Course Booklet for Java Programming Language

Know how of Java Programming Language
By Emertxe

Version 2.0 (Jan 6, 2015)

Table of Contents

Chapter 1: Getting Started

1.1: Objectives

Upon completion of this module, you should be able to:

- 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

1.2: What Is the Java Technology?

The Java technology is:

- A programming language
- A development environment
- An application environment
- A deployment environment

1.2.1:Primary Goals of the Java Technology

Java technology provides the following:

- A language that is easy to program because it:
  - Eliminates many pitfalls of other languages, such as pointer arithmetic and memory management that affect the robustness of the code
  - Is object-oriented to help you visualize the program in real-life terms
  - Enables you to streamline the code
- An interpreted environment resulting in the following benefits:
  - Speed of development – Reduces the compile-link-load-test cycle
  - Code portability – Enables you to write code that can be run on multiple operating systems on any certified JVM
- A way for programs to run more than one thread of activity
- A means to change programs dynamically during their runtime life by enabling them to download code modules
- A means of ensuring security by checking loaded code modules

The Java technology architecture uses the following features to fulfill the previously listed goals:

- The JVM
- Garbage collection
- The JRE
- JVM tool interface

1.3: The Java Virtual Machine

The Java Virtual Machine Specification defines the JVM as:

An imaginary machine that is implemented by emulating it in software on a real machine. Code for the JVM is stored in .class files, each of which contains code for at most one public class.

This specification enables the Java software to be platform-independent because the compilation is done for a generic machine, known as the JVM. You can emulate this generic machine in software to run on various existing computer systems or implement it in hardware.

The compiler takes the Java application source code and generates bytecodes. Bytecodes are machine code instructions for the JVM. Every Java technology interpreter, regardless of whether it is a Java technology development tool or a web browser that can run applets, has an implementation of the JVM.

The JVM design enables the creation of implementations for multiple operating environments. For example, Sun Microsystems provides implementations of the JVM for the Solaris OS and the Linux and Microsoft Windows operating environments.

1.4: Garbage Collection

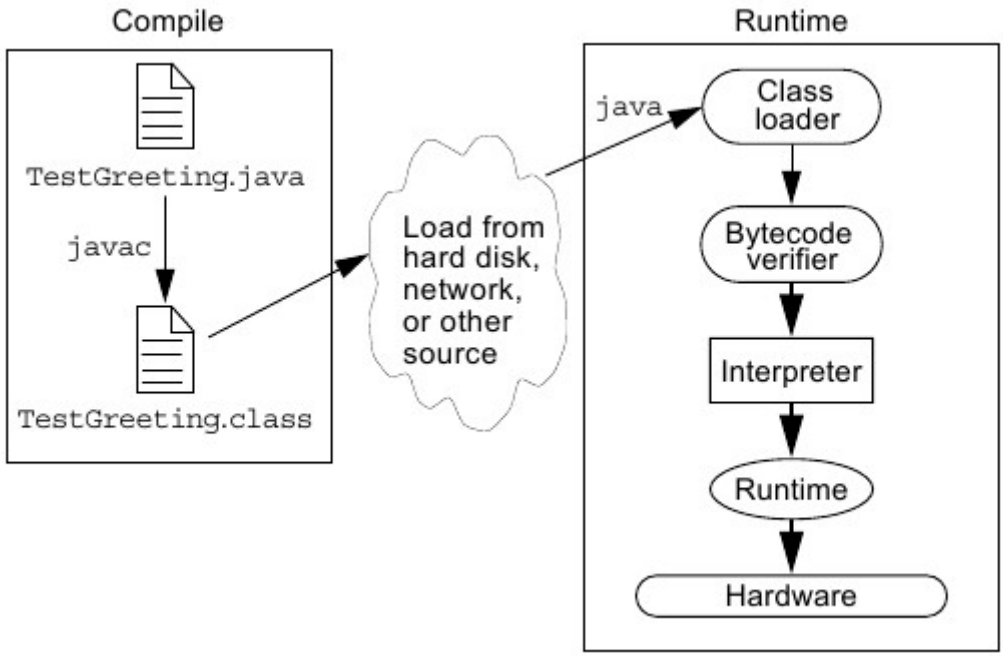
Garbage collection happens automatically during the lifetime of a Java technology program, eliminating the need to deallocate memory and avoiding memory leaks. However, garbage collection schemes can vary dramatically across JVM implementations.

1.5: The Java Runtime Environment

Java software source files are compiled in the sense that they are converted into a set of bytecodes from the text format in which you write them. The bytecodes are stored in .class files.

At runtime, the bytecodes that make up a Java software program are loaded, checked, and run in an interpreter. In the case of applets, you can download the bytecodes, and then they are interpreted by the JVM built into the browser. The interpreter has two functions: it executes bytecodes

and it makes the appropriate calls to the underlying hardware.



1.5.1: JVM Tasks

The following section provides a more comprehensive discussion of the three main tasks performed by the JVM:

- Loads code – Performed by the class loader
- Verifies code – Performed by the bytecode verifier
- Executes code – Performed by the runtime interpreter

1.5.2:The Class Loader

The class loader loads all classes needed for the execution of a program.

1.5.3:The Bytecode Verifier

Java software code passes several tests before running on your machine. The JVM puts the code through a bytecode verifier that tests the format of code fragments and checks code fragments for illegal code, which is code that forges pointers, violates access rights on objects, or attempts to change object type.

1.6: A Simple Java Application

Like any other programming language, you use the Java programming language to create applications. A simple Java application that prints a greeting to the world.

SampleHello.Java

```
public class SampleHello
{
  public static void main(String[] args)
  {
    System.out.println(“Hello world...!”);
  }
}
```

1.6.1:Compiling and Running the SampleHello Program

After you have created the SampleHello.java source file, compile it with the following line:

javac SampleHello.java

If the compiler does not return any messages, the new file SampleHello.class is stored in the same directory as the source file, unless specified otherwise. To run your SampleHello application, use the Java technology interpreter. The executables for the Java technology tools (javac, java, javadoc, and so on) are located in the bin directory.

java SampleHello

Chapter 2: Object-Oriented Programming

2.1: Objectives

Upon completion of this module, you should be able to:

- 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 API online documentation

2.2: Software Engineering

Software engineering is a difficult and often unruly discipline. For the past half-century, computer scientists, software engineers, and system architects have sought to make creating software systems easier by providing reusable code.

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)

2.3: Abstraction

Software design has moved from low-level constructs, such as writing in machine code, toward much higher levels. There are two interrelated forces that guided this process: simplification and abstraction. Simplification was at work when early language designers built high-level language constructs, such as the IF statements and FOR loops, out of raw machine codes. Abstraction is the force that hides private implementation details behind public interfaces.

The concept of abstraction led to the use of subroutines (functions) in high-level languages and to the pairing of functions and data into objects. At higher levels, abstraction led to the development of frameworks and APIs.

2.4: Classes as Blueprints for Objects

A class is a software blueprint that you can use to instantiate (that is, create) many individual objects. A class defines the set of data elements (attributes) that define the objects, as well as the set of behaviors or functions (called methods) that manipulate the object or perform interactions between related objects. Together, attributes and methods are called members.

2.4.1:Declaring Java Technology Classes

The Java technology class declaration takes the following form:

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

2.4.2:Declaring Attributes

The declaration of an object attribute takes the following form:

<modifier>* <type> <name> [= <initial_value>];

2.4.3:Declaring Methods

To define methods, the Java programming language uses an approach that is similar to other languages, particularly C and C++. The declaration takes the following basic form:

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

2.4.4:Accessing Object Members

2.5: Encapsulation

Encapsulation is the methodology of hiding certain elements of the implementation of a class but providing a public interface for the client software. This is an extension of information hiding because the information in the data attributes is a significant element of a class's implementation.

2.6: Source File Layout

A Java technology source file takes the following form:

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

The order of these items is important. That is, any import statements must precede all class

declarations and, if you use a package declaration, it must precede both the classes and imports.

2.7: Software Packages

Most software systems are large. It is common to group classes into packages to ease the management of the system. UML includes the concept of packages in its modeling language. Packages can contain classes as well as other packages that form a hierarchy of packages.

2.7.1:The package Statement

The Java technology programming language provides the package statement as a way to group related classes. The package statement takes the following form:

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

2.7.2:The import Statement

The import statement takes the following form:

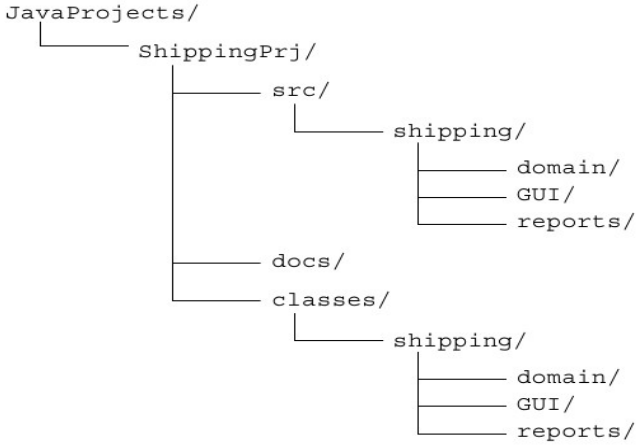
import <pkg_name>[.<sub_pkg_name>].<class_name>;

OR

import <pkg_name>[.<sub_pkg_name>].*;

2.7.3:Compiling Using the -d Option

Normally, the Java compiler places the class files in the same directory as the source files. You can reroute the class files to another directory using the -d option of the javac command. The simplest way to compile files within packages is to be working in the directory one level above the beginning of the package.



for ex:

cd JavaProjects/ShippingPrj/src

javac -d ../classes shipping/domain/*.java

2.7.4:Deployment

You can deploy an application on a client machine without manipulating the user's CLASSPATH environment variable. Usually, this is best done by creating an executable Java archive (JAR) file. To create an executable JAR file, you must create a temporary file that indicates the class name that contains your main method, like this:

Main-Class: mypackage.MyClass

Next, build the JAR file as normal except that you add an additional option so that the contents of this temporary file are copied into the META-INF/MANIFEST.MF file. Do this using the “m” option, like this:

jar cmf tempfile MyProgram.jar

Finally, the program can be run simply by executing a command like this:

java -jar /path/to/file/MyProgram.jar

2.8: Using the Java Technology API Documentation

A set of HTML files document the supplied API. The layout of this documentation is hierarchical, so that the home page lists all the packages as hyperlinks. When you select a particular package hotlink, the classes that are members of that package are listed. Selecting a class hotlink from a package page presents a page of information about that class.

The main sections of a class document include the following:

- The class hierarchy
- A description of the class and its general purpose
- A list of attributes
- A list of constructors
- A list of methods
- A detailed list of attributes with descriptions
- A detailed list of constructors with descriptions and formal parameter lists
- A detailed list of methods with descriptions and formal parameter lists

Chapter 3: Identifiers, Keywords, and Types

3.1: Objectives

Upon completion of this module, you should be able to:

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

3.2: Semicolons, Blocks, and White Space

In the Java programming language, a statement is one or more lines of code terminated with a semicolon (;).

A block, sometimes called a compound statement, is a group of statements bound by opening and closing braces ({ }).

3.3: Identifiers

In the Java programming language, an identifier is a name given to a variable, class, or method. Identifiers start with a letter, underscore (_), or dollar sign ($). Subsequent characters can be digits. Identifiers are case-sensitive and have no maximum length.

3.4: Java Programming Language Keywords

Keywords have special meaning to the Java technology compiler. They identify a data type name or program construct name.

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

3.5: Basic Java Programming Language Types

The Java programming language has many built in data types. These fall into two broad categories: class types and primitive types. Primitive types are simple values, are not objects. Class types are used for more complex types, including all of the types that you declare yourself. Class types are used to create objects.

3.5.1:Primitive Types

The Java programming language defines eight primitive data types, which can be considered in four categories:

- Logical – boolean
- Textual – char
- Integral – byte, short, int, and long

● Floating point – double and float

3.5.2:Java Reference Types

As you have seen, there are eight primitive Java types: boolean, char, byte, short, int, long, float, and double. All other types refer to objects rather than primitives. Variables that refer to objects are reference variables.

3.6: Constructing and Initializing Objects

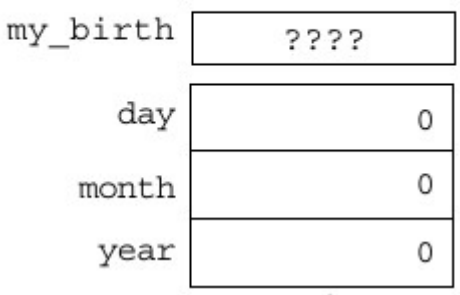
3.6.1:Memory Allocation and Layout

In a method body, the following declaration allocates storage only for the reference

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

The keyword new in the following example implies allocation and initialization of storage

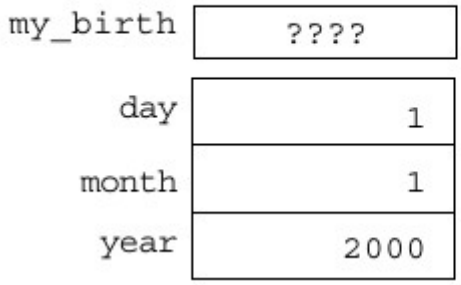
```
MyDate my_birth = new MyDate(22, 7, 1964);
```



3.6.2:Explicit Attribute Initialization

If you place simple assignment expressions in your member declarations, you can initialize members explicitly during construction of your object.

