Fresher Program
Java Fundamentals For Mobil

Duong Cong Hoan
SSE

July, 2013

Warm up - Introductions

- Your role
- Your background, experiences in the subject
- What do you want from this course

Course Objectives

At the end of the course, you will have acquired sufficient knowledge to:

- Create Java™ technology applications that leverage the object-oriented features of the Java language
- 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

Agenda

- Session 1:
  - Getting Started
  - Developing and Testing a Java Technology Program
  - Object-Oriented Programming
  - Identifiers, Keywords, and Types
  - Creating and Using Objects
- Session 2:
  - Using Operators and Decision Constructs
  - Using Loop Constructs
  - Arrays
  - Advanced Class Features

Agenda

- Session 3:
  - Advanced Class Features (cont)
  - Exceptions and Assertions
  - Text-Based Applications

Course Audience and Prerequisite

- The course is for any one who wants to learn Java

Assessment Disciplines

- Class Participation: at least 80% of course time
- Assignment: complete 80% of the exercises and send to me in a zip file before a new session begins

Duration and Course Timetable

- Course Duration: 9 Hours in 3 sessions
- Break 15 minutes each session

References

This course composed based on the following documents

- SL-110: Fundamentals of the Java™ Programming Language (Revision D2)
- SL-275: Java™ Programming Language (Revision F)
- Graphical model in slide 55 of the presentation “Visual Modeling and the UML” by Nguyet Pham

Set Up Environment

- To complete the course, your PC must install:
  - JDK5.0
  - J2SE API Spec. (optional)
  - http://java.sun.com/j2se/1.5.0/docs/api/

Notations

- Normal text: Font Arial, black color
- Emphasized text: Font Arial, red color
- Computer code: Font Courier, size 24

```
public class Sample {
   final int months = 12;
   public void mymethod() {
   }
   public void yourmethod() {
   }
}
```

Course Administration

- In order to complete the course you must:
  - Sign in the Class Attendance List
  - Participate in the course
  - Provide your feedback in the End of Course Evaluation

Getting Started: Objectives

- Describe the key features of Java technology
- Identifying Java Technology Product Groups
- 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
- Using the J2SE5.0 JDK Components

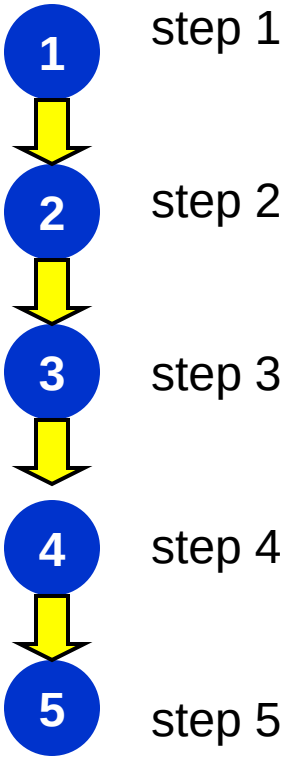
Table of Contents

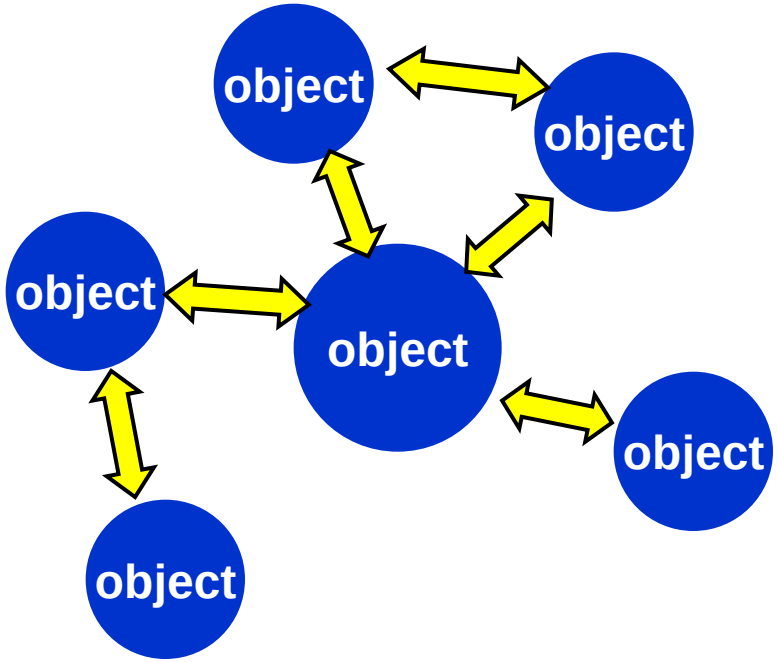
Key Concepts of the Java Programming Language

- Object-oriented
- Distributed
- Simple
- Multithreaded
- Secure
- Platform-independent

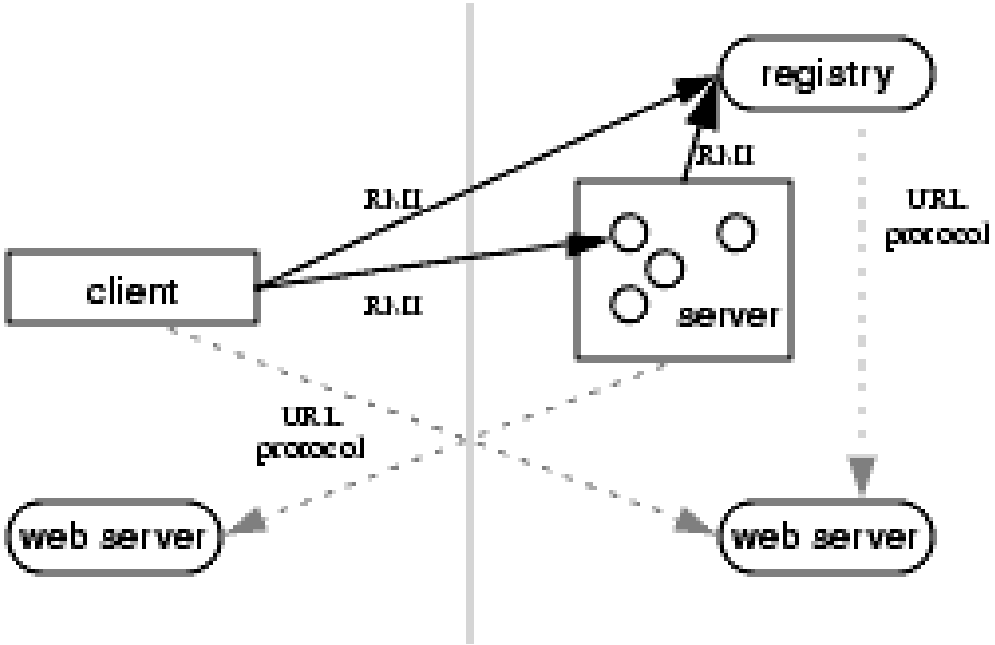
Object Oriented

procedural programming vs. object-oriented





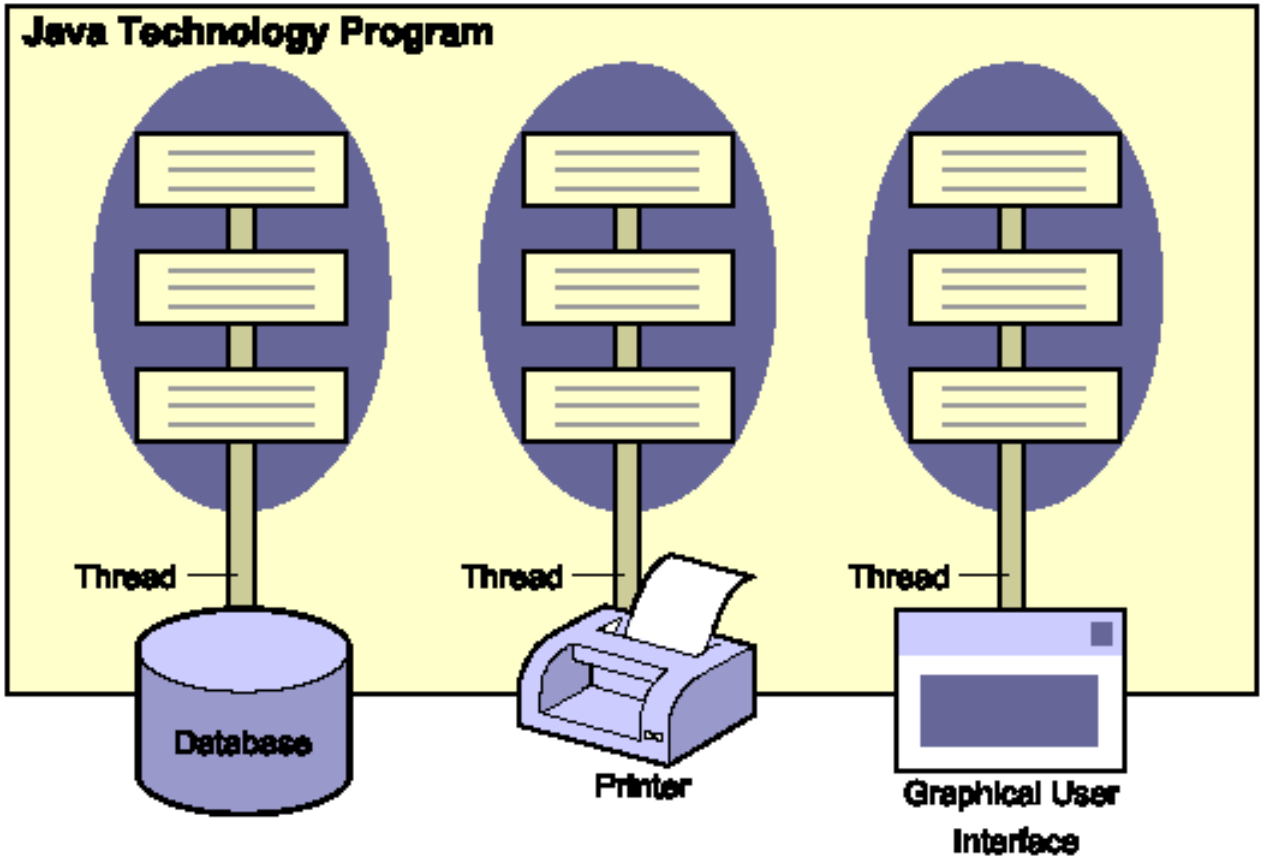
Distributed



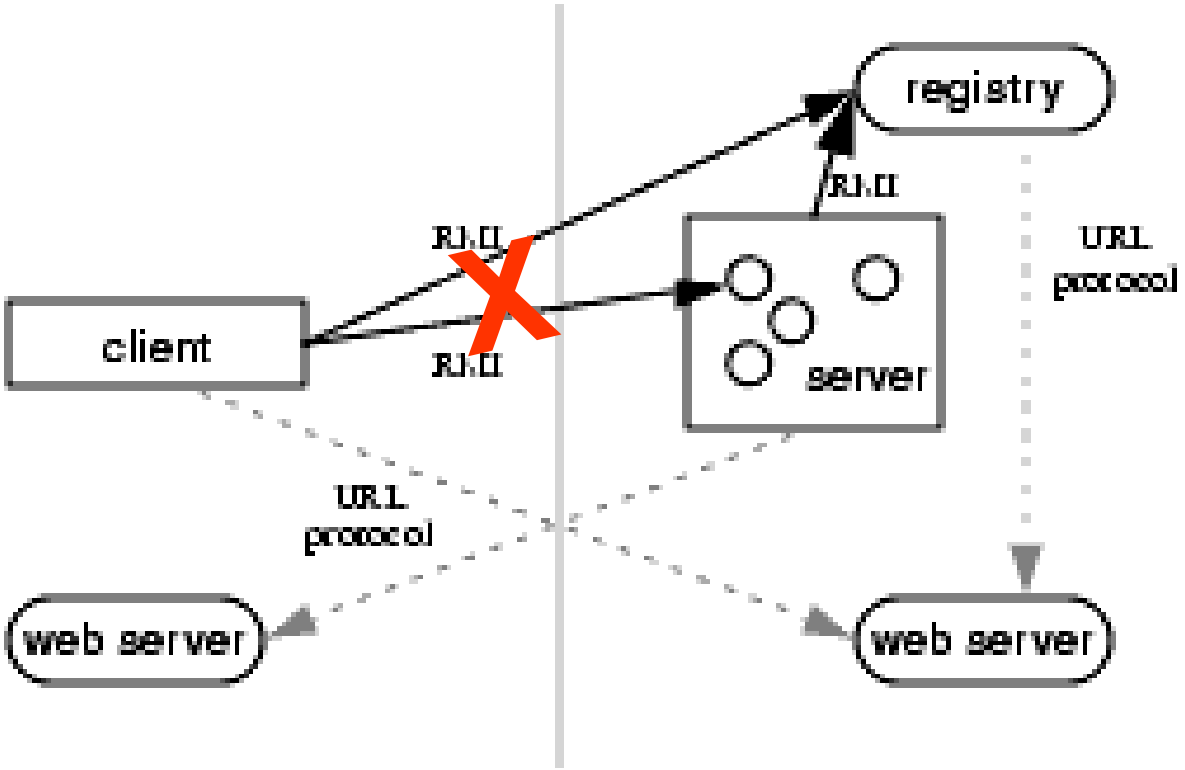
Simple

- References are used instead of pointers.
- A boolean data type can have a value of either true or false.

