Java Cheatsheet via JPassion

# Java Documentation

<http://docs.oracle.com/javase/8/docs/api/index.html>

Java

- What is the best Java library to use for HTTP POST, GET etc.? [closed]

- http://stackoverflow.com/questions/1322335/what-is-the-best-java-library-to-use-for-http-post-get-etc

# Passing End of Transmission (Ctrl + D) character in Eclipse console

If you leave the console to focus on another view, and then refocus on the console, then ctrl + d (EOF) works as expected in "Run" node. Still doesn't work in "Debug" mode.

# Primitive Java types

* byte, short, int, long, float, double, boolean, and char

# Comments

## single line

// comment until end of line

## multiple line comment

/\* Mltiple

\* line comment

\*/

## Javadoc comments

/\*\*

\* Javadoc multiple-line comment

\* @author Jeanne and Scott

\*/

# Statements

## if

if (hourOfDay < 11) {

System.out.println("Good Morning");

} else if (hourOfDay < 15) {

System.out.println("Good Afternoon");

} else {

System.out.println("Good Evening");

}

- Examine Warning: Beware of indentation suggesting block structure when placement of braces

define block structure and not spacing (i.e. morningGreeting is always executed below).

if(houorOfDah < 11)

System.out.println("Good Morning");

morningGreeting++;

- Examine Warning: Unreachable code can result if order of if-else's are incorrect.

- Examine Warning: Verify that if expressions evaluate to a boolean expression.

- Examine Warning: Make sure == in boolean expressions and not =

## while loop

* While loop may terminate after its first evaluation of the boolean epression, the statement block may never be executed.
* Make sure the loop variable is modified, else infinite loop will occur.

String[] days = {"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};

System.out.println("Display days of week using while loop");

int counter = 0;

while (counter < days.length) {

System.out.println(days[counter]);

counter++;

}

## do-while loop

* Will execute the statement block first, and then checks the loop condition.

String[] days = {"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};

int counter = 0;

do {

System.out.println(days[counter]);

counter++;

} while (counter < days.length);

## for loop

* Two types of for statements: (1) basic for loop, and (2) enhanced for loop (aka for-each).
* Initialization and update sections may contain multiple statements, separated by commas.
* Variables declared in the initialization block of a for loop have limited scope and are only accessible within the for loop.
* The boolen condition is evaluated on every iteration of the loop before the loop executes.
* Veriables in the initialization block must all be of the same type.

### Basic for loop

// Basic for loop

String[] days = {"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};

for (int counter = 0; counter < days.length; counter++){

System.out.println(days[counter]);

}

Non-standard for loops: 1. Creating in infinite loop

for( ; ; ) { // Infinite loop

System.out.printn('Hello World");

}

Non-standard for loops: 2. Adding Multiple Terms to the for statement

int x = 0;

for(long y = 0, z = 4; x < 5 && y < 10; x++, y++) {

System.out.printn(y + " ");

}

Non-standard for loops: 3. Redeclaring a Variable in the Initialization Block

int x = 0;

for (long y = 0, x = 4; x < 4 && y < 10; x++, y++) { DOES NOT COMPILE

System.out.println( x + " ");

}

Non-standard for loops: 4. Using Incompatible Data Types in the Initialization Block

for(long y = 0, int x = 4; x < 5 && y < 10; x++, y++ { DOES NOT COMPILE

System.out.println(x + " ");

}

Non-standard for loops: 5. Using Loop Variables Outside the loop

for(long y = 0, x = 4; x < 5 && y < 10; x++, y++) {

System.out.println(y + " ");

}

System.out.println(x); // DOES NOT COMPILE

### for-each

* Used to traverse an object.
* The right-hand side of the for-ach loop statement must be a bult-in Java array or an object whose class implements java.lang.Iterable; and the left-hand side is a matching type.
* OCA exam only uses List and ArrayList from Collection Framework.

for (datatype instance : collection) {

// body

{

// Enhanced for loop (aka for-each)

String[] days = {"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};

for (Object o: days){

System.out.println( o );

}

or

String[] days = {"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};

for (String s: days){

System.out.println( s );

}

## switch statement

* Default gets all values not expectedly checked.
* Default can go anywhere in the list.
* Data type of the switch variable must match the data type of the case statements.
* Each case statement must be compile-time constant values of the same data type as the switch value: literals, enum constants or final constant variable of the same data type. A final constant is one defined with the final modifier and initialized with a literal value in the same expression in which it is declared.
* Data types supported by switch statements include:

- byte and Byte

- short and Short

- int and Integer

- char and Character

- String

- enum value

switch (expression) {

case value:

//Statements

break; //optional

case value:

//Statements

break; //optional

//You can have any number of case statements.

default: //Optional

//Statements

}

# [Where is Java Installed on Mac OS X](http://stackoverflow.com/questions/15826202/where-is-java-installed-on-mac-os-x)?

$ /usr/libexec/java\_home

/Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home

java -version

java version "1.8.0"

$ /usr/libexec/java\_home -v 1.8.0

/Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home

# javac cmd line args

$ javac

Usage: javac <options> <source files>

where possible options include:

-g Generate all debugging info

-g:none Generate no debugging info

-g:{lines,vars,source} Generate only some debugging info

-nowarn Generate no warnings

-verbose Output messages about what the compiler is doing

-deprecation Output source locations where deprecated APIs are used

-classpath <path> Specify where to find user class files and annotation processors

-cp <path> Specify where to find user class files and annotation processors

-sourcepath <path> Specify where to find input source files

-bootclasspath <path> Override location of bootstrap class files

-extdirs <dirs> Override location of installed extensions

-endorseddirs <dirs> Override location of endorsed standards path

-proc:{none,only} Control whether annotation processing and/or compilation is done.

-processor <class1>[,<class2>,<class3>...] Names of the annotation processors to run; bypasses default discovery process

-processorpath <path> Specify where to find annotation processors

-parameters Generate metadata for reflection on method parameters

-d <directory> Specify where to place generated class files

-s <directory> Specify where to place generated source files

-h <directory> Specify where to place generated native header files

-implicit:{none,class} Specify whether or not to generate class files for implicitly referenced files

-encoding <encoding> Specify character encoding used by source files

-source <release> Provide source compatibility with specified release

-target <release> Generate class files for specific VM version

-profile <profile> Check that API used is available in the specified profile

-version Version information

-help Print a synopsis of standard options

-Akey[=value] Options to pass to annotation processors

-X Print a synopsis of nonstandard options

-J<flag> Pass <flag> directly to the runtime system

-Werror Terminate compilation if warnings occur

@<filename> Read options and filenames from file

# java tools

$ pwd

/Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home/bin

$ ls -la

total 8064

drwxrwxr-x 45 root wheel 1530 Mar 4 2014 .

drwxrwxr-x 16 root wheel 544 Mar 4 2014 ..