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

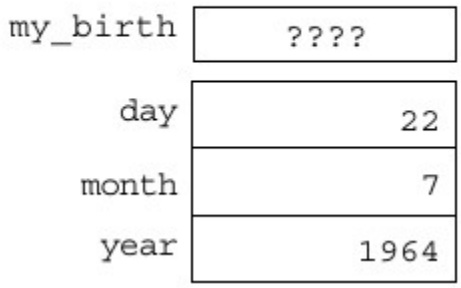


3.6.3:Executing the Constructor

The final stage of initializing a new object is to call the constructor. The constructor enables you to override the default initialization. You can perform computations. You can also pass arguments into the construction process so that the code that requests the construction of the new object

can control the object it creates.

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



3.6.4:Assigning a Variable

3.6.5:Assigning References

3.7: The this Reference

Two uses of the this keyword are:

- To resolve ambiguity between instance variables and parameters
- To pass the current object as a parameter to another method

3.8: Java Programming Language Coding Conventions

The following are the coding conventions of the Java programming language:

- Packages –
- Classes –
- Interfaces –
- Methods –
- Variables –
- Constants –
- Spacing –

- Comments –

Chapter 4: Expressions and Flow Control

4.1: 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

• 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

4.2: 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?

4.3: 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.

4.4: Variable Scope Example

4.5: 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

4.6: Initialization Before Use Principle

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

4.7: Operator Precedence

Operators	Associative
++ -- + *unary* - *unary* ~ ! (<data_type>)	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

4.8: Logical Operators

• The boolean operators are:

! – NOT

| – OR

& – AND

^ – 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
}
```

4.9: Bitwise Logical Operators

• The integer bitwise operators are:

~ – Complement

^ – XOR

& – AND

| – OR

4.10: 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.

4.11: Left-Shift Operator <<

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

- 128 << 1 returns 128 * 2 1 = 256
- 16 << 2 returns 16 * 2 2 = 64

4.12: 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.

4.13: 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
```

4.14: 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

4.15: Simple if, else Statements

The if statement syntax:

if (<boolean_expression>)

<statement_or_block>

4.16: Complex if, else Statements

4.16.1: The if-else statement syntax:

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

4.16.2: The if-else-if statement syntax:

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

4.17: 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;]

}

4.18: Looping Statements

4.18.1: The for loop:

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

4.18.2: The while loop:

while (<test_expr>)
<statement_or_block>

4.18.3: The do/while loop:

do
<statement_or_block>
while (<test_expr>);

4.19: Special Loop Flow Control

• The break [<label>]; command

• The continue [<label>]; command

• The <label> : <statement> command, where

<statement> should be a loop

4.19.1: The break Statement

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

4.19.2: The continue Statement

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

4.19.3: Using break Statements with Labels

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

4.19.4: Using continue Statements with Labels

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

Chapter 5: Arrays

5.1: 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

5.2: Relevance

- What is the purpose of an array?

5.3: Declaring Arrays

• Group data objects of the same type.
• Declare arrays of primitive or class types:

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

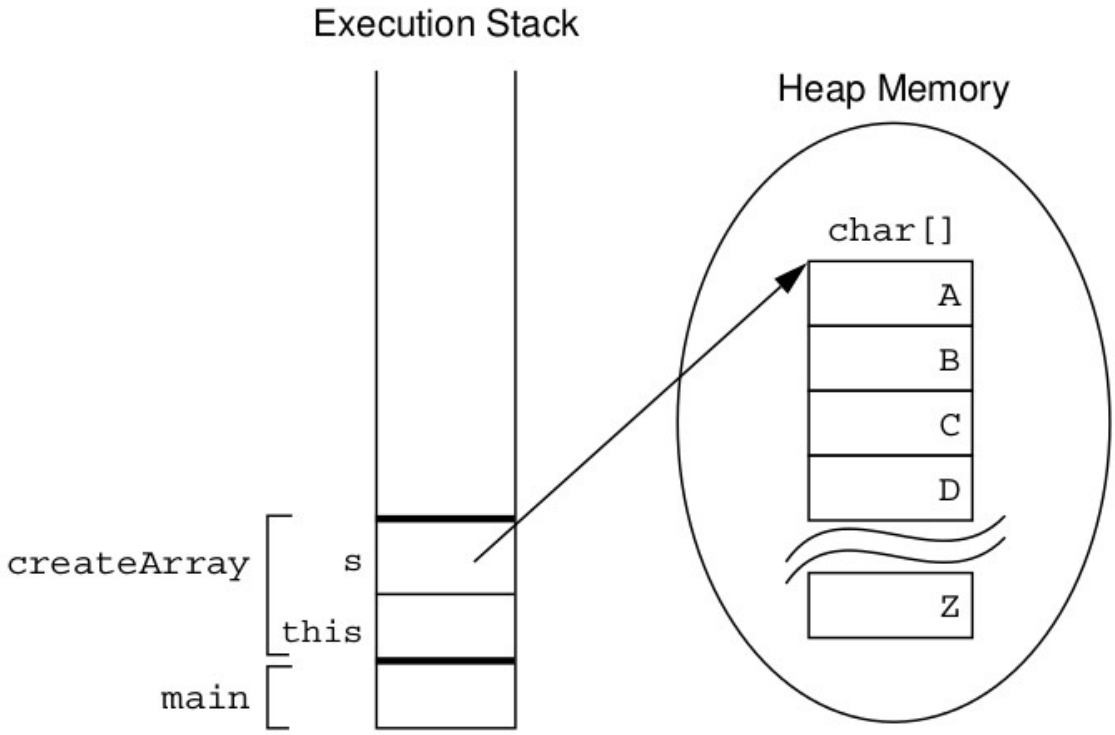
• Create space for a reference.
• An array is an object; it is created with new.

5.4: 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;
}
```

5.5: Creating an Array of Character Primitives

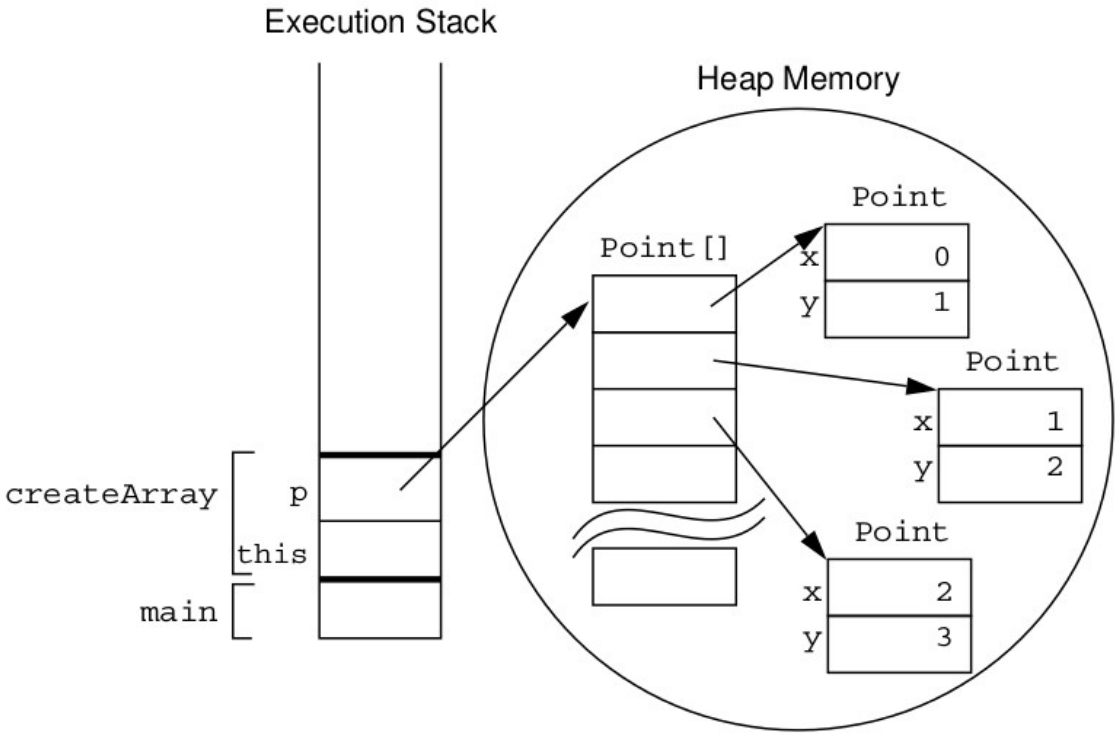


5.6: Creating Reference Arrays

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;
}
```

5.7: Creating an Array of Character Primitives With Point Objects


Execution Stack
Heap Memory
createArray
p
this
main
Point[]
Point
x
0
y
1
Point
x
1
y
2
Point
x
2
y
3

5.8: 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"
};
```

5.9: 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
  ```

- 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];
  ```

5.10: 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]);
}
}
```

5.11: Using the Enhanced for Loop

Java 2 Platform, Standard Edition (J2SETM) 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.

5.12: 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];
```

5.13: 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);
```

Chapter 6: Class Design

6.1: Objectives

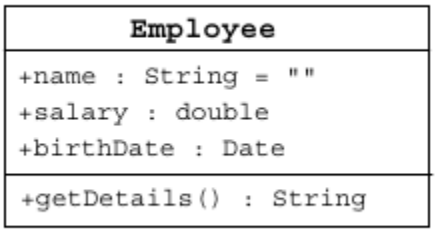
Upon completion of this module, you should be able to:

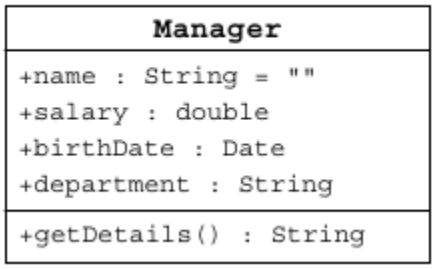
- Define inheritance, polymorphism, overloading, overriding, and virtual method invocation
- Use the access modifiers protected and the default (package- friendly)
- Describe the concepts of constructor and method overloading
- Describe the complete object construction and initialization operation

6.2: Subclassing

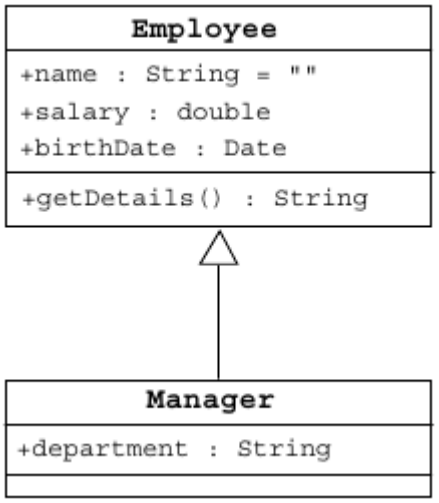
In programming, you often create a model of something (for example, an employee), and then need a more specialized version of that original model. For example, you might want a model for a manager. A manager is an employee, but an employee with additional features.

In





object-oriented languages, special mechanisms are provided that enable you to define a class in terms of a previously defined class.



Class Diagrams for Employee and Manager Using Inheritance

6.2.1:Single Inheritance

The Java programming language permits a class to extend one other class only. This restriction is called single inheritance. The relative merits of single and multiple inheritance are the subject of extensive discussions among object-oriented programmers.

6.3: Access Control

Variables and methods can be at one of four access levels: public, protected, default, or private. Classes can be at the public or default

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

6.4: Overriding Methods

If a method is defined in a subclass so that the name, return type, and argument list match exactly those of a method in the parent class, then the new method is said to override the old one.

6.4.1:Overridden Methods Cannot Be Less Accessible

6.4.2:Invoking Overridden Methods

6.5: Polymorphism

An object has only one form (the one that is given to it when constructed). However, a variable is polymorphic because it can refer to objects of different forms. The Java programming language, like most object-oriented languages, actually permits you to refer to an object with a variable that is one of the parent class types. So you can say:

Employee e = new Manager(); //legal

6.6: The instanceof Operator

Given that you can pass objects around using references to their parent classes, sometimes you might want to know what actual objects you have. This is the purpose of the instanceof operator.

6.6.1:Casting Objects

In circumstances where you have received a reference to a parent class, and you have determined that the object is, in fact, a particular subclass by using the instanceof operator, you can access the full functionality of the object by casting the reference.

6.7: Overloading Methods

In some circumstances, you might want to write several methods in the same class that do the same basic job with different arguments. Consider a simple method that is intended to output a textual representation of its argument. This method could be called println().

By reusing the method name, you end up with the following methods:

public void println(int i)

public void println(float f)

public void println(String s)

6.7.1:Methods Using Variable Arguments

6.8: Overloading Constructors

When an object is instantiated, the program might be able to supply multiple constructors based on the data for the object being created.

