JAVA

\*USING IntelliJ

// Single line comment

/\* …………………………  
 Multiple lines comment  
\*/

#Right click on project > Open module setting > Java 8   
#File > project structure > Program language level – 8

# JAVA BASICS

## Variables

into number; //declaration  
number = 7; //assignment or definition

A variable can be reassigned but not re-declared. Variable names start with letters and should only include letters, numbers, and underscores.

\n //new line character

## Input

import java.util.Scanner; //imports scanner class

Scanner in = new Scanner(System.in); //declare scanner, scanner\_name is “in”  
System.out.println(“Enter a radius: “); //ask for input  
double radius = in.nextDouble(); //store input  
in.close(); //close scanner

scanner\_name.nextInt() – to get an into  
scanner\_name.next() – get a string  
scanner\_name.nextLine() – to get a line from user

NOTE: Use scanner.nextLine() after using scanner.nextInt() or scanner.nextDouble() in order to remove the enter key from the input buffer

scanner.hasNextInt(); //it returns true if the input in scanner is int type, otherwise false

**>>> Program 1\_Hello\_world <<<**

**>>> Program 3.2\_User\_input <<<**

## Output

\r in System.out.println(“Enter: “); takes the cursor to next line. It is similar to \n in System.out.printf(“Enter: \n”);

## Data Types

Key data types:

* byte - from -128 to 127 //width – 1 byte
* short – from -32767 to 32767 //width – 2 bytes
* int – integer value, 4 bytes //Integer.BYTES returns 4 bytes
* long – integer value, 8 bytes
* float – whole number, 4 bytes
* double – whole number, 8 bytes //default whole number data type for Java
* char – single character, 2 bytes
* boolean – true or false, 8 bytes

char a = ‘\u00A9’ //to represent Unicode character, eg 00A9 is for copyright symbol character  
long num1 = 123L  
float num2 = 1.1f  
double num3 = 2.22d

//use underscores to make up large numbers  
long number1 = 123\_323\_567\_890L

NOTE: Java automatically converts integers to int on assignments  
byte new\_byte = 12;  
byte byte\_2 = new\_byte/2; //this will give an error  
byte byte\_2 = (byte) (new\_byte/2); //casting to byte

### String

String my\_string = “a string example”;  
String string2 = “String – “;

String concatenation

String new\_string = string2 + my\_string + “ – \u00A9 2015”;  
//new\_string will be “String – a string example - © 2015”

NOTE: if you add an int or any other number data type to a string, Java will convert the int or any other number data type to string before adding them as strings

Operators

Assignment =  
Arithmetic + - \* / %  
Unary ++ -- !  
Equality/Relational == != > < >= <=  
Conditional && || ?:  
Bitwise >> << & | ^

Ternary Operator (shorthand for if-then-else)

boolean a = true;  
int b = a? 2:3; // here b will be 2 if a is true and b will be 3 if a is false (a must be boolean type)

KEYWORDS

Keywords are one of 53 reserved words  
List - <https://en.wikipedia.org/wiki/List_of_Java_keywords>

## Expression And Statement

Expression – variables + operator + values  
Statement – Expression + data\_type + ;

int number = (100\*123); //Statement  
number = (100\*123) //Expression

System.out.println(“This part is expression”); //statement

NOTE: The following code is a valid Java statement

System.out.println(“This is”+  
 “still a valid“+  
 “Java statement”);

Whitespaces – space, tab, enter

Use whitespaces and indents to format the code for better appearance.

## Control Flow Statements

if(condition)  
 statement;

if(condition)  
{  
 statement;  
}  
else if(condition2)  
{  
 statement2;  
}  
else  
{  
 statement3;  
}

VARIABLE SCOPE (Code Block)  
The region from the variable declaration to the corresponding } where it is alive refers to variable scope.

METHOD in main class

In Java, you can declare and define functions after the main function.

public static return\_type method\_name(arg\_type arg\_name, … )  
{  
 //method implementation  
 return return\_value;  
}

**>>> Program 2.1\_Circle\_area <<<**

**>>> Program 2.2\_Leap\_year <<<**

**INTELLIJ tricks**psvm +tab //creates public static void main(String[] args) { }   
sout +tab //create System.out.println(“ “);

## Method Overloading

Creating new implementation for already existing functions, with the same name but with different arguments (number or type) – Method overloading

Method overloading does not affect return type. In other words, the return type of the new function should be same as that of overloaded function.

CONSTANTS for a class

private static final c\_type c\_name = c\_value; //final makes it constant

STATIC

Static methods can be accessed without creating an instance of the class. The constants that are accessed by such static methods need to be static as well.

**>>> Program 3.1\_Method\_overloading <<<**

CONTROL FLOW STATEMENTS continued…

NOTE:  
- break is used to get out from a loop (for,switch, or while)  
- continue is used to jump to next iteration of the loop

SWITCH Statement

switch(switch\_vaue)  
{  
 case 1:   
 …………………  
 break;  
 case 2:  
 …………………  
 break;  
 …  
 default:  
 ………………..  
 break;  
}

//when multiple cases have same statements to execute  
case 1: case 2: case 3:  
 ……………………………  
 break;

FOR LOOP

for(initialization; condition; increment)  
{  
 …………  
}

//declaring i within for loop is allowed in Java

NOTE: To print a printf type line in a println statement:  
System.out.println(“My lucky number is” + String.format(“%d”,10));

WHILE LOOP

It is used when we do not know the number of iterations.

initialization;  
while(condition)  
{  
 ……………  
 increment;  
}

DO WHILE LOOP

It is used when the code needs to run once before checking the condition.

initialization  
do  
{  
 ………..  
 increment;  
} while(condition)

## String Operations

* Length of a string  
  int length = var\_name.length();
* String concatenation  
  String name = f\_name + l\_name;
* Characters  
  char var\_char = string\_name.charAt(int\_position);
* Substring  
  String var\_name = string\_name.substring(start\_index,end\_index);
* Split  
  String[] data = my\_string.split(delimiters);  
  String[] data = my\_string.split(delimiters, int limit); //limit is the max number of elements  
  //single delimiter = “x”  
  //multiple delimiters = “[xyz]”

