The `TestGreeting` Application

- Comment lines
- Class declaration
- The `main` method
- Method body

The `Greeting` Class

- Class declaration
- The `greet` method

Compiling and Running the `TestGreeting` Program

- Compile `TestGreeting.java`:

  `javac TestGreeting.java`
- The `Greeting.java` is compiled automatically.
- Run the application by using the following command:

  `java TestGreeting`
- Locate common compile and runtime errors.

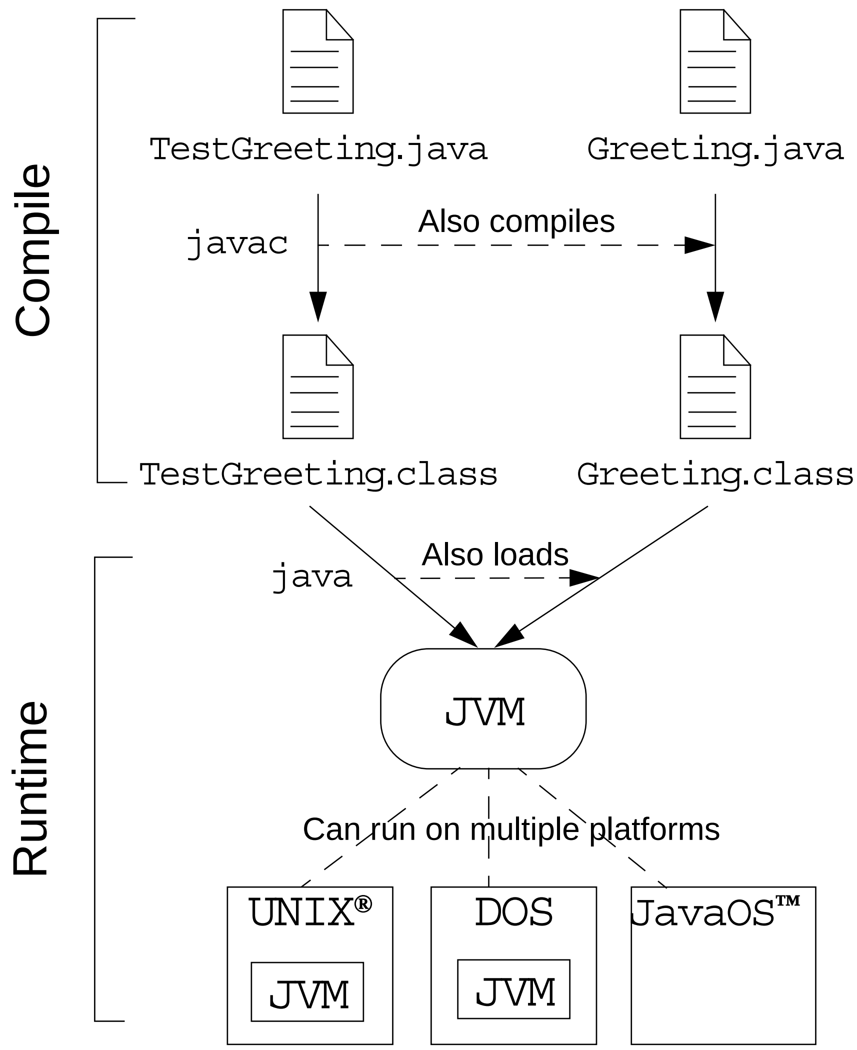
Compile-Time Errors

- `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 Technology Runtime 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
- 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 / J.F.C./Swing	Jini™	JavaBeans™	JDBC™

Object-Oriented Languages (1980s–Up)					
SELF	Smalltalk	Common Lisp Object System	Eiffel	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	Microsoft Windows

Machine Code (Late 1940s–Up)

The Analysis and Design Phase

- Analysis describes *what* the system needs to do:

  Modeling the real-world, including 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:

```
<modifier>* class <class_name> {
  <attribute_declaration>*
  <constructor_declaration>*
  <method_declaration>*
}
```

- Example:

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

Declaring Attributes

- Basic syntax of an attribute:

```
<modifier>* <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:

```
<modifier>* <return_type> <name> ( <argument>* ) {
  <statement>*
}
```

- Examples:

```
public class Dog {
  private int weight;
  public int getWeight() {
    return weight;
  }
  public void setWeight(int newWeight) {
    if ( newWeight > 0 ) {
      weight = newWeight;
    }
  }
}
```

Accessing Object Members

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

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

Information Hiding

The problem:

MyDate
+day : int +month : int +year : int

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:

```
MyDate
-day : int
-month : int
-year : int
+getDay() : int
+getMonth() : int
+getYear() : int
+setDay(int) : boolean
+setMonth(int) : boolean
+setYear(int) : boolean
```

Verify days in month

Client code must use setters and 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 : long
+getDay() : int +getMonth() : int +getYear() : int +setDay(int) : boolean +setMonth(int) : boolean +setYear(int) : boolean -isDayValid(int) : boolean

Declaring Constructors

- Basic syntax of a constructor:

```
[<modifier>] <class_name> ( <argument>* ) {
  <statement>*
}
```

- Example:

```
public class Dog {

  private int weight;

  public Dog() {
     weight = 42;
  }
}
```

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
- The default 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 is:

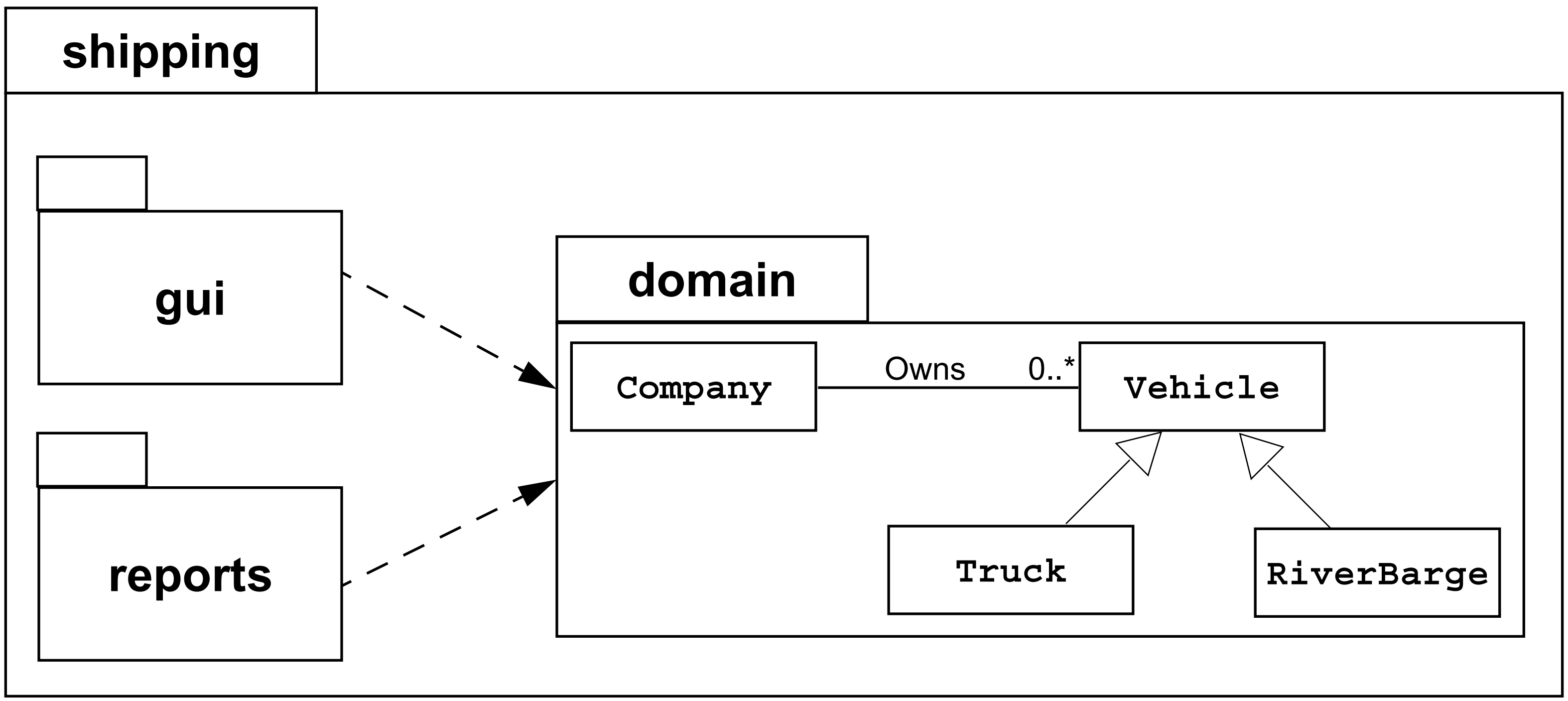
```
[<package_declaration>]
<import_declaration>*
<class_declaration>+
```

- For example, the `VehicleCapacityReport.java` file is:

```
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 is:

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

- Examples of the statement are:

```
package shipping.gui.reportscreens;
```

- 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 is placed into the default package.
- Package names must be hierarchical and separated by dots.

The `import` Statement

- Basic syntax of the `import` statement is:

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

- Examples of the statement are:

```
import java.util.List;
import java.io.*;
import shipping.gui.reportscreens.*;
```

- The import statement does the following:
  - Precedes all class declarations
  - Tells the compiler where to find classes

Directory Layout and Packages

- Packages are stored in the directory tree containing the package name.
- An example is the shipping application packages.

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

Development

```
JavaProjects/
  └── ShippingPrj/
        ├── src/
        │     └── shipping/
        │           ├── domain/
        │           ├── gui/
        │           └── reports/
        ├── docs/
        └── classes/
              └── shipping/
                    ├── domain/
                    ├── gui/
                    └── reports/
```

Compiling Using the `-d` Option

```
cd JavaProjects/ShippingPrj/src
javac -d ../classes shipping/domain/*.java
```

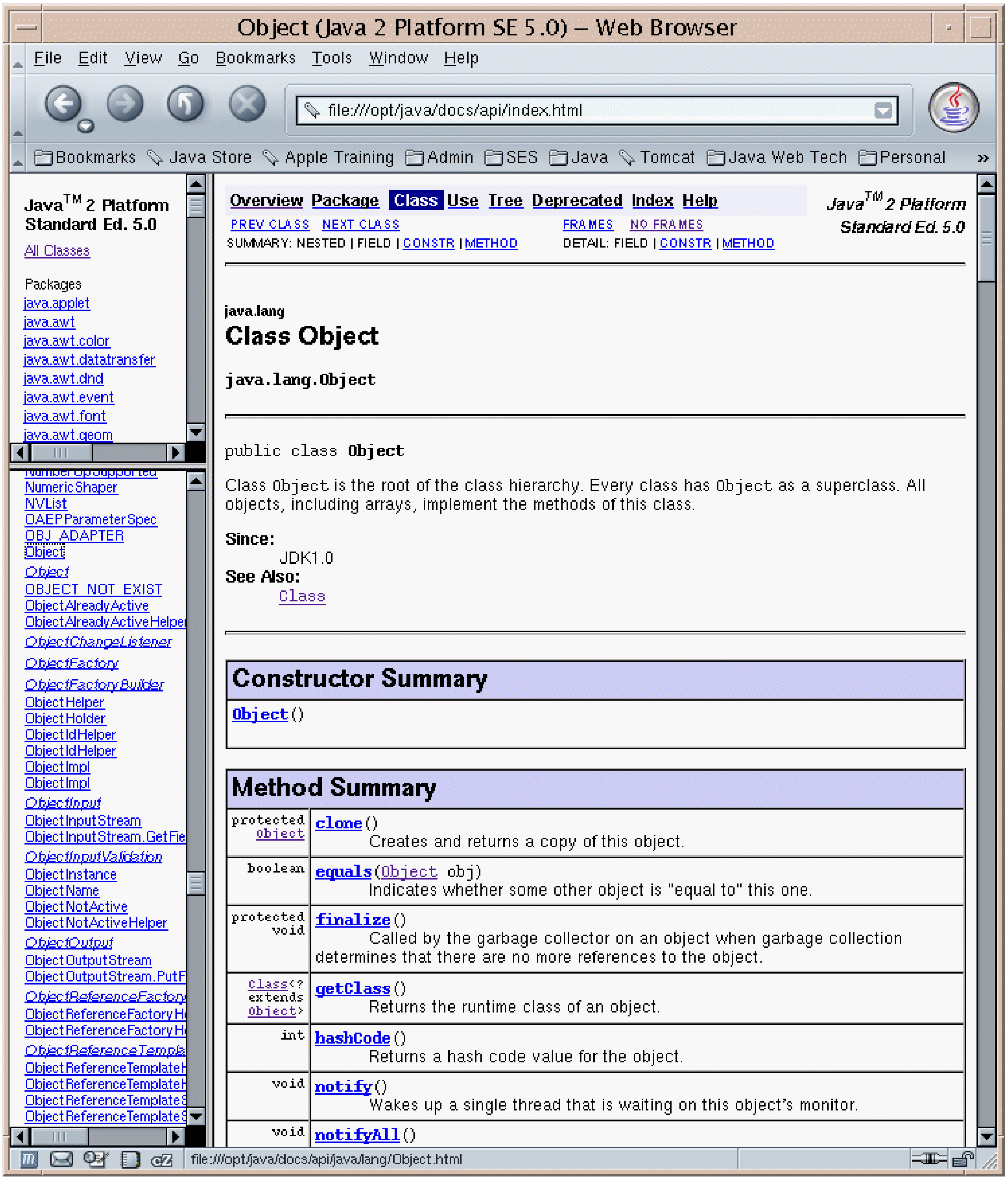
Terminology Recap

- 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*, and *data field*
- Method – A behavioral element of an object;
  also known as *algorithm*, *function*, and *procedure*
- Constructor – A *method-like* construct used to initialize a new object
- Package – A grouping of classes and sub-packages

Using the Java Technology 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.

Java Technology API Documentation



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
 *  can also span one or more lines
 */
```

Semicolons, Blocks, and White Space

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

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

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

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

Semicolons, Blocks, and White Space

- A *class* definition uses a special block:

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

- You can nest block statements.

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

Semicolons, Blocks, and White Space

- Any amount of *white space* is permitted in a Java program.

  For example:

  ```
  {int x;x=23*54;}
  ```

  is equivalent to:

  ```
  {
    int x;

    x = 23 * 54;
  }
  ```

Identifiers

Identifiers have the following characteristics:

- 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 Programming Language Keywords

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

Reserved literal words: `null`, `true`, and `false`

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` primitive has the following characteristics:

- 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`

The textual `char` primitive has the following characteristics:

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

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

Textual – `String`

The textual `String` type has the following characteristics:

- 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`

The integral primitives have the following characteristics:

- Integral primates use 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`

The floating point primitives have the following characteristics:

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

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

Floating Point – `float` and `double`

- Literals have a default type of `double`.
- Floating-point data types have the following sizes:

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

Variables, Declarations, and Assignments

```java
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

- In Java technology, beyond primitive types all others are reference types.
- A *reference variable* contains a *handle* to an object.
- For 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 String toString() { ... }
}
```

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

Constructing and Initializing Objects

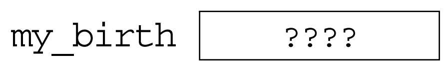
- Calling `new` *`Xyz`*`()` performs the following actions:
  a. Memory is allocated for the object.
  b. Explicit attribute initialization is performed.
  c. A constructor is executed.
  d. The object reference is returned by the `new` operator.
- The reference to the object is assigned to a variable.
- An example is:

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

Memory Allocation and Layout

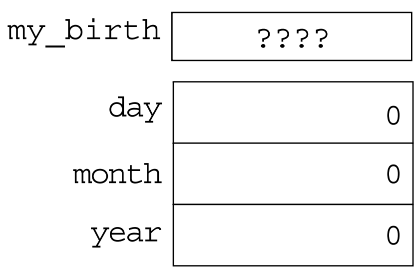
- A declaration allocates storage only for a reference:

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



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

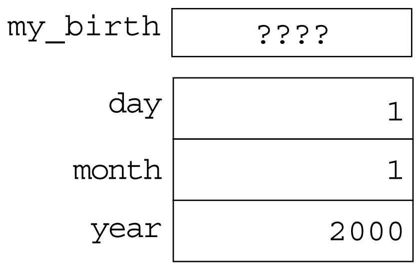
```
MyDate my_birth = new MyDate(22, 7, 1964);
```



Explicit Attribute Initialization

- Initialize the attributes as follows:

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

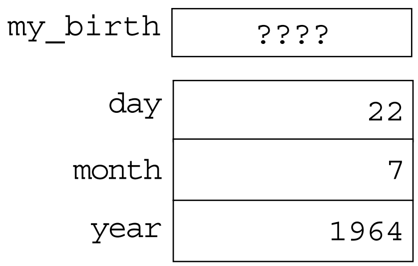


- The default values are taken from the attribute declaration in the class.

Executing the Constructor

- Execute the matching constructor as follows:

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

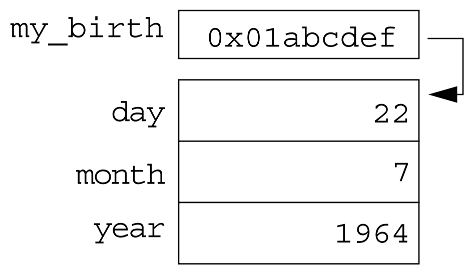


- In the case of an overloaded constructor, the first constructor can call another.

Assigning a Variable

- Assign the newly created object to the reference variable as follows:

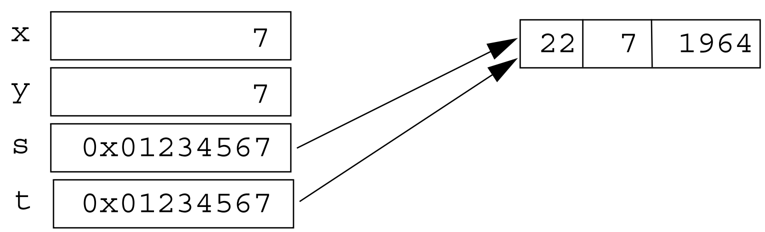
```
MyDate my_birth = new MyDate(22, 7, 1964);
```



Assigning References

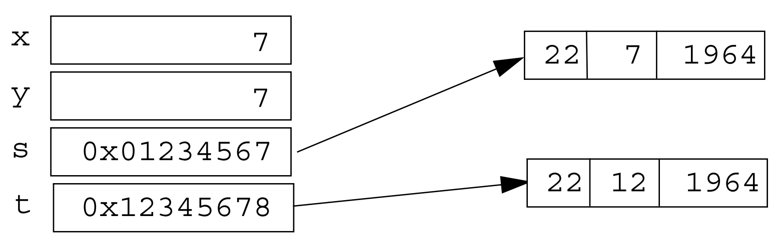
- Two variables refer to a single object:

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



- Reassignment makes two variables point to two objects:

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



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);
  }
```

Pass-by-Value

```

  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);
```

The result of this output is:

```
Int value is: 11
```

Pass-by-Value

```
        // Assign the date
        date = new MyDate(22, 7, 1964);
        // Try to change it
        changeObjectRef(date);
        // What is the current value?
        System.out.println("MyDate: " + date);
```

The result of this output is:

```
MyDate: 22-7-1964
```

Pass-by-Value

```
    // Now change the day attribute
    // through the object reference
    changeObjectAttr(date);
    // What is the current value?
    System.out.println("MyDate: " + date);
  }
}
```

The result of this output is:

```
MyDate: 4-7-1964
```

The `this` Reference

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

- To resolve ambiguity between instance variables and parameters
- 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;
  }
```

The `this` Reference

```
  public MyDate addDays(int moreDays) {
    MyDate newDate = new MyDate(this);
    newDate.day = newDate.day + moreDays;
    // Not Yet Implemented: wrap around code...
    return newDate;
  }
  public String toString() {
    return "" + day + "-" + month + "-" + year;
  }
}
```

The `this` Reference

```
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);

    System.out.println(the_next_week);
  }
}
```

Java Programming Language Coding Conventions

- Packages:

  `com.example.domain;`

- Classes, interfaces, and `enum` types:

  `SavingsAccount`

- Methods:

  `getAccount()`

- Variables:

  `currentCustomer`

- Constants:

  `HEAD_COUNT`

Java Programming Language Coding Conventions

- Control structures:

```
if ( condition ) {
  statement1;
} else {
  statement2;
}
```

- Spacing:
  - Use one statement per line.
  - Use two or four spaces for indentation.
- Comments:
  - Use `//` to comment inline code.
  - Use `/** documentation */` for class members.

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 labelled 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:

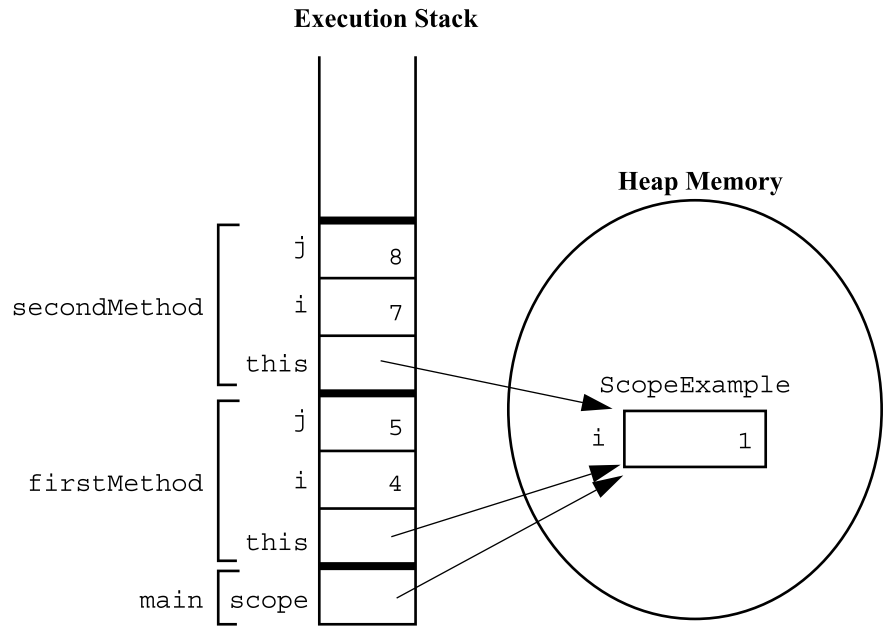
- Variables that are defined inside a method and are called *local*, *automatic*, *temporary*, or *stack* variables
- Variables that are created when the method is executed are destroyed when the method is exited

Variable initialization comprises the following:

- Local variables require explicit initialization.
- Instance variables are initialized automatically.

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`

Initialization Before Use Principle

The compiler will verify that local variables have been initialized before used.

```
3     public void doComputation() {
4       int x = (int)(Math.random() * 100);
5       int y;
6       int z;
7       if (x > 50) {
8         y = 9;
9       }
10      z = y + x;  // Possible use before initialization
11    }
```

```
javac TestInitBeforeUse.java
TestInitBeforeUse.java:10: variable y might not have been initialized
    z = y + x;  // Possible use before initialization
        ^
1 error
```

Operator Precedence

<table>
<tr><th>Operators</th><th>Associative</th></tr>
<tr><td>++ -- + unary - unary ~ ! (<data_type>)</td><td>R to L</td></tr>
<tr><td>* / %</td><td>L to R</td></tr>
<tr><td>+ -</td><td>L to R</td></tr>
<tr><td><< >> >>></td><td>L to R</td></tr>
<tr><td>< > <= >= instanceof</td><td>L to R</td></tr>
<tr><td>== !=</td><td>L to R</td></tr>
<tr><td>&</td><td>L to R</td></tr>
<tr><td>^</td><td>L to R</td></tr>
<tr><td>|</td><td>L to R</td></tr>
<tr><td>&&</td><td>L to R</td></tr>
<tr><td>||</td><td>L to R</td></tr>
<tr><td><boolean_expr> ? <expr1> : <expr2></td><td>R to L</td></tr>
<tr><td>= *= /= %= += -= <<= >>= >>>= &= ^= |=</td><td>R to L</td></tr>
</table>

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 = reservation.getDepartureDate();
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 include:

~	0	1	0	0	1	1	1	1
	1	0	1	1	0	0	0	0

	0	0	1	0	1	1	0	1
&	0	1	0	0	1	1	1	1
	0	0	0	0	1	1	0	1

	0	0	1	0	1	1	0	1
^	0	1	0	0	1	1	1	1
	0	1	1	0	0	0	1	0

	0	0	1	0	1	1	0	1
|	0	1	0	0	1	1	1	1
	0	1	1	0	1	1	1	1

Right-Shift Operators >> and >>>

- *Arithmetic* or *signed* right shift (>>) operator:
  - Examples are:

```
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.
- *Logical* or *unsigned right-shift* (>>>) operator:
  - This operator is used for bit patterns.
  - The sign bit is not copied during the shift.