6.8.1:Constructors Are Not Inherited

6.8.2:Invoking Parent Class Constructors

6.9: Constructing and Initializing Objects: A Slight Reprise

First, the memory for the complete object is allocated and the default values for the instance variables are assigned. Second, the top-level constructor is called and follows these steps recursively down the inheritance tree:

1. Bind constructor parameters.

2. If explicit this(), call recursively, and then skip to Step 5.

3. Call recursively the implicit or explicit super(...), except for Object because Object has no parent class.

4. Execute the explicit instance variable initializers.

5. Execute the body of the current constructor.

6.10: The Object Class

The Object class is the root of all classes in the Java technology programming language. If a class is declared with no extends clause, then the compiler adds implicitly the code extends Object to the declaration;

6.10.1: The equals Method

6.10.2: The toString Method

6.11: Wrapper Classes

The Java programming language provides wrapper classes to manipulate primitive data elements as objects. Such data elements are wrapped in an object created around them. Each Java primitive data type has a corresponding wrapper class in the java.lang package. Each wrapper class object encapsulates a single primitive value.

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

6.11.1: Autoboxing of Primitive Types

If you have to change the primitive data types to their object equivalents (called boxing), then you need to use the wrapper classes. Also to get the primitive data type from the object reference (called unboxing), you need to use the wrapper class methods.

Chapter 7: Advanced Class Features

7.1: Objectives

Upon completion of this module, you should be able to:

- 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

7.2: The static Keyword

The static keyword declares members (attributes, methods, and nested classes) that are associated with the class rather than the instances of the class.

7.2.1:Class Attributes

You achieve a shared effect by marking the variable with the keyword static. Such a variable is sometimes called a class variable to distinguish it from a member or instance variable, which is not shared.

7.2.2:Class Methods

Sometimes you need to access program code when you do not have an instance of a particular object available. A method that is marked using the keyword static can be used in this way and is sometimes called a class method.

7.2.3:Static Initializers

A class can contain code in static blocks that are not part of normal methods. Static block code executes once when the class is loaded. If a class contains more than one static block, they are executed in the order of their appearance in the class.

7.3: The final Keyword

7.3.1:Final Classes

The Java programming language permits you to apply the keyword final to classes. If you do this, the class cannot be subclassed.

7.3.2:Final Methods

You can also mark individual methods as final. Methods marked final cannot be overridden. For security reasons, you should make a method final if the method has an implementation that should not be changed and is critical to the consistent state of the object.

7.3.3:Final Variables

If a variable is marked as final, the effect is to make it a constant. Any attempt to change the value of a final variable causes a compiler error.

7.3.3.1: Blank Final Variables

A blank final variable is a final variable that is not initialized in its declaration. The initialization is delayed. A blank final instance variable must be assigned in a constructor, but it can be set once only. A blank final variable that is a local variable can be set at any time in the body of the method, but it can be set once only.

7.4: Enumerated Types

A common idiom in programming is to have finite set of symbolic names that represent the values of an attribute.

For example, to represent the suits of playing cards you might create a set of symbols: SPADES,

HEARTS, CLUBS, and DIAMONDS. This is often called an enumerated type.

7.4.1:Old-Style Enumerated Type Idiom

7.4.2:The New Enumerated Type

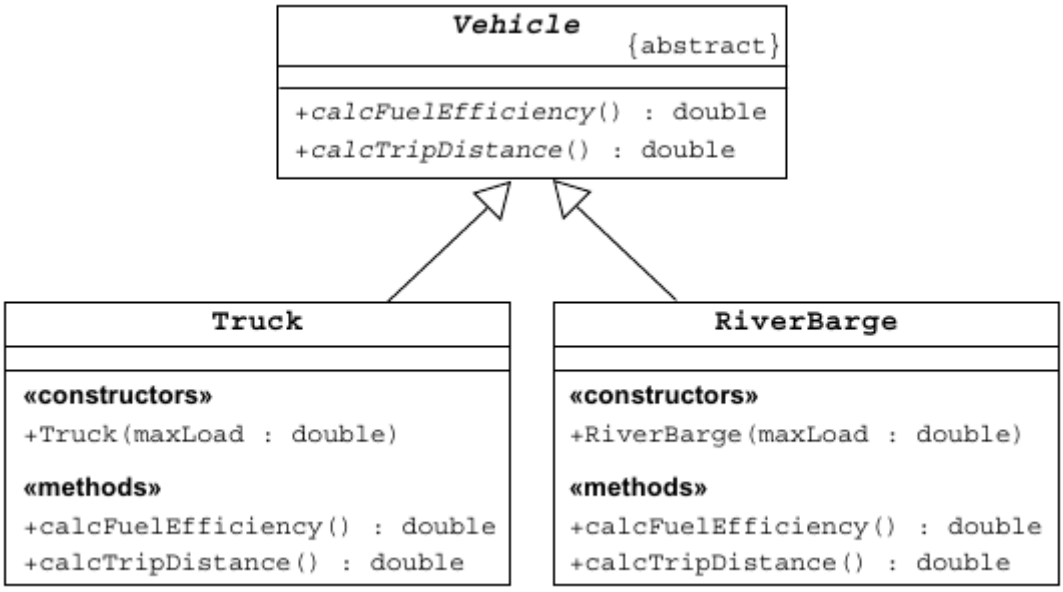
7.4.3:Advanced Enumerated Types

7.5: Static Imports

J2SE version 5.0 provides the static import feature that enables unqualified access to static members without having to qualify them with the class name.

7.6: Abstract Classes

The Java programming language enables a class designer to specify that a superclass declares a method that does not supply an implementation. This is called an abstract method. The implementation of this method is supplied by the subclasses. Any class with one or more abstract methods is called an abstract class .



7.7: Interfaces

The public interface of a class is a contract between the client code and the class that provides the service. Concrete classes implement each method. However, an abstract class can defer the implementation by declaring the method to be abstract, and a Java interface declares only the contract and no implementation.

A concrete class implements an interface by defining all methods declared by the interface. Many classes can implement the same interface. These classes do not need to share the same class hierarchy. Also, a class can implement more than one interface

7.7.1:Uses of Interfaces

You use interfaces to:

● Declare methods that one or more classes are expected to implement

● Reveal an object’s programming interface without revealing the actual body of the class (this can be useful when shipping a package of classes to other developers)

● Capture similarities between unrelated classes without forcing a class relationship

● Simulate multiple inheritance by declaring a class that implemensts several interfaces

Chapter 8: Exceptions and Assertions

8.1: Objectives

Upon completion of this module, you should be able to:

- 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

8.2: Exceptions

Exceptions are a mechanism used by many programming languages to describe what to do when something unexpected happens. Typically, something unexpected is an error of some sort, for example when a method is invoked with unacceptable arguments, or a network connection fails, or the user asks to open a non-existent file.

The Java programming language provides two broad categories of exceptions, known as checked and unchecked exceptions.

Checked exceptions are those that the programmer is expected to handle in the program, and that arise from external conditions that can readily occur in a working program. Examples would be a requested file not being found or a network failure.

Unchecked exceptions might arise from conditions that represent bugs, or situations that are considered generally too difficult for a program to handle reasonably. They are called unchecked because you are not required to check for them or do anything about them if they occur. Exceptions that arise from a category of situations that probably represent bugs are called runtime exceptions. An example of a runtime exception is attempting to access beyond the end of an array.

8.2.1:Exception Example

8.3: The try-catch Statement

The Java programming language provides a mechanism for figuring out which exception was thrown and how to recover from it.

8.3.1:Using Multiple catch Clauses

There can be multiple catch blocks after a try block, each handling a different exception type.

8.4: Call Stack Mechanism

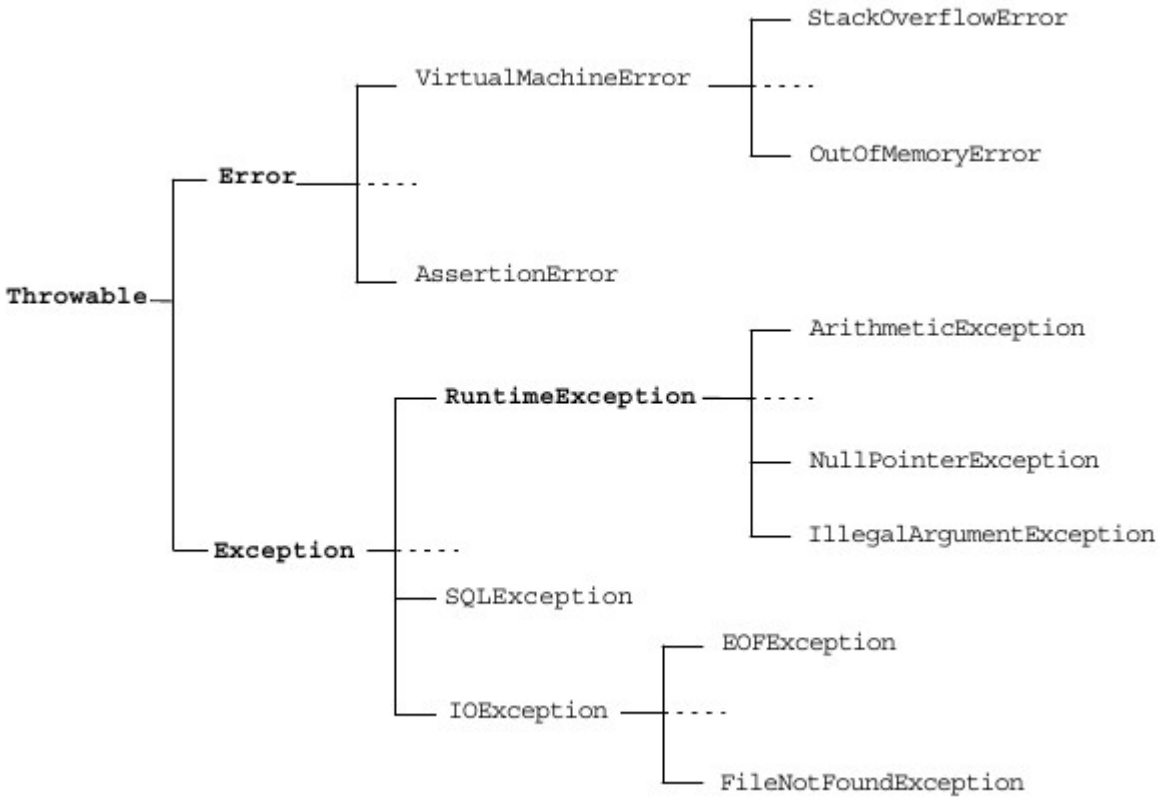
If a statement throws an exception, and that exception is not handled in the immediately enclosing method, then that exception is thrown to the calling method. If the exception is not handled in the calling method, it is thrown to the caller of that method. This process continues. If the exception is still not handled by the time it gets back to the main() method and main() does not handle it, the exception terminates the program abnormally.

8.5: The finally Clause

The finally clause defines a block of code that always executes, regardless of whether an exception was caught.

8.6: Exception Categories

The class java.lang.Throwable acts as the parent class for all objects that can be thrown and caught using the exception-handling mechanisms. Methods defined in the Throwable class retrieve the error message associated with the exception and print the stack trace showing where the exception occurred. There are three key subclasses of Throwable: Error, RuntimeException, and Exception .



8.7: Common Exceptions

The Java programming language provides several predefined exceptions. Some of the more common exceptions are:

- NullPointerException

- FileNotFoundException

- NumberFormatException

- ArithmeticException

- SecurityException

8.8: The Handle or Declare Rule

A method can declare that an exception might be thrown in the body of the method with a throws clause as follows:

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

Following the keyword throws is a list of all the exceptions that the method can throw back to its caller. Although only one exception is shown here, you can use a comma-separated list if this method throws multiple possible exceptions, like this:

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

8.9: Method Overriding and Exceptions

When overriding a method that throws exceptions, the overriding method can declare only exceptions that are either the same class or a subclass of the exceptions.

8.10: Creating Your Own Exceptions

User-defined exceptions are created by extending the Exception class. Exception classes contain anything that a regular class contains.

To throw an exception that you have created, use the following syntax:

```
throw new ServerTimedOutException("Could not connect", 80);
```

Always instantiate the exception on the same line on which you throw it, because the exception can carry line number information that will be added when the exception is created.

8.11: Assertions

Two syntactic forms are permitted for assertion statements, these are:

assert <boolean_expression> ;

assert <boolean_expression> : <detail_expression> ;

In either case, if the Boolean expression evaluates to false, then an AssertionError is thrown. This error should not be caught, and the program should be terminated abnormally.

If the second form is used, then the second expression, which can be of any type, is converted to a String and used to supplement the message that prints when the assertion is reported.

8.11.1: Recommended Uses of Assertions

Assertions should be used generally to verify the internal logic of a single method or a small group of tightly coupled methods. Generally, assertions should not be used to verify that code is being used correctly, but rather that its own internal expectations are being met.

8.11.1.1: Internal Invariants

8.11.1.2: Control Flow Invariants

8.11.1.3: Postconditions and Class Invariants

8.11.2: Controlling Runtime Evaluation of Assertions

One of the major advantages of assertions over using exceptions is that assertion checking can be disabled at runtime. If this is done, then there is no overhead involved in checking the assertions, and the code runs as fast as if the assertion test had never been present. This is better than using conditional compilation for two reasons. First, it does not require a different compile phase for production. Second, if required, the assertions can be re-enabled in the field to determine if the assertions have become invalid due to some unforeseen environmental effect.