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 appletviewer

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 extcheck

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 idlj

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jar

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jarsigner

-rwxrwxr-x 1 root wheel 99296 Mar 4 2014 java

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 javac

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 javadoc

-rwxrwxr-x 1 root wheel 2062 Mar 1 2014 javafxpackager

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 javah

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 javap

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jcmd

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jconsole

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jdb

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jdeps

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jhat

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jinfo

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jjs

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jmap

-rwxrwxr-x 1 root wheel 405 Mar 4 2014 jmc

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jps

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jrunscript

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jsadebugd

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jstack

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jstat

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 jstatd

-rwxrwxr-x 1 root wheel 5183 Sep 11 2013 jvisualvm

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 keytool

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 native2ascii

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 orbd

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 pack200

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 policytool

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 rmic

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 rmid

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 rmiregistry

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 schemagen

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 serialver

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 servertool

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 tnameserv

-rwxrwxr-x 1 root wheel 116480 Mar 4 2014 unpack200

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 wsgen

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 wsimport

-rwxrwxr-x 1 root wheel 99360 Mar 4 2014 xjc

# Java Archive Tool (jar) - Java zip/unzip utility

jar xvf <filename> // Unzip

jar tuf <filename> // View contents

## Common JAR file operations

* To create a JAR file

jar cf jar-file input-file(s)

* To view the contents of a JAR file

jar tf jar-file

* To extract the contents of a JAR file

jar xf jar-file

* To extract specific files from a JAR file

jar xf jar-file archived-file(s)

# Hello World!

**package** mypackage;

**public** **class** Hello {

**public** **static** **void** main(String[] args) {

System.***out***.println("Hello World!");

}

}

# STUDY- Logical Operators

See /Users/ben/Documents/ws2/jpassion/javacourse/chapter-01-02/javase\_progenv.pdf,

Slide # 54

# Round down to nearest 10

((grade/10)\*10)

# Dark UI

<https://www.youtube.com/watch?v=UbV56AmJsHs#t=493.959462>

# Convert string to integer

String age= "21";

int ageint = Integer.parseInt(age);

# Read/display keyboard input - at command line

BufferedReader dataIn = **new** BufferedReader(**new** InputStreamReader( System.***in***) );

try{

name = dataIn.readLine();

} catch ( IOException e ){

System.out.println("Error!");

}

# Read/display keyboard - in dialog box

import javax.swing.JOptionPane;

String name = "";

name = JOptionPane.showInputDialog("Please enter your name");

String msg = "Hello " + name + "!";

JOptionPane.showMessageDialog(null, msg);

# Mac keyboard shortcuts

Command ⌘

Shift ⇧

Option ⌥

Control ⌃

Caps Lock ⇪

Fn

# Identifiers

* May contain letter, number, dallar sign ($), or underscore (\_). Letter can be over 45,000 valid Unicode characters.
* Can not begin with number.
* Can't be same as reserve word.

# Array

## Declaration

For example,

int[] ages; // “ages” is int array type; more popular format

or

int ages[]; // This is allowed as well

## Instantiation

// Declaration

int[] ages;

// Instantiate int array object with length of 100

ages = new int[100];

or

// Declare and instantiate object

int[] ages = new int[100]; // Each item initialized to 0 (zero)

String[ ] ages = new String[10]; // Each item is initialized to null

## Instantiation with data

// Creates an array of int type initialized

int[] ages = { 1, 2, 3, 4, 5 };

// Creates an array of boolean type initialized

boolean[] results = { true, false, true, false };

// Create an array of 4 double type initialized

double[] grades = { 100, 90, 80, 75 };

// Create an array of Strings initialized

String[] days = { “Mon”, “Tue”, “Wed”, “Thu”, “Fri”, “Sat”, “Sun” };

## Access last element in the array

// Print the last element in the array

System.out.println(ages[ages.length - 1]);

## Multidimensional Arrays

// integer array 512 x 128 elements

int[][] twoD = new int[512][128];

// character array 8 x 16 x 24

char[][][] threeD = new char[8][16][24];

// String array 4 rows x 2 columns

String[][] dogs = {

{ "Terry", "brown" },

{ "Kristin", "white" },

{ "Toby", "gray"},

{ "Fido", "black"}

};

System.out.print( dogs[0][0] ); //prints the String "Terry"

## Print two-dimensional array

int[][] ages = new int[10][5];

for( int i = 0; i < ages.length; i++ ) {

System.out.println("\nStarting row " + i);

for( int j = 0; j < ages[i].length; j++ ) {

ages[i][j] = i \* j;

System.out.print( ages[i][j] + " " );

}

}

## Print three-dimensional array

// Declare and create three dimensional int array whose size is 2 x 4 x 6

int[][][] ages = new int[2][4][6];

int value = 1000;

// Display length of each dimension

System.out.println("ages.length = " + ages.length);

System.out.println("ages[1].length = " + ages[1].length);

System.out.println("ages[1][1].length = " + ages[1][1].length);

// Display the value of each entry in the array

for( int i = 0; i < ages.length; i++ ){

System.out.println("\n\nStarting row " + i);

for( int j = 0; j < ages[i].length; j++ ){

System.out.println("\nStarting column " + j);

for( int k = 0; k < ages[i][j].length; k++ ){

ages[i][j][k] = value++;

System.out.print( ages[i][j][k] + " " );

}

}

}

## Compute greatest numer

// Compute the greatest number

for (counter = 0; counter < num.length; counter++) {

if ((counter == 0) || (num[counter] > max))

max = num[counter];

}

## Arrays and loops

// Declare and initialize String array of the days of the week

String[] days = {"Sunday","Monday","Tuesday","Wednesday", "Thursday","Friday","Saturday"};

// Display days of the week using while loop

System.out.println("Display days of week using while loop");

int counter = 0;

while(counter < days.length){

System.out.println(days[counter]);

counter++;

}

// Display days of the week using do-while loop

System.out.println("Display days of week using do-while loop");

counter = 0;

do {

System.out.println(days[counter]);

counter++;

} while (counter < days.length);

// Display days of the week using for loop

System.out.println("Display days of week using for loop");

for (counter = 0; counter < days.length; counter++) {

System.out.println(days[counter]);

}

## Length For String and Array

public class Main {

public static void main( String[] args ){

String myString = "Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec";

String[] myArray = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

System.out.println( "The length of myString is " + myString.length() );

System.out.println( "The length of myArray is " + myArray.length );

}

}

# Iterator and ListIterator

**import** java.util.\*;

**public** **class** MyOwnClass {

**public** **static** **void** main(String[] args) {

ArrayList<String> names = **new** ArrayList<String>();

System.***out***.println("I am reading names as command line arguments.. ");

// Check for correct number of command line argument

**if**(args.length == 0){

System.***out***.println("Please provide one or more name");

System.*exit*(0);

}

**for** (**int** counter = 0; counter < args.length; counter++){

System.***out***.println("Name is " + args[counter]);

names.add(args[counter]);

}

*generateNewName*(names);

Iterator<String> itr = names.iterator();

System.***out***.println("\nNew names are .. ");

**while**(itr.hasNext()) {

Object element = itr.next();

System.***out***.println("Name is " + element);

}

}

**public** **static** **void** generateNewName(ArrayList<String> newNames ) {

ListIterator<String> litr = newNames.listIterator();

**while**(litr.hasNext()) {

Object element = litr.next();

String newElement = Character.*toString*(((String) element).charAt(1));

litr.set(newElement);

}

}

}

# Command-line arguments

## Stored in array

// Each String in the ags array contains one of the commandline arguments.

public class CommandLineSample {

public static void main( String[] args ){

for(int i = 0; i < args.length; i++){

System.out.println( args[i] );

}

}

## Convert from string to number

int firstArg = 0;

if (args.length > 0){

firstArg = Integer.parseInt(args[0]);

}

## Codign Guidelines

Always check the number of arguments before accessing the array elements so that there will be no exception generated.

if( args.length!= 5 ){

System.out.println(“Invalid number of arguments”);

System.out.println(“Please enter 5 arguments”);

}

else{ // Correct number of arguments are passed

//some statements here

}

## Comand line string arguments to int

int int1 = Integer.parseInt(args[0]);

int int2 = Integer.parseInt(args[1]);

# object

* An object is an instance of a class.
* To create an instance of a class, write new before it.