Example:   
String name = “Vader”;  
String substring = name.substring(0,2) //this will be “Va” and not “Vad”  
String substring = name.substring(2) //this will be “der”, from 2 to the end

* Converting numbers to String  
  String a = Integer.toString(int\_val);  
  String b = Double.toString(double\_val);
* Converting character to String  
  String c = Character.toString(char\_val);
* Changing to Uppercase and lowercase  
  String y\_string = x\_string.toUpperCase();  
  String z\_string = x\_string.toLowerCase();
* Comparison of Strings

a\_string.equals(b\_string) //return boolean  
a\_string.compareTo(b\_string)  
// returns 0 if strings are same  
// returns negative value if a\_string comes before the b\_string in Java  
// returns positive value if a\_string comes after the b\_string

NOTE: Uppercase letters come before lowercase letters in Java

PARSING VALUES FROM STRING

//parsing string to a number  
int number = Integer.parseInt(a\_string);  
double number2 = Double.parseDouble(b\_string);

//if the parsing fails, it will throw an error (java.lang.NumberFormatException)

STRING FORMAT

//TODO

# OBJECT ORIENTED PROGRAMMING

OBJECT – Real world objects with two characteristics: state and behavior

CLASS – a template for creating an object

Benefits of using class:  
1. Restrict access to the data  
2. Create your own data type

Access modifiers – public, private, protected

//class definition   
public class class\_name  
{  
 //variables  
 //constructor  
 //methods  
}

//creating object in main (using default empty class constructor)  
 Class\_name object\_name = new Class\_name();

Variables are initialized to NULL by default when a class is initialized without any parameter.

## **Getters And Setters**

Usually, the variables of a class are private. In order to access those variables, we need to use getters and setters methods. This is done so that only valid values could be set to the variables.

public void set\_variable\_x(int x)  
{  
 if(x > 0)  
 this.x = x;  
 else   
 this.x = 0;   
}

public int get\_variable\_x()  
{  
 return this.x;  
}

**INTELLIJ tricks:**Getters and setters as well as other general functions can be automically generated using IntelliJ   
( Code > Generate OR Alt+Insert ) > Choose function

## Constructors

Constructor is a special method that initializes the variables of the object. A class can have multiple constructors, each for different cases. A constructor name must be same as class name.

//calling another constructor form a constructor

public class\_name()  
{  
 this(11,”hello”); // when using “this()”, this line must be the first line in constructor  
 system.out.println(“Empty constructor”);  
}  
public class\_name(int x, String y)  
{  
 this.x = x; //it is better to set variables directly rather than using setter function here  
 this.y = y;  
}

**NOTE**: When coding multiple constructors for a class, use constructor chaining. That means, make a constructor initialize all the fields and call this base constructor in other constructors.

**>>> Program 4.1\_Getters\_and\_Setters <<<**

* “this” keyword is used to access current class members (variables and methods) – usually required when we have arguments with same name as class member
* “super” keyword is used to access parent class members (variables and methods)

## Inheritance

Inheritance is used to inherit the state and behavior of a class to another class.

Inheritance establishes IS-A relationship between a parent class and a child class. That means German shepherd (child class) is a dog (parent class). So, if we setup a function that has return type of parent class (dog), it can also return an object of type child class (german shepherd) and that returned child object can be assigned as parent class object in the callee function.

public class child\_class extends parent\_class   
{  
 private int child\_var1;  
 private String child\_var2;

public child\_class(int var0, int var1, String var2)  
 {  
 super(var0); //will set parent\_var0 to var0  
 this.child\_var1 = var1;  
 this.child\_var2 = var2;  
 }  
 }

**NOTE**: Every class created in java inherits from a default java class called Object.

### **Method Override**

A child class can access the parent methods to define its own behavior. When a parent method is rewritten in child class to make it unique to the child class, it is said to be method overriding.

@Override //symbol to represt method override in Java, helps compiler  
public parent\_method(int var1) //In child class  
{  
 child\_method(int var1); //to use different method when parent\_method is called for child object  
 //OR  
 super.method\_name(var1\*x); //remodelling variable before using parent’s method  
}

**>>> Program 4.2\_Inheritance <<<**

* this() call is used to call a constructor from another overloaded constructor in the same class
* super() call is used to call a parent constructor from a child class, a default super() call is always added automatically to a child class by Java
* These two calls must be the first line when used in a constructor
* These two calls cannot be made at the same time in a constructor

REFERENCE – In Java, you always have references to an object in memory. There is no way to access an object directly, everything is done using a reference. An object can have multiple reference and a class can have multiple instances.

|  |  |
| --- | --- |
| Method Overloading  OR  Compile-time Polymorphism | * Methods must have the same method name * Methods must have different parameters (type or number) * Methods may or may not have different return types and access modifiers |
| Method Overrding OR  Runtime Polymorphism | * Methods must have same name and same arguments * Return type can be a subclass of the return type in parent class * Cannot have a lower access modifier (private < protected < public) * Methods can be overriden only in child classes * Static methods, private methods, final methods, and constructors cannot be overriden |

|  |  |
| --- | --- |
| Static methods | Static methods can not access instance methods or instance variables of a class directly. Also, this method can be called without instantiating an object for a class. Class\_name.method\_name(); |
| Instance methods | To use an instance method, we have to instantiate the class first using the new keyword. These methods can access all instance and static methods and variables directly. |
| Static variables | Every instance of a class shares the same static variable. So any change made to static variable is visible to every instance of the class. |
| Instance variables | Every instance of a class has its own copy of an instance variable. |

## Composition

While inheritance exhibits IS-A relationship between child class and parent class, composition exhibits HAS-A relationship. Composition is when an object of a class is included as a state in another class. This does not need any inheritance, but both class files must be in the same package. Using composition, one class can be used to manage other classes.

**>>> Program 4.3\_Composition <<<**

## Encapsulation

We use encapsulation to restrict access to certain object, variable, or method. We use different levels of access modifiers for object, variables, and methods: public, protected, and private.