By default, assertions are disabled. To turn assertions on, use either of these forms:

java -enableassertions MyProgram

java -ea MyProgram

Chapter 9: Collections and Generics Framework

9.1: Objectives

Upon completion of this module, you should be able to:

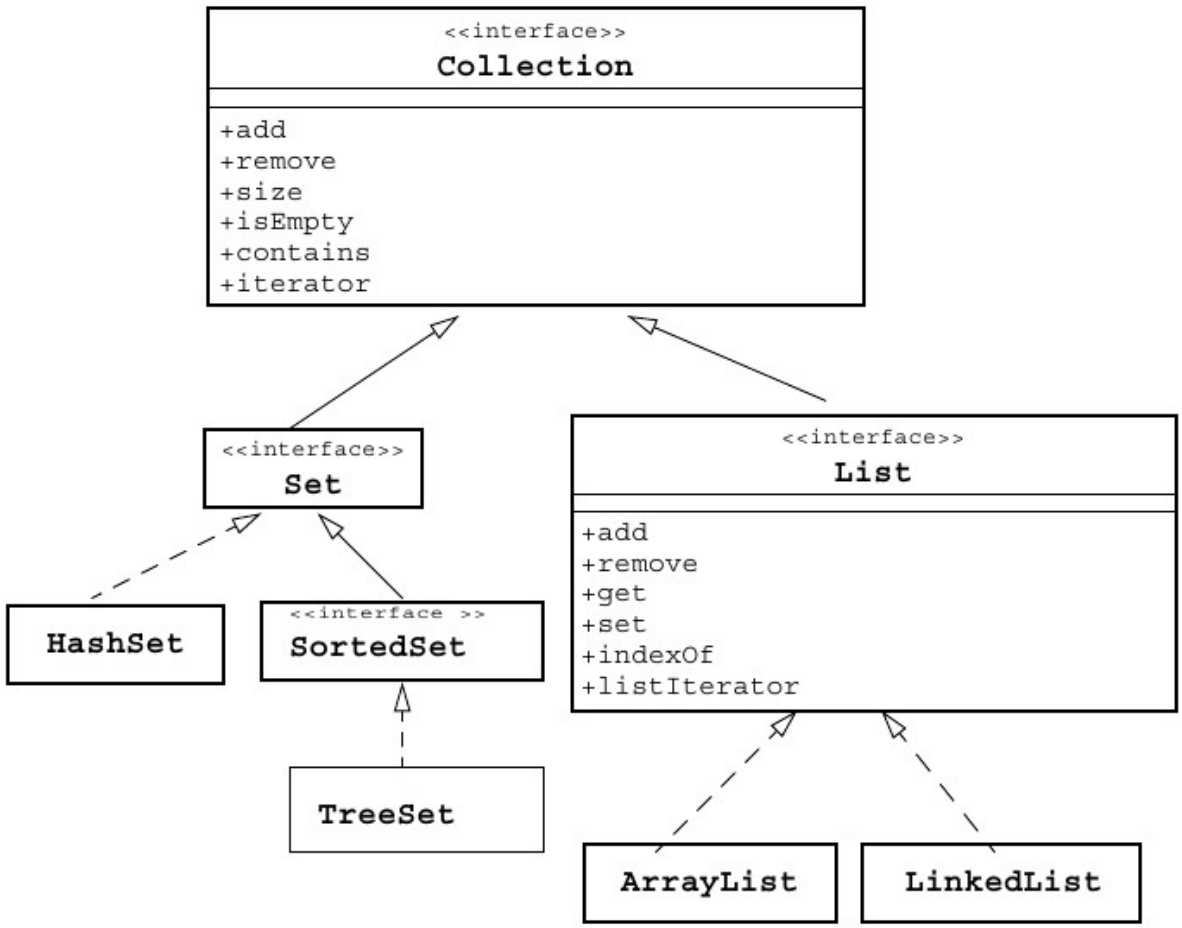
- Describe the implementations of the core interfaces in the Collections framework
- Examine the Map interface
- Examine the legacy collection classes
- Create custom ordering by implementing the Comparable and Comparator interfaces
- Use generic collections
- Use type parameters in generic classes
-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

9.2: The Collections API

A collection is a single object managing a group of objects. The objects in the collection are called elements. Typically, collections deal with many types of objects, all of which are of a particular kind (that is, they all descend from a common parent type).

The Collections API contains interfaces that group objects as one of the following:

- Collection – A group of objects known as 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



9.3: Collection Implementations

There are several general purpose implementations of the core interfaces (Set, List, Map and Deque) in the Collections framework.

	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

9.3.1:List

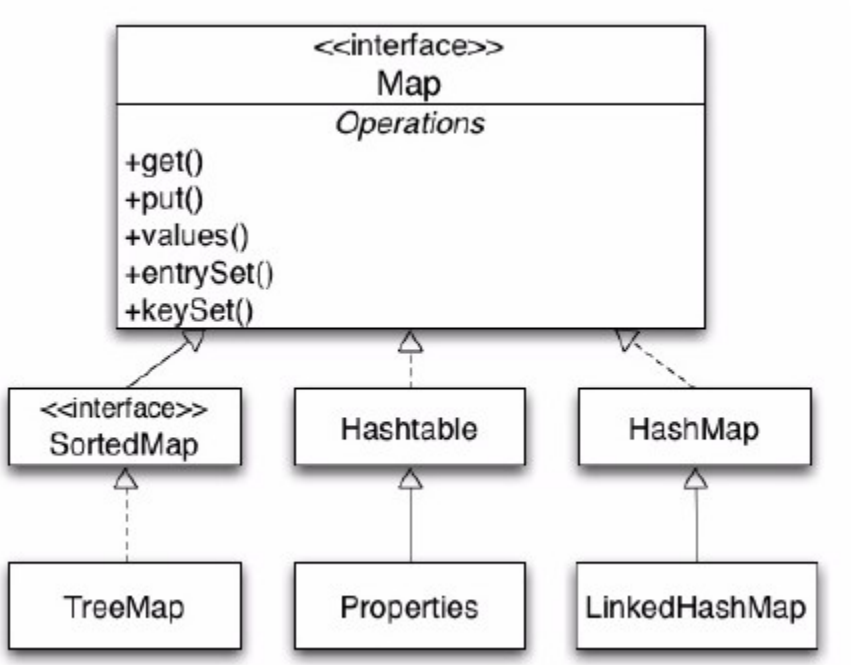
9.3.2:Set

9.4: The Map Interface

Maps are sometimes called associative arrays. A Map object describes mappings from keys to values. By definition, a Map object does not allow duplicate keys and a key can map to one value at most.

The Map interface provides three methods that allow map contents to be viewed as collections are:

- entrySet – Returns a Set that contains all the key value pairs.
- keySet – Returns a Set of all the keys in the map.
- values – Returns a Collection containing all the values contained in the map.



9.5: Legacy Collection Classes

● The Vector class implements the List interface.

● The Stack class is an extension of Vector that adds typical stack operations: push, pop, and peek.

● The Hashtable is an implementation of Map.

● The Properties class is an extension of Hashtable that only uses Strings for keys and values.

● Each of these collections has an elements method that returns an Enumeration object. The Enumeration is an interface similar to, but incompatible with, the Iterator interface. For example, hasNext is replaced by hasMoreElements in the Enumeration interface.

9.6: 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 is used to specify order relation. It can also be used to override natural ordering. These interfaces are useful for sorting the elements in a collection.

9.6.1:The Comparable Interface

The Comparable interface is a member of the java.lang package. When you declare a class, the JVM implementation has no means of determining the order you intend for objects of that class. You can, by implementing the Comparable interface, provide order to the objects of any class. You can sort collections that contain objects of classes that implement the Comparable interface.

9.6.2:The Comparator Interface

The Comparator interface provides greater flexibility with ordering. For example, if you consider the Student class, the sorting of students was restricted to sorting on GPAs. It was not possible to sort the student based on first name or some other criteria. This section demonstrates how sorting flexibility can be enhanced using the Comparator interface.

9.7: Generics

It provides information for the compiler about the type of collection used. Hence, type checking is resolved automatically at run time. This eliminates the explicit casting of the data types to be used in the collection. With the addition of autoboxing of primitive types, you can use generics to write simpler and more understandable code.

9.7.1:Generics: Examining Type Parameters

This section examines the use of type parameters in (the class, constructor and method declarations of) generic classes. Table compares the non- generic version (pre-Java SE 5.0 platform) and the generic version (since the Java SE 5.0 platform) of the ArrayList class.

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 (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> a3;`
		
		`ArrayList <Date> a4;`

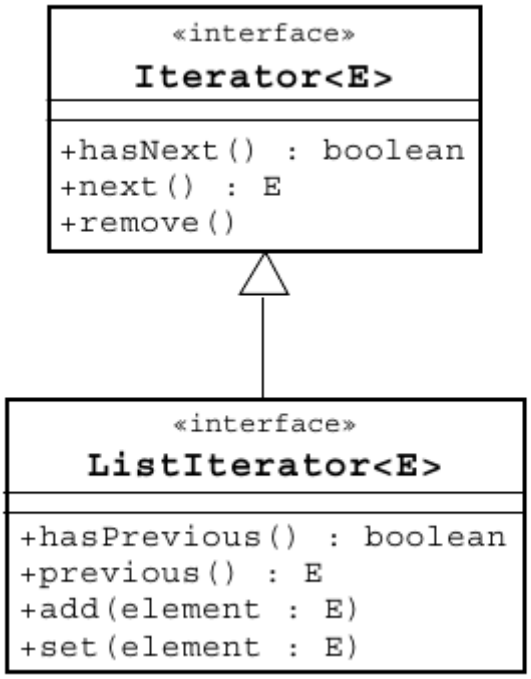
Instance declaration examples	`list1 = new ArrayList(10);`
	
	`list2 = new ArrayList(10);`
		`a3= new ArrayList<String> (10);`
		
		`a4= new ArrayList<Date> (10);`

9.7.2:Generics: Refactoring Existing Non-Generic Code

With generic collections, you can specify generic types without type arguments, which are called raw types. This feature is allowed to provide compatibility with the non-generic code. At compile time, all the generic information from the generic code is removed, leaving behind a raw type. This enables interoperability with the legacy code as the class files generated by the generic code and the legacy code would be the same. At runtime, ArrayList<String> and ArrayList<Integer> get translated into ArrayList, which is a raw type.

9.8: Iterators

You can scan (iterate over) a collection using an iterator. The basic Iterator interface enables you to scan forward through any collection. In the case of an iteration over a set, the order is non-deterministic. The order of an iteration over a list moves forward through the list elements. A List object also supports a ListIterator, which permits you to scan the list backwards and insert or modify list elements.



9.9: The Enhanced for Loop

The enhanced for loop makes traversing through a collection simple, understandable, and safe. The enhanced for loop eliminates the usage of separate iterator methods and minimizes the number of occurrences of the iterator variable.

Chapter 10: I/O Fundamentals

10.1: Objectives

Upon completion of this module, you should be able to:

- 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

10.2: System Properties

System properties are another mechanism used to parameterize a program at runtime. A property is a mapping between a property name and its value; both are strings. The Properties class represents this kind of mapping. The System.getProperties method returns the system properties object. The System.getProperty(String) method returns the string value of the property named in the String parameter. There is another method, System.getProperty(String, String), that enables you to supply a default string value (the second parameter), which is returned if the named property does not exist.

10.2.1: The Properties Class

An object of the Properties class contains a mapping between property names (String) and values (String). It has two main methods for retrieving a property value: getProperty(String) and getProperty(String, String); the latter method provides the capability of specifying a default value that is returned if the named property does not exist.

10.3: I/O Stream Fundamentals

A stream is a flow of data from a source to a sink. Typically, your program is one end of that stream, and some other node (for example, a file) is the other end. Sources and sinks are also called input streams and output streams, respectively. You can read from an input stream, but you cannot write to it. Conversely, you can write to an output stream, but you cannot read from it.

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

10.3.1: Data Within Streams

Java technology supports two types of data in streams: raw bytes or Unicode characters. Typically, the term stream refers to byte streams and the terms reader and writer refer to character streams.

10.4: Byte Streams

10.4.1: The InputStream Methods

The following methods provide access to the data from the input stream:

int read()

int read(byte[] buffer)

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

void close()

int available()

long skip(long n)

boolean markSupported()

void mark(int readlimit)

void reset()

10.4.2: The OutputStream Methods

The following methods write to the output stream:

void write(int)

void write(byte[] buffer)

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

void close()

void flush()

10.5: Character Streams

10.5.1: The Reader Methods

The following methods provide access to the character data from the reader:

int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
void close()
boolean ready()
long skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()

10.5.2: The Writer Methods

The following methods write to the writer:

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

Similar to output streams, writers include the close and flush methods.

void close()
void flush()

10.6: Node Streams

In the Java JDK, there are three fundamental types of nodes :

● Files

● Memory (such as arrays or String objects)

● Pipes (a channel from one process or thread [a light-weight process] to another; the output pipe stream of one thread is attached to the input pipe stream of another thread)

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`

10.7: I/O Stream Chaining

A program rarely uses a single stream object. Instead, it chains a series of streams together to process the data. Figure demonstrates an example input stream; in this case, a file stream is buffered for efficiency and then converted into data (Java primitives) items.