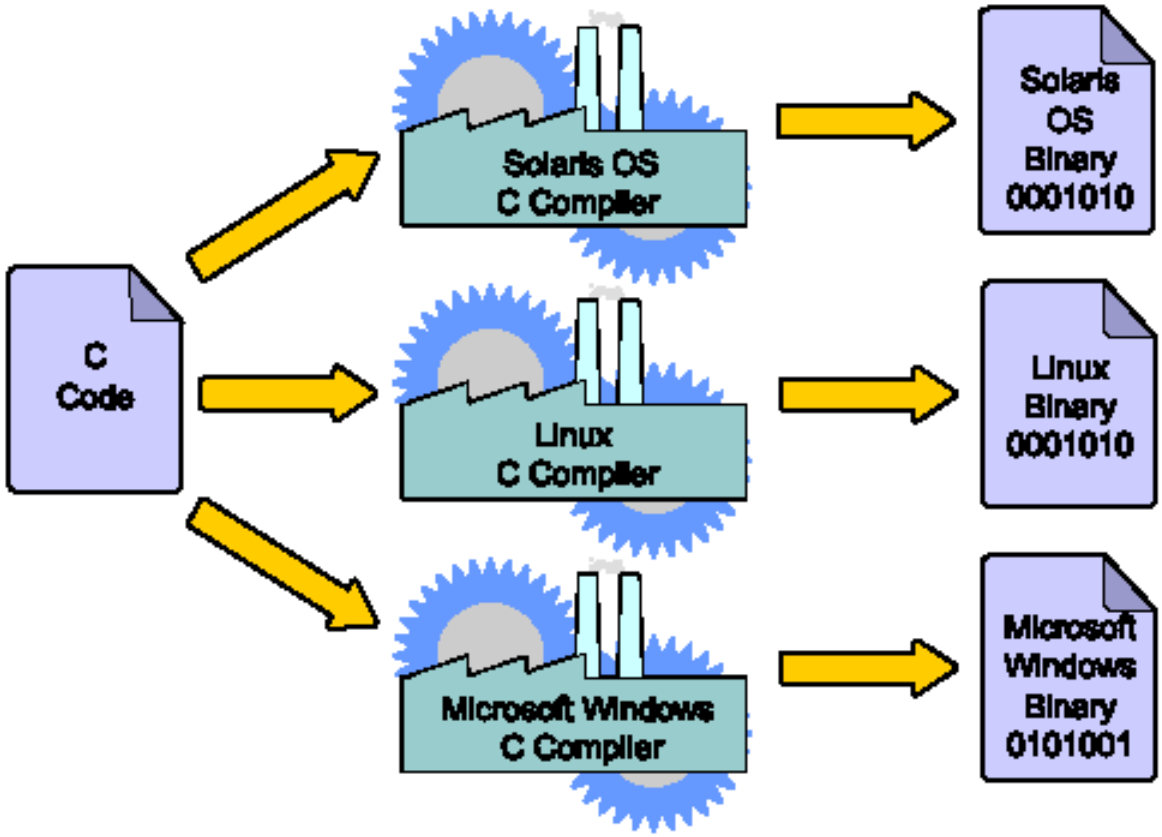
Multithreaded



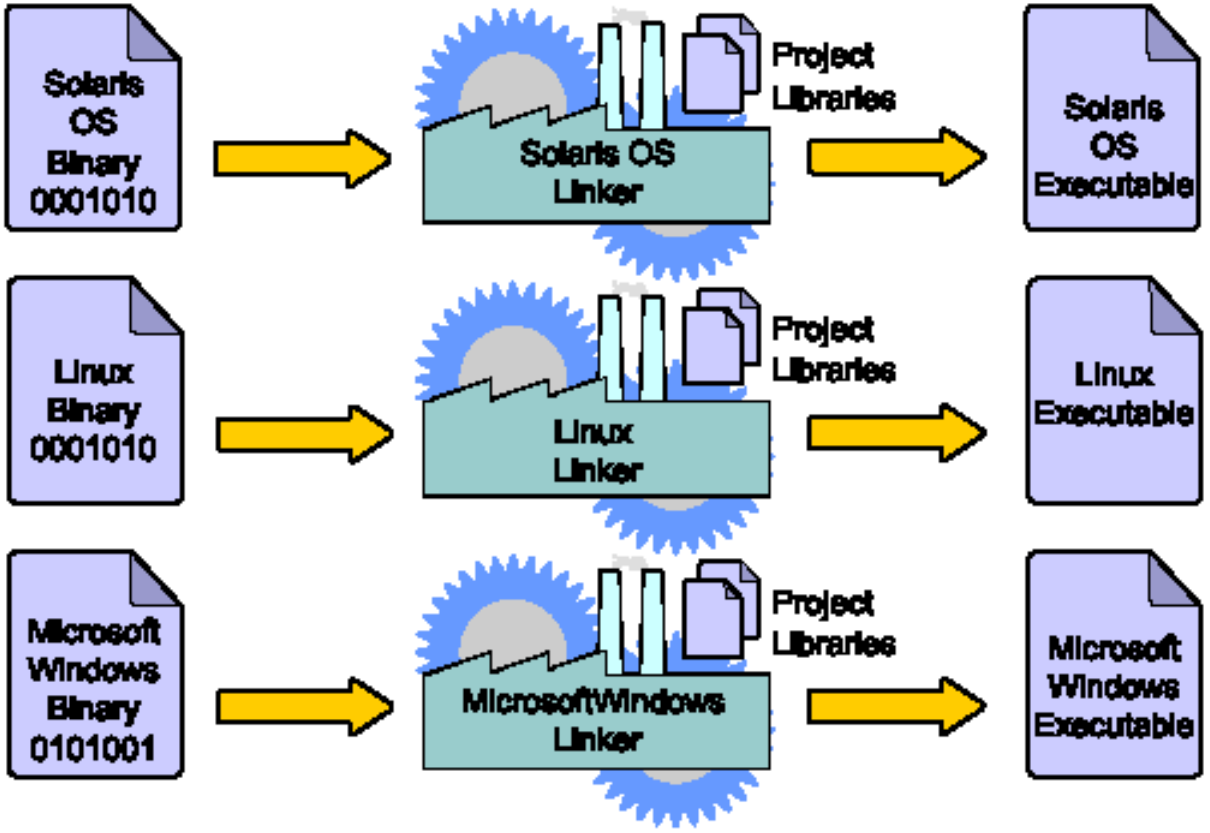
Secure


registry
RMI
RMI
RMI
URL
protocol
client
server
URL
protocol
web server
web server

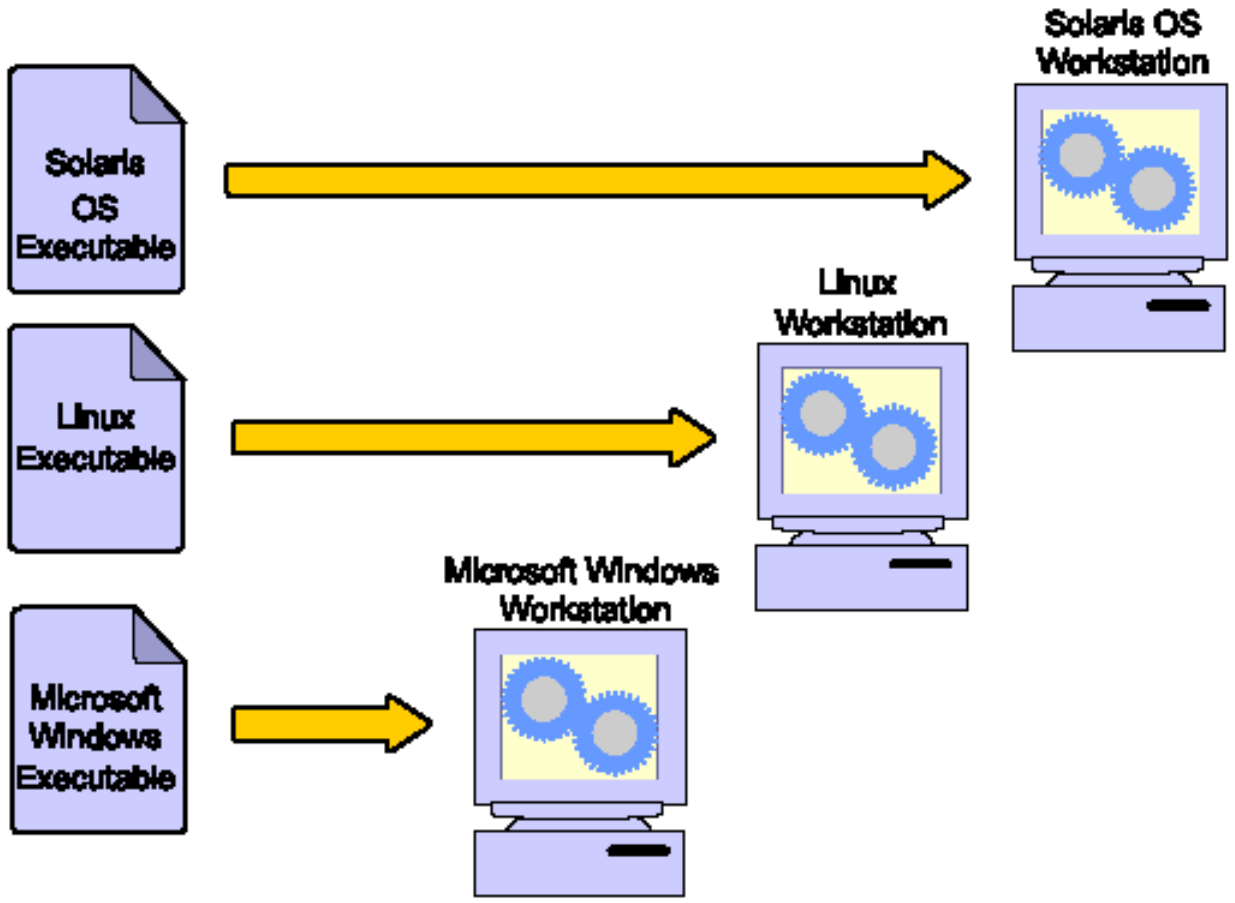
Platform-Dependent Programs



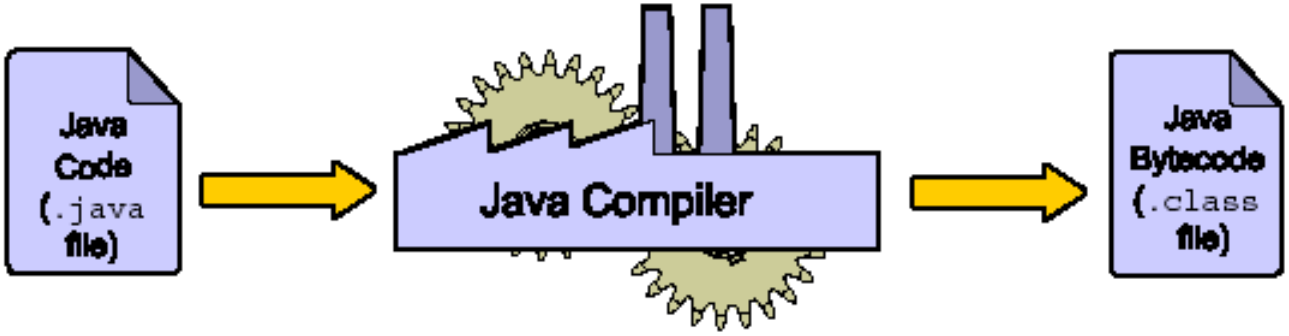
Platform-Dependent Programs (cont.)



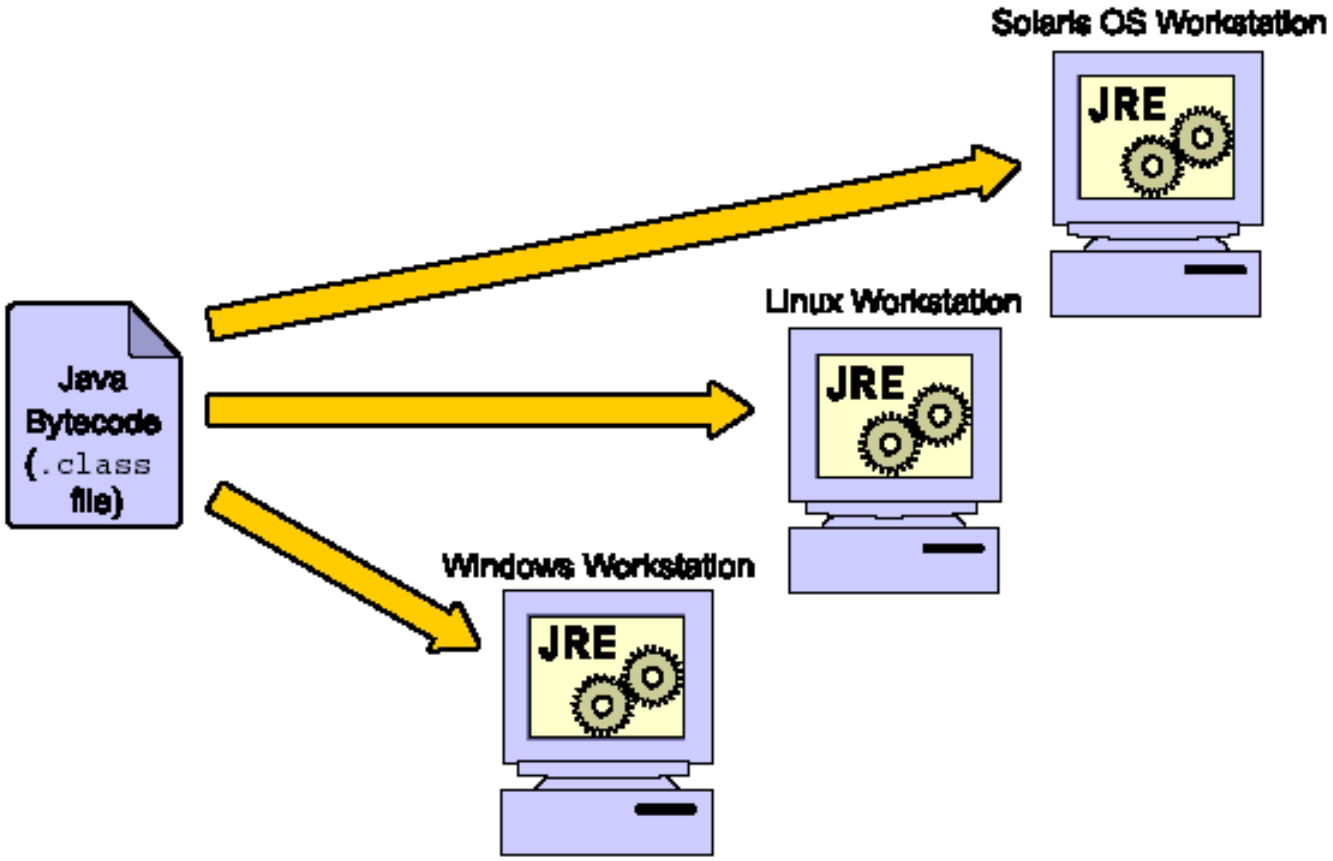
Platform-Dependent Programs (cont.)



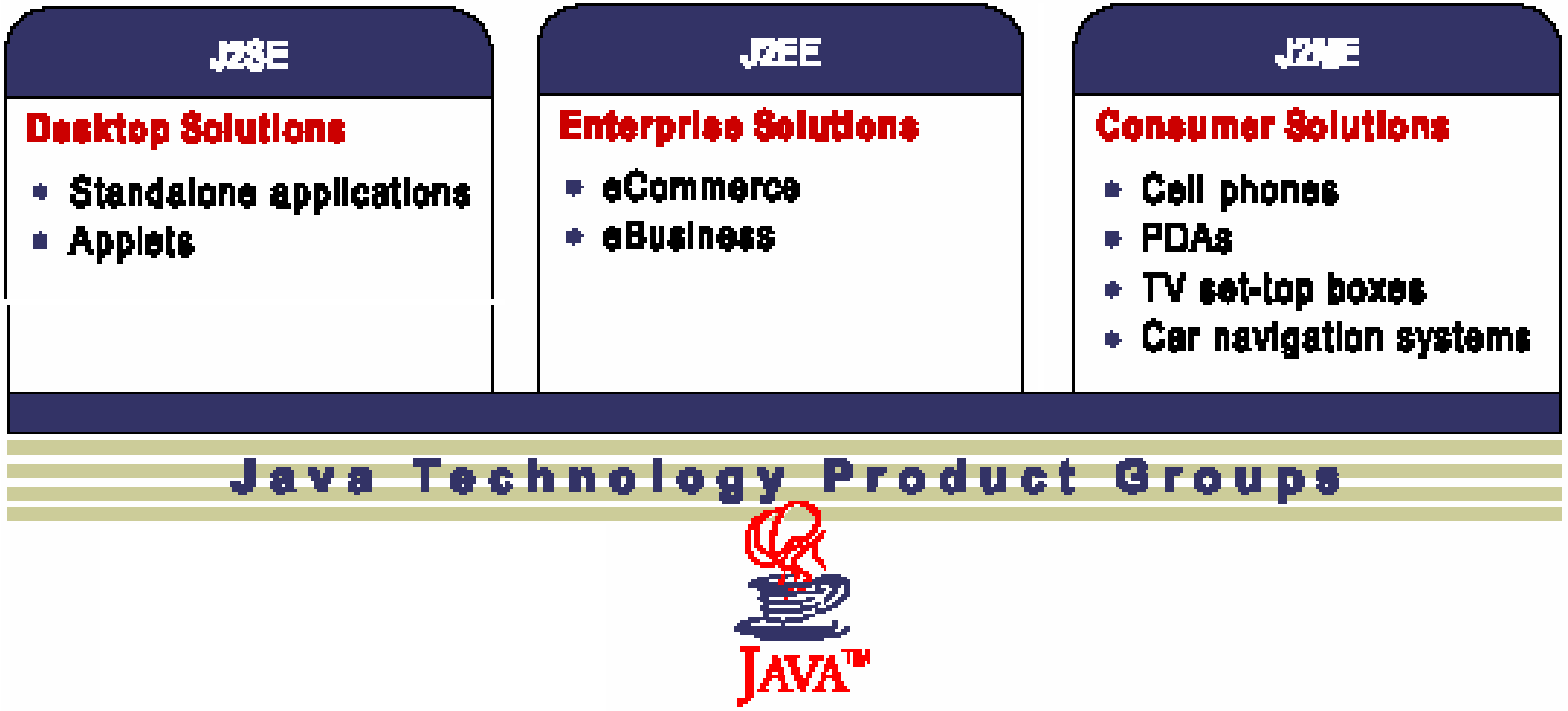
Platform-Independent Programs



Platform-Independent Programs



Java Technology Product Groups



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.

