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

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

**System.out.print('H' + 'a');**  
//outputs 169

### 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. An interface can not implement another interface and can not extend another class. But it can extend one or more other interfaces.

**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. It can not have defined constructor (no name) and can not implement more than one interface at a 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).

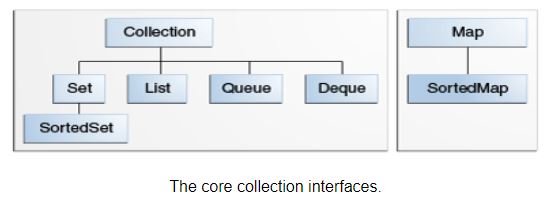
Static block

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

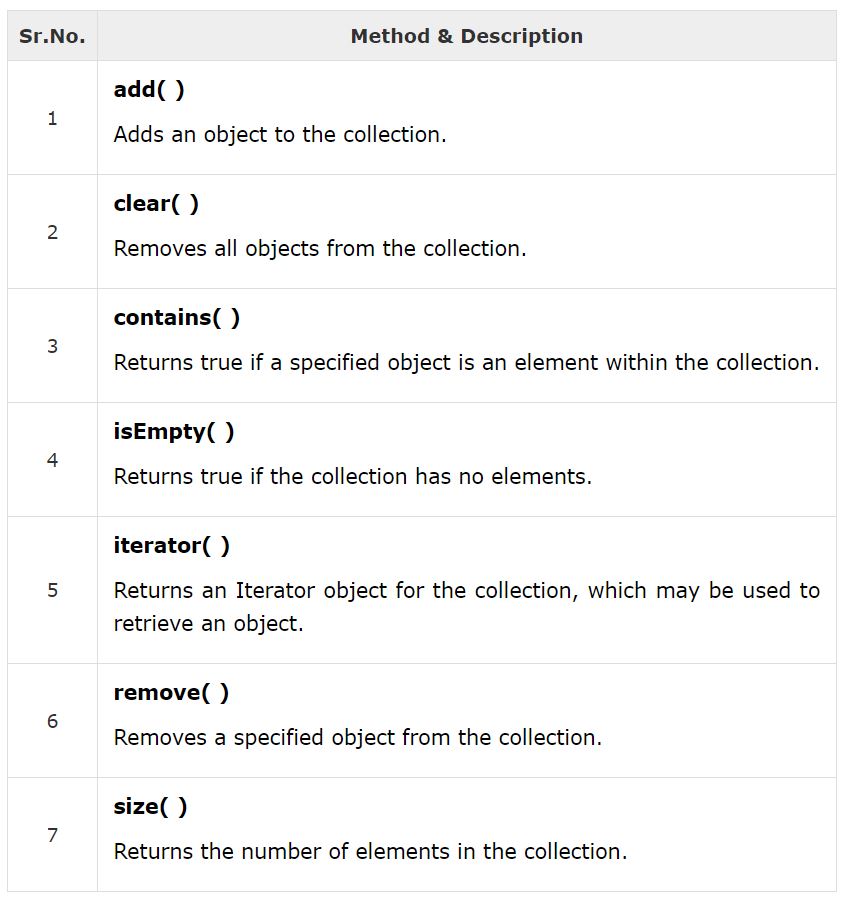
Requirements:

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.*

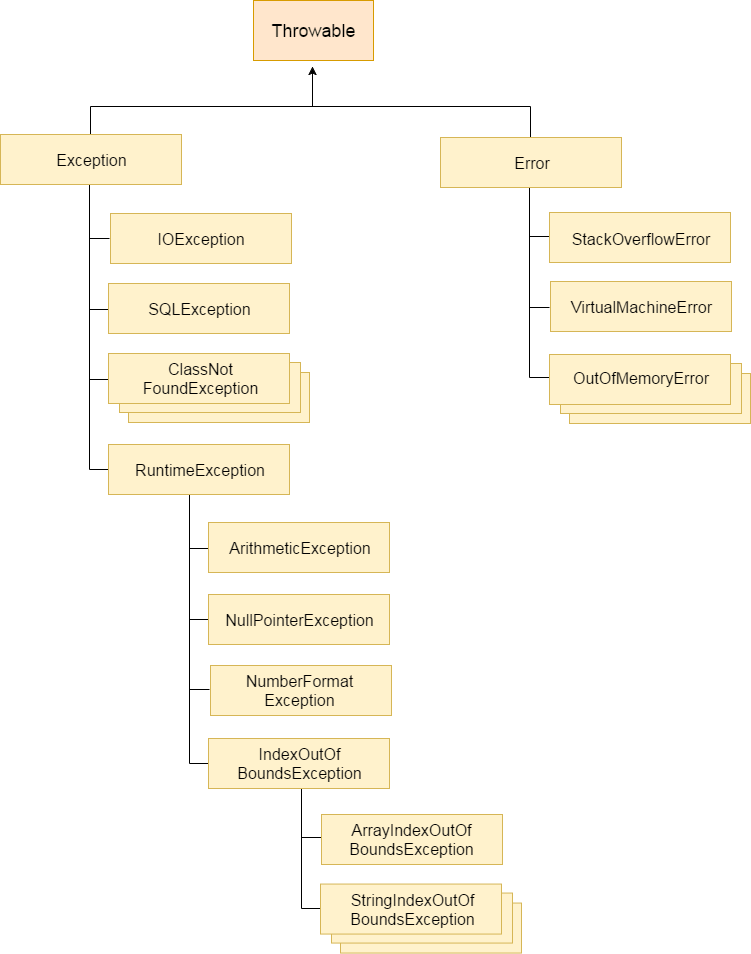
**Convert an array to a set (Collection)**  
Set<String> words = new HashSet(Arrays.asList(arr1)); //arr1 is a String[]

## 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. *(Exception – finally block will not be executed if the program exits from try block with System.exit(0) statement)*

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 (int must be converted to String)

**>>> 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  
  raf.writeInt(x);   
  raf.writeUTF(x);
* Read  
  raf.readInt(x);  
  raf.readUTF(x);
* Close the file   
  raf.close()

## Java NIO

Java NIO is an alternative way of working with I/O than the standard I/O APIs. The java.nio classes fall into one of the two buckets: those that deal with the file system, and those that deal with reading and writing data.

The classes in this package perform I/O in a non-blocking manner. When using classes in the java.io package, a thread will block while it’s waiting to read or write to a stream or buffer. However, threads using the java.nio classes will not block meaning they are free to continue executing. So, java.nio was introduced as a performance improvement.

When using java.nio, the data are processed in blocks (one block at a time). Channes and buffers are used to accomplish this. A single instance of channel can be used to both read and write data (except FileChannel). Also, the reading and writing are always buffered unlike java.io where buffered instance had to be used. Classes must implement java.nio.channel.Channel interface to be a channel.

Channels and Buffers

A channel is any data source that is read or written to. A buffer is a container for the block of data to be read or written, but can only hold one type of data. In the java.io package, you work with byte and character streams. In java.nio, you work with channels and buffers. Data is always read from a channel into a buffer, or written from a buffer to a channel.

Some implementations of channel in java.NIO:

* FileChannel
* DatagramChannel
* SocketChannel
* ServerSocketChannel

**Selectors**

A selector is an object that allows single thread to manage the io from multiple channels.

### Working with java.io stream using java.nio

Java.nio can be wrapped using java.io streams and used accordingly.

**BufferWriter and BufferReader**

*Path path = FileSystems.getDefault().getPath(“filename.txt”);  
BufferedWriter writer = Files.newBufferedWriter(path);  
BufferedReader reader = Files.newBufferedReader(path);*    
//rest is same as using the BufferedWriter and try-with-resources

**ObjectOutputStream and ObjectInputStream**

*Path path = FileSystems.getDefault().getPath(“filename.dat”);  
ObjectOutputStream out = new ObjectOutputStream(new BufferedOutputStream(Files.newOutputStream(path)));  
ObjectInputStream in = new ObjectInputStream(new BufferedInputStream(Files.newInputStream(path)));*

**>>> Program 10.4\_NIOWrapper <<<**

### Reading and writing with java.nio

#### **Text Files**

*Path dataPath = FileSystems.getDefault().getPath(“data.txt”);  
Files.write(dataPath, “data string to be written”.getBytes(“UTF-8”), StandardOpenOption.APPEND);  
List<String> lines = Files.readAllLines(dataPath);  
for(String line : lines)  
 System.out.println(line);*

Note that write method writes the data in bytes so the string has to be converted as shown. Doing so throws an IOException which needs to be handled using try-catch. The third argument in the write method specifies the write mode. It can be omitted to create a new file or overwrite. Alternatively, StandardOpenOption.CREATE or StandardOpenOption.TRUNCATE\_EXISTING can be used to specify the mode explicitly.

Each write method is isolated write meaning file is opened, written, and then closed at each call. So, it is usually better to use StringBuilder or other techniques to write the data in bulk.

#### Binary Files

**WRITE**- create a channel- create a buffer and write the bytes to it  
- write the bytes from the buffer to the channel

//Setting up channel  
try(FileOutputStream out = new FileOutputStream(“fileName.dat”);  
FileChannel channel = out.getChannel()){

//String  
 byte[] outputBytes = “Hello World!”.getBytes(); //Get bytes to be written  
 ByteBuffer buffer = ByteBuffer.wrap(outputBytes); //Create a buffer and write the bytes in it  
 channel.write(buffer); //Write bytes from buffer to the channel  
 //also returns the number of bytes written

//int   
 ByteBuffer intBuffer = ByteBuffer.allocate(Integer.BYTES); //Allocate a buffer   
 intBuffer.putInt(x); //write the int x in the buffer  
 intBuffer.flip(); //reset the buffer position to 0 for reading  
 channel.write(intBuffer); //Write the bytes from the buffer  
 intBuffer.flip(); //reset the buffer position to 0 for writing  
 intBuffer.putInt(y); //write the int y in the buffer  
 intBuffer.flip();   
 channel.write(intBuffer);

The static wrap() method wraps the bytes into the buffer, resets the buffer position, and set the capacity of the buffer as required. It is similar to calling allocate, put, and flip methods together. Instead of using different buffers, a single buffer of bigger size can be used to write the String and int.

**READ**

//setting up the channel  
try(RandomAccessFile ra = new RandomAccessFile(“fileName.dat”, “rwd”);  
FileChannel channel = ra.getChannel())

//String  
 buffer.flip(); //the same buffer that was used to write the string  
 channel.read(buffer); //read the bytes from channel to the buffer  
 String x = new String(buffer.array()); //reading the string from the buffer  
 //OR  
 byte[] bytes = new byte[size];   
 buffer.read(bytes); //read the data into the bytes  
 String y = new String(bytes);

//int   
 intBuffer.flip(); //the same buffer that was used to write the int  
 channel.read(intBuffer); //read the bytes from channel to the buffer  
 //ends itself when the eof is reached  
*Relative read*  
 intBuffer.flip();  
 int x = intBuffer.getInt(); //read the int from the buffer  
*Absolute read*  
 int x = intBuffer.getInt(0); //resets the buffer position to 0 before reading  
 //the int and set it back to wherever it was before that

Chaining the put method

buffer.put(“xyz”.getBytes()).putInt(x1).putInt(x2).put(“abc”.getBytes());

This is possible because the put method returns the byte buffer.

### Random Access using java.nio

For random access, the channel is used to random access the data and not the buffer. So, the channel must implement seekableByteChannel interface. For example: FileChannel.

Methods:

* read(ByteBuffer) – read bytes beginning at the channel’s current position and updates the position accordingly after the read
* write(ByteBuffer) – writes similarly
* position() – returns the channel’s position
* position(long) – sets the channel’s position to the passed value
* truncate(long) – truncates the size of the attached datasource to the passed value
* size() - returns the size of the attached datasource

Similar to java.io.RandomAcessFile, we use an index to store information about the data to be written and read.

**>>> 10.6\_NIO <<<**

## Comparison

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **IO Type** | | | **Setup** | **read/write string** | **read/write int** |
| java.io | Character Stream | FileReader | filename | Use scanner | |
| FileWriter | filename | write(String) | write(Int) |
| BufferedReader | FileReader | ReadLine() | |
| BufferedWriter | FileWriter | write(String) | |
| Byte Stream | DataInputStream | FileInputStream | readUTF() | readInt() |
| DataOutputStream | FileOutputStream | writeUTF() | readUTF() |
| ObjectInputStream | FileInputStream | readObject() | |
| ObjectOutputStream | FileOutputStream | writeObject() | |
| RandomAccessFile | filename, mode | readUTF() writeUTF() | readInt() writeInt() |
| java.nio | Character Stream | Files | Path | Files.readAllLines() Files.write(Path, bytes[], mode) | |
| Byte Stream | ByteBuffer & Channel | FileInputStream | channel.read(buffer) buffer.read(bytes[]) buffer.readInt() | |
| ByteBuffer & Channel | FileOutput Stream | buffer.put(bytes[])  buffer.wrap(bytes[])  buffer.putInt(int)  channel.write(buffer) | |

Copy files using FileChannel

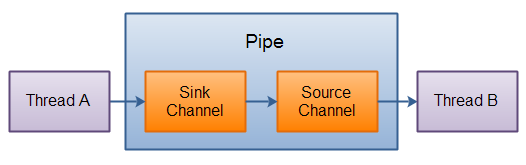
long numOfBytesTransferred = channelDest.transferFrom(channelSrc, 0, channelSrc.size());

//0 is the start position and channel.size() is the length   
//channelSrc must be positioned at 0 before doing this (that means the second argument is relative)

long numOfBytesTransferred = channelSrc.transferTo(0, channelSrc.size(), channelDest);

## Java NIO Pipe

* A Java NIO Pipe is a one-way data connection between two threads.
* A Pipe has a sink channel where data is written and a source channel from where the data can be read



Java NIO: Pipe Internals <src: http://tutorials.jenkov.com/java-nio/pipe.html>

* Creating a Pipe  
  Pipe pipe = Pipe.open();
* Writing to a Pipe  
  Pipe.SinkChannel sinkChannel = pipe.sink();  
  sinkChannel.write(ByteBuffer);
* Reading from a Pipe  
  Pipe.SourceChannel sourceChannel = pipe.source();  
  sourceChannel.read(ByteBuffer);

## Filesystem using Java NIO

The package java.nio.file is dedicated to working with the file system like copy, delete, or move files.

### Path

**Problems with java.io.File:**

* Many methods in the class do not throw exceptions or provide specific error messages
* It does not understand symbolic links (a kind of file that points to another file)
* It does not provide a way to get metadata about a file (like its permissions and owner)
* It is not efficient when working with lots of data

**java.nio approach:**

* java.nio.file.Path deals only with pointing to the file/directory whereas java.nio.file.Files handles the file operations using static methods

The Path interface is located in the java.nio.file package. A Path instance represents a path in the file system which can point to either a file or a directory. It is very similar to the java.io.File class. A Path instance is created using a static method in the Paths class (java.nio.file.Paths) named Paths.get().

Absolute path – starting at a root node – C:\downloads\data.txt  
Relative path – doesn’t specify a root node – documents\file.txt

Relative paths have to be combined with other path (like current directory) to specify the entire Path of the file or the directory.