**INTELLIJ tricks**When you have to change a name of a variable in a class, do the following for one of them and it will update all the copies of that name.  
select > right click > Refactor > rename

## Polymorphism

Polymorphism allows a method to behave differently at runtime depending on the object calling it (this usually needs inheritance in conjuction).

**INTELLIJ tricks**To create a similar class, we can copy a class:  
right click on class name > refactor > copy > //set new class name > OK

To move a class from external file to main class:  
right click on class name > refactor > move > make inner class of > main > OK

We can use getClass().getSimpleName() function to get class name as a string. getClass() is a method from Object class.

**>>> Program 4.4\_OOP\_review <<<**

# DATA STRUCTURES

## Array

import java.util.Arrays;  
  
**Declaration and definition**

data\_type[] my\_array = new data\_type[size];  
my\_array[0] = val0; //for loop can be used here  
my\_array[1] = val1;   
….  
OR  
data\_type[] my\_array = {val1, val2, …, valN};

**To get the size of an array**

int size = my\_array.length;

**Passing to and returning from a function**

public static data\_type[] my\_function(data\_type[] my\_array)   
 { return my\_array };  
Trying to access index that is out of range will result ArrayIndexOutOfBoundsException.

**NOTE:** When an array is declared, all the elements are initialized to default values: 0 for int, false for boolean, NULL for String, etc.

**To print an array in same line using single print statement**System.out.println(“Array = “ + Arrays.toString(my\_array));

|  |  |
| --- | --- |
| **Reference types** | These are data types which hold reference to other data types or objects  Example: Array, String  int[] array1 = new int[5]; int[] array2 = array1;  These two arrays will point to same set data. So changing one will change both |
| **Value types** | All primitive data types are value types. That means they hold value.  int a = 5;  int b = a;  Here a and b share same value. But changing one does not change another |

**NOTE:** int max = Integer.MAX\_VALUE; //this will give the max value possible for an integer data type

**>>> Program 5.1\_Arrays <<<**

## Array List

import.java.util.ArrayList;

**Declaration and definition**

ArrayList<class\_data\_type> my\_list = new ArrayList<data\_type>()  
my\_list.add(val1);  
my\_list.add(val2);  
…

ArrayList<class\_data\_type> my\_list = new ArrayList<>(Arrays.asList(val1, val2, …));

An array is a data type whereas an ArrayList is a class. So, we have to use class constructor and methods. The size of an array can not be changed directly after its declaration. The size of the ArrayList is dynamic.

|  |  |
| --- | --- |
| **Add an element at position pos** | my\_list.add(pos, val); |
| **Get the size of an ArrayList** | my\_list.size(); |
| **Access the elements** | my\_list.get(position); |
| **Change an element at position pos** | my\_list.set(pos, new\_val); |
| **Remove an element at position pos** | my\_list.remove(pos); |
| **Check if the list is empty** | boolean check = my\_list.isEmpty(); |
| **Check if an element is present** | boolean value = my\_list.contains(value); |
| **Get index of an element** | int position = my\_list.indexOf(value);  //returns -1 if the element is not present |
| **Copy an ArrayList to new ArrayList** | ArrayList<data\_type> new\_list = new ArrayList<data\_type>(); new\_list.addAll(my\_list)); OR ArrayList<data\_type> new\_list = new ArrayList<data\_type>(my\_list); |
| **Convert ArrayList to an array** | data\_type[] my\_arr = new data\_type[my\_list.size()]; my\_arr = my\_list.toArray(my\_arr); |
| **Convert an array to ArrayList** | my\_list.addAll(Arrays.asList(new Integer[]{1,2,3,4,5})); |
| **Print an ArrayList** | System.out.println(my\_list); |

**>>> Program 5.2\_ArrayList <<<**

## Autoboxing and Unboxing

The data\_type to be used in an arraylist must be a class. We cannot make an arraylist of primitive data type. But there exist corresponding class data type for each primitive data type. This process of converting primitive data type to class data type (which acts as a wrapper to the primitive data type) is refered as autoboxing. On the other hand, unboxing is when we convert the class data type to a primitive data type (removing the wrapper class).

For example, Integer is class data type for int, Double is class data type for double, etc.

**NOTE:** String is not a primitive data type. It is rather a class data type.

Integer my\_integer = new Integer(2);  
OR  
Integer my\_integer = Integer.valueOf(2); //Autoboxing  
int my\_int = my\_integer.intValue(); //Unboxing

However, Java does the autoboxing and unboxing automatically. So the following code executes same as above.

Integer my\_integer = 2;   
int my\_int = my\_integer;

FOR EACH LOOP

ArrayList<String> names = new ArrayList<String>();  
for(int i=0; i < names.size(); i++)  
 System.out.println(names.get(i));

OR  
ArrayList<String> names = new ArrayList<String>();  
for(String name: names)  
 System.out.println(name);

**>>> Program 5.3\_Autoboxing\_and\_unboxing <<<**

## Linked List

import.java.util.ArrayList;

For operations on large number of data, ArrayList would take up a lot of time. In that case, we prefer linked list, since operations such as adding and removing are faster using linked list.

**Declaration**LinkedList<String> my\_list = new LinkedList<String>();

LinkedList has same operations as an ArrayList.

|  |  |
| --- | --- |
| **To get first element** | my\_list.getFirst(); |

## Iterator

import.java.util.ArrayList;An iterator iterates through a given list.

Iterator<String> it = my\_list.iterator(); //it does not point to first item here  
while(it.hasNext()) //it points to first item after first it.hasNext()  
 System.out.println(it.next());

Capture.PNG

ListIterator provides more flexibility as it is bi-directional. But it only applies to list classes.

ListIterator<Integer> it = my\_list.listIterator();  
while(it.hasNext()) //ListIterator also has it.hasPrevious()  
{  
 if(val1 < it.next())   
 {  
 it.previos();  
 it.add(val1);  
 }

}

**NOTE:** it.remove() removes the last item returned by next() or previous(), and it can be called only once per next() or previous()

**>>> Program 5.4\_LinkedList <<<**

# ABSTRACTION

## Interfaces

Interface does not contain any code implementation and is rather used for abstraction.