## Comparing Objects with == and != (1st example)

When equality operators equal (==) and not equal (!=) are applied to objects, they determine whether both sides of the operator refer to the same object instance instead of checking whether one object has the same value as the other object.

NOTE: Java String literal object behavior is different bacuse strings in Java are optimized.

public class EqualsTestString {

public static void main(String[] args) {

String strLiteral1 = "Hello";

String strLiteral2 = "Hello";

// String literal and String literal

System.out.println("Same object? " + (strLiteral1 == strLiteral2)); // true

System.out.println("Same value? " + strLiteral1.equals(strLiteral2)); // true

System.out.println();

String strObj3 = new String("Hello");

String strObj4 = new String("Hello");

// String literal and String object

System.out.println("Same object? " + (strLiteral2 == strObj3)); // false

System.out.println("Same value? " + strLiteral2.equals(strObj3)); // true

System.out.println();

// String object and String object

System.out.println("Same object? " + (strObj3 == strObj4)); // false

System.out.println("Same value? " + strObj3.equals(strObj4)); // true

}

}

## Comparing Objects with == and != (2nd example)

package equality;

import java.util.\*;

public class Equality {

public static void main (String[] args) {

int p1 = 1;

int p2 = 1;

String s1 = "Hello World";

String s2 = "Hello World";

String so1 = new String("Sundance");

String so2 = new String("Sundance");

String[] sArray1 = {"Brown", "Fox"};

String[] sArray2 = {"Brown", "Fox"};

String[] sArray3 = sArray1;

char[] cArray1 = so1.toCharArray();

char[] cArray2 = so2.toCharArray();

char[] cArray3 = cArray1;

// Primitive

System.out.println( (p1 == p2) + " int == int (and all other primitives)\n");

// String Literal

System.out.println( (s1 == s2) + " String == String\n");

// String Object

System.out.println( (so1 == so2) + " new StringObj == new StringObj");

System.out.println( (so1.equals(so2)) + " StringObj.equals(StringObj)\n");

// String Array (Object)

System.out.println( (sArray1 == sArray2) + " String[] == String[]");

System.out.println( (sArray1.equals(sArray2)

+ " String[].equals(String[]) // when pointing to differnt memeory locations"));

System.out.println( (sArray1.equals(sArray3)

+ " String[].equals(String[]) // when pointing to same memeory locations"));

System.out.println( (Arrays.equals(sArray1, sArray2)) + " Arrays.equals(String[], String[])\n");

// Char Array (Object)

System.out.println( (cArray1 == cArray2) + " char[] == char[]");

System.out.println( (cArray1.equals(cArray2)

+ " char[].equals(char[]) // when pointing to differnt memeory locations"));

System.out.println( (cArray1.equals(cArray3)

+ " char[].equals(char[]) // when pointing to same memeory locations"));

System.out.println( (Arrays.equals(cArray1, cArray2)) + " Arrays.equals(char[], char[])\n");

}

}

\*\*\* Output

true int == int (and all other primitives)

true String == String

false new StringObj == new StringObj

true StringObj.equals(StringObj)

false String[] == String[]

false String[].equals(String[]) // when pointing to differnt memeory locations

true String[].equals(String[]) // when pointing to same memeory locations

true Arrays.equals(String[], String[])

false char[] == char[]

false char[].equals(char[]) // when pointing to differnt memeory locations

true char[].equals(char[]) // when pointing to same memeory locations

true Arrays.equals(char[], char[])

## Object getClass() method and instanceof operator

Integer int1 = new Integer(34);

System.out.println("int1.getClass() - " + int1.getClass() );

// class java.lang.Integer

System.out.println("int1.getClass().getName() - " + int1.getClass().getName() );

// java.lang.Integer

System.out.println("int1.getClass().hashCode() - " + int1.getClass().hashCode() );

// 118352462

System.out.println("int1.getClass().toString() - " + int1.getClass().toString() );

// class java.lang.Integer

System.out.println("int1 instanceof Integer - " + (int1 instanceof Integer) );

// true

System.out.println("int1 instanceof Number - " + (int1 instanceof Number) );

// true

System.out.println("int1 instanceof Object - " + (int1 instanceof Object) );

// true

# java.lang.\* Classes

## Wrapper Classes

* Provides Object representations of primitive types
* – Boolean (Wrapper class), boolean (Primitive)
* – Integer (Wrapper class), int (Primitive)
* – Long (Wrapper class), long (Primitive)
* – Double (Wrapper class), double (Primitive)
* Why #1: Need an object representation for the primitive type variables to use to take advantage of the properties and methods of Object class
* Why #2: Collection can take only Object as its element – no primitive type can be added to a Collection

## Math Class

* Provides predefined constants and methods for performing different mathematical operations
* Methods: Note that they are all static methods

## System Class

* Provides many useful fields and methods
* – Standard input
* – Standard output
* – Utility method for fast copying of a part of an array
* Display System Properties

import java.util.Properties;

public class Sandbox {

public static void main(String[] args) {

Properties p1 = System.getProperties();

p1.list(System.out);

}

}

## Process Class

* Provides methods for manipulating processes
* - Killing the process
* - Running the process
* - Checking the status of the process
* Represents running programs
* - destroy()
* - waitFor()

## Runtime Class

* Represents the runtime environment
* - getRuntime()
* - exec()

# OOP Intro

## inheritance

* *One class acquires the behavior (methods) and properties (fields) of another.*
* The scheme of hiding implementation details of a class
* The implementation can change without affecting the user of the class
* Inheritance is the concept of a child class automatically inheriting the properties (fields) and behavior (methods) defined in its parent class.
* - Parent class is also called as super class
* - Child class is also called as sub class

## inheritance, why? Reusability

* Once a set of properties (fields) are defined in a super class, the same set of properties are inherited by all subclasses
* - A class and its children share common set of properties
* Once a set of behavior (methods) are defined in a super class, those behavior are automatically inherited by all subclasses
* - Thus, you write a method only once in a super class and it can be used by all subclasses.
* A subclass only needs to implement the differences (methods and properties) between itself and its parent

## encapsulation

* *Wrapping the properties (fields) and code acting on the data (methods) together as a single unit.*

## polymorphism

* *Polymorphism is the ability of an object to take on many forms.*
* The ability in computer programming to present the same programming interface for differing underlying forms

## class

* Represents a “type” from which an object can be created
* Made up of fields and methods
* Provide the benefit of reusability

## object

* Is a runtime instance of a class in menory

## fields, (variables, properties, attributes)

* Hold the state of the program
* Specify the data types defined by the class

## methods (functions, procedures)

* Operate on the state
* Specify the behavior
* A block of code (set of statements) that can be called to perform some specific task

## members of the class

* Fields and methods are the members of the class

## "Object" class

* Object class is mother of all classes
* - In Java language, all classes are subclassed (extended) from the Object super class
* - Object class is the only class that does not have a parent class
* Defines and implements behavior common to all classes
* - .equals() // lang.object
* - .getClass() // lang.object
* - .getClass().getName() // java.lang.Class<T>
* - .hashCode() // lang.object
* - .toString() // lang.object

## child class

* Child class uses the extends keyword to extend the parent class
* A subclass inherits all of the “public” and “protected” members (fields or methods) of its parent, no matter what package the subclass is in
* If the subclass is in the same package as its parent, it also inherits the default (sometimes called package-private) members (fields or methods) of the parent default (package-private) members are members with no modifier (these are called default modifier as well)

## child class, fields (properties)

* The inherited fields from the parent class can be used directly, just like any other fields defined in the subclass
* You can declare new fields in the subclass that are not in the super class
* A subclass does not inherit the private members of its parent class, however, and cannot access them directly

## child class, methods (behavior)

* The inherited methods from the parent class can be used directly
* You can write a new instance method in the subclass that has the same signature as the one in the super class, thus overriding it, thus providing a new behavior other than the one from the super class
* This is called “**overriding a method**” (note that this is different from “overloading method” concept)
* You can declare new methods in the subclass that are not in the super class.