* Path to a file in current/project directory  
  Path dataPath = Paths.get(“.”, “filename.txt”);  
  Path dataPath = FileSystems.getDefault().getPath(“filename.txt”);
* Path to a file inside folders in current/project directory  
  Path dataPath = Paths.get(“.”, “folder1”, “folder2”, “filename.txt”);
* Path to a file in the parent directory of current/project directory  
  Path dataPath = Paths.get(“..”, “filename.txt”);

datapath.toAbsolutePath(); //returns the absolute path  
datapath.normalize(); //removes any redundant direction  
datapath.getFileName(); //returns the name of the file/directory

### Files

The Files class (java.nio.file.Files) provides several methods for manipulating files in the file system. It works along with Path instances. A path might have a value that does not point to a valid file/directory, so you have to check if the file exists or not before performing the file operations.

|  |  |
| --- | --- |
| Check if exists | Files.exists(Path)  //returns a boolean |
| Create file | Files.createFile(Path)  //writing to a file also creates the file if it does not exist |
| Create tempFile | Files.createFile(prefix, suffix) //prefix – filename prefix, suffix – extension (String)  //tempFiles are stored in separate location automatically |
| Create directory | Files.createDirectory(Path) |
| Check permission | Files.isReadable(Path) |
| Files.isWritable(Path) |
| Copy | Files.copy(srcPath, destPath)  //an exception is thrown if destPath already exists  //can also copy a folder but not the files inside it |
| Files.copy(srcPath, destPath, StandardCopyOption.REPLACE\_EXISTING) |
| Move | Files.move(srcPath, destPath)  //destPath must contain a name for the file |
| Rename | Files.move(srcPath, destPath)  //srcPath and destPath should point to the same directory |
| Replace | Files.move(srcPath, destPath, StandardCopyOption.REPLACE\_EXISTING) |
| Delete | Files.delete(Path);  Files.deleteIfExists(Path); |
| Get file size | Files.size(Path); |
| Get all attributes | BasicFileAttributes attrs = Files.readAttributes(Path, BasicFileAttributes.class); |
| attrs.size(); |
| attrs.creationTime(); |
| attrs.isDirectory(); |
| attrs.isRegularFile(); |

### Directories

* **Getting contents of a directory**   
  *try( DirectoryStream<Path> contents = Files.newDirectoryStream(Path) )*  
  //here DirectoryStream is an interface, but it is instantiated using anonymous class using the newDirectoryStream (which also implements iterable interface)
* While iterating through the contents, a **DirectoryIteratorException** might be thrown which needs to be catched.
* **Filter the contents**  
  *try( DirectoryStream<Path> contents = Files.newDirectoryStream(Path,* ***glob****) )*
* **Filter out directory or file**  
  *try( DirectoryStream<Path> contents = Files.newDirectoryStream(Path, filter) )*//filter is an interface DirectoryStream.Filter<Path> with only one method accept(Path) which needs to be implemented before using this
* **Get filepath separators**File.seperator; //java.io – returns a string   
  FileSystems.getDefault().getSeparator(); //java.nio

Glob

It is similar to regular expression but simplified. It is used as a PathMatcher.

\* - will match any string   
\*.dat – will match path with the .dat extension  
\*.{dat, txt} – will match any path that has the extension .dat or .txt   
myfile\* - will match any path that begin with myfile  
a?\*.txt – will match any path that is atleast 2 characters long, starts with a, and is .txt   
? – will match exactly one character

File Stores

These are the storage drives like C drive.

*Iterable<FileStore> stores = FileSystems.getDefault().getFileStores();  
for(FileStore store : stores){  
 System.out.println(store); //volume  
 System.out.println(store.name()); //drive letter  
}*

//To get the root directories C:\ D:\ …  
*Iterable<Path> roots = FileSystems.getDefault().getRootDirectories();  
for(Path path : roots){  
 System.out.println(path);  
}*

### Walk File Tree

*Files.walkFileTree(Path, FileVisitor);*

This method is used to traverse a directory tree in depth-first order. It requires a Path instance and a FileVisitor implementation as parameters.

FileVisitor Interface  
- preVisitDirectory()  
- postVisitDirectory()  
- visitFile()  
- visitFileFailed()

If you do not need to hook into all of these methods, you can extend the SimpleFileVisitor class, which contains default implementations of all methods in the FileVisitor interface. These methods return FileVisitResult which is an enum with the values:  
CONTINUE – to continue the traversing   
TERMINATE – to stop the traversing  
SKIP\_SUBTREE – to skip the current directory  
SKIP\_SIBLINGS – to skip the files and directories on the same level

walkFileTree method can also be used to copy the directory with an implementation of FileVisitor where the methods are overridden for that purpose.

path – dir1/dir2/data.txt  
source – dir1  
target – dir3

To get the new path – dir3/dir2/data.txt  
target.resolve(source.relativize(path));

**>>> Program 10.6\_NIOFiles <<<**

## Mapping IO and NIO methods

|  |  |
| --- | --- |
| **IO** | **NIO** |
| File  File file = new File(“file.txt”); | Path  Path path = file.toPath(); |
| File resolvedFile = new File(parent, “file.txt”);  //parent/file.txt | Path resolvedPath = parentPath.resolve(childPath);  //parentPath/childPath |
| Working directory  new File(“”).getAbsoluteFile().getAbsolutePath(); | Working directory  FileSystems.getDefault().getPath(“.”).getAbsolutePath(); |
| Get all contents File[] allFiles = file.listFiles(); | Get all contents DirectoryStream<Path> = Files.newDirectoryStream(path); |

# CONCURRENCY

**Process**A unit of execution that has its own memory space. An instance of most of the Java Virtual Machine (JVM) runs as a process. It can be used interchangeably with the term application.

**Thread**A thread is a unit of execution within a process. Each process can have multiple threads. In Java, every process has at least one thread, the main thread. Creating a thread doesn’t require as many resources as creating a process. Every thread created by a process shares the process’s memory and files. However, each thread has memory that only it can access which is referred as a thread stack.

Every Java process/application runs as a single process, and each process can have multiple threads.

**Reasons to use threads**

* Heavy tasks can be offloaded from main thread to another thread and run them in background
* An API might require to use threads

**Concurrency**

It means multiple computations are being handled at the same time.

When working with threads, we can not be sure about when threads are scheduled to run. In other words, we can not guarantee in which order the threads will run.

## Threads

Thread thread = new Thread();  
thread.start();

There are two ways to specify what code the thread can execute.

* The first way is to create a subclass of Thread and override the run() method. This can be done in two ways:
  + Create a Java class that extends Thread class and override the run() method
  + Create an anonymous subclass of Thread  
    Thread thread = new Thread() {  
     public void run(){  
     //do sth  
     }  
    };
* The second way is to pass an object that implements Runnable (java.lang.runnable) to the Thread constructor. **[PREFERRED]**  
  public interface Runnable() {  
   public void run();  
  }  
  This can be implemented in three ways:
  + Create a Java class that implements the Runnable interface
  + Create an anonymous class that implements the Runnable interface
  + Create a Java Lambda that implements the Runnable interface  
    Runnable runnable = () -> { //do sth }; };   
    This is possible because the Runnable interface only has a single unimplemented method.

An instance of a thread or its subclass can only be started once.

*NOTE: Do not call run() method directly. If done so, the run() method will be executed in the main thread and not in the separate thread. Always use start() method.*