Like a class, an interface can have methods and variables, but   
- the methods declared in interface are by default abstract (only method signature, no body)  
- all methods are public and abstract (can with implementation since Java 8 and can be private since Java 9)  
- all fields are public, static, and final

Interface acts as a blueprint for a class. A class that implements an interface must define all the methods present in the interface, otherwise it is considered as abstract class.

**Syntax:**public interface Interface\_name  
{  
 //declare constant fields and abstract methods  
 return\_type method(arg1, …);  
 data\_type var1;  
}

//implementation  
public class class\_name implements Interface\_name { }

**INTELLIJ tricks**To implement all the methods from an interface in a class:  
Alt+insert > Implement Methods > Select methods > OK

**NOTE:** An object can be **declared** as an interface type but it must be **defined** as a class type object.  
Example: Interface\_name my\_object = new class\_name(args, … );  
Here you will have to use class casting to access methods of the class that my\_object belongs to. On the positive side, you can hold different objects that have common interface in this way.

**NOTE:** A class can not inherit from multiple classes, but can implement multiple interfaces.

**>>> Program 6.1\_Interface <<<**

## Inner Classes (Nested Classes)

The class written within a class is called the nested class, and the class that holds the inner class is called the outer class.

**Syntax**public class Outer\_class()  
{  
 class Inner\_class() { }  
}

  
<source: tutorialspoint.com>

|  |  |
| --- | --- |
| Inner class | An object of outer class must be defined in order to declare an object for an inner class. The inner class can be either be public or private. The public inner class can be accessed from main as: Outer\_class.Inner\_class my\_obect = outer.new Inner\_class(args,…); |
| Method-local inner class | The scope of the inner class is limited within a method of the outer class. |
| Anonymous inner class | It has no name and thus needs to be defined and instantiated at the same time. |
| Static nested class | It is a static member of outer class. It can be accessed without instantiating the outer class. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. Outer\_class.Static\_nested my\_object = new Outer\_class.Static\_nested(args,…); |

**>>> Program 6.2\_Nested\_classes <<<**

## Abstract Class

Abstraction – Declaring what needs to be done but not how it is to be done

Abstraction can be achieved by using either an interface or an abstract class. An interface can extend another interface.

The purpose of an Abstract class is to provide a common definition of a base class that multiple derived classes can share.

* If a class is declared abstract, it cannot be instantiated.
* Abstract classes may or may not contain abstract methods, i.e., methods without body ( public abstract void get(); )
* If a class has at least one abstract method, then the class **must** be declared abstract.
* To use an abstract class, you have to inherit it from another class, provide implementations to all the abstract methods in it.

Interface is used primarily when unrelated classes are expected to implement the interface.

## Generics

We can make a method, class or interface that operates on a general data type. In other words, same declaration can accept multiple types of arguments.

**Generic method:**  
public static <E> return\_type foo(E my\_var)   
//<E> is not written if the method is a member of a generic class  
{  
 System.out.printf(“%s\n”,my\_var);  
 return E;  
}

**Generic class:**  
public class House<T> // T type can be any class-type and not primitive data types  
{  
 private T t;  
 public void House(T t)  
 {  
 this.t = t;  
 }  
 public T get()  
 {  
 return t;  
 }  
}

**In main:**  
foo(1); //prints 1  
foo(“ablaze”); //prints ablaze  
House<String> my\_house = new House<String>(“Green”);  
System.out.println(my\_house.get()); //prints Green

If no restriction is placed on T, any class-type can be set to the generic class. But if we can also set a upperbound on the type T:  
public class House<T extends upper\_class>   
{ … }

Here, any object of upper\_class or its child classes can only be the type for generic class House.

**Note:** A type in generic class can have multiple upperbounds (one class but multiple interfaces).  
public class House<T extends class\_A & interface\_B & interface\_C> { … }

Comparable InterfaceWhen a generic class implements a comparable interface, it has to implement compareTo() method. We can set that method to only allow comparing generic object with similar objects. We can also set desired field to be compare.   
public class House <T> implements Comparable<House<T> { … }

**>>> Program 7.1\_Generics <<<**

# NAMING CONVENTIONS & PACKAGES

|  |  |
| --- | --- |
| **CATEGORY** | **CONVENTIONS** |
| Packages | * lowercase, unique * internet domain name in reverse * if domain name contains invalid characters, start with number, or include Java keywords, it should be replaced/adjusted with underscore * Example: java.lang, java.io |
| Class name | * CamelCase * Example: ArrayList, LinkedList, String |
| Interface | * CamelCase |
| Method name | * mixedCase |
| Constants | * UPPER\_CASE |
| Variable name | * mixedCase * no underscores and start with lower case |
| Type parameters | * Single capital character * E – Element, K – Key, T – Type, V – Value |

## Packages

We can not import two packages that have classes with same name. If we need to access the class from second package, we can do:  
com.package.second.MyClass myClassObject;  
However, this needs to be done for every time we use the object of MyClass from second package.

Java object class is imported from the package java.lang, and this import is done automatically.

import java.util.\*  
//this imports all the classes in package java.util  
//this is different than  
import java.util.test.\*  
//here java.util.test is a different package from java.util

**NOTE:**You can use:  
com.example.package\_name;  
org.example.package\_name;  
for the packages you are not going to distribute.

**>>> Program 8.1\_Packages <<<**

## Scope & Access Modifiers

A class can access private variables from its inner class.

Top Level  
Classes, Enum, and Interfaces at top level must be public or package-private. Package-private is specified by not specifying (except interfaces where all methods and variables are private).

Member Level   
Public – can be accessed from anywhere  
Private – only visible to the class it is declared in  
Protected – visible in the same package or in subclasses anywhere

A static method can not call a non static method since non static method requires an instance of the class. But non static method can call static method.

Final keyword in a variable means that the variable can only be defined once either at the class definition or in the constructor. Final keyword in a class means that the class can not be changed or extended.