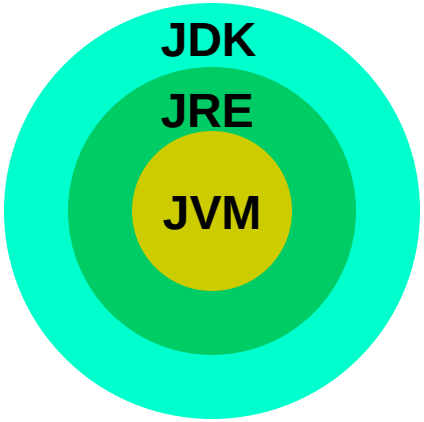
Primary Goals of the Java Technology

- 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

The following features fulfill these goals:

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



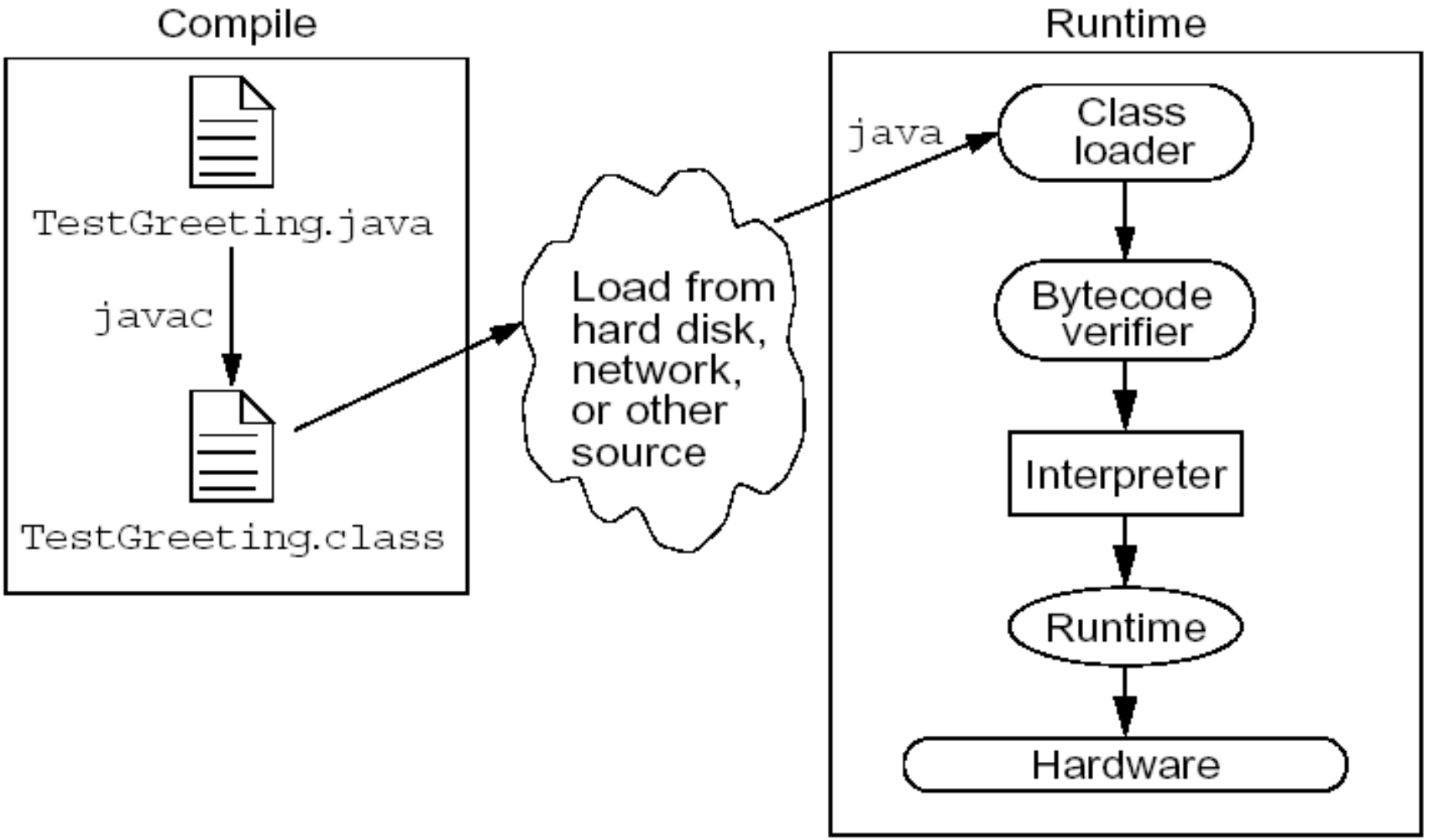
The Java Virtual Machine

- 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 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.

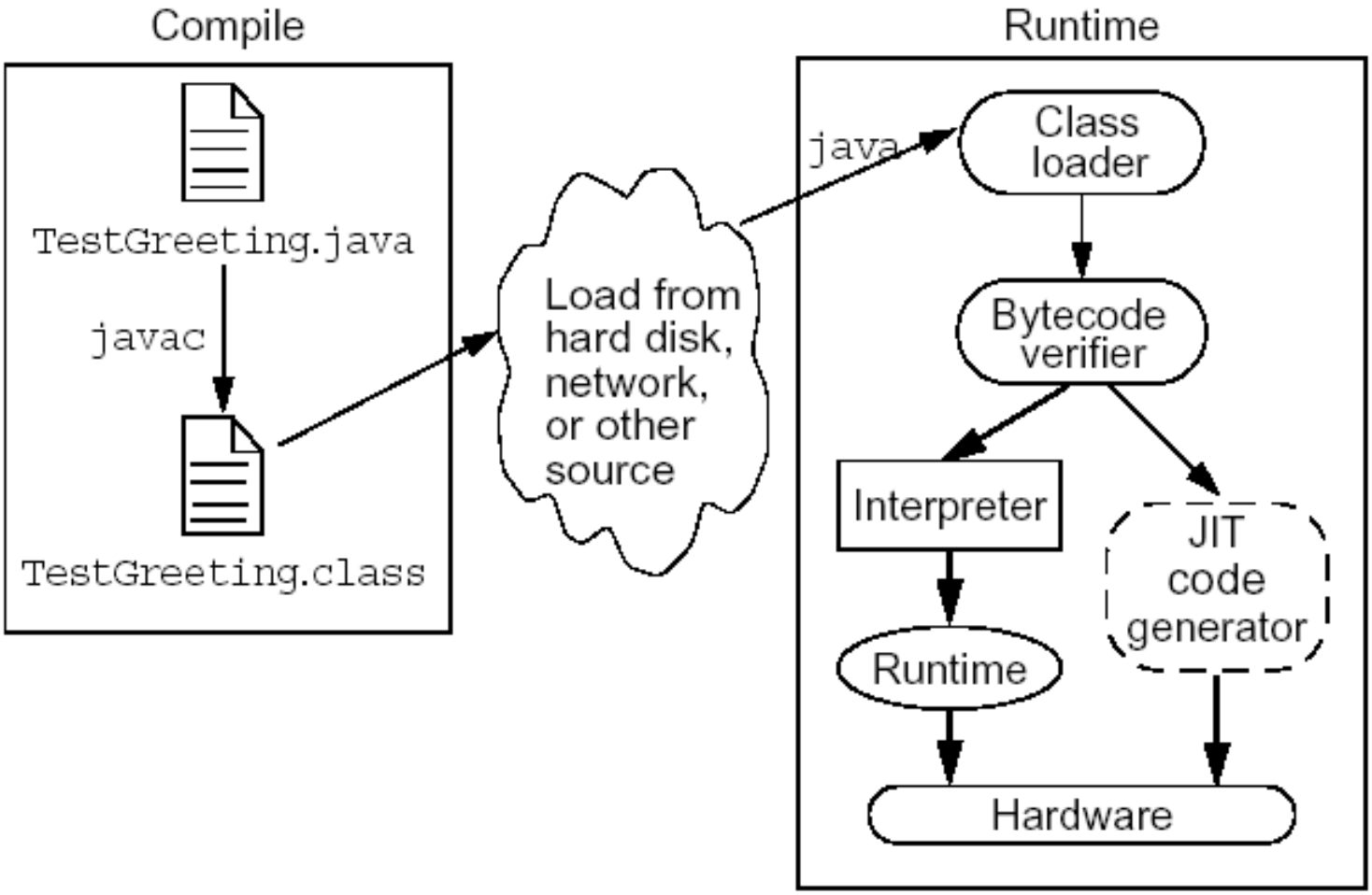
Garbage Collection

- 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



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



A Simple Java Application

The `TestGreeting.java` Application

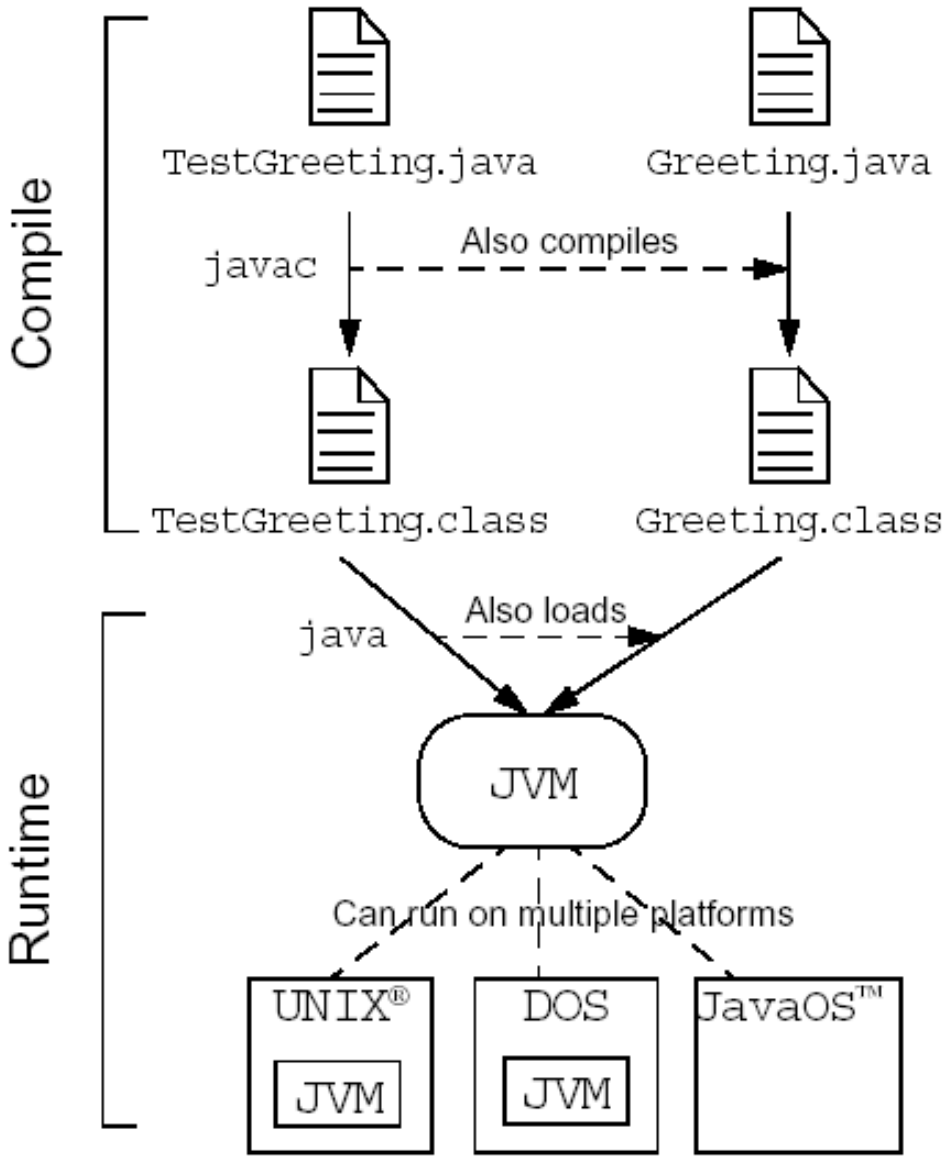
```
1 //
2 // Sample "Hello World" application
3 //
4 public class TestGreeting{
5    public static void main (String[] args)
  {
6       Greeting hello = new Greeting();
7       hello.greet();
8    }
9 }
```

A Simple Java Application

The `Greeting.java` Class

```
1 public class Greeting {
2   public void greet() {
3     System.out.println(“hi”);
4   }
5 }
```

Java Technology Runtime Environment



Compile-Time Errors

- `javac: Command not found`

  check the path environment variable

- ```
  Greeting.java:4: cannot resolve symbol
  symbol : method printl (java.lang.String)
  location: class java.io.PrintStream
  System.out.printl("hi");
                    ^
  ```

  check method naming

- ```
  TestGreet.java:4: Public class TestGreeting
  must be defined in a file called
  "TestGreeting.java".
  ```

  check file name against class name

Runtime Errors

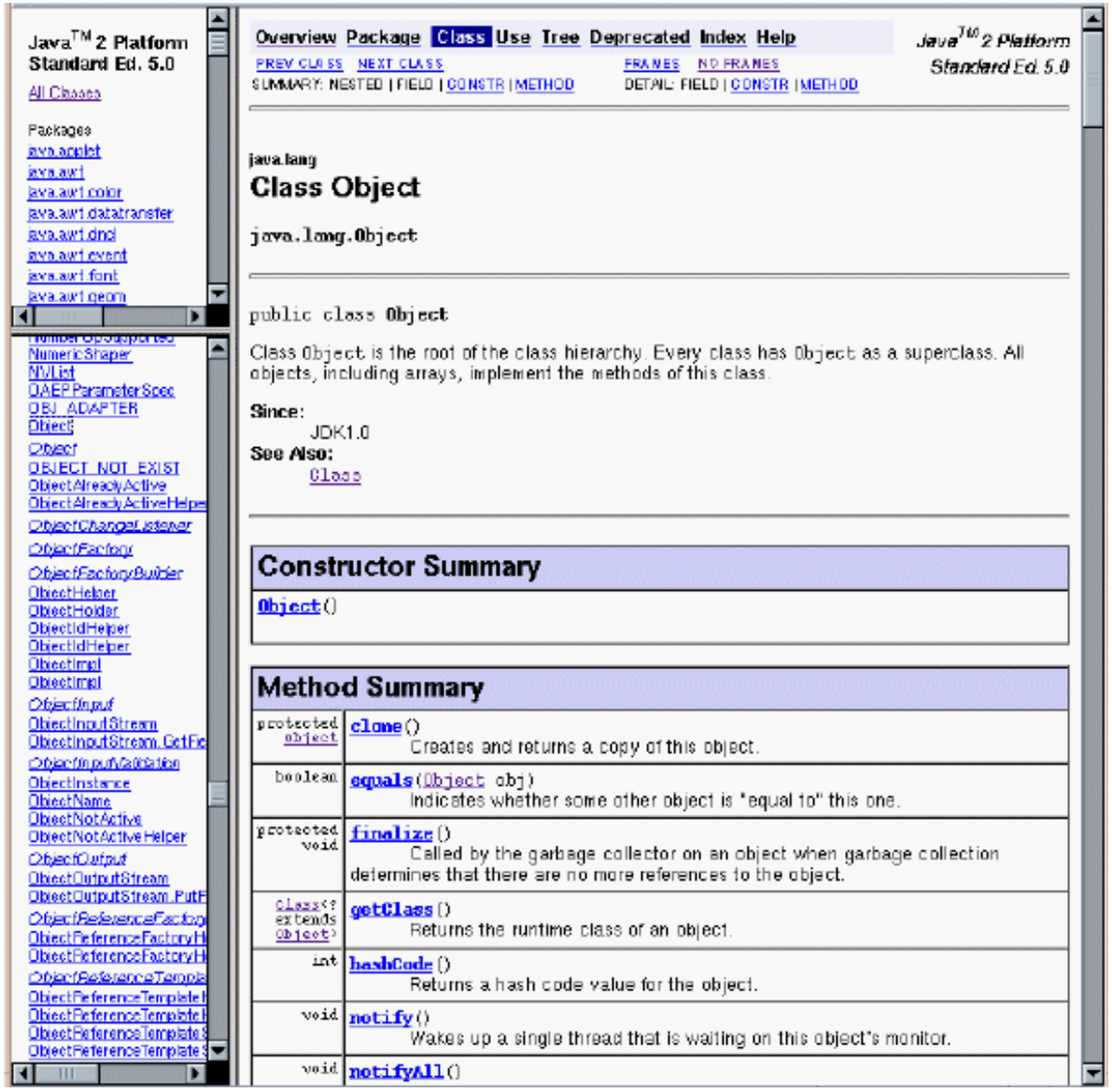
- `Can't find class TestGreeting`

  ensure the class exist in the working directory

- `Exception in thread "main" java.lang.NoSuchMethodError: main`

  ensure the class contains the `main` method

Using the J2SE5.0 JDK Components



Points to Remember

Check Your Progress

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

Developing and Testing a Java Technology Program: Objectives

- Identify the four components of a class in the Java programming language
- Use the main method in a test class to run a Java technology program from the command line
- Compile and execute a Java technology program