![A screenshot of a cell phone

Description automatically generated](data:image/jpeg;base64,/9j/4AAQSkZJRgABAQEAeAB4AAD/4RDyRXhpZgAATU0AKgAAAAgABAE7AAIAAAANAAAISodpAAQAAAABAAAIWJydAAEAAAAaAAAQ0OocAAcAAAgMAAAAPgAAAAAc6gAAAAgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAFB1bnR1IE1haWNoYQAAAAWQAwACAAAAFAAAEKaQBAACAAAAFAAAELqSkQACAAAAAzM5AACSkgACAAAAAzM5AADqHAAHAAAIDAAACJoAAAAAHOoAAAAIAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAyMDE5OjA1OjMxIDEzOjE0OjU2ADIwMTk6MDU6MzEgMTM6MTQ6NTYAAABQAHUAbgB0AHUAIABNAGEAaQBjAGgAYQAAAP/hCx9odHRwOi8vbnMuYWRvYmUuY29tL3hhcC8xLjAvADw/eHBhY2tldCBiZWdpbj0n77u/JyBpZD0nVzVNME1wQ2VoaUh6cmVTek5UY3prYzlkJz8+DQo8eDp4bXBtZXRhIHhtbG5zOng9ImFkb2JlOm5zOm1ldGEvIj48cmRmOlJERiB4bWxuczpyZGY9Imh0dHA6Ly93d3cudzMub3JnLzE5OTkvMDIvMjItcmRmLXN5bnRheC1ucyMiPjxyZGY6RGVzY3JpcHRpb24gcmRmOmFib3V0PSJ1dWlkOmZhZjViZGQ1LWJhM2QtMTFkYS1hZDMxLWQzM2Q3NTE4MmYxYiIgeG1sbnM6ZGM9Imh0dHA6Ly9wdXJsLm9yZy9kYy9lbGVtZW50cy8xLjEvIi8+PHJkZjpEZXNjcmlwdGlvbiByZGY6YWJvdXQ9InV1aWQ6ZmFmNWJkZDUtYmEzZC0xMWRhLWFkMzEtZDMzZDc1MTgyZjFiIiB4bWxuczp4bXA9Imh0dHA6Ly9ucy5hZG9iZS5jb20veGFwLzEuMC8iPjx4bXA6Q3JlYXRlRGF0ZT4yMDE5LTA1LTMxVDEzOjE0OjU2LjM4ODwveG1wOkNyZWF0ZURhdGU+PC9yZGY6RGVzY3JpcHRpb24+PHJkZjpEZXNjcmlwdGlvbiByZGY6YWJvdXQ9InV1aWQ6ZmFmNWJkZDUtYmEzZC0xMWRhLWFkMzEtZDMzZDc1MTgyZjFiIiB4bWxuczpkYz0iaHR0cDovL3B1cmwub3JnL2RjL2VsZW1lbnRzLzEuMS8iPjxkYzpjcmVhdG9yPjxyZGY6U2VxIHhtbG5zOnJkZj0iaHR0cDovL3d3dy53My5vcmcvMTk5OS8wMi8yMi1yZGYtc3ludGF4LW5zIyI+PHJkZjpsaT5QdW50dSBNYWljaGE8L3JkZjpsaT48L3JkZjpTZXE+DQoJCQk8L2RjOmNyZWF0b3I+PC9yZGY6RGVzY3JpcHRpb24+PC9yZGY6UkRGPjwveDp4bXBtZXRhPg0KICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICA8P3hwYWNrZXQgZW5kPSd3Jz8+/9sAQwAHBQUGBQQHBgUGCAcHCAoRCwoJCQoVDxAMERgVGhkYFRgXGx4nIRsdJR0XGCIuIiUoKSssKxogLzMvKjInKisq/9sAQwEHCAgKCQoUCwsUKhwYHCoqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioqKioq/8AAEQgBfwG/AwEiAAIRAQMRAf/EAB8AAAEFAQEBAQEBAAAAAAAAAAABAgMEBQYHCAkKC//EALUQAAIBAwMCBAMFBQQEAAABfQECAwAEEQUSITFBBhNRYQcicRQygZGhCCNCscEVUtHwJDNicoIJChYXGBkaJSYnKCkqNDU2Nzg5OkNERUZHSElKU1RVVldYWVpjZGVmZ2hpanN0dXZ3eHl6g4SFhoeIiYqSk5SVlpeYmZqio6Slpqeoqaqys7S1tre4ubrCw8TFxsfIycrS09TV1tfY2drh4uPk5ebn6Onq8fLz9PX29/j5+v/EAB8BAAMBAQEBAQEBAQEAAAAAAAABAgMEBQYHCAkKC//EALURAAIBAgQEAwQHBQQEAAECdwABAgMRBAUhMQYSQVEHYXETIjKBCBRCkaGxwQkjM1LwFWJy0QoWJDThJfEXGBkaJicoKSo1Njc4OTpDREVGR0hJSlNUVVZXWFlaY2RlZmdoaWpzdHV2d3h5eoKDhIWGh4iJipKTlJWWl5iZmqKjpKWmp6ipqrKztLW2t7i5usLDxMXGx8jJytLT1NXW19jZ2uLj5OXm5+jp6vLz9PX29/j5+v/aAAwDAQACEQMRAD8A7C3t31GW6uLq8vDIbmQfLOwGA1Tf2Qn/AD93v/gS1Gkf6m5/6+pf/Qq0K40k0fRylJSsjP8A7IT/AJ+73/wJaj+yE/5+73/wJatCijlRPtJdzP8A7IT/AJ+73/wJaj+yE/5+73/wJatCijlQe0l3M/8AshP+fu9/8CWo/shP+fu9/wDAlq0KKOVB7SXcz/7IT/n7vf8AwJaj+yE/5+73/wACWrQoo5UHtJdzP/shP+fu9/8AAlqP7IT/AJ+73/wJatCijlQe0l3M/wDshP8An7vf/AlqP7IT/n7vf/Alq0KKOVB7SXcz/wCyE/5+73/wJaj+yE/5+73/AMCWrQoo5UHtJdzP/shP+fu9/wDAlqP7IT/n7vf/AAJatCijlQe0l3M/+yE/5+73/wACWo/shP8An7vf/Alq0KKOVB7SXcz/AOyE/wCfu9/8CWo/shP+fu9/8CWrQoo5UHtJdzP/ALIT/n7vf/AlqP7IT/n7vf8AwJatCijlQe0l3M/+yE/5+73/AMCWo/shP+fu9/8AAlq0KKOVB7SXcz/7IT/n7vf/AAJaj+yE/wCfu9/8CWrQoo5UHtJdzP8A7IT/AJ+73/wJaj+yE/5+73/wJatCijlQe0l3M/8AshP+fu9/8CWo/shP+fu9/wDAlq0KKOVB7SXcz/7IT/n7vf8AwJaj+yE/5+73/wACWrQoo5UHtJdzP/shP+fu9/8AAlqP7IT/AJ+73/wJatCijlQe0l3M/wDshP8An7vf/AlqP7IT/n7vf/Alq0KKOVB7SXcz/wCyE/5+73/wJaj+yE/5+73/AMCWrQoo5UHtJdzP/shP+fu9/wDAlqP7IT/n7vf/AAJatCijlQe0l3M/+yE/5+73/wACWo/shP8An7vf/Alq0KKOVB7SXcz/AOyE/wCfu9/8CWo/shP+fu9/8CWrQoo5UHtJdzP/ALIT/n7vf/AlqP7IT/n7vf8AwJatCijlQe0l3M/+yE/5+73/AMCWo/shP+fu9/8AAlq0KKOVB7SXcz/7IT/n7vf/AAJaj+yE/wCfu9/8CWrQoo5UHtJdzP8A7IT/AJ+73/wJaj+yE/5+73/wJatCijlQe0l3M/8AshP+fu9/8CWqImbR9UsprO8utzM6sHnYgjYa1ay9W/4/LD/ro/8A6AaGkioyctH5/kSaR/qbn/r6l/8AQq0Kz9I/1Nz/ANfUv/oVaFEdianxMp3mpwWMiRyiRncEqsaFjgfSoP7dt/8Anhd/9+GqeD/kcrP/AK9pK6isJVJ8zSPKrYmpCo4o5D+3bf8A54Xf/fhqP7dt/wDnhd/9+Grr6Kn2k+5l9cq+RyH9u2//ADwu/wDvw1H9u2//ADwu/wDvw1dfRR7SfcPrlXyOQ/t23/54Xf8A34aj+3bf/nhd/wDfhq2tQ8U+H9Juja6rrum2VwAGMNzeRxuAehwxBotPFXh7UJoorDXtMuZJnKRJDeRuXYDJAAPJA5wKfNUf/DB9bqmL/btv/wA8Lv8A78NR/btv/wA8Lv8A78NXX0UvaT7h9cq+RyH9u2//ADwu/wDvw1H9u2//ADwu/wDvw1dfRR7SfcPrlXyOQ/t23/54Xf8A34aj+3bf/nhd/wDfhq6ebUbK3vreynvLeK6uQxggeVQ8u0ZbapOWwOuOlSXNzBZ2slzdzRwQRKXkllcKqKOpJPAHvT55h9cq+Ryn9u2//PC7/wC/DUf27b/88Lv/AL8NXSabq+m6zbtcaRqFrfwq2xpLWZZVDYBwSpIzgjj3qxDNFcQrLBIksbDKujAg/QijnqLcPrlXyOT/ALdt/wDnhd/9+Go/t23/AOeF3/34auvqC9vrTTbN7vUbqG0to8b5p5AiLk4GWPA5IFL2k+4fW6py/wDbtv8A88Lv/vw1H9u2/wDzwu/+/DV1ysroGRgysMgg5BFLT559w+uVPI5D+3bf/nhd/wDfhqP7dt/+eF3/AN+Grr6KXtJ9w+uVfI5D+3bf/nhd/wDfhqP7dt/+eF3/AN+Grr6KPaT7h9cq+RycOtWs1xHDtmjeQ4XzIioJrQqLxP8A8fmkf9fJ/wDQalranJyTuduGqyqxbkNdxHGzt0UEms6DU7y5gWa30e6kicZVgRyKvXX/AB5zf9c2/lV3wz/yLNj/ANchU1JSTSTM8VWnTa5TI+2al/0BLr8xR9s1L/oCXX5iurorPmn3OT61V7nKfbNS/wCgJdfmKPtmpf8AQEuvzFdXRRzT7h9aq9zlPtmpf9AS6/MUfbNS/wCgJdfmK6DVdVs9E0q41LVJvItLZN8sm0ttHrgAk/gK5G3+M/gG6uYreDXt0srhEX7HOMknAHKU06j2YfWq1r3L32zUv+gJdfmKPtmpf9AS6/MVt6Hr2m+JNLXUdFuftNo7MiyeWyZKnBGGAPUelaFJyqLdh9ardzlPtmpf9AS6/MUfbNS/6Al1+Yrq6KOafcPrVXucp9s1L/oCXX5ij7ZqX/QEuvzFdU7rHGzucKoJJ9BVHRNc07xHo8OqaNcfabKfd5cuxk3YYqeGAPUHtRzT3uH1qr3MP7ZqX/QEuvzFH2zUv+gJdfmKfqvxJ8IaJro0fVNcgt74lVMZR2VCx4DOAVT33EYHJ4rdfVLGPVIdNe6jF7PG0scGcsyL1bHYc9ad6lr3D61WXU5/7ZqX/QEuvzFH2zUv+gJdfmK6uilzT7h9aq9zlPtmpf8AQEuvzFH2zUv+gJdfmK2rXXtNvNdvdGtrnff2Co9zD5bDyw4yvzEYOR6E1o0c0+4fWq3c5T7ZqX/QEuvzFH2zUv8AoCXX5iuroo5p9w+tVe5yn2zUv+gJdfmKPtmpf9AS6/MV1dFHNPuH1qr3OU+2al/0BLr8xU1herf2vnKjRkMVZW6gg10tchoX/HlN/wBfMn/oVVTlJys2dGHr1Jz5ZM06y9W/4/LD/ro//oBrUrL1b/j8sP8Aro//AKAa3lsepDf7/wAiTSP9Tc/9fUv/AKFWhWfpH+puf+vqX/0KtCiOwVPiZTg/5HKz/wCvaSuorl4P+Rys/wDr2krqK5ZfEzwcR/FkFFFFI5wooooA81+N2i6W/wAOdV1R9Ns21BfJVbswKZgPNUYD4zjBI61k+MtDi0zxh4Bg8J2un6Rc3E1wfMS0UIG8pAXZF27iBnGT1x2r0nxR4bs/Fvh240bUpJ4ra4K72gYBxtYMMEgjqPSo9S8KWOqa1omp3Etws2is7W6oyhX3qFO/IJPA7EVrCaikn3/Qvm0+T/FaHml7408WaJ4b8dWN3rCXupeH5LU22pC0jjLCVhkGPBXgfXqfapvFPifxn4ag0fSP7WudQ1TWGkupLjTtHjmltY1VSIoocgOMk5ZjnHNdjqXw20fVD4jNxc3y/wDCRGA3eyRBs8rG3ZlTjOOc5/CrvibwVp3ii3slubi9srmwbda3tjN5U8WRggNg9R14queOmna/3L9df+HDTp5/pb9Th4fHHiYfD1n1id9C1P8AtJbKK/1PS3Sa4ibkSJbKrbpe20fLx1rN0r4n+JbDwf4vn1UT3d5o7wLay6hYC0lzMduJIkJA2/eHPIPJ547eX4V6DJ4ag0iOfUYHgu/ty6jFc4ujcHrKZMcsRxnH0xUlh8MNCsrTWLaaa/1CHWo40vBe3Pml2QYEm7G7eSdxOeuMAAAUc1P3rr+tPu6iVtP66/5HG/2f4jsPi94GHijXYNakkju3jljtVgMZMPzJheGXoQ2AeTntXYfFLSf7d8Ix6fFf2FrdSXkLW0V/IEiu5FbIhPru9BnOPxqLRfhVpWja9purjVtbv7vTQ6QG+vBKAjJs2Y28KASQFxyec8AdF4l8Naf4r0ZtN1VZPL3iSOWF9kkMg+66N2YUpyT5bdP82xL4rvseSaSnkv4t06TSJPB3iOfRRLLa24jkspY0XmSNFGATllxubG48kg0/TdW1/QPhf4YtLbxQkVxqUbS26w6X9ou1iWMFYIYVBWQ5zl3K9etehaF8OtK0Se+uZrzUtYu76A20t1qlz50ohPWMNgYXvWVH8HNIh0+yt4Nd8Qwy2DubW7jvws0KOoBiUhcBPlzgDqTzyarni3r5fhf/AIH49NB3X9eiX6fkczpvxH8T3fw+hUuE1m61/wDsWK+urQRtGGwfMeIfKHGcbemR3xzc8f2PinTvhH4nt/FGr2usw4ga1u47cQSn96m5XRRtAB6EEnrmuoj+Fvh+Pw1e6Hvvmtbq8N8jvcZltZiBho3xkEY43bj1znNRD4UaK+g6rp13f6tey6sI1udQu7oS3JRGDKoYrgAEenf2GFKUHt5fp92tyoNRkn2f6/5EHw68TXfivUtRn+2G3sLBIraDS2hCyEFAwuJCRu+f+EAgY68139YkHhSxtfE8eu2stxDdCzFnKiMvl3CD7pcbclh2II9OlbdRUkpO6/r+tzJaBRRRWYwooooA57xP/wAfmkf9fJ/9BqWovE//AB+aR/18n/0Gpa2o9T1MF8LIrr/jzm/65t/Krvhn/kWbH/rkKpXX/HnN/wBc2/lV3wz/AMizY/8AXIVNX4kZ43eJqUUUVmeeFFFFABXnnhP/AJLP48/3LL/0Ua9DrOtNB02x1q/1a1tvLvtRCC6l8xj5mwYXgnAwPQCmna/mrfkVf3WvT8zwHS4tV0T4Kx+LNN8RapbTWV+wgsYpQlsQZwpDoBl8kk8n2xXeww6lrPxe8SCXXdWhsNIjtLmKwtLpkjlcxZ2kf3Tg5UY3E89K7H/hA/DZ8KN4b/