Left-Shift Operator <<

- Left-shift (<<) operator works as follows:

  128 << 1 returns 128 * 2^{1} = 256
  16 << 2 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 works as follows:
  - 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 promoted automatically 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
```

Simple `if, else` Statements

The `if` statement syntax:

```
if ( <boolean_expression> )
  <statement_or_block>
```

Example:

```
if ( x < 10 )
  System.out.println("Are you finished yet?");
```

or (*recommended*):

```
if ( x < 10 ) {
  System.out.println("Are you finished yet?");
}
```

Complex `if, else` Statements

The `if-else` statement syntax:

```
if ( <boolean_expression> )
  <statement_or_block>
else
  <statement_or_block>
```

Example:

```
if ( x < 10 ) {
  System.out.println("Are you finished yet?");
} else {
  System.out.println("Keep working...");
}
```

Complex `if, else` Statements

The `if-else-if` statement syntax:

```
if ( <boolean_expression> )
  <statement_or_block>
else if ( <boolean_expression> )
  <statement_or_block>
```

Example:

```
int 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.");
}
```

Switch Statements

The `switch` statement syntax:

```
switch ( <expression> ) {
  case <constant1>:
    <statement_or_block>*
    [break;]
  case <constant2>:
    <statement_or_block>*
    [break;]
  default:
    <statement_or_block>*
    [break;]
}
```

Switch Statements

A `switch` statement example:

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

Switch Statements

This `switch` statement is equivalent to the previous example:

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

Without the `break` statements, the execution falls through each subsequent `case` clause.

Looping Statements

The `for` loop:

```
for ( <init_expr>; <test_expr>; <alter_expr> )
  <statement_or_block>
```

Example:

```
for ( int i = 0; i < 10; i++ )
  System.out.println(i + " squared is " + (i*i));
```

or (*recommended*):

```
for ( int i = 0; i < 10; i++ ) {
  System.out.println(i + " squared is " + (i*i));
}
```

Looping Statements

The `while` loop:

```
while ( <test_expr> )
  <statement_or_block>
```

Example:

```
int i = 0;
while ( i < 10 ) {
  System.out.println(i + " squared is " + (i*i));
  i++;
}
```

Looping Statements

The do/while loop:

```
do
  <statement_or_block>
while ( <test_expr> );
```

Example:

```
int i = 0;
do {
  System.out.println(i + " squared is " + (i*i));
  i++;
} while ( i < 10 );
```

Special Loop Flow Control

- The **`break`** ***`[<label>]`*** **`;`** command
- The **`continue`** ***`[<label>]`*** **`;`** command
- The ***`<label>`*** **`:`** ***`<statement>`*** command, where ***`<statement>`*** should be a loop

The `break` Statement

```
do {
  statement;
  if ( condition ) {
    break;
  }
  statement;
} while ( test_expr );
```

The `continue` Statement

```
do {
  statement;
  if ( condition ) {
    continue;
  }
  statement;
} while ( test_expr );
```

Using `break` Statements with Labels

```
outer:
  do {
    statement1;
    do {
      statement2;
      if ( condition ) {
        break outer;
      }
      statement3;
    } while ( test_expr );
    statement4;
  } while ( test_expr );
```

Using `continue` Statements with Labels

```
test:
  do {
    statement1;
    do {
      statement2;
      if ( condition ) {
        continue test;
      }
      statement3;
    } while ( test_expr );
    statement4;
  } while ( test_expr );
```

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 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`.

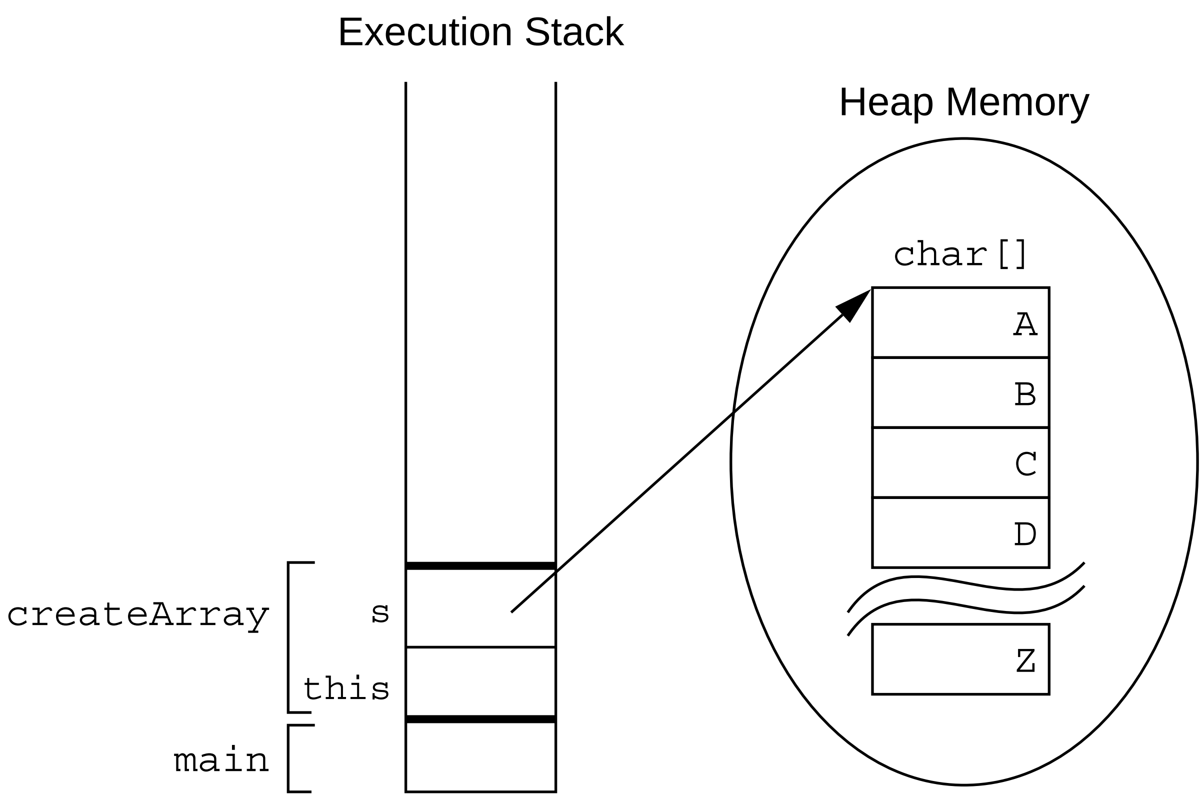
Creating Arrays

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 an Array of Character Primitives

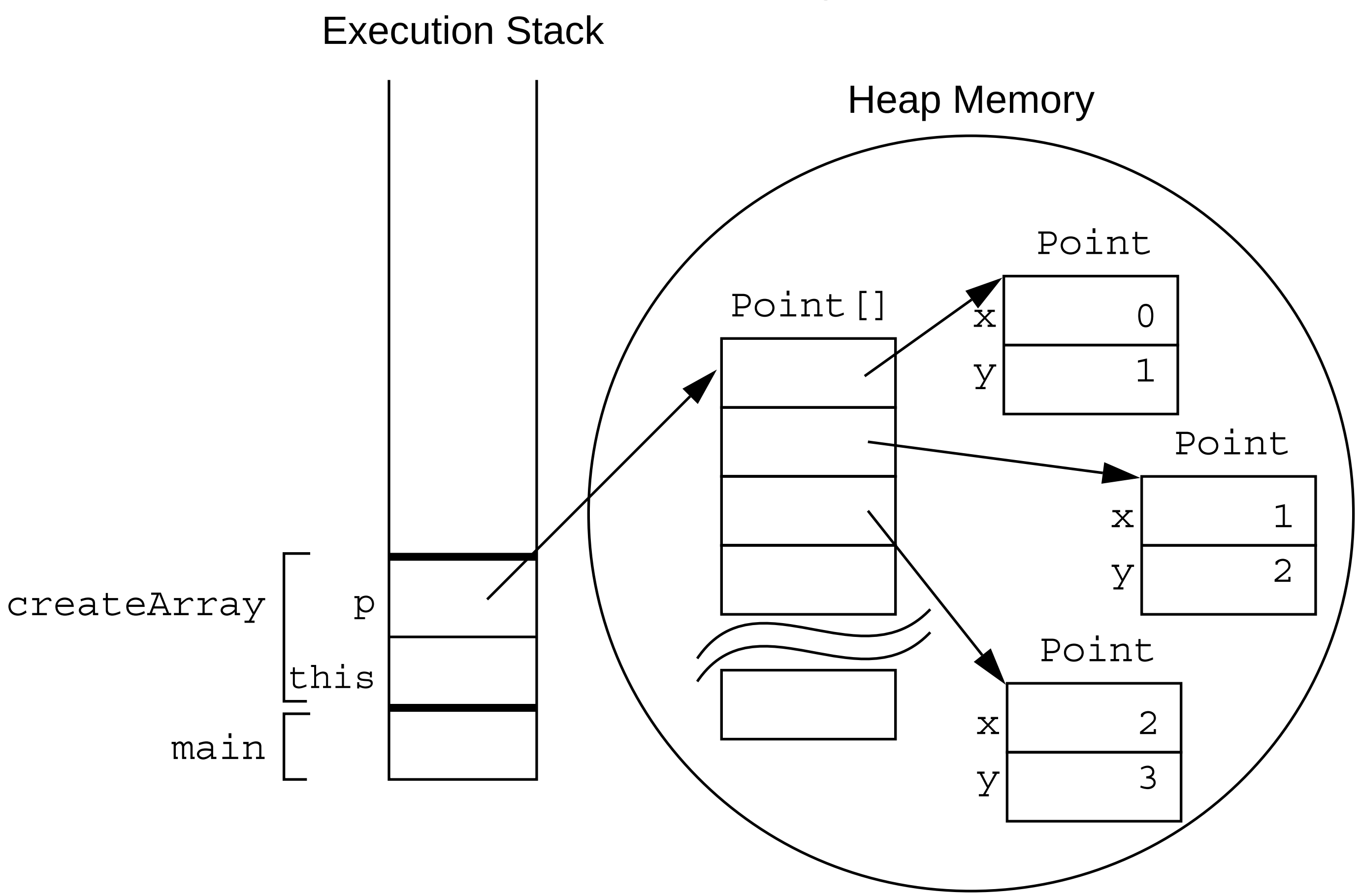


Creating Reference Arrays

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;
}
```

Creating an Array of Character Primitives With `Point` Objects



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:

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

Using the Enhanced `for` Loop

Java 2 Platform, Standard Edition (J2SE™) version 5.0 introduced an enhanced `for` loop for iterating over arrays:

```
public void printElements(int[] list) {
  for ( int element : list ) {
    System.out.println(element);
  }
}
```

The `for` loop can be read as *for each* `element` *in* `list` *do*.

Array Resizing

- You cannot resize an array.
- You can use the same reference variable to refer to an entirely new array, such as:

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

Copying Arrays

The `System.arraycopy()` method to copy arrays is:

```
//original array
int[] myArray = { 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 myArray array to the hold
// array, starting with the 0th index
System.arraycopy(myArray, 0, hold, 0, myArray.length);
```

Module 6

Class Design

Objectives

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

Relevance

How does the Java programming language support object inheritance?

Subclassing

The `Employee` class is shown here.

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

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

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

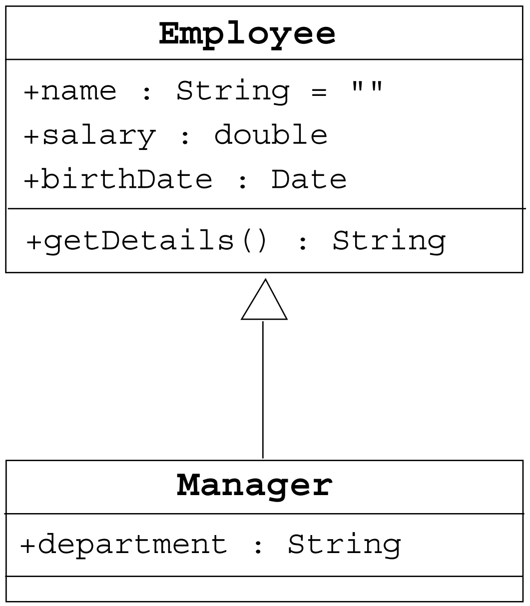
Subclassing

The `Manager` class is shown here.

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() {...}
}
```

Class Diagrams for `Employee` and `Manager` Using Inheritance




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

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

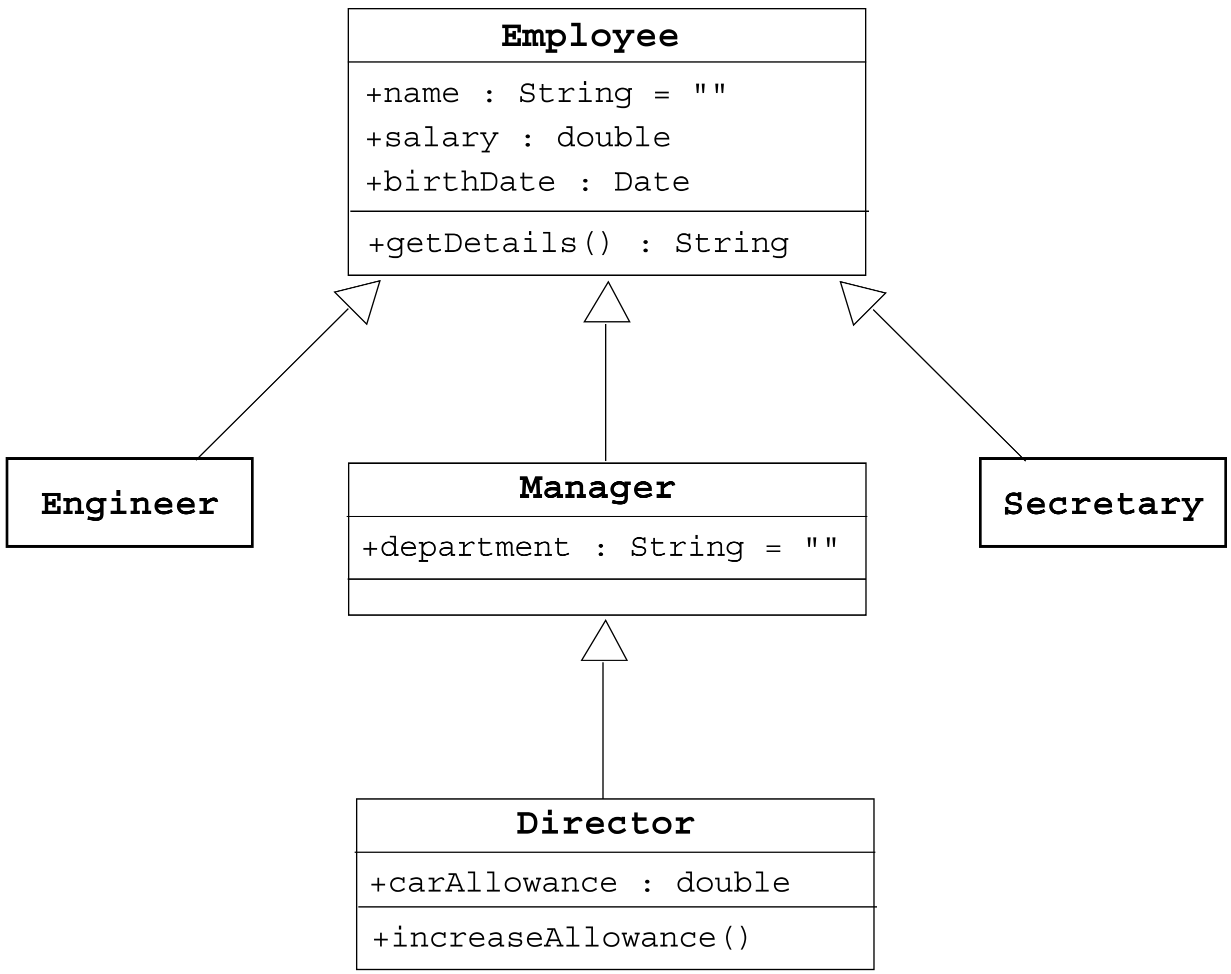
```
public class Manager extends Employee {
  public String department;
}
```

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 is as follows:

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

Single Inheritance



Access Control

Access modifiers on class member declarations are listed here.

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[1]
  - Argument list

1. In J2SE version 5, the return type can be a subclass of the overridden return type.

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;
  }
}
```

Overridden Methods Cannot Be Less Accessible

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

```java
public class Child extends Parent {
  private void doSomething() {} // illegal
}
```

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

Invoking Overridden Methods

A subclass method may invoke a superclass method using the `super` keyword:

- The keyword `super` is used in a class to refer to its superclass.
- The keyword `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.

Invoking Overridden Methods

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

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

Virtual Method Invocation

- Virtual method invocation is performed as follows:

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

- Compile-time type and runtime type invocations have the following characteristics:
  - The method name must be a member of the declared variable type; in this case `Employee` has a method called `getDetails`.
  - The method implementation used is based on the runtime object's type; in this case the `Manager` class has an implementation of the `getDetails` method.

Heterogeneous Collections

- Collections of objects with the same class type are called *homogeneous* collections. For example:

```
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. For example:

```
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`, the following is valid:

```
public class TaxService {
  public TaxRate findTaxRate(Employee e) {
    // calculate the employee's tax rate
  }
}

// Meanwhile, elsewhere in the application class
TaxService taxSvc = new TaxService();
Manager m = new Manager();
TaxRate t = taxSvc.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

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

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 *upward* in the 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 Methods

- Use overloading 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.

Methods Using Variable Arguments

- Methods using *variable arguments* permit multiple number of arguments in methods.

  For example:

```
public class Statistics {
  public float average(int... nums) {
    int sum = 0;
    for ( int x : nums ) {
      sum += x;
    }
    return ((float) sum) / nums.length;
  }
}
```

- The *vararg* parameter is treated as an array. For example:

```
float gradePointAverage = stats.average(4, 3, 4);
float averageAge = stats.average(24, 32, 27, 18);
```

Overloading Constructors

- As with methods, constructors can be overloaded.

  An example is:

```
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);
  }
  // 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 Manager extends Employee {
  private String department;

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

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 Examples

```
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 Examples

```
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
```

Constructor and Initialization Examples

```
    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 implies `extends Object`. For example:

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

is equivalent to:

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

- Two important methods are:
  - `equals`
  - `toString`

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.

An `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;
  }
```

An `equals` Example

```

    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);
    }
  }
```

An `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");
    }
```

An `equals` Example

```
    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");
    }
  }
}
```

This example generates the following output:

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

The `toString` Method

The `toString` method has the following characteristics:

- This method converts an object to a `String`.
- Use this method during string concatenation.
- Override this method to provide information about a user-defined object in readable format.
- Use the wrapper class’s `toString` static method to convert primitive types to a `String`.

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

An example of a wrapper class is:

```
int pInt = 420;
Integer wInt = new Integer(pInt); // this is called boxing
int p2 = wInt.intValue(); // this is called unboxing
```

Other methods are:

```
int x = Integer.valueOf(str).intValue();
int x = Integer.parseInt(str);
```

Autoboxing of Primitive Types

Autoboxing has the following description:

- Conversion of primitive types to the object equivalent
- Wrapper classes not always needed
- Example:

```
int pInt = 420;
Integer wInt = pInt; // this is called autoboxing
int p2 = wInt; // this is called autounboxing
```

- Language feature used most often when dealing with collections
- Wrapped primitives also usable in arithmetic expressions
- Performance loss when using autoboxing

Module 7

Advanced Class Features

Objectives

- Create static variables, methods, and initializers
- Create final classes, methods, and variables
- Create and use enumerated types
- Use the static import statement
- Create abstract classes and methods
- Create and use an interface

Relevance

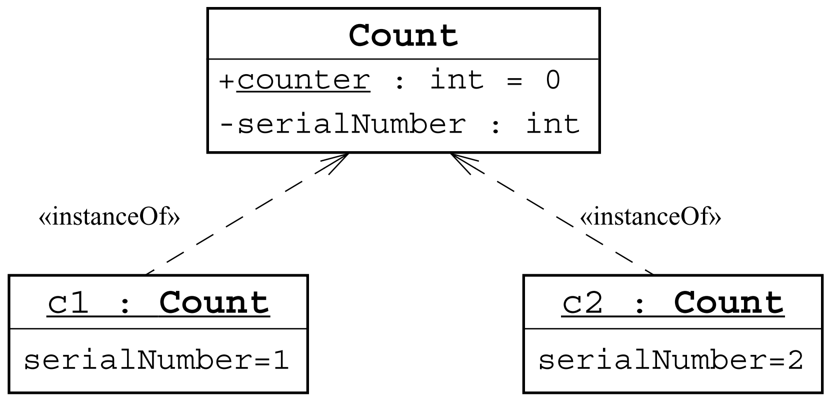
- How can you create a constant?
- How can you declare data that is shared by all instances of a given class?
- How can you keep a class or method from being subclassed or overridden?

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

Class attributes are shared among all instances of a class:




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

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

Class Attributes

If the static member is `public`:

```
public class Count1 {
  private int serialNumber;
  public static int counter = 0;
  public Count1() {
    counter++;
    serialNumber = counter;
  }
}
```

it can be accessed from outside the class without an instance:

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

Class Methods

You can create `static` methods:

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

  public static int getTotalCount() {
    return counter;
  }

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

Class Methods

You can invoke `static` methods without any instance of the class to which it belongs:

```
public class TestCounter {
  public static void main(String[] args) {
    System.out.println("Number of counter is "
                       + Count2.getTotalCount());
    Count2 counter = new Count2();
    System.out.println("Number of counter is "
                       + Count2.getTotalCount());
  }
}
```

The output of the `TestCounter` program is:

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

Class Methods

Static methods cannot access instance variables:

```
public class Count3 {
  private int serialNumber;
  private static int counter = 0;

  public static int getSerialNumber() {
    return serialNumber;  // COMPILER ERROR!
  }
}
```

Static Initializers

- A class can contain code in a *static block* that does not exist within a method body.
- Static block code executes once only, when the class is loaded.
- Usually, a static block is 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 -DmyApp.Count4.counter=47 TestStaticInit
counter = 47
```

The `final` Keyword

- You cannot subclass a `final` class.
- You cannot override a `final` method.
- A `final` variable is a constant.
- You can set a `final` variable once only, but that assignment can occur independently of the declaration; this is called a *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 are static final variables.

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

Blank Final Variables

```
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

}
```

Old-Style Enumerated Type Idiom

Enumerated types are a common idiom in programming.

```
package cards.domain;

public class PlayingCard {

  // pseudo enumerated type
  public static final int SUIT_SPADES   = 0;
  public static final int SUIT_HEARTS   = 1;
  public static final int SUIT_CLUBS    = 2;
  public static final int SUIT_DIAMONDS = 3;

  private int suit;
  private int rank;

  public PlayingCard(int suit, int rank) {
    this.suit = suit;
    this.rank = rank;
  }
```

Old-Style Enumerated Type Idiom

```
public String getSuitName() {
    String name = “”;
    switch ( suit ) {
      case SUIT_SPADES:
        name = “Spades”;
        break;
      case SUIT_HEARTS:
        name = “Hearts”;
        break;
      case SUIT_CLUBS:
        name = “Clubs”;
        break;
      case SUIT_DIAMONDS:
        name = “Diamonds”;
        break;
      default:
        System.err.println(“Invalid suit.”);
    }
    return name;
  }
```

Old-Style Enumerated Type Idiom

Old-style idiom is not type-safe:

```
package cards.tests;

import cards.domain.PlayingCard;

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

    PlayingCard card1
      = new PlayingCard(PlayingCard.SUIT_SPADES, 2);
    System.out.println(“card1 is the “ + card1.getRank()
                       + “ of “ + card1.getSuitName());

    // You can create a playing card with a bogus suit.
    PlayingCard card2 = new PlayingCard(47, 2);
    System.out.println(“card2 is the “ + card2.getRank()
                       + “ of “ + card2.getSuitName());
  }
}
```

Old-Style Enumerated Type Idiom

This enumerated type idiom has several problems:

- Not type-safe
- No namespace
- Brittle character
- Uninformative printed values

The New Enumerated Type

Now you can create type-safe enumerated types:

```
package cards.domain;

public enum Suit {
  SPADES,
  HEARTS,
  CLUBS,
  DIAMONDS
}

```

The New Enumerated Type

Using enumerated types is easy:

```
package cards.domain;

public class PlayingCard {

  private Suit suit;
  private int rank;

  public PlayingCard(Suit suit, int rank) {
    this.suit = suit;
    this.rank = rank;
  }

  public Suit getSuit() {
    return suit;
  }
```

The New Enumerated Type

```
public String getSuitName() {
    String name = "";
    switch ( suit ) {
      case SPADES:
        name = "Spades";
        break;
      case HEARTS:
        name = "Hearts";
        break;
      case CLUBS:
        name = "Clubs";
        break;
      case DIAMONDS:
        name = "Diamonds";
        break;
      default:
      // No need for error checking as the Suit
      // enum is finite.
    }
    return name;
  }
```

The New Enumerated Type

Enumerated types are type-safe:

```
package cards.tests;

import cards.domain.PlayingCard;
import cards.domain.Suit;

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

    PlayingCard card1
      = new PlayingCard(Suit.SPADES, 2);
    System.out.println(“card1 is the “ + card1.getRank()
                       + “ of “ + card1.getSuitName());

    // PlayingCard card2 = new PlayingCard(47, 2);
    // This will not compile.
  }
}
```

Advanced Enumerated Types

Enumerated types can have attributes and methods:

```
package cards.domain;

public enum Suit {
  SPADES   ("Spades"),
  HEARTS   ("Hearts"),
  CLUBS    ("Clubs"),
  DIAMONDS ("Diamonds");

  private final String name;

  private Suit(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

Advanced Enumerated Types

Public methods on enumerated types are accessible:

```
package cards.tests;

import cards.domain.PlayingCard;
import cards.domain.Suit;

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

    PlayingCard card1
      = new PlayingCard(Suit.SPADES, 2);
    System.out.println(“card1 is the “ + card1.getRank()
                       + “ of “ + card1.getSuit().getName());

    // NewPlayingCard card2 = new NewPlayingCard(47, 2);
    // This will not compile.
  }
}
```

Static Imports

- A *static import* imports the static members from a class:

```
import static <pkg_list>.<class_name>.<member_name>;
OR
import static <pkg_list>.<class_name>.*;
```

- A static import imports members individually or collectively:

```
import static cards.domain.Suit.SPADES;
OR
import static cards.domain.Suit.*;
```

- There is no need to qualify the static constants:

```
PlayingCard card1 = new PlayingCard(SPADES, 2);
```

- *Use this feature sparingly.*

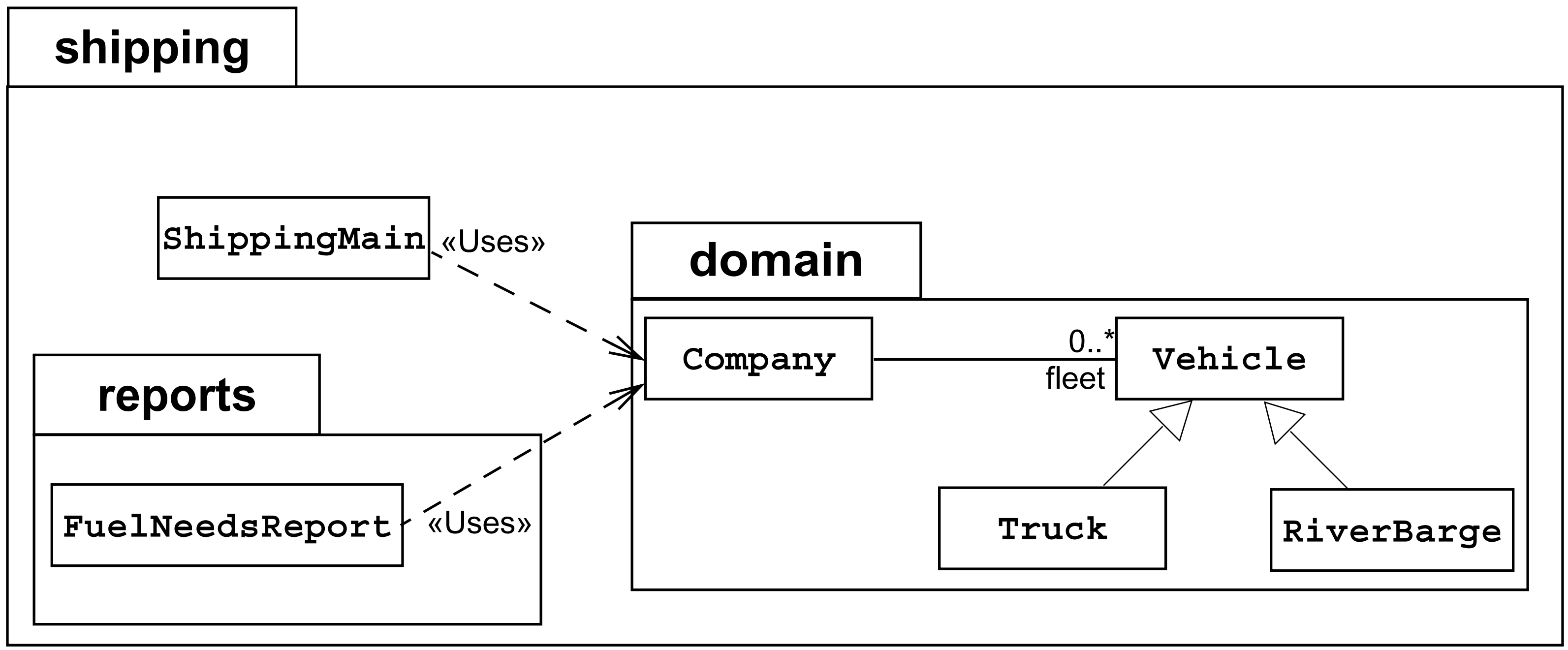
Static Imports

An example of a static import is:

```
package cards.tests;

import cards.domain.PlayingCard;
import static cards.domain.Suit.*;

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

    PlayingCard card1 = new PlayingCard(SPADES, 2);
    System.out.println(“card1 is the “ + card1.getRank()
                       + “ of “ + card1.getSuit().getName());

    // NewPlayingCard card2 = new NewPlayingCard(47, 2);
    // This will not compile.
  }
}
```

