Java Fundamentals

Author: Nam Vu
Principle Software Engineer

Dec 2014

Course Objectives

- At the end of the course, you will have acquired sufficient knowledge to:
- Use Java technology data types and expressions
- Use Java technology flow control constructs
- Use arrays and other data collections
- Implement error-handling techniques using exception handling

Agenda

Audience and Prerequisite

- The course is for any one who wants to learn Java
- The following are beneficial if you already have knowledge and experiences as:
  - Created and compiled programs with C/C++/Java

Assessment Disciplines

- Class Participation: at least 80% of course time
- Assignment: get al least 70/100 score for final exercise

Java Basics

Overview

- Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).
- Java is:
  - Object Oriented
  - Platform independent
  - Simple
  - Multithreaded
  - Distributed

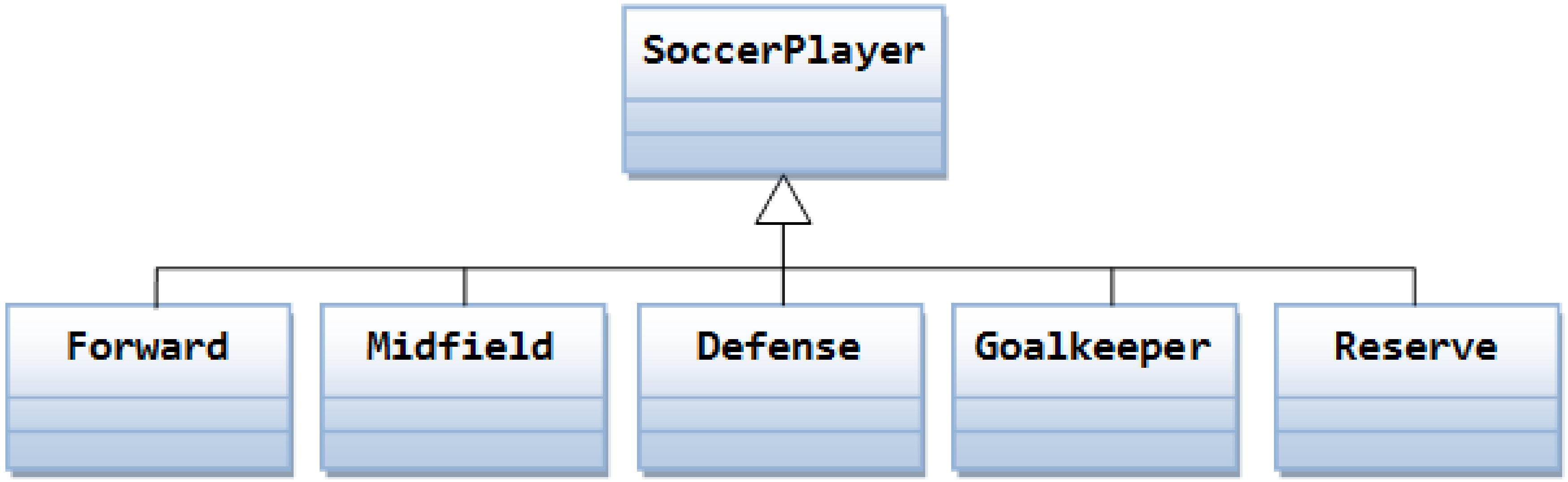
Java Object Oriented

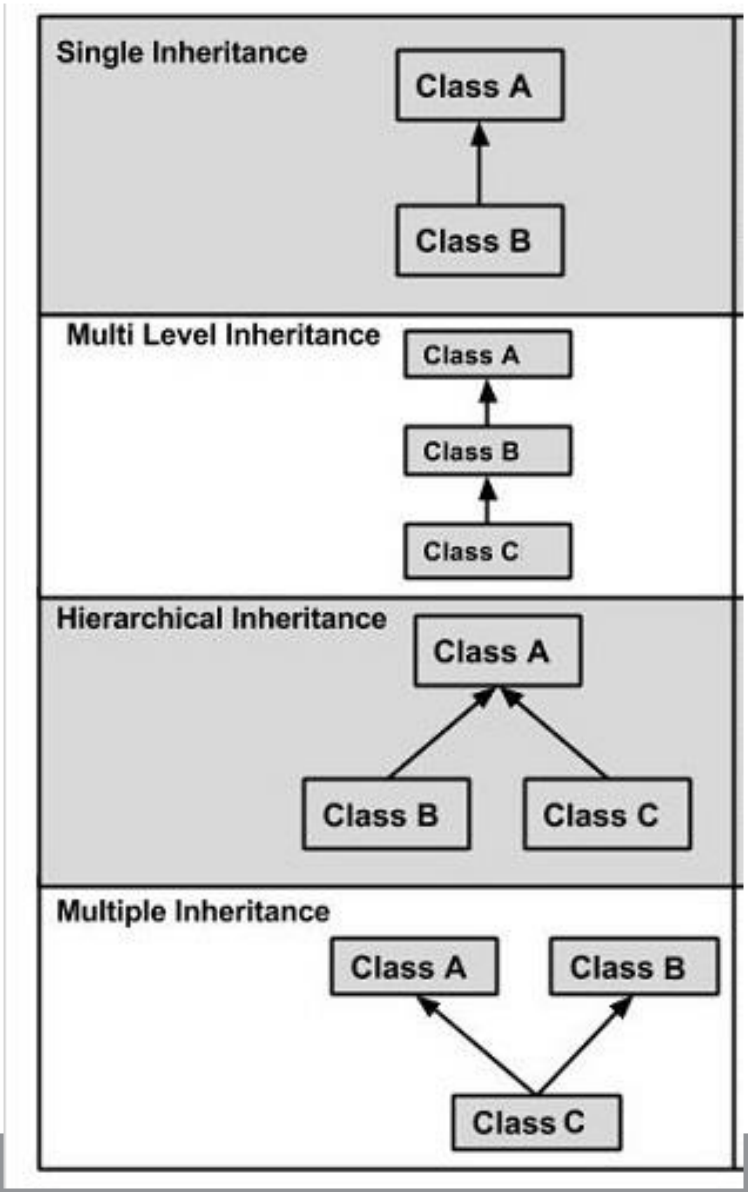
- Java - Inheritance
- Java - Polymorphism
- Java - Abstraction
- Java - Encapsulation
- Java - Interfaces
- Java – Packages
- Java – Overriding
- Java - Overloading

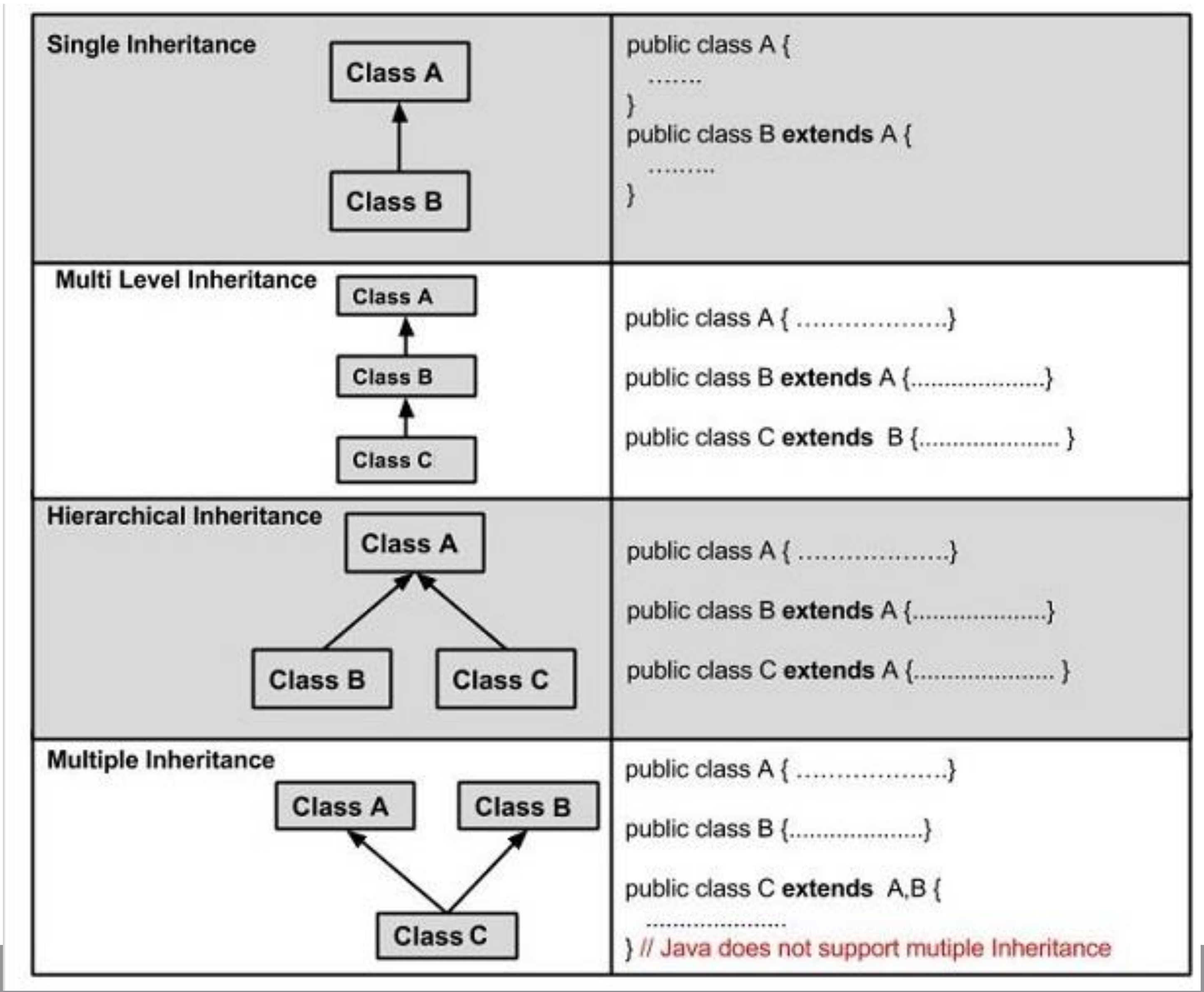
Java - Inheritance

- IS-A Relationship

HAS-A relationship




Single Inheritance
Class A
Class B
Multi Level Inheritance
Class A
Class B
Class C
Hierarchical Inheritance
Class A
Class B
Class C
Multiple Inheritance
Class A
Class B
Class C



Single Inheritance	public class A {
	  ....
	}
	public class B **extends** A {
	  ....
	}

Multi Level Inheritance	public class A { ....}
	public class B **extends** A {....................}
	public class C **extends** B {...................... }

Hierarchical Inheritance	public class A { ....}
	public class B **extends** A {....................}
	public class C **extends** A {...................... }

Multiple Inheritance	public class A { ....}
	public class B {....................}
	public class C **extends** A,B {
	  ....
	} // Java does not support mutiple Inheritance

Java - Polymorphism

- Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

- **Java – Abstraction:** A class which contains the **abstract** keyword in its declaration is known as abstract class.
- **Java – Encapsulation:** Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as as single unit. In encapsulation the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class, therefore it is also known as data hiding.
- **Java – Interfaces:** An interface is a reference type in Java, it is similar to class, it is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

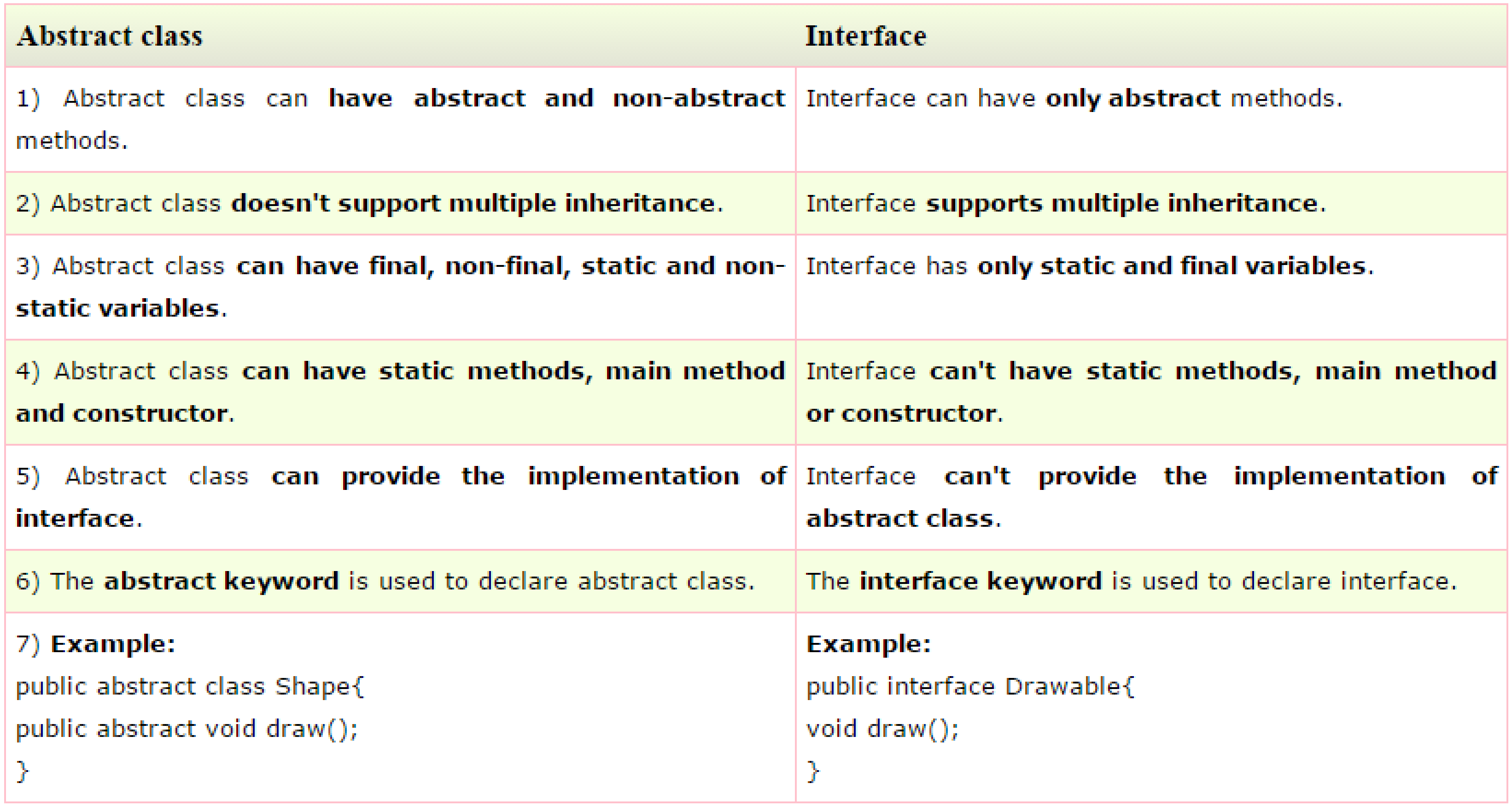
```
public interface NameOfInterface
```

- **Java – Packages:** Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A Package can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and name space management.