s7/iUNJ5htvPk5bdvzu3bvvc9a0bTQtOsdZvdVtbfy72/WNbmXex8wIMLwTgYHoBWzqJrTfX9P8mOUk5N9/wDO/wCR4X4X1/x1e6ho/iSSW7aHUNQEMputdt1s5kLlWijtWwyuAOMEtlScHNP8Z+I9eF9qXiHw7quvTw2Wq/ZRL9pjtrGIqygRCDJac8kFvlHfGK9bsvh14T0/xI2vWeiwRakzl/NDMQrHqQhO1T7gCq158KfBOoaheXt3oEL3F6SZnEki5JIYlQGwpJHVcE5Pqc1GpBSi7bf8D+unoJtO/n/wTl9QbUfFXxH8R2Vx4n1DQ7fQbSGS2gs5hGkhZN7SSg5DqDgEHsccc52vgl/ySDRu/M//AKOetrXPh/4W8Sajb3+t6PDd3NsoVJGdxkA5AYAgOPZs9/WtTRND07w5o8Ol6Nb/AGayg3eXFvZ9uWLHliT1J71HOvZ8vp+F/wA7ilrb+uljx3xFZTxHxhqvhq80/XPD73DHXdG1BJIJI5UKltj/ACnIwSDkD0DEVbjtLTV/ix4Y1W31HVNPtJtAS+jhW6I8tEKkRHIJKED5h/Fyc13ms/DPwfr+snVdW0SGe9ON0gkdA+OhZVYKx9yDkcGr+reDfD+t3+nXup6bHNcaYwa0cOyeVgggYUgEAgcHIojNJLy/ya+f9alNp3/rqv8ALc8as/EHiH/hKfDetWOp69d2Gr6qIGuL65jitbhS7Kyx2YLFVC4w5bqOgJFdFZrqfirUPEut3nje+8Ovo2qvaW8SzKtpDFGRgyxkgPuyRkkfiBiuvtfhT4Jsbxbu00GKKdbhLlJFlkykiElSvzfKMn7owpwMjgYsap8N/COta8NZ1PQ7e4vgVLSMzBXI6FkB2t/wIHPQ1XtIK1v62/4PbcTd23/XX/gfceVeJ9ev9K+JHi99Ne4itbqLT1vdVswGNnblVDSLz947gBjOBk9q91sIoodPt47eaSeJY1CSySmRnGOCWJJYn1PWs8+FdFbUNTvWsVafVoVgvWZ2ImRV2hducDjjgCrumaba6Ppdvp2no0drbII4kaRnKqOgyxJOPc1m5JwUe39f18wk7u/9bL+vuLVFFFZkhRRRQAVyGhf8eU3/AF8yf+hV19choX/HlN/18yf+hVdP4zrwn8U06y9W/wCPyw/66P8A+gGtSsvVv+Pyw/66P/6Aa6JbHtw3+/8AIk0j/U3P/X1L/wChVoVn6R/qbn/r6l/9CrQojsFT4mU4P+Rys/8Ar2krqK5eD/kcrP8A69pK6iuWXxM8HEfxZBRRRSOcKKKKACiiigAooooAKKKKACiiigAooooAKKKKACiiigAooooAKKKKAOe8T/8AH5pH/Xyf/QalqLxP/wAfmkf9fJ/9BqWtqPU9TBfCyK6/485v+ubfyq74Z/5Fmx/65CqV1/x5zf8AXNv5Vd8M/wDIs2P/AFyFTV+JGeN3ialFFFZnnhRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAUUUUAFFFFABRRRQAVyGhf8eU3/XzJ/wChV19choX/AB5Tf9fMn/oVXT+M68J/FNOsvVv+Pyw/66P/AOgGtSsvVv8Aj8sP+uj/APoBrolse3Df7/yJNI/1Nz/19S/+hVoVn6R/qbn/AK+pf/Qq0KI7BU+JlOD/AJHKz/69pK6iuXg/5HKz/wCvaSuorll8TPBxH8WQUV4mvi3x3JeJdW2vWpt5fE02ix2ktgjBUJIV2YFSdvYDGcck9KlufiF4n0nwXd29xdNf6rH4ik0Vb+CxV5CoGQ6wAhWfsFzjpnNaeyk1df1t/mjHlf8AXz/yZ7PUH260/tAWH2qH7YYvOFt5g8zy8437eu3PGema8T1HxF401H4YeJotTk1axewWOWLULvSvsUt3ExIkjKglVwcfMpzjr1NX10vxK/xH0yxsPEcUOpf8IsDLqclgrHZ9oyAsWdoP3Vyc8ZOM0ez1d3/Vm/0C2n9d0v1PZKK8i03xr4o8R6D4Q0601O30zUtaF19o1NrZZMeQxGFjOFLNjkfXGKa/jrxONJfRP7Rsxqq6+mi/22IB5ZUqW8zy/u7+MFemTj3odKSdv63t+Ymrb/8ADb/5HrEV9aT3k9pBdQyXNtt8+FJAXi3DK7lHIyORnrVS08SaHf6g1hYazp9zeJu3W8N0jyLjrlQc8d68V1STWdOi+Ji3OqLfXscWmxXF9bReSWjI2udqk7TtJBxx16V0Pj7w54P0T4Yxap4dtbGyvIDA+lXtnhZpZMjbhx80hIyec+vbNP2a0u97fik/1+ZTjrb+tket0V5JruueNrnXfGEOka9DplvoNlBdrG1lHKxYwF2jDEcAkEkkMeBjAzml4h+I3ia7n0Ox0b+0LSa60aHUp5NJ0lb+R2fgrsdgFQevJyQKXs3/AF6X/QXK/wCvl/mj2O5u7azjWS8uIrdGdY1aVwoLMcKoJ7knAHei2u7a8V2tLiKdY3aNzE4YK6nDKcdCDwR2ry211zVvEmu/D6LxFYTafcPJd3NzbSwNHukhQqjbW5H3t3frXQeF5Xtvip4y05Cfs7/ZbxVxwsjx7X/E7AfwodO2j8/wdv8AMXT+v67Hb0UUVkIKKKKACiiigAooooAKKKKAOe8T/wDH5pH/AF8n/wBBqWovE/8Ax+aR/wBfJ/8AQalraj1PUwXwsiuv+POb/rm38qu+Gf8AkWbH/rkKpXX/AB5zf9c2/lV3wz/yLNj/ANchU1fiRnjd4mpRXE/F29vdP+HF3Ppd5PZXIngVZ4HKMuZVB5Hselec69/wkOiw+MWg8Ya5JH4eubOe1EtyCZGl27xIQoJTnhAQvtTp0+fr1t+X+Zw2u7I98pjTRJKkTyIskgJRCwBbHXA74zXg/j/xR4jb4ganZQX11YrYJEbER67b6dESyBt7pKP343dQDgAY4zWqIJbv4t+E9S1/WLixu7zRklZIb5BFJMCo8pMZDI55Kgnd2NL2b5U77/5N/oJ6L+vL/M9Y0XWtP8Q6TFqWj3H2i0mLBJNjJnaxU8MAeoI6VerwHwrDqej+D/BeuWmv6kq3WuCxbTxIFtRC80obKAfMxIJ3MT146Cp9b8R3mPEWtz+NL3TvEGmak9vYaEk6rFJGrgIDb4zJuBPzf4VcqXvNLz/T/NFSjaTX9df8j3C9vINOsLi9vH8u3tomllfBO1VGScDk8DtWRqfjjw5o2gWmtanqkdvYXiK9vIyMWlVgCCqAbjwQTxx3xXlfi/V5dQ1TxYniLxVdaDLp+nItjpUU6xx3RkgJcMjA+bliV45Hr0q3oN5p+jeKPBupeJpIrexk8LQw2F1cjEUU4ALjeeFYoepxwcZ5oVLS7/rRv9BaWv8A10/zPT/DnizQ/Ftm914e1GO9ijba+0FWQ+6sAwzjjI5rYrx/VdUs9R8d+JdX8IXKSRWvhidb2/sm+Rp/vR4kXguFHUdMYzxxH4dfXNK17wJeS+I9V1P/AISDT5WurW7n3xApbq6bFwMHpljknkk8nK9ndX20/wA/8g5X/Xon+p7JWRe+KdG0979bu82f2cIzdkROwhEn3dxAI9z6Dk4HNeG+FPF/i3UPFGm3xvpmurm+Ed5a3XiC1ERj37WRLJtrowAGDknjPO6u30NEvvBPxHkv/vzajqCTbmHyqsYVR7YUClKnyJt9v8v8xqPvcvml+f8AkepA56UVgeA7ie6+HugT3WTM+nwliTkn5Bz+PWt+onHlk49jOLukwoooqRhRRRQAUUUUAFFFFABXIaF/x5Tf9fMn/oVdfXIaF/x5Tf8AXzJ/6FV0/jOvCfxTTrL1b/j8sP8Aro//AKAa1Ky9W/4/LD/ro/8A6Aa6JbHtw3+/8iTSP9Tc/wDX1L/6FWhWfpH+puf+vqX/ANCrQojsFT4mU4P+Rys/+vaSuorl4P8AkcrP/r2krqK5ZfEzwcR/FkYUfgrw/EqLHYYEeonU1/fScXJ6yfe9+nT2pJ/BPh250vUdOuNMSW11O5a7uo3dzvmbGXBzlTwPu4x2reoou/6/ryRhd/1/XmznNO+H3hfStAvdF0/SI4rC/BFzH5js0oxjlyS3HbnjtU2jeCtA8P3MFxpFgYJbe1aziYzyPtiMhkK/Mx/iJOevbpxW7WPoGi33iGxlvpNcurc/aJEEcartABwKpOcnZa3DyKVx8PPCt34ag0C50iOTTbd2khiaR90bMxYkPu3DJJ7+3SpT4E8Mnwr/AMI4dHg/snO77ONw+b+9uzu3f7Wc+9bn/CE3X/QyX3/fC0f8ITdf9DJff98LV+zrPp+I/evcxdC8EeHPDSXaaJpUVsl5GkVwm5nWVVBABDEjoTn1zzmqOl/C7wZo2tLqunaDBFeKxdHMjuqMTnKozFVI7YAx2xXUf8ITdf8AQyX3/fC0f8ITdf8AQyX3/fC0+Sve9vxCztYzn8MaRJc6tO9pmXWYlhv281/3yKhQDr8vykj5cVnat8OvCmuadYWOq6Qk8GnRCG1HmyK0aAYC71YMRgdya6L/AIQm6/6GS+/74Wj/AIQm6/6GS+/74Wl7Ot2/EPeOZ1HwZDHJ4bl8Ow29mdBucxQsSqGB1KSrkAncQc5PUjnrmrXh7w9Npuua9q988TXWrXKsBESQkMaBI1JIHOMk9snHbNbn/CE3X/QyX3/fC0f8ITdf9DJff98LT5K3b+tP8gs7W/r+tSSio/8AhCbr/oZL7/vhaP8AhCbr/oZL7/vhan2NXsKzJKKj/wCEJuv+hkvv++Fo/wCEJuv+hkvv++Fo9jV7BZklFR/8ITdf9DJff98LUV14Pu7ezmmXxHfExxs4BVecDNL2VTsFmWaKz9Bupb3QbSe4bdK8Y3N6n1rQrMQUUUUAc94n/wCPzSP+vk/+g1LUXif/AI/NI/6+T/6DUtbUep6mC+FkV1/x5zf9c2/lV3wz/wAizY/9chVK6/485v8Arm38qu+Gf+RZsf8ArkKmr8SM8bvEu3thaalam21G1gu4GIJinjDqSDkHB44IzUE+h6TcrdLc6ZZzC82/aRJbo3n7fu78j5sYGM9KvUVF2jgM/UPD+javNFLqukWN9JCMRvc2ySFB1wCwOKkvNH0zULq3ub/TrS6ntW3W8s8Cu0JyDlSRlTkA8egq5WPrcAvNT0exkklSG4udsgjcqSNp7ild9BFpND0mOzt7RNLsltrWUTQQi3QJFICSHVcYVsknI55NE2h6TcarHqc+l2Ut/EAI7t7dDKmOmHIyOp71f/4V/o3/AD1vv/ApqP8AhX+jf89b7/wKat/Y1f6ZVmUL3QtJ1K5S41HS7K7nRDGks9ujsqkEFQSMgEE5HuaW40bS7vS0026020msEVVW1kgVolC/dAQjAAxxxxV7/hX+jf8APW+/8Cmo/wCFf6N/z1vv/ApqXsKv9MLMo22jaZZaa+nWem2lvYyBg9rFAqxMG+8CgGDnv605dJ05Gs2WwtVaxUpaEQrm3UjaQnHyjAxxjirn/Cv9G/5633/gU1H/AAr/AEb/AJ633/gU1P2NX+mFmZ0fh/RotWbVItIsU1BiS12tsgmJIwfnxnpx1rDvfA5ng8SW1nqX2S38QFWlQQbjExULKVO4ffUDtweeeldb/wAK/wBG/wCet9/4FNR/wr/Rv+et9/4FNQqNVdh2e5Da20NlZw2tsgjhhRY40HRVAwB+VS0v/Cv9G/5633/gU1H/AAr/AEb/AJ633/gU1J0Kr1f5kqNtBKKX/hX+jf8APW+/8Cmo/wCFf6N/z1vv/ApqPYVey+8fKxKKX/hX+jf89b7/AMCmo/4V/o3/AD1vv/ApqPYVey+8OViUVi3elQ+H/GNjbWEtwYri3dpFllL5IIx1rarJpp2YgooopCCuQ0L/AI8pv+vmT/0KuvrkNC/48pv+vmT/ANCq6fxnXhP4pp1l6t/x+WH/AF0f/wBANalZerf8flh/10f/ANANdEtj24b/AH/kSaR/qbn/AK+pf/Qq0Kz9I/1Nz/19S/8AoVaFEdgqfEynB/yOVn/17SV1FcvB/wAjlZ/9e0ldRXLL4meDiP4sgooopHOFJ8Pzjw3J/wBfc3/oVLUfgI48Oyf9fc3/AKFW1D+L8mVHcb4B13UNaXxF/adx5/2LXbq0g+RV2RIRtXgDOM9Tz701viVpK3rr9kvzpsd//Zsmr+WgtkuM7dhy3mY3kJvCFMn73eq+i+E9f8PX2ptpmu6abPUdTl1B4rjSpHkTzGBKBxcKOgxnb+Haqj/DV2EmmjWf+Kel1T+1HsTa/vhJv8zy1m34EfmANjYWxkbu9eivs37K/wCF/wBfmXL7Vu7t+Nv0+RuWPji21HWDaWWk6rPaC7ks/wC044Fe385AdwOGLgAgrvKBcjGap6f8TdJvvF0HhyWzvrC+ulka3F2IlaQIM8xhzLHlcsvmImQDTdJ8H6lol9NDp3iDyNElvZrw2SWS+dmXJaPzixGzexYYQN0G6sbw98K7jQtS8PTNrdvLbeH2n+zQxaYImmEqlXMr+Yd0n3PnAH3TkHOQLpft+Nv8wfWxu/DjxBqOt/DqDVtXke9vDLdBikaqzhJpFVQBgZwoHb3qax+ImkakuktZQ3s39qafLqKLHDvaGKPaGDqpJ3bm2hVDZYEVP4M8Of8ACI+F4NH+1/a/Jlmk87y/Lz5krPjGT03Y69qw7f4Z28MPieE6nL5WuQPbQBIgpsIXLsyKc/MN8rtzjqB2olfW39f1+hWl/n+F/wDImsvixot3Pq1tNZ31neaTp7ahPazNA8hiXO4Yjlba44yjlWG4cU1PG0mq614Rktxqek2usSTlLa4tYH+2RiDzFZmEhaIDORgbiRggDms22+Fl0slxJd65aEy6DLoccdppQgjijbBVwvmH5gdxIyAcjG3HO0fA+6bwk/8AaTD/AIRuF4vlhwbjdB5WQd3yY6/xenvTdlt/W/8AwPvJ/wCD+St+NxbD4m6TfeLYPDktnfWF9dLI1uLsRK0gQZ5jDmWPK5ZfMRMgGsf4efEe41Lw94ej8Sw3rXmrmaOHUmhjWCaVGc+XhDlW2rwSgU7TgmmeHvhXcaHqPh6Ztbt5bbw+ZxbQxaYImlEqFXaV/MO6T7nzgD7pyDnIn8O/DO50ePw/aX2vJf6foMstzbwrY+U7zPvALPvPyqHbCgA55JPSl0/rz/T8Qezt/X9aDPB/xFuJfCOjf2rFd61rmpz3git7OOJHaOGZ1LHcyRqqqEHJBJI6812+g6/Z+JNEh1TTTJ5Mu4FJU2PG6kqyMOzKwIPuK4D/AIU9arpOiwm7sLy70l7ra2paWLm3mSeQuVaEuDlSRhg/Y8c8dt4Z0OPw34ettMjaFzECXeC0jtkZickiOMBVGTwOTjqSckvdMcvi0/r+vyOb8C+I/FGq+OPE+n+Ko7W0FnFaSW1jbMJBbiRXJDSYBduBk/dyOOK7XUm/4lV3/wBcH/8AQTWNpnhv+zvGeua/9r8z+1o7aPyPLx5XlKwzuz82d3oMY71q6i3/ABK7v/ri/wD6CamXwie7f9bHIeFv+RZsv+uf9a1qyfC3/Is2X/XP+ta1eQtkZBRRRTA57xP/AMfmkf8AXyf/AEGpai8T/wDH5pH/AF8n/wBBqWtqPU9TBfCyK6/485v+ubfyq74Z/wCRZsf+uQqldf8AHnN/1zb+VXfDP/Is2P8A1yFTV+JGeN3ialFFFZnnhWXqH/Ix6B/19n/0GtSsrUv+Ri0D/r7P/oNHVeqGWPi3qF3pvwx1O6066mtLhHg2TQSFHXM8YOGByMgkfjWx4q8TQ+FPDcmqT28t2weOGG3iIDTSyMERcngZZhk9hk89KTxR4dtfFnhu50XUJp4be5KFpLdlDqVcOMFgR1UdQaoXXgxdV0e60zX9c1TV7W4C7RP9niaB1YMro0MSEMCAeSRx0r1+lvP/ACNexm3vxMl0VtTsdd0dIdYs1t2gtrW886K6E7+XHiVkQr8+Q2V4AyN3SruqeMtX0PRJLnWtDs7O6+1pbQh9XT7PMGXO9ZCgkPQjYIi5I+VSOQyX4c6ZeW2pDVr7UNSvdRWJX1CeSNZ4hEd0Qj8tFRdr/N93knnPSnXPgGK9ghN34g1ua+t75b6DUHnjMkMgQphU8vygu0kFQnOSTk80f1+X/B/QX9fh/mcrqPxCufE3hvSbuyS50i5t/F1tpl5FHLIu4CQblyyo5VlIyrIp6gjivQdf8RHQb3Ro5LQy22pXos5JxJj7OzKxQlcHILKF6jGRWDbfDLSbaxFsb/U5x/bEetNJNMjM9wuM5OzG1iuSAByTjA4HQeItDtvEug3Ok30k0UU4H723fZJEysGV0bBwwIBBx2oekdO9/wAI3/FMN3/Xd2/Q5eb4nzHRrO/tNJtyL+7uIrNLi/KGeGJtokREjeV2fghI43wDksBjMVh8VbjWrLwo+h6Ek1x4jW5xHdXphS2aD74LCNiwznB2g8DjnjXvPAVhMujnT9Q1DSZtHtWtLaeyePcYmVVZWDoynO1TnAIIyDUej/DnSNEfQjZ3F8y6Cbk2iyyqwxOcsrHbkgduQfXNNWT1/rf89Px+YY9n43OkWXiW5CSXEsXiN7JE1TVlWNTsQ4jJTcqYziJFkcnoDk4ytU+It94j0PQ7vTLe4068g8XRaZd2qTyRrLtDEqSyI2wgqSGQEd1yK6uf4caZI00sF/qNrdSau2sR3MMke+GcpsIUMhUrt4wwbqearp8LNJXTxa/2nqzEauus+eZ08w3IXaWzsxgn5iMdTxgcUo7K/Tl/Dlv+o5W1t1v+PN/wDH8ReNtR1C2srUxyaPqWn+K7KwvorW8MiSI5DcOApZGVhkFR3BFWl+NGmNr5tdlj9i/tT+y8/wBpp9s37tnm/ZtufK38bt+cfNtrXX4daYbVUub3ULm4/taPV5ruV4/NnnjxtDYQKEAUDaqjgVPa+B7ax1R7iw1bVbWzkvWv30yCdUgaZuWJIXzNpb5im/aT2xxTjpZP+tI/5MUtdv61f/AJviC2ujwXfP4a1CHTbiOKSSW6kiMjpGsbMfLGQN5IUZPQEnqBU3gK6nvPh14dubuaSeebTLd5JZXLM7GNSWJPJJPetLUbOPU9LurCcssV1C8LlCAwVgQcZ781Houlw6FoVjpNo0jwWNulvG0pBYqihQSQAM4HoKUdObzt+t/0G9bfP9DnvEpz470r/r1k/nV2qHiI58daX/17Sfzq/XmVPjfqZPcKKKKzEFchoX/HlN/18yf+hV19choX/HlN/wBfMn/oVXT+M68J/FNOsvVv+Pyw/wCuj/8AoBrUrL1b/j8sP+uj/wDoBrolse3Df7/yJNI/1Nz/ANfUv/oVaFZ+kf6m5/6+pf8A0KtCiOwVPiZTg/5HKz/69pK6iuXg/wCRys/+vaSuorll8TPBxH8WQUUUUjnCsbwl4i0rTNIlt76+iglF1KSjnnBatmsDQvDEniSzm1CfU5YXa5kTYkKEABvcVVNzU1yLUavfQ6P/AITLQP8AoK2/5mj/AITLQP8AoK2/5ms//hXQ/wCg1cf9+I/8KP8AhXQ/6DVx/wB+I/8ACurmxH8q/r5lamh/wmWgf9BW3/M0f8JloH/QVt/zNZ//AArof9Bq4/78R/4Uf8K6H/QauP8AvxH/AIUc2I/lX9fMNTQ/4TLQP+grb/maP+Ey0D/oK2/5ms//AIV0P+g1cf8AfiP/AAo/4V0P+g1cf9+I/wDCjmxH8q/r5hqaH/CZaB/0Fbf8zR/wmWgf9BW3/M1n/wDCuh/0Grj/AL8R/wCFH/Cuh/0Grj/vxH/hRzYj+Vf18w1ND/hMtA/6Ctv+Zo/4TLQP+grb/maz/wDhXQ/6DVx/34j/AMKP+FdD/oNXH/fiP/CjmxH8q/r5hqaH/CZaB/0Fbf8AM0f8JloH/QVt/wAzWf8A8K6H/QauP+/Ef+FH/Cuh/wBBq4/78R/4Uc2I/lX9fMNTQ/4TLQP+grb/AJmoL7xfoUmn3CJqkDM0TAAE8nBqt/wrof8AQauP+/Ef+FRXXw/+z2c0y6xOTHGzgGCPnAz6UnLEW+Ff18w1IvC//Is2X/XP+ta1Zvh64e58P2csuC7RjJAxn8q0q41sQFFFFMDnvE//AB+aR/18n/0Gpai8T/8AH5pH/Xyf/Qalraj1PUwXwsiuv+POb/rm38qu+Gf+RZsf+uQqldf8ec3/AFzb+VXfDP8AyLNj/wBchU1fiRnjd4mpRRRWZ54Vi69P9j1HSLySOR4be5LSGNCxA2+1bVY+twC81PR7GSSVIbi52yCNypI2nuKTv0A0P+E90n/nne/+AzUf8J7pP/PO9/8AAZqs/wDCv9G/5633/gU1H/Cv9G/5633/AIFNXb/tPkX7xW/4T3Sf+ed7/wCAzUf8J7pP/PO9/wDAZqs/8K/0b/nrff8AgU1H/Cv9G/5633/gU1H+0+Qe8Vv+E90n/nne/wDgM1H/AAnuk/8APO9/8Bmqz/wr/Rv+et9/4FNR/wAK/wBG/wCet9/4FNR/tPkHvFb/AIT3Sf8Anne/+AzUf8J7pP8Azzvf/AZqs/8ACv8ARv8Anrff+BTUf8K/0b/nrff+BTUf7T5B7xW/4T3Sf+ed7/4DNR/wnuk/8873/wABmqz/AMK/0b/nrff+BTUf8K/0b/nrff8AgU1H+0+Qe8Vv+E90n/nne/8AgM1H/Ce6T/zzvf8AwGarP/Cv9G/5633/AIFNR/wr/Rv+et9/4FNR/tPkHvFb/hPdJ/553v8A4DNR/wAJ7pP/ADzvf/AZqs/8K/0b/nrff+BTUf8ACv8ARv8Anrff+BTUf7T5B7xz1xq8GueMbGeyjnEcMDq7SRFcEketdBWLd6VD4f8AGNjbWEtwYri3dpFllL5IIx1rarllzcz5tyQoooqRBXIaF/x5Tf8AXzJ/6FXX1yGhf8eU3/XzJ/6FV0/jOvCfxTTrL1b/AI/LD/ro/wD6Aa1Ky9W/4/LD/ro//oBrolse3Df7/wAiTSP9Tc/9fUv/AKFWhWfpH+puf+vqX/0KtCiOwVPiZTg/5HKz/wCvaSuorl4P+Rys/wDr2krqK5ZfEzwcR/FkFFFFI5wpPh+ceG5P+vub/wBCpaj8BHHh2T/r7m/9Crah/F+TKjuN8A67qGtL4i/tO48/7Frt1aQfIq7IkI2rwBnGep596a3xK0lb11+yX502O/8A7Nk1fy0FslxnbsOW8zG8hN4Qpk/e71X0Xwnr/h6+1NtM13TTZ6jqcuoPFcaVI8ieYwJQOLhR0GM7fw7VUf4auwk00az/AMU9Lqn9qPYm1/fCTf5nlrNvwI/MAbGwtjI3d69FfZv2V/wv+vzLl9q3d2/G36fI3LHxxbajrBtLLSdVntBdyWf9pxwK9v5yA7gcMXABBXeUC5GM1T0/4m6TfeLoPDktnfWF9dLI1uLsRK0gQZ5jDmWPK5ZfMRMgGm6T4P1LRL6aHTvEHkaJLezXhsksl87MuS0fnFiNm9iwwgboN1Y3h74V3Ghal4embW7eW28PtP8AZoYtMETTCVSrmV/MO6T7nzgD7pyDnIF0v2/G3+YPrY3fhx4g1HW/h1Bq2ryPe3hlugxSNVZwk0iqoAwM4UDt71NY/ETSNSXSWsob2b+1NPl1FFjh3tDFHtDB1Uk7tzbQqhssCKn8GeHP+ER8LwaP9r+1+TLNJ53l+XnzJWfGMnpux17Vh2/wzt4YfE8J1OXytcge2gCRBTYQuXZkU5+Yb5XbnHUDtRK+tv6/r9CtL/P8L/5E1l8WNFu59WtprO+s7zSdPbUJ7WZoHkMS53DEcrbXHGUcqw3Dimp42k1XWvCMluNT0m11iScpbXFrA/2yMQeYrMwkLRAZyMDcSMEAc1m23wsulkuJLvXLQmXQZdDjjtNKEEcUbYKuF8w/MDuJGQDkY2452j4H3TeEn/tJh/wjcLxfLDg3G6Dysg7vkx1/i9Pem7Lb+t/+B95P/B/JW/G4th8TdJvvFsHhyWzvrC+ulka3F2IlaQIM8xhzLHlcsvmImQDWP8PPiPcal4e8PR+JYb1rzVzNHDqTQxrBNKjOfLwhyrbV4JQKdpwTTPD3wruND1Hw9M2t28tt4fM4toYtMETSiVCrtK/mHdJ9z5wB905BzkT+Hfhnc6PH4ftL7Xkv9P0GWW5t4VsfKd5n3gFn3n5VDthQAc8knpS6f15/p+IPZ2/r+tBng/4i3EvhHRv7Viu9a1zU57wRW9nHEjtHDM6ljuZI1VVCDkgkkdea7bQtetPEmhw6npbOIpgw2zRlXidSVZHXsVYEEe3XvXA/8KetV0nRYTd2F5d6S91tbUtLFzbzJPIXKtCXBypIwwfseOeO08OaGPDnhuDS7eSAvGrEyQ2cdvGXYkkiKMBQMnp1x1JOSX0Y38Wn9f1+Rz2jalr1h8S4vDt5rz6+h017vUGa0jhWzfeBEE2LkBgW+V2Y4XOa7TUm/wCJVd/9cH/9BNcn4K8I6z4Vkuft+uWWqi8me4upzprxXM8jdC0nnMuFHAUJgAADFdRqLf8AEru/+uL/APoJqXpG39f10J6v+v67nIeFv+RZsv8Arn/Wtasnwt/yLNl/1z/rWtXkLZGYUUUUwOe8T/8AH5pH/Xyf/QalqLxP/wAfmkf9fJ/9BqWtqPU9TBfCyK6/485v+ubfyq74Z/5Fmx/65CqV1/x5zf8AXNv5Vd8M/wDIs2P/AFyFTV+JGeN3ialFFFZnnhWXqH/Ix6B/19n/ANBrUrK1L/kYtA/6+z/6DR1Xqhlj4t6hd6b8MdTutOuprS4R4Nk0EhR1zPGDhgcjIJH41seKvE0PhTw3Jqk9vLdsHjhht4iA00sjBEXJ4GWYZPYZPPSk8UeHbXxZ4budF1CaeG3uShaS3ZQ6lXDjBYEdVHUGqF14MXVdHutM1/XNU1e1uAu0T/Z4mgdWDK6NDEhDAgHkkcdK9fpbz/yNexm3vxMl0VtTsdd0dIdYs1t2gtrW886K6E7+XHiVkQr8+Q2V4AyN3SruqeMtX0PRJLnWtDs7O6+1pbQh9XT7PMGXO9ZCgkPQjYIi5I+VSOQyX4c6ZeW2pDVr7UNSvdRWJX1CeSNZ4hEd0Qj8tFRdr/N93knnPSnXPgGK9ghN34g1ua+t75b6DUHnjMkMgQphU8vygu0kFQnOSTk80f1+X/B/QX9fh/mcrqPxCufE3hvSbuyS50i5t/F1tpl5FHLIu4CQblyyo5VlIyrIp6gjivQdf8RHQb3Ro5LQy22pXos5JxJj7OzKxQlcHILKF6jGRWDbfDLSbaxFsb/U5x/bEetNJNMjM9wuM5OzG1iuSAByTjA4HQeItDtvEug3Ok30k0UU4H723fZJEysGV0bBwwIBBx2oekdO9/wjf8Uw3f8AXd2/Q5eb4nzHRrO/tNJtyL+7uIrNLi/KGeGJtokREjeV2fghI43wDksBjMVh8VbjWrLwo+h6Ek1x4jW5xHdXphS2aD74LCNiwznB2g8DjnjXvPAVhMujnT9Q1DSZtHtWtLaeyePcYmVVZWDoynO1TnAIIyDUej/DnSNEfQjZ3F8y6Cbk2iyyqwxOcsrHbkgduQfXNNWT1/rf89Px+YY9n43OkWXiW5CSXEsXiN7JE1TVlWNTsQ4jJTcqYziJFkcnoDk4ytU+It94j0PQ7vTLe4068g8XRaZd2qTyRrLtDEqSyI2wgqSGQEd1yK6uf4caZI00sF/qNrdSau2sR3MMke+GcpsIUMhUrt4wwbqearp8LNJXTxa/2nqzEauus+eZ08w3IXaWzsxgn5iMdTxgcUo7K/Tl/Dlv+o5W1t1v+PN/wDH8ReNtR1C2srUxyaPqWn+K7KwvorW8MiSI5DcOApZGVhkFR3BFWl+NGmNr5tdlj9i/tT+y8/2mn2zfu2eb9m258rfxu35x822tdfh1phtVS5vdQubj+1o9Xmu5Xj82eePG0NhAoQBQNqqOBU9r4HtrHVHuLDVtVtbOS9a/fTIJ1SBpm5YkhfM2lvmKb9pPbHFOOlk/60j/AJMUtdv61f8AwDR8V21lcaMZ9W1e90uwsm+03Elpdm23ooOVZ1w4XvhSpOBz2rO+Gc+rT+BLSbXDcmSWSV7YXhzOLcufJEhPJbZtyTz685qx4v8ACVv4y023sb2/vrOGC5S5xaGP94yHKhw6MGUHBwRjIGa1NNtJ7GzEN1qNzqUgJPn3SxK59sRoi8fSkuv9f1/XcH0/r+v67HOeJTnx3pX/AF6yfzq7VDxEc+OtL/69pP51frzKnxv1M3uFFFFZiCuQ0L/jym/6+ZP/AEKuvrkNC/48pv8Ar5k/9Cq6fxnXhP4pp1l6t/x+WH/XR/8A0A1qVl6t/wAflh/10f8A9ANdEtj24b/f+RJpH+puf+vqX/0KtCs/SP8AU3P/AF9S/wDoVaFEdgqfEynB/wAjlZ/9e0ldRXLwf8jlZ/8AXtJXUVyy+Jng4j+LIKKKKRzhWN4S8RaVpmkS299fRQSi6lJRzzgtWzULWVq7FntoWYnJJjBJqoycJcyHsWv+Ey0D/oK2/wCZo/4TLQP+grb/AJmqf2Cz/wCfSD/v2KPsFn/z6Qf9+xW31ip5D5mXP+Ey0D/oK2/5mj/hMtA/6Ctv+Zqn9gs/+fSD/v2KPsFn/wA+kH/fsUfWKnkHMy5/wmWgf9BW3/M0f8JloH/QVt/zNU/sFn/z6Qf9+xR9gs/+fSD/AL9ij6xU8g5mXP8AhMtA/wCgrb/maP8AhMtA/wCgrb/map/YLP8A59IP+/Yo+wWf/PpB/wB+xR9YqeQczLn/AAmWgf8AQVt/zNH/AAmWgf8AQVt/zNU/sFn/AM+kH/fsUfYLP/n0g/79ij6xU8g5mXP+Ey0D/oK2/wCZo/4TLQP+grb/AJmqf2Cz/wCfSD/v2KPsFn/z6Qf9+xR9YqeQczLn/CZaB/0Fbf8AM1BfeL9Ck0+4RNUgZmiYAAnk4NRfYLP/AJ9IP+/Yo+wWf/PpB/37FJ4ib7BzMo+F/wDkWbL/AK5/1rWpFVUUKihVHAAGAKWsOhIUUUUAc94n/wCPzSP+vk/+g1LUXif/AI/NI/6+T/6DUtbUep6mC+FkV1/x5zf9c2/lV3wz/wAizY/9chVK6/485v8Arm38qu+Gf+RZsf8ArkKmr8SM8bvE1KKKKzPPCsXXp/seo6ReSRyPDb3JaQxoWIG32raooAr/APCe6T/zzvf/AAGaj/hPdJ/553v/AIDNViitvb1CuZlf/hPdJ/553v8A4DNR/wAJ7pP/ADzvf/AZqsUUe3qBzMr/APCe6T/zzvf/AAGaj/hPdJ/553v/AIDNViij29QOZlf/AIT3Sf8Anne/+AzUf8J7pP8Azzvf/AZqsUUe3qBzMr/8J7pP/PO9/wDAZqP+E90n/nne/wDgM1WKKPb1A5mV/wDhPdJ/553v/gM1H/Ce6T/zzvf/AAGarFFHt6gczK//AAnuk/8APO9/8Bmo/wCE90n/AJ53v/gM1WKKPb1A5mc/cavBrnjGxnso5xHDA6u0kRXBJHrXQUUVk227sQUUUUhBXIaF/wAeU3/XzJ/6FXX1yGhf8eU3/XzJ/wChVdP4zrwn8U06y9W/4/LD/ro//oBrUrL1b/j8sP8Aro//AKAa6JbHtw3+/wDIk0j/AFNz/wBfUv8A6FWhWfpH+puf+vqX/wBCrQojsFT4mU4P+Rys/wDr2krqK5eD/kcrP/r2krqK5ZfEzwcR/FkFFci/xU8Ex6k9hN4gt4bmOSSJ1lR0CNHncCxUAdDjJ57Zq5B4+8L3HhaTxHFq8P8AZUbFGuGVlww/h2kBifQYye1Pldr2MLO9joqK42/+Knha28GT+JbK+/tC0ik8kJCjB2lOcIVYArnGckDjnmsp/i/pEHiG0a7vbW30C50f7aty8cnmtN53l7FHVhweAueCelNQk3t/VrhZ2/rvY9Hornr3x74Y07w3ba9eavDHpt3jyJtrEyewQDdkdxjjvilPjvwyPCv/AAkZ1iD+yc7ftA3H5v7u3G7d/s4z7UuVroG50FFcDovjq/8AGGoeJbfwhLpdxHYxWr6bcTrIEcyKS/m4OeCCMAAjvWbpfin4i3HirU9L1BPC6Q6KYJNQeJLklopFLnyuSSwUHggc461Xs3ez06js/wCv68z1Cisa28XaHd3Wm29vqCPLqlsbqzXYw82IDJbkccdjg9eODWdf/E3whpmk2WpX2tRxW18Cbc+TIWkAJBOwLuAyCMkYqeV9v6/pC32Oqorl9U8YR+f4cg0CW2u21252xTHLoIEUvIwwRzgYHoTz0xVjw94hm1LXNe0i+SJbrSblVBiBAeGRA8bEEnnGQe2Rnvijlf8AXy/zDpf+v60OgoooqRBRRRQAUUUUAFFFFABRRRQBz3if/j80j/r5P/oNS1F4n/4/NI/6+T/6DUtbUep6mC+FkV1/x5zf9c2/lV3wz/yLNj/1yFUrr/jzm/65t/Krvhn/AJFmx/65Cpq/EjPG7xNSisfxV4msvCHh+bWNUjnktoWRWEChm+Zgo4JHc+tcq/xo8Pwif7RpmuwPa3Cw3SS6eVNsrY2yyEnCoc8ZO4/3amMJS2X9f0zgsehUVyWvfEfStB1WSxaw1XUGtlV72bT7MzR2SkZBlbIx8uW4zwKpXPxKUfEDTNAsNJvLyzvrZbhb+GIsrK+Nrrz/AKsZ+Zu3TFHK7XsFtLndUV5Z4G+K6XWkaLa+I49Snvb+4e2bUhZBLUSmRwkZcYG7aF+6D15743dQ+K2g6brFxaS22py2tpMLe71SG0LWltJnG15M8EEgHAPUVTpyTsNpptPodtRXB+MfiZZaOuq6bpkOoXN/a2LTNd2dp50FmzRlozI3YHg9CPXvVLUpfF9/8ONP8S6X4s/s14dEW7uYf7Nhm+0yCLeTub7memAMD0pcr5eZ7f8AD/5DUW2l3/4H+Z6TRXnvg7WdasLDQ7rxbr02sDxIkX2QJp8UKWkhiMm1mUgtuHAOOq9s1r2fxF0e+gtZLeG8JudTOmpGY1DBwu7eRu+5tw2euCOKcqbjLlI6X/r+tGdXRXEWnxX0C81eG1W31KKzuJ/s1vq0toVs55c42LJnkkgjkAcGqsviPW9S0rxtqWlXZhTR5mt7GLyUYFoEDSE5GTvJK+wHGDzU8rtd/wBbf5lcrvY9Boqloupx61oVjqcA2x3luk6qewZQcfrV2k04uzJTurhRRRSAKKKKACiiigAooooAK5DQv+PKb/r5k/8AQq6+uQ0L/jym/wCvmT/0Krp/GdeE/imnWXq3/H5Yf9dH/wDQDWpWXq3/AB+WH/XR/wD0A10S2Pbhv9/5Emkf6m5/6+pf/Qq0Kz9I/wBTc/8AX1L/AOhVoUR2Cp8TKcH/ACOVn/17SV1FcvB/yOVn/wBe0ldRXLL4meDiP4sjyO38B62Le3SfTFyvjJtUf97H/wAe+TiT730+Xr7VBq/w78RXmg6x9kilt7pfFMuq2sUF2kMlxCVAGyQZEbckgsOMcivY6K09rJL+vL/JGPM/6+f+bPG7D4e67N4F8WC6stSi1fVoEjSLUtZivZJjHypLrGoX0GWP4V0Wg6Nrl14/07Xta0QafHFoBsnRrmObyphNwAVPOU5yBjnGa9CrHg1q9vRI+naHeXUKSNGJUK4JBwaTqNt6b/5NfqK7tb+un+SPNtJ8G+KfD+geE9SttGhvtS0NrxZtLlukQss0jYZJOUBAwfocdac/gTxKdJfXBptn/az6+msnRROvl7QpXy/M+7v5yW6Z59q9M+36x/0LV9+a0fb9Y/6Fq+/Nabqzbvb8H3v+Y27tt9f+D/mct4F0rxBF4y8T654i0iLS/wC1VtGhiiuUmA2IwYEjuMjJwAT0yOa0NB0S/s/iP4s1O7t9llqKWYtpN6nzNkZV+AcjBPcCtn7frH/QtX35rR9v1j/oWr781qXJvp0ts/L/ACC718/+B/keXN4E8VWWn3zWNmJbrTZxbaMrXKDfbEzgsTn5cLcdCQT5YrZu/DOu+EfEWn6t4U0ODXYYtHTSmtHulgeHYd28M4wQe46k13H2/WP+havvzWj7frH/AELV9+a0/aS7fg+1v8/m2Dbbu/61ueZ6T4WvPBut+A21Mx5a6vop1gP7qCWdSyIvt8pH1rqPC8T3PxU8ZaigP2dPstmrZ4aRI9z/AIjeB+Nb1zJqF5Gsd54SurhFdZFWVUYBlOVYA9wRkHtRbSahZq62nhK6gWR2kcRKihnY5ZjjqSeSe9Pnb3XdbPv/AMOJ6/Pf77/5GzRWZ9v1j/oWr781o+36x/0LV9+a1lr2f3AadFZn2/WP+havvzWj7frH/QtX35rRr2f3AadFZn2/WP8AoWr781pr6lq0UbSSeG75VUEsSV4Ao17P7gNWiq9heJqFhDdxBlSVQwDdRVigQUUUUAc94n/4/NI/6+T/AOg1LUXif/j80j/r5P8A6DUtbUep6mC+FkV1/wAec3/XNv5Vd8M/8izY/wDXIVSuv+POb/rm38qu+Gf+RZsf+uQqavxIzxu8TO+IHhy78V+D59K06SCOeSaGQNOxC4SRWPIBPQelc54i+HeqatB40W2nskfXzZ/ZjI7DYIdu7fhTjocYz+FekUUozcdv62/yRwptNM8h8Y/B671nxjdazp9vot/HfLGZY9VluUMDKoUlPJZdwIAPzenFdFL4K1XTfFXhvUfDjaatrptj/Z91FcNKD5RZSTFyxzgHG5j2yT1HeVnarf3NrNZ29hAk1xdy+Wgkfao4z1o9o7Jdv+G/URw9l8O9Wt/AvhjRXuLI3Okayl/Owd9jRiWRyFO3JbDjqAOvNQ6h8PPFDx6voOmarpkfhrWbt7q4kmic3cPmMGdEA+QjI6nnk+leg/ZPFv8A0DLH/wACaPsni3/oGWP/AIE1fNUbu4/h6f5FNtu/9df82cFrHgHxNDqGuR+E9Q0yHTNdtFhuUvkkaWFliMY2FeDkYyT07A456eLw7dp8LV8NmSH7YNI+w+YGPl+Z5OzOcZ259s47VrfZPFv/AEDLH/wJo+yeLf8AoGWP/gTUv2jjytP7uwk2mmuhzOo+DtRuPhjpWi2k9tHrOlRWrW87M3lLNEFBOQM7SNw6dDVTRPh3PpXimzuXmt30u0sFRYgW8w3XlLC0nIwB5aY65yTxXY/ZPFv/AEDLH/wJo+yeLf8AoGWP/gTTbqO+j18hK6SS6Hndn8OPEwtdO8N3+qaY/hbTLtbmF4onF5MqsXWN/wCADJxkc8CmQTDwz4a+Imm3AAnW7nuLePnMq3KDyseuXyvHcYr0f7J4t/6Blj/4E0fZPFv/AEDLH/wJobm004vXy9NfwGnZp22/4P8AmVvCWlSaH4O0jTJv9baWccUmDn5go3frmteqP2Txb/0DLH/wJo+yeLf+gZY/+BNTJTlJtxevkSo2Vi9RVH7J4t/6Blj/AOBNH2Txb/0DLH/wJpcs+z+4dmXqKo/ZPFv/AEDLH/wJo+yeLf8AoGWP/gTRyz7P7gsy9RWUl7qlrrcOn6zZwQtPE0kbQy7uh71q1IgooooAK5DQv+PKb/r5k/8AQq6+uQ0L/jym/wCvmT/0Krp/GdeE/imnWXq3/H5Yf9dH/wDQDWpWXq3/AB+WH/XR/wD0A10S2Pbhv9/5Emkf6m5/6+pf/Qq0Kz9I/wBTc/8AX1L/AOhVoUR2Cp8TKcH/ACOVn/17SV1FcvB/yOVn/wBe0ldRXLL4meDiP4sgooopHOFHw/P/ABTcn/X5N/6FRTPAJx4dk/6+5v8A0KtqH8X5MqO5uaVrmn60Lv8Asy48/wCxXT2k/wAjLslT7y8gZxnqMj3q/mvK/h54p0DSr3xVYanrmm2d5J4mvClvcXcccj5YAYUkE5PFYcuv6mLydzrF+PGa+I/Ig0Y3LqjWfmAYFtkI0XkZfzSuc878ivRWvKu6X42/z+4uWnN5N/hf/L7z3DNGa8n8N6zbT+Jpzr3iDVYfEyavdRjSILmR1eBQ3lqbbBQR+WFYS7VycHfk4rm/B3iu+vPiB4Wmg16aWLVBdrqOnTX811JA2wsizZIijk3I21EijwFIyw5oWtrdr/hcHpfyPbdF1zT/ABDpceo6PcfaLSRnRZNjJkoxVuGAPDKR0q/mvPfg+gm+E1rEWdBJPeKWRirDNzLyCOh965rSdd1/7NqIvLzUriXwZpFxbXS27sHvbslgj/MGDkRRq4LK3MuccYpSaV/v/r8F8yra287fjb+vQ9nzVC91zT9P1XT9NvLjy7vUmdLSPYx8wou5uQMDCjPJFeFaR4lv0vNei07xM99ZS+EZ7xDDqFxdeRcp2E0jH96quC2wRgbl+QYrp7G5vdNuvhp5ep6pcf2nHPcXwmvpZjcv9j3chmIxuGQowoPIAqmrav8Arf8AyJ328/wSf6nru7HeqGi65p/iHR4NU0e4+0WU4Jjl2Mm7BKnhgCOQe1eI+D/Fd9eeP/C80GvTSxaoLtdR06a/mupIG2FkWbJEUcm5G2okUeApGWHNN+GupCGy8C22iatdS6hNPcpqWnfanaNbUead7Qn5UwwTa4ALE4yaVtP68/69Aeif9d/8j3DRdc0/xDpceo6PcfaLSRnRZNjJkoxVuGAPDKR0q/mvnbSdV1Kx8IeEIGuUtNDmuNTNzNNq0umRNMJ38tHuY1LLwXIXgMRz0r134dXWoXfgOwl1a+XUJWD+XdL5h82LcfLJZ0RmO3Hz7Ru+8M5zTto2hy0lb+v6/U2rDxHpOqaxqGladfx3N7puwXcUeT5JbOAT0z8pyAcjHOKs6kf+JTd/9cH/APQTXnfgLR9O8P8AxT8Z6do1nFZ2kNrpwSKJcD7kmSe5J7k5JPWvQNSb/iVXf/XF/wD0E1MvhE9G1/Wxx3hb/kWbL/rn/Wtasnwt/wAizZf9c/61rV5C2RkFFFFMDnvE/wDx+aR/18n/ANBqWovE/wDx+aR/18n/ANBqWtqPU9TBfCyK6/485v8Arm38qu+Gf+RZsf8ArkKpXX/HnN/1zb+VXfDP/Is2P/XIVNX4kZ43eJqUUUVmeeFZeof8jJ4f/wCvw/8AoJrUrK1H/kYtA/6+z/6DR1Xqhm/408T/APCIeErvW/sn2z7M0Y8nzPL3b5FT72DjG7PTtW1JcRQW7zzyLFFGpd3dsKqgZJJPQCuS+J+kah4g+HWpabo1sbq8laExwh1QttmRjyxAHCnqazvF9rrPjbwZeaXD4dvNPmSSCdYtSmt/KvBHKrtDmKWQjcFxyAOR716/T5/hoa9jsLPxJomoaZPqWn6zYXVjb5866gukeKLAydzg4GAQTk9KYnirw/Joz6vHrumvpkbbHvVu4zCrZAwXztByR37ivNte8I+IfEt5r+tQaU2m/aI7BItLuZ4t96babzW3mNmRcg7Fyx6c7RWj4gsdf1e1ivLDwj/Zrf2vHNdxxS2Zv7iJYivmqxzGkgJABEm/aOGUmj+vy/ryF/X4XOl1vx1YafpWk6jpLW+sWup6nBp6TW10pRfMfaXDKGDbSOn6iuhmv7S3ube3uLmGKe5YrBE8gVpSBkhQeWIAJ47V43pngbxHDoQgfS5onbxnDq4jnv0ndLXKMSzlyWZeQeSSQSN2c16B470m91LRbe80W3Fzq2k3cd9ZQmQR+a68NGWPADIzrzxyKHZRv5/haP5XYdbL+nd