Abstract Classes

The design of the Shipping system looks like this:



Abstract Classes

Fleet initialization code is shown here:

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

    // 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(c);
    report.generateText(System.out);
  }
}
```

Abstract Classes

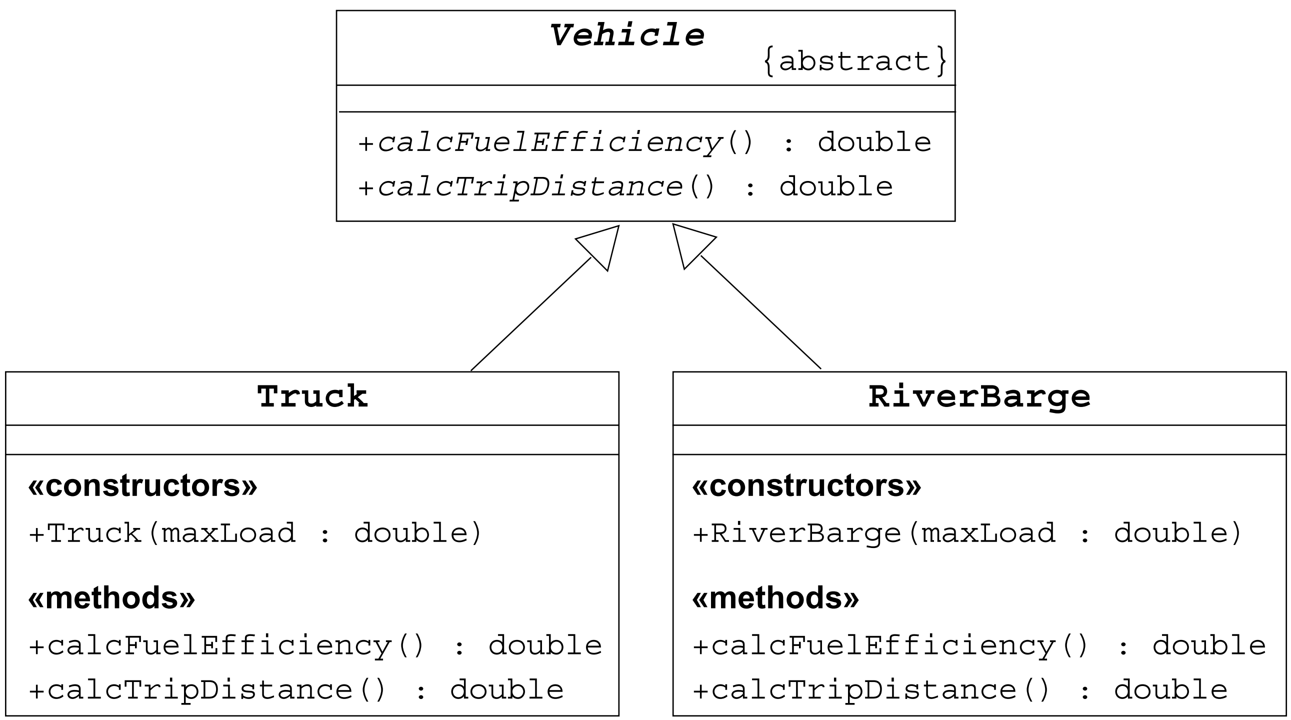
```
public class FuelNeedsReport {
  private Company company;

  public FuelNeedsReport(Company company) {
    this.company = company;
  }

  public void generateText(PrintStream output) {
    Vehicle1 v;
    double fuel;
    double total_fuel = 0.0;

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

```

Abstract Classes

```
      // 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.");
  }
}
```

The Solution

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



The Solution

The declaration of the `Vehicle` class is:

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

The `Truck` class must create an implementation:

```
public class Truck extends Vehicle {
  public Truck(double maxLoad) {...}
  public double calcFuelEfficiency() {
    /* calculate the fuel consumption of a truck at a given load */
  }
  public double calcTripDistance() {
    /* calculate the distance of this trip on highway */
  }
}
```

The Solution

Likewise, the `RiverBarge` class must create an implementation:

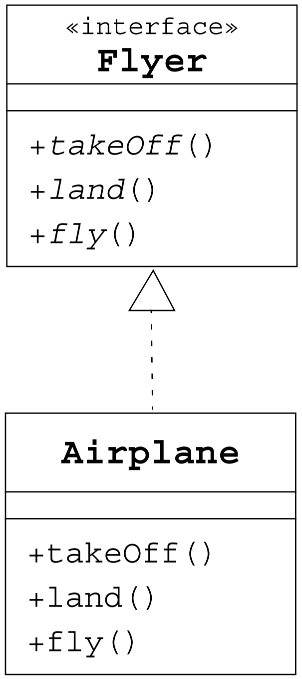
```
public class RiverBarge extends Vehicle {
  public RiverBarge(double maxLoad) {...}
  public double calcFuelEfficiency() {
    /* calculate the fuel efficiency of a river barge */
  }
  public double calcTripDistance() {
    /* calculate the distance of this trip along the river-ways */
  }
}
```

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 is as follows:

```
<modifier> class <name> [extends <superclass>]
           [implements <interface> [,<interface>]* ] {
  <member_declaration>*
}
```

The Flyer Example

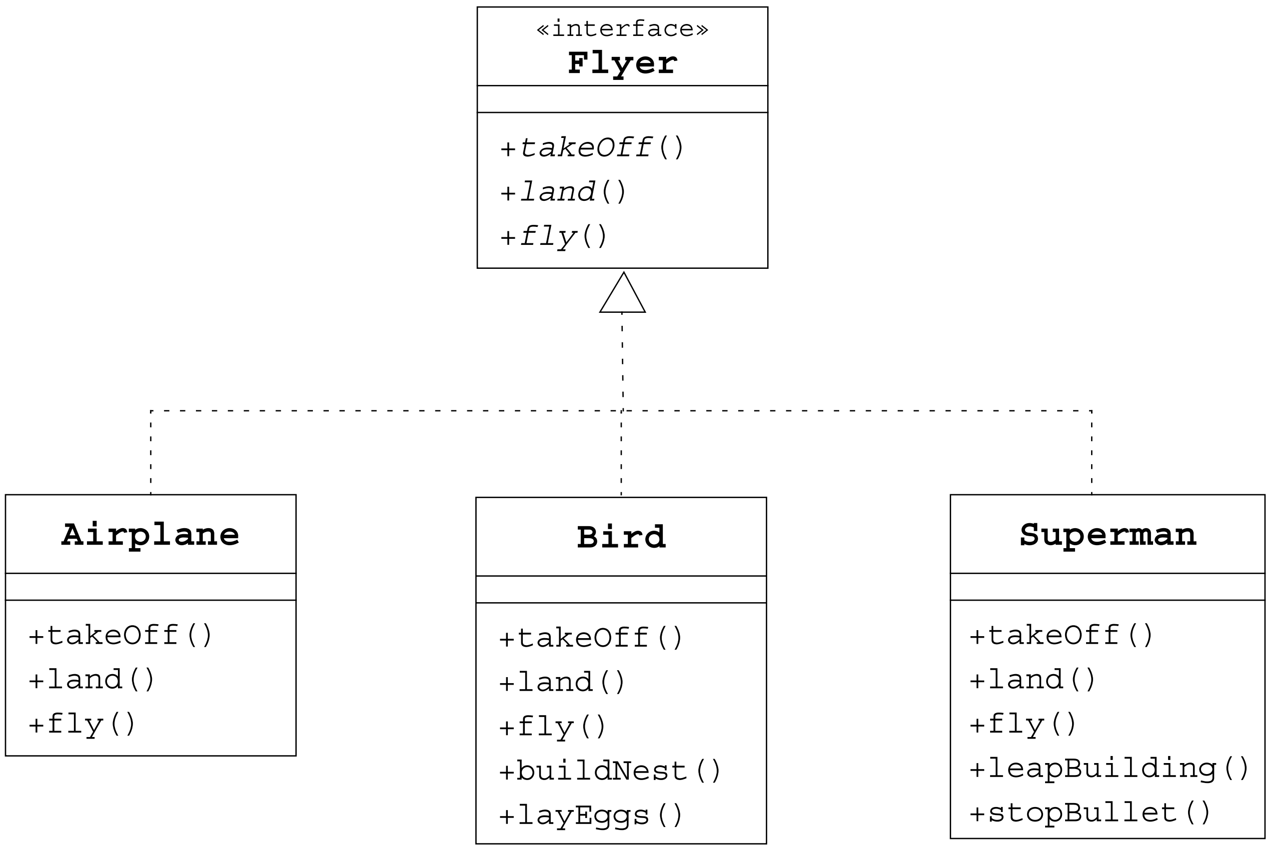



```
public interface Flyer {
  public void takeOff();
  public void land();
  public void fly();
}
```

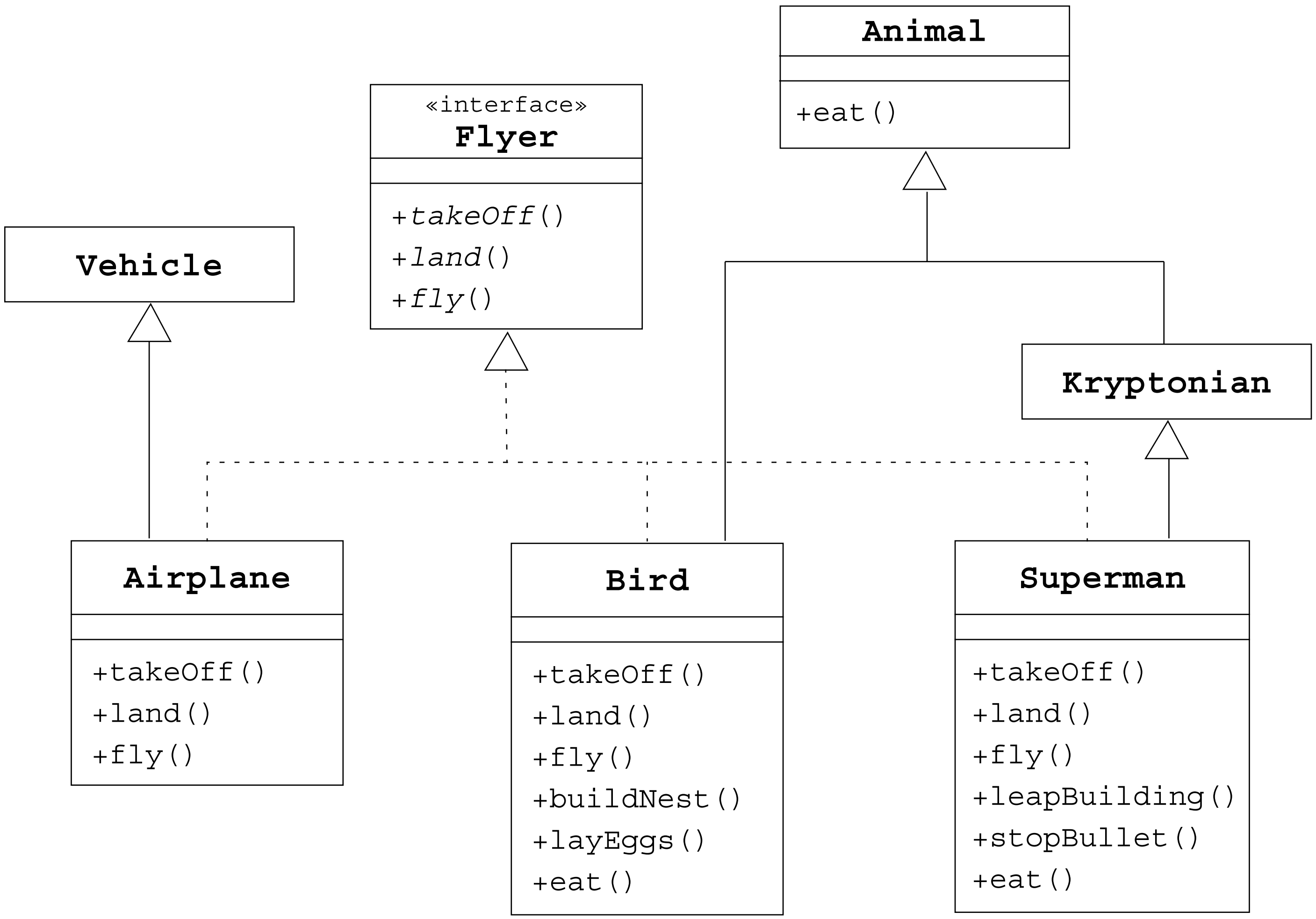
The Flyer Example

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

The Flyer Example



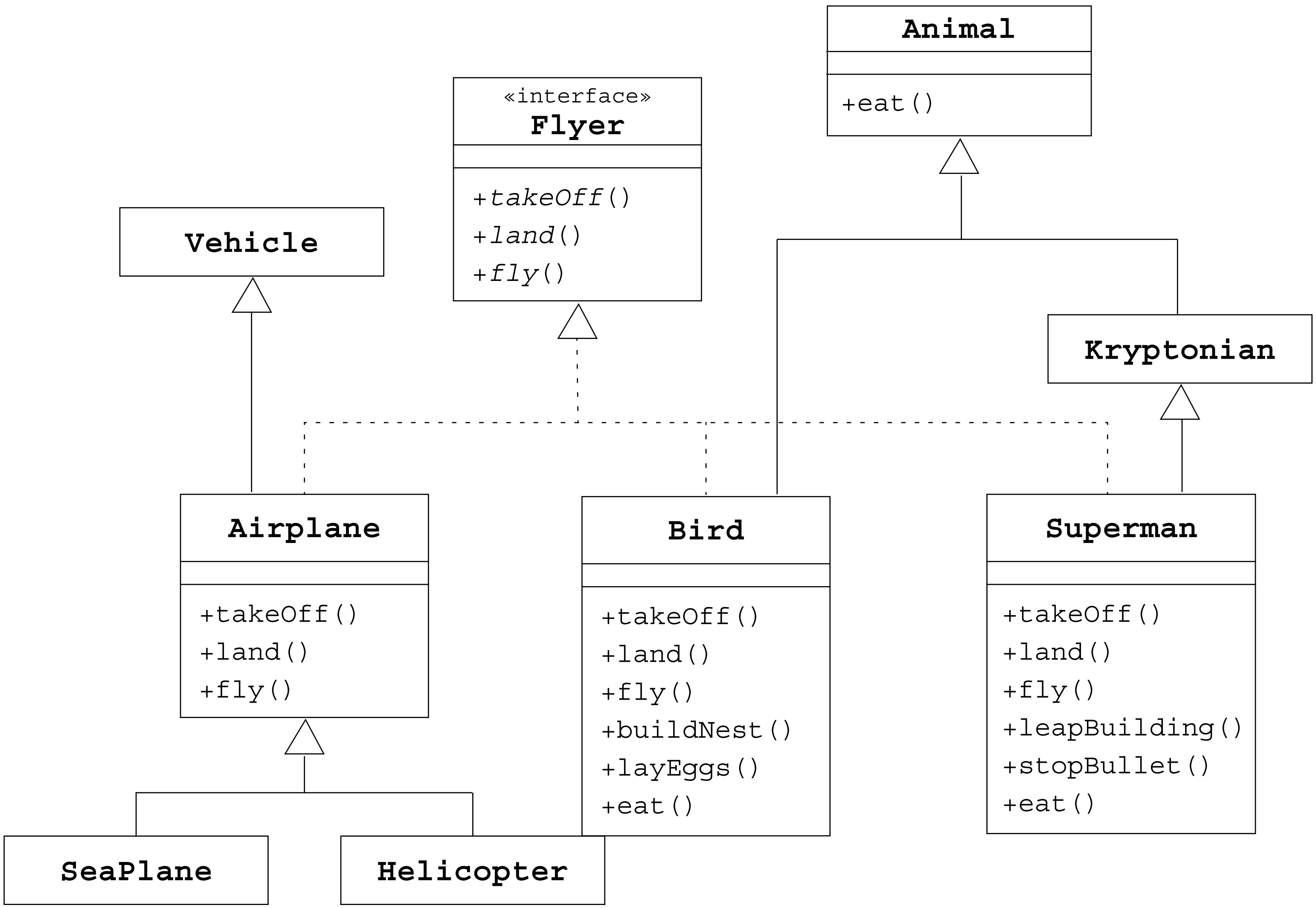
The Flyer Example



The Flyer 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 */ }
}
```

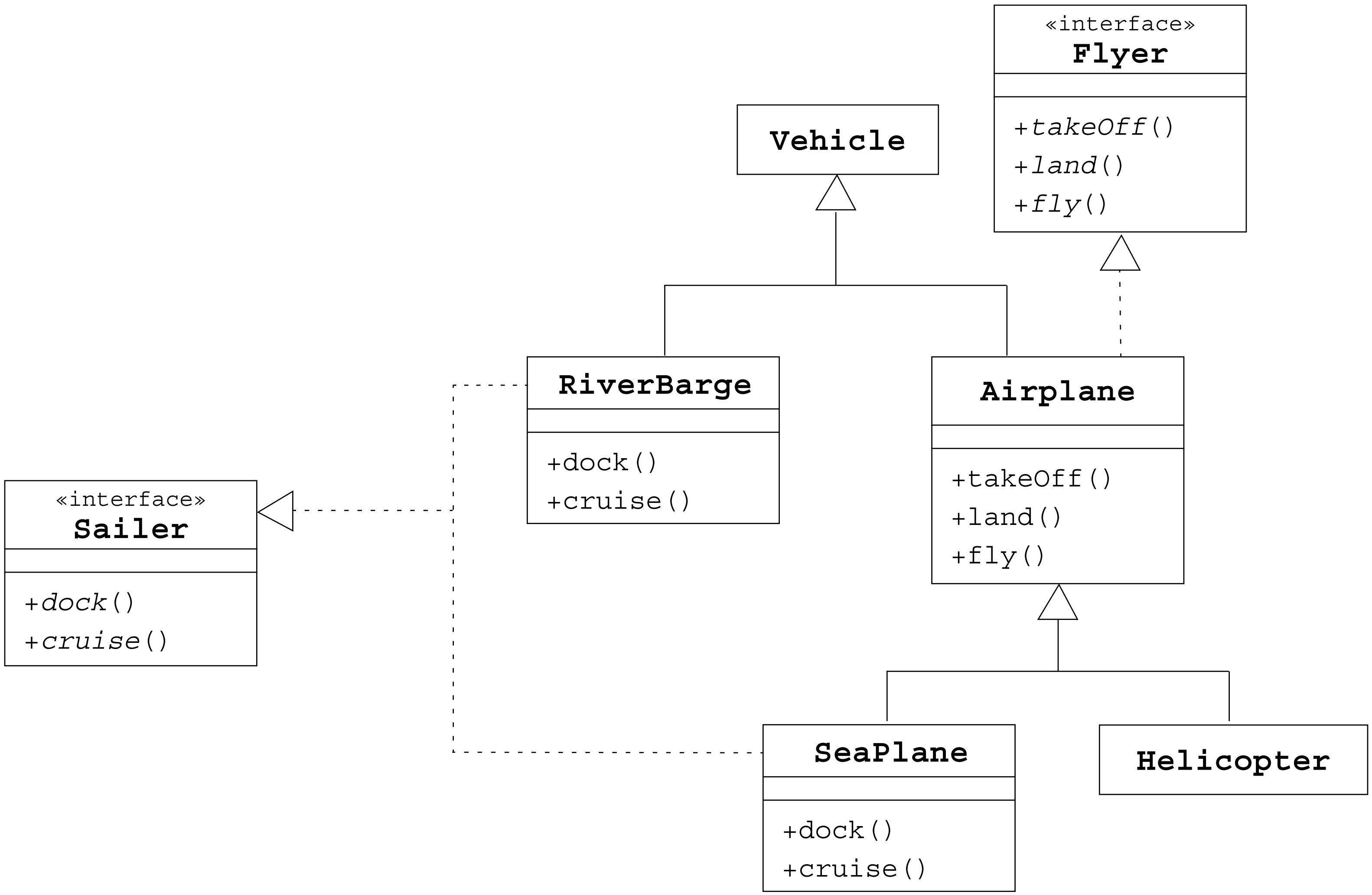
The Flyer Example



The Flyer Example

```
public class Airport {
  public static void main(String[] args) {
    Airport metropolisAirport = new Airport();
    Helicopter copter = new Helicopter();
    SeaPlane sPlane = new SeaPlane();

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

  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

Interface uses include the following:

- 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

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
- Enable 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 – Illegal argument, network failure, or file not found
- 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*.

  These are represented by the `Exception` class.

- Severe problems that normally are 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 documentation shows checked exceptions that can be thrown from a method.

Exception Example

```
public class AddArguments {
  public static void main(String args[]) {
    int sum = 0;
    for ( String arg : args ) {
      sum += Integer.parseInt(arg);
    }
    System.out.println("Sum = " + sum);
  }
}
```

```
java AddArguments 1 2 3 4
Sum = 10
```

```
java AddArguments 1 two 3.0 4
Exception in thread "main" java.lang.NumberFormatException: For input string: "two"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at java.lang.Integer.parseInt(Integer.java:497)
    at AddArguments.main(AddArguments.java:5)
```

The `try-catch` Statement

```
public class AddArguments2 {
  public static void main(String args[]) {
    try {
      int sum = 0;
      for ( String arg : args ) {
        sum += Integer.parseInt(arg);
      }
      System.out.println("Sum = " + sum);
    } catch (NumberFormatException nfe) {
      System.err.println("One of the command-line "
                         + "arguments is not an integer.");
    }
  }
}
```

```
java AddArguments2 1 two 3.0 4
One of the command-line arguments is not an integer.
```

The `try-catch` Statement

```
public class AddArguments3 {
  public static void main(String args[]) {
    int sum = 0;
    for ( String arg : args ) {
      try {
        sum += Integer.parseInt(arg);
      } catch (NumberFormatException nfe) {
        System.err.println("[" + arg + "] is not an integer"
                           + " and will not be included in the sum.");
      }
    }
    System.out.println("Sum = " + sum);
  }
}
```

```
java AddArguments3 1 two 3.0 4
[two] is not an integer and will not be included in the sum.
[3.0] is not an integer and will not be included in the sum.
Sum = 5
```

The `try-catch` Statement

A `try-catch` statement can use multiple catch clauses:

```
try {
  // code that might throw one or more exceptions

} catch (MyException e1) {
  // code to execute if a MyException exception is thrown

} catch (MyOtherException e2) {
  // code to execute if a MyOtherException exception is thrown

} catch (Exception e3) {
  // code to execute if any other exception is thrown
}
```

Call Stack Mechanism

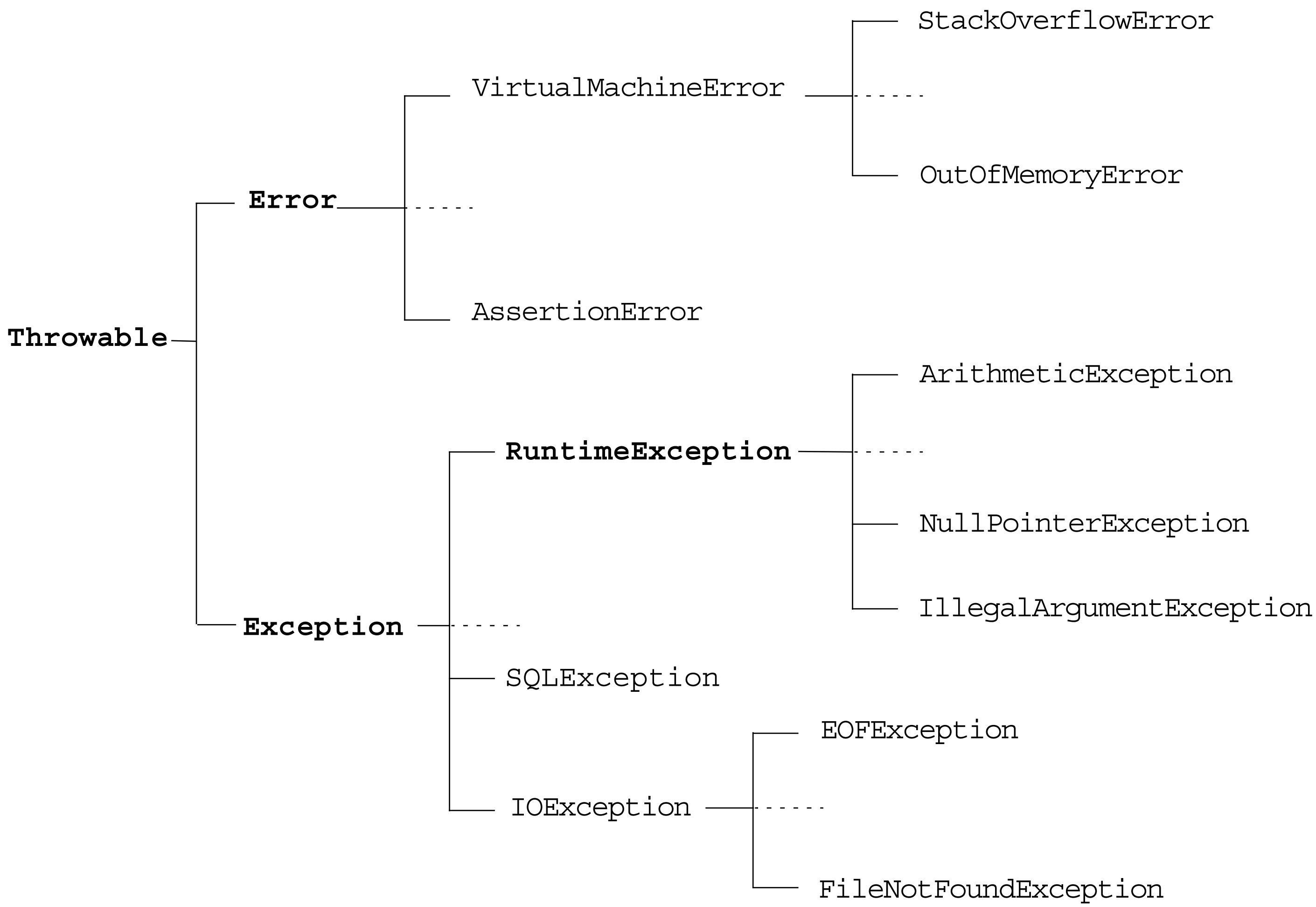
- If an exception is not handled in the current `try-catch` block, it is thrown to the caller of that method.
- If the exception gets back to the main method and is not handled there, the program is terminated abnormally.

The `finally` Clause

The `finally` clause defines a block of code that *always* executes.

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

Exception Categories



Common Exceptions

- `NullPointerException`
- `FileNotFoundException`
- `NumberFormatException`
- `ArithmeticException`
- `SecurityException`

The Handle or Declare Rule

Use the handle or declare rule as follows:

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