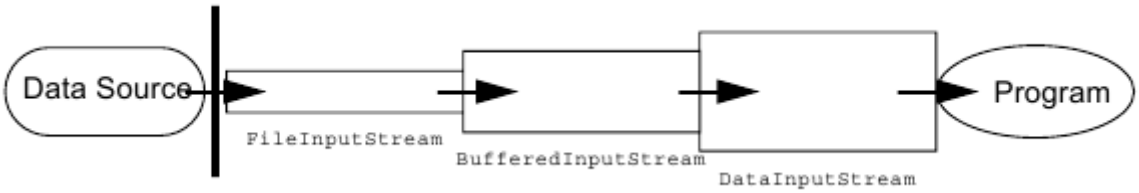
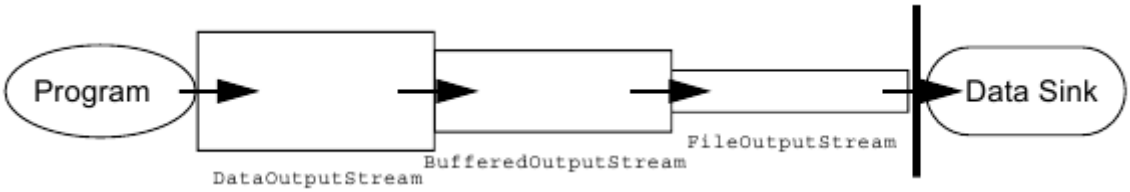


Figure demonstrates an example output stream; in this case, data is written, then buffered, and finally written to a file.



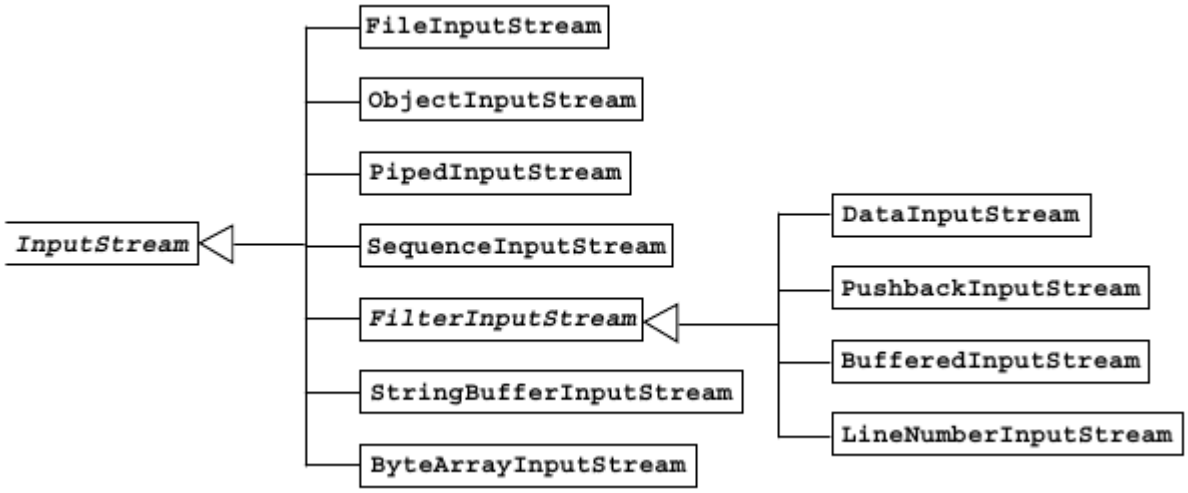
10.8: Processing Streams

A processing stream performs some sort of conversion on another stream. Processing streams are also known as filter streams. A filter input stream is created with a connection to an existing input stream. This is done so that when you try to read from the filter input stream object, it supplies you with characters that originally came from the other input stream object. This enables you to convert the raw data into a more usable form for your application.

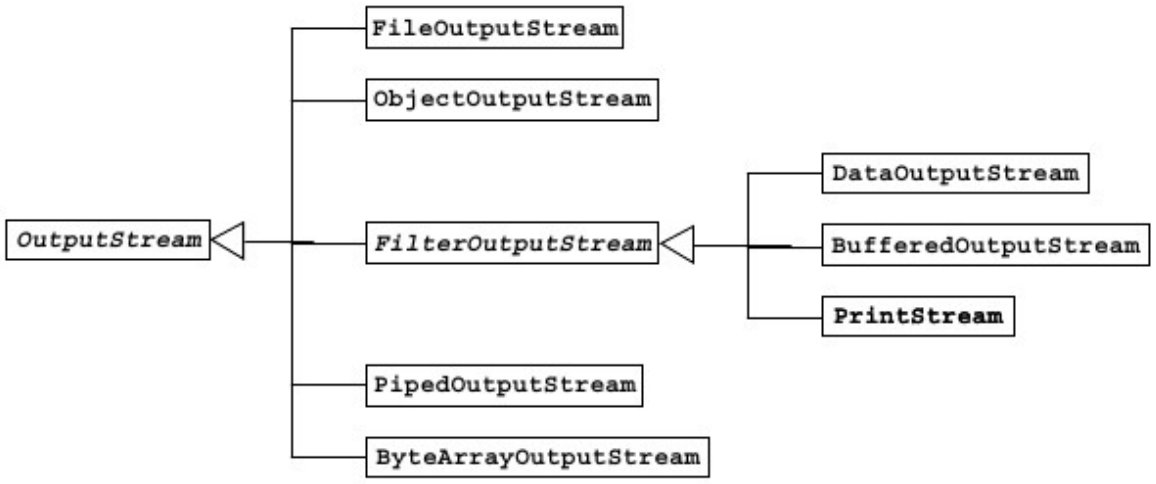
10.9: Basic Byte Stream Classes

The hierarchy of input byte stream classes in the java.io package.

The InputStream Class Hierarchy



The OutputStream Class Hierarchy



10.10: Serialization

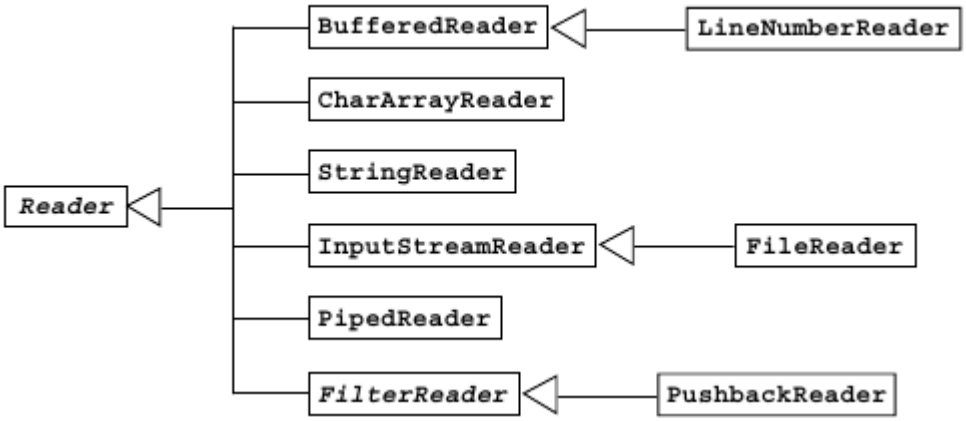
Saving an object to some type of permanent storage is called persistence. An object is said to be persistent-capable when you can store that object on a disk or tape or send it to another machine to be stored in memory or on disk. The non-persisted object exists only as long as the Java Virtual Machine is running.

Serialization is a mechanism for saving the objects as a sequence of bytes and later, when needed, rebuilding the byte sequence back into a copy of the object.

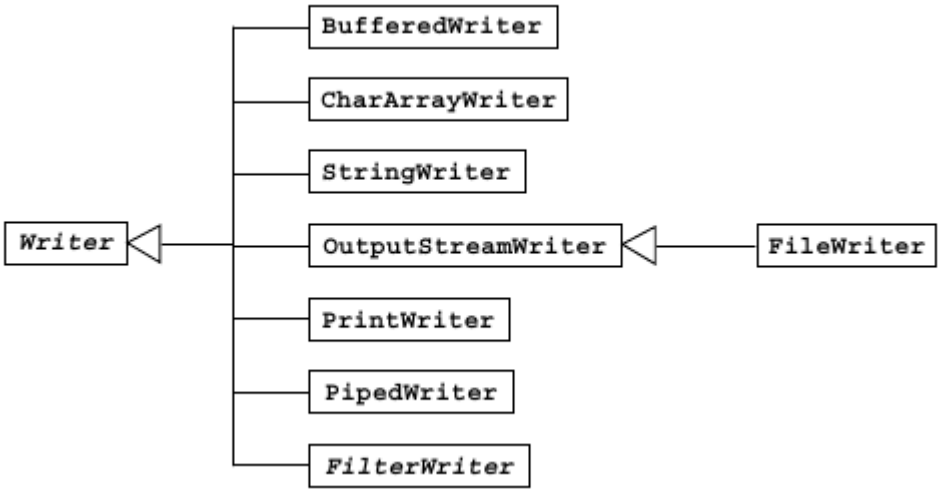
10.11: Basic Character Stream Classes

The hierarchy of Reader character stream classes in the java.io package.

The Reader Class Hierarchy



The Writer Class Hierarchy



Chapter 11: Console I/ O and File I/O

11.1: Objectives

Upon completion of this module, you should be able to:

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

11.2: Console I/O

Most applications must interact with the user. Such interaction is sometimes accomplished with text input and output to the console (using the keyboard as the standard input and using the terminal window as the standard output).

Java JDK supports console I/O with three public static variables on the java.lang.System class:

● The variable System.out is a PrintStream object that refers (initially) to the terminal window that launched the Java technology application.

● The variable System.in is an InputStream object that refers (initially) to the user's keyboard.

● The variable System.err is a PrintStream object that refers (initially) to the terminal window that launched the Java technology application.

11.2.1: Writing to Standard Output

You can write to standard output through the System.out.println(String) method. This PrintStream method prints the string argument to the console and adds a newline character at the end.

11.2.2: Reading From Standard Input

11.2.3: Simple Formatted Output

The Java programming language version 5 provided C language-style printf functionality. It provides standard formatted output from the program. This enables programmers to migrate from legacy code.

You can use the printf as normal C and C++ syntax.

System.out.printf("%s %5d %f%n",name,id,salary);

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.

11.2.4: Simple Formatted Input

The Scanner class provides formatted input functionality. It is part of the java.util package. It provides methods for getting primitive values and strings and blocks as it waits for input from the user.

11.3: Files and File I/O

- Creating File objects
- Manipulating File objects
- Reading and writing to file streams

11.3.1: Creating a New File Object

In Java technology, a directory is just another file. You can create a File object that represents a directory and then use it to identify other files, as shown in the third bullet .

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

11.3.2: The File Tests and Utilities

After you create a File object, you can use any methods in the following sections to gather information about the file.

File Names

The following methods return file names:

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

Directory Utilities

The following methods provide directory utilities:

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

General File Information and Utilities

The following methods return general file information:

- long lastModified()
- long length()
- boolean delete()

File Tests

The following methods return information about file attributes:

- boolean exists()
- boolean canWrite()
- boolean canRead()
- boolean isFile()
- boolean isDirectory()
- boolean isAbsolute()
- boolean isHidden()

11.4: File Stream I/O

The Java SE Development Kit supports file input in two forms:

- Use the FileReader class to read characters
- Use the BufferedReader class to use the readLine method

The Java SE Development Kit supports file output in two forms:

- Use the FileWriter class to write characters
- Use the PrintWriter class to use the print and println methods

Chapter 12: Building Java GUIs Using the Swing API

12.1: Objectives

Upon completion of this module, you should be able to:

- 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

12.2: What Are the Java Foundation Classes (JFC)?

JFC, or Java Foundation Classes, is a set of Graphical User Interface (GUI) support packages that are available as part of the Java SE platform and which became core application program interfaces (APIs) in JDK 1.2.

12.3: What Is Swing?

Swing is an enhanced component set that provides replacement components for those in the original AWT and a number of more advanced components. These components enable you to create user interfaces with the type of functionality that has become expected in modern applications. Such components include trees, tables, advanced text editors, and tear-off toolbars.

12.3.1: Pluggable Look-and-Feel

Pluggable look-and-feel enables developers to build applications that execute on any platform as if they were developed for that specific platform. A program executed in the Microsoft Windows environment appears as if it was developed for this environment; and the same program executed on the UNIX platform appears as if it was developed for the UNIX environment.

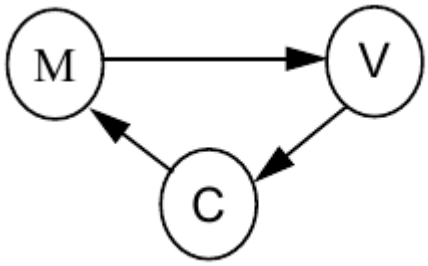
12.4: Swing Architecture

Swing components are designed based on the Model-View-Controller (MVC) architecture. The Swing architecture is not strictly based on the MVC architecture but has its roots in the MVC.