- **Java – Overriding:** In the previous chapter, we talked about super classes and sub classes. If a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final.
- **Java – Overloading:** Method overloading means that the *same method name* can have *different implementations* (versions). However, the different implementations must be distinguishable by their parameter list (either the number of parameters, or the type of parameters, or their order).

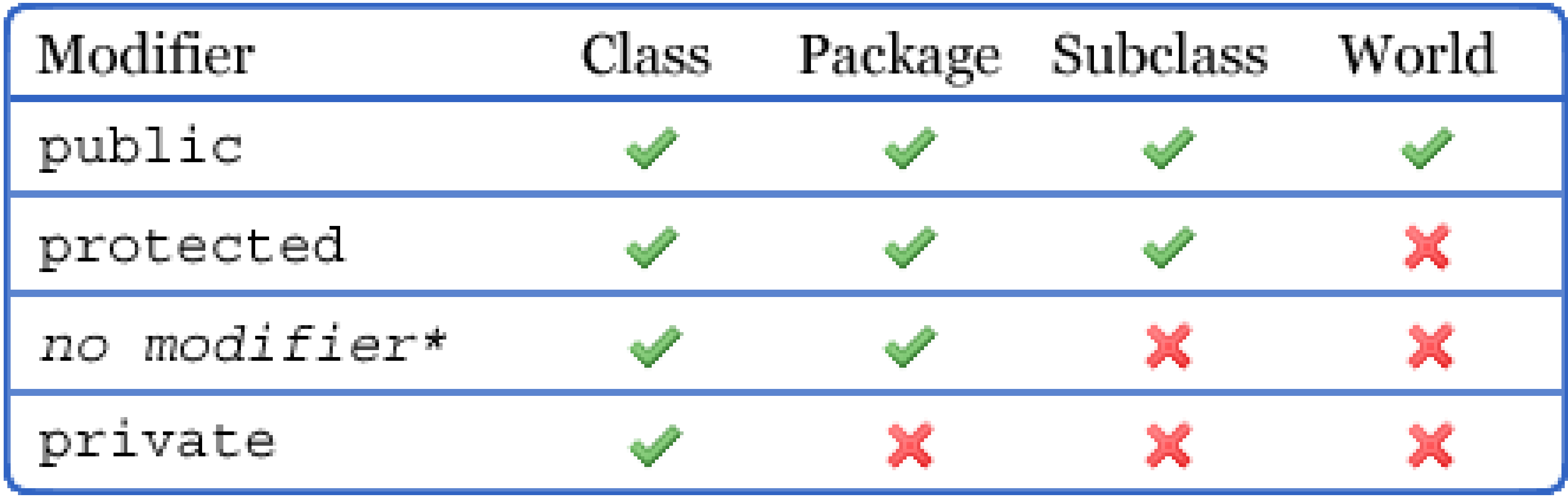
Abstract vs Interface class



Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Java Access Modifier



Modifier	Class	Package	Subclass	World
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
`no modifier*`	✓	✓	✗	✗
`private`	✓	✗	✗	✗

Java Basic

- Java - Basic Syntax
- Java - Object & Classes
- Java - Basic Datatypes
- Java - Variable Types
- Java - Modifier Types
- Java - Basic Operators
- Java - Loop Control
- Java - Decision Making
- Java - Numbers
- Java - Characters
- Java - Strings
- Java - Arrays
- Java - Date & Time
- Java - Regular Expressions
- Java - Methods
- Java - Files and I/O
- Java – Exceptions

Basic Syntax

- Classes
- Methods
- Constructors
- Instance variables
- Local variables
- Class variables
- Objects

Basic Syntax

- Identifiers:
  - All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
  - After the first character identifiers can have any combination of characters.
  - A key word cannot be used as an identifier.
  - Most importantly identifiers are case sensitive.
  - Examples of legal identifiers: age, $salary, _value, __1_value
  - Examples of illegal identifiers: 123abc, -salary
- Modifiers
  - Access Modifiers: default, public, protected, private
  - Non-access Modifiers: static, final, abstract, synchronized

Basic Syntax

```
class Example {
       private int var; public Example2() {
              //code for default one var = 10;
      }
      public Example(int num) {
        //code for parameterized one var = num;
      }
      public int getValue() {
        return var;
      }
      public static void main(String args[]) {
             Example2 obj2 = new Example2();
             System.out.println("var is: "+obj2.getValue());
      }
}
```

Basic Operators

- Arithmetic Operators: +, -, *, /, %, ++, --
- Relational Operators: ==, !=, >, <, >=, <=
- Bitwise Operators: &, |, ^…
- Logical Operators: &&, ||, !
- Assignment Operators: =, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=
- Misc Operators:
  - Conditional operator (?:)

  variable x = (expression) ? value if true : value if false
  - Instance of

  (Object reference variable) instanceof (class/interface type)

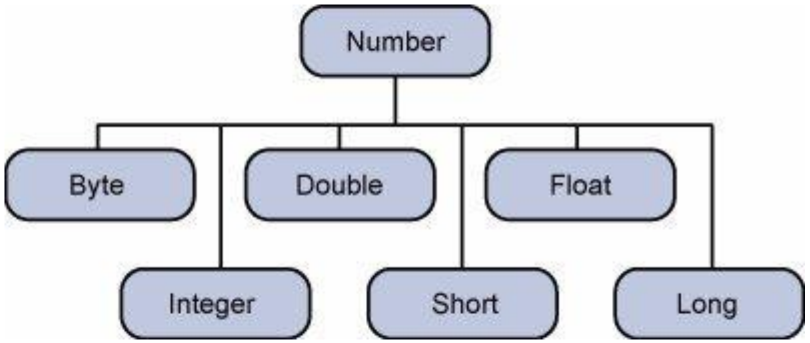
Flow Control

- Loops:
  - while Loop
  - do...while Loop
  - for Loop
- Decision making:
  - if … else if ….else
  - switch
- continue, break:

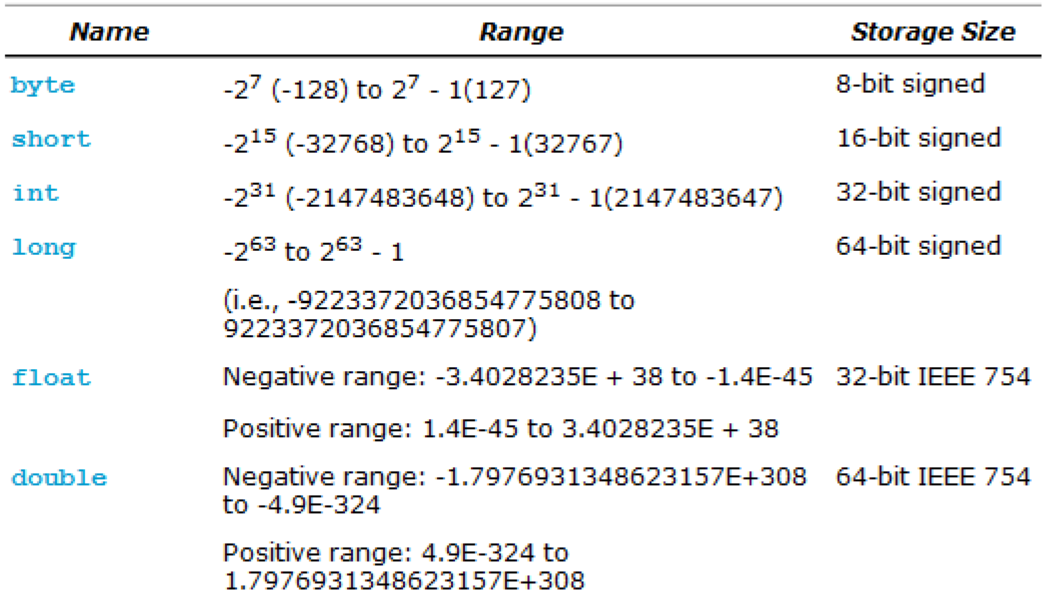
Primitive Types and Wrapper Classes

- Logical – `boolean`
- Textual – `char`
- Integral – `byte, short, int,` and `long`
- Floating – `double` and `float`

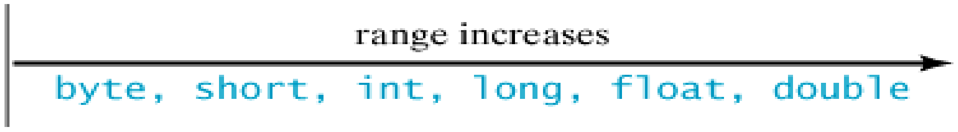
- Wrapping Classes:



Primitive Types and Wrapper Classes

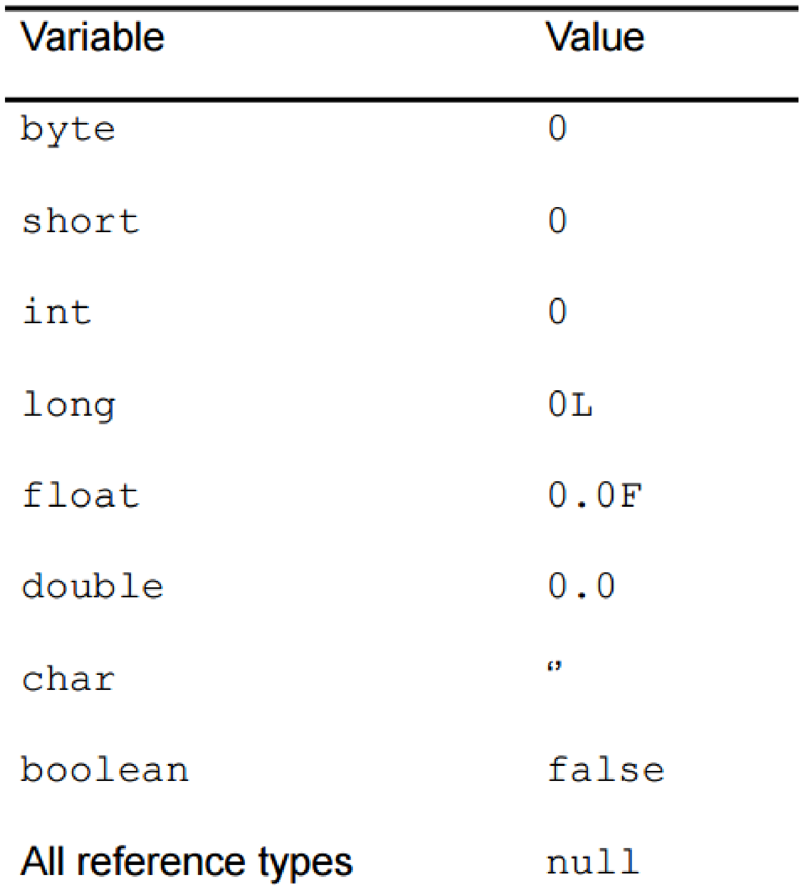


Name	Range	Storage Size
byte	-2^7 (-128) to $2^7 - 1$(127)	8-bit signed
short	-2^{15} (-32768) to $2^{15} - 1$(32767)	16-bit signed
int	-2^{31} (-2147483648) to $2^{31} - 1$(2147483647)	32-bit signed
long	-2^{63} to $2^{63} - 1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
float	Negative range: -3.4028235E + 38 to -1.4E-45 Positive range: 1.4E-45 to 3.4028235E + 38	32-bit IEEE 754
double	Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308	64-bit IEEE 754



Range increases

Primitive Types and Wrapper Classes

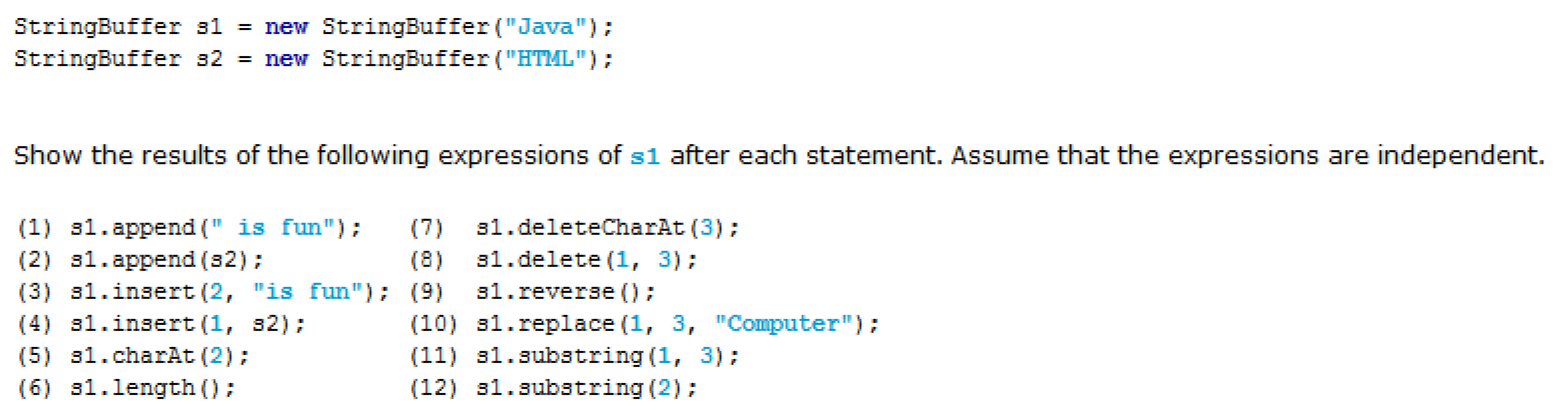


Variable	Value
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0L`
`float`	`0.0F`
`double`	`0.0`
`char`	`‘’`
`boolean`	`false`
All reference types	`null`

Default Value of Variables

String

- String: String greeting = "Hello world!";
- StringBuilder
- StringBuffer



```
StringBuffer s1 = new StringBuffer("Java");
StringBuffer s2 = new StringBuffer("HTML");
```

Show the results of the following expressions of s1 after each statement. Assume that the expressions are independent.