```
void trouble() throws IOException { ... }
void trouble() throws IOException, MyException { ... }
```

Other Principles

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

Method Overriding and Exceptions

The overriding method can throw:

- No exceptions
- One or more of the exceptions thrown by the overridden method
- One or more subclasses of the exceptions thrown by the overridden method

The overriding method cannot throw:

- Additional exceptions not thrown by the overridden method
- Superclasses of the exceptions thrown by the overridden method

Method Overriding and Exceptions

```
public class TestA {
  public void methodA() throws IOException {
    // do some file manipulation
  }
}
```

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

```
public class TestB2 extends TestA {
  public void methodA() throws Exception { // WRONG
    // do some file manipulation
  }
}
```

Creating Your Own Exceptions

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

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

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

Use the `getMessage` method, inherited from the `Exception` class, to get the reason for which the exception was made.

Handling a User-Defined Exception

A method can throw a user-defined, checked exception:

```
public void connectMe(String serverName)
       throws ServerTimedOutException {
  boolean successful;
  int portToConnect = 80;

  successful = open(serverName, portToConnect);

  if ( ! successful ) {
    throw new ServerTimedOutException("Could not connect",
                                       portToConnect);
  }
}
```

Handling a User-Defined Exception

Another method can use a `try-catch` block to capture user-defined exceptions:

```
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

- Syntax of an assertion is:

```
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.

Recommended Uses of Assertions

Use assertions to document and verify the assumptions and internal logic of a single method:

- Internal invariants
- Control flow invariants
- Postconditions and class invariants

Inappropriate Uses of Assertions

- Do not use assertions to check the parameters of a public method.
- Do not use methods in the assertion check that can cause side-effects.

Internal Invariants

The problem is:

```
if (x > 0) {
  // do this
} else {
  // do that
}
```

The solution is:

```
if (x > 0) {
  // do this
} else {
  assert ( x == 0 );
  // do that, unless x is negative
}
```

Control Flow Invariants

For example:

```
switch (suit) {
    case Suit.CLUBS: // ...
      break;
    case Suit.DIAMONDS: // ...
      break;
    case Suit.HEARTS: // ...
      break;
    case Suit.SPADES: // ...
      break;
    default: assert false : "Unknown playing card suit";
      break;
}
```

Postconditions and Class Invariants

For example:

```
public Object pop() {
  int size = this.getElementCount();
  if (size == 0) {
    throw new RuntimeException("Attempt to pop from empty stack");
  }

  Object result = /* code to retrieve the popped element */ ;

  // test the postcondition
  assert (this.getElementCount() == size - 1);

  return result;
}
```

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 disabled by default. Enable assertions with the following commands:

```
java -enableassertions MyProgram
```

or:

```
java -ea MyProgram
```

- Assertion checking can be controlled on class, package, and package hierarchy bases, see:
`docs/guide/language/assert.html`

Module 9

Collections and Generics Framework

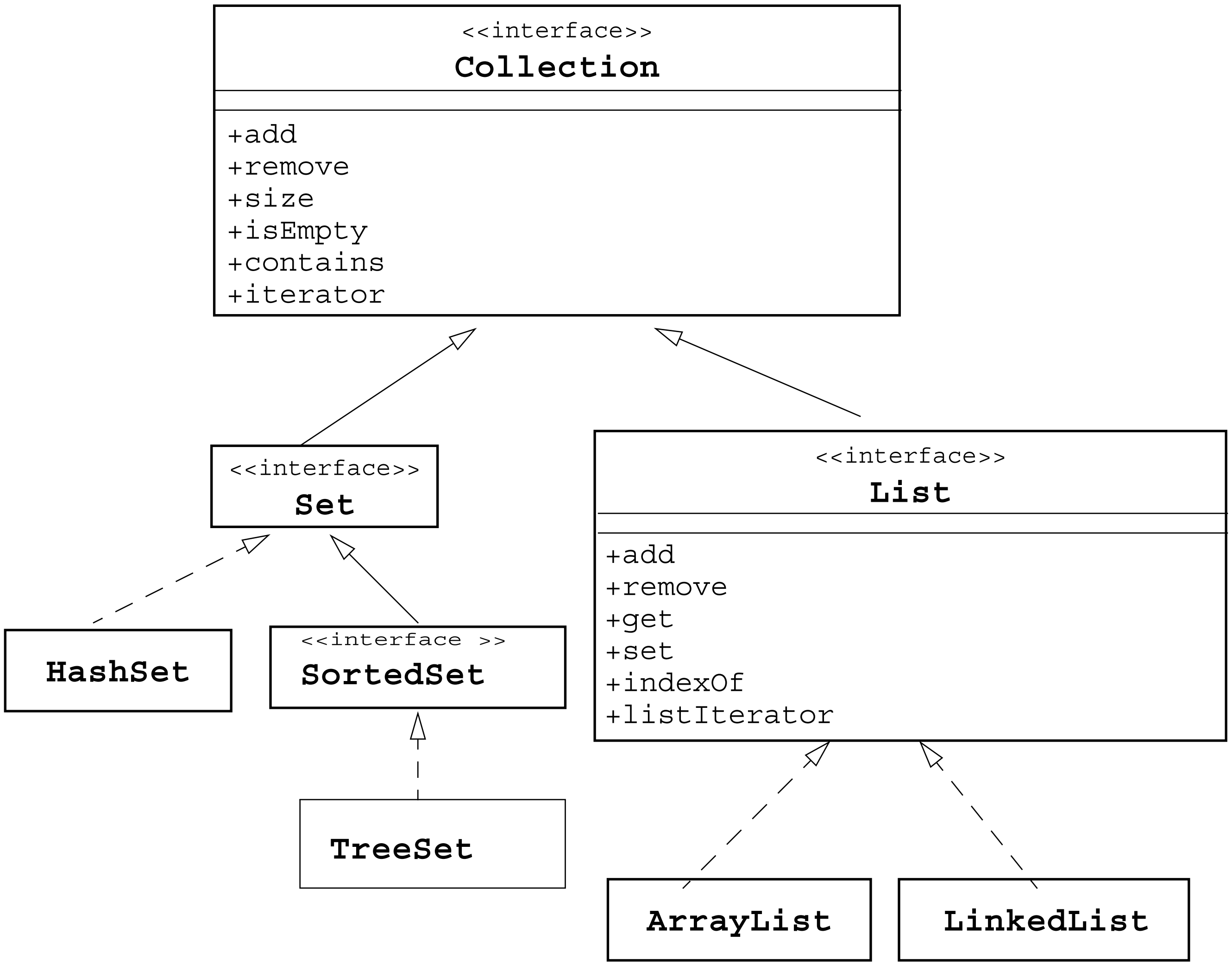
Objectives

- Describe the Collections
- Describe the general purpose implementations of the core interfaces in the Collections framework
- Examine the `Map` interface
- Examine the legacy collection classes
- Create natural and custom ordering by implementing the `Comparable` and `Comparator` interfaces
- Use generic collections
- Use type parameters in generic classes
- Refactor existing non-generic code
- Write a program to iterate over a collection
- Examine the enhanced `for` loop

The Collections API

- A *collection* is a single object managing a group of objects known as its elements.
- The Collections API contains interfaces that group objects as one of the following:
  - `Collection` – A group of objects called elements; implementations determine whether there is specific ordering and whether duplicates are permitted.
  - `Set` – An unordered collection; no duplicates are permitted.
  - `List` – An ordered collection; duplicates are permitted.

The Collections API



Collection Implementations

There are several general purpose implementations of the core interfaces (`Set`, `List`, `Deque` and `Map`)

	Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**	**Hash Table + Linked List**
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

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);
  }
}
```

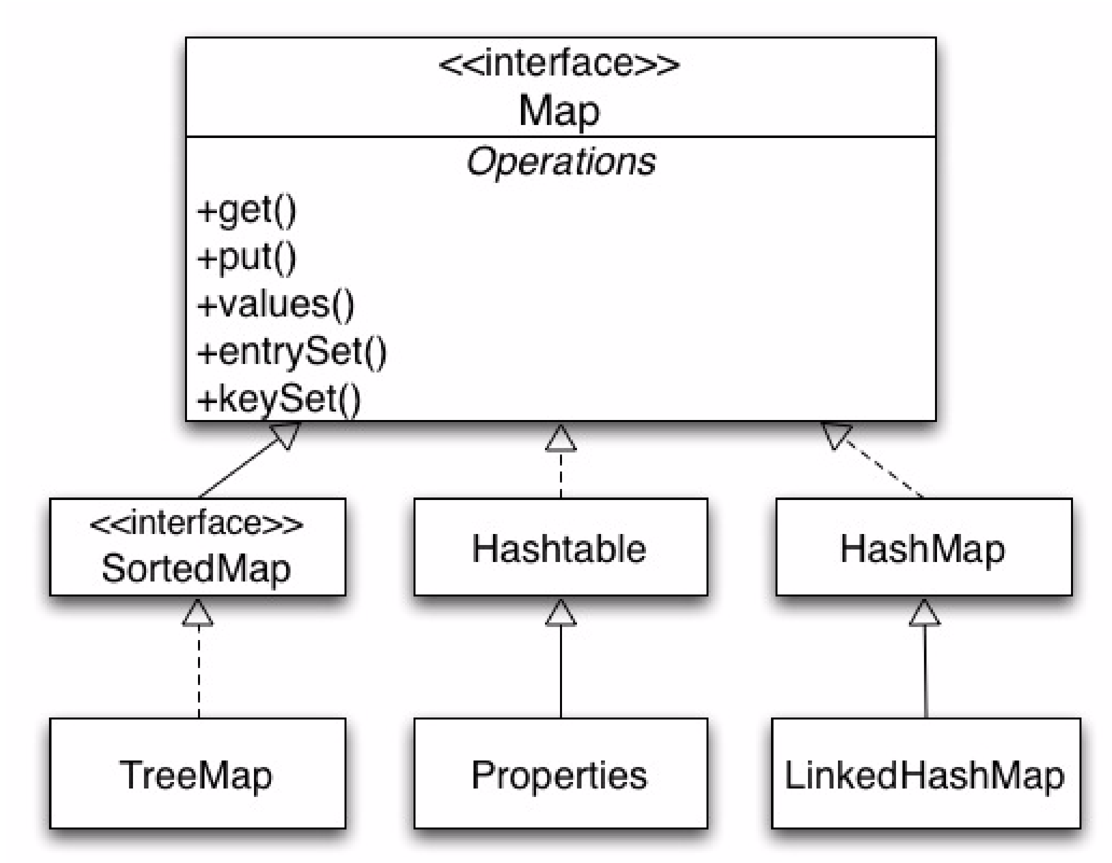
The output generated from this program is:

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

The `Map` Interface

- Maps are sometimes called associative arrays
- A `Map` object describes mappings from keys to values:
  - Duplicate keys are not allowed
  - One-to-many mappings from keys to values is not permitted
- The contents of the `Map` interface can be viewed and manipulated as collections
  - `entrySet` – Returns a `Set` of all the key-value pairs.
  - `keySet` – Returns a `Set` of all the keys in the map.
  - `values` – Returns a `Collection` of all values in the map.

The `Map` Interface API



A `Map` Example

```
import java.util.*;
public class MapExample {
  public static void main(String args[]) {
    Map map = new HashMap();
    map.put("one","1st");
    map.put("second", new Integer(2));
    map.put("third","3rd");
    // Overwrites the previous assignment
    map.put("third","III");
    // Returns set view of keys
    Set set1 = map.keySet();
    // Returns Collection view of values
    Collection collection = map.values();
    // Returns set view of key value mappings
    Set set2 = map.entrySet();
    System.out.println(set1 + "\n" + collection + "\n" + set2);
  }
}
```

A `Map` Example

Output generated from the `MapExample` program:

```
[second, one, third]
[2, 1st, III]
[second=2, one=1st, third=III]
```

Legacy Collection Classes

Collections in the JDK include:

- The `Vector` class, which implements the `List` interface.
- The `Stack` class, which is a subclass of the `Vector` class and supports the `push`, `pop`, and `peek` methods.
- The `Hashtable` class, which implements the `Map` interface.
- The `Properties` class is an extension of `Hashtable` that only uses `String`s for keys and values.
- Each of these collections has an `elements` method that returns an `Enumeration` object. The `Enumeration` interface is incompatible with, the `Iterator` interface.

Ordering Collections

The `Comparable` and `Comparator` interfaces are useful for ordering collections:

- The `Comparable` interface imparts natural ordering to classes that implement it.
- The `Comparator` interface specifies order relation. It can also be used to override natural ordering.
- Both interfaces are useful for sorting collections.

The `Comparable` Interface

Imparts natural ordering to classes that implement it:

- Used for sorting
- The `compareTo` method should be implemented to make any class comparable:
  - `int compareTo(Object o)` method
- The `String`, `Date`, and `Integer` classes implement the `Comparable` interface
- You can sort the `List` elements containing objects that implement the `Comparable` interface

The `Comparable` Interface

- While sorting, the `List` elements follow the natural ordering of the element types
  - `String` elements – Alphabetical order
  - `Date` elements – Chronological order
  - `Integer` elements – Numerical order

Example of the `Comparable` Interface

```
import java.util.*;
class Student implements Comparable {
  String firstName, lastName;
  int studentID=0;
  double GPA=0.0;
  public Student(String firstName, String lastName, int studentID,
    double GPA) {
    if (firstName == null || lastName == null || studentID == 0
     || GPA == 0.0) {throw new IllegalArgumentException();}
   this.firstName = firstName;
   this.lastName = lastName;
   this.studentID = studentID;
    this.GPA = GPA;
  }
  public String firstName() { return firstName; }
  public String lastName()  { return lastName;  }
  public int studentID() { return studentID; }
  public double GPA() { return GPA; }
```

Example of the `Comparable` Interface

```
    // Implement compareTo method.
    public int compareTo(Object o) {
     double f = GPA-((Student)o).GPA;
     if (f == 0.0)
      return 0;    // 0 signifies equals
     else if (f<0.0)
        return -1;   // negative value signifies less than or before
     else
        return 1;    // positive value signifies more than or after
    }
  }
```

Example of the `Comparable` Interface

```
import java.util.*;
public class ComparableTest {
  public static void main(String[] args) {
   TreeSet studentSet = new TreeSet();
   studentSet.add(new Student("Mike", "Hauffmann",101,4.0));
   studentSet.add(new Student("John", "Lynn",102,2.8 ));
   studentSet.add(new Student("Jim", "Max",103, 3.6));
   studentSet.add(new Student("Kelly", "Grant",104,2.3));
    Object[] studentArray = studentSet.toArray();
    Student s;
    for(Object obj : studentArray){
      s = (Student) obj;
      System.out.printf("Name = %s %s ID = %d GPA = %.1f\n",
        s.firstName(), s.lastName(), s.studentID(), s.GPA());
    }
  }
}
```

Example of the `Comparable` Interface

Generated Output:

```
Name = Kelly Grant ID = 104 GPA = 2.3
Name = John Lynn ID = 102 GPA = 2.8
Name = Jim Max ID = 103 GPA = 3.6
Name = Mike Hauffmann ID = 101 GPA = 4.0
```

The `Comparator` Interface

- Represents an order relation
- Used for sorting
- Enables sorting in an order different from the natural order
- Used for objects that do not implement the `Comparable` interface
- Can be passed to a sort method

You need the `compare` method to implement the `Comparator` interface:

- `int compare(Object o1, Object o2)` method

Example of the `Comparator` Interface

```
class Student {
  String firstName, lastName;
  int studentID=0;
  double GPA=0.0;
  public Student(String firstName, String lastName,
    int studentID, double GPA) {
    if (firstName == null || lastName == null || studentID == 0 ||
      GPA == 0.0) throw new NullPointerException();
      this.firstName = firstName;
      this.lastName = lastName;
      this.studentID = studentID;
      this.GPA = GPA;
  }
  public String firstName() { return firstName; }
  public String lastName()  { return lastName;  }
  public int studentID() { return studentID; }
  public double GPA() { return GPA; }
}
```

Example of the `Comparator` Interface

```
import java.util.*;
public class NameComp implements Comparator {
  public int compare(Object o1, Object o2) {
    return
      (((Student)o1).firstName.compareTo(((Student)o2).firstName));
  }
}
```

```
import java.util.*;
public class GradeComp implements Comparator {
  public int compare(Object o1, Object o2) {
   if (((Student)o1).GPA == ((Student)o2).GPA)
    return 0;
   else if (((Student)o1).GPA < ((Student)o2).GPA)
    return -1;
   else
    return 1;
  }
}
```

Example of the `Comparator` Interface

```
import java.util.*;
public class ComparatorTest {
  public static void main(String[] args) {
    Comparator c = new NameComp();
    TreeSet studentSet = new TreeSet(c);
    studentSet.add(new Student("Mike", "Hauffmann",101,4.0));
    studentSet.add(new Student("John", "Lynn",102,2.8 ));
    studentSet.add(new Student("Jim", "Max",103, 3.6));
    studentSet.add(new Student("Kelly", "Grant",104,2.3));
    Object[] studentArray = studentSet.toArray();
    Student s;
    for(Object obj : studentArray) {
      s = (Student) obj;
      System.out.println("Name = %s %s ID = %d GPA = %.1f\n",
        s.firstName(), s.lastName(), s.studentID(), s.GPA());
    }
  }
}
```

Example of the `Comparator` Interface

```
Name = Jim Max ID = 0 GPA = 3.6
Name = John Lynn ID = 0 GPA = 2.8
Name = Kelly Grant ID = 0 GPA = 2.3
Name = Mike Hauffmann ID = 0 GPA = 4.0
```

Generics

Generics are described as follows:

- Provide compile-time type safety
- Eliminate the need for casts
- Provide the ability to create compiler-checked homogeneous collections

Generics

Using non-generic collections:

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer)list.get(0)).intValue();
```

Using generic collections:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

Generic `Set` Example

```
import java.util.*;
public class GenSetExample {
  public static void main(String[] args) {
    Set<String> set = new HashSet<String>();
    set.add("one");
    set.add("second");
    set.add("3rd");
// This line generates compile error
    set.add(new Integer(4));
    set.add("second");
// Duplicate, not added
    System.out.println(set);
  }
}
```

Generic Map Example

```
import java.util.*;

public class MapPlayerRepository {
  HashMap<String, String> players;

  public MapPlayerRepository() {
    players = new HashMap<String, String> ();
  }

  public String get(String position) {
    String player = players.get(position);
    return player;
  }

  public void put(String position, String name) {
    players.put(position, name);
  }
```

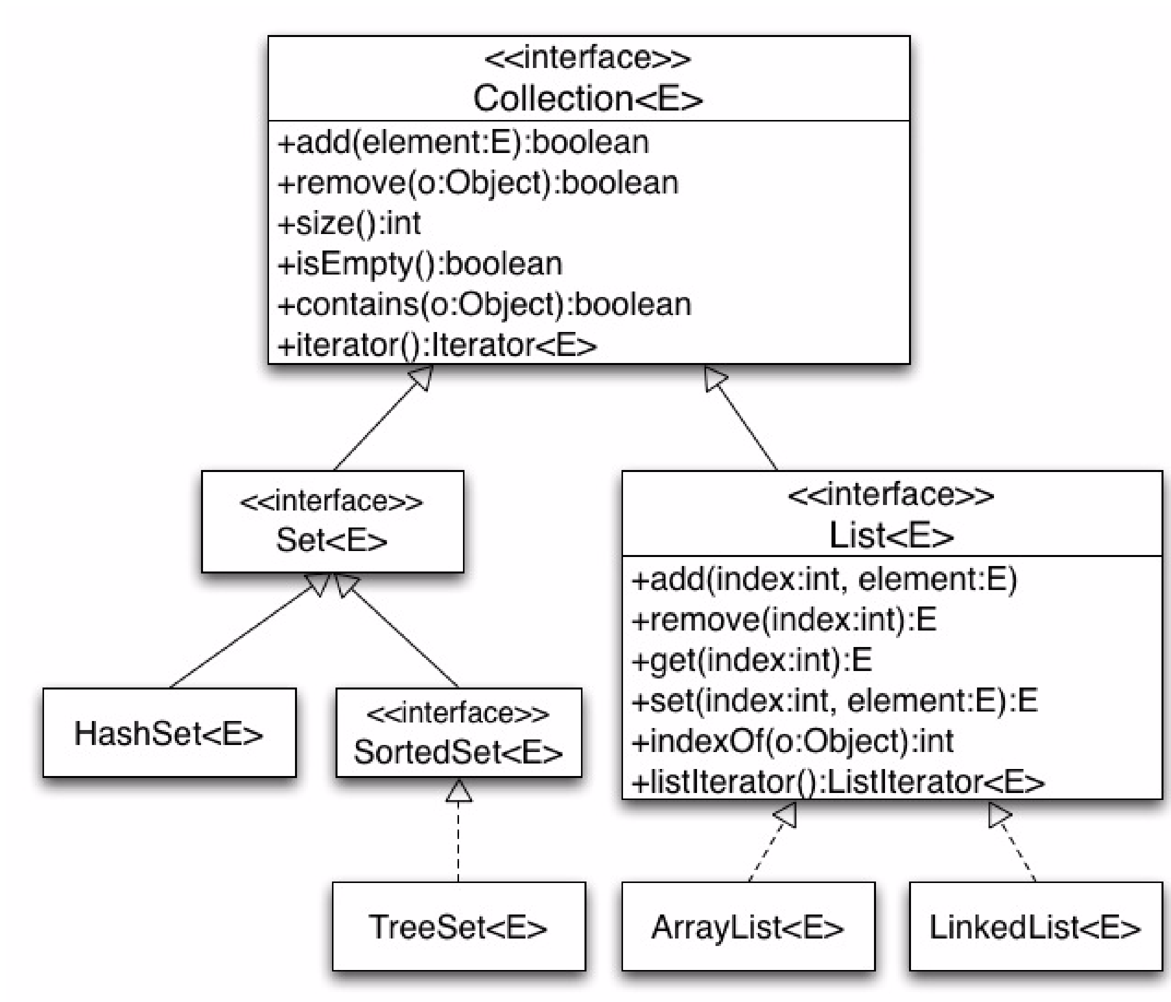
Generics: Examining Type Parameters

Shows how to use type parameters

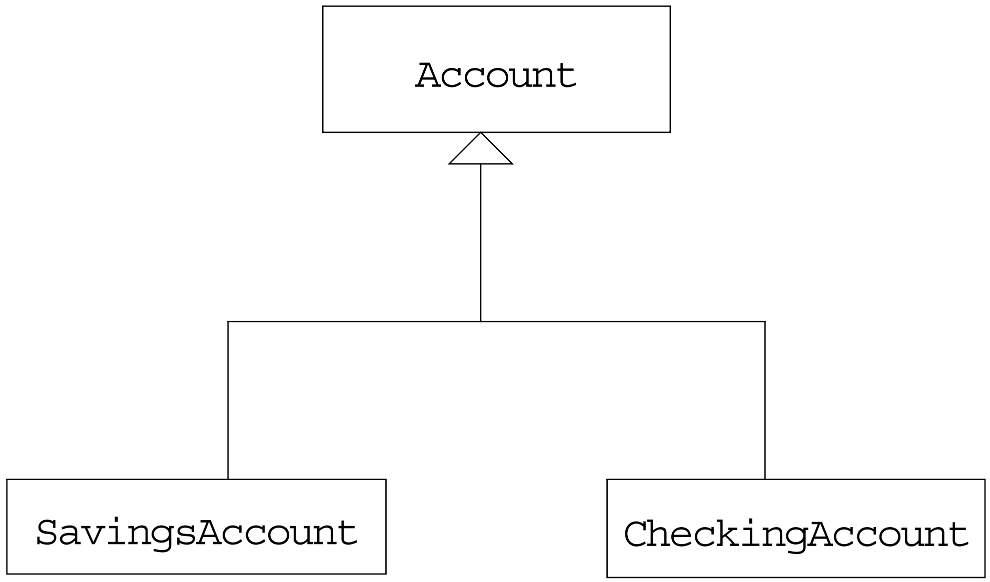
Category	**Non Generic Class**	**Generic Class**
Class declaration	public class ArrayList extends AbstractList implements List	public class ArrayList<E> extends AbstractList<E> implements List <E>
Constructor declaration	public ArrayList (int capacity);	public ArrayList<E> (int capacity);
Method declaration	public void add((Object o) public Object get(int index)	public void add(E o) public E get(int index)
Variable declaration examples	ArrayList list1; ArrayList list2;	ArrayList <String> list1; ArrayList <Date> list2;

Instance declaration examples	list1 = new ArrayList(10);
	
	list2 = new ArrayList(10);
		list1= new ArrayList<String> (10);
		
		list2= new ArrayList<Date> (10);

Generic Collections API



Wild Card Type Parameters



The Type-Safety Guarantee

```
public class TestTypeSafety {

  public static void main(String[] args) {
    List<CheckingAccount> lc = new ArrayList<CheckingAccount>();

    lc.add(new CheckingAccount("Fred")); // OK
    lc.add(new SavingsAccount("Fred"));  // Compile error!