Table of Contents

Structuring Classes

- The class declaration
- Attribute variable declarations and initialization (optional)
- Methods (optional)
- Comments (optional)

Structuring Classes

```
public class Shirt {
  public int shirtID = 0;
  public String description = “”;
  // The color codes are R=Red, B=Blue,
 G=Green, U=Unset
  public char colorCode = ‘U’;
  public double price = 0.0; // Default
 price for all shirts


  // This method displays the values for an
 item
  public void displayShirtInformation() {
  …
```

Class Declaration

- Syntax:

```
[modifiers] class class_identifier { }
```

- Example:

```
public class Shirt { }
```

Variable Declarations and Assignments

```
public int shirtID = 0;
public String description = “”;
public char colorCode = ‘U’;
public double price = 0.0;
public int quantityInStock = 0;
```

Methods

- Syntax:

```
[modifiers] return_type
method_identifier ([arguments]){
    method_code_block
}
```

Methods

- Example:

Comments

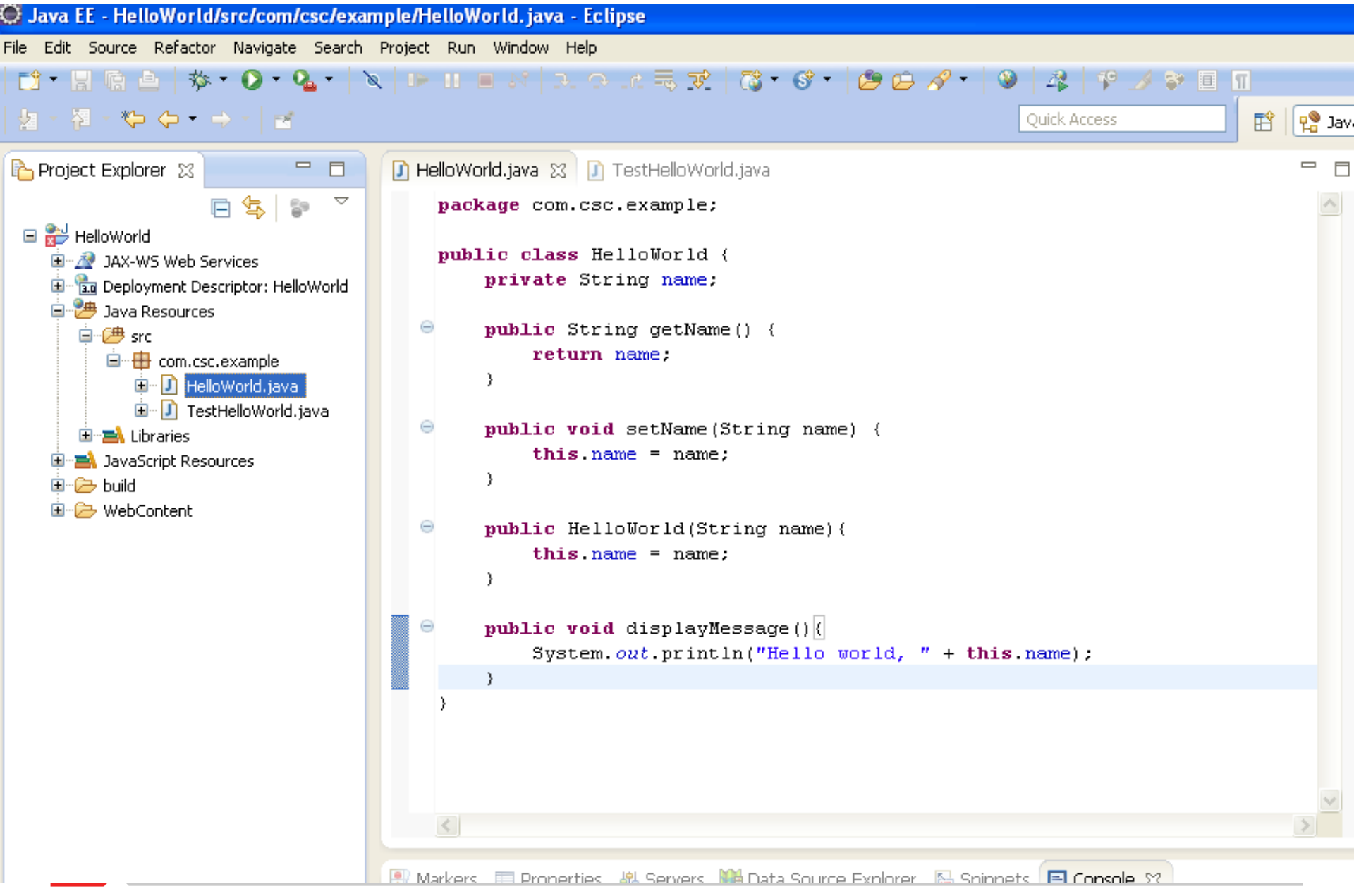
- Single-line:

```
public int shirtID = 0; // Default ID for
the shirt

public double price = 0.0; // Default price
for all shirts

// The color codes are R=Red, B=Blue,
G=Green
```

- Traditional:

```
/**********************************************
*

* Attribute Variable Declaration Section *

**********************************************
/
```

Creating and Using a Test Class

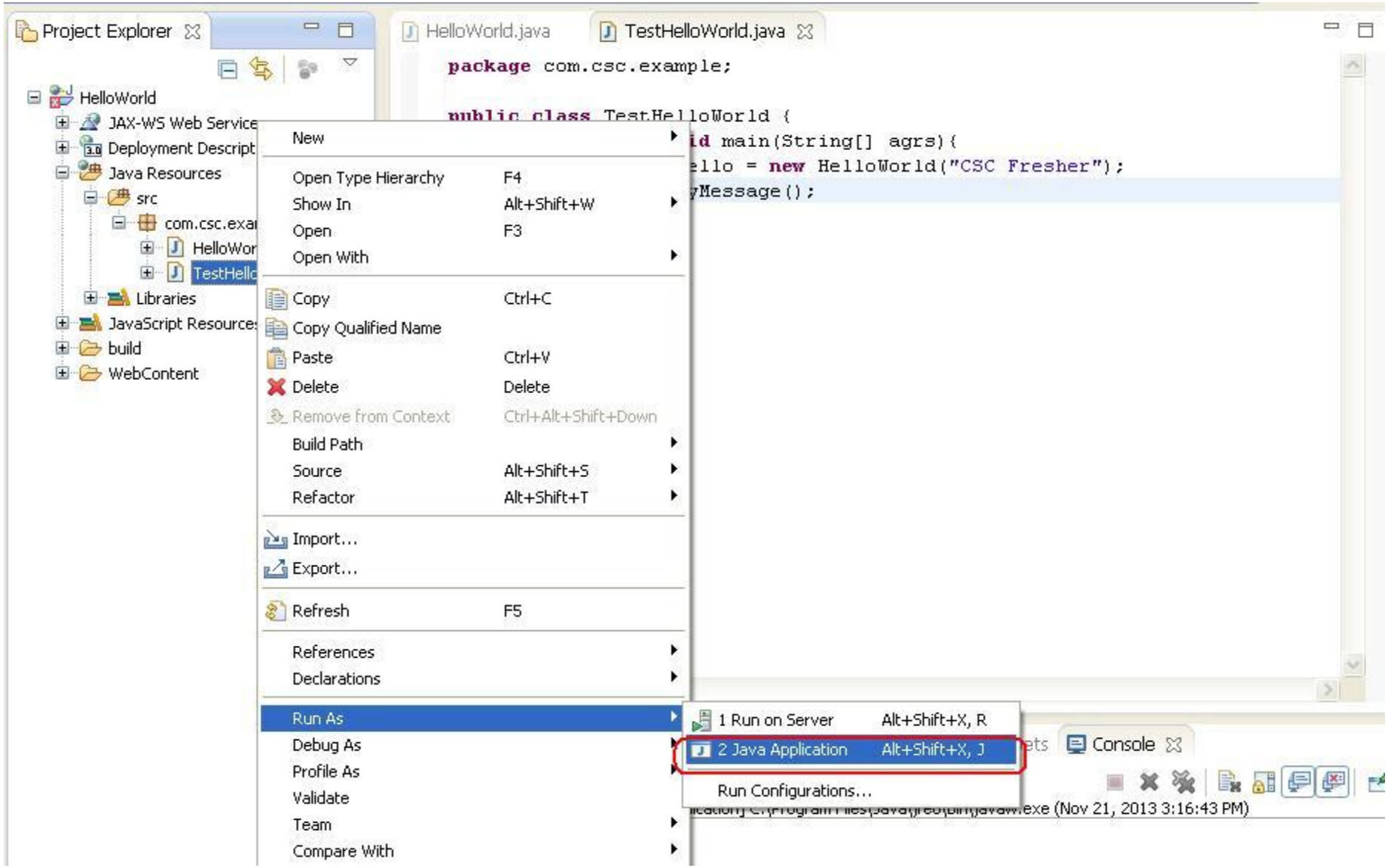


Creating and Using a Test Class - Cont

HelloWorld.java | TestHelloWorld.java

```
package com.csc.example;

public class TestHelloWorld {
    public static void main(String[] agrs){
        HelloWorld hello = new HelloWorld("CSC Fresher");
        hello.displayMessage();
    }
}
```

Executing (Testing) a Program



Debugging Tips

- Error messages state the line number where the error occurs. That line might not always be the actual source of the error.
- Be sure that you have a semicolon at the end of every line where one is required, and no others.
- Be sure that you have an even number of braces.
- Be sure that you have used consistent indentation in your program, as shown in examples in this course.

Exercise

- Do lab 1: exercise 2

Points to Remember

Object-Oriented Programming: Objectives

- Define modeling concepts: abstraction, encapsulation, and packages
- Define inheritance, polymorphism, overriding, and virtual method invocation
- 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

Table of Contents

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

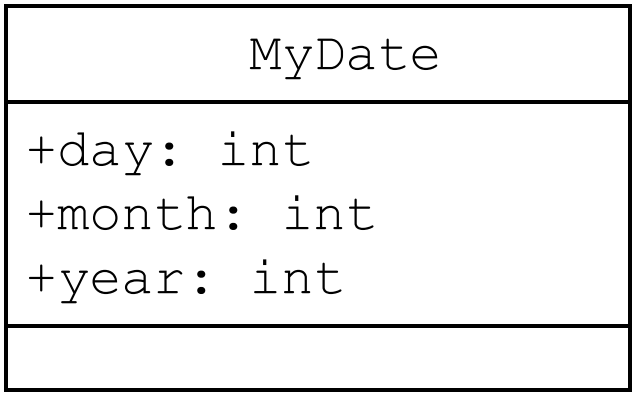
Accessing Object Members

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

```
Shirt myShirt = new Shirt();
 myShirt.colorCode='R';
 myShirt.price = 10.0;
 myShirt.displayShirtInformation();
```

Information Hiding

The problem:

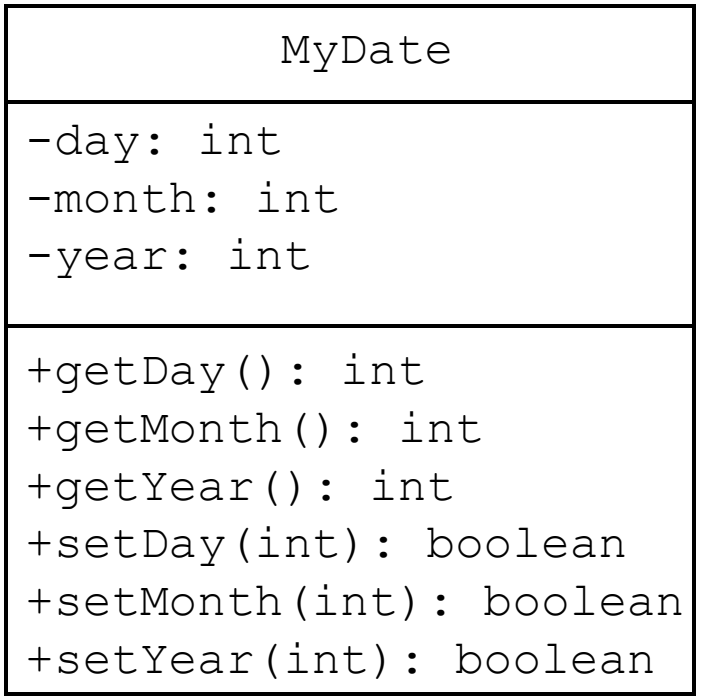


Client code has direct access to internal data (d refers to a MyDate object):

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

Information Hiding

The solution:



Client code must use setters 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

Declaring Constructors

- Basic syntax of a constructor:

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

- Example:

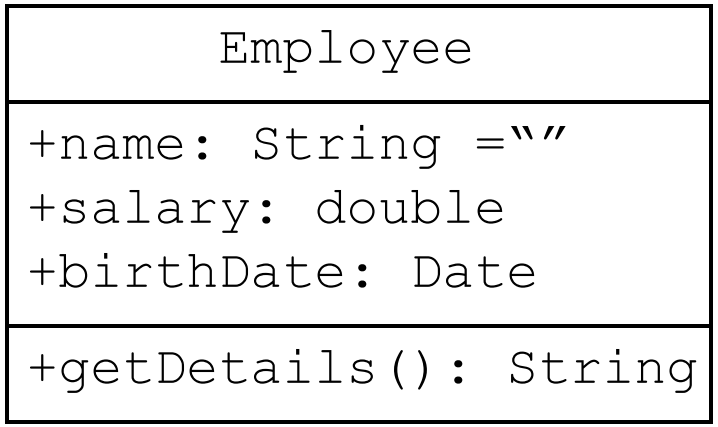
```
public class Dog {

    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 class_identifier() without having to write a constructor.

Subclassing

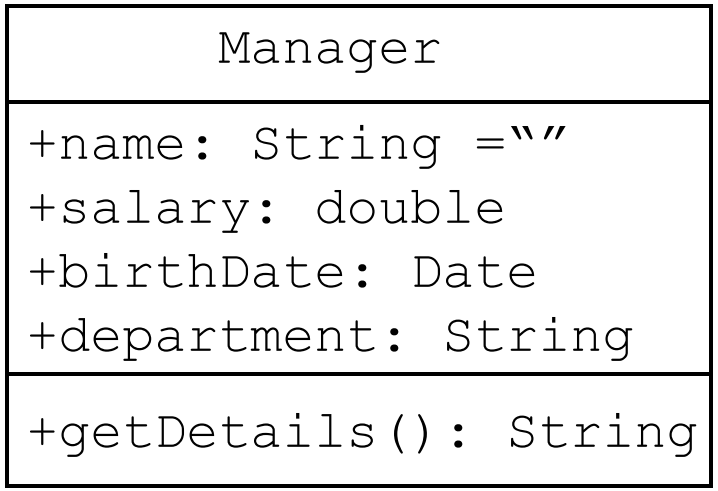
The `Employee` class is shown here.




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

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

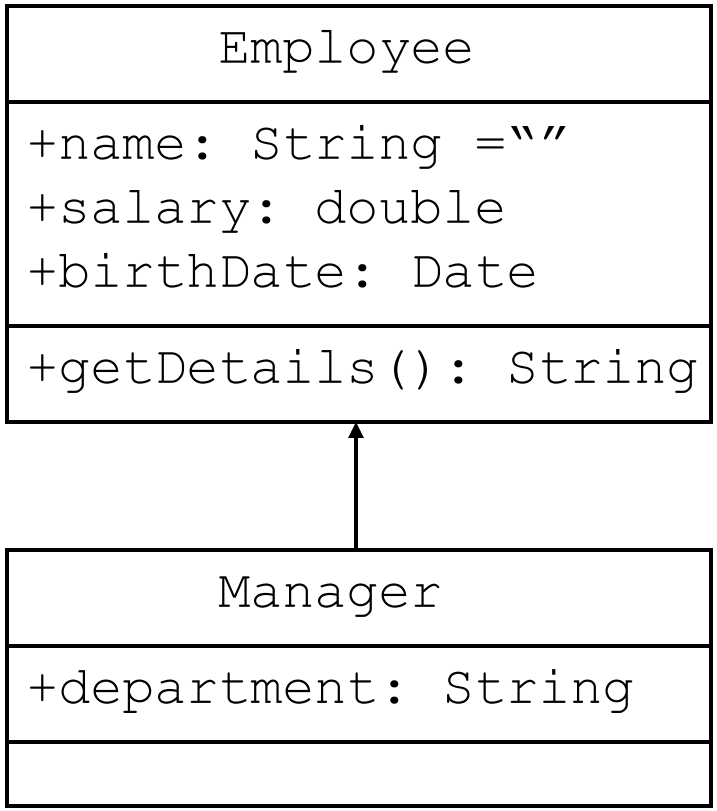
Subclassing

The `Manager` class is shown here.




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

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

Subclassing



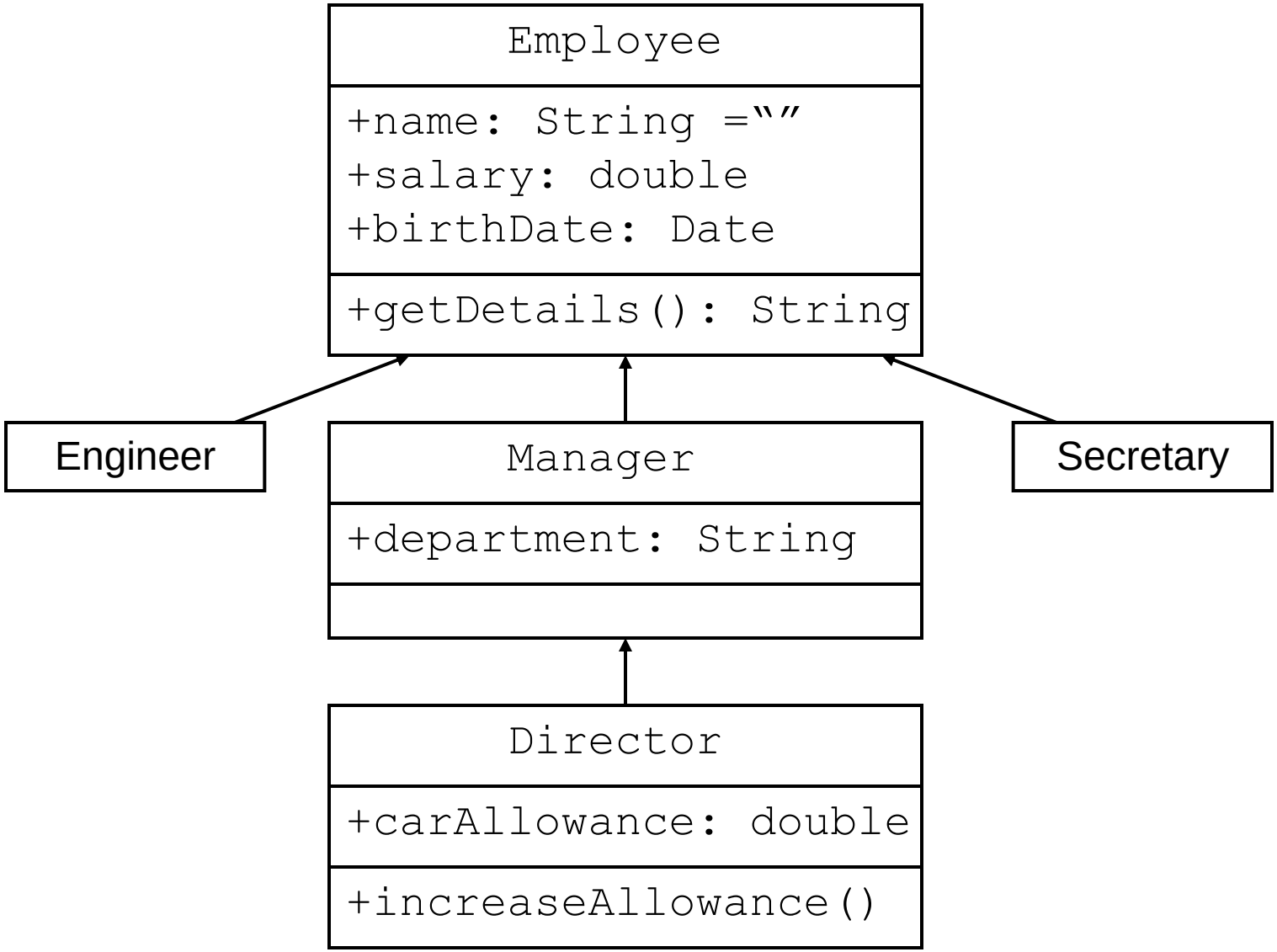

```
public class Employee {
  public String name = "";
  public double salary;
  public Date birthDate;

  public String
         getDetails() {...}
 }
 public class Manager
          extends Employee{
  public String department;
 }
```

Single Inheritance

- When a class inherits from only one class, it is called *single inheritance*.
- *Interfaces* provide the benefits of multiple inheritance without drawbacks.
- Syntax of a Java class 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 5.0, 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;
   }
}
```

Overriding Methods

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

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

Overriding Methods Cannot Be Less Accessible

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

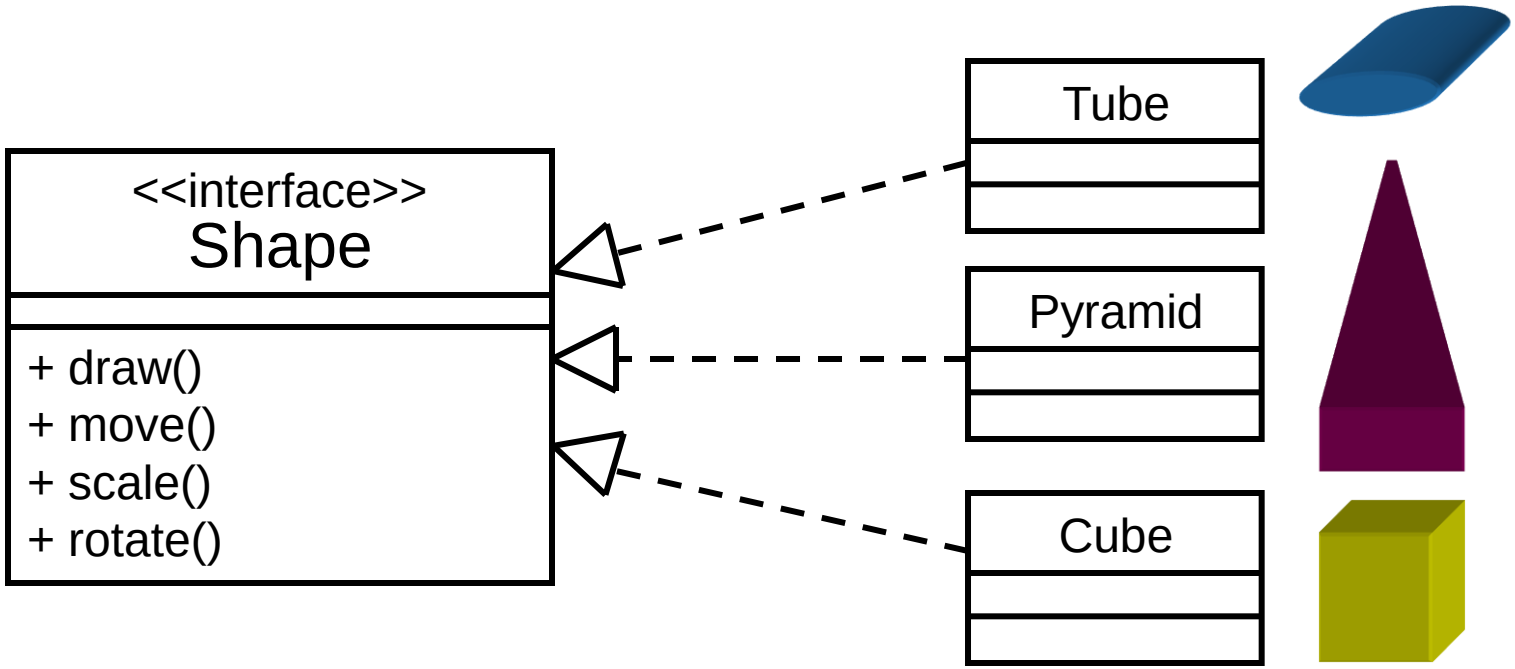
- The keyword `super` is used in a class to refer to its superclass.
- The keyword `super` is used to refer to the members of superclass, both data attributes and methods.
- Behavior invoked does not have to be in the superclass; it can be further up in the hierarchy.

Invoking Overridden Methods

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

  public String getDetails() {
       return super.getDetails()
                 + “Manager of: “ +
 department;
  }
}
```

Polymorphism

- Polymorphism is the ability to have many different forms.
- 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.

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.

Exercise

- Do lab 2: exercise 2,3

Source File Layout

- Basic syntax of a Java source file is:

  [<package_declaration>]

  <import_declaration>*

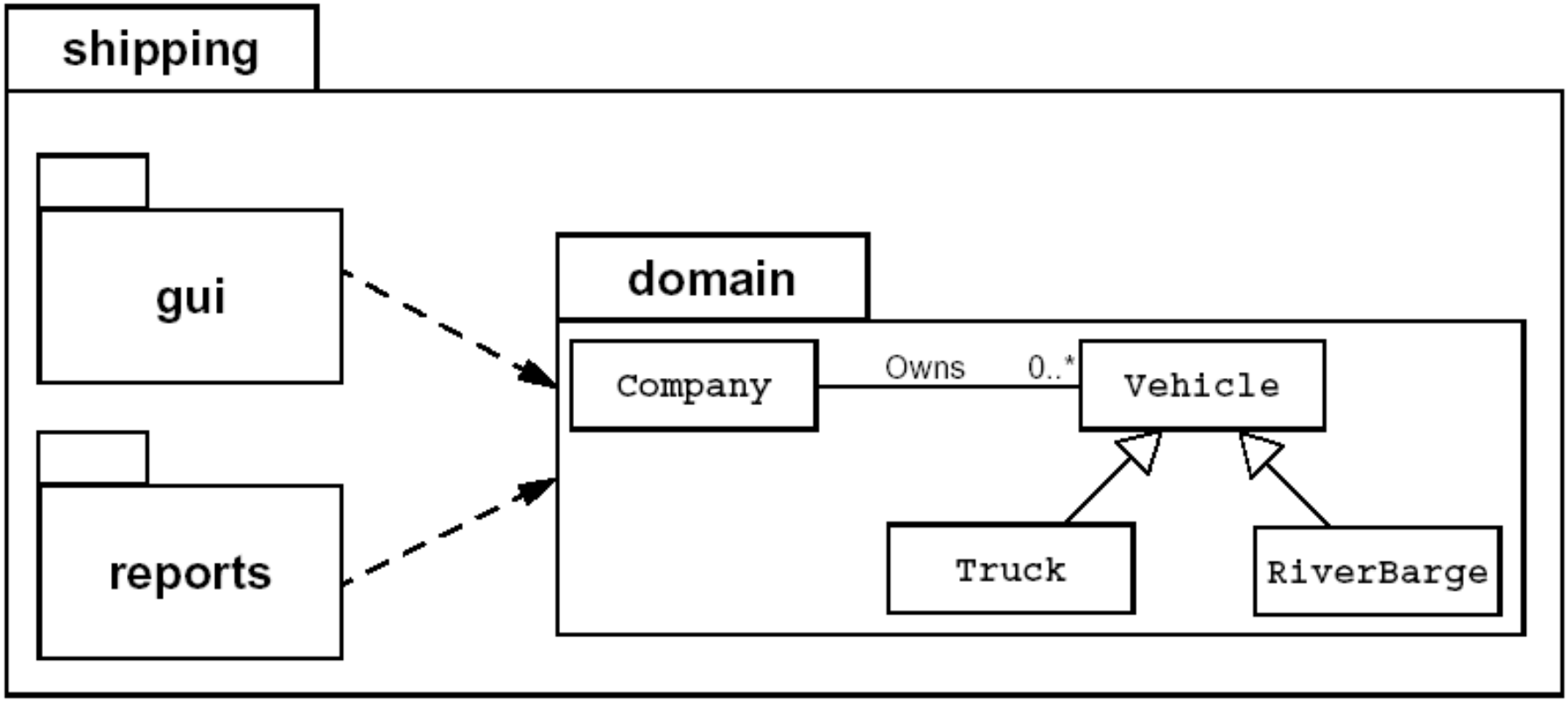
  <class_declaration>+

- For example, the VehicleCapacityReport.java file is:

```
package shipping.reports;

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

public class VehicleCapacityReport {
}
```

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/
                doc/
                classes/
                        shipping/
                                domain/
                                gui/
                                reports/
```

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

Common Java Packages

```
java.lang
java.util
java.net
```

Points to Remember

Check Your Progress

- Define modeling concepts: abstraction, encapsulation, and packages
- Define inheritance, polymorphism, overriding, and virtual method invocation
- 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

Identifiers, Keywords, and Types: Objectives

- Identify the uses for variables and define the syntax for a variable
- 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`
- Describe the significance of a reference variable

Table of Contents

Identifying Variable Use and Syntax

```
public class Shirt {
  public int shirtID = 0;
  public String description = “”;
  // The color codes are R=Red, B=Blue,
 G=Green, U=Unset
  public char colorCode = ‘U’;
  public double price = 0.0; // Default
 price for all shirts


  // This method displays the values for an
 item
  public void displayShirtInformation() {
  …
```

Uses for Variables

- Holding unique data for an object instance
- Assigning the value of one variable to another
- Representing values within a mathematical expression
- Printing the values to the screen
- Holding references to other objects

Variable Declaration and Initialization

- Syntax (attribute or instance variables):

```
[modifiers] type identifier [= value];
```

- Syntax (local variables):

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

- Several Variables in One Line of Code:

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

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.

Describing Variable Scope

```
public class Person {
   // begin scope of int age
   private int age = 34;

   public void displayName() {
      // begin scope of String name
      String name = “Peter Simmons”;
      System.out.println(“My name is “ + name + “
 and I am “ + age );
   } // end scope of String name

   public String getName () {
      return name; // this causes an error
   }
} // end scope of int age
```

Identifiers

- Are names given to a variable, class, or method
- Can start with a Unicode letter, underscore (_), or dollar sign ($)
- Cannot contain punctuation, spaces, or dashes.
- Java technology keywords cannot be used.
- Are case-sensitive and have no maximum length
- Examples:

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

Guidelines

- Begin each variable with a lowercase letter; subsequent words should be capitalized, such as `myVariable`.
- Choose names that are mnemonic and that indicate to the casual observer the intent of the variable.

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`

Constants

- Variable (can change):

```
double salesTax = 6.25;
```

- Constant (cannot change):

```
final double SALES_TAX = 6.25;
```

- Guideline – Constants should be capitalized with words separated by an underscore (_).

Standard Mathematical Operators

Purpose	Operator	Example	Comments
Remainder	%	`mod = num1 % num2;` If `num1` is 31 and `num2` is 6, `mod` is 1.	Remainder always gives an answer with the same sign as the first operand.

Increment and Decrement Operators

The short way:

Operator	Purpose	Example	Notes

++	Pre-increment (++*variable*)	`int i = 6;`
		`int j = ++i;`
		`i is 7, j is 7`	

	Post-increment (*variable*++)	`int i = 6;`
		`int j = i++;`
		`i is 7, j is 6`
			The value of `i` is assigned to `j` before `i` is incremented. Therefore, `j` is assigned 6.

--	Pre-decrement (--*variable*)	`int i = 6;`
		`int j = --i;`
		`i is 5, j is 5`	

	Post-decrement (*variable*--)	`int i = 6;`
		`int j = i--;`
		`i is 5, j is 6`
			The value `i` is assigned to `j` before `i` is decremented. Therefore, `j` is assigned 6.