Obj instaceof myClass – returns true if obj is an instance of myClass (or its subclasses) otherwise it will return false.  
Obj1.getClass() == Obj2.getClass() – returns true only if obj1 and obj2 are objects of same class (not subclasses).

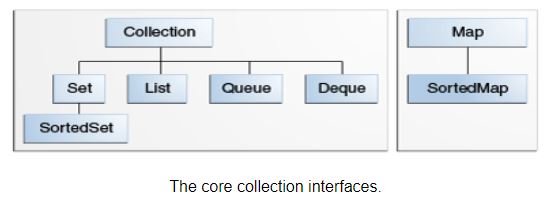
Static block

Static block is a static initializer that is executed when the class is loaded (before the execution of main). This block of code executes only once and has only one copy which is shared between the instances of the class.

# JAVA COLLECTIONS

## Collection frameworks

* Interfaces
* Implementations (Classes)
* Alogrithms

  
<src - docs.oracle.com/javase/tutorial/collections/interfaces/ >

**>>> Program 9.1\_CollectionBinarySearch <<<**

**Copying ArrayList**  
  
List<int> numbers = new ArrayList<>(prime); //prime is an arraylist of an object

This is a shallow copy. It means that if we change a data in one of the array list, the data is changed in the both. In other words, both array lists point to same set of object. To make a deep copy of an arraylist, we have to use implement clone method that makes a copy of the element object and returns new reference.

Collections.reverse(prime); //reverses the order of the data in prime numbers

Here prime will still point to same data set (object) as numbers but will have them in reversed order. It’s because the order is the property of the array list which can be changed separately but changing the properties of one object will change the properties in another object as well.

### Some Collections Algorithms

|  |  |
| --- | --- |
| **Method** | **Description** |
| static int binarySearch(List list, Object value, Comparator c) | Searches for value in the list ordered according to **c**. Returns the position of value in list, or -1 if value is not found (c can be null)  Note – comparator is not required for built-in objects |
| static void copy(List dest, List src) | Copies the elements of src to dest,  does a deep copy and generally NOT USED since it requires dest to already have object in it to be replaced |
| static Object max(Collection c, Comparator comp) | Returns the maximum element in c as determined by the comp |
| static Object min(Collection c, Comparator comp) | Returns the minimum element in c as determined by the comp |
| static boolean replaceAll(List list, Object old, Object new) | Replaces all occurrences of old with new in the list. Returns true if at least one replacement occurred. Returns false, otherwise. |
| static void reverse(List list) | Reverses the sequence in list |
| static void shuffle(List list, Random r)  or  static void shuffle(List list) | Shuffles (i.e., randomizes) the elements in the list by using r as a source of random numbers. |
| static void sort(List list, comparator comp)  or  static void sort(List list) | Sorts the elements of list as determined by comp or by their natural ordering as specified  *Stable* |
| static void swap(List list, int idx1, int idx2) | Excha nges the elements in the list at the indices specified by idx1 and idx2 |

Note – For methods that require comparing the objects in the list, the object must have implemented comparable<T> interface (overriden compareTo method)

Comparator

It defines a way to sort a list of object based on a particular value or property. Alternatively, a class can implement comparator<T> and override compare method.

//defined in a class  
static final Comparator<T> MY\_ORDER;

static  
{  
 MY\_ORDER = new Comparator<T>()  
 {  
 @Override  
 public int compare(T obj1, T obj2)  
 {  
 if(obj1 … obj2) return -1;   
 if(obj1 … obj2) return 1;   
 if(obj1 … obj2) return 0;  
 }  
 };  
}

## Maps

java-map-hierarchy.png<source - https://www.javatpoint.com/java-map>

* A Map cannot contain duplicate keys and each key can map to at most one value.
* **A Map doesn't allow duplicate keys, but you can have duplicate values.**
* HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.
* A Map can't be traversed, so you need to convert it into Set using keySet() or entrySet() method in order to traverse it.
* The order of a map depends on specific implementations, e.g TreeMap and LinkedHashMap have predictable order, while HashMap does not.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public Object put(Object key, Object value) | Insert an entry in the map and returns the previous value object or null  //if an object already exists with the same key, it is replaced with new one [contrary to Set] |
| public void putAll(Map map) | Insert the specified map in this map |
| public Object replace(Object key, Object newValue)  public Object replace(Object key, Object oldValue, Object newValue) | Replace the old value with new value  To avoid changing wrong pair |
| public Object remove(Object key)  public boolean remove(Object key, Object value) | Delete an entry for the specified key  Delete the key-value pair and return true if successful |
| public Object get(Object key) | Returns the value for the specified key |
| Public Object getOrDefault(Object key, Object default) | Returns the value for the specified key, but if not found, returns the default value |
| void clear() | Clears the map (removes all elements) |
| Public Collection<T> values() | Returns a collection of values of the elements in the map |
| public boolean containsKey(Object key) | To check if the specified key is in the map |
| public Set keySet() | Returns the Set view containing all the keys |
| public Set entrySet() | Returns the Set view containing all the keys and values |
| boolean isEmpty() | Check if the map is empty |
| int size() | Returns the size of the map |
| Collections.unmodifiableMap(Map map) | Returns a map that cannot be modified  However, the elements in the map can be modified unless the elements are immutable objects |

//Example – create and print a map

Map<String, String> myMap = new HashMap<>();  
if(!myMap.containsKey(“Java”)  
 myMap.put(“Java”, “I like Java”);  
for(Object key: myMap.keySet())   
{  
 System.out.println(key + “:” + myMap.get(key));  
}

**Alternative way to iterate over a map**for(Map.Entry<String, String> item : myMap.entrySet())  
{  
 System.out.println(item.getKey() + “ – “ + item.getValue());  
}

**>>> Program 9.2\_Adventure <<<**

Immutable Classes

Immutable class means that once its object is created, we cannot change its content.   
Examples – all the wrapper classes (Integer, Boolean, Bytem Short) and String class