    // therefore...
    CheckingAccount ca = lc.get(0);      // Safe, no cast required
  }
}
```

The Invariance Challenge

```
List<Account> la;
List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
List<SavingsAccount> ls = new ArrayList<SavingsAccount>();

//if the following were possible...
la = lc;
la.add(new CheckingAccount("Fred"));

//then the following must also be possible...
la = ls;
la.add(new CheckingAccount("Fred"));

//so...
SavingsAccount sa = ls.get(0); //aarrgghh!!
```

In fact, `la=lc;` is illegal, so even though a `CheckingAccount` is an `Account`, an `ArrayList<CheckingAccount>` is not an `ArrayList<Account>`.

The Covariance Response

```
public static void printNames(List <? extends Account> lea) {
    for (int i=0; i < lea.size(); i++) {
      System.out.println(lea.get(i).getName());
    }
  }

  public static void main(String[] args) {
    List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
    List<SavingsAccount> ls = new ArrayList<SavingsAccount>();

    printNames(lc);
    printNames(ls);

    //but...
    List<? extends Object> leo = lc; //OK
    leo.add(new CheckingAccount("Fred"));//Compile error!
  }
}
```

Generics: Refactoring Existing Non-Generic Code

```
import java.util.*;
public class GenericsWarning {
  public static void main(String[] args) {
    List list = new ArrayList();
    list.add(0, new Integer(42));
    int total = ((Integer)list.get(0).intValue();
  }
}
```

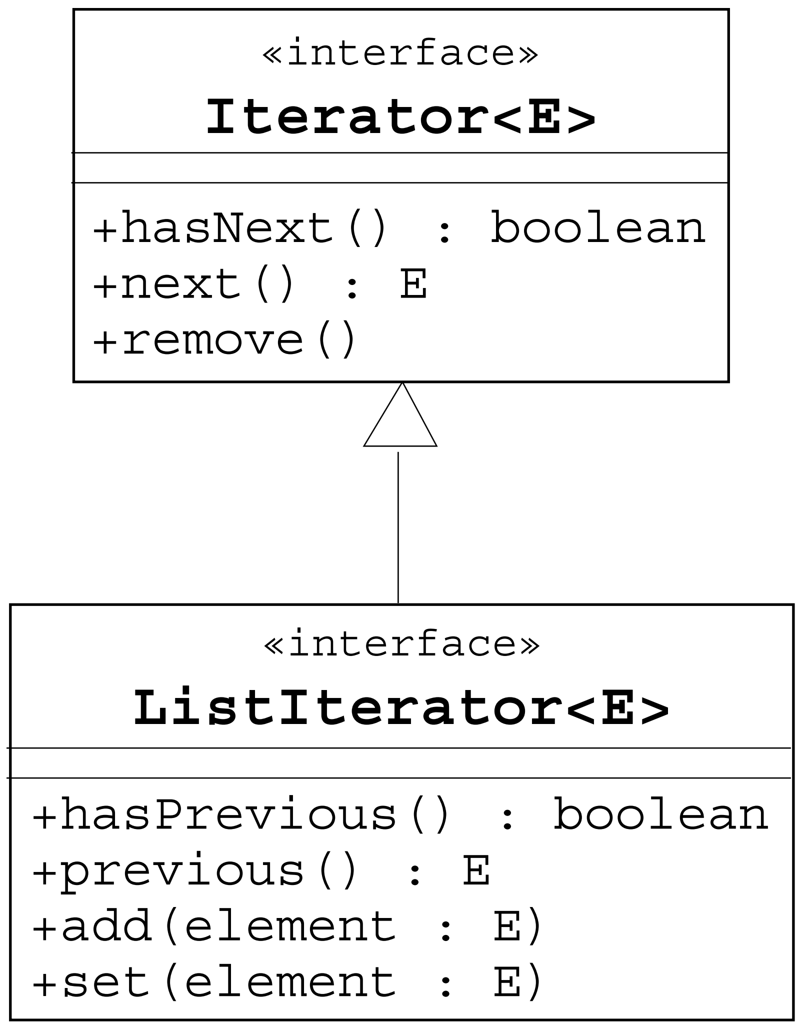
```
javac GenericsWarning.java
Note: GenericsWarning.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

javac -Xlint:unchecked GenericsWarning.java
GenericsWarning.java:7: warning: [unchecked] unchecked call to add(int,E)
as a member of the raw type java.util.ArrayList
    list.add(0, new Integer(42));
            ^

1 warning
```

Iterators

- Iteration is the process of retrieving every element in a collection.
- The basic `Iterator` interface allows you to scan forward through any collection.
- A `List` object supports the `ListIterator`, which allows you to scan the list backwards and insert or modify elements.

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

Generic Iterator Interfaces



The Enhanced `for` Loop

The enhanced `for` loop has the following characteristics:

- Simplified iteration over collections
- Much shorter, clearer, and safer
- Effective for arrays
- Simpler when using nested loops
- Iterator disadvantages removed

Iterators are error prone:

- Iterator variables occur three times per loop.
- This provides the opportunity for code to go wrong.

The Enhanced `for` Loop

An enhanced `for` loop can look like the following:

- Using the iterator with a traditional `for` loop:

```
public void deleteAll(Collection<NameList> c){
  for ( Iterator<NameList> i = c.iterator() ; i.hasNext() ; ){
    NameList nl = i.next();
    nl.deleteItem();
  }
}
```

- Iterating using an enhanced `for` loop in collections:

```
public void deleteAll(Collection<NameList> c){
  for ( NameList nl : c ){
    nl.deleteItem();
  }
}
```

The Enhanced `for` Loop

- Nested enhanced `for` loops:

```
List<Subject> subjects=...;
List<Teacher> teachers=...;
List<Course> courseList = ArrayList<Course)();
for (Subject subj: subjects) {
  for (Teacher tchr: teachers) {
    courseList.add(new Course(subj, tchr));
  }
}
```

Module 10

I/O Fundamentals

Objectives

- Write a program that uses command-line arguments and system properties
- Examine the `Properties` class
- Construct node and processing streams, and use them appropriately
- Serialize and deserialize objects
- Distinguish readers and writers from streams, and select appropriately between them

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] + "'");
    }
  }
}
```

Example execution:

```
java TestArgs arg0 arg1 "another arg"
args[0] is 'arg0'
args[1] is 'arg1'
args[2] is 'another arg'
```

System Properties

- System properties are a feature that replaces the concept of *environment variables* (which are 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 on the command line 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`.

The Properties Class

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

public class TestProperties {
  public static void main(String[] args) {
    Properties props = System.getProperties();
    props.list(System.out);
  }
}
```

The `Properties` Class

The following is an example test run of this program:

```
java -DmyProp=theValue TestProperties
```

The following is the (partial) output:

```
java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\jse\jdk1.6.0\jre\bin
java.vm.version=1.6.0-b105
java.vm.vendor=Sun Microsystems Inc.
java.vm.name=Java HotSpot(TM) Client VM
file.encoding.pkg=sun.io
user.country=US
myProp=theValue
```

I/O Stream Fundamentals

- A *stream* is 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

Stream	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.

The `InputStream` Methods

- The three basic `read` methods are:

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

- Other methods include:

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

The `OutputStream` Methods

- The three basic `write` methods are:

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

- Other methods include:

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

The `Reader` Methods

- The three basic `read` methods are:

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

- Other methods include:

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

The `Writer` Methods

- The basic `write` methods are:

```
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)
```

- Other methods include:

```
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`	N/A
Pipe	`PipedReader` `PipedWriter`	`PipedInputStream` `PipedOutputStream`

A Simple Example

This program performs a copy file operation using a manual buffer:

`java TestNodeStreams file1 file2`

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

          // read the first buffer
          charsRead = input.read(buffer);
          while ( charsRead != -1 ) {
            // write buffer to the output file
```

A Simple Example

```
            output.write(buffer, 0, charsRead);

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

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

Buffered Streams

This program performs a copy file operation using a built-in buffer:

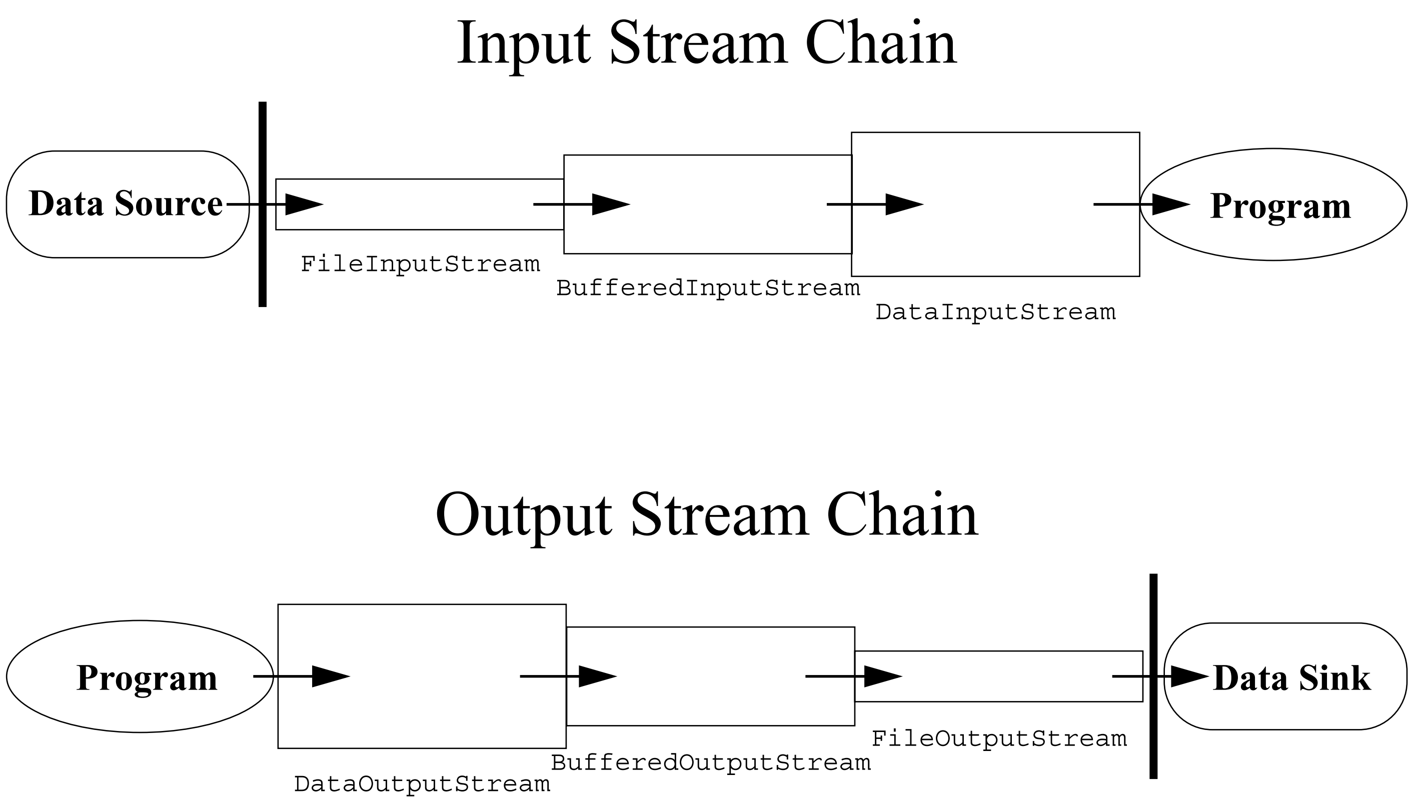
`java TestBufferedStreams file1 file2`

```
import java.io.*;
public class TestBufferedStreams {
  public static void main(String[] args) {
    try {
      FileReader input = new FileReader(args[0]);
      BufferedReader bufInput = new BufferedReader(input);
      try {
        FileWriter output = new FileWriter(args[1]);
        BufferedWriter bufOutput= new BufferedWriter(output);
        try {
          String line;
          // read the first line
          line = bufInput.readLine();
          while ( line != null ) {
            // write the line out to the output file
```

Buffered Streams

```
            bufOutput.write(line, 0, line.length());
            bufOutput.newLine();
            // read the next line
            line = bufInput.readLine();
          }
        } finally {
          bufOutput.close();
        }
      } finally {
        bufInput.close();
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}


```

I/O Stream Chaining



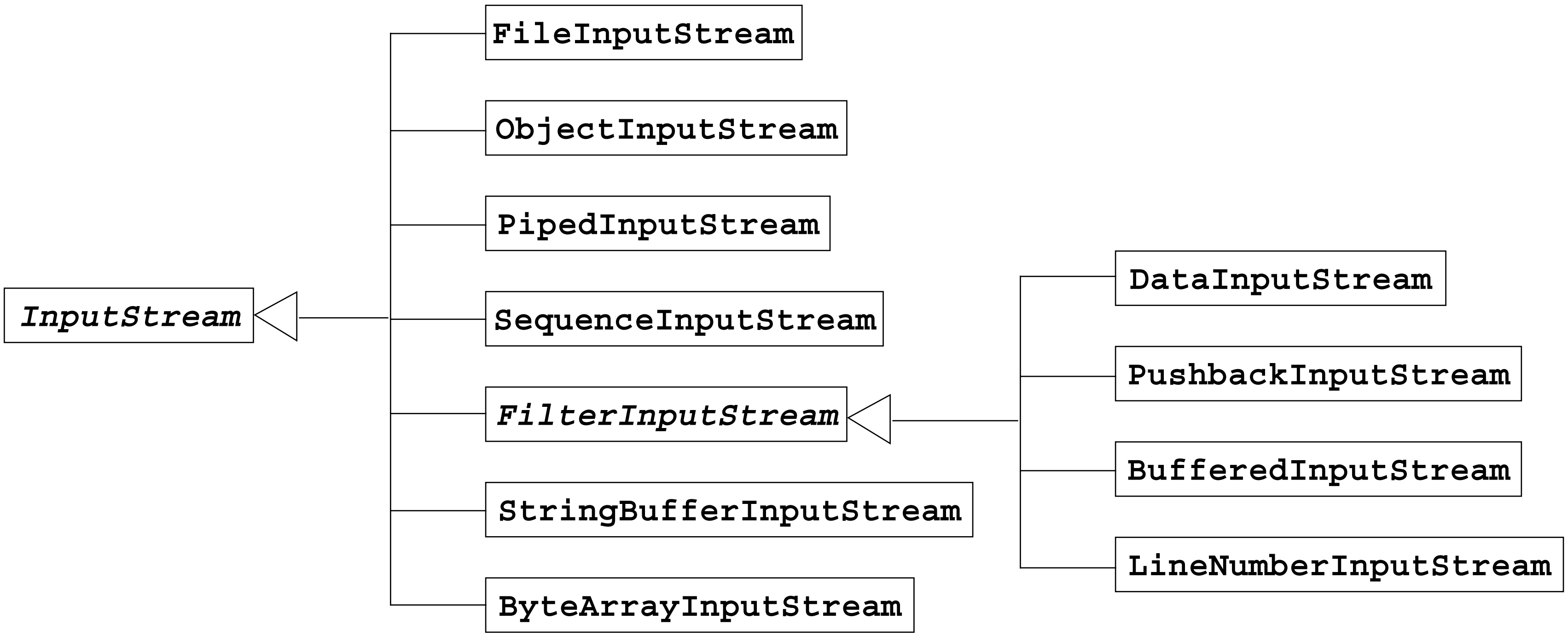
Processing Streams

Type	Character Streams	Byte Streams
Buffering	`BufferedReader` `BufferedWriter`	`BufferedInputStream` `BufferedOutputStream`
Filtering	*`FilterReader`* *`FilterWriter`*	*`FilterInputStream`* *`FilterOutputStream`*
Converting between bytes and character	`InputStreamReader` `OutputStreamWriter`	
Performing object serialization		`ObjectInputStream` `ObjectOutputStream`

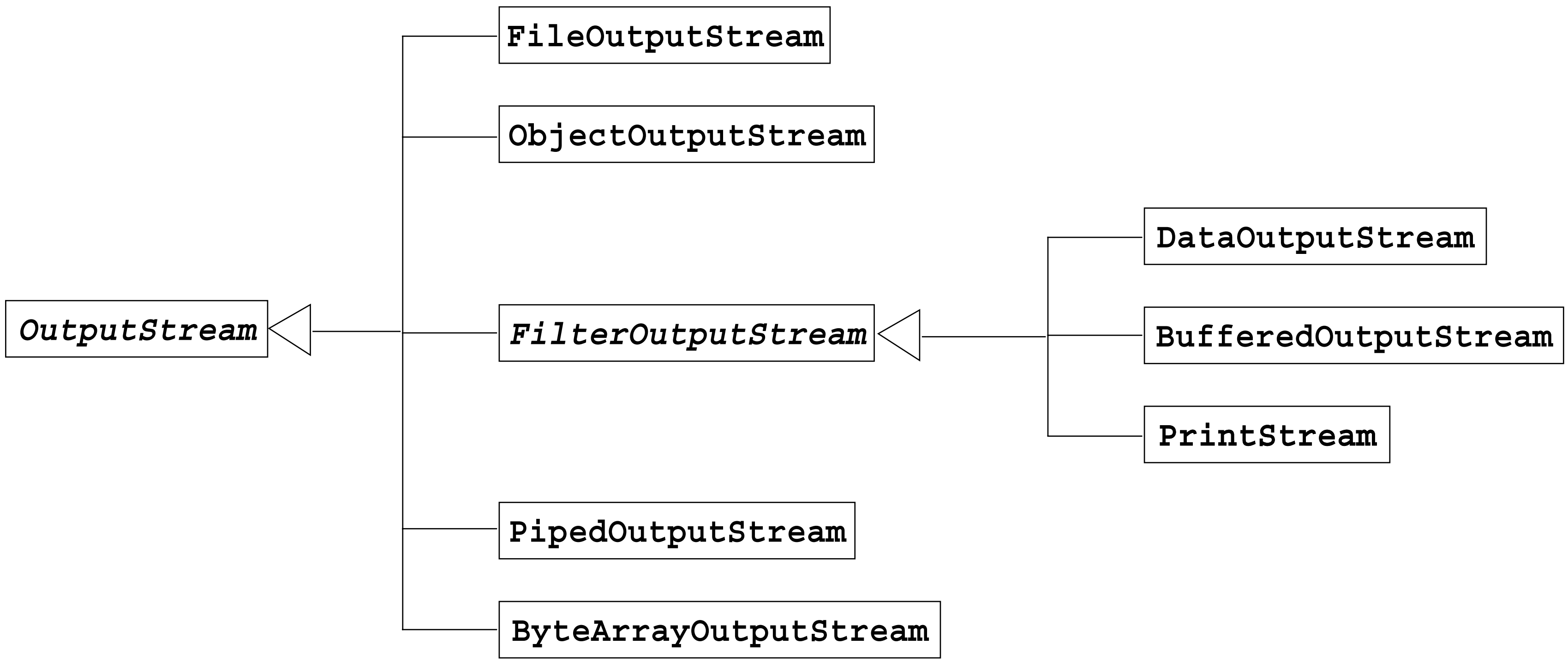
Processing Streams

Type	Character Streams	Byte Streams
Performing data conversion		`DataInputStream` `DataOutputStream`
Counting	`LineNumberReader`	`LineNumberInputStream`
Peeking ahead	`PushbackReader`	`PushbackInputStream`
Printing	`PrintWriter`	`PrintStream`

The `InputStream` Class Hierarchy



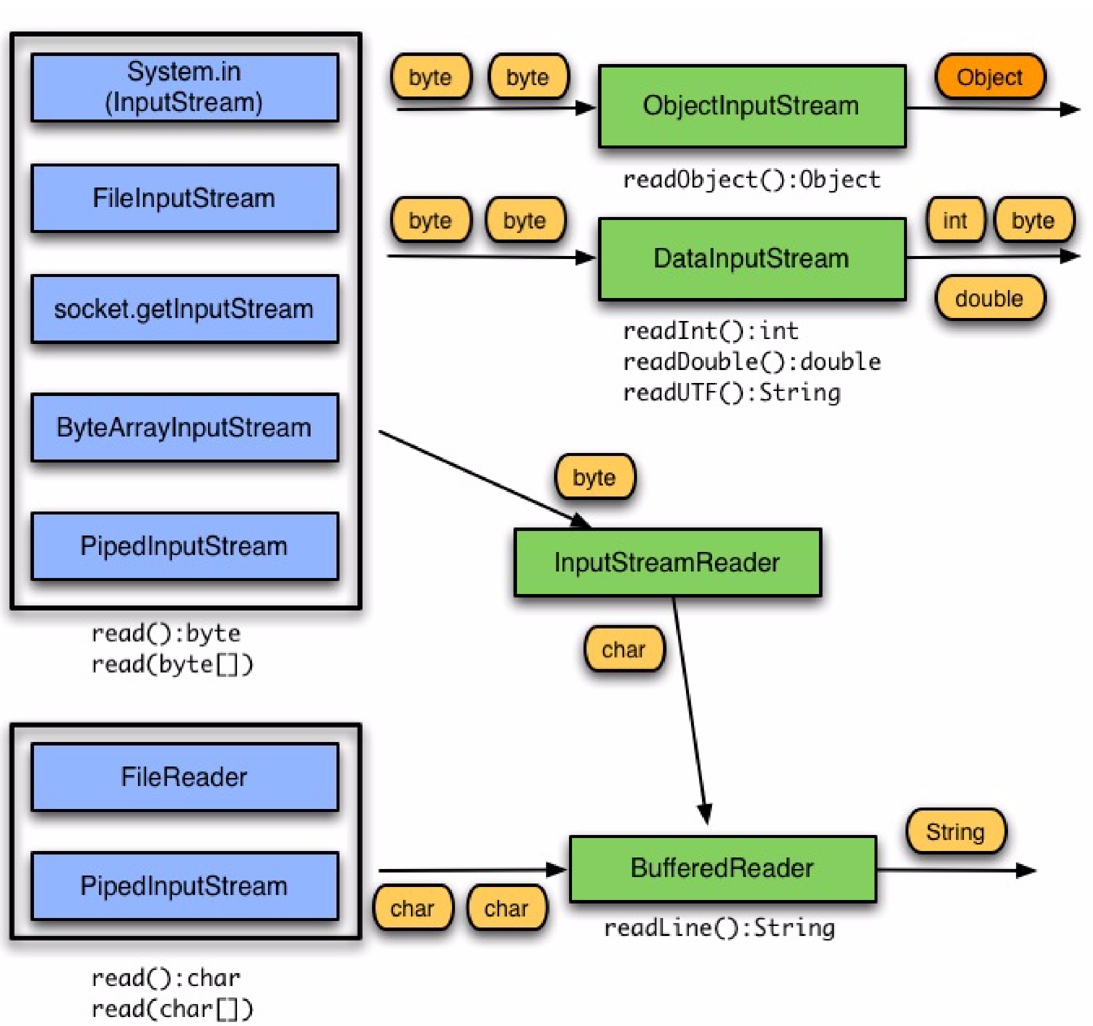
The OutputStream Class Hierarchy



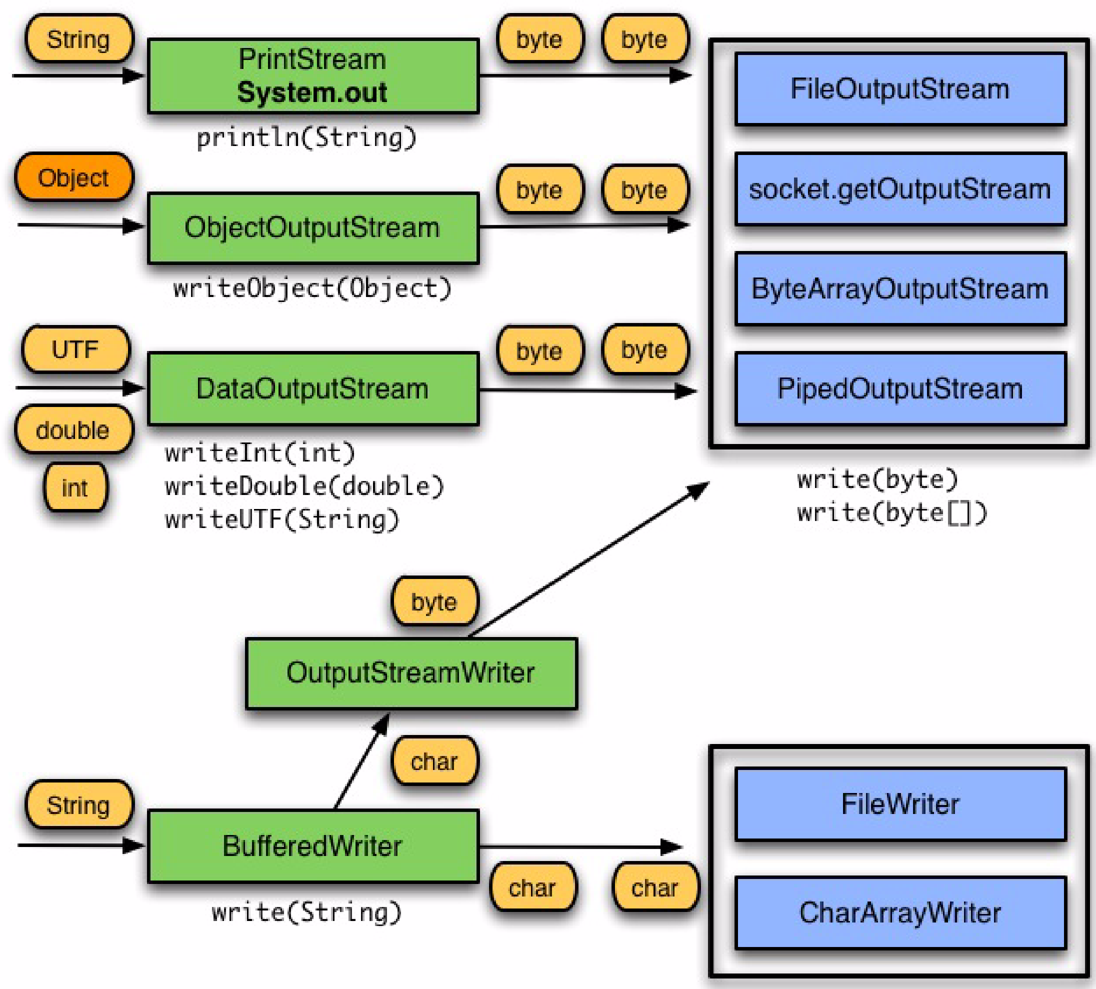
The `ObjectInputStream` and The `ObjectOutputStream` Classes

- The Java API provides a standard mechanism (called object serialization) that completely automates the process of writing and reading objects from streams.
- When writing an object, the object input stream writes the class name, followed by a description of the data members of the class, in the order they appear in the stream, followed by the values for all the fields on that object.
- When reading an object, the object output stream reads the name of the class and the description of the class to match against the class in memory, and it reads the values from the stream to populate a newly allocation instance of that class.
- Persistent storage of objects can be accomplished if files (or other persistent storage) are used as streams.

Input Chaining Combinations: A Review



Output Chaining Combinations: A Review



Serialization

- Serialization is a mechanism for saving the objects as a sequence of bytes and rebuilding them later when needed.
- When an object is serialized, only the fields of the object are preserved
- When a field references an object, the fields of the referenced object are also serialized
- Some object classes are not serializable because their fields represent transient operating system-specific information.