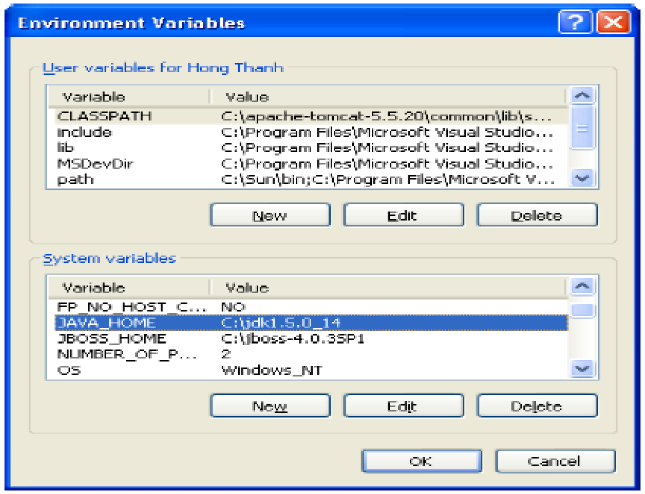
```
(1) s1.append(" is fun");    (7)  s1.deleteCharAt(3);
(2) s1.append(s2);           (8)  s1.delete(1, 3);
(3) s1.insert(2, "is fun");  (9)  s1.reverse();
(4) s1.insert(1, s2);        (10) s1.replace(1, 3, "Computer");
(5) s1.charAt(2);            (11) s1.substring(1, 3);
(6) s1.length();             (12) s1.substring(2);
```

Example for StringBuffer

Setup environment

- Set Up Java Development Kit
- Set Up IDE
- Set Up CLASSPATH
- HelloWorld Application Example

- Download JDK from: http://www.oracle.com
- Run downloaded file
- Set up JAVA_HOME

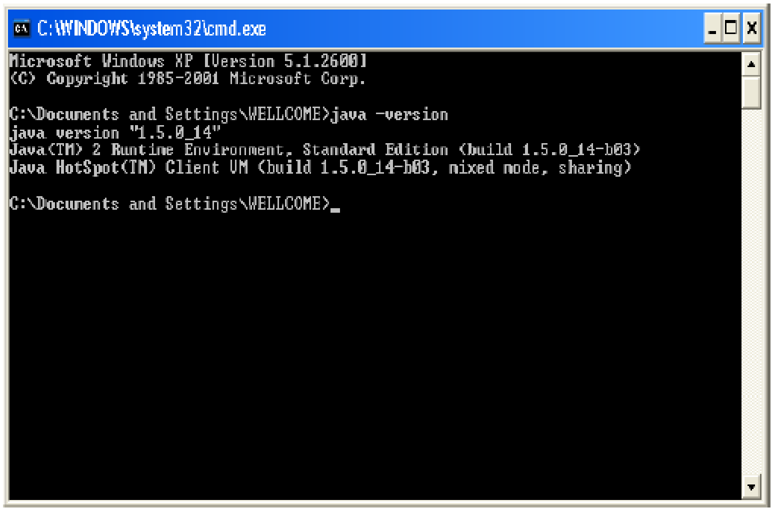


User variables for Hong Thanh

Variable	Value
CLASSPATH	C:\apache-tomcat-5.5.20\common\lib\s...
include	C:\Program Files\Microsoft Visual Studio...
lib	C:\Program Files\Microsoft Visual Studio...
MSDevDir	C:\Program Files\Microsoft Visual Studio...
path	C:\Sun\bin;C:\Program Files\Microsoft V...

System variables

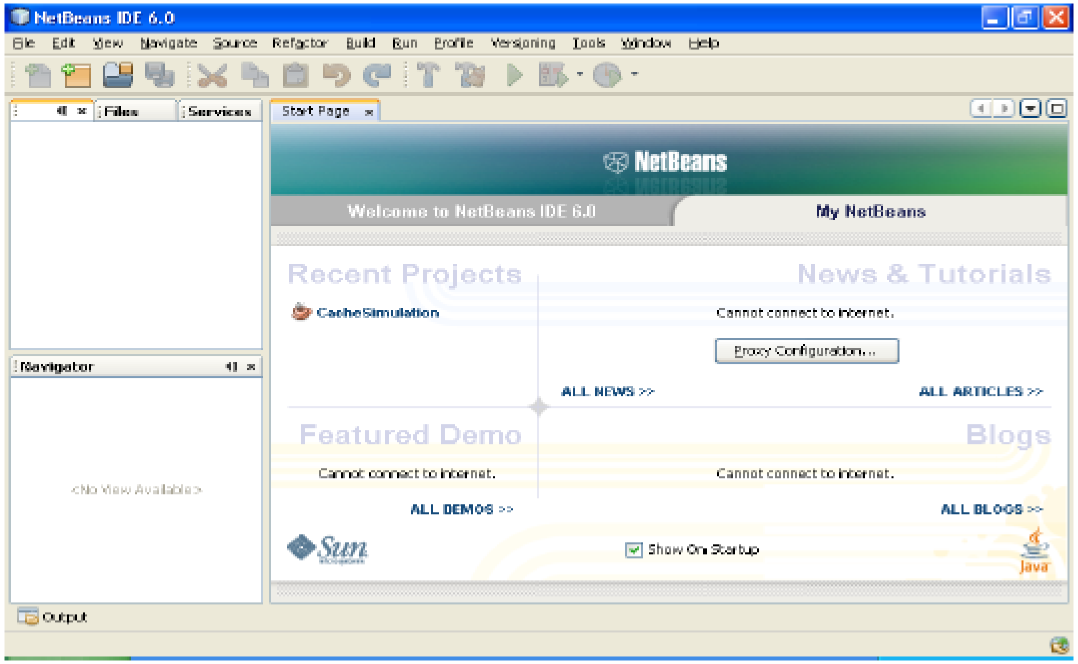
Variable	Value
FP_NO_HOST_C...	NO
JAVA_HOME	C:\jdk1.5.0_14
JBOSS_HOME	C:\jboss-4.0.3SP1
NUMBER_OF_P...	2
OS	Windows_NT



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\WELLCOME>java -version
java version "1.5.0_14"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_14-b03)
Java HotSpot(TM) Client VM (build 1.5.0_14-b03, mixed mode, sharing)

C:\Documents and Settings\WELLCOME>_
```

Setup environment

- **Eclipse**
  - Download from: www.eclipse.org
- **Netbeans** – version 7.3
  - Download from: **www.netbeans.org**



Example

```java
// HelloWorld.java
// Printing multiple lines in a dialog box

// Java extension packages
import javax.swing.JOptionPane;  // import class JOptionPane

public class HelloWorld {

   // main method begins execution of Java application
    public static void main( String args[] )
    {
       JOptionPane.showMessageDialog(
          null, "Welcome\nto\nJava\nProgramming!" );