Operator Precedence

Rules of precedence:

- 1. Operators within a pair of parentheses
- 2. Increment and decrement operators
- 3. Multiplication and division operators, evaluated from left to right
- 4. Addition and subtraction operators, evaluated from left to right
- Example of need for rules of precedence (is the answer 34 or 9?):

```
c = 25 - 5 * 4 / 2 - 10 + 4;
```

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

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

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

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

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:

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

Floating Point – float and double

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:

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

Variable Initialization

Variable	Value
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0F`
`double`	`0.0`
`char`	‘’
`boolean`	`false`
All reference types	`null`

Type Casting

- Syntax:

```
identifier = (target_type) value
```

- Example of potential issue:

```
int num1 = 53; // 32 bits of memory
int num2 = 47; // 32 bits of memory
byte num3; // 8 bits of memory reserved
num3 = (num1 + num2); // compiler error
```

- Example of potential solution:

```
int num1 = 53; // 32 bits of memory
int num2 = 47; // 32 bits of memory
byte num3; // 8 bits of memory reserved
num3 = (byte)(num1 + num2); // no data loss
```

Type Casting

Examples:

```
int myInt;
long myLong = 99L;
myInt = (int) (myLong); // No data loss
                        // A much larger number
 would
                        // result in data loss.
int myInt;
long myLong = 123987654321L;
myInt = (int)(myLong); // Number is
 "chopped"
```

Compiler Assumptions for Integral Data Types

- Example of potential problem:

```
short a, b, c;
a = 1 ;
b = 2 ;
c = a + b ; //compiler error
```

- Example of potential solutions:
  - Declare c as an `int` type in the original declaration:

```
int c;
```

  - Type cast the (a+b) result in the assignment line:

```
c = (short)(a+b);
```

Floating Point Data Types and Assignment

- Example of potential problem:

```
float float1 = 27.9; //compiler error
```

- Example of potential solutions:
  - The `F` notifies the compiler that 27.9 is a `float` value:

```
float float1 = 27.9F;
```

  - 27.9 is cast to a float type:

```
float float1 = (float) 27.9;
```

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) { ... }
}
```

Java Reference Types

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

```
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: Example

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

      val = 11; // Assign the int
      changeInt(val); // Try to change it
      // 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: Example

```
// 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: Example

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

Exercise

- Do lab 3: exercise 1,2

Points to Remember

Check Your Progress

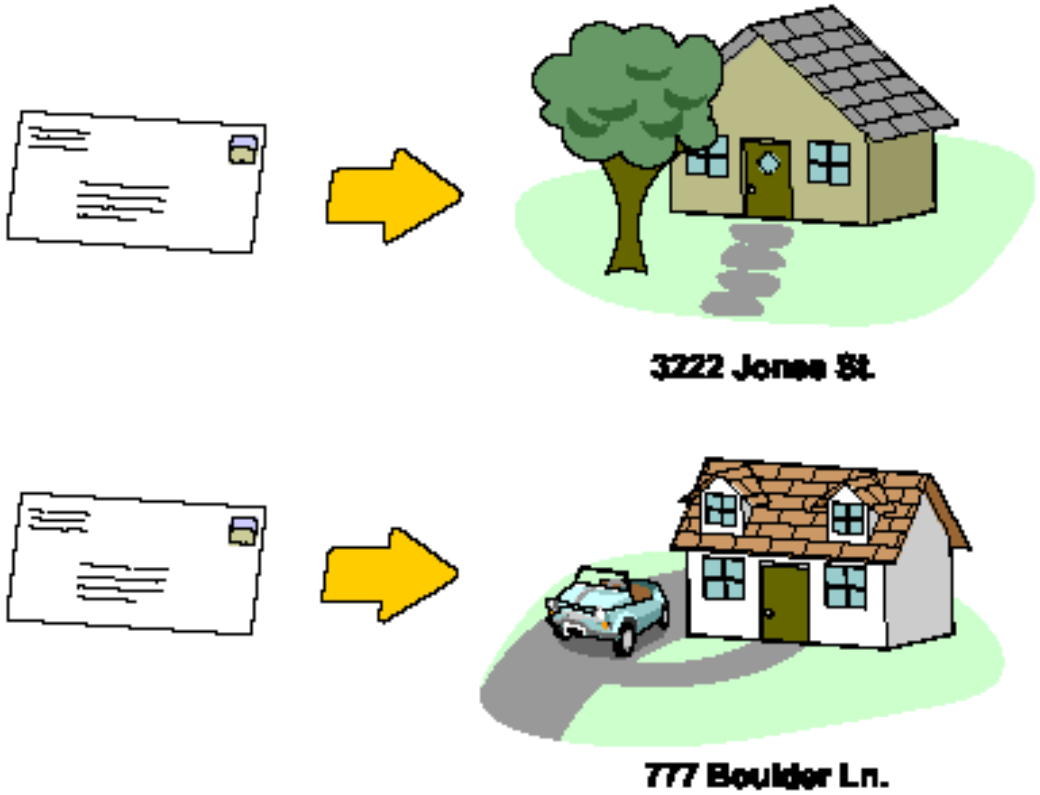
- Use comments in a source program
- Distinguish between valid and invalid identifiers
- Recognize Java technology keywords
- List the eight primitive types
- Define literal values for numeric and textual types
- Define the terms primitive variable and reference variable

Creating and Using Objects: Objectives

- Declare, instantiate, and initialize object reference variables
- Describe default initialization
- State the consequences of assigning variables of class type
- Compare how object reference variables are stored in relation to primitive variables
- Use a class (the `String` class) included in the Java SDK

Table of Contents

Object Reference Variables



Object Reference Variables

```
public class ShirtTest {

  public static void main (String args[]) {

     Shirt myShirt;
     myShirt = new Shirt();

     myShirt.displayShirtInformation();
  }
}
```

Object Reference Variables

- Declaring Object Reference Variables

  Syntax: `Classname identifier;`

  Example: `Shirt myShirt;`

- Instantiating an Object

  `new Classname()`

- Initializing Object: the assignment operator

  `myShirt = new Shirt();`

Object Reference Variables

```
public class ShirtTestTwo {

  public static void main (String args[]) {

     Shirt myShirt = new Shirt();
     Shirt yourShirt = new Shirt();

     myShirt.colorCode='R';
     yourShirt.colorCode='G';

     myShirt.displayShirtInformation();
     yourShirt.displayShirtInformation();
  }
}
```

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.

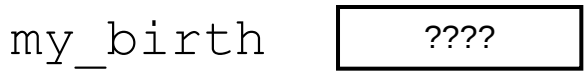
  d. The object reference is returned by the new operator.
- The reference to the object is assigned to a variable.
- An example is:

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

Memory Allocation and Layout

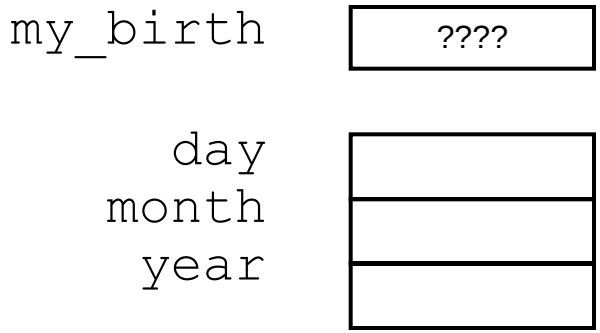
- A declaration allocates storage only for a reference:

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



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

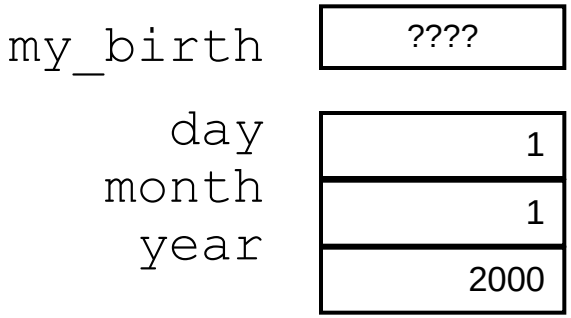
```
MyDate my_birth = new MyDate(22, 7, 1964);
```



Explicit Attribute Initialization

- Initialize the attributes as follows:

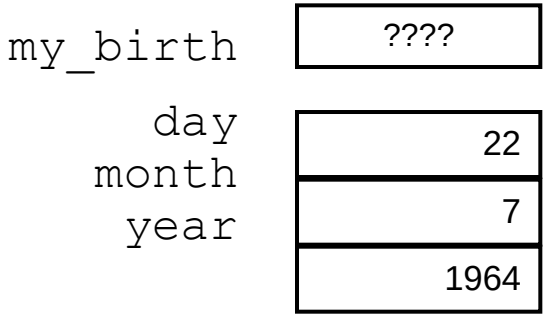
```
MyDate my_birth = new MyDate(22, 7, 1964);
```



Executing the Constructor

- Execute the matching constructor as follows:

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

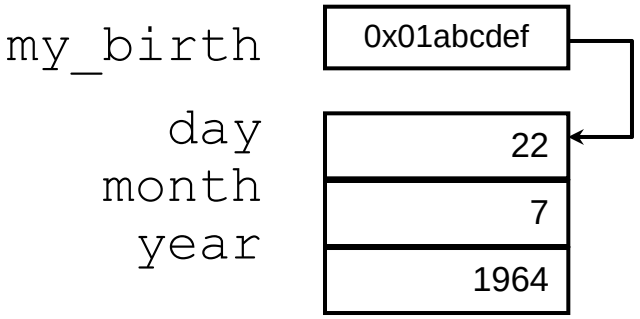


- In the case of an overloaded constructor, the first constructor can call another.

Assigning a Variable

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

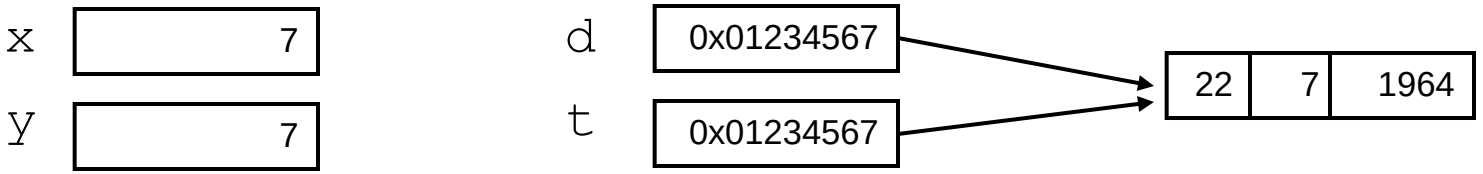
```
MyDate my_birth = new MyDate(22, 7, 1964);
```



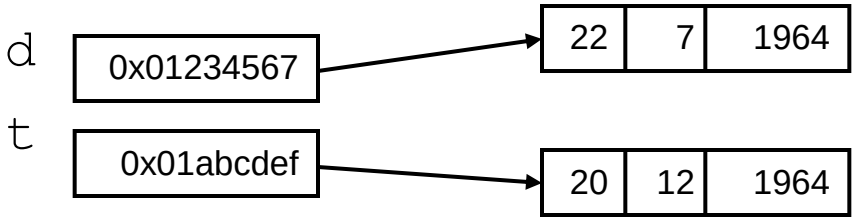
Assigning References

- Two variables refer to a single object:

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

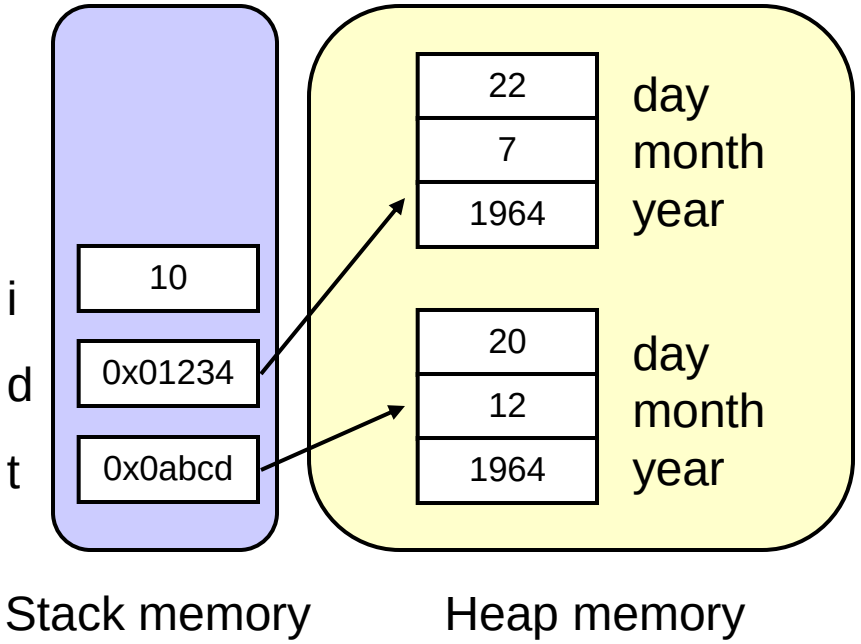


- Reassignment makes two variables point to two objects: `t = new MyDate(20, 12, 1964);`



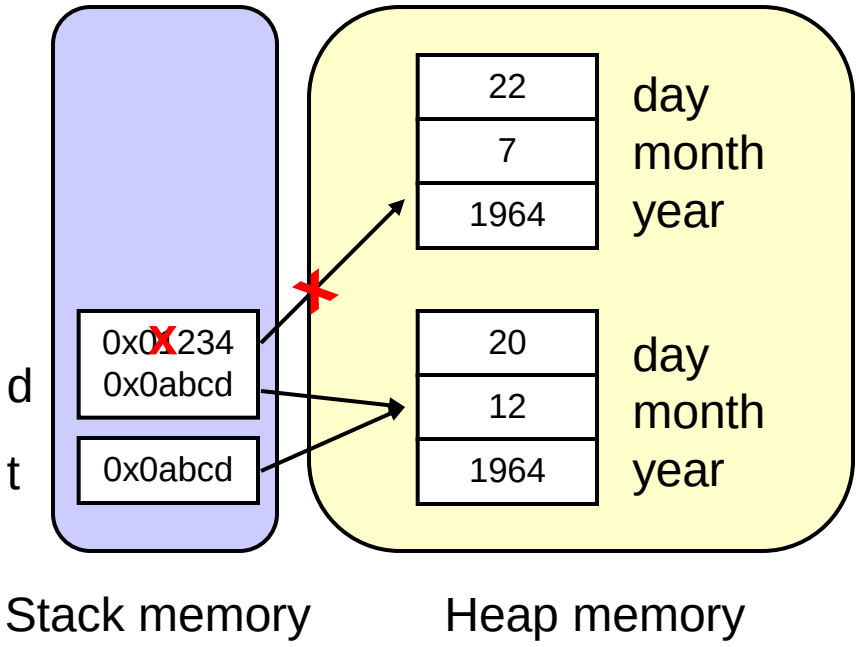
Storing Object Reference Variables in Memory

```
public static void main (String args[]) {
     int i = 10;
      MyDate d = new MyDate(22, 7, 1964);
    MyDate t = new MyDate(20, 12, 1964);
  }
```



Assigning an Object Reference Variable

```
MyDate d = new MyDate(22, 7, 1964);
MyDate t = new MyDate(20, 12, 1964);
d = t;
```



Using the String Class

- Creating a String object with the new keyword:

```
String myName = new String(“Fred Smith”);
```

- Creating a String object without the new keyword:

```
String myName = “Fred Smith”;
```

Points to Remember

Check Your Progress

- Declare variables of class type
- Construct an object using new
- Describe default initialization
- State the consequences of assigning variables of class type

Using Operators and Decision Constructs: Objectives

- Recognize, describe, and use Java software operators
- Create if and if/else constructs
- Use the switch constructs

Table of Contents

Operator Precedence

Operators	Associative
++ -- + unary - unary ~ !	R to L
* / %	L to R
+ -	L to R
<< >> >>>	L to R
< > <= >= instanceof	L to R
== !=	L to R
^	L to R
&	L to R
|	L to R
&&	L to R
||	L to R
<boolean_expr> ?<expr1> : <expr2>	R to L
= *= /= %= += -= <<= >>= >>>= &= |=	R to L

Logical Operators

- The boolean operators are:

  `!` – NOT `&` – AND

  `|` – OR `^` – XOR

- The short-circuit boolean operators are:

  `&&` – AND `||` – OR

- You can use these operators as follows:

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

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.

Simple if Constructs

The if statement syntax:

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

Example:

```
 if ( x = 10 )
   System.out.println(“Finished !");
```

or (recommended):

```
if ( x = 10 ) {
   System.out.println(“Finished !");
}
```

Complex if, else Constructs

The if-else statement syntax:

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

Example:

```
if ( x < 10 ) {
      System.out.println(“Keep working...”);
}else {
      System.out.println(“Finished !”);
}
```

Complex if, else Constructs

The if-else-if statement syntax:

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

Example:

```
if ( x < 10 ) {
      System.out.println("Keep working...");
}else if (x > 10 && x < 20) {
      System.out.println("Finished !");
}
```

Using the `switch` Construct

The switch statement syntax:

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

When to Use `switch` Constructs

- Equality tests
- Tests against a single value, such as customerStatus
- Tests against the value of an int, short, byte, or char type

Exercise

- Do lab 4: exercise 1

Points to Remember

Using Loop Constructs: Objectives

- Create `while` loops
- Develop `for` loops
- Create `do/while` loops
- Use the labeled forms of `break` and `continue`

Table of Contents

Creating `while` Loops

Syntax:

```
while (boolean_expression) {
   code_block;
} // end of while construct
// program continues here
```

Example:

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

Developing a `for` Loop

Syntax:

```
for (initialize[,initialize];
    boolean_expression; update[,update]) {
  code_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));
}
```

Coding a `do/while` Loop

Syntax:

```
do {
    code_block;
}
while (boolean_expression);// Semicolon is
mandatory.
```

Example:

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

Nested Loops

```
public class ForRectangle {


  public void displayRectangle(int height,
 int width) {
    for (int rowCount = 0; rowCount <
 height; rowCount++) {
       for (int colCount = 0; colCount <
 width; colCount++) {
          System.out.print("@");
       }
       System.out.println();
    }
  }
}
```

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 );
```

Comparing Loop Constructs

- Use the `while` loop to iterate indefinitely through statements and to perform the statements zero or more times.
- Use the `do/while` loop to iterate indefinitely through statements and to perform the statements one or more times.
- Use the `for` loop to step through statements a predefined number of times.