The `SerializeDate` Class

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

public class SerializeDate {

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

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

The `SerializeDate` Class

```

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

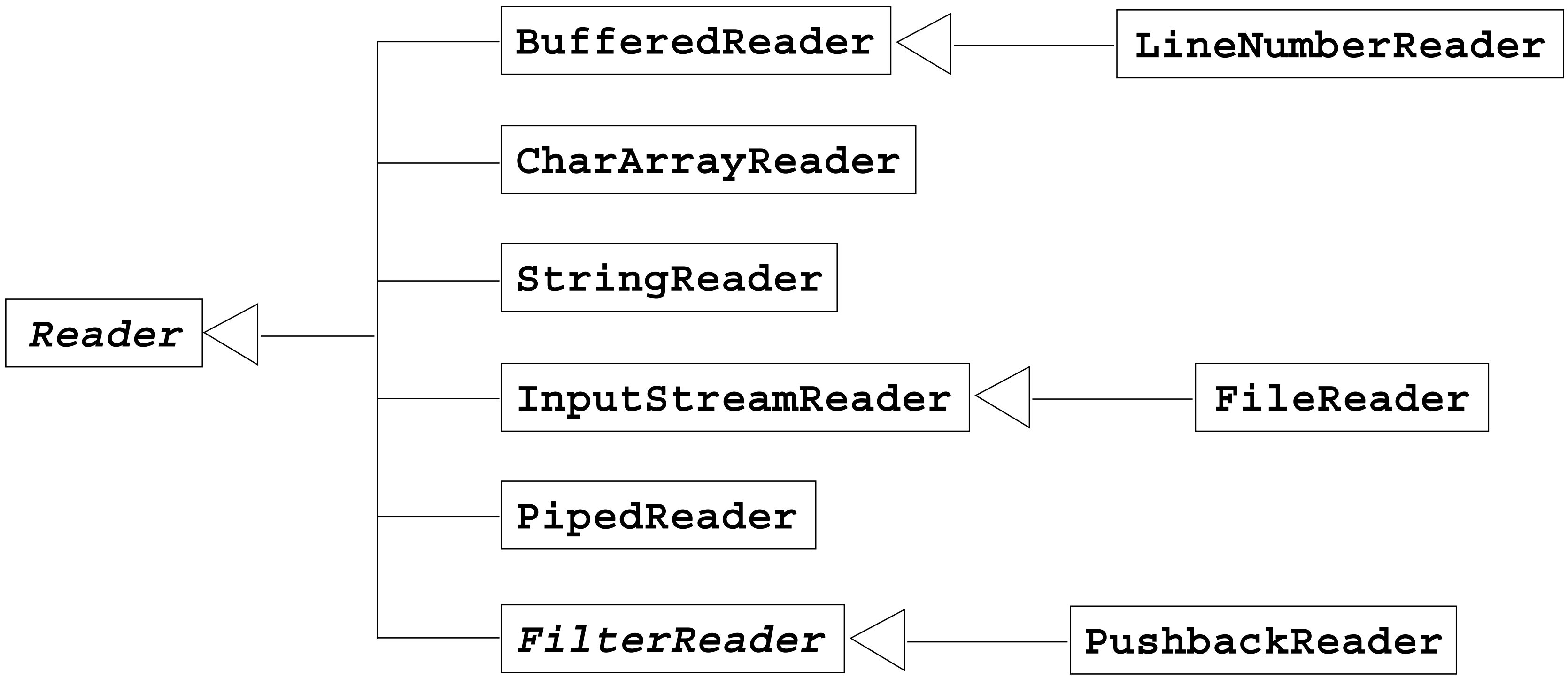
The `DeSerializeDate` Class

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

public class DeSerializeDate {

  DeSerializeDate () {
    Date d = null;

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

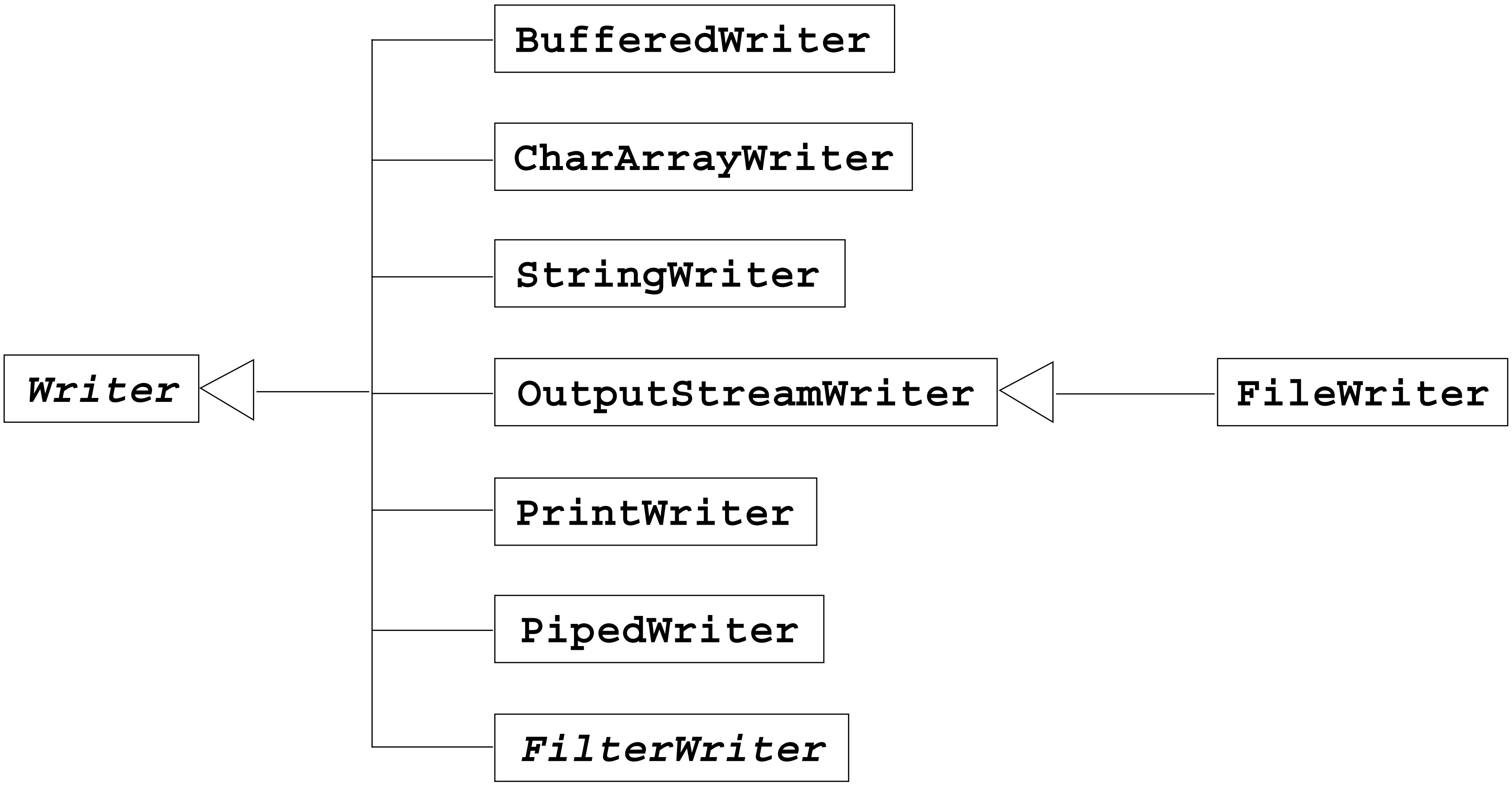
The `DeSerializeDate` Class

```

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

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

The Reader Class Hierarchy



The Writer Class Hierarchy



Module 11

Console I/O and File I/O

Objectives

- Read data from the console
- Write data to the console
- Describe files and file I/O

Console I/O

- The variable `System.out` enables you to write to *standard output*.

  `System.out` is an object of type `PrintStream`.
- The variable `System.in` enables you to read from *standard input*.

  `System.in` is an object of type `InputStream`.
- The variable `System.err` enables you to write to *standard error*.

  `System.err` is an object of type `PrintStream`.

Writing to Standard Output

- The `println` methods print the argument and a newline character (`\n`).
- The `print` methods print the argument without a newline character.
- 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 to exit." +
                       "\nWindows: Type ctrl-z to exit");
```

Reading From Standard Input

```
    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();
    }
  }
}
```

Simple Formatted Output

- You can use the formatting functionality as follows:

```
out.printf("name count\n");
String s = String.format("%s %5d%n", user, total);
```

- Common formatting codes are listed in this table.

Code	Description
`%s`	Formats the argument as a string, usually by calling the `toString` method on the object.
`%d %o %x`	Formats an integer, as a decimal, octal, or hexadecimal value.
`%f %g`	Formats a floating point number. The `%g` code uses scientific notation.
`%n`	Inserts a newline character to the string or stream.
`%%`	Inserts the `%` character to the string or stream.

Simple Formatted Input

- The `Scanner` class provides a formatted input function.
- A `Scanner` class can be used with console input streams as well as file or network streams.
- You can read console input as follows:

```
import java.io.*;
import java.util.Scanner;
public class ScanTest {
  public static void main(String [] args) {
    Scanner s = new Scanner(System.in);
    String param = s.next();
    System.out.println("the param 1" + param);
    int value = s.nextInt();
    System.out.println("second param" + value);
    s.close();
  }
}
```

Files and File I/O

The `java.io` package enables you to do the following:

- Create `File` objects
- Manipulate `File` objects
- Read and write to file streams

Creating a New `File` Object

The `File` class provides several utilities:

- `File myFile;`
- `myFile = new File("myfile.txt");`
- `myFile = new File("MyDocs", "myfile.txt");`

Directories are treated like files in the Java programming language. You can create a `File` object that represents a directory and then use it to identify other files, for example:

```
File myDir = new File("MyDocs");
myFile = new File(myDir, "myfile.txt");
```

The `File` Tests and Utilities

- File information:

```
String getName()
String getPath()
String getAbsolutePath()
String getParent()
long lastModified()
long length()
```

- File modification:

```
boolean renameTo(File newName)
boolean delete()
```

- Directory utilities:

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

The `File` Tests and Utilities

- File tests:

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

File Stream I/O

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

File Input Example

A file input example is:

```
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;

```

Printing a File

```
        // 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
        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.
      InputStreamReader isr
        = new InputStreamReader(System.in);
      BufferedReader in
        = new BufferedReader(isr);
      // Create a print writer on this file.
      PrintWriter out
        = new PrintWriter(new FileWriter(file));
      String s;
```

File Output Example

```

      System.out.print("Enter file text.  ");
      System.out.println("[Type ctrl-d 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();
    }
  }
}
```

Module 12

Building Java GUIs Using the Swing API

Objectives

- Describe the JFC Swing technology
- Define Swing
- Identify the Swing packages
- Describe the GUI building blocks: containers, components, and layout managers
- Examine top-level, general-purpose, and special-purpose properties of container
- Examine components
- Examine layout managers
- Describe the Swing single-threaded model
- Build a GUI using Swing components

What Are the Java Foundation Classes (JFC)?

Java Foundation Classes are a set of Graphical User Interface (GUI) support packages, including:

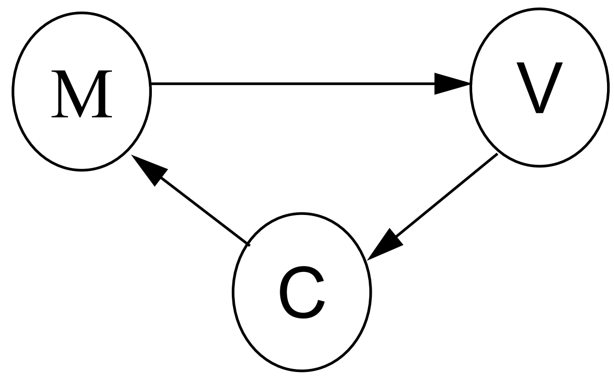
- Abstract Window Toolkit (AWT)
- The Swing component set
- 2D graphics
- Pluggable look-and-feel
- Accessibility
- Drag-and-drop
- Internationalization

What Is Swing?

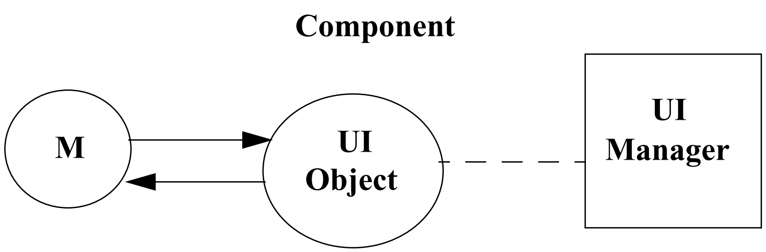
- An enhanced GUI component set
- Provides replacement components for those in the original AWT
- Has special features, such as a pluggable look-and feel

Swing Architecture

- Has its roots in the Model-View-Controller (MVC) architecture



- The Swing components follow Separable Model Architecture



Swing Packages

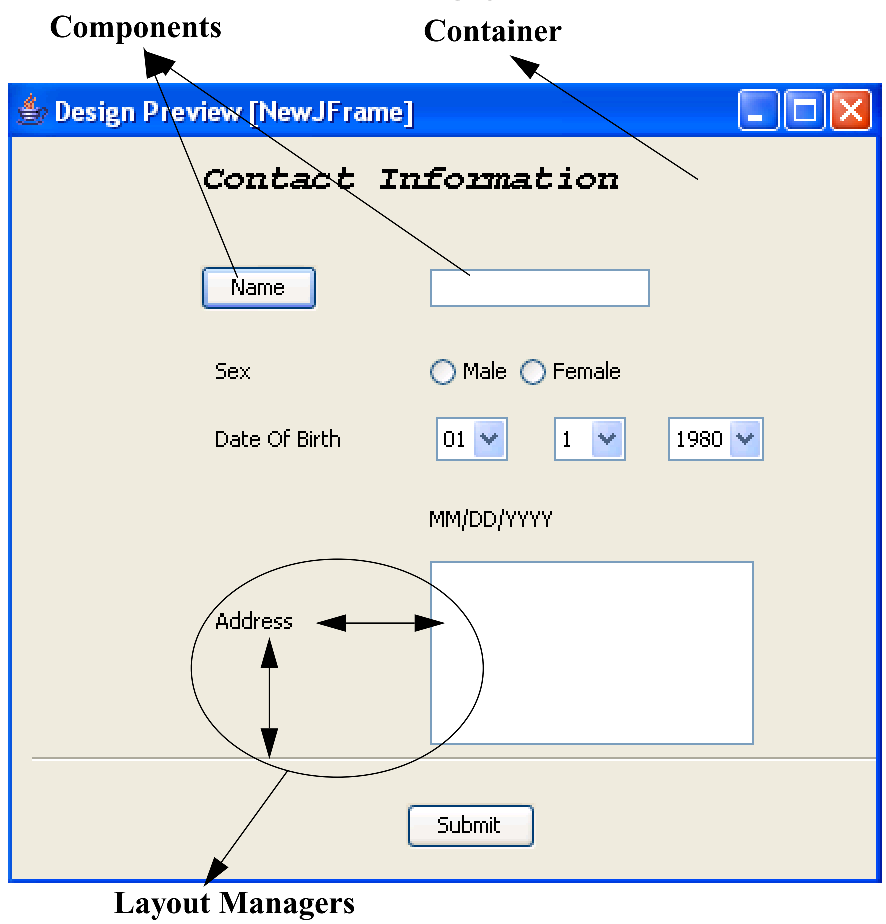
Package Name	Package Name
`javax.swing`	`javax.swing.colorchooser`
`javax.swing.border`	`javax.swing.filechooser`
`javax.swing.event`	`javax.swing.table`
`javax.swing.undo`	`javax.swing.tree`
`javax.swing.plaf`	`javax.swing.text`
`javax.swing.plaf.basic`	`javax.swing.text.html`
`javax.swing.plaf.metal`	`javax.swing.text.html.parser`
`javax.swing.plaf.multi`	`javax.swing.text.rtf`
`javax.swing.plaf.synth`	`javax.swing.undo`

Examining the Composition of a Java Technology GUI

A Swing API-based GUI is composed of the following elements:

- Containers – Are on top of the GUI containment hierarchy.
- Components – Contain all the GUI components that are derived from the `JComponent` class.
- Layout Managers – Are responsible for laying out components in a container.

Examining the Composition of a Java Technology GUI

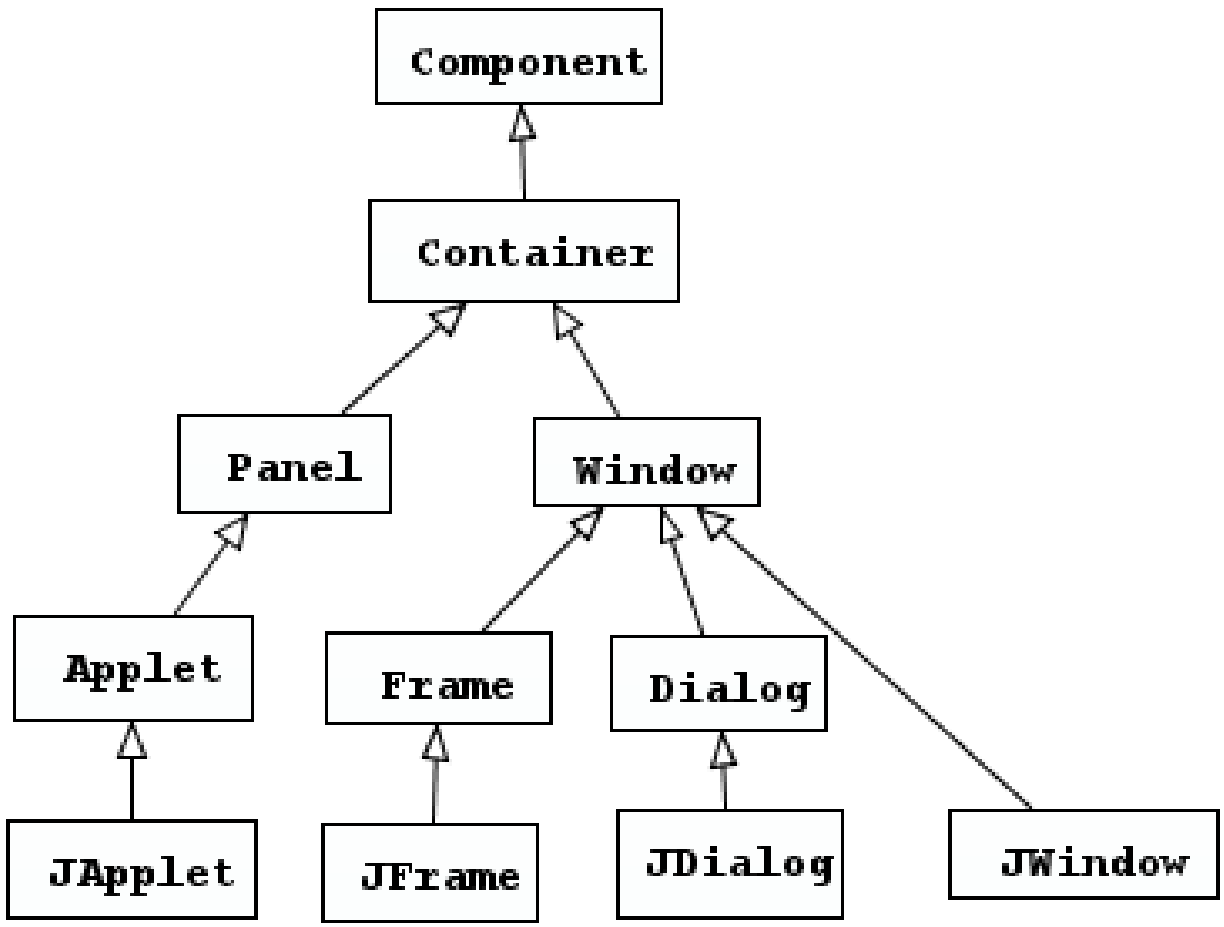


Swing Containers

Swing containers can be classified into three main categories:

- Top-level containers:
  - `JFrame`, `JWindow`, and `JDialog`
- General-purpose containers:
  - `JPanel`, `JScrollPane`, `JToolBar`, `JSplitPane`, and `JTabbedPane`
- Special-purpose containers:
  - `JInternalFrame` and `JLayeredPane`

Top-Level Containers

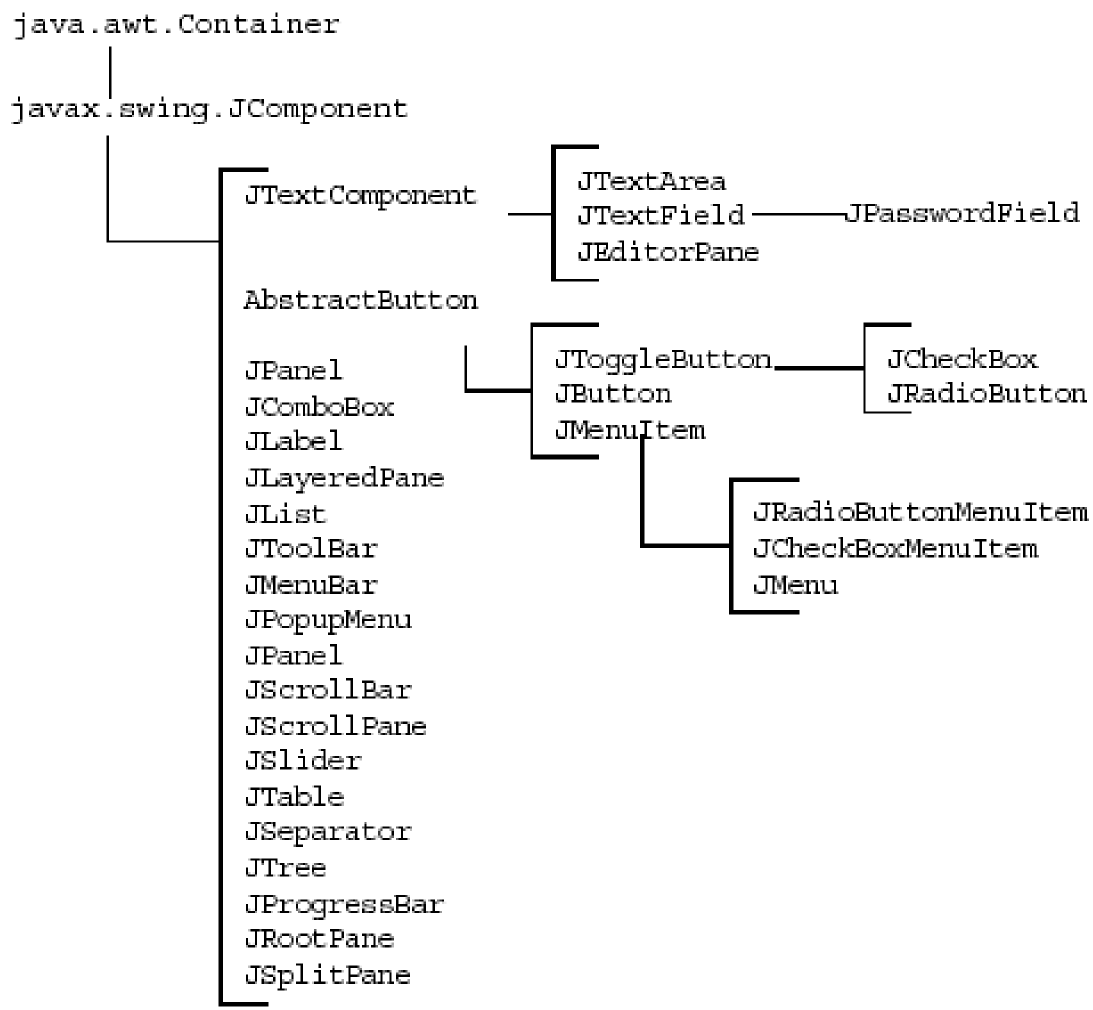


Swing Components

Swing components can be broadly classified as:

- Buttons
- Text components
- Uneditable information display components
- Menus
- Formatted display components
- Other basic controls

Swing Component Hierarchy



Text Components

Swing text components can be broadly divided into three categories.

- Text controls – `JTextField`, `JPasswordField` (for user input)
- Plain text areas – `JTextArea` (displays text in plain text, also for multi-line user input
- Styled text areas – `JEditorPane`, `JTextPane` (displays formatted text)

Swing Component Properties

Common component properties:

- All the Swing components share some common properties because they all extend `JComponent`.

Component-specific properties:

- Each component defines more specific properties.

Common Component Properties

Property	Methods
Border	`Border getBorder()` `void setBorder(Border b)`
Background and foreground color	`void setBackground(Color bg)` `void setForeground(Color bg)`
Font	`void setFont(Font f)`
Opaque	`void setOpaque(boolean isOpaque)`
Maximum and minimum size	`void setMaximumSize(Dimension d)` `void setMinimumSize(Dimension d)`
Alignment	`void setAlignmentX(float ax)` `void setAlignmentY(float ay)`
Preferred size	`void setPreferredSize(Dimension ps)`

Component-Specific Properties

The following shows properties specific to `JComboBox`.

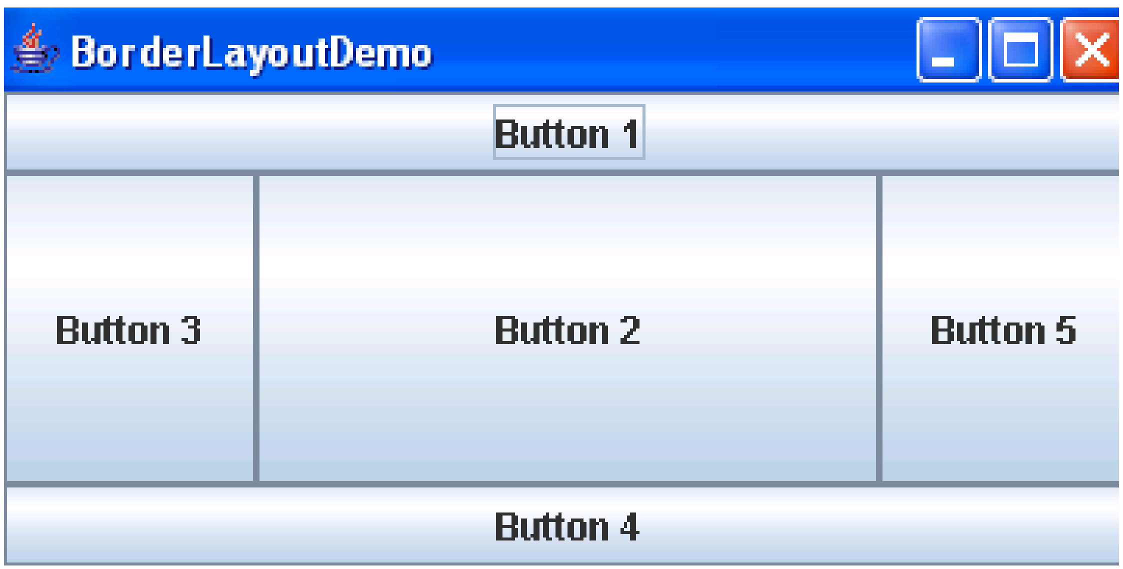
Properties	**Methods**
Maximum row count	`void setMaximumRowCount(int count)`
Model	`void setModal(ComboBoxModel cbm)`
Selected index	`int getSelectedIndex()`
Selected Item	`Object getSelectedItem()`
Item count	`int getItemCount()`
Renderer	`void setRenderer(ListCellRenderer ar)`
Editable	`void setEditable(boolean flag)`

Layout Managers

- Handle problems caused by:
  - GUI resizing by user
  - Operating system differences in fonts
  - Locale-specific text layout requirements
- Layout manager classes:
  - `BorderLayout`
  - `FlowLayout`
  - `BoxLayout`
  - `CardLayout`
  - `GridLayout`
  - `GridBagLayout`

The `BorderLayout` Manager

The `BorderLayout` manager places components in top, bottom, left, right and center locations.



BorderLayout Example