       System.exit( 0 );  // terminate application

    }  // end method main

 }  // end class HelloWorld
```

Exercise

Demo 1: Greeting

- Points to remember:
  - Java coding convention
  - Review: class, constructors, methods, variables, constants …

Java Basic

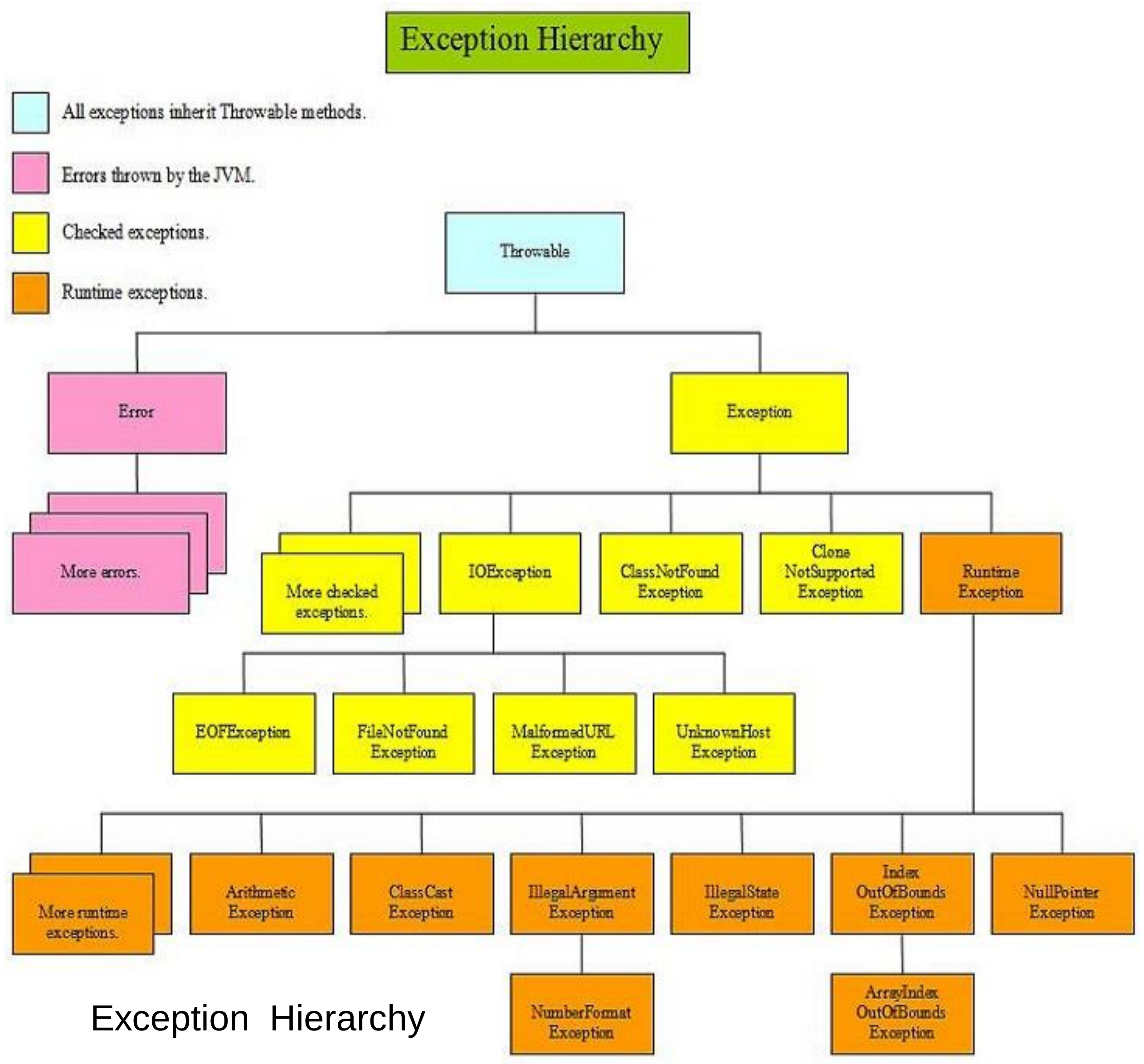
Exceptions handling

Exception

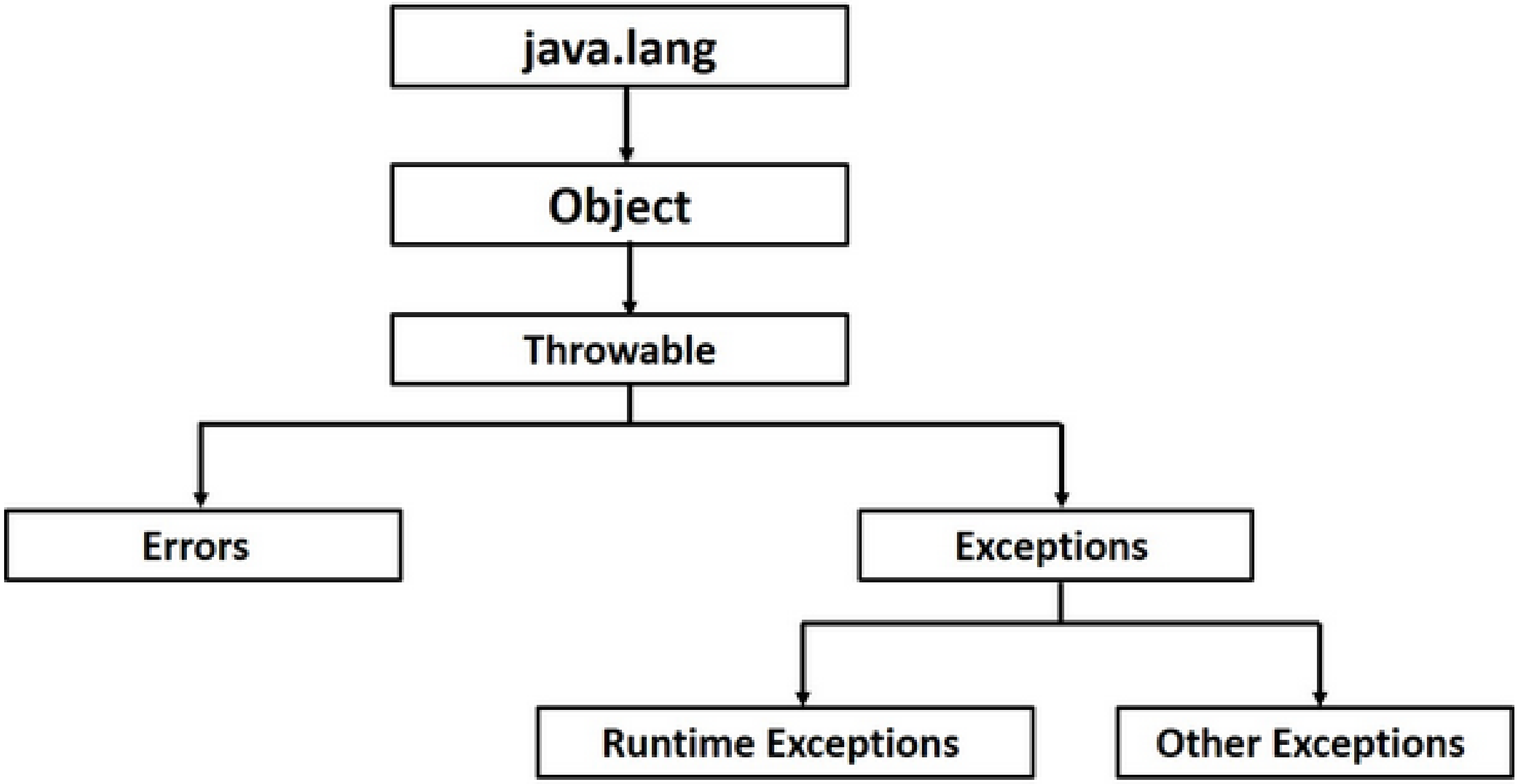
- All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.
- Errors are abnormal conditions that happen in case of severe failures, these are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors

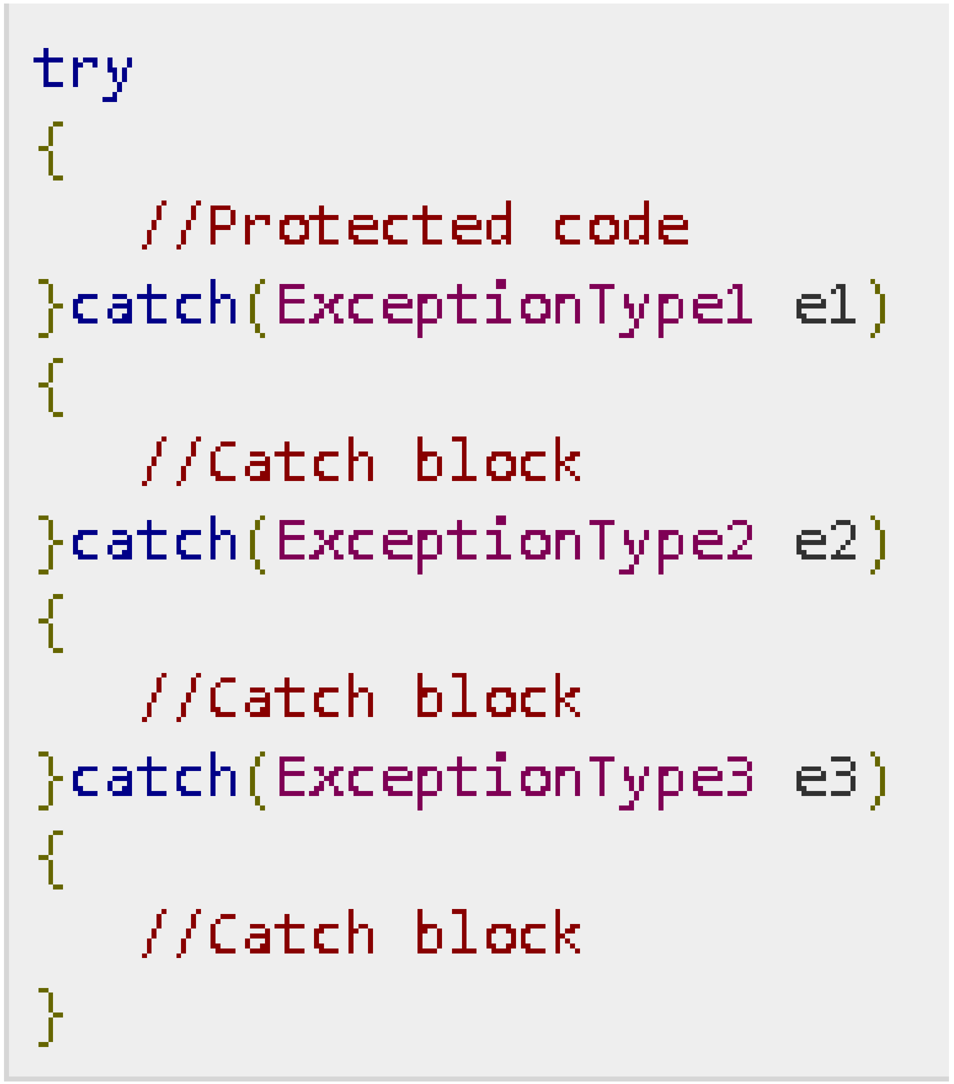
```
try {
        //Protected code
}catch(ExceptionName e1) {
        //Catch block
}
```

Exception Hierarchy



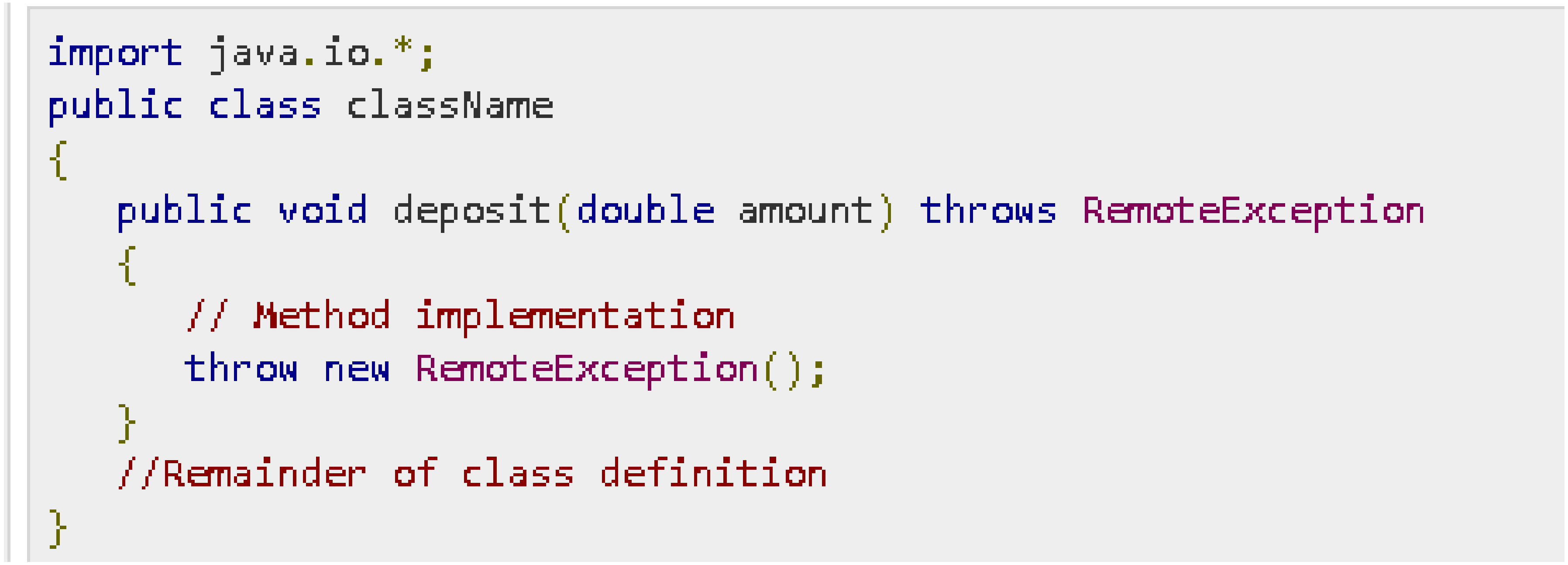
Exception Hierarchy


java.lang
Object
Throwable
Errors
Exceptions
Runtime Exceptions
Other Exceptions



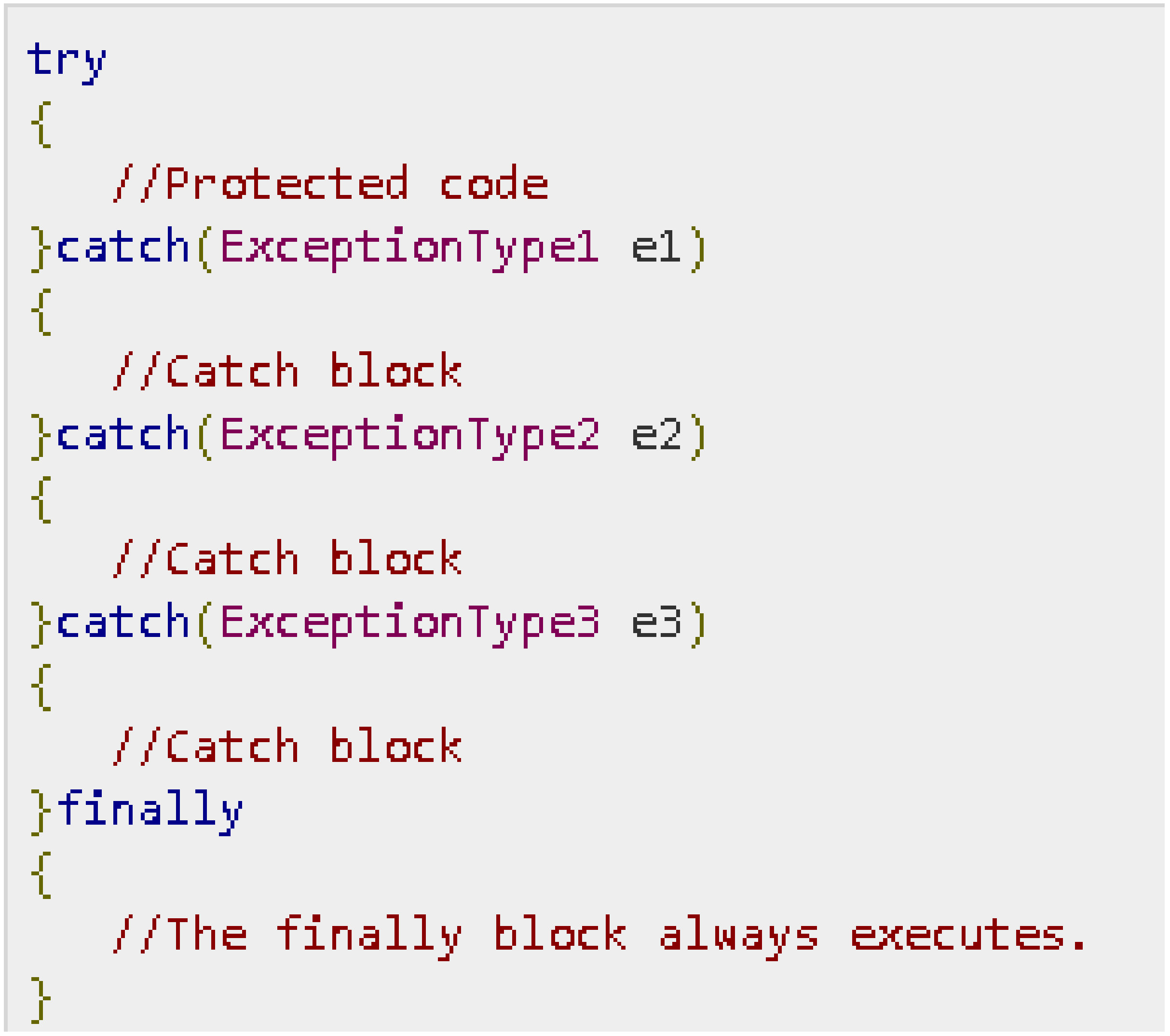
```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}
```

The throws/throw Keywords:



```
import java.io.*;
public class className
{
   public void deposit(double amount) throws RemoteException
   {
      // Method implementation
      throw new RemoteException();
   }
   //Remainder of class definition
}
```

The finally block