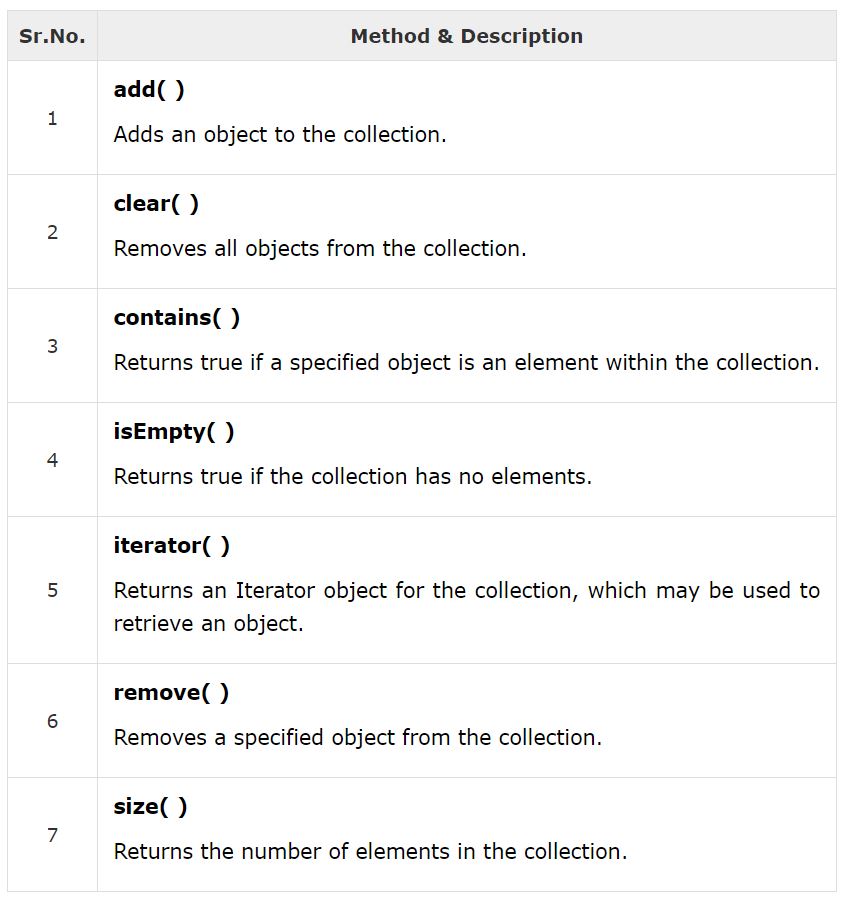
Requirements:

1. The class must be final (so a subclass cannot be created)
2. The instance variables of the class are final and private
3. There is no setter method

Note: Be careful not to have external variable pointing to the instance variable of the immutable class.

## Sets

* An interface which extends Collection
* **Duplicate values are not allowed**
* Unordered collection of objects [Collections.sort works only on List]
* Implementations – HashSet, LinkedHashSet *or TreeSet (sorted representation)*

  
<source - https://www.tutorialspoint.com/java/java\_set\_interface.htm>

**Note**: *No get method, meaning it is not possible to directly access an element and rather have to use an iterator.*

Let a and b be two HashSet

* **Union**: a.addAll(b) or b.addAll(a) – transforms a into the union of a and b

Example  
Set<Integer> union = new HashSet<Integer>(a);  
union.addAll(b);

* **Intersection**: a.retainAll(b) or b.retainAll(a)
* **Difference (a – b)**: a.removeAll(b)
* **Subset**: a.containsAll(b) – returns true if b is a subset of a
* **All of above methods return boolean value**

In a hashset, an object is first hashed to a bucket using hashCode() and only compared with the objects present in that particular bucket using equals(). This results in higher efficiency.

## equals() and hashCode()

If you are using your own object as an element in the set or key in the map, you should override the equals() and hashCode() methods. This is needed to ensure the uniqueness property of keys in the map and elements in the set.

equals() method is only called if two objects have same hashCode() in order to check if they are equal. Two objects with different hashCode are never equal. And two objects with equal hashCode might not be equal. Also, if two objects are true on equals() then they must have same hashCode() and hashed to same bucket.

By default, the equals() method of the object class compares the references of the given object. The equals method implements an equivalence relation on non-null object references (reflexive, symmetric, transitive, and consistent). The overridden equals and hashCode methods must have the following signatures:

public boolean equals(Object obj)

{  
 //handle references to same object  
 //handle null reference and different class objects, use .getClass() to get the class name or use instanceOf to validate subclasses as well  
 //compare for equality  
}  
public int hashCode() { return x; } //x is a hashed integer

It’s good to make equals() and hasCode() final in the base class so that problems arising form comparison between a class and its subclass instance can be avoided.

Note: *String class has a hashCode method can be utilized in the hashCode method of our object class.*