```java
import java.awt.*;
import javax.swing.*;

public class BorderExample {
  private JFrame f;
  private JButton bn, bs, bw, be, bc;

  public BorderExample() {
    f = new JFrame("Border Layout");
    bn = new JButton("Button 1");
    bc = new JButton("Button 2");
    bw = new JButton("Button 3");
    bs = new JButton("Button 4");
    be = new JButton("Button 5");
  }

```

BorderLayout Example

```
  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(400,200);
    f.setVisible(true);
  }

  public static void main(String args[]) {
    BorderExample guiWindow2 = new BorderExample();
    guiWindow2.launchFrame();
  }
}

```

The `FlowLayout` Manager

The `FlowLayout` manager places components in a row, and if the row fills, components are placed in the next row.

FlowLayout Example

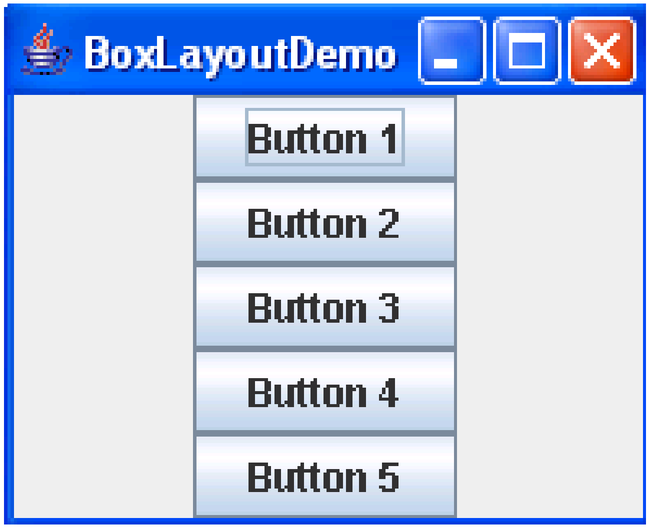
```
import javax.swing.*;
import java.awt.*;

public class LayoutExample {
    private JFrame f;
    private JButton b1;
    private JButton b2;
    private JButton b3;
    private JButton b4;
    private JButton b5;

    public LayoutExample() {
        f = new JFrame("GUI example");
        b1 = new JButton("Button 1");
        b2 = new JButton("Button 2");
        b3 = new JButton("Button 3");
        b4 = new JButton("Button 4");
        b5 = new JButton("Button 5");
    }
```

FlowLayout Example

```

    public void launchFrame() {
        f.setLayout(new FlowLayout());
        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);
        f.pack();
        f.setVisible(true);
    }

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

} // end of LayoutExample class
```

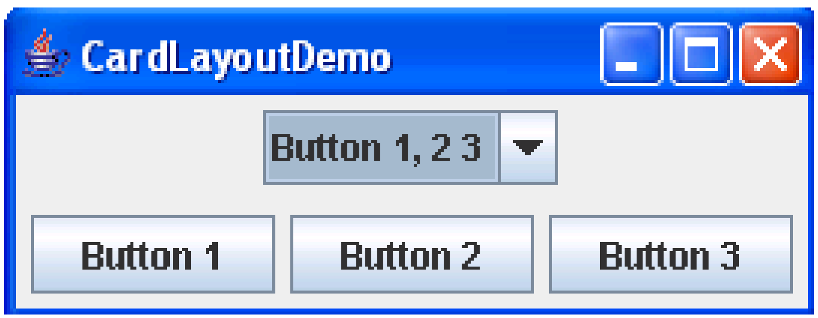
The BoxLayout Manager

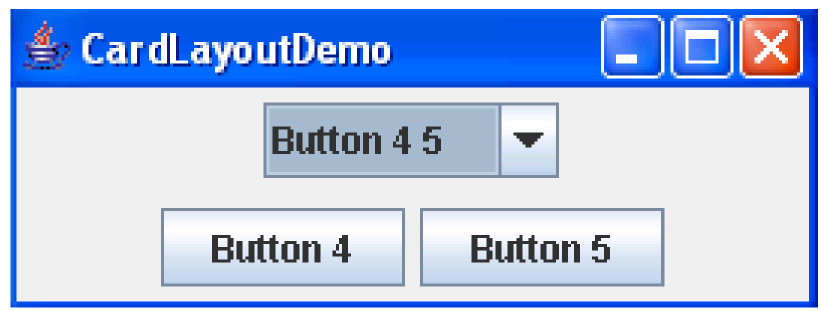
The `BoxLayout` manager adds components from left to right, and from top to bottom in a single row of column.



The CardLayout Manager

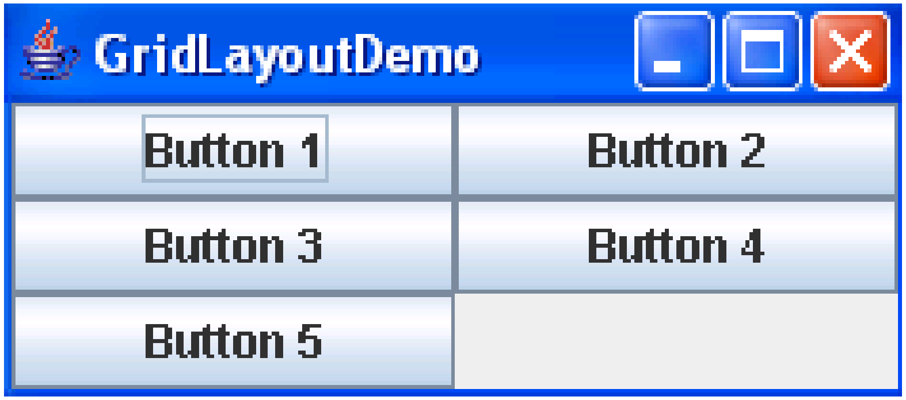
The `CardLayout` manager places the components in different cards. Cards are usually controlled by a combo box.





The `GridLayout` Manager

The `GridLayout` manager places components in rows and columns in the form of a grid.



GridLayout Example

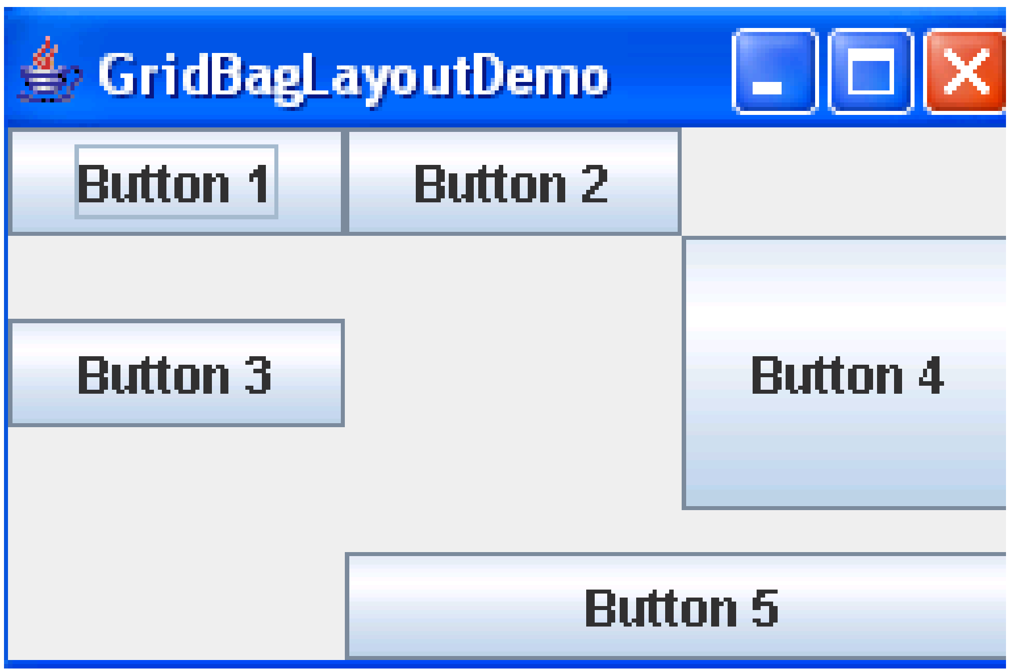
```
import java.awt.*;
import javax.swing.*;

public class GridExample {
  private JFrame f;
  private JButton b1, b2, b3, b4, b5;

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

```

GridLayout Example

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

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

The `GridBagLayout` Manager

The `GridBagLayout` manager arranges components in rows and columns, similar to a grid layout, but provides a wide variety of options for resizing and positioning the components.



GUI Construction

- Programmatic
- GUI builder tool

Programmatic Construction

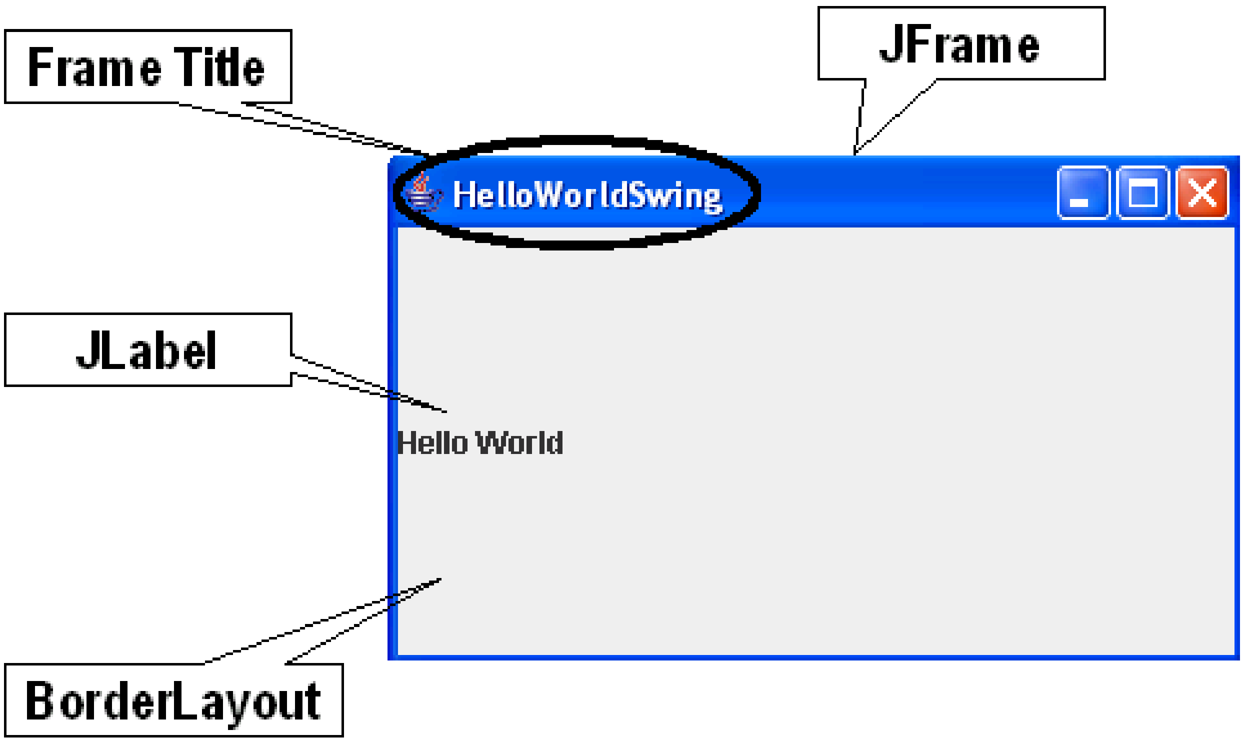
```
import javax.swing.*;
public class HelloWorldSwing {
    private static void createAndShowGUI() {
        JFrame frame = new JFrame("HelloWorldSwing");
      //Set up the window.
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel label = new JLabel("Hello World");
      // Add Label
        frame.add(label);
        frame.setSize(300,200);
      // Display Window
        frame.setVisible(true);}
```

Programmatic Construction

```
public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            //Schedule for the event-dispatching thread:
            //creating,showing this app's GUI.
           public void run() {createAndShowGUI();}
        });
    }
}
```

Programmatic Construction

The output generated from the program



Key Methods

Methods for setting up the `JFrame` and adding `JLabel`:

- `setDefaultCloseOperationJFrame.EXIT_ON_CLOSE)` –Creates the program to exit when the close button is clicked.
- `setVisible(true)` – Makes the `JFrame` visible.
- `add(label)` – `JLabel` is added to the content pane not to the `JFrame` directly.

Key Methods

- Tasks:
  - Executing GUI application code, such as rendering
  - Handling GUI events
  - Handling time consuming (background) processes
- The `SwingUtilities` class:
  - `SwingUtilites.invokeLater(new Runnable())`

Module 13

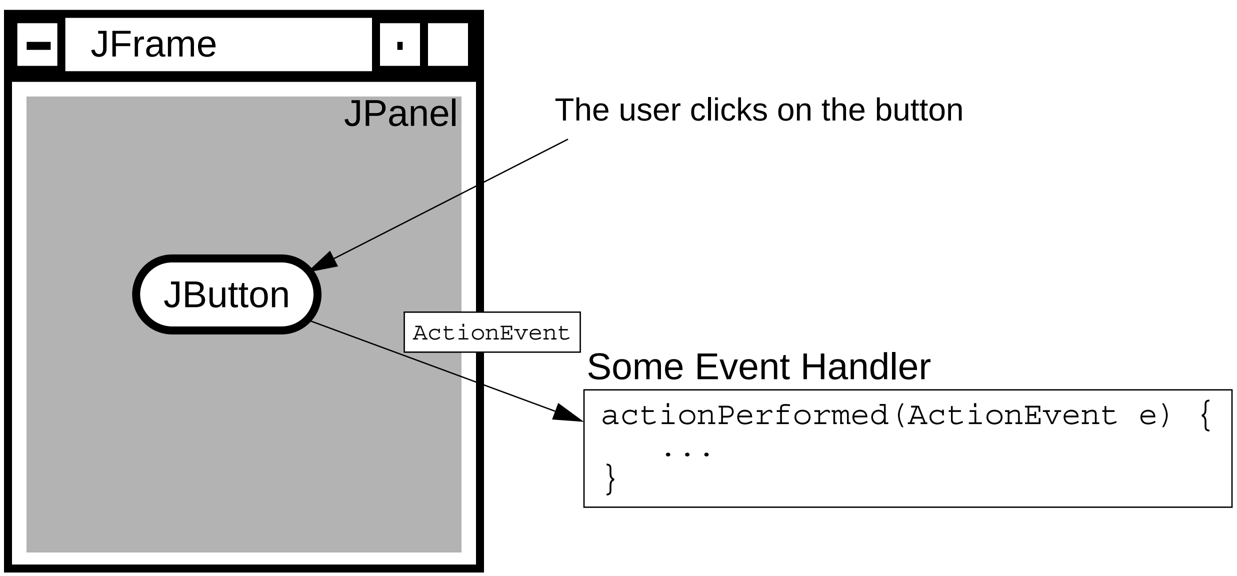
Handling GUI-Generated Events

Objectives

- Define events and event handling
- Examine the Java SE event model
- Describe GUI behavior
- Determine the user action that originated an event
- Develop event listeners
- Describe concurrency in Swing-based GUIs and describe the features of the `SwingWorker` class

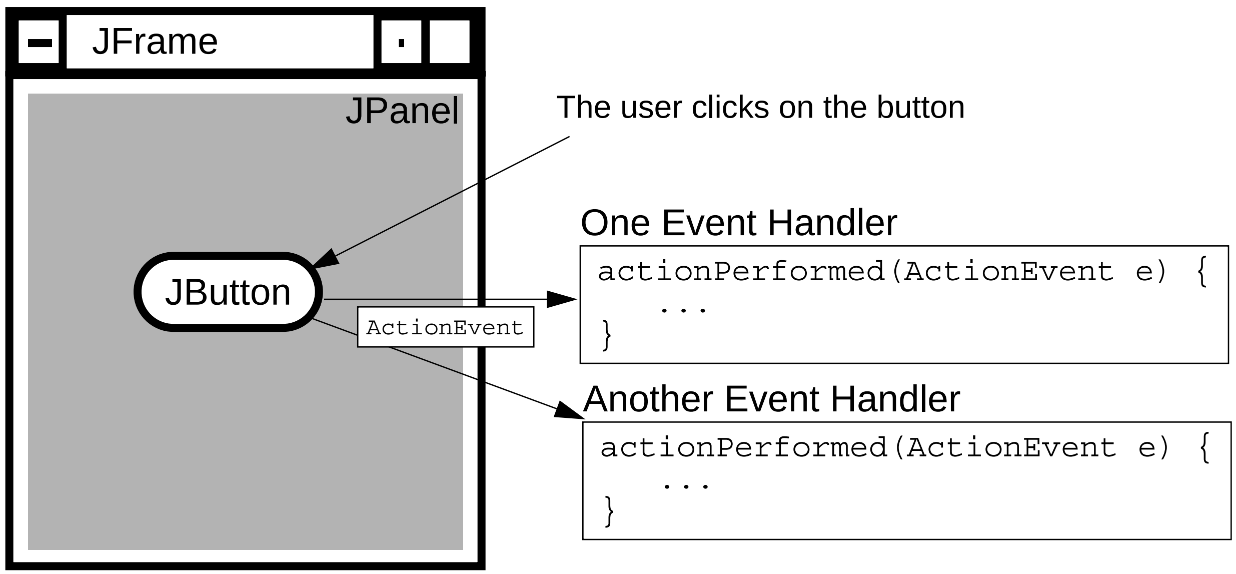
What Is an Event?

- 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

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

A Listener Example

```
import java.awt.*;
import javax.swing.*;
public class TestButton {
  private JFrame f;
  private JButton b;

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

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

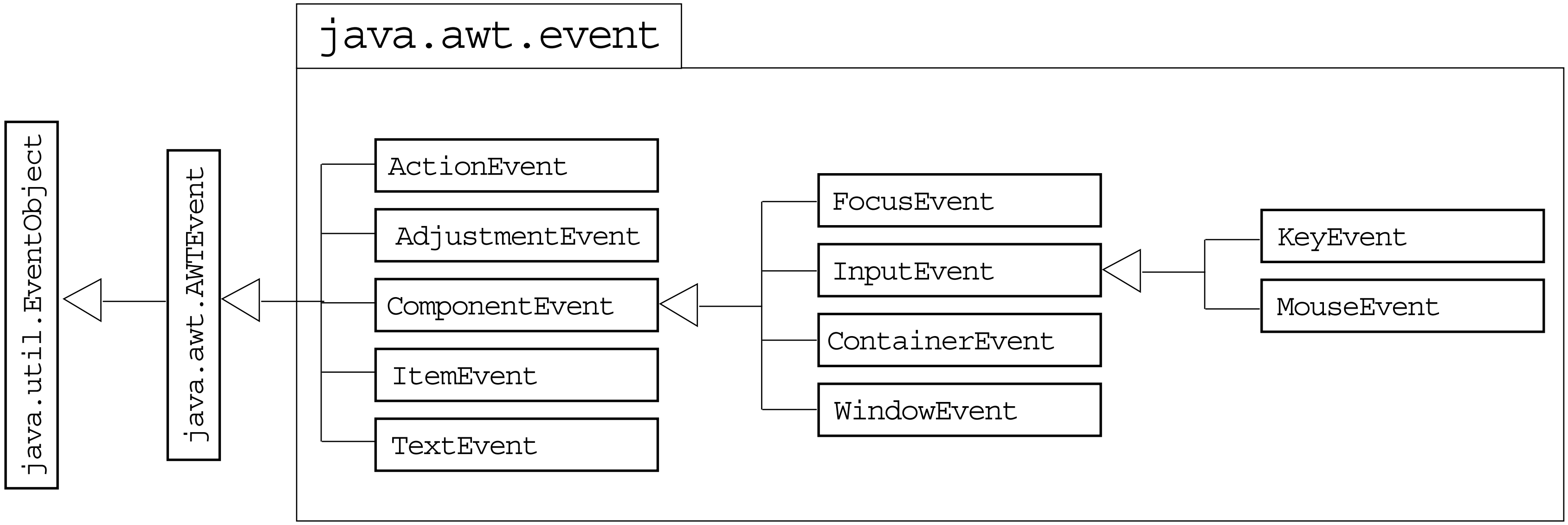
A Listener Example

```java

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

Code for the event listener looks like the following:

```java
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());
  }
}
```

Event Categories



Method Categories and Interfaces

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)`

Method Categories and Interfaces

Category	Interface Name	Methods
Focus	`FocusListener`	`focusGained(FocusEvent)` `focusLost(FocusEvent)`
Adjustment	`AdjustmentListener`	`adjustmentValueChanged` `(AdjustmentEvent)`

Component	`ComponentListener`	`componentMoved(ComponentEvent)`
		`componentHidden(ComponentEvent)`
		`componentResized(ComponentEvent)`
		`componentShown(ComponentEvent)`

Method Categories and Interfaces

Category	Interface Name	Methods
Window	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
Container	ContainerListener	componentAdded(ContainerEvent) componentRemoved (ContainerEvent)
Window state	WindowStateListener	windowStateChanged(WindowEvent e)
Window focus	WindowFocusListener	windowGainedFocus(WindowEvent e) windowLostFocus(WindowEvent e)

Method Categories and Interfaces

Category	Interface Name	Methods
Mouse wheel	`MouseWheelListener`	`mouseWheelMoved (MouseWheelEvent e)`
Input methods	`InputMethodListener`	`caretPositionChanged (InputMethodEvent e)` `inputMethodTextChnaged (InputMethodEvent e)`
Hierarchy	`HierarchyListener`	`hierarchyChanged (HierarchyEvent e)`
Hierarchy bounds	`HierarchyBoundsListener`	`ancestorMoved(HierarchyEvent e)` `ancestorResized(HierarchyEvent e)`
AWT	`AWTEventListener`	`eventDispatched(AWTEvent e)`
Text	`TextListener`	`textValueChanged(TextEvent)`

Complex Example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class TwoListener
        implements MouseMotionListener, MouseListener {
  private JFrame f;
  private JTextField tf;

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

Complex Example

```
  public void launchFrame() {
    JLabel label = new JLabel("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);
  }
```

Complex Example

```java

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

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

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

Complex Example

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

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

    public static void main(String args[]) {
      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.
- An example is:

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

public class MouseClickHandler extends MouseAdapter {

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

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

Event Handling Using Inner Classes

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class TestInner {
  private JFrame f;
  private JTextField tf; // used by inner class

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

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

Event Handling Using Inner Classes

```java

  public void launchFrame() {
    JLabel label = new JLabel("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);
  }

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

Event Handling Using Anonymous Classes

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

public class TestAnonymous {
  private JFrame f;
  private JTextField tf;

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

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

```

Event Handling Using Anonymous Classes

```
  public void launchFrame() {
    JLabel label = new JLabel("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);
  }
}
```

Concurrency In Swing

To handle a GUI efficiently, the Swing program needs different threads to:

- Execute the application code (current threads)
- Handle the events that arise from the GUI (event dispatch threads)
- Handle background tasks that might be time consuming (worker threads)

Each task in a worker thread is represented by an instance of `javax.swing.SwingWorker`.

The `SwingWorker` Class

The `SwingWorker` class has methods to service the following requirements:

- To provide communication and coordination between worker thread tasks and the tasks on other threads:
  - Properties: `state` and `progress`
- To execute simple background tasks:
  - `doInBackground` method
- To execute tasks that have intermediate results:
  - `publish` method
- To cancel the background threads:
  - `cancel` method

Module 14

GUI-Based Applications

Objectives

- 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?

How to Create a Menu

1. Create a `JMenuBar` object, and set it into a menu container, such as a `JFrame`.
2. Create one or more `JMenu` objects, and add them to the menu bar object.
3. Create one or more `JMenuItem` objects, and add them to the menu object.