Model-View-Controller Architecture

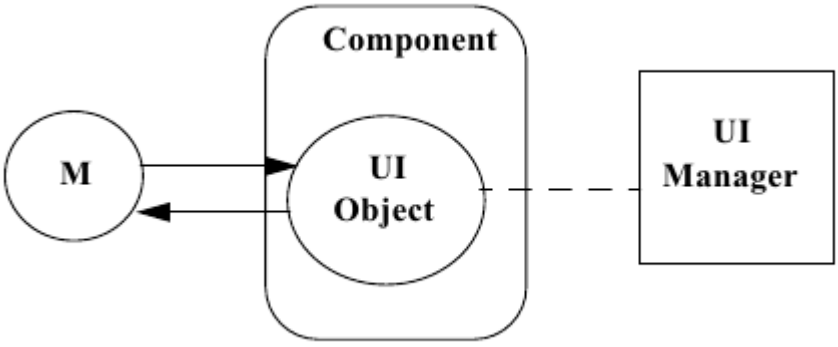
According to the MVC architecture, a component can be modeled as three separate parts.

- Model – The model stores the data used to define the component.
- View – The view represents the visual display of the component. This display is governed by the data in the model.
- Controller – The controller deals with the behavior of the components when a user interacts with it. This behavior can include any updates to the model or view.



Separable Model Architecture

The Swing components follow a separable model architecture. In this architecture the view and the controller are merged as a single composite object, because of their tight dependency on each other. The model object is treated as a separate object just like in MVC architecture.



12.5: Swing Packages

The Swing API has a rich and convenient set of packages that makes it powerful and flexible.

Package Name	Purpose
javax.swing	Provides a set of *light-weight* components such as, JButton, JFrame, JCheckBox, and much more
javax.swing.border	Provides classes and interfaces for drawing specialized borders such as, bevel, etched, line, matte, and more
javax.swing.event	Provides support for events fired by Swing components
javax.swing.undo	Allows developers to provide support for undo/redo in applications such as text editors
javax.swing.colorchooser	Contains classes and interfaces used by the JColorChooser component
javax.swing.filechooser	Contains classes and interfaces used by the JFileChooser component
javax.swing.table	Provides classes and interfaces for handling JTable
javax.swing.tree	Provides classes and interfaces for handling JTree
javax.swing.plaf	Provides one interface and many abstract classes that Swing uses to provide its pluggable look-and-feel capabilities
javax.swing.plaf.basic	Provides user interface objects built according to the Basic look-and-feel

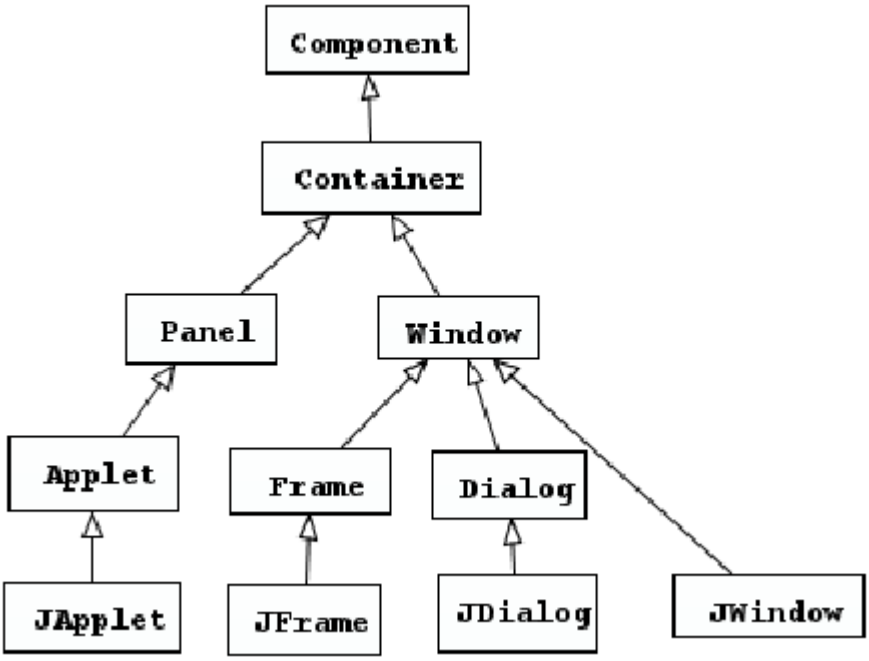
12.6: Swing Containers

Swing containers can be classified into three main categories

- Top-level containers
- General-purpose containers
- Special-purpose containers

12.6.1: Top-level Containers

Top-level containers are at the top of the Swing containment hierarchy. There are three top-level Swing containers: JFrame, JWindow, and JDialog. There is also a special class, JApplet, which, while not strictly a top-level container, is worth mentioning here because it should be used as the top-level of any applet that uses Swing components.

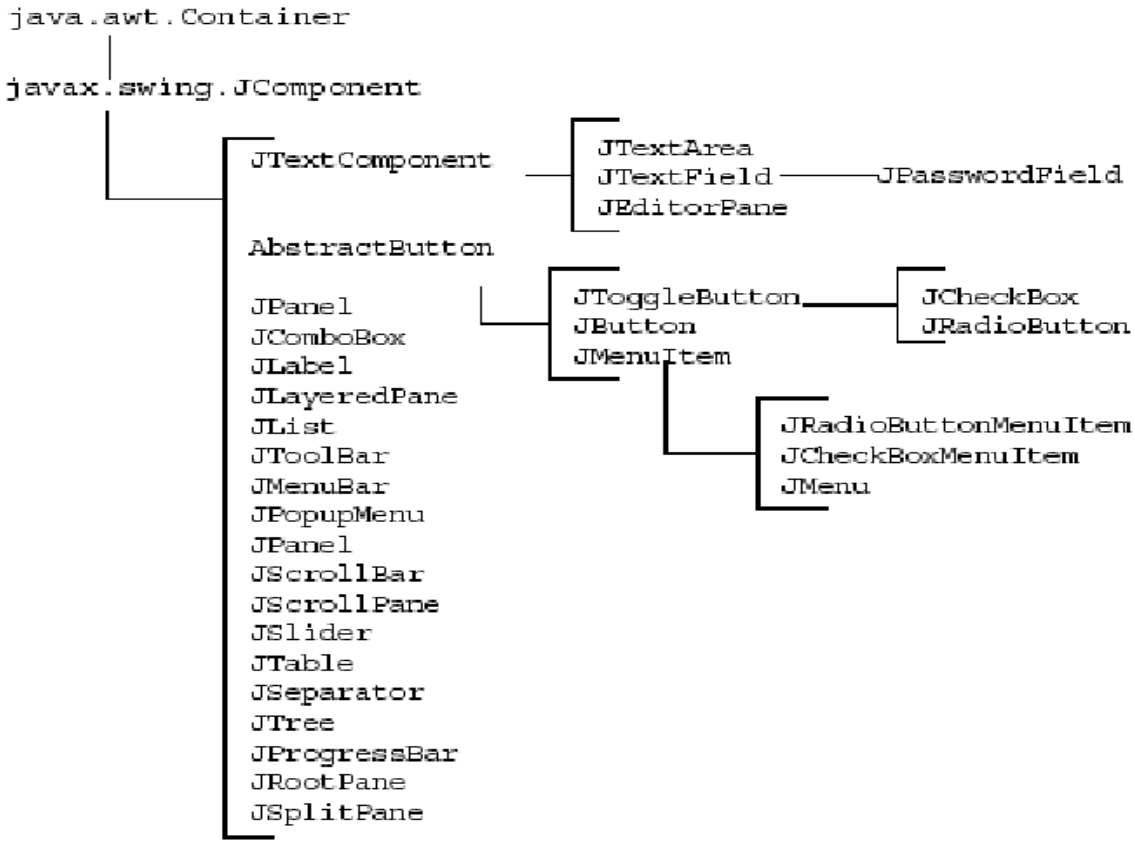


12.7: Swing Components

Swing GUIs use two kinds of classes: GUI classes and non-GUI support classes. The GUI classes are visual and descendants of JComponent, and are called J classes. The non-GUI classes provide services and perform vital functions for GUI classes; however, they do not produce any visual output

Swing components can be broadly classified as follows:

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



12.8: Properties of Swing Components

12.9: Layout Managers

A layout manager determines the size and position of the components within a container. The alternative to using layout managers is absolute positioning by pixel coordinates. Absolute positioning is achieved through setting a container's layout property to null. Absolute positioning is not platform-portable. Issues such as the sizes of fonts and screens ensure that a layout that is correct based on coordinates can potentially be unusable on other platforms.

Unlike absolute positioning, layout managers have mechanisms to cope with the following situations:

- The resizing of the GUI by the user
- Different fonts and font sizes used by different operating systems or by user customization
- The text layout requirements of different international locales (left-right, right-left, vertical)

12.10: GUI Construction

A Java technology GUI can be created using either of the following techniques:

● Programmatic construction

This technique uses code to create the GUI. This technique is useful for learning GUI construction. However, it is very laborious for use in production environments.

● Construction using a GUI builder tool

This technique uses a GUI builder tool to create the GUI. The GUI developer uses a visual approach to drag-and-drop containers and components to a work area. The tool permits the positioning and resizing of containers and components using a pointer device such as a computer mouse. With each step, the tool automatically generates the Java technology classes required to reproduce the GUI.

Chapter 13: Handling GUI-Generated Events

13.1: Objectives

Upon completion of this module, you should be able to:

- 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

13.2: What Is an Event?

When the user performs an action at the user interface level (clicks a mouse or presses a key), this causes an event to be issued. Events are objects that describe what has happened. A number of different types of event classes exist to describe different categories of user action.

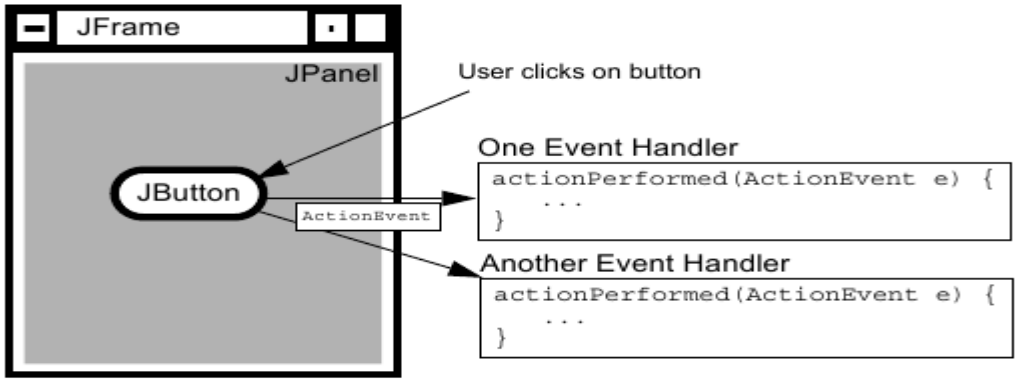
Event Handlers

An event handler is a method that receives an event object, deciphers it, and processes the user's interaction.

13.3: Java SE Event Model

13.3.1: Delegation Model

The delegation event model came into existence with JDK version 1.1. With this model, events are sent to the component from which the event originated, but it is up to each component to propagate the event to one or more registered classes, called listeners. Listeners contain event handlers that receive and process the event. In this way, the event handler can be in an object separate from the component. Listeners are classes that implement the EventListener interface.

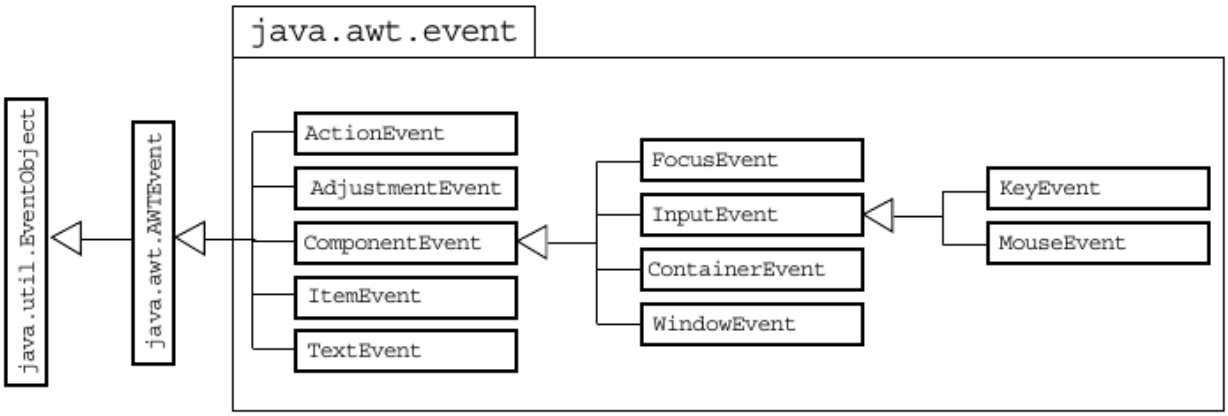


Events are objects that are reported only to registered listeners. Every event has a corresponding listener interface that mandates which methods must be defined in a class suited to receiving that type of event. The class that implements the interface defines those methods, and can be registered as a listener.

13.4: GUI Behavior

13.4.1: Event Categories

The general mechanism for receiving events from components has been described in the context of a single type of event. Many of the event classes reside in the java.awt.event package, but others exist elsewhere in the API.



13.4.2: A Listener Example

13.5: Concurrency in Swing

Applications that contain a GUI require several threads for handling the GUI efficiently. Threads responsible for executing the application code are called current threads.