Exercise

- Do lab 4: exercise 2

Points to Remember

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

Table of Contents

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 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;
}
```

Initializing Arrays

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

```
String[] names = new String[2];
names[0] = "Georgia";
names[1] = "Simon";

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

Initializing Arrays

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

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

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

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];
```

Exercise

- Do lab 5: exercise 1,2

Points to Remember

Check Your Progress

- Declare and create arrays of primitive, class, or array types
- Explain why elements of an array are initialized
- Explain how to initialize the elements of an array
- Determine the number of elements in an array
- Create a multidimensional array

Advanced Class Features: Objectives

- Describe the concepts of constructor and method overloading
- Describe the complete object construction and initialization operation
- 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

Table of Contents

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;
    }
}
```

Methods Using Variable Arguments

- 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);
```

Comparison of Overloading and Overriding

Overloading	Overriding
Can be in different classes	Method must be in subclass
Method must have same name but have different parameters	Method must have the same name AND the same parameters
Return type may be different	Return type can be a subclass of the overridden return type
Different method signature	Same method signature
Method's access modifier can be more private than parent method's	Method's access modifier CANNOT be more private than parent method's
Methods can throw more exceptions than the parent method they overload	Methods cannot throw more exceptions than the parent method they override

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.

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.

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

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.

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

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; // autoboxing
int p2 = wInt; // autounboxing
```

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

Exercise

- Do lab 6: exercise 1,2

The `static` Keyword

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

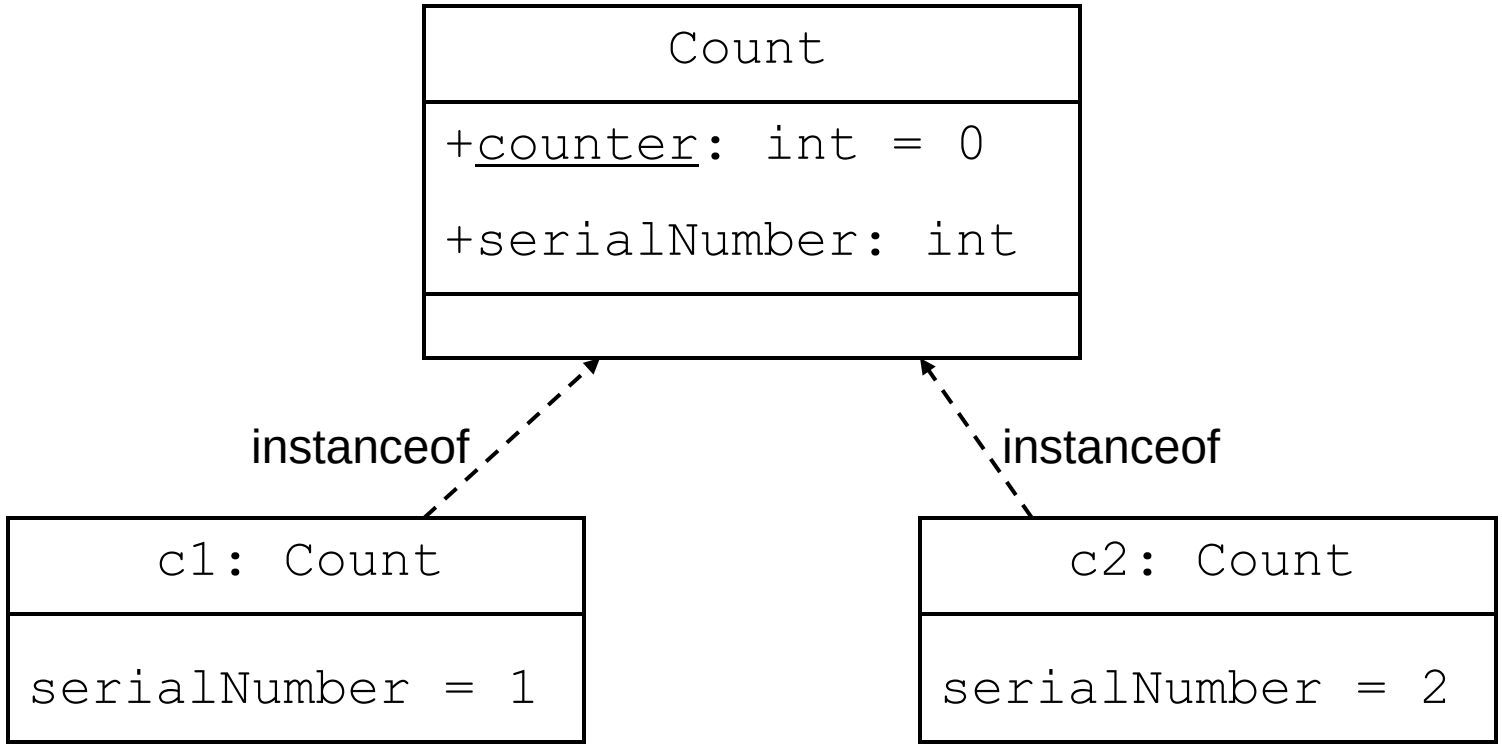
Class Attributes

Are shared among all instances of a class:

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

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

Class Attributes

Are shared among all instances of a class:



Class Attributes

If the static member is `public`, it can be accessed from outside the class without an instance:

```
public class OtherClass {
   public void incrementNumber() {
      Count.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
 " 4 + Count2.getTotalCount());


    Count2 counter = new Count2();
    System.out.println("Number of counter is
 " 7 + Count2.getTotalCount());
  }
}
```

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

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
}
```

Exercise

- Do lab 7: exercise 1

The Enumerated Type

```
package cards.domain;

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

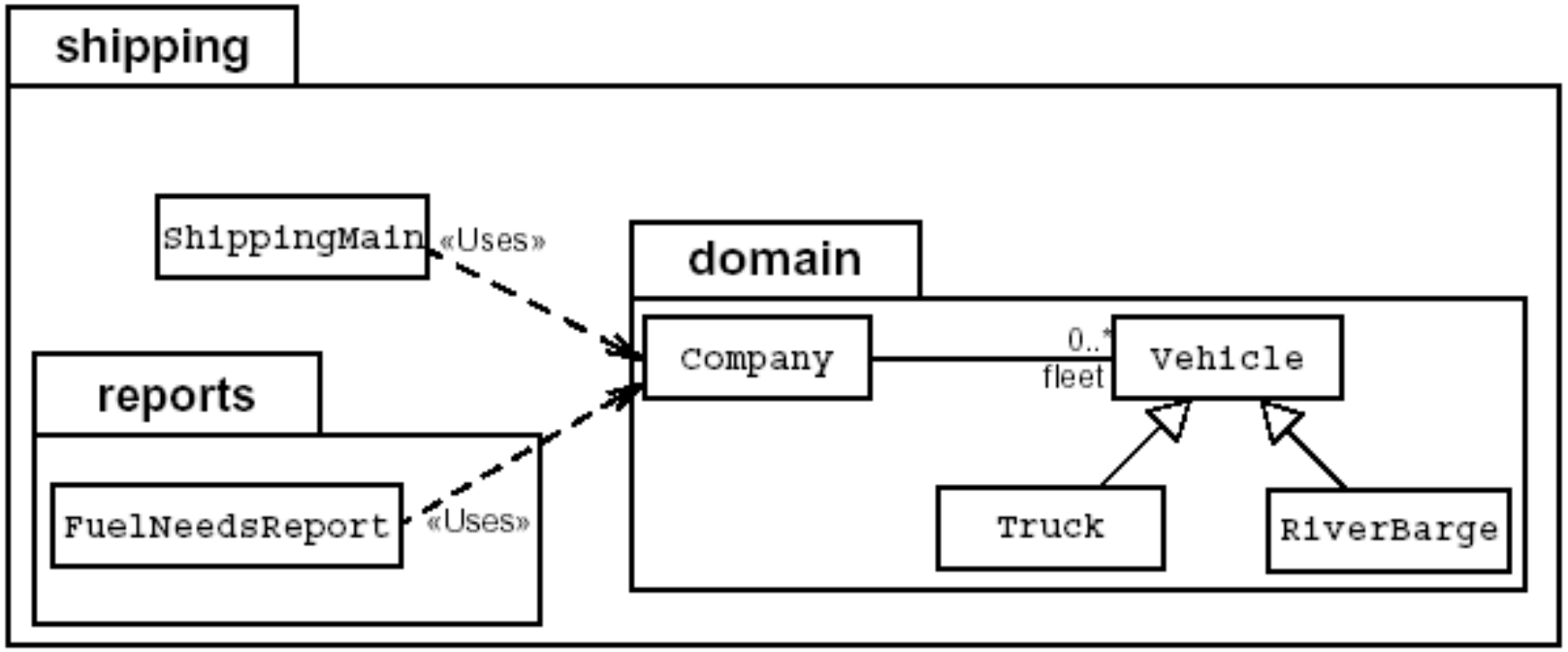
Advanced Enumerated Types

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;
    }
}
```

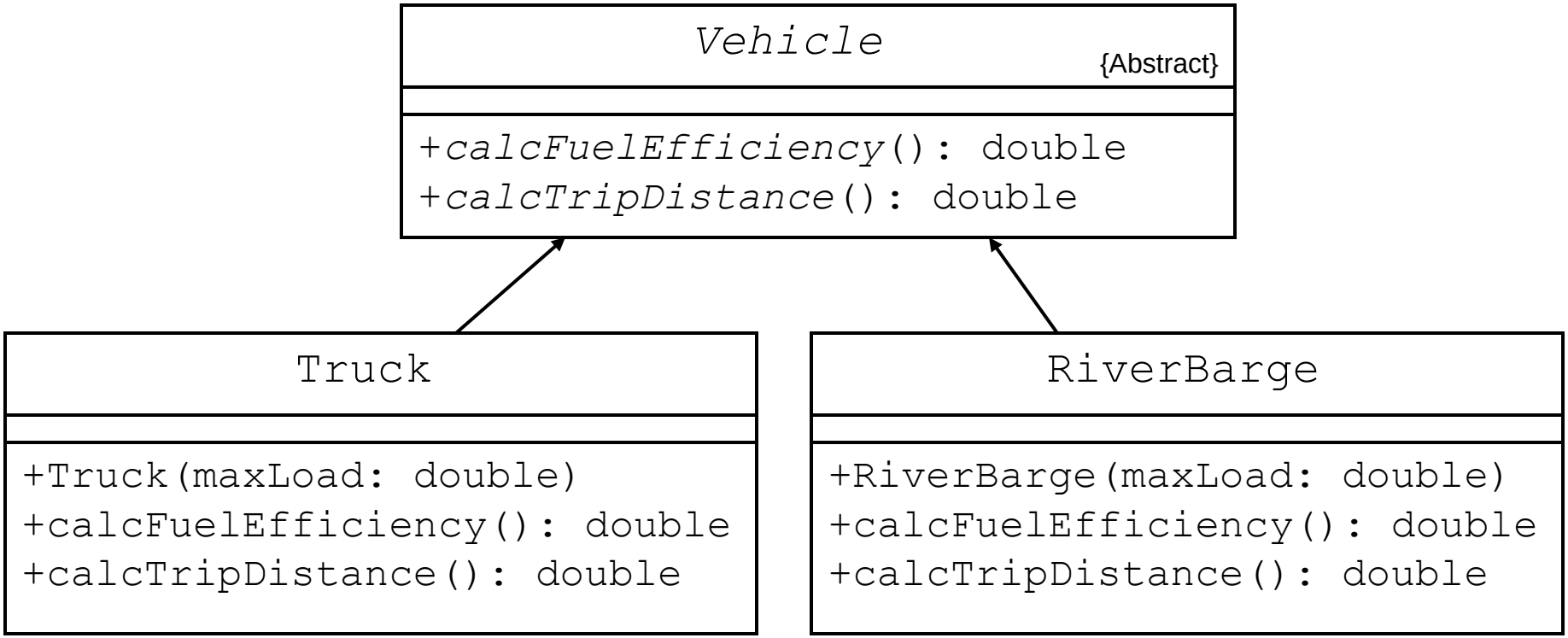
Abstract Classes

- The design of the Shipping system looks like this:



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 Solution

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:

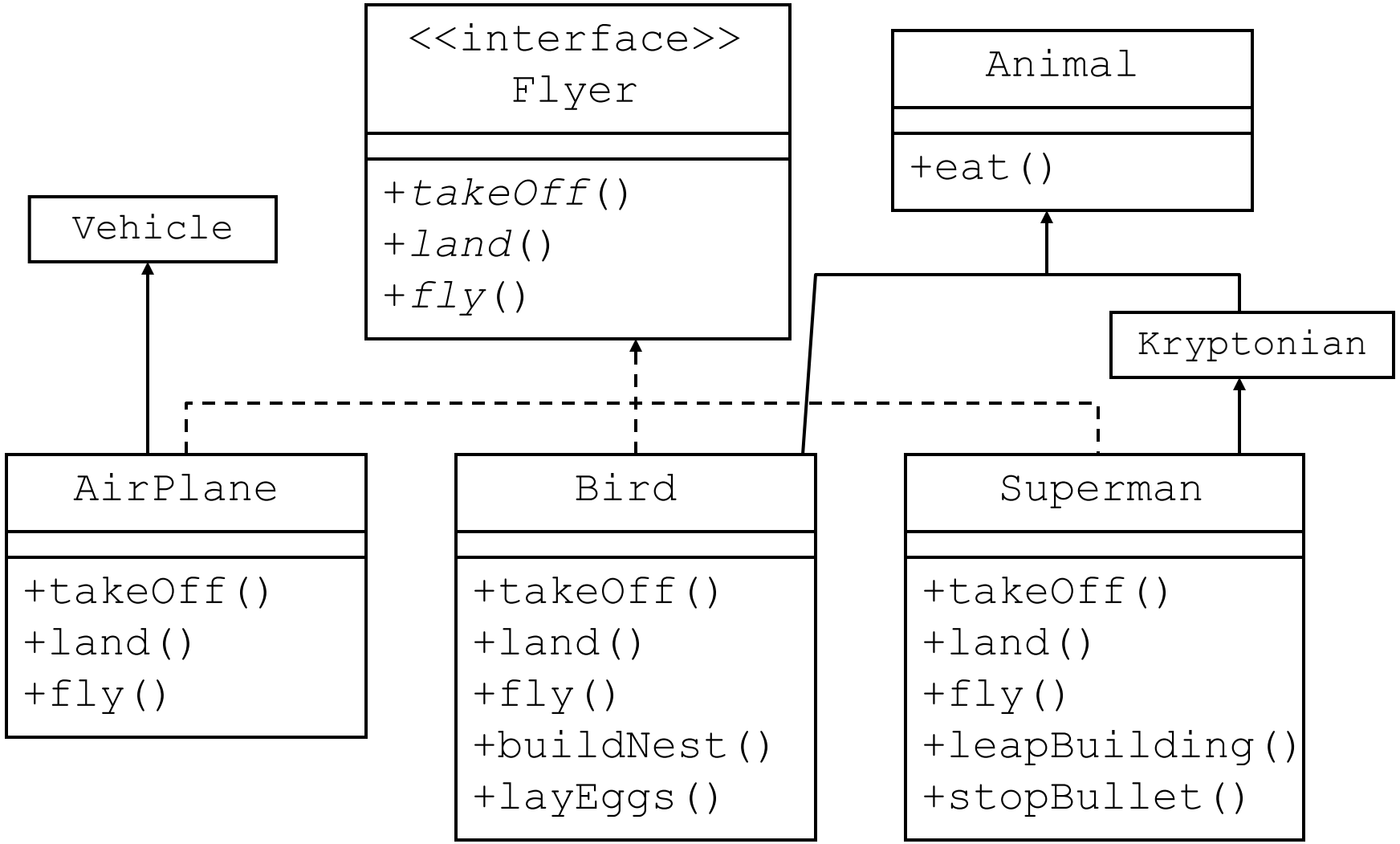
```
public class RiverBarge extends Vehicle {
  public RiverBarge(double maxLoad) {...}
  public double calcFuelEfficiency() {
    /* calculate the fuel consumption of a
 truck at a given load */
  }
  public double calcTripDistance() {
    /* calculate the distance of this trip
 on highway */
  }
}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