```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}finally
{
   //The finally block always executes.
}
```

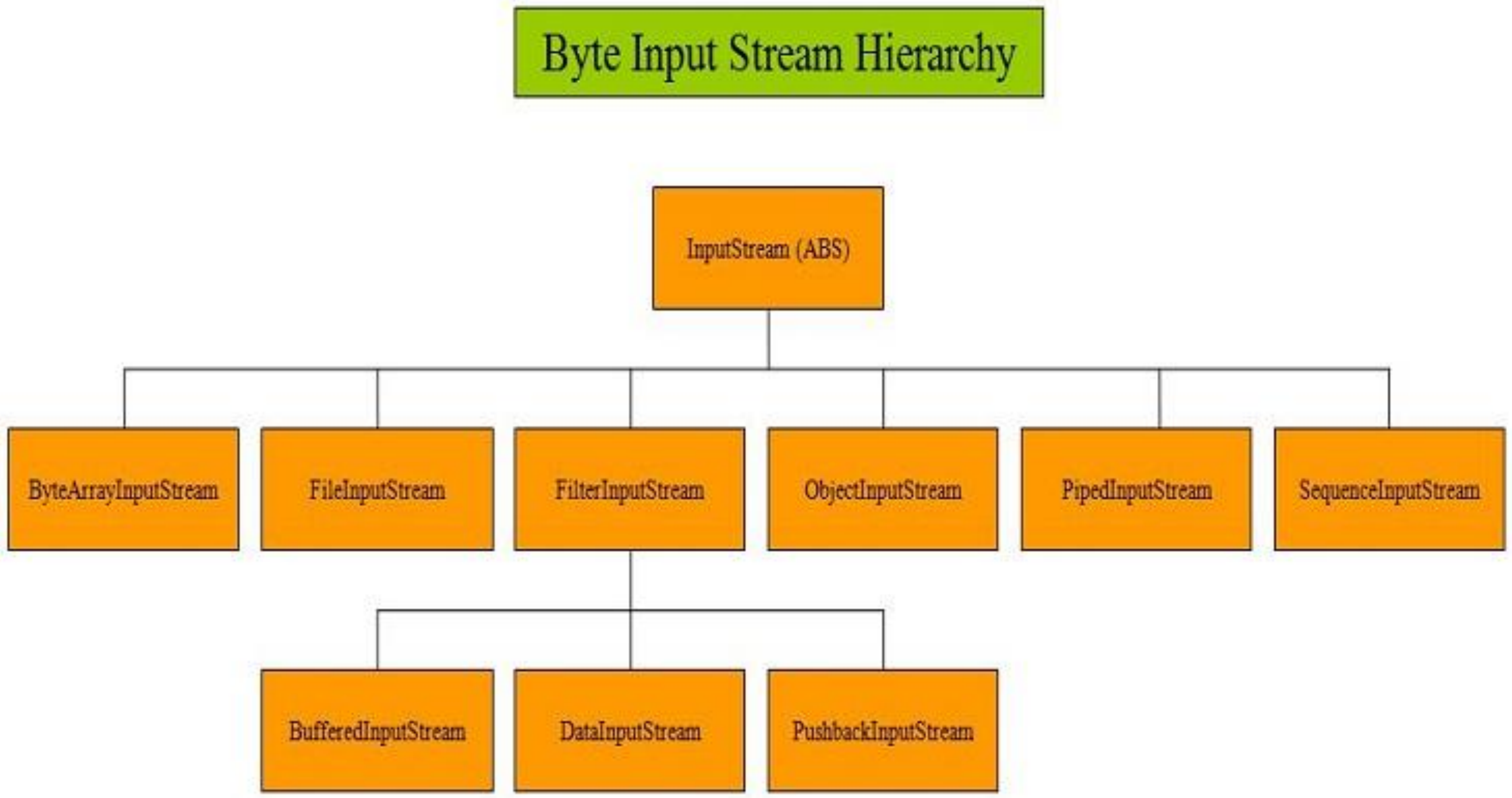
Java Basic

Basic I/O

Basic I/O

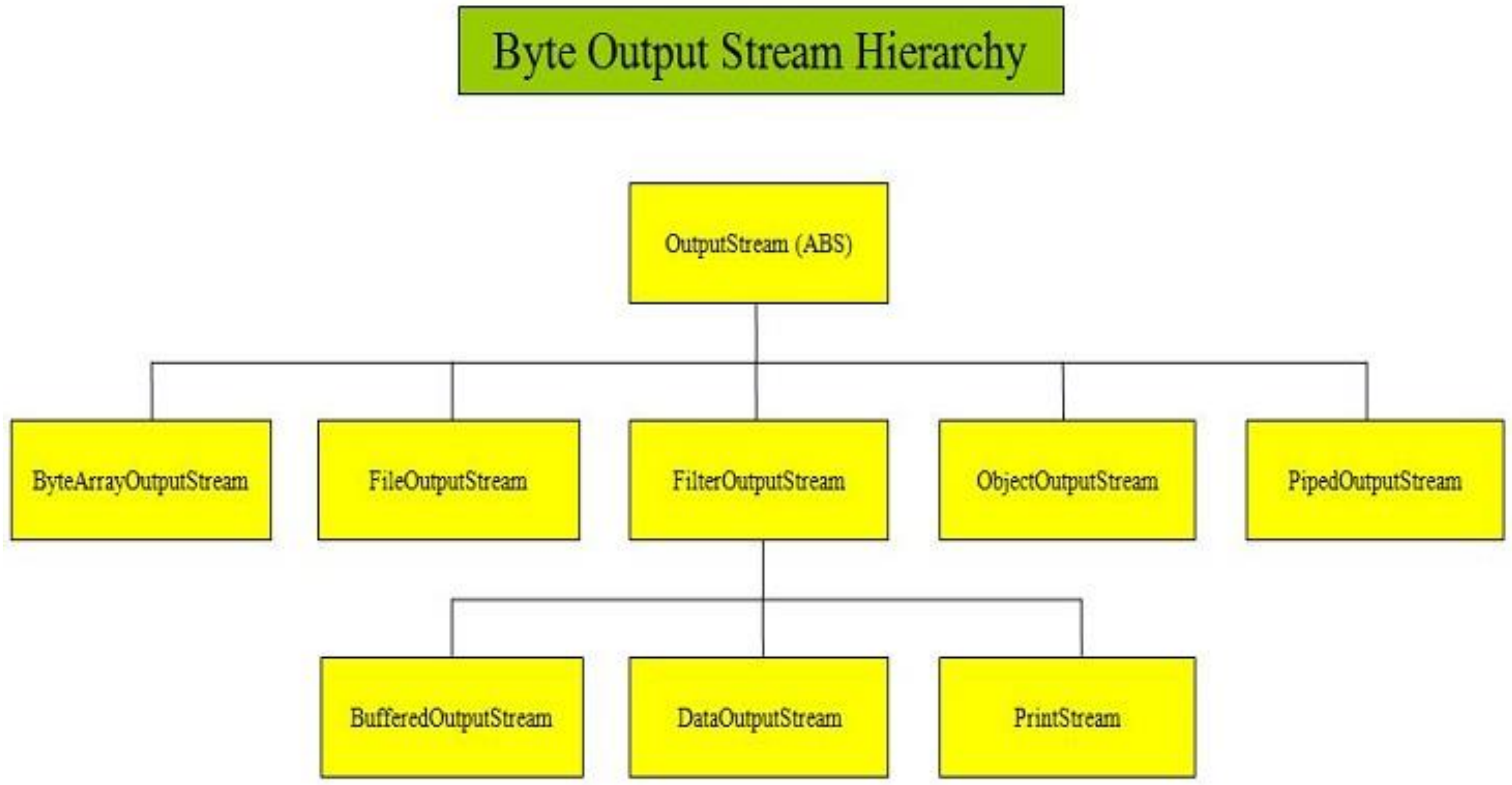
- A *stream* can be thought of as a flow of data from a source or to a sink.
- A *source* stream initiates the flow of data, also called an input stream.
- A *sink* stream terminates the flow of data, also called an output stream.
- Sources and sinks are both *node streams*.
- Types of node streams:
  - Byte streams
  - Character streams
  - Buffered streams
  - Data streams
  - Object streams

Basic I/O



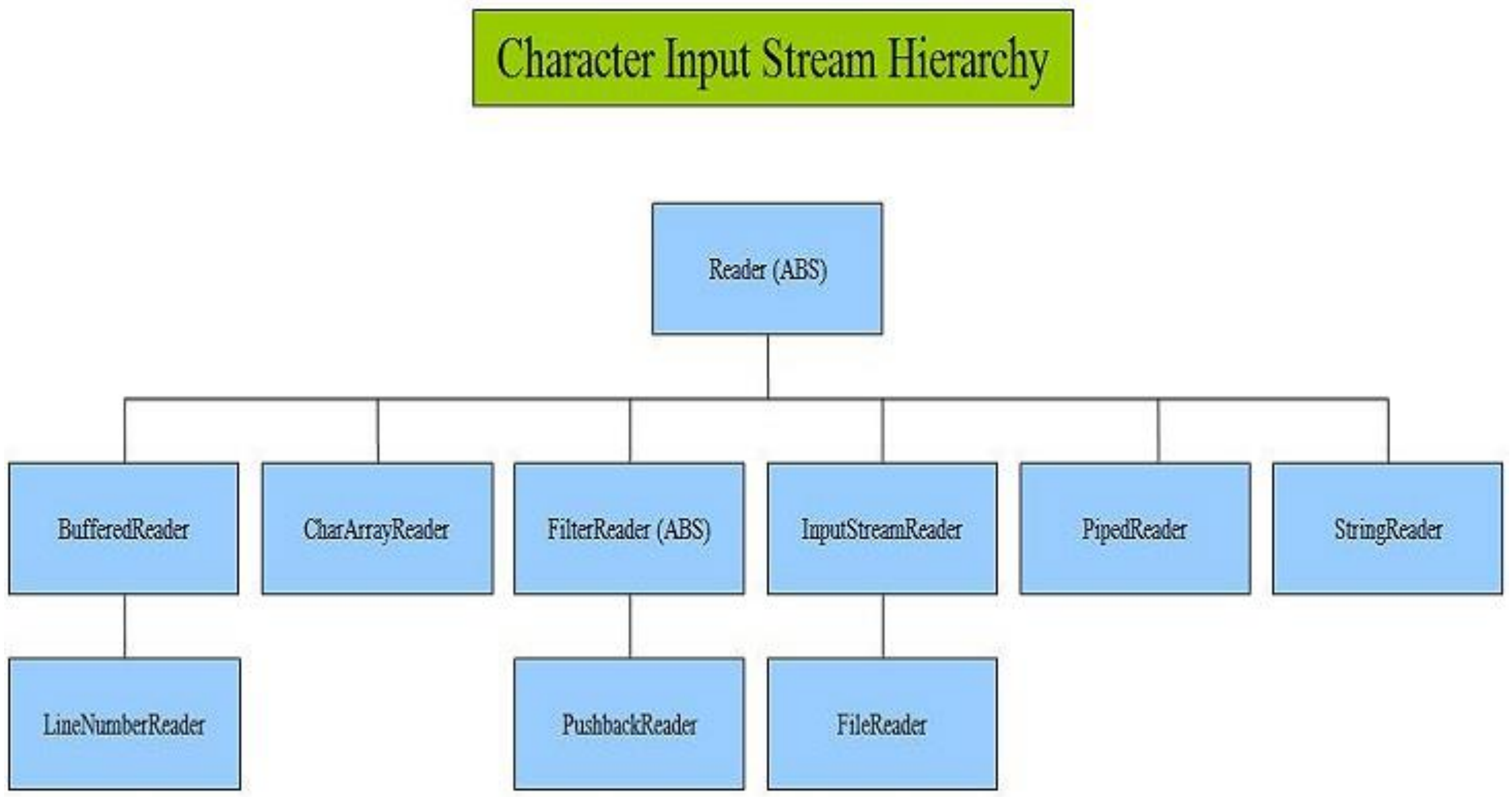
Byte Input Stream Hierarchy

Basic I/O



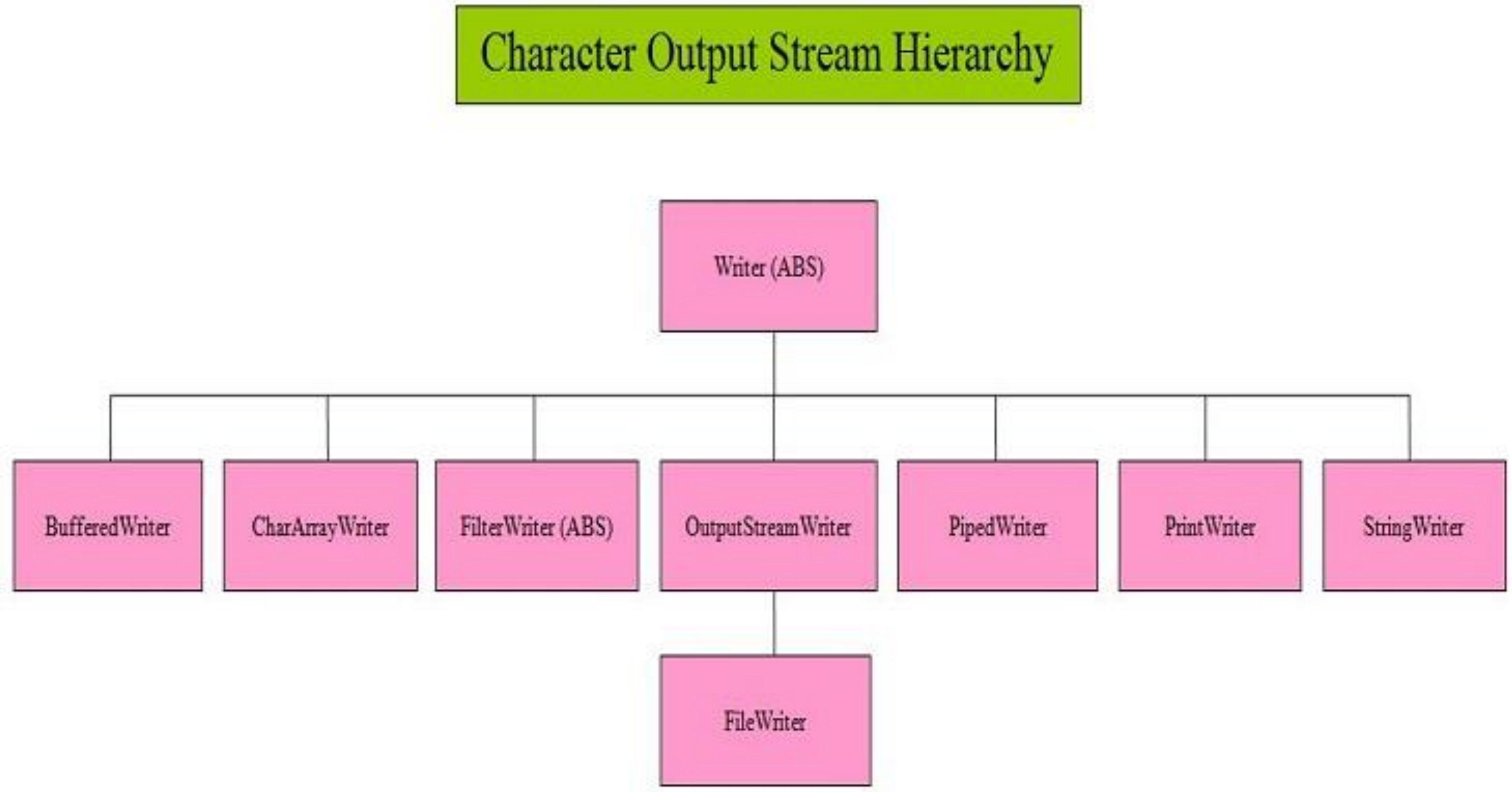
Byte Output Stream Hierarchy

Basic I/O



Character Input Stream Hierarchy

Basic I/O



Character Output Stream Hierarchy

Demo 4: Basic I/O

- Exercise: Save customer with orders to file
- Points to remember:
  - Use classes provided in I/O package to read and write from/to a file
  - Handle exception

Java Advanced

- Java - Data Structures
- Java - Collections
- Java - Generics
- Java - Serialization
- Java - Networking
- Java - Sending Email
- Java – Multithreading
- Java - Documentation

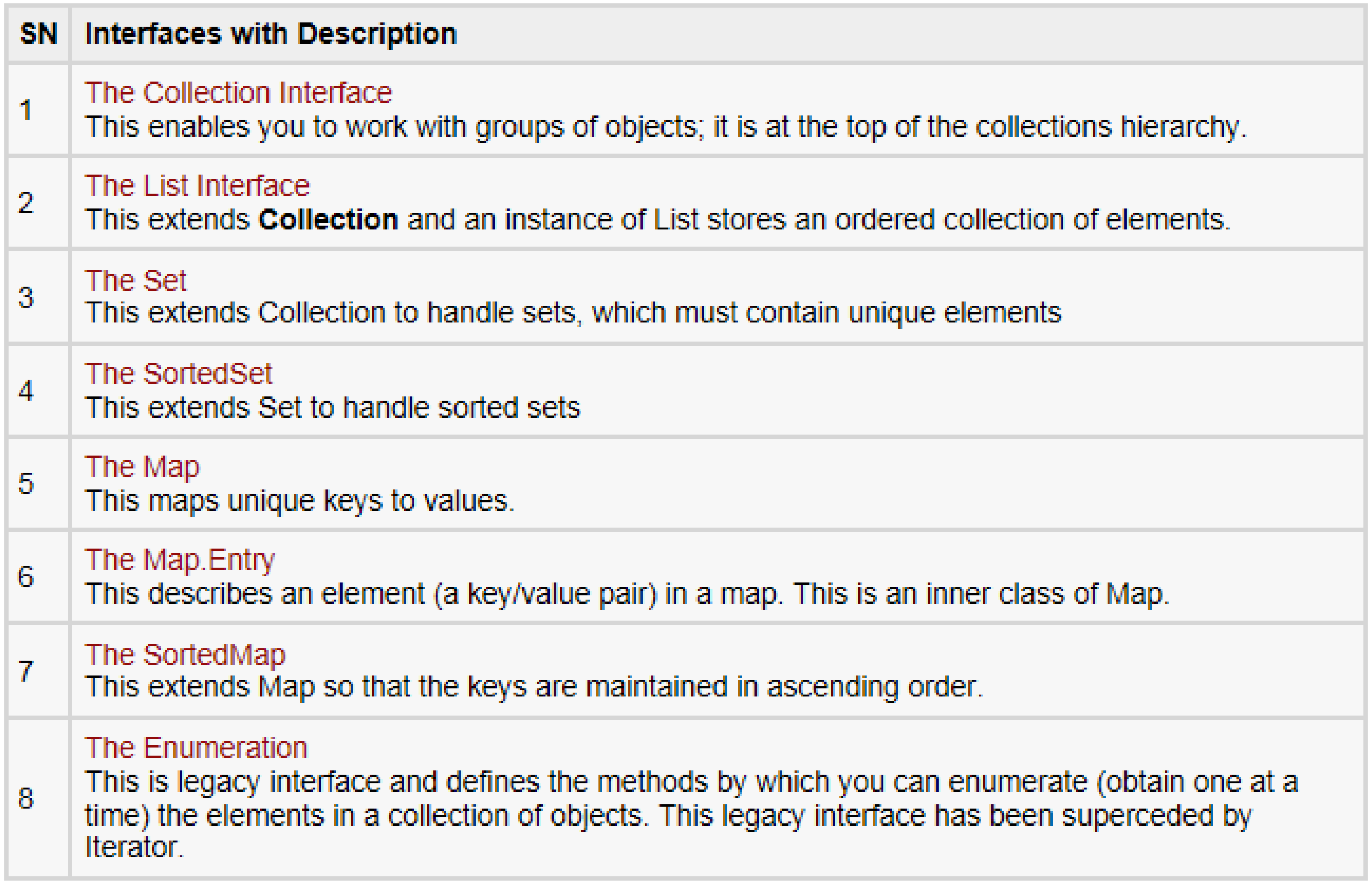
Java - Data Structures

- The data structures provided by the Java utility package are very powerful and perform a wide range of functions. These data structures consist of the following interface and classes:
  - **Enumeration** is interface isn't itself a data structure
  - **BitSet** is class implements a group of bits or flags that can be set and cleared individually
  - **Vector** is class is similar to a traditional Java array
  - **Stack** is class implements a last-in-first-out (LIFO) stack of elements
  - **Dictionary** is class is an abstract class that defines a data structure for mapping keys to values
  - **Hashtable** is class provides a means of organizing data based on some user-defined key structure
  - **Properties** is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

Java Collections Framework

- A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:
  - Interfaces: Collection, List, Map
  - Implementations, i.e., Classes: ArrayList, HashSet, HashMap
  - Algorithms: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces

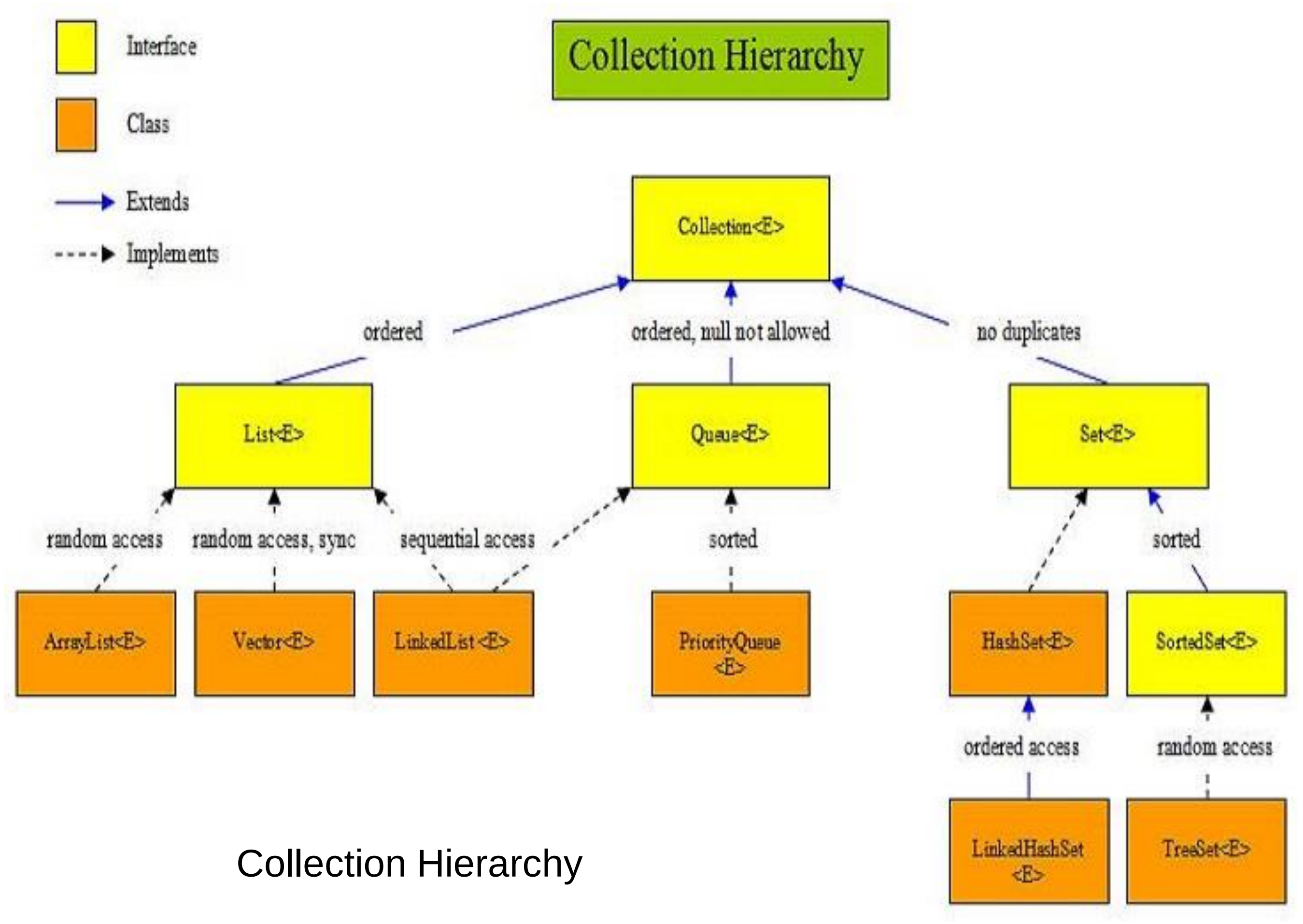
Java Collections Framework



SN	Interfaces with Description
1	The Collection Interface This enables you to work with groups of objects; it is at the top of the collections hierarchy.
2	The List Interface This extends **Collection** and an instance of List stores an ordered collection of elements.
3	The Set This extends Collection to handle sets, which must contain unique elements
4	The SortedSet This extends Set to handle sorted sets
5	The Map This maps unique keys to values.
6	The Map.Entry This describes an element (a key/value pair) in a map. This is an inner class of Map.
7	The SortedMap This extends Map so that the keys are maintained in ascending order.
8	The Enumeration This is legacy interface and defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator.

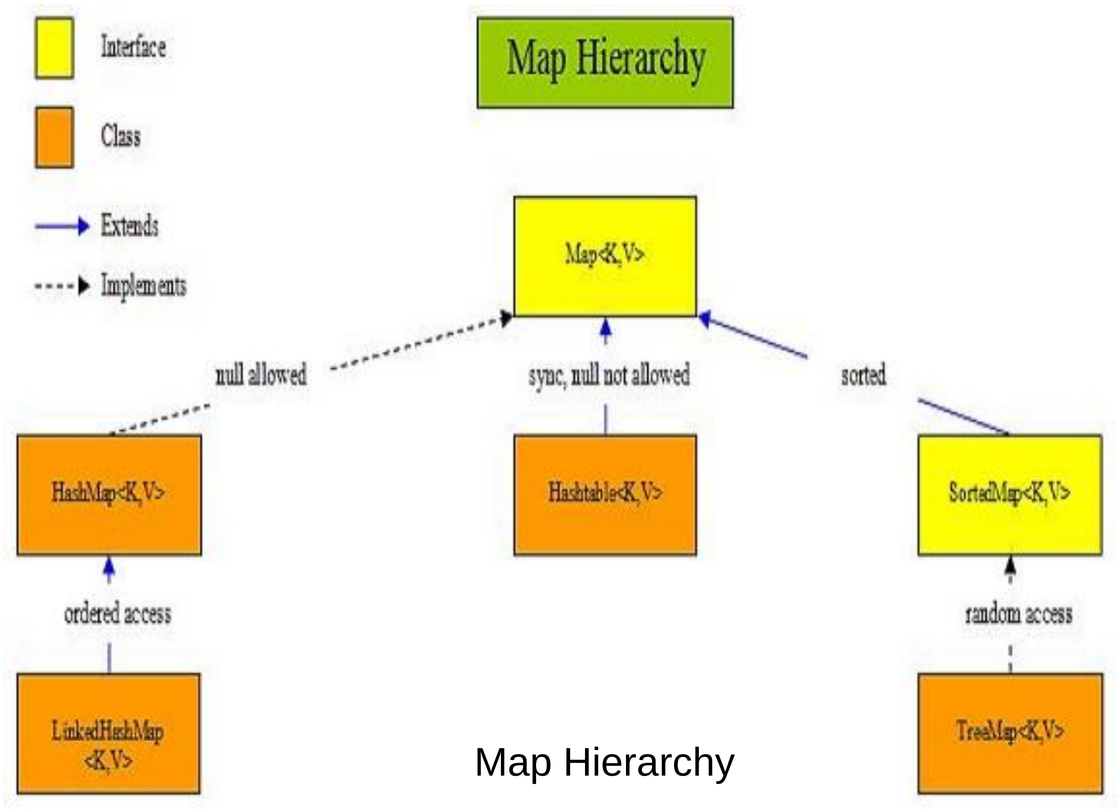
Interface of Collection

Java Collections Framework



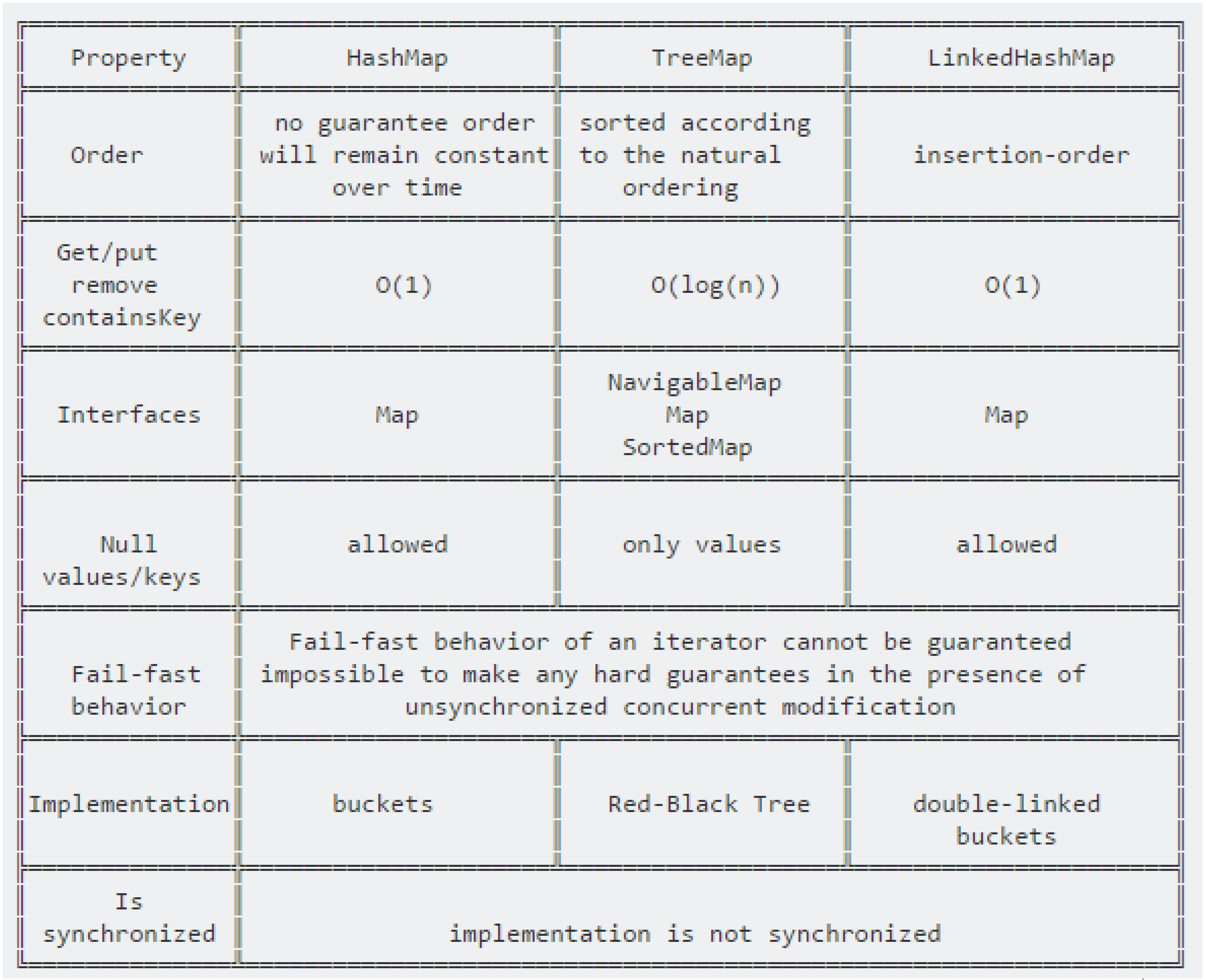
Collection Hierarchy

Java Collections Framework



Map Hierarchy

Comparable HashMap, TreeMap and LinkedHashMap



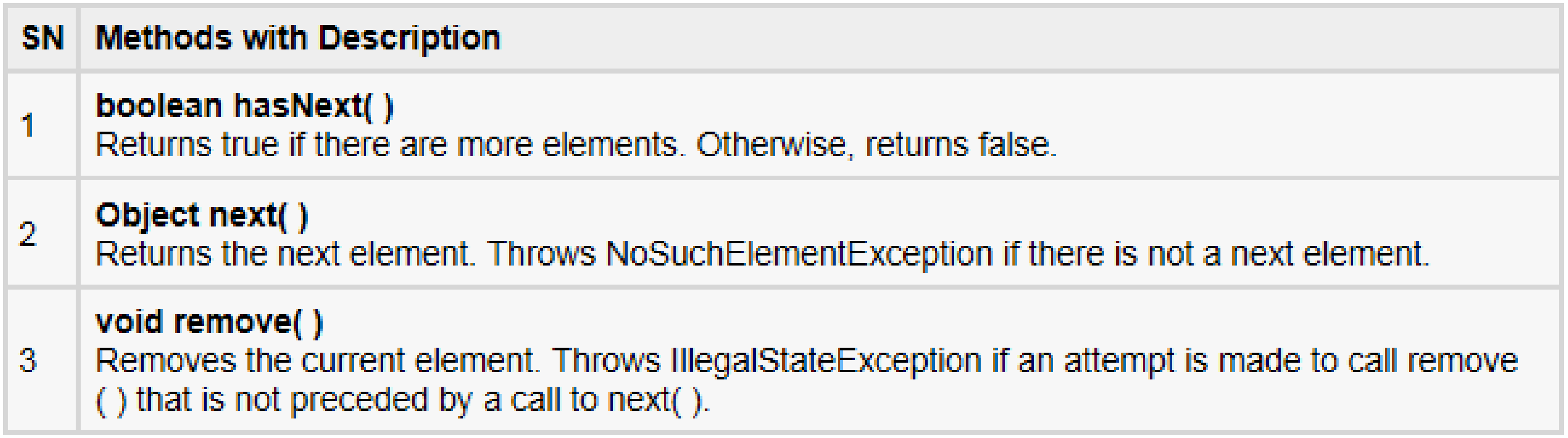
Property	HashMap	TreeMap	LinkedHashMap
Order	no guarantee order will remain constant over time	sorted according to the natural ordering	insertion-order
Get/put remove containsKey	O(1)	O(log(n))	O(1)
Interfaces	Map	NavigableMap Map SortedMap	Map
Null values/keys	allowed	only values	allowed