## method signature

* Identified by method name, number of parameters and type of parameters.
* Return type has no effect on method signature.

## final classes

* Classes that cannot be extended
* Examples of final classes are Java primitive wrapper classes (Interger, Long, etc) and String class.
* Reason: Don't want any deviation from desired behavior

public final class ClassName{

. . .

}

## final method

* Methods that cannot be overridden
* Static methods are automatically final
* Reason: Don't want any deviation from desired behavior

public final [returnType] [methodName]([parameters]){

. . .

}

# Abstract and Interface

## abstraction

* *Hiding the implementation details from the user, only the functionality will be provided to the user*

## abstract method

* Method that does not have implementation (code block)
* To create an abstract method, just write the method declaration without the body (code block) and use the abstract keyword
* Since there is no body of code, there is no { }
* Example:

public abstract void someAbstractMethod();

## abstract class

* An abstract class is a class that contains one or more abstract methods
* An abstract class cannot be instantiated - you cannot create an object instance from it
* Another class (Concrete class) has to be created to provide implementation of all abstract methods
* Concrete class uses extends keyword to extend the abstract class
* Concrete class has to implement all abstract methods of the abstract class in order to be used for instantiation - compile error if not all abstract methods are implemented

## why abstract class and methods?

* Abstract methods are usually declared where two or more subclasses (Concrete classes) are expected to fulfill a similar role in different ways through different implementations (this is Polymorphism)
* These subclasses extend the same Abstract class and provide different implementations for the abstract methods of the Abstract class
* Use abstract classes to define broad types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

## interface

* All methods of an interface are abstract methods (while in an abstract class, only some methods are abstract methods)
* - Defines the signatures of a set of methods, without the body (implementation of the methods)
* - No need to use abstract modifier for the methods since all methods in an interface are considered as abstract methods
* A concrete class must implement the interface
* - All abstract methods have to be implemented (otherwise, it is a compile error)
* Use interface modifier to the declaration

## why interface?

* It defines a public way of specifying the behavior of classes
* Defines a contract between “client” (user of the interface) and “provider” (implementation) of some task

Reason #1

* In order to reveal an object's programming interface (functionality of the object) without revealing its implementation.This is the concept of **encapsulation**
* The **implementation can change** without affecting the caller (user) of the interface
* The caller does not need the implementation class at compile time only interface is needed
* - Different groups can work in parallel with only interfaces defined
* - During run-time, object instance gets created (from implementation class) and is associated with the interface type

Reason #2

* To have unrelated classes implement similar methods (behaviors)

Reason #3

* To model multiple inheritance - you want to impose multiple sets of behaviors to your class
* Each set is defined as an interface
* A class can implement multiple interfaces while it can extend only one class

## Interface vs. Abstract Class

* All methods of an Interface are abstract methods while some methods of an Abstract class are abstract methods
* Abstract methods of Abstract class must have abstract modifier
* An interface can only define constants while abstract class can have fields

# Interface as a Type

## Interface as a Type (Very important) #1

* When you define a new interface, you are defining a new type - you can use an interface anywhere you can use a class as a type
* Use of Interface rather than Class as a type is strongly recommended
* Let's say Person class implements PersonInterface interface, now you can do

Person p1 = new Person(); // Class is used as reference type

PersonInterface pi1 = p1; // Interface is used as reference type

PersonInterface pi2 = new Person(); // Interface is used as reference type

## Interface as a Type (Very important) #2

* Interfae used as argument type is recommended practice

// Class is used as an argument type

aMethod(Person p){

...

}

// Interface is used as argument type

aMethod(PersonInterface p){

...

}

## Interface vs. Class: Commonality

* Interface and Class can both define methods
* Interfaces and classes are both types - this means that an interface can be used in places where a class can be used

// Recommended practice

PersonInterface pi = createPerson();

// Not recommended practice because

// - Person class implementation has to be present even during compile time

// - Tightly coupled to Person class implementation

Person pc = createPerson();

## Interface vs. Class: Differences

* The methods of an Interface are all abstract methods; they cannot have bodies
* You cannot create an instance from an interface - you have to create a concrete class that implements the interface for object instantiation

// Compile error

PersonInterface pi = new PersonInterface();

## Defining an Interface

* To define an interface, use interface keyword
* Example: Create an interface that defines relationships between two objects according to “some order” of the objects.

public interface Relation {

public boolean isGreater( Object a, Object b);

public boolean isLess( Object a, Object b);

public boolean isEqual( Object a, Object b);

}

## Implementing Interface

* To create a concrete class that implements an interface, use the implements keyword.

// Line class implements Relation interface

public class Line implements Relation {

// Code ...

}

* When your class tries to implement an interface, always make sure that you implement all methods of that interface, or else, you would encounter this error.

## Which one to use Class vs Interface/Impl

* Let's assume you are provide a class that will be used by other people
* You have two choices
* – Option #1: Provide Person class
* – Option #2: Provide PersonInterface and PersonImpl class
* Choose Option #2 for most cases
* – Other people do need just interface at compile time
* – You can change the implementation without affecting them
* Choose Option #1
* – When the class will be used rarely and do not change

## Relationship of an Interface to a Class

* A concrete class can only extend one super (parent) class through inheritance relationship, but it can implement multiple Interfaces
* – The Java programming language does not permit multiple inheritance, but interfaces provide an alternative.
* All abstract methods of all interfaces have to be implemented by the concrete class, however
* – Otherwise, a compile error will occur
* Example:

public class ComputerScienceStudent

extends Student

implements PersonInterface,

AnotherInterface,

ThirdInterface{

// All abstract methods of all interfaces

// need to be implemented.