● Threads responsible for handling the events generated by various components are called event dispatch threads.

● Threads responsible for executing lengthy tasks are called worker threads. Some examples of these lengthy tasks include waiting for some shared resource, waiting for user input, blocking for network or disk I/O, performing either CPU or memory-intensive calculations. These tasks can be executed in the background without affecting the performance of the GUI. You can use instances of the SwingWorker class to represent these worker threads. The SwingWorker class extends the Object class and implements the RunnableFuture interface.

The SwingWorker class provides the following utility methods:

● For communication and coordination between worker thread tasks and tasks on other threads, the SwingWorker class provides properties such as progress and state to support inter-thread communication.

● To execute simple background tasks, the doInBackground method can be used for running the tasks background.

● To execute tasks that have intermediate results, the results are published in a GUI using the publish and process methods.

● To cancel the background threads, they can be canceled using the cancel method.

Chapter 14: GUI-Based Applications

14.1: Objectives

Upon completion of this module, you should be able to:

- 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

14.2: How to Create a Menu

A JMenu is different from other components because you cannot add a JMenu component to ordinary containers and have them laid out by the layout manager. You can add menus only to a menu container. The following procedure 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.

14.2.1: Creating a JMenuBar

14.2.2: Creating a JMenu

14.2.3: Creating a JMenuItem

14.3: Controlling Visual Aspects

You can control the colors used for the foreground and the background of AWT components.

14.3.1: Colors

You use two methods to set the colors of a component:

setForeground()

setBackground()

Both of these methods take an argument that is an instance of the java.awt.Color class. You can use constant colors referred to as Color.red, Color.blue, and so on. The full range of predefined colors is listed in the documentation page for the Color class.

Chapter 15: Threads

15.1: Objectives

Upon completion of this module, you should be able to:

- Define a thread
- Create separate threads for,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

15.2: Threads

In this module, a thread, or execution context, is considered to be the encapsulation of a virtual CPU with its own program code and data. The class java.lang.Thread enables you to create and control threads.

A thread, or execution context, is composed of three main parts:

- A virtual CPU
- The code that the CPU executes
- The data on which the code works

15.3: Creating the Thread

A Thread constructor takes an argument that is an instance of Runnable. An instance of Runnable is made from a class that implements the Runnable interface (that is, it provides a public void run() method).

15.3.1: Starting the Thread

A newly created thread does not start running automatically. You must call its start method.

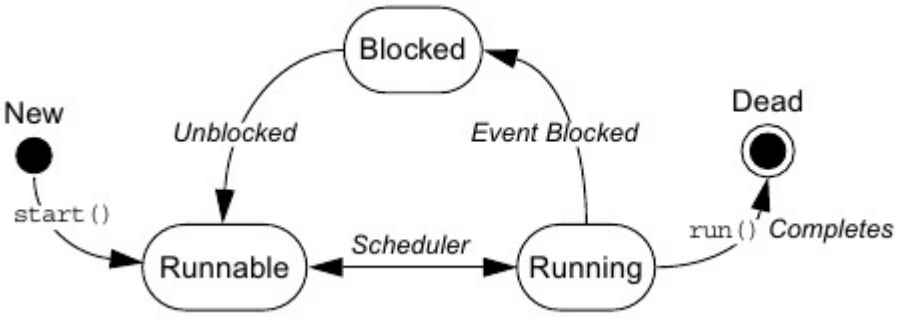
syntax:

threadrefname.start();

15.4: Thread Scheduling

The model of a pre-emptive scheduler is that many threads might be runnable, but only one thread is running. This thread continues to run until it ceases to be runnable or until another thread of higher priority becomes runnable. In the latter case, the lower priority thread is pre- empted by the thread of higher priority, which gets a chance to run instead.

A Thread object can exist in several different states throughout its lifetime



15.5: Basic Control of Threads

15.5.1: Testing Threads

A thread can be in an unknown state. Use the method isAlive to determine if a thread is still viable. The term Alive does not imply that the thread is running; it returns true for a thread that has been started but has not completed its task.

15.5.2: Accessing Thread Priority

Use the getPriority method to determine the current priority of the thread. Use the setPriority method to set the priority of the thread. The priority is an integer value. The Thread class includes the following constants:

Thread.MIN_PRIORITY

Thread.NORM_PRIORITY

Thread.MAX_PRIORITY

15.5.3: Putting Threads on Hold

Mechanisms exist that can temporarily block the execution of a thread. You can resume execution as if nothing happened. The thread appears to have executed an instruction very slowly.

The Thread.sleep() Method

The join() Method

The Thread.yield() Method

15.6: Using the synchronized Keyword

It provides the Java programming language with a mechanism that enables a programmer to control threads that are sharing data.

Problem:

15.6.1: The Object Lock Flag

In Java technology, every object has a flag associated with it. You can think of this flag as a lock flag. The keyword synchronized enables interaction with this flag, and provides exclusive access to code that affects shared data. The following is the modified code fragment:

15.6.2: Releasing the Lock Flag

The lock flag is given back to its object automatically. When the thread that holds the lock passes the end of the synchronized code block for which the lock was obtained, the lock is released. Java technology ensures that the lock is always returned automatically, even if an encountered exception, a break statement, or a return statement transfers code execution out of a synchronized block.

15.6.3: Using synchronized – Putting It Together

The synchronized mechanism works only if all access to delicate data occurs within the synchronized blocks.

synchronized blocks

synchronized method

15.7: Thread Interaction – wait and notify

The java.lang.Object class provides two methods, wait and notify, for thread communication. If a thread issues a wait call on a rendezvous object x, that thread pauses its execution until another thread issues a notify call on the same rendezvous object x. For a thread to call either wait or notify on an object, the thread must have the lock for that particular object. In other words, wait and notify are called only from within a synchronized block on the instance on which they are being called.

15.7.1: The Pool Story

When a thread executes synchronized code that contains a wait call on a particular object, that thread is placed in the wait pool for that object. Additionally, the thread that calls wait releases that object's lock flag automatically. You can invoke different wait methods.

wait()

wait(long timeout)

When a notify call is executed on a particular object, an arbitrary thread is moved from that object's wait pool to a lock pool, where threads stay until the object's lock flag becomes available. The notifyAll method moves all threads waiting on that object out of the wait pool and into the lock pool. Only from the lock pool can a thread obtain that object's lock flag, which enables the thread to continue running where it left off when it called wait.

15.8: Putting It Together

An example of thread interaction that demonstrates the use of wait and notify methods to solve a classic producer-consumer problem.

Chapter 16: Networking

16.1: Objectives

At the end of this module, you should be able to:

- 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

16.2: Networking

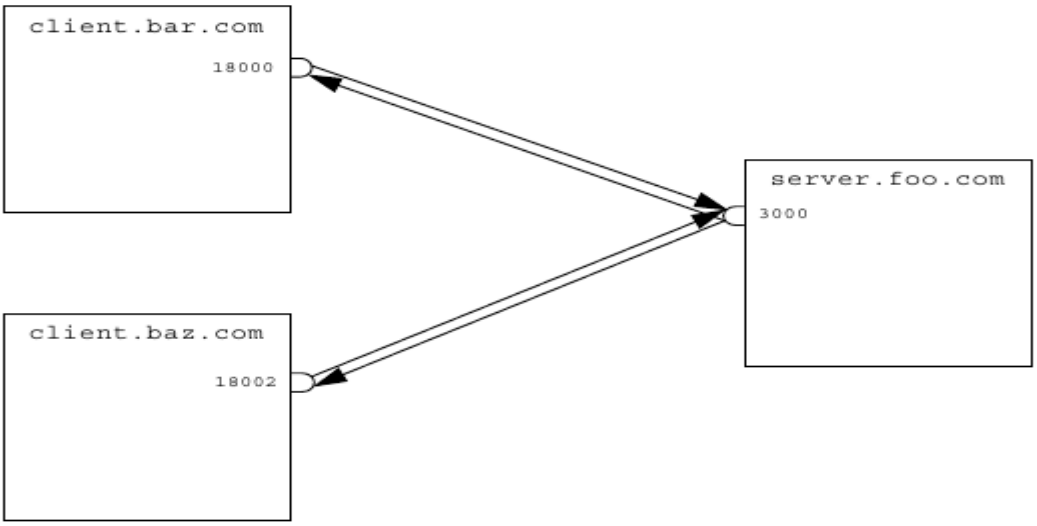
16.2.1: Sockets

Socket is the name given, in one particular programming model, to the endpoints of a communication link between processes.

16.2.2: Setting Up the Connection

To set up the connection, one machine must run a program that is waiting for a connection, and a second machine must try to reach the first. This is similar to a telephone system, in which one party must make the call, while the other party is waiting by the telephone when that call is made.

A description of the TCP/IP network connections is presented in this module.



16.3: Networking With Java Technology

16.4: Addressing the Connection

A network connection requires a port number, which you can think of as a telephone extension number. After you connect to the proper computer, you must identify a particular purpose for the connection.

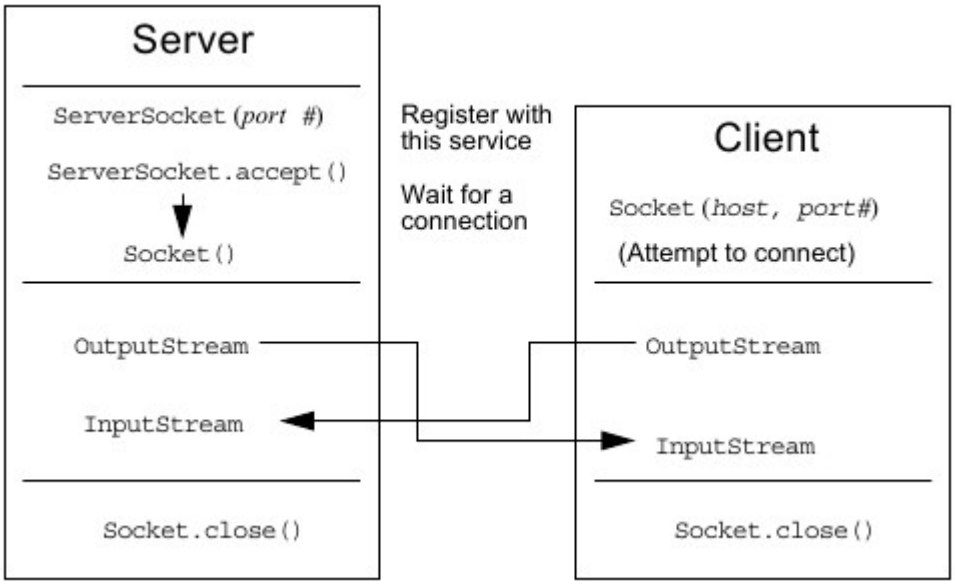
16.4.1: Port Numbers

Port numbers in TCP/IP systems are 16-bit numbers and the values range from 0–65535. In practice, port numbers below 1024 are reserved for predefined services, and you should not use them unless communicating with one of those services (such as telnet, Simple Mail Transport Protocol [SMTP] mail, ftp, and so on). Client port numbers are allocated by the host OS to something not in use, while server port numbers are specified by the programmer, and are used to identify a particular service.

Both client and server must agree in advance on which port to use. If the port numbers used by the two parts of the system do not agree, communication does not occur.

16.5: Java Networking Model

In the Java programming language, TCP/IP socket connections are implemented with classes in the java.net package.



● The server assigns a port number. When the client requests a connection, the server opens the socket connection with the accept() method.

● The client establishes a connection with host on port port#.

● Both the client and server communicate by using an InputStream and an OutputStream.

16.6: Minimal TCP/IP Server

TCP/IP server applications rely on the ServerSocket and Socket networking classes provided by the Java programming language. The ServerSocket class takes most of the work out of establishing a server connection.

16.7: Minimal TCP/IP Client

The client side of a TCP/IP application relies on the Socket class. Again, much of the work involved in establishing connections is done by the Socket class.

16.8: A Network Application using UDP

Course Booklet for Developing Applications With the **Java SE 6 Platform**

Know how of Developing Applications With the **Java SE 6 Platform**
By Emertxe

version1.0 (Jan 16, 2013)

Chapter 17: Implementing the JavaTM Database Connectivity (JDBC) API

17.1: Objectives

Upon completion of this module, you should be able to:

● Describe the JDBC API

● Explain how using the abstraction layer provided by the JDBC API can make a database front end portable across platforms

● Describe the five major tasks involved with the JDBC programmer’s interface

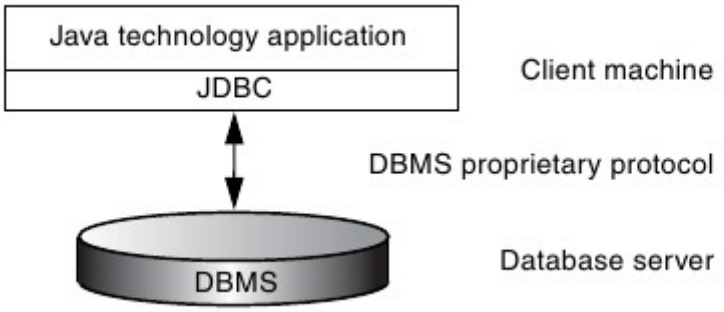
● State the requirements of a JDBC driver and its relationship to the JDBC driver manager

● Describe the data access objects (DAO) pattern and its applicability to a given scenario

17.2: Introducing the JDBC Interface

The JDBC API enables you to write database applications in the Java programming language without having to concern yourself with the underlying details of a particular database. The JDBC API requires a driver that passes the appropriate JDBC API and calls the database engine. The JDBC driver insulates you from the specifics of communicating with the database.