/noa934p0CwtXur7XNOtreOc2zyzXcaIso6xkk4DDByvWkuvFXh+ws7a7vtd022trtPMt5pruNEmXj5kYnDDkcj1FedT+Edd0nTvDENvp1xqEcENzJqv9n3EMN211NtdiszlWRC+4N5bq2MdQMVX8HeBdf0lvACanp4H9htqQu3Nwkgj80ny2U5yQe2Bkd8U0tdf63/peoabnoVj4ttXt9XutZksdLtNNvmtftD6jE6MoC4ZyDiJiWxsbkceoqrrnj7TdL07RtQ0+W11Ox1TUo7D7VBdr5UYbduk3DIO3YcjI+oxXFXHhXxPbpqj2+mtJHceLH1Bkgkt/tD2phCh4WlO2N93fKuADgjOazbPwN4mt9EjX+w5XdPGSayLa41COZhbbM8yMxywPBySSwJyRzSjra/8Ad/8Abb/m/uY5aXt5/wDt1vyX3o9J1rx3p9hpelahpD22sWupapBpyzWt0pRTI+0sGUMDt9P1Fa//AAkOjHWzow1ex/tQDJsftKefjG7Pl53dOenSvMZ/B/iC+abUF0lrNr3xdaaobF5ot0EEQVXkYqxXcdpYhSx5HU5qCz8Ba/FrL2N6mqXFqfETasl1Dc2Udqql/MEhJja48wfcKgBT03BTmnHW1/60j/m/ut6KWm39av8AyX3/AH+keMPFtp4N8Pyand291dtkpDbWsTSSTOFZ8DA4AVWJY8AAmrnhzWP7f8L6ZrHk/Z/t9pFc+Tv3eXvUNtzgZxnrgVH4itpr/wAL6paWqeZPcWcsUaZA3MyEAZPA5NVfBdjc6T4E0LTtQi8m6tNPghmj3BtjrGARkZB5HalH7V/L9b/oN9Lef6GV4mOfHelf9esn86u1Q8RnPjrSv+vaT+dX68yp8b9TJ7hRRRWYgrkNC/48pv8Ar5k/9Crr65DQv+PKb/r5k/8AQqun8Z14T+KadZerf8flh/10f/0A1qVl6t/x+WH/AF0f/wBANdEtj24b/f8AkSaR/qbn/r6l/wDQq0Kz9I/1Nz/19S/+hVoUR2Cp8TKcH/I5Wf8A17SV1FcvB/yOVn/17SV1FcsviZ4OI/iyCiiikc4VD4DOPD0v/X3N/wChVNWHFoN7ab00/W7q2hZ2fy1AwCTk1dOfJPmsNaM7zNGa4f8AsvWP+hjvP++Vo/svWP8AoY7z/vla6frK7Fcx3GaM1w/9l6x/0Md5/wB8rR/Zesf9DHef98rR9ZXYOY7jNGa4f+y9Y/6GO8/75Wj+y9Y/6GO8/wC+Vo+srsHMdxmjNcP/AGXrH/Qx3n/fK0f2XrH/AEMd5/3ytH1ldg5juM0Zrh/7L1j/AKGO8/75Wj+y9Y/6GO8/75Wj6yuwcx3GaM1w/wDZesf9DHef98rR/Zesf9DHef8AfK0fWV2DmO4zVbUT/wASu6/64v8A+gmuQ/svWP8AoY7z/vlaR9I1aSNkfxFeMrAggqvIoeJTWwcw/wALf8izZf8AXP8ArWtVbTbJdO06G0Ri6xLtDHqas1xrYgKKKKAOe8T/APH5pH/Xyf8A0Gpai8T/APH5pH/Xyf8A0Gpa2o9T1MF8LIrr/jzm/wCubfyq74Z/5Fmx/wCuQqldf8ec3/XNv5Vd8M/8izY/9chU1fiRnjd4mpRRRWZ54Vk6n/yMOg/9fZ/9BrWqlqWlW2qxxrc+YDE25GjcqVOPUUAdZuo3Vwv/AAitr/z933/gS1H/AAitr/z933/gS1df1l9vx/4BfMd1uo3Vwv8Awitr/wA/d9/4EtR/witr/wA/d9/4EtR9Zfb8f+AHMd1uo3Vwv/CK2v8Az933/gS1H/CK2v8Az933/gS1H1l9vx/4Acx3W6jdXC/8Ira/8/d9/wCBLUf8Ira/8/d9/wCBLUfWX2/H/gBzHdbqN1cL/wAIra/8/d9/4EtR/wAIra/8/d9/4EtR9Zfb8f8AgBzHdbqN1cL/AMIra/8AP3ff+BLUf8Ira/8AP3ff+BLUfWX2/H/gBzHdbqN1cL/witr/AM/d9/4EtR/witr/AM/d9/4EtR9Zfb8f+AHMW/EJz460v/r2k/nWhWVZ+HbSyvVukknklVSqmWUtgH61q1zSlzNskKKKKkQVyGhf8eU3/XzJ/wChV19choX/AB5Tf9fMn/oVXT+M68J/FNOsvVv+Pyw/66P/AOgGtSsvVv8Aj8sP+uj/APoBrolse3Df7/yJNI/1Nz/19S/+hVoVn6R/qbn/AK+pf/Qq0KI7BU+JlOD/AJHKz/69pK6iuVu9Pe4u4rmC7ltpo1KhowOQaT7HqP8A0G7v8lrnlCfM2keRWw9SVRySOrorlPseo/8AQbu/yWj7HqP/AEG7v8lpck+xj9Wq9jq6K5T7HqP/AEG7v8lo+x6j/wBBu7/JaOSfYPq1XsdXRXKfY9R/6Dd3+S0fY9R/6Dd3+S0ck+wfVqvY6uiuU+x6j/0G7v8AJaPseo/9Bu7/ACWjkn2D6tV7HV0Vyn2PUf8AoN3f5LR9j1H/AKDd3+S0ck+wfVqvY6uiuU+x6j/0G7v8lo+x6j/0G7v8lo5J9g+rVex1dFcp9j1H/oN3f5LR9j1H/oN3f5LRyT7B9Wq9jq6K5T7HqP8A0G7v8lo+x6j/ANBu7/JaOSfYPq1XsdXRXKfY9R/6Dd3+S0fY9R/6Dd3+S0ck+wfVqvY6uiuU+x6j/wBBu7/JaPseo/8AQbu/yWjkn2D6tV7HV0Vyn2PUf+g3d/ktH2PUf+g3d/ktHJPsH1ar2LXif/j80j/r5P8A6DUtZ39mTyXUE15qVxciBtyK4GM4rRrWlFxTud+FpypxakRXX/HnN/1zb+VXfDP/ACLNj/1yFVmUOhVhlWGCKzF0C1RQqS3SqOgE5AFKpCUmmicTRnUacTsKK5D+wrf/AJ73f/f9qP7Ct/8Anvd/9/2qPZzOT6pVOvorkP7Ct/8Anvd/9/2o/sK3/wCe93/3/aj2cw+qVTr6K5D+wrf/AJ73f/f9qP7Ct/8Anvd/9/2o9nMPqlU6+iuQ/sK3/wCe93/3/aj+wrf/AJ73f/f9qPZzD6pVOvorkP7Ct/8Anvd/9/2o/sK3/wCe93/3/aj2cw+qVTr6K5D+wrf/AJ73f/f9qP7Ct/8Anvd/9/2o9nMPqlU6+iuQ/sK3/wCe93/3/aj+wrf/AJ73f/f9qPZzD6pVOvorkP7Ct/8Anvd/9/2o/sK3/wCe93/3/aj2cw+qVTr6K5D+wrf/AJ73f/f9qP7Ct/8Anvd/9/2o9nMPqlU6+iuQ/sK3/wCe93/3/aj+wrf/AJ73f/f9qPZzD6pVOvorkP7Ct/8Anvd/9/2o/sK3/wCe93/3/aj2cw+qVTr65DQv+PKb/r5k/wDQqP7Ct/8Anvd/9/2q5Z2cVjbiG3BC5J+Y5JJqoQkpXZvh8POnPmkT1l6t/wAflh/10f8A9ANalZerf8flh/10f/0A1tLY9SG/3/kSaR/qbn/r6l/9CrQrP0j/AFNz/wBfUv8A6FWhRHYKnxMKKZJLHCu6V1RemWOBUX2+0/5+oP8Av4KuzZm2luWKKr/b7T/n6g/7+Cj7faf8/UH/AH8FHK+wuaPcsUVX+32n/P1B/wB/BR9vtP8An6g/7+CjlfYOaPcsUVX+32n/AD9Qf9/BR9vtP+fqD/v4KOV9g5o9yxRVf7faf8/UH/fwUfb7T/n6g/7+CjlfYOaPcsUVX+32n/P1B/38FH2+0/5+oP8Av4KOV9g5o9yxRVf7faf8/UH/AH8FH2+0/wCfqD/v4KOV9g5o9yxRVf7faf8AP1B/38FH2+0/5+oP+/go5X2Dmj3LFFV/t9p/z9Qf9/BR9vtP+fqD/v4KOV9g5o9yxRVf7faf8/UH/fwUfb7T/n6g/wC/go5X2Dmj3LFFV/t9p/z9Qf8AfwUfb7T/AJ+oP+/go5X2Dmj3LFFV/t9p/wA/UH/fwUfb7T/n6g/7+CjlfYOaPcsUVX+32n/P1B/38FH2+0/5+oP+/go5X2Dmj3LFFV/t9p/z9Qf9/BR9vtP+fqD/AL+CjlfYOaPcsUVX+32n/P1B/wB/BR9vtP8An6g/7+CjlfYOaPcsUVX+32n/AD9Qf9/BR9vtP+fqD/v4KOV9g5o9yxRVf7faf8/UH/fwUfb7T/n6g/7+CjlfYOaPcsUVX+32n/P1B/38FH2+0/5+oP8Av4KOV9g5o9yxRVf7faf8/UH/AH8FH2+0/wCfqD/v4KOV9g5o9yxRVf7faf8AP1B/38FH2+0/5+oP+/go5X2Dmj3LFFV/t9p/z9Qf9/BR9vtP+fqD/v4KOV9g5o9yxRVf7faf8/UH/fwUfb7T/n6g/wC/go5X2Dmj3LFFV/t9p/z9Qf8AfwUfb7T/AJ+oP+/go5X2Dmj3LFFV/t9p/wA/UH/fwUfb7T/n6g/7+CjlfYOaPcsUVX+32n/P1B/38FH2+0/5+oP+/go5X2Dmj3LFFV/t9p/z9Qf9/BUyOsiBo2DKehU5Bos0NNPYdWXq3/H5Yf8AXR//AEA1qVl6t/x+WH/XR/8A0A1EtjSG/wB/5Emkf6m5/wCvqX/0KtCs/SP9Tc/9fUv/AKFWhRHYKnxM53xPbxXeo6Jb3KCSGS6IdD0Py1d/4RPQv+gZB+Rqtr//ACGtB/6+z/6DXQ17WGbVFHzWNS9vL5fkY/8Awiehf9AyD8jR/wAInoX/AEDIPyNbFFdHMzjsjH/4RPQv+gZB+Ro/4RPQv+gZB+RrYoo5mFkY/wDwiehf9AyD8jR/wiehf9AyD8jWxRRzMLIx/wDhE9C/6BkH5Gj/AIRPQv8AoGQfkaZb65f36ySaZ4fvbuBJGjEqFcMVODU327Xf+hU1D81rP20e5fs32Gf8InoX/QMg/I0f8InoX/QMg/I0/wC3a7/0KmofmtH27Xf+hU1D81o9tHuHs32Gf8InoX/QMg/I0f8ACJ6F/wBAyD8jT/t2u/8AQqah+a0fbtd/6FTUPzWj20e4ezfYZ/wiehf9AyD8jR/wiehf9AyD8jT/ALdrv/Qqah+a0fbtd/6FTUPzWj20e4ezfYZ/wiehf9AyD8jR/wAInoX/AEDIPyNMu9a1WwtZLq98NX8NvEN0kjFcKPWtmKRZoUkT7rqGGfQ1UanNsxOFt0ZP/CJ6F/0DIPyNH/CJ6F/0DIPyNbFFVzMmyMf/AIRPQv8AoGQfkaP+ET0L/oGQfka2Kztc1KTS9PE0EKzSvKkSKzYGWOOTScmldsaim7EH/CJ6F/0DIPyNH/CJ6F/0DIPyNWv7P8a/9ArT/wDwKNH9n+Nf+gVp/wD4FGs/bruX7J9ir/wiehf9AyD8jR/wiehf9AyD8jVr+z/Gv/QK0/8A8CjR/Z/jX/oFaf8A+BRo9uu4eyfYq/8ACJ6F/wBAyD8jR/wiehf9AyD8jVr+z/Gv/QK0/wD8CjR/Z/jX/oFaf/4FGj267h7J9ir/AMInoX/QMg/I0f8ACJ6F/wBAyD8jVr+z/Gv/AECtP/8AAo1HPaeMre3kmk0rT9kal2xdHoBmj267h7J9iH/hE9C/6BkH5Gj/AIRPQv8AoGQfkauaTff2npNteFPLMyBiuc4NXK0Um1e5HKkY/wDwiehf9AyD8jR/wiehf9AyD8jWxRT5mKyMf/hE9C/6BkH5Gj/hE9C/6BkH5Gr+o3sem6dPeTKzJChYhepqjHqWtyxLJH4W1BkcBlIZeQaiVRR3ZShfZCf8InoX/QMg/I0f8InoX/QMg/I0/wC3a7/0KmofmtH27Xf+hU1D81pe2j3H7N9hn/CJ6F/0DIPyNH/CJ6F/0DIPyNP+3a7/ANCpqH5rR9u13/oVNQ/NaPbR7h7N9hn/AAiehf8AQMg/I0f8InoX/QMg/I0/7drv/Qqah+a0fbtd/wChU1D81o9tHuHs32Gf8InoX/QMg/I0f8InoX/QMg/I0/7drv8A0KmofmtS6Vqp1FrmOW0ltJ7WTy5YpcZU4z2pqqm7JidO2rRX/wCET0L/AKBkH5Gj/hE9C/6BkH5GtisjQ/Elpr8+pxWcc6Npt21pMZVADOvUrgnI+uPpVczva4uVWvYT/hE9C/6BkH5Gj/hE9C/6BkH5Gl8P+JLPxJHfPYxzxixvJLOTzlAy6YyRgnjn2+la9HM2r3DlS0Mf/hE9C/6BkH5Gj/hE9C/6BkH5GtiinzMVkY//AAiehf8AQMg/I1U8GKE0OREGFS6lVR6DdXR1zvg//kDTf9fc3/oVcuKbdL5ndgNK/wAmb9Zerf8AH5Yf9dH/APQDWpWXq3/H5Yf9dH/9ANePLY+jhv8Af+RJpH+puf8Ar6l/9CrQrP0j/U3P/X1L/wChVoUR2Cp8TMHX/wDkNaD/ANfZ/wDQa6Gue1//AJDWg/8AX2f/AEGuhr2cP/CR81jf48vl+QUUUV0HIFFFFABRRRQBN8Mv+RTk/wCv2f8A9Cq/qPi+G11i40vTdL1DWry0iWW7jsBF/owblQxkkQFmGSFUk4GcDIzQ+GX/ACKcn/X7P/6FVGzlm8GeNPEs2padqN1Z6xMl7aXNhZS3WSsao0TiNWKEFQQSApB65BFed1Oz+v6/rzNu28daNe3mhQWLzXK66szWsqIAq+UAXDgkMpGcYxnIIOK1LzVPseqafZfYb2f7czr58EO6K32ruzK2flB6Drk8V5CnhW706+8JTeJNCuby0e51W81C1gtWuktVnwyRuqAhuoG0A5OcZxS6RYajY6j8P7eaGazZr/VWsbe5Uh7eBopDCjKeRhdvy9hgdqL6bf12/rcLfr+uv4fie2VFc3MFlay3V5MkEEKF5JZGCqigZJJPQAV4r4E8O65baxo7Txz2Ws263P8Aac66HLGZ2OQftF482y4BcqylFY8ZwBk1Jp/hu2uvAN7pr+E9SHiptGuor+8ltHjjnn6/PISFuGZwGRhvx1ytEm1Ftf1uNJOST7nsS6jaPpa6l9ojSyaET+fIdiiPbu3EnGBjnmsnwl4z0jxtZXl5oDyS21pdNamZ02iRlVW3Lznb8w5OPpUfg2xsf+Fe6ZYrpgtrY2axTWc9mYPmxiQNE6g8tuzkfNnPOc1n/D2wurG88XG7tZrdbjxBPLCZYygkjKRgMueq8HkccVbSVRx6Wf5r/Mzi7wUuv/AZpePf+RC1f/r3P8xWFYf8g22/65L/ACFbvj3/AJELV/8Ar3P8xWFYf8g22/65L/IVpR+Jk1NixRRRXUYBWH4r/wCQda/9fsP/AKFW5WH4r/5B1r/1+w/+hVnU