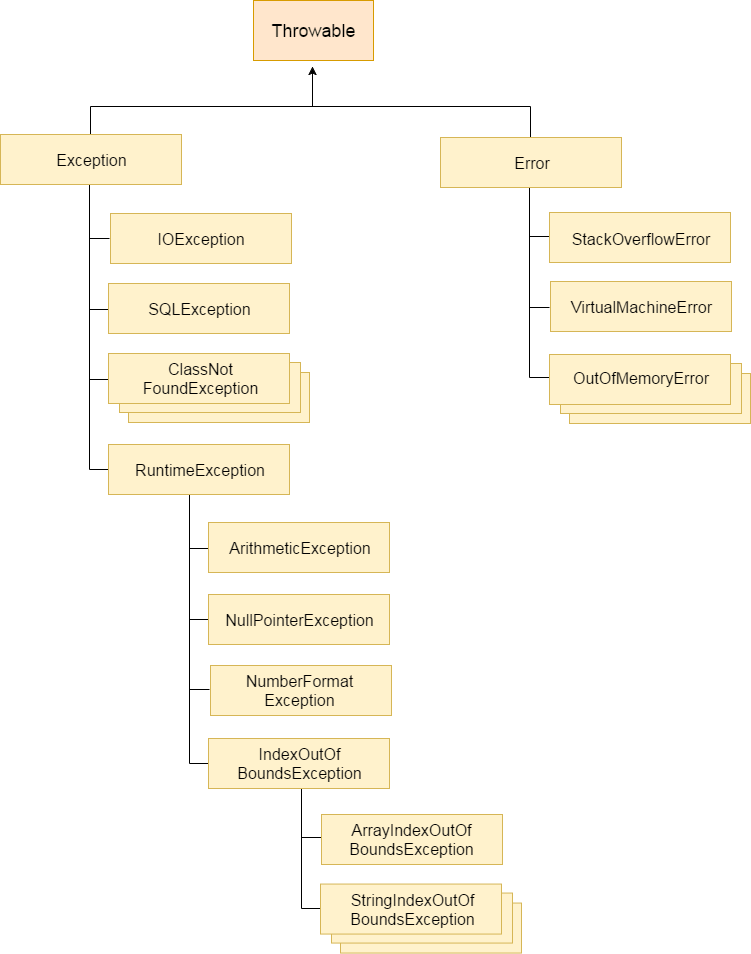
**Convert an array to a set (Collection)**  
Set<String> words = new HashSet(Arrays.asList(arr1)); //arr1 is a String[]

## Comparison

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **HashMap/HashSet** | **LinkedHashMap/LinkedHashSet** | **TreeMap/TreeSet** |
| Order | Unordered | Insertion order | Sorted |
| Override method | equals() & hashCode() | equals() & hashCode() | compateTo() |
| Runtime | Faster | Fast | Slower |

# EXCEPTION HANDLING

In Java, an exception is an event that disrupts the normal flow of the program.

Two ways to handle exceptions:

* Look before you leap (LBYL) – check for all possible cases before you do the required task
* Easy to ask forgiveness than permission (EAFP) – do the required task and then handle any exception raised, *try-catch and throws*

**Stack trace**

It is a list of method calls that were made up to a point when the exception was thrown. It is a snapshot of call stack at a given time (typically at some sort of failure).

Call stack – “the current stack of operations”

It is a stack data structure that stores information about the active subroutines of a computer program.

Types of Java Exceptions

1. **Checked exception**  
   These are checked at compile-time. All the classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions. These can not be ignored. For eg, IOException.
2. **Unchecked exception**  
   These are checked at run-time. For eg, ArrayIndexOutOfBoundsException.

**Error**  
These are irrecoverable. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise.

**try-catch**try{   
} catch(){  
} catch(){  
} catch(){  
} finally(){  
}

For each try block there can be zero or more catch blocks, but only one finally block. The finally block is optional.It always gets executed whether an exception occurred in try block or not. For example, code to close a file should be written inside the finally block so that it is always implemented.

When an exception is found in the try block, it stops right there and executes catch block. If an exception is not found, the catch block will not be executed. It is really important to ensure that the code in catch block does not raise any exception.

An exception in a method can be left unhandled which is then passed back to its caller method. And then the exception can be handled there as well.

**Alternative to catch multiple exception**  
try{  
} catch(ExceptionType1 | ExceptionType2 ex){  
}

**throw**To explicitly throw an exception from a method or any block of code

throw new ExceptionType();  
throw new ExceptionType(“description”);

**throws**throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block. The main method can also have this signature.

Type method\_name(parameters) throws exception1, exception2 {  
}

*NOTE: throws keyword is required only for checked expression*

**Some methods of Exception class**

toString() - returns the information about the exception  
printStackTrace() - prints the stack trace

**try-with-resources statement**

A try-with-resources statement ensures that each resource declared in the try statement is closed after the execution of the block. The resource must implement the AutoCloseable interface.

try(FileWriter writer = new FileWriter(“fileName.txt”)){  
}

*Using multiple resources:*

try(resource1; resource2; resource3){  
}

A try-with-resources statement can have catch and finally blocks just like an ordinary try statement. In a try-with-resources statement, any catch or finally block is run after the resources declared have been closed.

If the try block and the try-with-resources block throw exceptions, then the exception from try-with-resources are suppressed and the one from the try block is thrown and passed back.

# INPUT & OUTPUT

## Stream

A stream is a sequence of data.

* InputStream – used to read data from a source
* OutputStream – used to write data to a destination

**Byte Stream**Data are stored as a sequence of bytes. FileInputStream and FileOutputStream are most frequently used byte streams classes.

**Character Stream**Data are stored as a sequence of characters (16-bit Unicode). FileReader and FileWriter are most frequenctly used character streams classes. Though these classes internally use FileInputStream or FileOutputStream, the major difference is that these classes read/write two bytes at a time.

**Standard Stream**These are used by the user to take input from keyboard and output on the computer screen.

* Standard input (System.in)
* Standard output (System.out)
* Standard error (System.err)

A screenshot of a cell phone

Description automatically generated  
fig - Streams

### FileWriter & FileReader

|  |  |
| --- | --- |
| **FileWriter** | |
| **Constructors** | FileWriter(String fileName)  FileWriter(String fileName, Boolean append) FileWriter(File file)  //if not found, creates the file |
| **Close** | public void close() throws IOException  //ensure that this code executes |
| **Write** | public void write(int c) throws IOException  public void write(String str) throws IOException  public void write(String str, int offset, int length) throws IOException |
| **Flush the stream** | public void flush() throws IOException |
|  |  |
| **FileReader** | |
| **Constructors** | FileReader(String fileName)  FileReader(File file) |
| **Close** | public void close() throws IOException |
| **Read** | public int read() throws IOException  //reads a single character  public int read(char[] cbuff) throws IOException  //reads characters into an array  Public |
|  |  |

***NOTE: These methods throws IOException which needs to be handled using try catch block***

### Using scanner to wrap FileReader

Scanner offers more ways to read a file.

try(Scanner scanner = new Scanner(new FileReader(“fileName.txt”)))  
{  
 scanner.useDelimiter(“,”); //to set the delimiter (comma is used as an example)  
 while(scanner.hasNextLine())  
 {  
 String word = scanner.next();  
 //int num = scanner.int();  
 scanner.skip(scanner.delimiter);   
 //if delimiter is not found, NoSuchElementException is thrown  
 //use scanner.nextLine() to read sth at the end of the line as it will handle the ‘\n’  
 }  
}

*Note: the FileReader does not need to be closed manually here because when the scanner is closed by the try-with-resources statement, the close method of scanner closes the readable source (provided that it implements closeable) it was using*

### BufferedReader & BufferedWriter

While FileReader reads the input one character at a time, BufferedReader reads a chunk of data by creating a buffer. This improves the efficiency due to the decrease in disk read overhead time. However, BufferedReader also needs an input stream like FileReader to be supplied in its constructor.