}

## Inheritance Among Interfaces

* Interfaces are not part of the class hierarchy
* However, interfaces can have inheritance relationship among themselves

public interface PersonInterface {

void doSomething();

}

public interface StudentInterface

extends PersonInterface {

void doExtraSomething();

}

## Problem of Rewriting an Existing Interface

* If you add a new method to an existing interfade, old code with break because it has not impleneeted the new intervase.
* Solution, create a new interface that extends the old and adds the new method.
* Example:

public interface DoItPlus extends DoIt {

boolean didItWork(int i, double x, String s);

}

## When to use an Abstract Class over Interface?

* For non-abstract methods, you want to use them when you want to provide common implementation code for all subclasses – Reducing the duplication
* For abstract methods, the motivation is the same with the ones in the interface – to impose a common behavior for all childclasses without dictating how to implement it
* Remember a concrete class can extend only one super class whether that super class is in the form of concrete class or abstract class

# Polymorphism

## Polymorphism in a Java program

* The ability of a reference variable to change behavior according to what object instance it is referring to
* This allows objects of different subclasses to be treated as objects of a single super class, while automatically selecting the proper methods to apply of a particular object based on the subclass it belongs to
* This allows objects of implementation classes to be treated as objects of an interface, while automatically selecting the proper methods to apply of a particular object based on implementation class

## Benefits of Polymorphism: Simplicity

* If you need to write code that deals with a family of subtypes, the code can ignore type-specific implementation details and just interact with the base type of the family
* - Higher level of abstraction results in simpler code
* Even though the code thinks it is using an object of the base class, the object's class could actually be the base class or any one of its subclasses
* - Depending on which type of object gets used
* This makes your code easier to write and easier for others to understand

## Benefits of Polymorphism: Extensibility

* Other subclasses could be added later to the family of subtypes, and objects of those new subclasses would also work with the existing code
* No code change is required in order to accommodate the addition of the other subtypes in the future

## 3 Forms of Polymorphism in Java program

* Scheme #1 - Method overriding
* > Methods of a subclass override the methods of a superclass
* > This is the example code we've seen so far
* Scheme #2 - Implementation of abstract methods of an Abstract class
* > Methods of a subclass implement the abstract methods of an
* abstract class
* Scheme #3 - Implementation of abstracts methods of a Java interface
* > Methods of a concrete class implement the methods of the interface

# Variables (fields, properties, attributes)

## variables, 3 types

* Class variable (Also called as Static variable)
* Instance variable (Also called as Non-static variable)
* Local variable (Also called as automatic variable)

public class Car {

// Class (Static) variable

private static String manufacturer = "Ford";

// Instance (non-Static) variable

private String plateNumber;

private String color;

public Car() {

}

public void accelerate(){

// Local (automatic) variable

int x = 10;

}

}

## Class (static) variable

* Declared inside a class body but outside of any method bodies (same as Instance variable)
* Prepended with the static modifier (different from Instance variable)
* Exists per each class – Come to existence when the class is loaded
* Shared by all object instances of the class
* When create static variables?
* - Variables that belong to the class
* - This means that they have the same value for all the object instances in the same class

## Instance (non-static) variable

* Declared inside a class body but outside of any method bodies (like static variable)
* Exists per each object instance
* Different object instances typically have different values for these instance variables
* Come to existence when an object instance is created
* When create instance variable?
* - Belongs to an object instance
* - Values of variables of an object instance are different from the ones of other object object instances from the same class

## Local (automatic) variable

* Declared within a method body
* Visible only within the method body
* Come to existence only when the method gets executed

# Variables, scope

Class (static variables)

* In scope from declaration until program ends

Instance variables

* In scopoe from declaration unitl object garbage collected

Local variables

* In scope from declaration to end of block
* Set of braces define new block of code
* Each block of code has its own scope
* Blocks can contain other blocks
* The smaller contained blocks can reference veriables defind in the larger scoped blocks, but not vice versa.

# Methods

## methods, 2 types

* class (static) methods
* instance (non-static) methods

## method, class (static) methods

* Object instance does not have to be created
* Calling syntax [ClassName].[methodName]
* When create static methods?
* - When the logic and state does not involve specific object instance
* - Computation method, for example
* - Computer.add(int x, int y); // add(..) is a static method
* - When the logic is a convenience without creating an object instance
* - Integer.parseInt(“34”); // parseInt(..) is a static method

## method, instance (non-static) methods

* Can be called only through an object instance
* Can be called only after object instance is created
* Calling syntax [NameOfObjet].[methoName]

## accessor (getter) Methods

* Used to get (retrieve) data (in the form of object or primitive) from our class variables (instance/static).
* Written as: get<NameOfInstanceVariable>

## mutator (setter) Methods

* Used to write or change values of a variable
* Written as: set<NameOfInstanceVariable>

## overloading methods

* What is Method overloading?
* - Methods with the same names but different number of arguments or different types of arguments can coexist in a single class
* Why Method overloading?
* - To allow different implementations (but similar behavior) with a same method name but with different number of parameters or different types of parameters
* Always remember that overloaded methods have the following properties:
* - The same method name
* - Different number of parameters or different types of parameters
* - Return types can be different or the same, however

## overriding methods

* If a child class needs to have a different implementation (meaning different behavior) of a certain instance method from that of the parent class, override that instance method in the child class
* The overriding method has the same method signature (same name, number and type of parameters) as the method it overrides
* The overriding method can also return a subtype of the type returned by the overridden method
* Only applies to instance methods - can not override static methods

## contructor methods

* It is a special method where initialization of the newly created object is done
* Differences from a regular method
* - Constructors have the same name as the class
* - Constructors does not have any return value
* - public void Chick() {} // NOT A CONSTRUCTOR
* - You cannot call a constructor directly, it gets called indirectly when object gets instantiated
* Using the new keyword results in the class constructor getting invoked automatically
* NOTE: The new operator is not required with Java String literal object bacuse strings in Java are optimized.
* When multiple constructors with different parametes, constructor overloading

## default constructor (method)

* Is the constructor with no arguments.
* Also called no-arg constructor
* If the class does not specify any constructors, then a default constructor gets created automatically by the compiler

## constructor calling chain (default)

* A subclass constructor invokes the constructor of the super class implicitly (automatically).
* This is repeated for each super class before sub-class's constructor method is invoked.
* This is fhe default constructor chain (**not passing argumants**)

## constructor calling chain (super)

* A subclass can also explicitly call a constructor of its immediate super class by calling super(..) constructor call.
* Useful when the constructor of the subclass wants to call a constructor of a parent **passing arguments**
* This is in replacement of the default behavior of the automatic constructor call chaining - if super(..) is not called, the default (no-arg) constructor of the parent will be called
* A super(..) constructor call in the constructor of a subclass will result in the execution of relevant constructor from the super class, based on the arguments passed
* Constraints

- The super(..) call must occur as the first statement in a constructor

- The super(..) call can only be used in a constructor (not in ordinary methods)

## “this(..)” constructor call

* Constructor calls can be chained, meaning, you can call another constructor from inside a constructor
* - Use this(..) call for that purpose
* There are a few things to remember when using the this(..) constructor call:
* - It can only be used in a constructor
* - It must occur at the first statement in a constructor

## What is the difference between super() and this()?

* super is used to access methods of the base class while this is used to access methods of the current class.
* super() is used to call super class constructor, whereas this() used to call constructors in the same class, means to call parameterized constructors.

## “this” reference

* Refers to current object instance itself
* Used to access the instance variables
* To use the this reference, we type, this.<nameOfTheInstanceVariable>
* You can only use the this reference for instance variables and NOT static variables because this refers to an object instance
* The this reference is assumed when you call a method from the same object

## method coding guidelines

* Method names should start with a small letter
* Method names should sound like a verb
* Provide documentation before the declaration of the method. You can use Javadocs style for this.
* The instance variables of a class should normally be declared **private**, and the class will just provide **public** accessor (getter) and mutator (setter) methods to these variables

## access modifiers

* There are four different types of member access modifiers
* - public (Least restrictive)
* - protected
* - default (no access modifier is specified - package-private)
* - private (Most restrictive)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | private | default/package | protected | public |
| Same class | Yes | Yes | Yes | Yes |
| Same package |  | Yes | Yes | Yes |
| Different package  (subclass) |  |  | Yes | Yes |
| Different package  (no-subclass) |  |  |  | Yes |

## access modifiers, public

* Specifies that class members (variables or methods) are **accessible to anyone**, inside and outside the class and outside of the package to which the class belongs