Fail-fast behavior	Fail-fast behavior of an iterator cannot be guaranteed impossible to make any hard guarantees in the presence of unsynchronized concurrent modification		
Implementation	buckets	Red-Black Tree	double-linked buckets
Is synchronized	implementation is not synchronized		

Java Collections Framework

- Iterator: In general, to use an iterator to cycle through the contents of a collection, follow these steps:
  - Obtain an iterator to the start of the collection by calling the collection's iterator() method.
  - Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as hasNext() returns true.
  - Within the loop, obtain each element by calling next().

Java Collections Framework

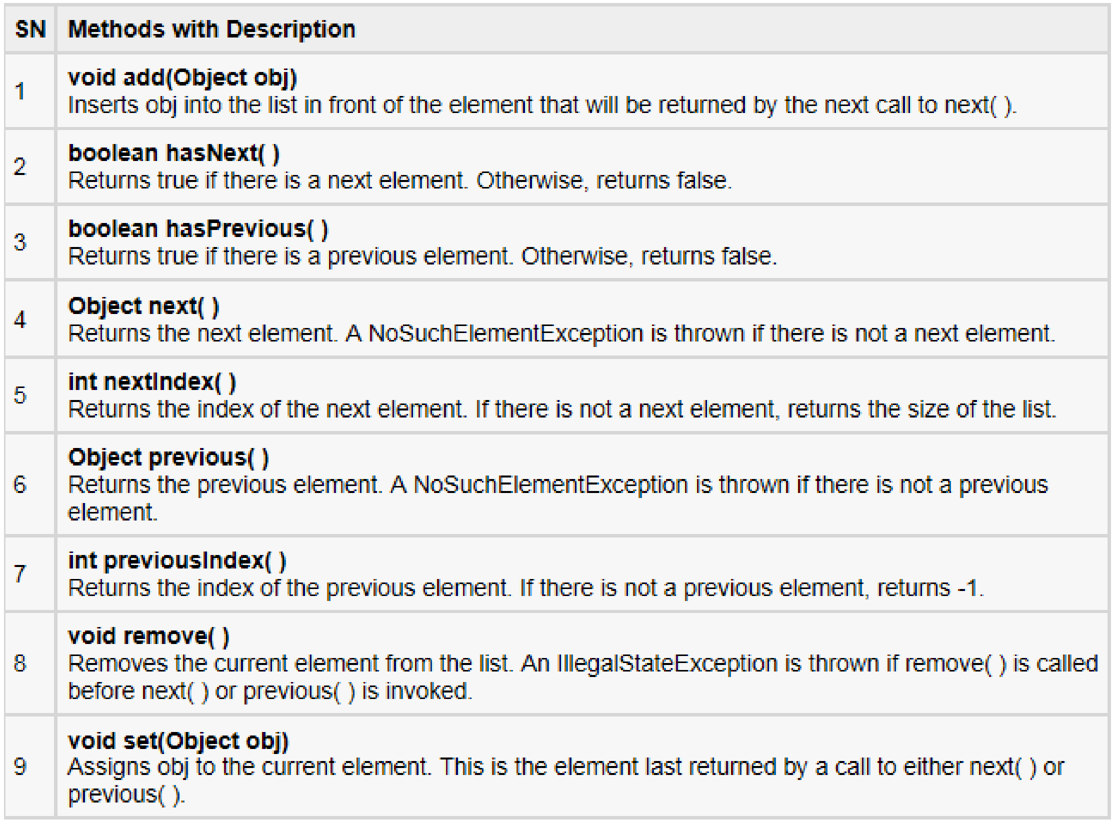
- The Methods Declared by Iterator:



SN	Methods with Description
1	**boolean hasNext( )** Returns true if there are more elements. Otherwise, returns false.
2	**Object next( )** Returns the next element. Throws NoSuchElementException if there is not a next element.
3	**void remove( )** Removes the current element. Throws IllegalStateException if an attempt is made to call remove ( ) that is not preceded by a call to next( ).

Java Collections Framework

- The Methods Declared by ListIterator:



SN	Methods with Description
1	**void add(Object obj)** Inserts obj into the list in front of the element that will be returned by the next call to next( ).
2	**boolean hasNext( )** Returns true if there is a next element. Otherwise, returns false.
3	**boolean hasPrevious( )** Returns true if there is a previous element. Otherwise, returns false.
4	**Object next( )** Returns the next element. A NoSuchElementException is thrown if there is not a next element.
5	**int nextIndex( )** Returns the index of the next element. If there is not a next element, returns the size of the list.
6	**Object previous( )** Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
7	**int previousIndex( )** Returns the index of the previous element. If there is not a previous element, returns -1.
8	**void remove( )** Removes the current element from the list. An IllegalStateException is thrown if remove( ) is called before next( ) or previous( ) is invoked.
9	**void set(Object obj)** Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ).

Methods Declared by ListIterator

Java Collections Framework

Comparator	Comparable
A comparator object is capable of comparing two different objects. The class is not comparing its instances, but some other class's instances. This comparator class must implement the java.util.Comparator interface	A comparable object is capable of comparing itself with another object. The class itself must implements the java.lang.Comparable interface in order to be able to compare its instances.
int compare(Object o1, Objecto2)	int compareTo(Object o1)

TreeSet(Comparator)
java.util.Collections.sort(List, Comparator)
	TreeSet()
	java.util.Collections.sort(List)

Comparator and Comparable

Demo 3: Collection Framework

- Exercise:
  - Add Customer class which has: name, memberType (VIP, MEMBER, OTHERS) and list of orders
  - Sort orders in demo 2 by date, desc.
  - Display orders in console
- Points to remember:
  - Initialize collections
  - Manage collections
  - Order collections
  - Generic, advanced loop

Java - Generics

- Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods or, with a single class declaration, a set of related types, respectively.
- Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic method

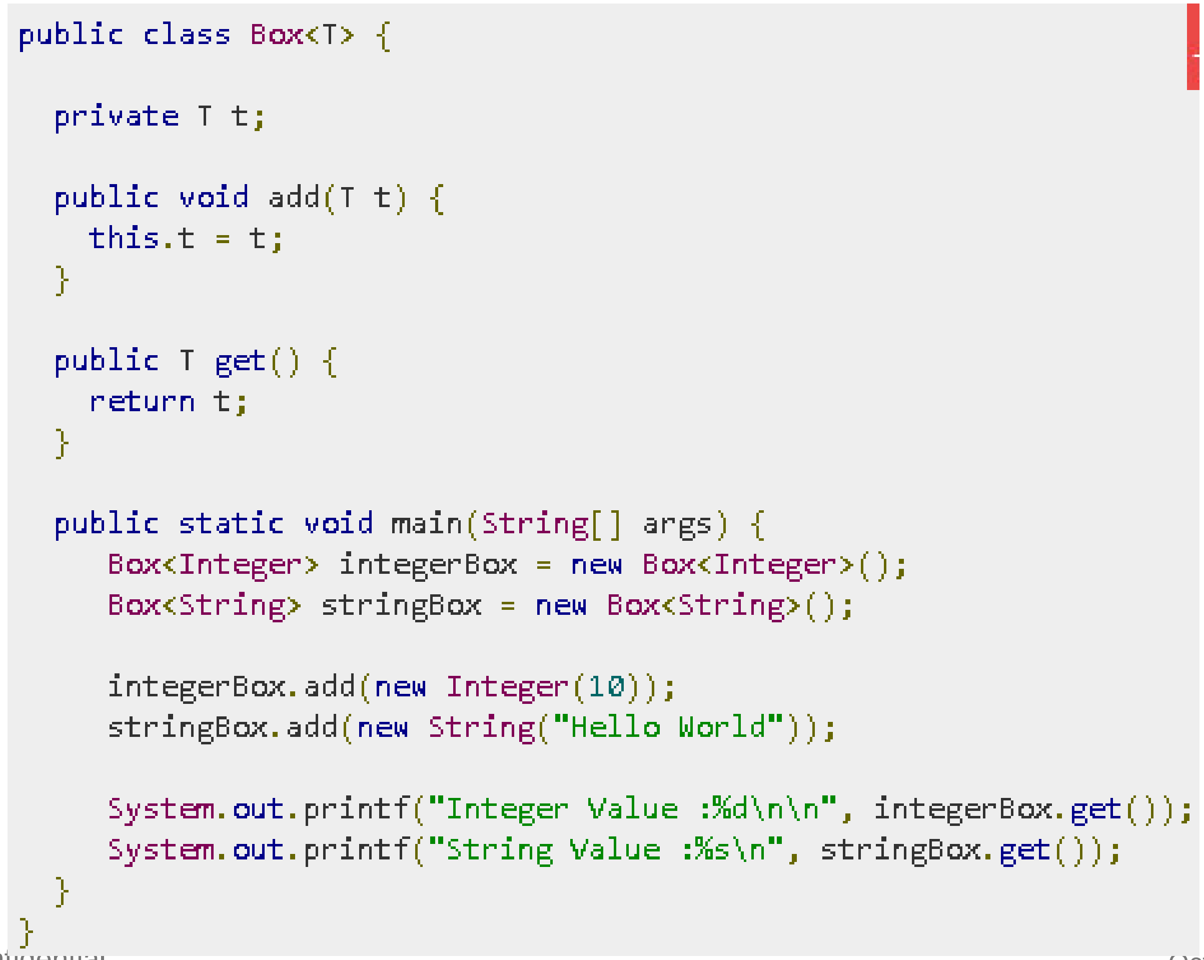
- You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods:
  - All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
  - Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
  - The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
  - A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Generic method