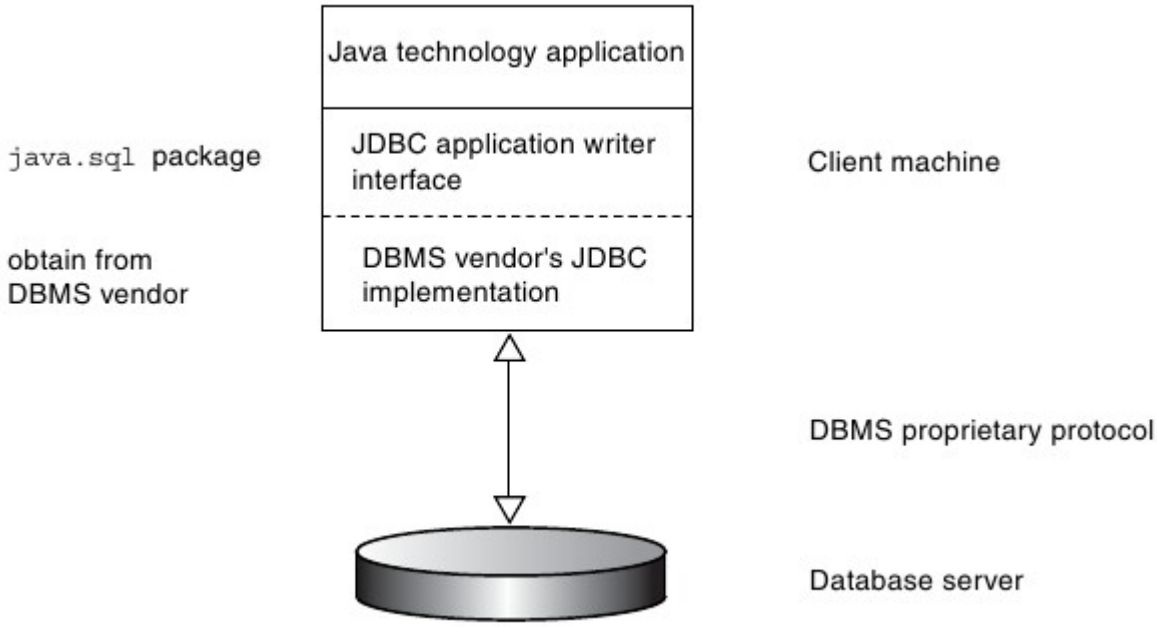
17.2.1: The Two



Components of the JDBC API

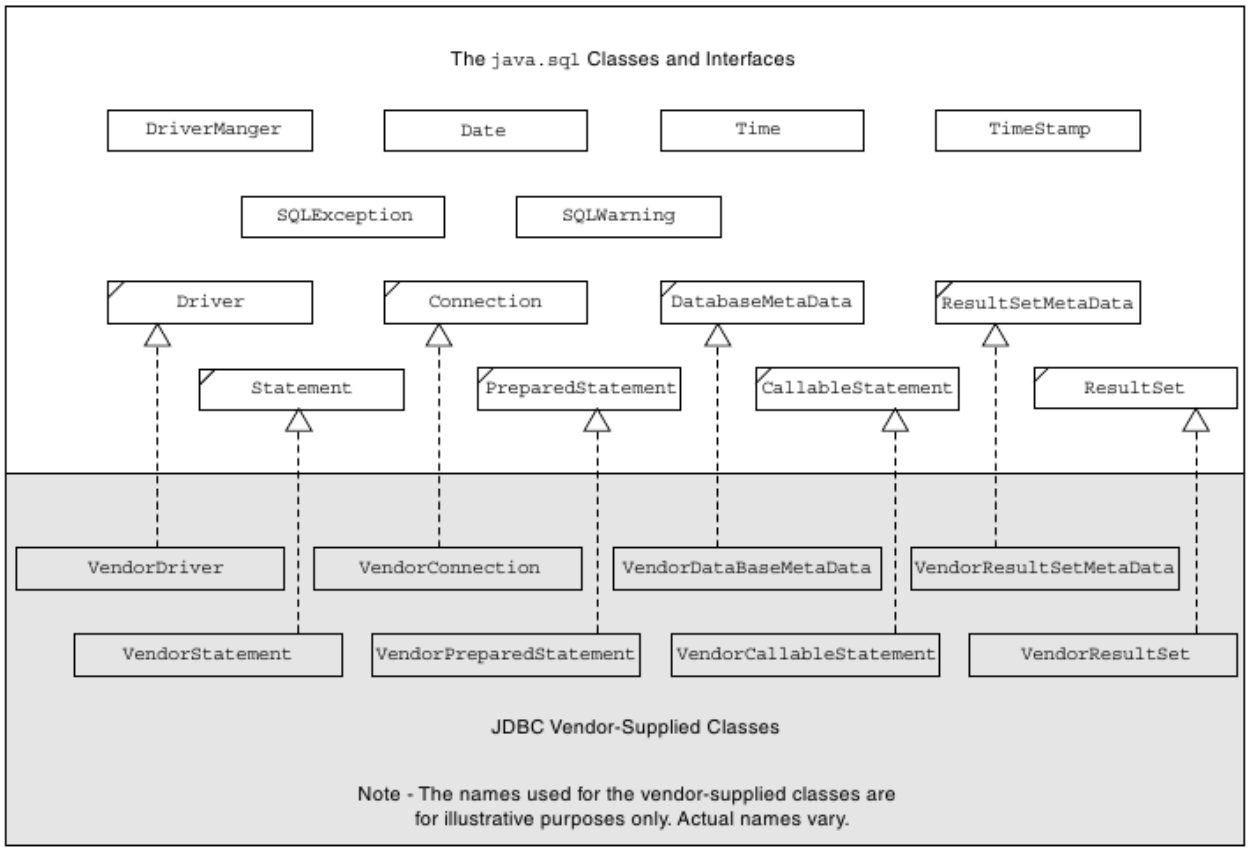
There are two major components of the JDBC API:

- An implementation interface for database manufacturers
- An interface for application and applet writers



The JDBC API components are contained in the java.sql package, which is a core API of the Java Development Kit (JDK) distribution.

Components of the JDBC API in Detail


The java.sql Classes and Interfaces
DriverManger
Date
Time
TimeStamp
SQLException
SQLWarning
Driver
Connection
DatabaseMetaData
ResultSetMetaData
Statement
PreparedStatement
CallableStatement
ResultSet
VendorDriver
VendorConnection
VendorDataBaseMetaData
VendorResultSetMetaData
VendorStatement
VendorPreparedStatement
VendorCallableStatement
VendorResultSet
JDBC Vendor-Supplied Classes
Note - The names used for the vendor-supplied classes are
for illustrative purposes only. Actual names vary.

17.2.2: The JDBC API

The classes and interfaces of the JDBC API can be grouped into six functional categories .

Category	Class or Interface	Supplied by	Comment
Connecting to the data resource	• `DriverManager` class • `Driver` interface • `Connection` interface	• `java.sql` • Vendor • Vendor	• Manages a set of JDBC drivers • Supplies DBMS connection • Manages a database session
Sending SQL statements	• `Statement` interface • `PreparedStatement` interface • `CallableStatement` interface	Vendor	• Static SQL statement wrapper object • Precompiled SQL statement wrapper object • SQL stored procedure wrapper object
Retrieving results of a query	`ResultSet` interface	Vendor	Database `Resultset` wrapper object
Mapping between SQL types and Java technology types	• `Date` class • `Time` class • `TimeStamp` class	`java.sql`	• SQL date value • SQL time value • SQL timestamp value
Providing database metadata	• `DatabaseMetaData` interface • `ResultSetMetaData` interface	Vendor	• Database information wrapper object • `Resultset` column information wrapper object
Throwing exceptions	• `SQLException` class • `SQLWarning` class	`java.sql`	• Database access errors • Database access warnings

17.2.3: Using Java DB: A Real-World JDBC Driver

Java DB is an multi-user database written entirely in Java technology. It is an open source project being developed as part of the Apache DB project. (See db.apache.org.) .

The Java DB driver file comes packaged in a.jar file called derbyclient.jar. To make this driver file available to your Java Runtime Environment (JRE), copy it to a location that is included in your class path.

17.3: Connecting Through the JDBC Interface

The common tasks you perform while connecting through the JDBC interface.

1. Register the vendor-supplied JDBC driver with the DriverManager class.

Most JDBC drivers are designed to register automatically with DriverManager when the JDBC driver class is loaded by the classloader.

2. Establish a DBMS session.

To establish a DBMS session, you create an instance of the vendor-supplied class that implements the java.sql.Connection interface. To create the Connection object, use the static getConnection method of the DriverManager class.

3. Create an SQL query wrapper object.

To create a query wrapper object, you create an instance of the vendor-supplied class that implements the java.sql.Statement interface. To do this, use the createStatement method of the Connection interface.

4. Submit a query and receive the result.

To submit a query, first create a string containing the SQL statement. Next, use this query string as the argument to the executeQuery method of the Statement interface. Finally, receive the query result wrapper object in a variable of type ResultSet.

5. Extract the data from the result wrapper object.

Use the methods of the ResultSet interface to extract the data returned by the query.

6. Close all the acquired objects namely the result set, the statement, and the connection.

17.4: Connecting Through the JDBC Interface Detailed Review

17.4.1: Registering a JDBC Driver

When a driver is loaded, it is the responsibility of the driver implementation to register itself with the driver manager.

- It creates an instance of itself in a static code block.
- When the constructor is called either explicitly or implicitly, it registers itself with the driver manager.

17.4.2: Establishing a DBMS Session

The first step is to specify the database to which you want to connect. To do this in JDBC API, specify a Universal Resource Locator (URL) string that indicates the database type. The proposed URL syntax for a JDBC database is:

jdbc:subprotocol:subname

The DriverManager.getConnection method takes a URL string, a user string, and a password string as arguments. The JDBC driver management layer attempts to locate a driver that can connect to the database represented by the URL. If a driver succeeds in establishing a connection, it returns an appropriate java.sql.Connection object.

17.4.3: Creating an SQL Query Wrapper Object

To create an SQL query wrapper object,get a Statement object from the Connection.createStatement method.

Statement stmt = con.createStatement ();

17.4.4: Submitting a Query and Receiving the Results

Use the Statement.executeUpdate method to submit an INSERT, UPDATE, or DELETE statement to the database. The JDBC API passes the SQL statement to the underlying database connection unaltered. It does not attempt to interpret queries.

17.4.5: Extracting the Data From the Result Wrapper Object

The result of executing a query statement is a set of rows that are accessible using a java.sql.ResultSet object. The rows are received in order.

17.5: Mapping Between SQL Data Types and Java Data Types

The mapping between SQL types and Java technology types.

SQL Type	Java Technology Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

17.6: Production Code Strategies for JDBC API Usage

The main issue is the hard-coding of the following information:

- The JDBC driver class name
- URL used for establishing a connection with the JDBC API
- User name
- Password

By hard coding this information, your code lacks flexibility and portability.

To resolve this problem, the first step is to identify and parameterize all deployment-level configuration information. This translates to using variables for each hard-coded piece of data and initializing the variables at application start-up.

The following sections describe strategies you can use to initialize variables on start up.

They are:

- Loading properties from a file
- Loading properties from the command line
- Loading JDBC drivers through jdbc.drivers
- Loading configuration data from a GUI

17.6.1: Loading Properties From a File

You can choose to create a property file with each line containing a property-value pair and use the load method of the java.util.Properties class to load all the properties.

For example, you could create a file named krtkjdbc.txt containing the following property-value pairs.

jdbcDriver = org.apache.derby.jdbc.ClientDriver

jdbcUrl = jdbc:derby://localhost:1527/StockMarket

jdbcUser = public

jdbcPassword = public

The steps to be taken to load properties from a file in the code .

1) The first step is to create an empty java.util.Property object.

```
// create a properties object
Properties p = new Properties();
```

2) The next step is to load the properties stored as text in the file krtkjdbc.txt.

```
p.load(new FileInputStream(“brokerjdbc.txt”));
```

3) The final step is to initialize the corresponding variables to the property values.

```
String driver = p.getProperty(“jdbcDriver”;
Class.forName(driver);
System.out.println("LOADED DRIVER ---> " + driver);
.....
```

17.7: Introducing the DAO Pattern

The role of a DAO is to segregate completely the persistence logic from business or presentation logic. This ensures that the user of DAO is unaware of the database, its physical representation, and the relationships between the objects of the database

The DAO design pattern is implemented using the following steps:

1. Define a DAO interface.

The DAO interface contains methods for interacting (reading and writing) with a data source. These methods signatures must be generic. This would enable these methods not to contain any aspects that would bind them to a specific database.

2. Write an implementation of the DAO interface that is specific to the data source used by the application.

3. Code the business logic of the application to access the data source using only the methods of the DAO interface.