## access modifiers, protected

* Specifies that the class members are accessible only to methods in **that class and the subclasses** of the class.
* The subclass can be in different packages, however

## access modifiers, default

* Specifies that only classes in the **same package** can have access to the class' variables and methods
* Sometimes called “Package-private”

## access modifiers, prinate

* Specifies that the class members are only accessible **within the class**

# Type Casting (of objects)

## What is “Type”?

* When an object instance is created from a class, we say the object instance is a “type” of the class and its super classes – Let's say we have created a Student object
* Student student1 = new Student();
* - student1 object instance is the type of Student or it is Student type
* - student1 object instance is also type of Person or it is Person type if Student is a child class of Person
* - student1 object instance is also type of Object because every class is subclass of Object class

## What is the Significance of "Type"?

* An object instance of a particular type can be used in any place where an instance of the type and its super type is called for
* Example:
* - Let's say student1 object instance is a “type” of JavaStudent, Student, and Person
* - Then the student1 object can be used in any place where object instance of the type of JavaStudent, Student, or Person is called for
* This enables polymorphism (We will cover polymorphism later in detail in another presentation)

## Implicit Type Casting (Very Important)

* An object instance of a subclass can be assigned to a variable (reference) of a parent class through implicit type casting – this is safe since an object instance of a subclass “is” also the type of the super class
* Example:
* - Let's assume Student class is a child class of Person class
* - Let's assume JavaStudent class is a child class of Student class

JavaStudent javaStudent = new JavaStudent();

Student student = javaStudent; // Implicit type casting

Person person = javaStudent; // Implicit type casting

Object object = javaStudent; // Implicit type casting

## Explicit Type Casting

* An object instance of a super class must be assigned to a variable (reference) of a child class through explicit type casting
* - Not doing it will result in a compile error since the type assignment is not safe
* - Compiler wants to make sure you know what you are doing
* Example
* Let's assume Student class is a child class of Person class

// This is safe because every Student is a type of Person

Person person1 = new Student();

// Explicit type casting required – because not every Person

// object is a Student type

Student student1 = (Student) person1;

## Runtime Type Mismatch Exception

* Even with explicit casting, you could still end up having a runtime error
* Example
* - Let's say Student class is a child class of Person class
* - Let's say Teacher class is also a child class of Person class

Person person1 = new Student();

Person person2 = new Teacher();

Student student1 = (Student) person1; // Explicit type casting

// No compile error, but runtime type mismatch exception

// because person2 refers to Teacher object

Student student2 = (Student) person2;

## Use instanceof Operator to Prevent Runtime Type Mismatch Error

* You can check the type of the object instance using instance of before the type casting
* Example

Person person1 = new Student();

Person person2 = new Teacher();

// Do the casting only when the type is verified

if (person2 instanceof Student) {

Student student2 = (Student) person2;

}

## Better to use Generics to detect Type Mismatch problem during Compile time

* Generics is introduced from Java SE 5
* Generics is designed to detect Type mismatch problem during compile time not during runtime

# Pass by value versus pass by reference

* Primitive Types are passed by value, no chane of original.
* Reference Types (objects) are are passed by reference, changes original.

# Error - "Cannot make a static reference to the non-static field myInstanceVariable"

public class ThreeVariablesTypes {

String myInstanceVariable = "instance variable";

public static void main(String[] args) {

System.out.println("myInstanceVariable = " + myInstanceVariable);

}

}

# Error - you cannot invoke instance method from a static method

static void mymethod3(){

String s1 = mymethod2("Sang Shin");

String s2 = this.mymethod2("Sang Shin");

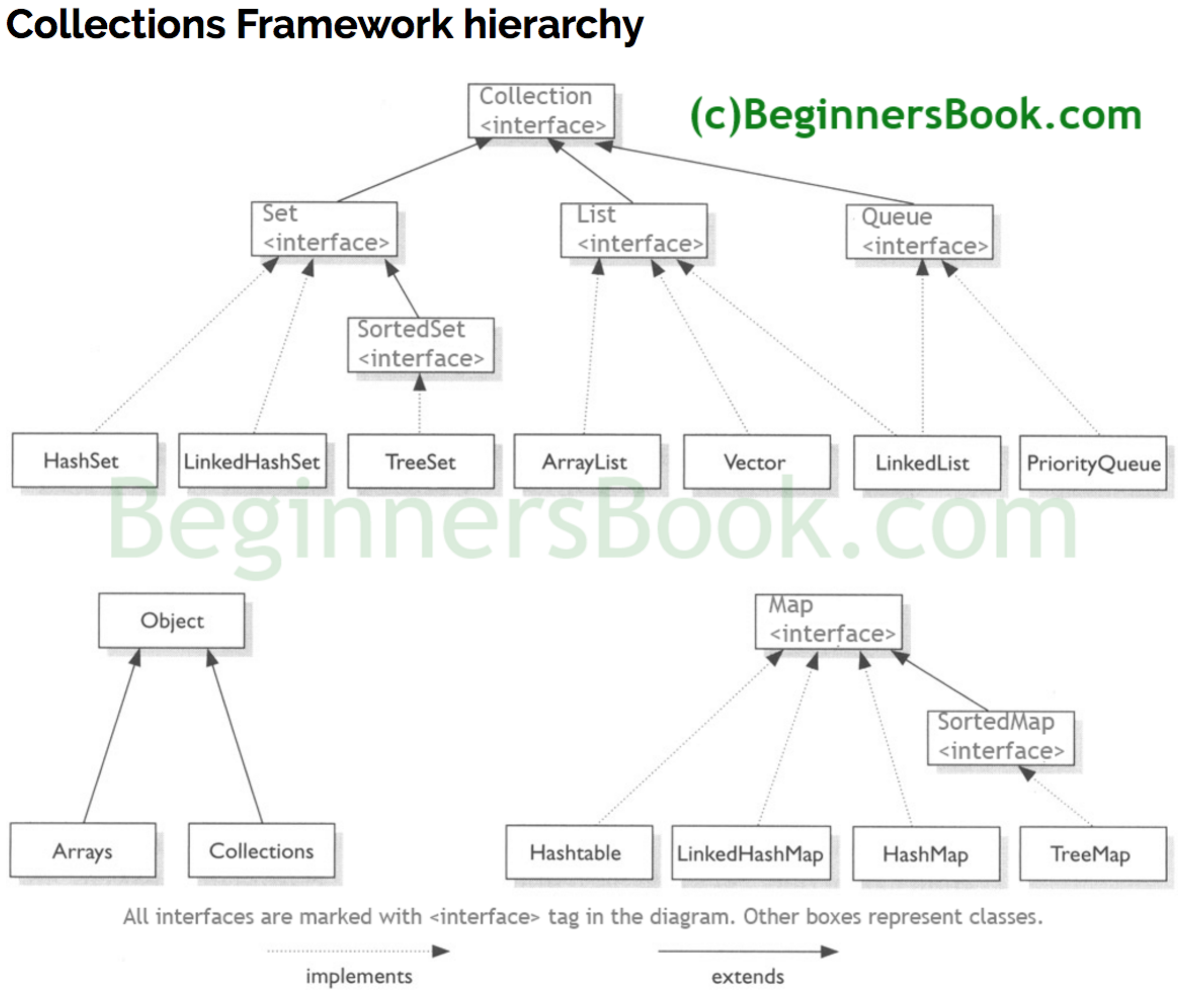
System.out.println("s1 = " + s1 + " s2 = " + s2);