```
public class GenericMethodTest
{
   // generic method printArray
   public static < E > void printArray( E[] inputArray )
   {
      // Display array elements
         for ( E element : inputArray ){
            System.out.printf( "%s ", element );
         }
         System.out.println();
    }

    public static void main( String args[] )
    {
        // Create arrays of Integer, Double and Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println( "Array integerArray contains:" );
        printArray( intArray  ); // pass an Integer array

        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // pass a Double array

        System.out.println( "\nArray characterArray contains:" );
        printArray( charArray ); // pass a Character array
    }
```

Generic Class

- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.



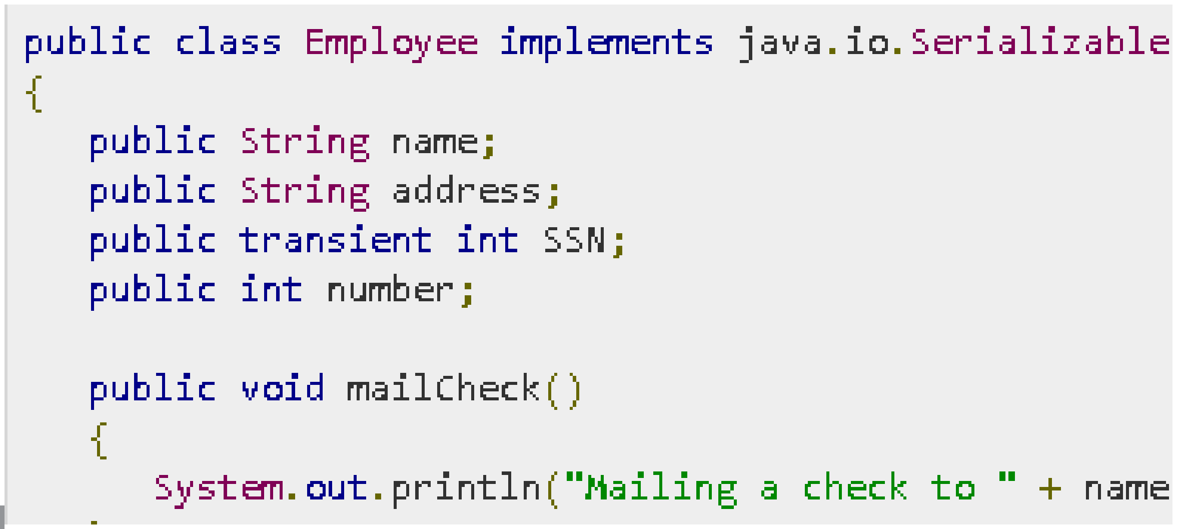
```java
public class Box<T> {

  private T t;

  public void add(T t) {
    this.t = t;
  }

  public T get() {
    return t;
  }

  public static void main(String[] args) {
     Box<Integer> integerBox = new Box<Integer>();
     Box<String> stringBox = new Box<String>();

     integerBox.add(new Integer(10));
     stringBox.add(new String("Hello World"));

     System.out.printf("Integer Value :%d\n\n", integerBox.get());
     System.out.printf("String Value :%s\n", stringBox.get());
  }
}
```

Java Advance

Serialization

Serialization

- Interface Serializable:
  - A marker interface, when implementing it, it enables classes to serialize/deserialize their state
  - Use ObjectOutputStream/ObjectInputStream to write/read an object to/from a stream (or file)



```
public class Employee implements java.io.Serializable
{
   public String name;
   public String address;
   public transient int SSN;
   public int number;

   public void mailCheck()
   {
      System.out.println("Mailing a check to " + name
```

Serializing an Object

- FileOutputStream
- The ObjectOutputStream class is used to serialize an Object
- Note: When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.