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



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 */ }

}
```

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

Exercise

- Do lab 7: exercise 2

Points to Remember

Check Your Progress

- Identify constructor and method overloading
- Identify the use of this to overloaded constructors
- Describe the complete object construction and initialization operation
- Identify the invocation of super class methods and constructors
- Describe static variables, methods, and initializers
- Describe final classes, methods, and variables
- Explain how and when to use abstract classes and methods
- Explain how and when to use an interface

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

Table of Contents

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 // result?
java AddArguments 1 two 3.0 4 // result?
```

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.");
    }
  }
```

The `try-catch` Statement

```
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);
}
```

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

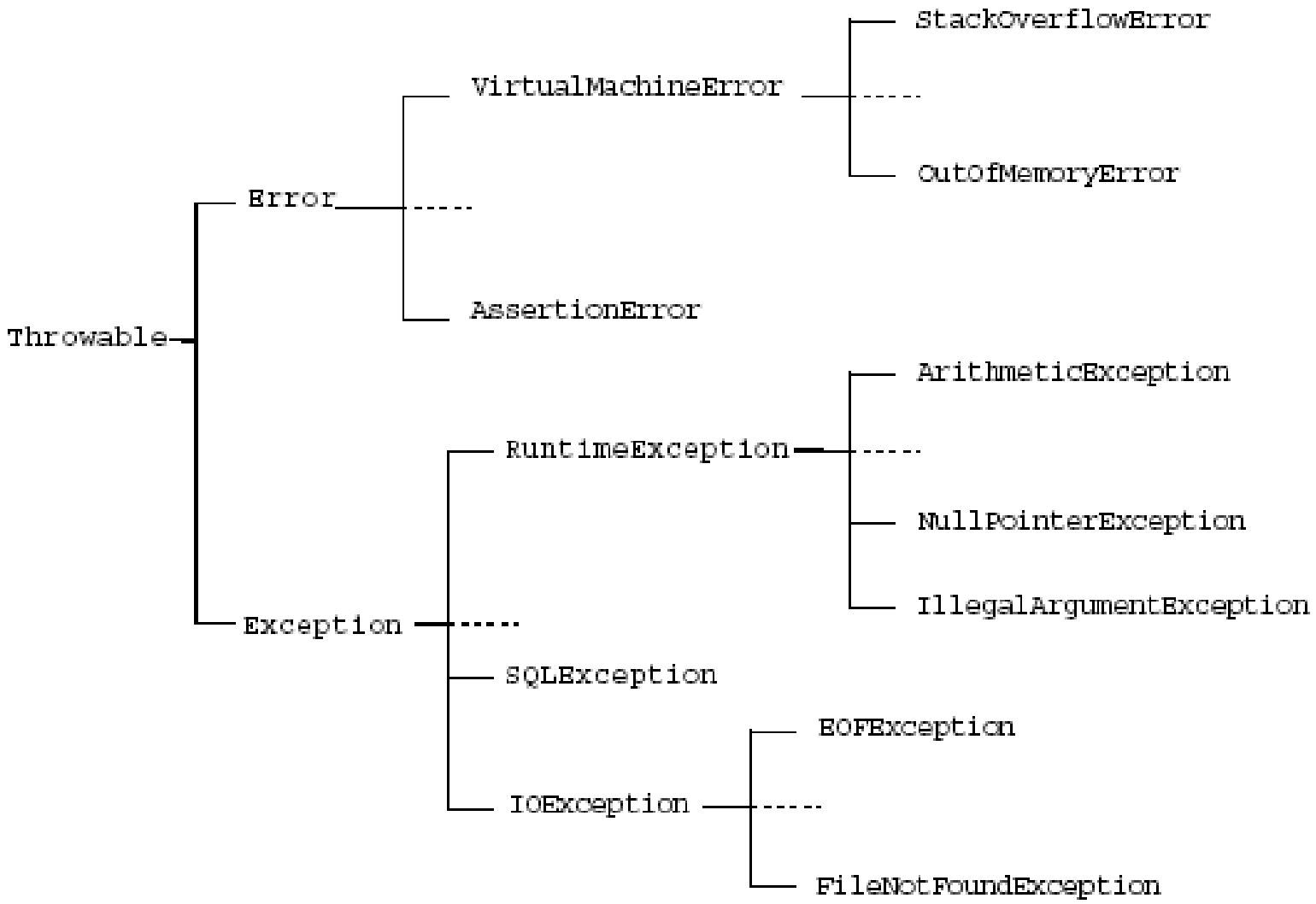
- If an exception is not handled in the current `try-catch` block, it is thrown to the caller of that method.
- If the exception gets back to the main method and is not handled there, the program is terminated abnormally.

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`

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;
   }
}
```

Handling a User-Defined Exception

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

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

   successful = open(serverName, portTo);
   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");
  }
}
```

Post-conditions 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
   return result;
}
```

Exercise

- Do lab 8: exercise 1

Points to Remember

Check Your Progress

- Define exceptions
- Use try, catch, and finally statements
- Describe exception categories and identify common exceptions
- Develop programs to handle your own exceptions
- Use and enable assertions at runtime

Text-Based Applications: Objectives

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

Table of Contents

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 just like files in Java; the File class supports methods for retrieving an array of files in the directory, as follows:

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

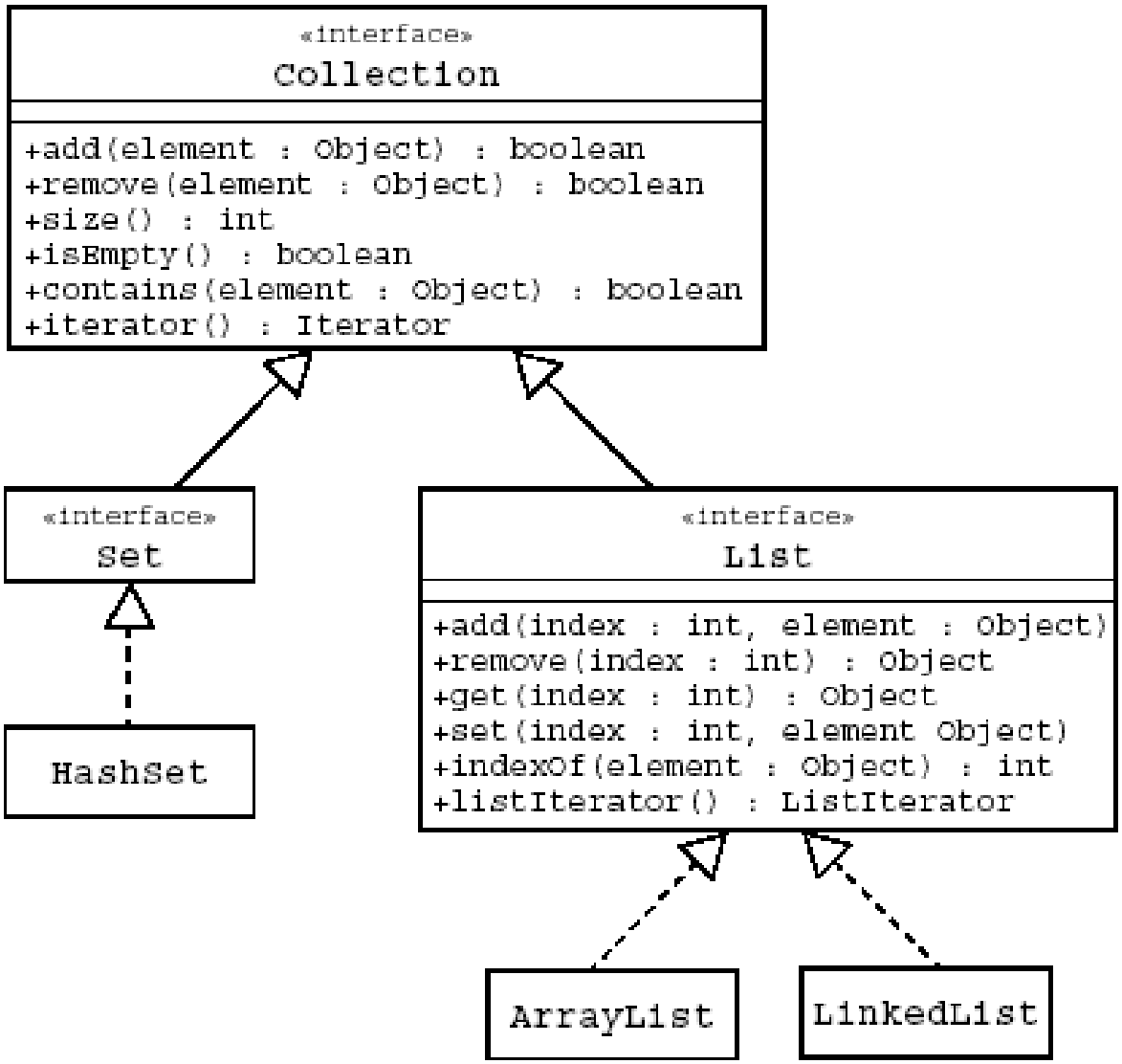
File Stream I/O

- 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.
- Example:

The Collections API

- A *collection* is a single object representing 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; any specific ordering (or lack of) and allowance of duplicates is specified by each implementation
  - `Set` – An unordered collection; no duplicates are permitted
  - `List` – An ordered collection; duplicates are permitted

The Collections API



A Set Example

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

The output generated from this program is:

```
[one, 3rd, second, 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("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, second, 4]
```

Generics

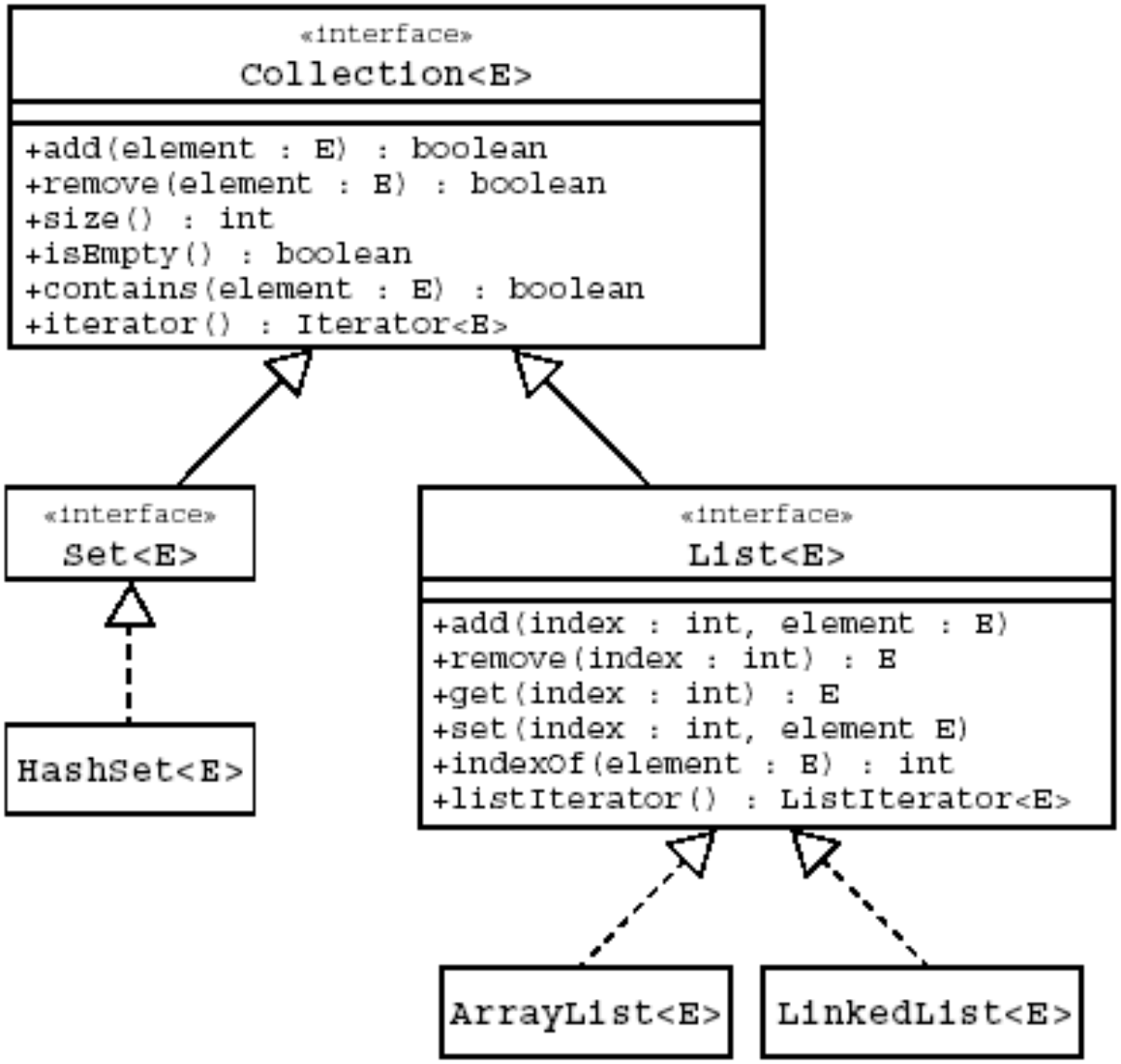
- Generics are described as follows:
  - Provides compile-time type safety
  - Eliminates the need for casts
- Before Generics

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

- After Generics

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

Generic Collections API



Compiler Warnings

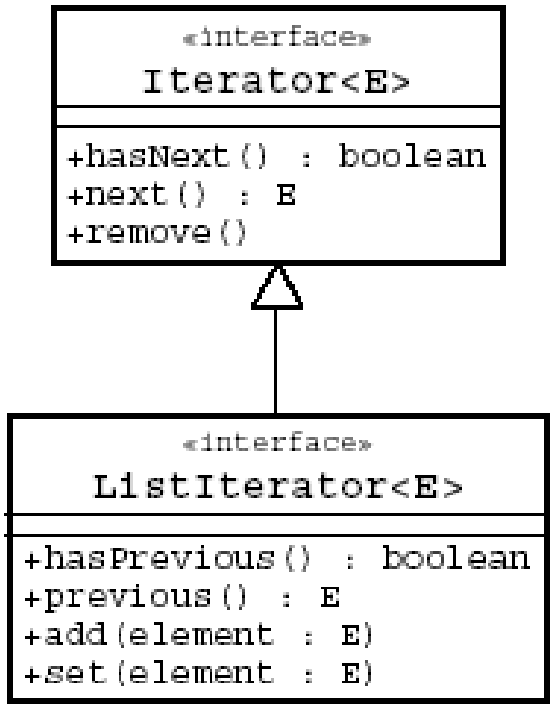
```
javac GenericsWarning.java
Note: GenericsWarning.java uses unchecked or
 unsafe operations.
Note: Recompile with -Xlint:unchecked for
 details.


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.
- An `Iterator` of a `Set` is unordered.
- A `ListIterator` of a `List` can be scanned forwards (using the next method) or backwards (using the previous method).

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

The Iterator Interface Hierarchy



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.

Enhanced for Loop

- An enhanced for loop can look like this:
- Using iterators:

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

- Using enhanced for loop in collections:

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

Exercise

- Do lab 9: exercise 1,2

Points to Remember

Check Your Progress

- Write a program that uses command-line arguments and system properties
- Write a program that reads from standard input
- Create, read, and write files
- Describe the collection API
- Write a program to iterate over a collection

Points to Remember

Check Your Progress

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

Experience. Results.