}

# Hash

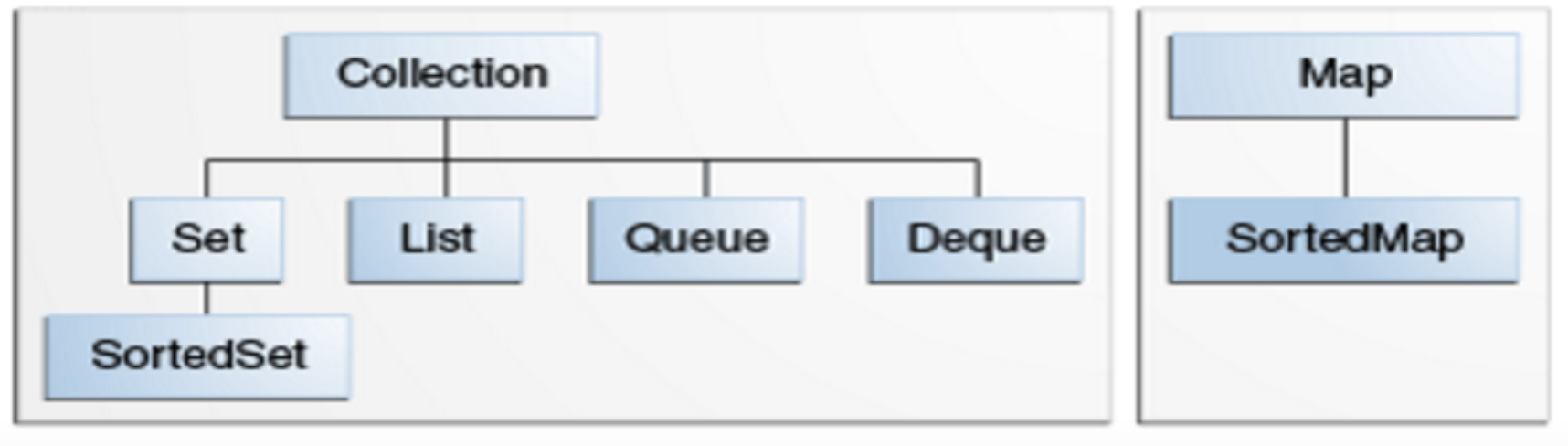
* A hash function is any function that can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

# Collection



## What is a Collection?

* A “collection” object — sometimes called a “container” — is simply an object that contains other objects
* Collections are used to store, retrieve, manipulate, and communicate aggregate data
* Java Collection Framework
  + Interfaces



* + Implementations

|  |  |  |
| --- | --- | --- |
| **Collection** |  |  |
| *Interface* | *Implementation* | *Historical* |
| Set | HashSet |  |
|  | TreeSet |  |
|  | LinkedHashSet |  |
| List | ArrayList | Vector |
|  | LinkedList | Stack |
| Queue |  |  |
| Dequeue |  |  |
|  |  |  |
| **Map** |  |  |
| *Interface* | *Implementation* | *Historical* |
| Map | HashMap | Hashtable |
|  | TreeMap | Properties |
|  | LinkedHashMap |  |

* + Algorithms



## Hierarchical relationships of the four basic interfaces

* The Collection interface is a group of objects, with duplicates allowed.
* The Set interface extends Collection but forbids duplicates.
* The List interface extends Collection, allows duplicates, and introduces positional indexing.
* The Map interface extends neither Set nor Collection.
* All operations are unsynchronized with the new classes.

## Iterator

* An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  | - un-synchronized  - can be made synchronized | - always synchronized |
|  | **Collection** |  |  |
|  | *Interface* | *Implementation* | *Historical* |
| - forbids duplicates | **Set** | **HashSet** - Unordered collection |  |
|  |  | **TreeSet** - A sorted set |  |
|  |  | **LinkedHashSet** - A set that remembers the order in which elements were inserted. |  |
| - allows duplicates  - indexed | List | **ArrayList** - An indexed sequence that grows and shrinks dynamically | Vector |
|  |  | **LinkedList** - An ordered sequence that allows efficient insertions and removal at any location | Stack |
|  | Queue | **LinkedList** - Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner. |  |
|  | Dequeue | **ArrayDeque** - Resizable-array implementation of the Deque interface. Array deques have no capacity restrictions; they grow as necessary to support usage. Null elements are prohibited.  **LinkedList** -  implement the Deque interface. |  |
|  |  |  |  |
|  | **Map** |  |  |
|  | *Interface* | *Implementation* | *Historical* |
|  | Map | **HashMap** - A data structure that stores key/value associations | Hashtable |
|  |  | **TreeMap** - A map in which the keys are sorted | Properties |
|  |  | **LinkedHashMap** - A map that remembers the order in which entries were added |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  | - un-synchronized  - can be made synchronized | - always synchronized |
|  | **Collection** |  |  |
|  | *Interface* | *Implementation* | *Historical* |
| - allows duplicates  - indexed | **List** | **ArrayList** - An indexed sequence that grows and shrinks dynamically |  |
|  |  |  | **Vector -** |
|  |  |  | **Stack -** |
|  | **List + Queue** | **LinkedList** -  • An ordered sequence that allows efficient insertions and removal at any location  • Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner. |  |
| - first in first out  - not ordered | **Queue** | **Priority Queue -** |  |
| - element insertion and removal at both ends; double ended queue (deque) | **Queue + Dequeue** | **ArrayDeque** - Resizable-array implementation of the Deque interface. Array deques have no capacity restrictions; they grow as necessary to support usage. Forbids null elements. |  |
| - forbids duplicates | **Set** | **HashSet** - Unordered collection |  |
|  |  | **LinkedHashSet** - A set that remembers the order in which elements were inserted. |  |
|  | **Set+SortedSet** | **TreeSet** - A sorted set |  |
|  |  |  |  |
|  | **Map** |  |  |
|  | *Interface* | *Implementation* | *Historical* |
|  | Map | **HashMap** - A data structure that stores key/value associations | **Hashtable** - |
|  |  | **TreeMap** - A map in which the keys are sorted | **Properties -** |
|  |  | **LinkedHashMap** - A map that remembers the order in which entries were added |  |

## Java Collection Interfaces & Implementations

ArrayList An indexed sequence that grows and shrinks dynamically

LinkedList An ordered sequence that allows efficient insertions and removal at any location

**ArrayDeque** A double-ended queue that is implemented as a circular array

HashSet An unordered collection that rejects duplicates

TreeSet A sorted set

EnumSet A set of enumerated type values

LinkedHashSet A set that remembers the order in which elements were inserted

PriorityQueue A collection that allows efficient removal of the smallest element

HashMap A data structure that stores key/value associations

TreeMap A map in which the keys are sorted

EnumMap A map in which the keys belong to an enumerated type

LinkedHashMap A map that remembers the order in which entries were added

WeakHashMap A map with values that can be reclaimed by the garbage collector if they are not used

elsewhere

IdentityHashMap A map with keys that are compared by ==, not equals

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Interface Collection<E>** | **Collection** | **Set** | **List** | **Map** |
| **// Basic operations**  int size();  boolean isEmpty();  boolean contains(Object element);  boolean add(E element);  boolean remove(Object element);  Iterator<E> iterator();  boolean hasNext();  Object next();  void remove();  **// Bulk operations**  boolean containsAll(Collection<?> c);  boolean addAll(Collection<? extends E> c);  boolean removeAll(Collection<?> c);  boolean retainAll(Collection<?> c);  void clear();  **// Array operations**  Object[] toArray();  <T> T[] toArray(T[] a); | X | X | X | X |
| **// Positional access**  E get(int index);  E set(int index, E element);  boolean add(E element);  void add(int index, E element);  E remove(int index);  boolean addAll(int index, Collection<? extends E> c);  **// Search**  int indexOf(Object o);  int lastIndexOf(Object o);  **// Iteration**  ListIterator<E> listIterator();  ListIterator<E> listIterator(int index);  **// Range-view**  List<E> subList(int from, int |  |  | X |  |
| **// Map Basic operations**  V put(K key, V value);  V get(Object key);  V remove(Object key);  boolean containsKey(Object key);  boolean containsValue(Object value);  int size();  boolean isEmpty();  **// Bulk operations**  void putAll(Map<? extends K, ? extends V> m);  void clear();  **// Collection Views**  public Set<K> keySet();  public Collection<V> values();  public Set<Map.Entry<K,V>> entrySet();  **// Interface for entrySet elements**  public interface Entry {  K getKey();  V getValue();  V setValue(V value); |  |  |  | X |