```
        FileOutputStream fileOut =
        new FileOutputStream("/tmp/employee.ser");
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
```

Deserializing an Object

- FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
  ObjectInputStream in = new ObjectInputStream(fileIn);

Demo 5: Serialization

- Exercise:
  - Serialize customer with orders to a file
  - Deserialize customer from the file, print out to console and compare
- Points to remember:
  - Usage of Serializable interface and Object Input/Output Stream

Java Advance

Sending email

Java sending email

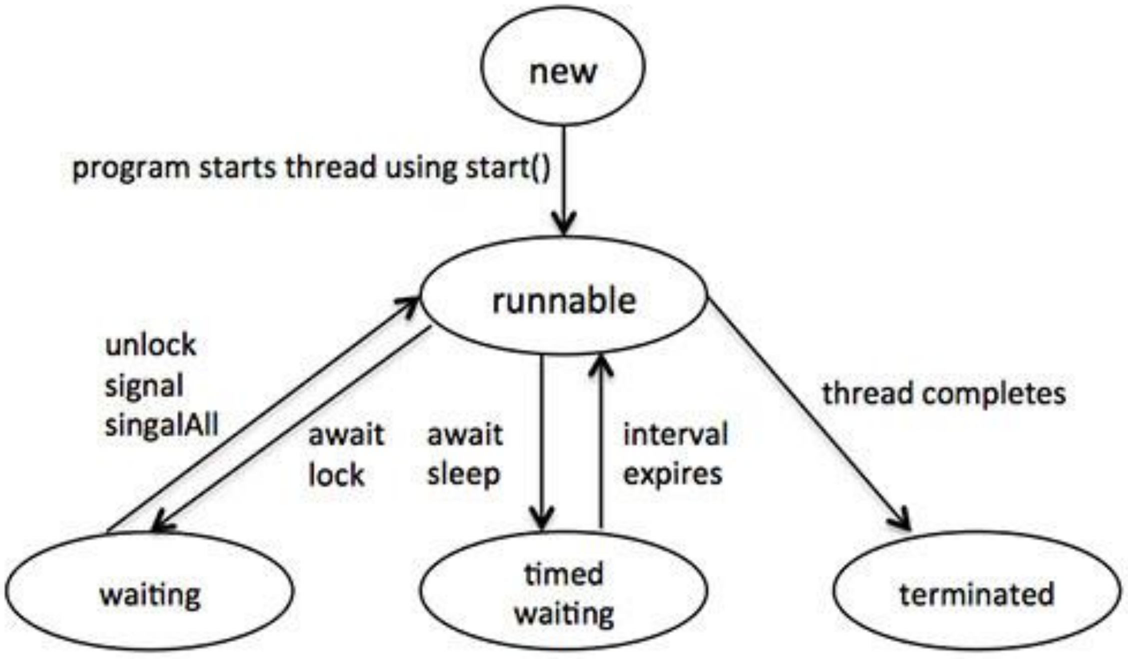
- To send an e-mail using your Java Application is simple enough but to start with you should have **JavaMail API** and **Java Activation Framework (JAF)** installed on your machine.
  - Send a Simple E-mail
  - Send an HTML E-mail:
  - Send Attachment in E-mail:
  - User Authentication Part: provide user ID and Password to the e-mail server for authentication purpose then you can set these properties

Java Advance

Multithreading

Multithreading – Thread definition

- A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.
- States:



Thread states

Multithreading – Creating thread

- Two ways to create a thread:
  - Implement Runnable interface
  - Extend Thread class

Multithreading - Concurrency

- Thread Interference
- Memory Consistency Errors
- Synchronization: synchronized methods, statements
- Deadlock
- Thread Communication: wait, notify, notifyAll

Demo 6: Thread

- Exercise:
  - Create two threads accessing and modifying one customer.
  - Each thread prints the order to console.
  - Investigate the result
- Points to remember:
  - Create thread
  - Experience multithreading issues.

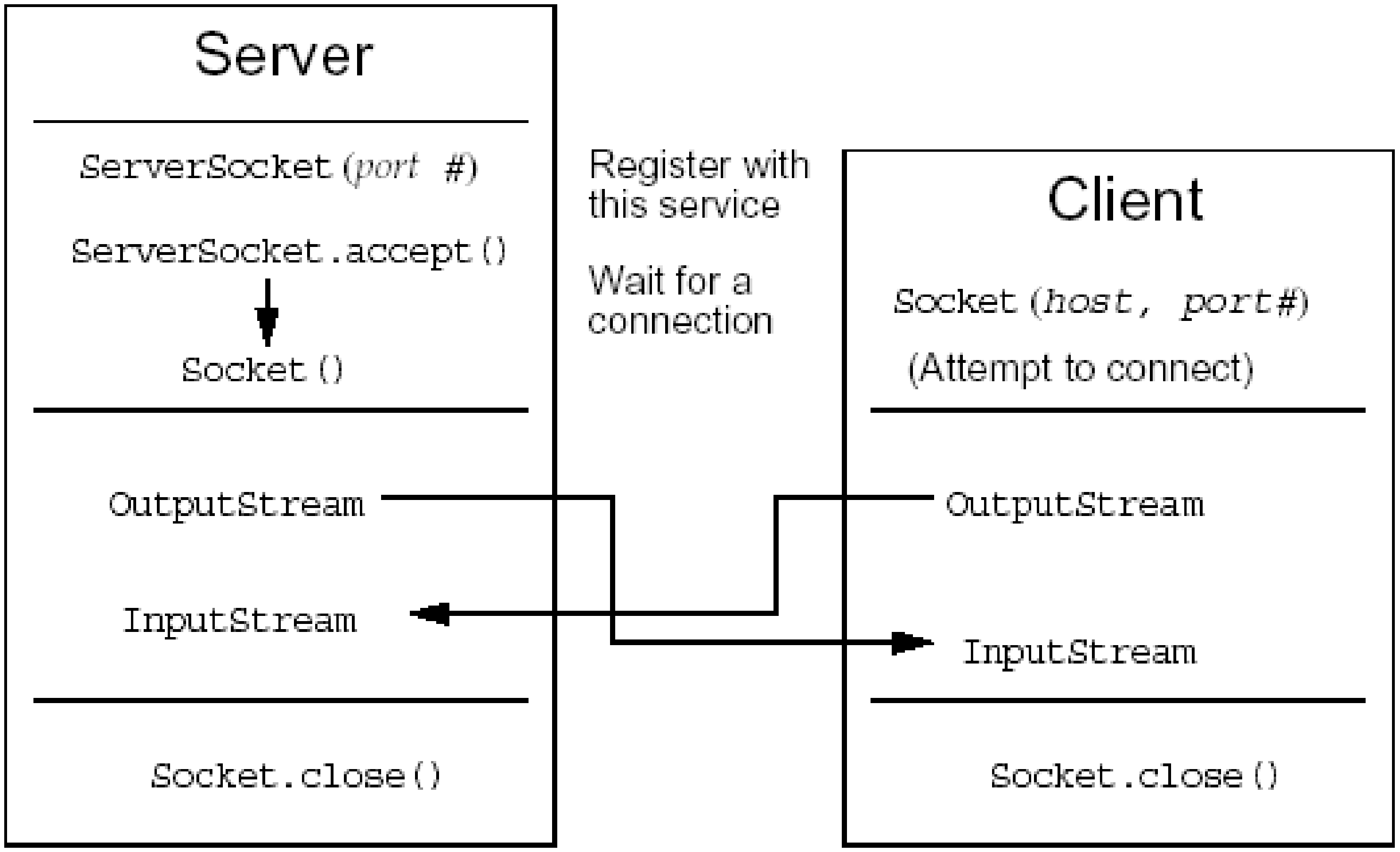
Java Advance

Sockets

Sockets

- A socket is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program
- The java.net package provides Socket that implements the client side of the connection
- The java.net package provides ServerSocket that implement the server side of the connection

Sockets



Socket server and client

Sockets

- Open a socket.
- Open an input stream and output stream to the socket.
- Read from and write to the stream according to the server's protocol.
- Close the streams.
- Close the socket.

Demo 7: Socket

- Exercise:
  - Create socket client and server.
  - Server hold an Order object.
  - When a client connecting, server sends client that order
  - Client shows order to console
- Points to remember:
  - Create socket
  - Communication between to sockets.

Java Advance

Java Document

Java Document

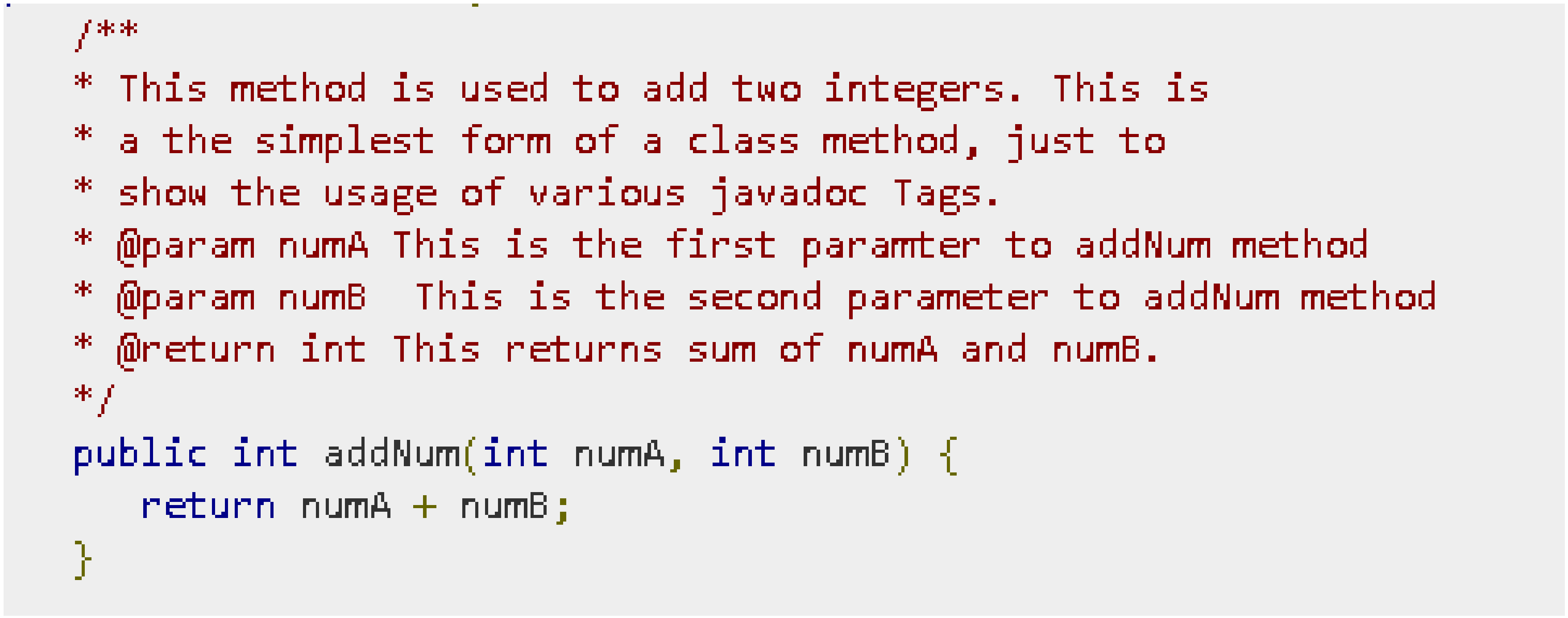
- The Java language supports three types of comments

  /* text */; // text; /** documentation */

- Javadoc is a tool which comes with JDK and it is used for generating Java code documentation in HTML format from Java source code which has required documentation in a predefined format.

```
/**
* The HelloWorld program implements an application that
* simply displays "Hello World!" to the standard output.
*
* @author  Zara Ali
* @version 1.0
* @since   2014-03-31
*/
public class HelloWorld {
    public static void main(String[] args) {
        /* Prints Hello, World! on standard output.
        System.out.println("Hello World!");
    }
}
```

Java Document



```
/**
* This method is used to add two integers. This is
* a the simplest form of a class method, just to
* show the usage of various javadoc Tags.
* @param numA This is the first paramter to addNum method
* @param numB  This is the second parameter to addNum method
* @return int This returns sum of numA and numB.
*/
public int addNum(int numA, int numB) {
   return numA + numB;
}
```

Final Project

- This is a client – server application
- Client asks for a customer with a name
- If server doesn't have a customer with that name, it creates new one, otherwise it returns the existing one.
- Client can:
  - add order(s) to customer
  - print orders to console
  - send customer back to server to save
- Server saves (serializes) customer to file
- One customer has a list of orders, sorting by date (desc)
- Depending on customer type (VIP, MEMBER, OTHERS) customer receives different discount percentage

Thank You

Revision History

Date	Version	Description	Updated by	Reviewed and Approved By
01/01/2015	1.0	Initialize	Nam Vu	
01/12/2015	2.0	Update Image and template	Khoa Le	
10/04/2016	3.0	Update detail slide	Trang Huynh	

CSC

BUSINESS SOLUTIONS
TECHNOLOGY
OUTSOURCING