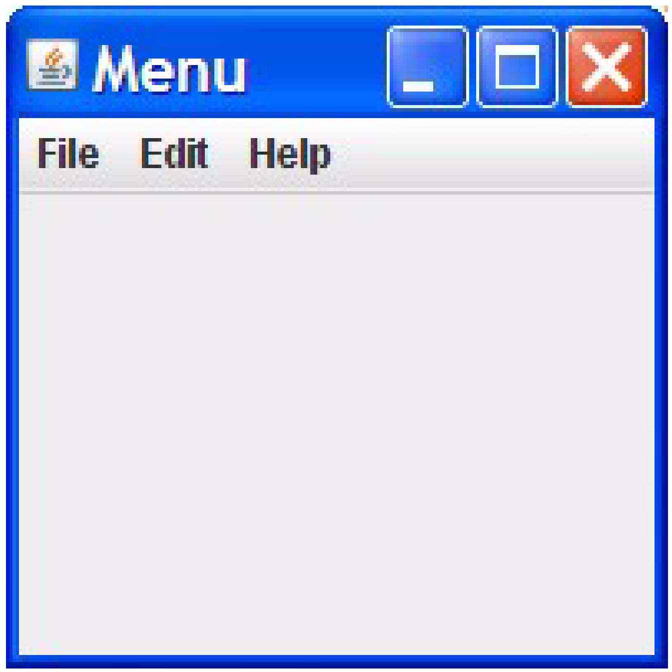
Creating a `JMenuBar`

```
f = new JFrame("MenuBar");
mb = new JMenuBar();
f.setJMenuBar(mb);
```

Creating a JMenu

```
f = new JFrame("Menu");
mb = new JMenuBar();
m1 = new JMenu("File");
m2 = new JMenu("Edit");
m3 = new JMenu("Help");
mb.add(m1);
mb.add(m2);
mb.add(m3);
f.setJMenuBar(mb);
```

Creating a `JMenu`



Creating a `JMenuItem`

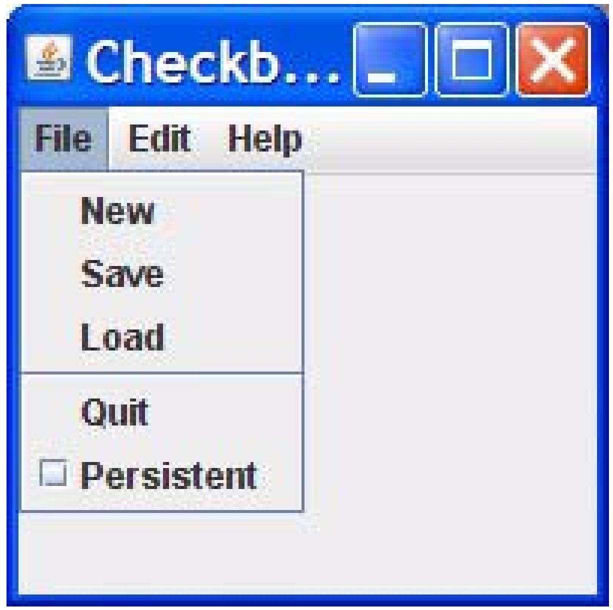
```
mi1 = new JMenuItem("New");
mi2 = new JMenuItem("Save");
mi3 = new JMenuItem("Load");
mi4 = new JMenuItem("Quit");
mi1.addActionListener(this);
mi2.addActionListener(this);
mi3.addActionListener(this);
mi4.addActionListener(this);
m1.add(mi1);
m1.add(mi2);
m1.add(mi3);
m1.addSeparator();
m1.add(mi4);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

Creating a `JMenuItem`

Creating a `JCheckBoxMenuItem`

```
f = new JFrame("CheckboxMenuItem");
mb = new JMenuBar();
m1 = new JMenu("File");
m2 = new JMenu("Edit");
m3 = new JMenu("Help");
mb.add(m1);
mb.add(m2);
mb.add(m3);
f.setJMenuBar(mb);
...........
mi5 = new JCheckBoxMenuItem("Persistent");
mi5.addItemListener(this);
m1.add(mi5);
```

Creating a `JCheckBoxMenuItem`



Controlling Visual Aspects

Commands to control visual aspects of the GUI include:

- Colors:

```
setForeground()
setBackground()
```

- Example:

```
Color purple = new Color(255, 0, 255);
JButton b = new JButton("Purple");
b.setBackground(purple);
```

Module 15

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

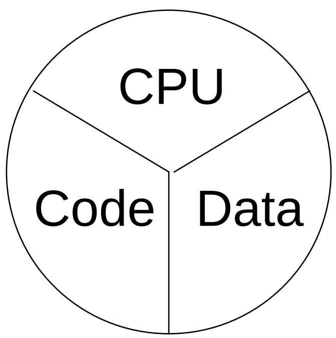
Relevance

How do you get programs to perform multiple tasks concurrently?

Threads

- What are threads?

  Threads are a virtual CPU.
- The three parts of at thread are:
  - CPU
  - Code
  - Data



A thread or
execution context

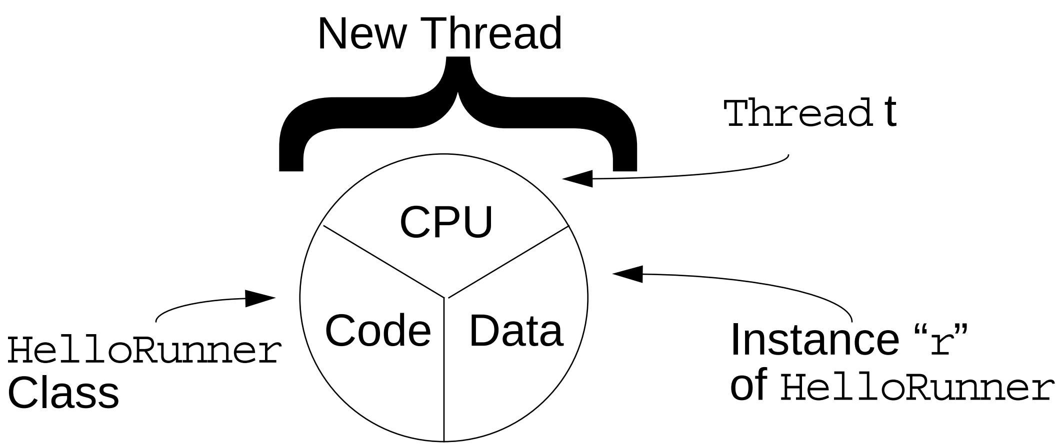
Creating the Thread

```
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 has these characteristics:
  - Multiple threads are from one `Runnable` instance.
  - Threads share the same data and code.
- For example:

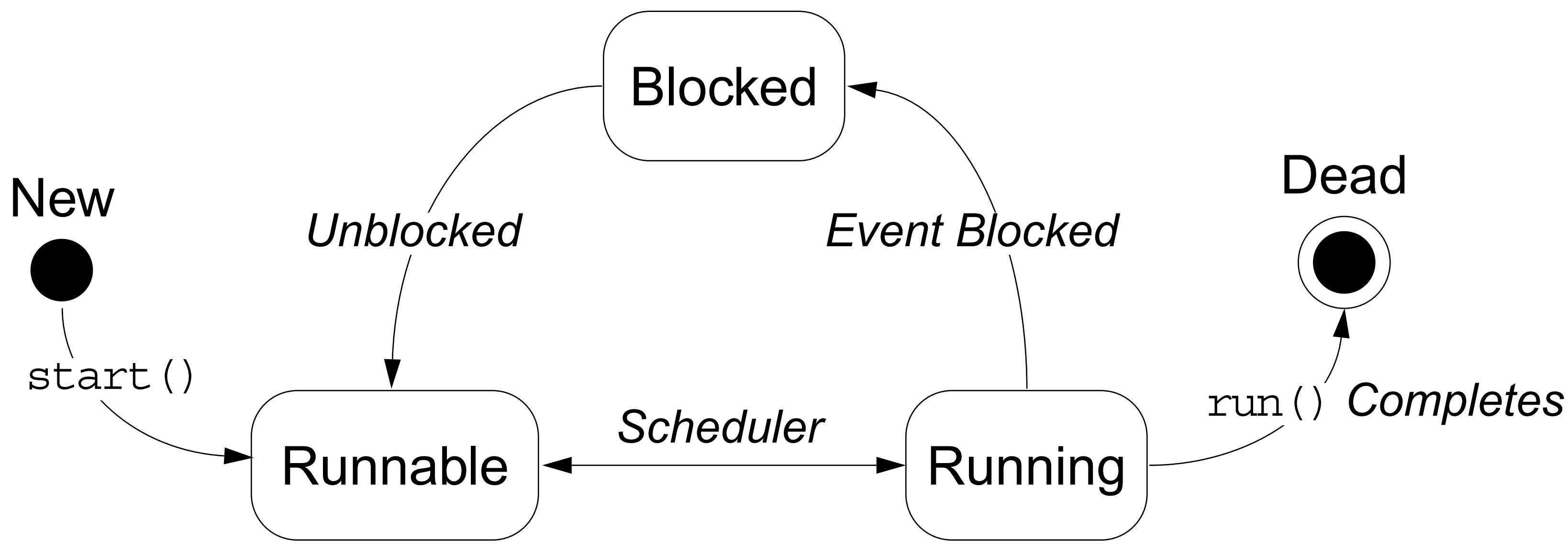
```
Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
```



Starting the Thread

- Use the `start` method.
- Place the thread in a runnable state.

Thread Scheduling



Thread Scheduling Example

```
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 ) {
      // continue doing work
    }
    // clean up before run() ends
  }

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

Terminating a Thread

```
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

- Test threads:

```
isAlive()
```

- Access thread priority:

```
getPriority()
setPriority()
```

- Put threads on hold:

```
Thread.sleep()  // static method
join()
Thread.yield()  // static method
```

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 other thread to finish
  try {
    t.join();
  } catch (InterruptedException e) {
    // the other thread came back early
  }
  ...
  // Now continue in this thread
  ...
}
```

Other Ways to Create Threads

```
public class MyThread extends Thread {
  public void run() {
    while ( true ) {
      // do lots of interesting stuff
      try {
        Thread.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

- Implement `Runnable`:
  - Better object-oriented design
  - Single inheritance
  - Consistency
- Extend `Thread`:

  Simpler code

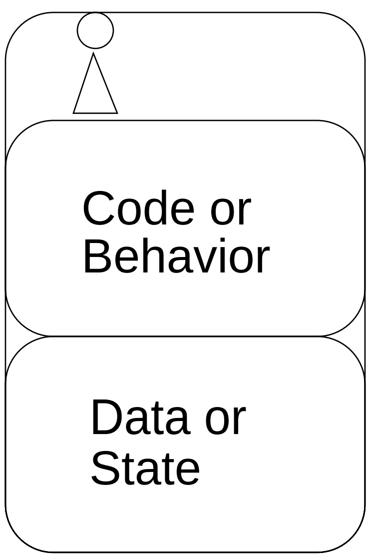
Using the `synchronized` Keyword

```java
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 is a type of *lock flag*.
- The `synchronized` enables interaction with the lock flag.

Object `this`



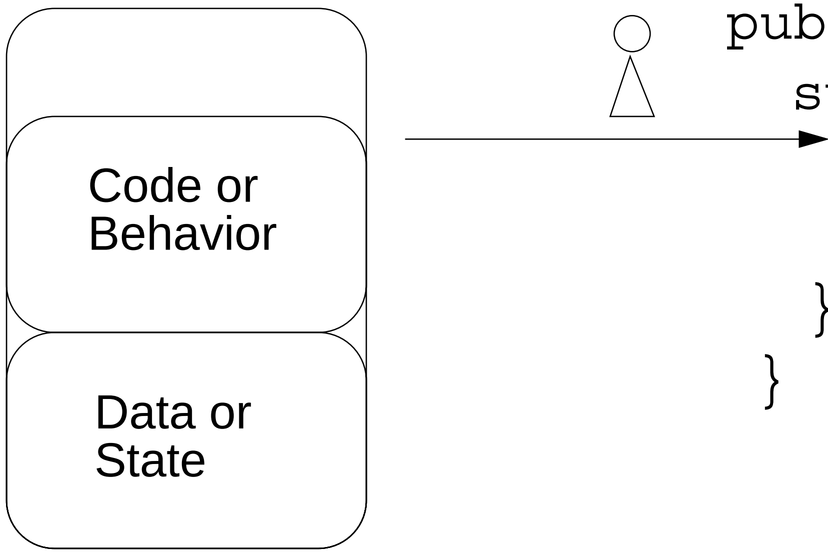
Thread before `synchronized(this)`

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

The Object Lock Flag

Object `this`

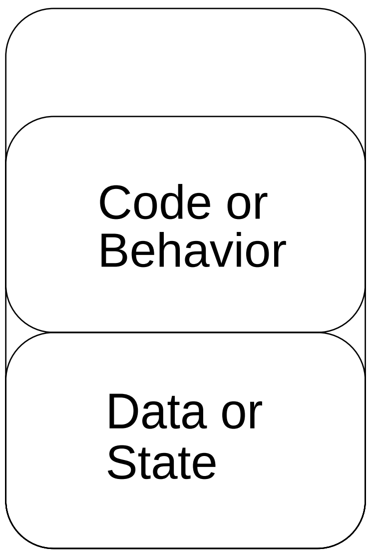
Thread after `synchronized(this)`




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

The Object Lock Flag

Object `this`
lock flag missing



Another thread, trying to
execute `synchronized(this)`

Waiting for
object lock

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

Releasing the Lock Flag

The lock flag is released in the following events:

- Released when the thread passes the end of the `synchronized` code block
- Released automatically when a break, return, or exception is thrown by the `synchronized` code block

Using `synchronized` – Putting It Together

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

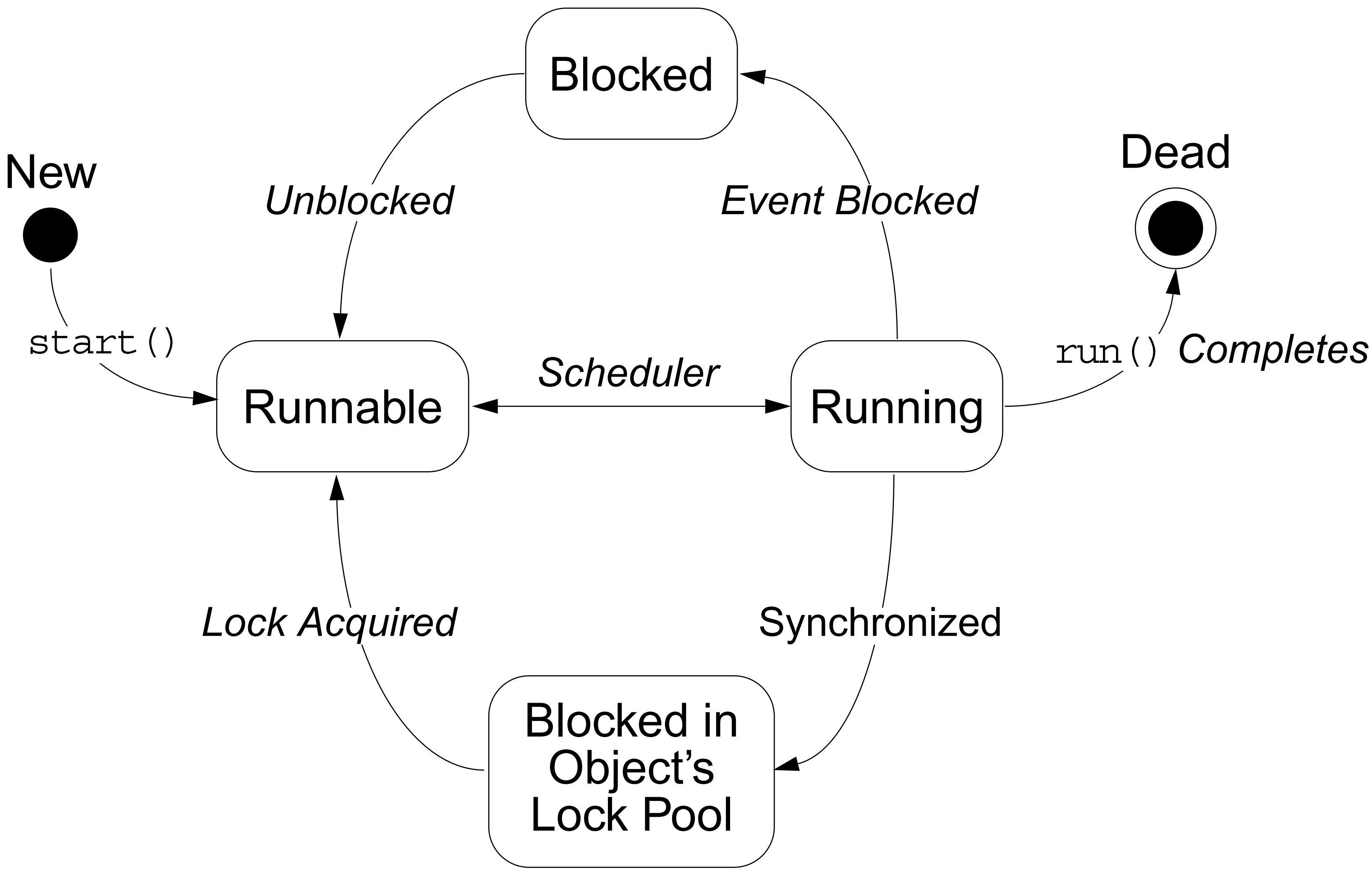
Using `synchronized` – Putting It Together

The following two code segments are equivalent:

```
public void push(char c) {
  synchronized(this) {
    // The push method code
  }
}

public synchronized void push(char c) {
  // The push method code
}
```

Thread State Diagram With

Synchronization



Deadlock

A deadlock has the following characteristics:

- It is two threads, each waiting for a lock from the other.
- It is not detected or avoided.
- Deadlock 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 do you determine when you are at your destination?
- The solution:
  - You notify the cab driver of your destination and relax.
  - The driver drives and notifies you upon arrival at your destination.

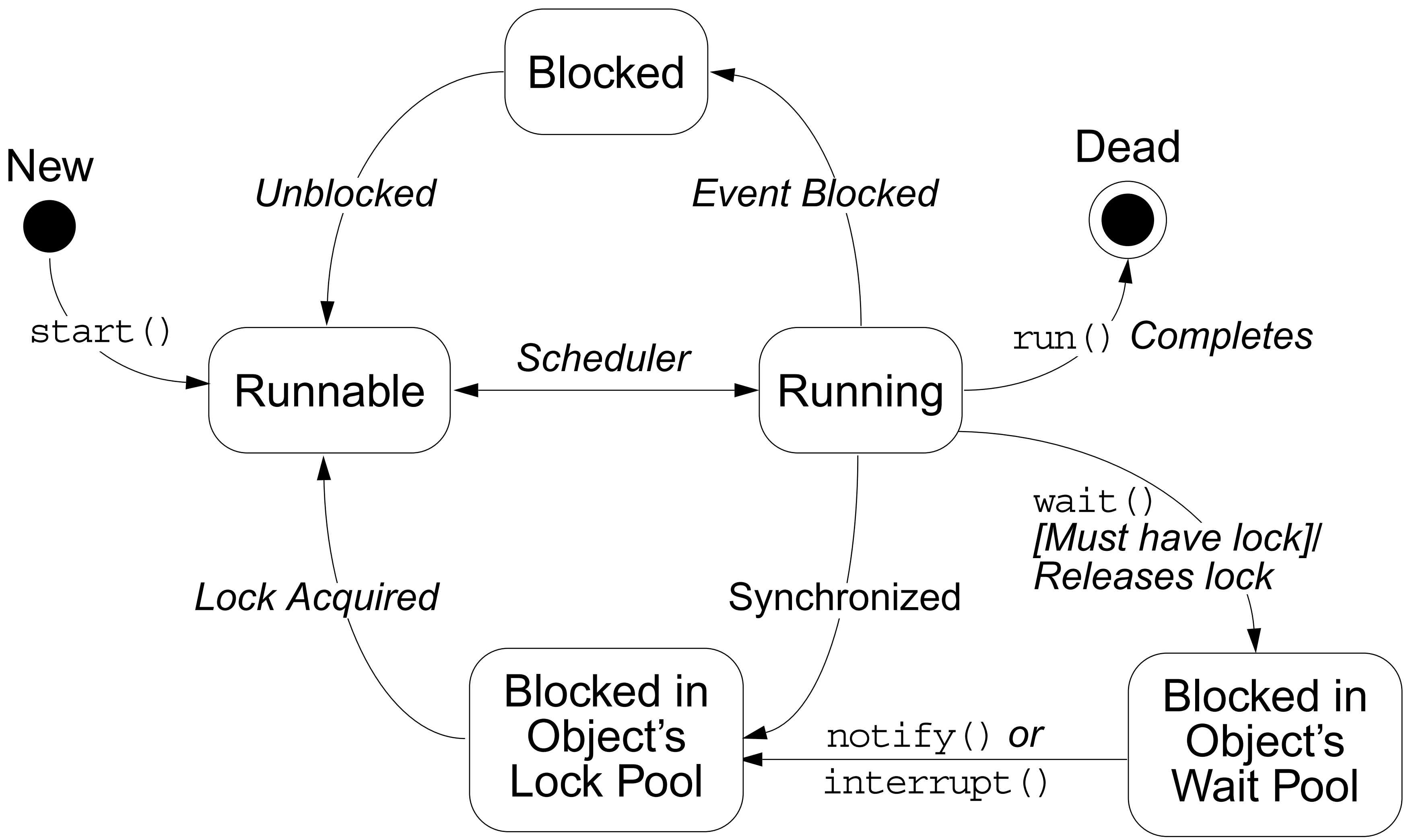
Thread Interaction

Thread interactions include:

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

Thread 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.

The `Producer` Class

```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++;
  }

```

The `Producer` Class

```
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
      }
    }
  } // END run method

} // END Producer class
```

The Consumer Class

```
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++;
  }

```

The Consumer Class

```
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) {
        // ignore it
      }
    }
  } // END run method

```

The `SyncStack` Class

This is a sketch of the `SyncStack` class:

```
public class SyncStack {

  private List<Character> buffer = new ArrayList<Character>(400);

  public synchronized char pop() {
    // pop code here
  }

  public synchronized void push(char c) {
    // push code here
  }
}
```

The pop Method

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

```

The `push` Method

```
public synchronized void push(char c) {
    this.notify();
    buffer.add(c);
  }
```

The SyncTest Class

```
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();
  }
}
```

The SyncTest Class

```
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
```

Module 16

Networking

Objectives

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

Relevance

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

Networking

This section describes networking concepts.

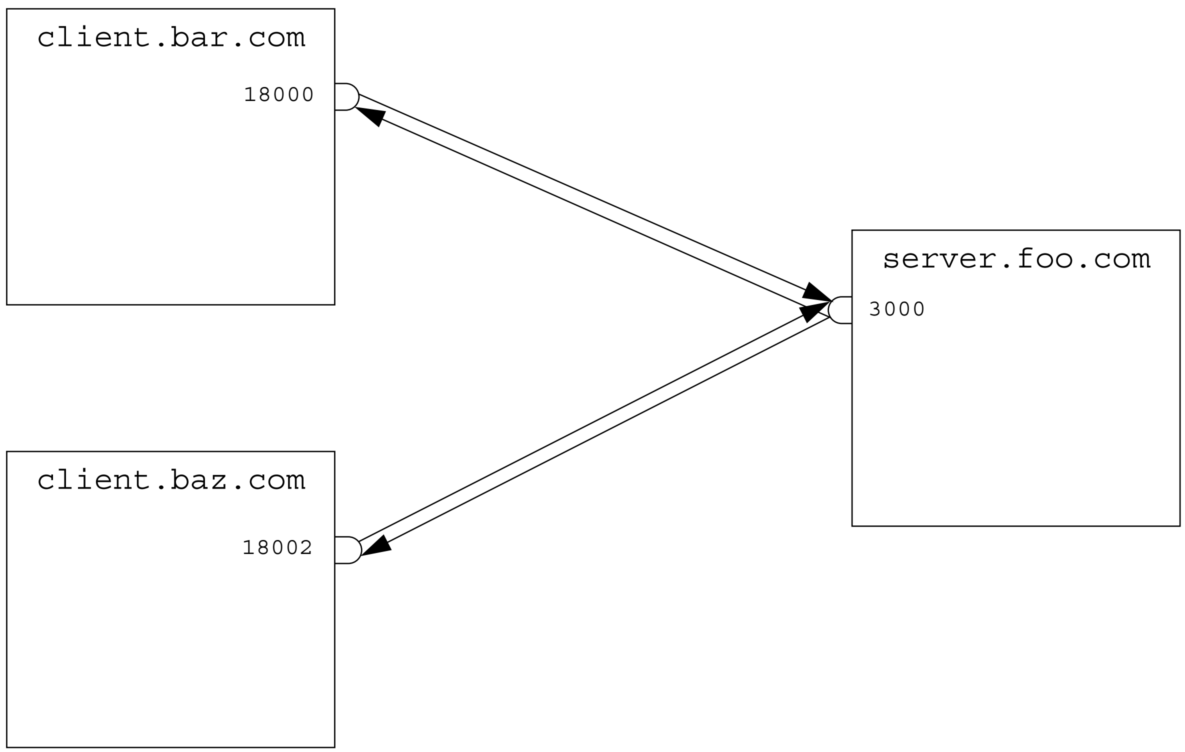
Sockets

- Sockets hold two streams: an input stream and an output stream.
- Each end of the socket has a pair of streams.

Setting Up the Connection

Set up of a network connection is similar to a telephone system: One end must *dial* the other end, which must be *listening*.

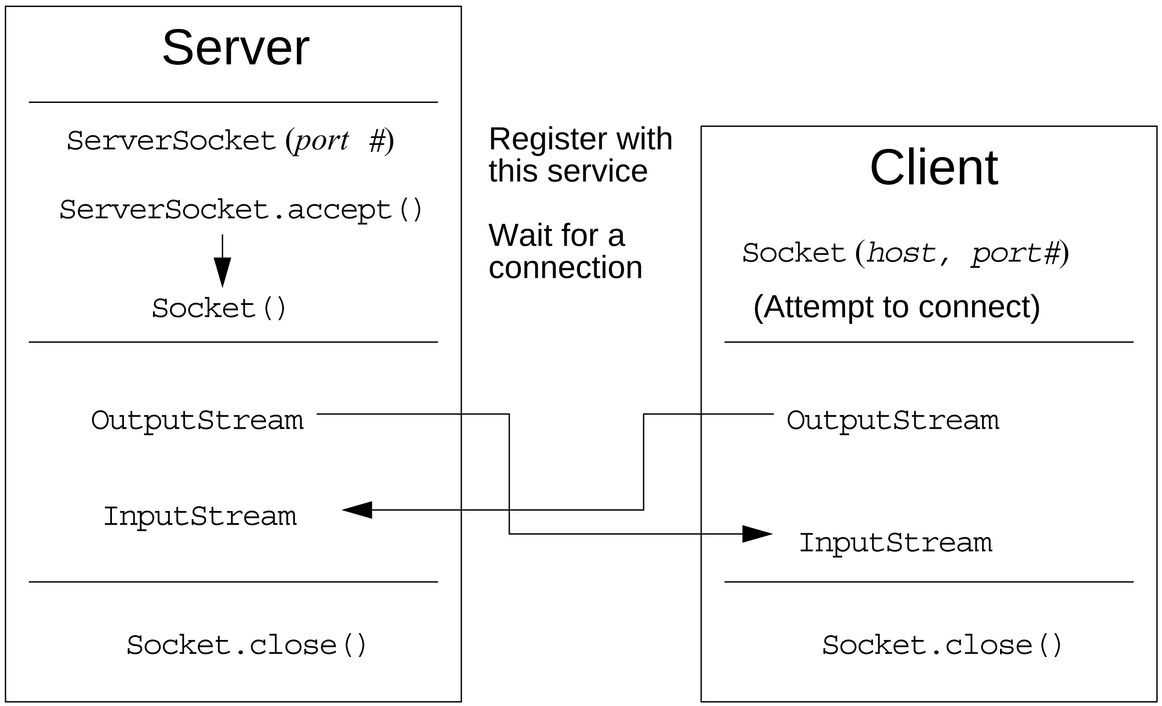
Networking


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

Networking With Java Technology

- To address the connection, include the following:
  - The address or name of remote machine
  - A port number to identify the purpose at the server
- Port numbers range from 0–65535.

Java Networking Model



Minimal TCP/IP Server

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

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

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

Minimal TCP/IP Server

```

    // 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”);
```

Minimal TCP/IP Server

```
        // Close the connection, but not the server socket
        bw.close();
        s1.close();

      } catch (IOException e) {
        e.printStackTrace();
      } // END of try-catch

    } // END of while(true)

  } // END of main method

} // END of SimpleServer program
```

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);
```

Minimal TCP/IP Client

```java

        // 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.");

      } catch (IOException e) {
        // ignore
      } // END of try-catch

    } // END of main method

  } // END of SimpleClient program
```