**Class Collections - Utility Functions (Algorithms)**

|  |
| --- |
| **// Empty set**  Collections.emptySet(); |
| **// Sorting**  Collections.sort(l);  Comparator comp = MyComparators.stringComparator();  Collections.sort(u2, comp); |
| **// Shuffling**  Collections.shuffle(l); |
| **// Routine data manipulation**  reverse  fill  copy  swap  addAll |
| **// Searching**  Collections.binarySearch(l, "Boston"); |
| **// Composition** |
| **// Find extreme values** |
|  |
| **// Misc. Functions**  Collections.frequency(myCollection, myElement)  Collections.disjoint(myCollection1, myCollection2) |
|  |

## Characteristics of a data structure

* Can it contain duplicates?
* How much effort does it take to search for an element?
* How much effort does it take to add/remove an elment?

## Array

* Defined as a type suffixed with square brackets.
* When defining, must provide size.
* Can access the element of an array directly by its index value.
* To make larger, must copy old array into new larger array (i.e. using System.arraycopy()).

## List

* Lists are sequential, orfered collection of values of a certain type.
* Different than arrays in that they are unbounded (don't have to specify size before use).
* An ordered collection (sometimes called a sequence)
* Lists can contain duplicate elements (as opposed to Set which does not allow duplicate elements)
* The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position)

### ArrayList

* **Duplicate values**
* **Random access**; offers constant-time positional access. Using get is instant.
* **Use when lots of random access.**
* As elements are added to an ArrayList, its capacity grows automatically.
* As elementa are removed, size doesn't automatically get smaller.
* Adding elemnts at beginning or middle is expensve because subsequent elemenets must be shifted.
* Each ArrayList instance has a capacity, ensureCapacity.
* Not synchronized. Can use Collections.synchronizedList(…)
* Resizable-array implementation of the List interface

### LinkedList

* **Duplicate values**
* **Sequential access**; not a good choice for random access; has to start from begining of list. If you want to get an element by its index, you need to traverse the length of the list, keeping count until you reach the given index.
* **Use when lots of insertions and removals.**
* As elementa are removed, size gets smaller.
* Linked list implementation of the List interface.
* LinkedList is good selection when a list oscillates between many and a few elements.
* Offers constant-time insertions or removals
* Not synchronized. Can use Collections.synchronizedList(…)
* If you want to get an element by its index, you need to traverse the length of the list, keeping count until you reach the given index.

## Set

* Represents a collection that cannot contain duplicate elements

### HashSet

* **Unique values**
* **Random order** (order is not guaranteed)
* HashSet is much faster than TreeSet
* Offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets.
* Mostly commonly used Set implementation
* HashSet stores the elements by using a mechanism called **hashing.**
* Caveats: Can specify initial capacity.
* Not synchronized. Can use Collections.synchronizedXxxx (Xxxx)

### TreeSet

* **Unique values**
* **Natural order** (order based upon natural ording of particular type)
* TreeSet is much slower than HashSet.
* Not synchronized. Can use Collections.synchronizedXxxx (Xxxx)

### LinkedHashSet

* **Unique values**
* **Insertion order** (order according to insertion)
* Implemented as a hash table with a linked list running through it
* Runs nearly as fast as HashSet.
* Spares its clients from the unspecified, generally chaotic ordering provided by HashSet without incurring the increased cost associated with TreeSet.
* Not synchronized. Can use Collections.synchronizedXxxx (Xxxx)

## Queue

* A Java interface that represents a “first in, first out” data structure.
* The interface has the methods add, to add a new element, remove to remove the oldest element, and peek, which returns the oldest element, but does not remove it from the data structure.

### PriorityQueue

### ArrayDeque

* Deque (pronounced “deck”) Is an extension of Queue, and allows addition and removal from either end of the data structure.
* The LinkedList class implements the Queue interface.

## Map

* A map, sometimes called a hash, associative array or dictionary, is a key-value store.
* Elements within the data structure can be queried by the key, which will return the associated value.
* Handles key/value pairs
* A Map cannot contain duplicate keys - value can be duplicated.
* V put(K key, V value);
* V get(Object key);

### HashMap

* **Unique key/value pairs**
* **Random order**
* Use it you want maximum speed and don't care about iteration order
* Most commonly used Map implementation
* Overrides existing entry if new entry has same key.

Integer ONE = new Integer(1);

Integer freq = (Integer) m.get(names[i]);

m.put(names[i], (freq==**null** ? ***ONE*** :

**new** Integer(freq.intValue() + 1)));

### TreeMap

* **Unique key/value pairs**
* **Natrual order**
* Implements SortedMap interface
* Use it when you need key-ordered Collection-view iteration.
* Use TreeMap when you want to sort the key-value pairs.

### LinkedHashMap

* **Unique key/value pairs**
* **Inserstion order**
* Use if you want near-HashMap performance and insertion-order iteration

### SortedMap

* A Map that maintains its mappings in ascending key order
* This is the Map analog of SortedSet (i.e. TreeSet)
* Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories

### Hashtable

* Overrides existing entry if new entry has same key.

### Properties

## Iterator

* Allows cycle through the elements in a collection
* Is an object that implements either the Iterator or the ListIterator interface
* Iterator enables you to cycle through a collection, obtaining or removing elements
* ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of element

# Stream I/O

## serialization

* Is the process of writing an Object into file along with its attributes and content. It internally converts the object in stream of bytes.

## de-Serialization

* Is the process of reading the Object and it’s properties from a file along with the Object’s content.

## I/O Class Hierarchy

<http://chortle.ccsu.edu/java5/notes/chap82/ch82_7.html>

* Object
  + Byte Streams are 8-bit bytes
    - InputStream
      * FileInputStream
        + BufferedInputStream

DataInputStream

double price = in.readDouble();

int unit = in.readInt();

String desc = in.readUTF();

* + - OutputStream
      * FileOutputStream
        + BufferedOutputStream

DataOutputStream

out.writeDouble(prices[i]);

out.writeInt(units[i]);

out.writeUTF(descs[i]);

* + Charater Streams are 16-bit Unicode, are wrapers for byte streams
    - Reader
      * FileReader
      * BufferedReader
    - Writer
      * FileWriter
      * BufferedWriter
  + Data Streams
    - DataInpuut interface
    - DataOutput interface
  + Object Stream
    - FileInputStream
      * ObjectInputStream
    - FileOutputStream
      * ObjectOutputStream