BufferedReader reader = new BufferedReader(new FileReader(“fileName.txt”));

//a scanner wrapper can also be used here   
Scanner scanner = new Scanner(new BufferedReader(new FileReader(“fileName.txt”)));

//the rest of the code to get the input is similar to the code sample in the above example  
//so here instead of reading from the input stream directly, we read from the buffer containing the data read in chunk from the input stream

String input;  
while( (input=reader.readLine()) != null)  
{  
 //do this for each line read  
}

reader.write(String str); //to write to files using BufferWriter

**>>> Program 10.1\_InputOutput <<<**

### Byte stream

**FileOutputStream & FileInputStream** – These are used to create an output/input stream, which is used to write/read data as a sequence of bytes to a file.

**DataOutputStream & DataInputStream** – These are similar but allows us to write/read primitive data types and Strings to an output stream. They use FileInput/OutputStream as a parameter in their constructor. DataInputStream takes care of parsing the data types from bytes.

**BufferedInputStream & BufferedOutputStream** – These are BufferedReader and BufferedWriter equivalent for byte stream.

|  |  |
| --- | --- |
| **DataOutputStream** | |
| Constructor | DataOutputStream a = new DataOutputStream(new BufferedOutputStream(new FileOutputStream(“**filename.dat**”)));  //BufferedOutputStream is optional (It is used to improve efficiency) |
| Write int | public final void writeInt(int x) throws IOException |
| Write double | public final void writeDouble(Double x) throws IOException |
| Write boolean | public final void writeBoolean(boolean x) throws IOException |
| Write String | public final void writeUTF(String x) throws IOException |
| Number of bytes written so far | int size(); |
| **DataInputStream** | |
| Constructor | DataInputStream a = new DataInputStream( new BufferedInputStream( new FileInputStream(“**filename.dat**”); |
| Read int | public final int readInt() throws IOException |
| Read double | public final double readDouble() throws IOException |
| Read Boolean | public final boolean readBoolean() throws IOException |
| Read String | public final String readUTF() throws IOException  //it knows the length of string to be read because writeUTF method writes the length followed by the string itself |
| Skips bytes | int skipBytes(int x); |

*Example of DataInputStream:*

try(DataInputStream a = new DataInputStream(new FileInputStream(“data.dat”))){  
 boolean eof = false;  
 while(!eof){  
 try{  
 System.out.println(a.readInt());  
 } catch(EOFException e){  
 eof = true;  
 }  
 }  
} catch (IOException io){  
 io.printStackTrace();  
}

When there is no more data to read from the file, EOFException is thrown. So, we stop reading the file when this exception is caught. It is better to use .dat files to store the binary data.

We do not need any delimiter to write bytes data.

### Object Stream & Serialization

ObjectInput/OutputStream allows to read and write object as a single unit. The method of translating a data structure or an object to a form that can be stored and recreated is called serialization. For this purpose, the class and all the fields of the object must be serializable meaning it should implement serializable interface. However, serializable interface do not have any method to be overridden and it is only used as an indication to JVM about the object being serializable. But it is recommended to declare the serialVersionUID field. If it is not explicitly set, then the compiler will automatically set one but that might create a problem later when the compiler is changed.

private long serialVersionUID = 1L;

ObjectInput/OuputStream implements the DataInput/OutputStream respectively. Due to that, object stream can contain a mix of primitive types and serialized objects.

A serialized object data will be written only once to the file even if there are multiple references to it.

**ObjectOutputStream**

* Constructor  
  ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(“data.dat”));  
  //BufferedOutputStream can also be used in between
* Write the object  
  out.writeObject(x); //x is an object

**ObjectInputStream**

* Constructor  
  ObjectInputStream in = new ObjectInputStream(new FileInputStream(“data.dat”));
* Read the object  
  MyObject x = (MyObject) in.readObject();
* Requires while(true) loop and EOFException to be catched to end reading the file
* ClassNotFoundException & InvalidClassException(thrown when serialVersionUID does not match) should also be caught along with IOException

**>>> Program 10.2\_ObjectStream <<<**

## RandomAccessFile

Random access to the data is required when only a portion of data is to be read/loaded on demand. This is usually done to improve efficiency. Instances of this class support both reading and writing to a random access file. However, objects can not be written or read directly (using writeObject/readObject) using this class. And it can not be chained with other classes of IO.

In order to access a random data, an index containing the location and the length of the data must be maintained. All the index records should be of same length and loaded in the memory.

**A sample file format**  
Byte 0-3: number of objects data  
Byte 4-7: start offset of the objects data section  
Byte 8: start of index section  
Final section: objects data

**To write the file**First write the header section except the index  
Then write the data into the objects data section and track index data temporarily elsewhere  
Lastly go back to the index section and write index data

**To read the file**First read the header section  
Then read and store the index data in memory (as a field in the class)  
Lastly use the index data to read the object data on demand (make the RandomAccessFile instance a field in the class and close it when reading the file is completed)

* Constructor  
  RandomAccessFile raf = new RandomAccessFile(“filename.data”, “rwd”);  
  //rwd – open the file for read and write synchronously
* Move filepointer to new location  
  raf.seek(newLocation); //newLocation is a long that represents a memory address
* Write an int  
  raf.writeInt(x);
* Write a String  
  raf.writeUTF(x);
* Close the file   
  raf.close()