+BlR+JHd674jtdCe0geG4vL6+kMdpY2oUyzkDLEbmVQFHJZmAHrkgHJufiLpdjomp39/aX1tPpMkUd7p8iJ9oi8xgqHhyjKd2QysRgHuCKg8VWt1p/jrQfFEdncXtjaQT2d3HaxNLLCJShWVY1BZgCuG2gkA5wQDXGeNtH1TxHp/i/XrDStQ8i7trGysrY2zx3Fz5VwHeTyiA6gbsDcAcAngYJ4k9v6t/wAP+vkdVlf7v+D936eZ6zrGqf2Ro0+ofYry/wDJUN9msYvNmkyQMKuRk81dByoOCM9jXhfi7TxB4a8fXekaXc6Voktjap5M1q9sst0JP3kiowGflKAuBhiOpxmpvFPh7Wbvxfrr3cM7TXLWp0a8g0SW8mhQKMeTcLKkdsVfcWDlQcknINNO/wDX9eq8gtp/Xl/me31VsdTstT+0HT7mO4FtM1vKYzkLIuNy59Rnn0PHUV5vJpcKeO9Vk8V+H9S1i9lu7R9JvbS0c+VEFUfLOpCwhZA7Mpdcg/xZxWp8L9Gi0OXxNbvpLWN0+sTyeZ9jaJZoGYmLbJtCuoG7ABO3PbNEdfu/y/zJei+79f8AI2E8b2w1+DTLzStUsEurmS1tby7gVIbiVASVUbt4yASpZQGxwTxW3q3/ACBb3/r3k/8AQTXBWd+/ib4oC51vTtXs7XRpng0iCTS7kRzyFdr3LybNgGMqgJ6ZPUiu91b/AJAt7/17yf8AoJoWsU/6/r+txv4rf1/X9bHnvhT/AJFXT/8ArkP51r1keFP+RV0//rkP51r12w+FHNL4mFFFFWSZHiv/AJFTUP8Arkf513tncw2fhu3ubuVYYIbRZJJHOFRQmSSfQCuC8V/8ipqH/XI/zrp9d0abxF8L7vR7SRY573SzDGzHADGPjPtnrXHiG1qux00Uno+5HZ+PrK4ksHudM1KwsNTkEdjqF3HGsNwzfcAAcum8crvVc9OpAOpoPiK08QjUTZRzJ/Z99LYy+coG6SMgErgn5eeM4PtXD6pLfeL/AAzovhy30XUrG/jurV743NnJFFZrA6s5WUjZJkrhdjNndngZI5tNBumvtQSDQNQi16Txg9zY6i1lIFjtvNUyP52MLGUDDGRuzwD1qNOe3T/gx1/F/d5D+xzdf+BLT8Eex6Zqn9pSXq/Yb20+yXLW+bqHYJsAHzI+fmQ5wD7Gr9eFXenapqNtr4sIvtFjF42eTUYDaPdq8PlJjdAjBpUDlSUB7A44q9FoN/D4bs/tNle33h0eITcXOmLo8kCi22EDZaF5JDCJcPsIB7hMAVMXdJ+n4pf5/gypKzfz/C/+X4nrs+p2VtqFrYz3MaXd5u8iEn5pNoyxA9AOp6cj1FZniXxQnhqFZH0jVNSBjklkNhArLEiLuZnd2VRx0Gdx7A1wX9gaX/wmPgvU7HwpfWukwS3kYjurB5Gt3ZgYn2fM0Kb9zLuChM5wtdn43v7ZNJl0vUNL1i7tr+CRTLpttJMFYDIRhFlxu6crsIyGODgkm1G/UI2crM39PvrfVNNtr+xk822uolmhkAI3IwyDz7GvPLH/AJGzxL/1+j/0EV2XhGLU4PBukRa8saaklpGtykaKio+0ZAC/KMdMDj04rjbH/kbPEv8A1+j/ANBFbJJVLIyu3C7NWvIvCuj+I9S17xbJoPin+xYV1qZXi/s+O43tn72WORxxivXagtrC0smmaztYLdriQyzGKMKZHPVmx1Pua6HG8r/10M1K0bf11PAXfUrbwHf2Zu42jn8VywX9zOWihkXaM+Z5fKozdcHitDS7v/hHvBHiqNtRtbnTVaCJLfw/dTbIJHOGCTzKwAYY3EFsc9M17QNG0tbS4tRptoLe6cyXEIgXZMx6swxhicDk0RaNpcGmPp0Gm2kdi4Ie1SBRE2euUAwc/Ss/Zv8ABL8Evu0L9ovx/W54d4eSOw1TxdZabLpkdg/h6eU2+l3slzFu/hy7k7nAJBIwOelGkw6BbaX4N1HwvqDyeJ5ryCG5RLlnkMfIkR4yflQAYHAGAOo5r2S98KaVNo91ZWFjZ2Dz2ctmk0NqgMSPnIGMcZJOMjJpvh3wpp/h/T7FBb2099a2y25vxbKksigY68kDjpk01Bpryt+bf/AE5Ll+/wDJL/gm5XO+D/8AkDTf9fc3/oVdFXO+D/8AkDTf9fc3/oVRiv4XzR0YH+OvRm/WXq3/AB+WH/XR/wD0A1qVl6t/x+WH/XR//QDXjy2Po4b/AH/kSaR/qbn/AK+pf/Qq0Kz9I/1Nz/19S/8AoVaFEdgqfEzB1/8A5DWg/wDX2f8A0Guhrntf/wCQ1oP/AF9n/wBBroa9nD/wkfNY3+PL5fkFFFFdByBRRRQAUUUUATfDL/kU5P8Ar9n/APQq7CvLrbQ9Q09ZItM8Q3tpA8jSCJAuAWOTU32HXf8Aoa9Q/Ja4/Zz7HRzx7npdZsugadP4gh1qaKSS+t42jhZ7iQpEGGCVjLbAxHBYDJHeuG+w67/0NeofktH2HXf+hr1D8lo9nPsHPHuel0V5p9h13/oa9Q/JaPsOu/8AQ16h+S0ezn2Dnj3PS6K80+w67/0NeofktH2HXf8Aoa9Q/JaPZz7Bzx7nVePf+RC1f/r3P8xWFYf8g22/65L/ACFZl3ouq39pJa3nia/mglG142VcMPStiGIQwpEmdqKFGfQVpTjKLuyJyTWg+iiiugyCsPxX/wAg61/6/Yf/AEKtyqOr6YurWP2dpnhIdZFkTGVZTkVE03FpFRdnc9EorzT7Drv/AENeofktH2HXf+hr1D8lrm9nPsb80e53OtaDp/iG1jttWilmgSQSeUtxJGjkHIDhWAcZH3WyPatKvNPsOu/9DXqH5LR9h13/AKGvUPyWl7OfYOaPc9LorzT7Drv/AENeofktH2HXf+hr1D8lp+zn2Dmj3PS6qat/yBb3/r3k/wDQTXn/ANh13/oa9Q/JabJputyxNHJ4p1BkcFWBVeQaXs59g5o9w8Kf8irp/wD1yH8616rabYppumwWcTM6QoFDN1NWa64q0UjCWrCiiiqJMjxX/wAipqH/AFyP869C0n/kC2X/AF7x/wDoIri9SsU1LTZ7OVmRJkKll6iqMem61FEscfinUFRFCqAq8AVz1ISbujWEklqem011DxsjZAYYO0kH8xyK82+w67/0NeofktH2HXf+hr1D8lrP2c+xfPHud3o2h6foFm9tpcTxpJK00jSzPM8jt1ZnclmPuSa0K80+w67/ANDXqH5LR9h13/oa9Q/JaPZz7BzR7npdFeafYdd/6GvUPyWj7Drv/Q16h+S0ezn2Dnj3PS680sf+Rs8S/wDX6P8A0EUfYdd/6GvUPyWpNK0ptNa6lmu5bue6k8yWWUDJOMdqqEJKSbQpSi1ZGjRRRXUYBRRRQAUUUUAFc74P/wCQNN/19zf+hV0Vc74P/wCQNN/19zf+hVzYr+F80duB/jr0Zv1l6t/x+WH/AF0f/wBANalZerf8flh/10f/ANANePLY+jhv9/5Emkf6m5/6+pf/AEKtCs/SP9Tc/wDX1L/6FWhRHYKnxMwdf/5DWg/9fZ/9Broa57X/APkNaD/19n/0Guhr2cP/AAkfNY3+PL5fkFFFFdByBRRRQAUUUUAYh8WWHmOsUN5MEcoXitmZSRwcGk/4Su0/589Q/wDARq6L4Zf8inJ/1+z/APoVa2reLdH0W8+yXs073Ii854bSzmuXjjzje4iRiinnBbAODjoa5PbSOj2aOH/4Su0/589Q/wDARqP+ErtP+fPUP/ARq71fE2jvdaXbxX8csmro8liYgXWdUUMxDAFQACDyRntU15rVhYapp+nXc/l3epM62sexj5hRdzcgYGAM84o9rMXJE88/4Su0/wCfPUP/AAEaj/hK7T/nz1D/AMBGr1KkJABJOAOpNHtZD9nE8u/4Su0/589Q/wDARqP+ErtP+fPUP/ARq9PiljmhSWF1kjdQyOhyGB5BB7iqGm+INL1fUdRsdNu1nudMlWG7RVYeU5GQMkYPHoTjoaPazvYXJE8+fxdYRIXlt76NB953tWAH1rcR1kjV0OVYAg+orW8e/wDIhav/ANe5/mKwrD/kG23/AFyX+QrSnNydmTOKSuixRRRW5kFVdR1G30uzNzdswjBC/KpJJJwABVqsPxX/AMg61/6/Yf8A0Kpk7RbKiruwf8JXaf8APnqH/gI1H/CV2n/PnqH/AICNXoWsa5p+hW0c2pzmMSyCKKOOJ5ZJXP8ACkaAs5xk4UHgE9BVA+NvD6aFdavLfNFaWcgiufNt5UlgcsFCvEVDqSWXqvQg9Oa5vayNvZo4z/hK7T/nz1D/AMBGo/4Su0/589Q/8BGr0TWNZsNA0efVdXn+z2VuoaWXYzbQSB0UEnkjoKughlBHQ8ij2shckTy7/hK7T/nz1D/wEaj/AISu0/589Q/8BGr1KmJNHKzrFIrmNtjhWB2tgHB9Dgg/iKPazDkieYf8JXaf8+eof+AjUHxZZqpLWmoAAZJNo3Fdzb+K9Ku9en0e1e7mu7eXypjHYTtFE+0PtaYJ5YOCDy3cDrV7Vv8AkC3v/XvJ/wCgmj2srXH7ON7HHWl3DfWcVzbNuilUMpxjipqyPCn/ACKun/8AXIfzrXrpi7pMwas7BRRRVCIbu7hsbOW5uW2xRKWY4zxWSPFlmygraagQRkEWjc1L4r/5FTUP+uR/nXoOlsF0OzZiAot0JJPT5RWFSo4vQ1hFNHnP/CV2n/PnqH/gI1H/AAldp/z56h/4CNXZ2HjfQNTvoLW0vJC1yWW2lktZY4bkr1EUrKEk4BI2McgEjgVoaXrVhrQu/wCzbjzvsdy9pP8AIy7JU+8vIGcZ6jj3rP2sy/ZxPPP+ErtP+fPUP/ARqP8AhK7T/nz1D/wEavQ9M1qw1iS9TTp/OawuWtLkbGXZKoBK8gZ4YcjIq/R7WQezieW/8JXaf8+eof8AgI1H/CV2n/PnqH/gI1envNHHJGkkiq8hKorMAXOCcD14BP4Vna54j0vw7DHJq1w8fmlhHHFC80j7RuYhI1ZiABknGAOtJ1pLVh7NM4H/AISu0/589Q/8BGq9pmr22rRyta+YDE+x0kQqynr0Nd7ZXttqNhBe2MyT21xGskUqHIdSMgj8K87sf+Rs8S/9fo/9BFaRqS5rMlxjy3Rq0UUV0GIUUUUAFFFFABXO+D/+QNN/19zf+hV0Vc74P/5A03/X3N/6FXNiv4XzR24H+OvRm/WXq3/H5Yf9dH/9ANalZerf8flh/wBdH/8AQDXjy2Po4b/f+RJpH+puf+vqX/0KtCs/SP8AU3P/AF9S/wDoVaFEdgqfEzB1/wD5DWg/9fZ/9Broa57X/wDkNaD/ANfZ/wDQa6GvZw/8JHzWN/jy+X5BRRRXQcgUUUUAFFFFAE3wy/5FOT/r9n/9CrJ03V9O8H+P/FreLLyDTf7Tmiu7K6u3EaXMKxKhRGPBZGBBTr8wOOa1vhl/yKcn/X7P/wChV2Fef1udfkeB2lpHpmoeCT4ikuNGsbi61m5UNK1s1tBIQyBnBDRcEHOVIzgkVsaFrmuSz+BwNQvLiG5vtUSBppmze26Rv9naTpv4CkFs54PevZKxrvw1b33iqx1y7urmSTT0cWtqfLEUTOu1n4XeWI45Yj2pW0svT+vMd+vr+N/8/wADynwJrniG91jRy+rQyatItydWsJtVuLiViM8Pa+V5doVfaBhlGDt5yKILqz1z4cajHL4h1e88TT6HdtqWl/a5ZFSYcsHj5EBVhtVRsDDIwwr2+iiUeZNf11GpWkmcZoNzPB8HbOfwco1e7XTl+yo155oebbggyO/RWzkbhjbtGMAVzvwkgew8X+MbFtMu7QrJaNK1y0JYyeSNxfy3YFnJL5GRyckHivVaKtu83LuZpWio9jnvHv8AyIWr/wDXuf5isKw/5Btt/wBcl/kK3fHv/Ihav/17n+YrCsP+Qbbf9cl/kK0o/EyamxYooorqMArD8V/8g61/6/Yf/Qq3Kw/Ff/IOtf8Ar9h/9CrOp8DKj8SNbxfImj/Ebwz4g1YiPRreC5tZLp/9XaTSbNjueihgpXceASASM1wvj6VNXsfHHiDSJPM0aW1061W7g+aO6mjuAWaMjIfaGC5GRnI5wa92orjStbyOq+t/T8P+GPEPEOu3Nto3jpvC2vX1zpdrY2rx3QvpJTbXbSESJHISWX5dpKA4UnGBnFL4q1zXk8Ya7BLq0emXULWo0UTarcQblIBzHaRROLvc+5WyCf4eMCvVvE3huDxVpP8AZt9d3MNm7q08UHl4uFBB2MWViFyB90qfetcDAAHQUkrf1/XovIL6W/rp/keXSarC3jvVYPFniHUdJv4bu0XSbSyuZFE0TKpykABWYNIXViUYqB1XGau/CW3tbN/FVslxIb2PXbnzreS7eRo0LkxsUZjt3DPzYBbHJOK9Fopx0+635f5EvVW9P1/zPJbX+zdM+IVjD4L1e+u9RvdauZNdspp3+SMoSzvAQqoFYRhH2gkYAZs16fq3/IFvf+veT/0E1bqpq3/IFvf+veT/ANBNC0jYb1lc898Kf8irp/8A1yH8616yPCn/ACKun/8AXIfzrXrth8KOaXxMKKKKskyPFf8AyKmof9cj/Ouj8Sade6v8J7/T9Kyby50lo4VBxuYx4259+n41zniv/kVNQ/65H+dehaT/AMgWy/694/8A0EVx4iPNp5HRRfLqeZaxrOmeKfCHhzQfDMiSaxHe2TfYk4m04QupkaVOsYQKV+YDJIAzkVz1nqAsr7VRpOq3UXiB/GsiW1hHcuq3ETSqJcw/ddQm4lyDtx1HSveaa6lo2VWKEjAZcZHvzxUX97m87/l/l+JX2OX+tn/meJ6rr+vx2msbL+b7AnjJrW7mmv5Ldbe28tCqGdQzQRl9oLKBjOONxqeLWtTXw3Ztf67nw5J4hMM+oWeo3EnkWuwkIbxkjZk87C+aCeDtL9a9Q8O+HLbw3a3UdvcXF1LeXT3dzc3OzfLI2MkhFVRwAMADpWvUxVkl6fgl/l+JUndv5/jf/P8AA8ekXRZfH3gW+t9d1C90wvfQ299eXsqb5AV2RrIdvmKWyoJ3eYAAS4Fa/wARjLp/jTRNXm1s6FZR6feQi/2RkRzHYyp+8DLlgh4IyQpA5Oa9KooaurLz/H/hwTs9TI8J3+oap4Q0q+1q3+zahcWsclxCUKbHK8jaeR9K4ux/5GzxL/1+j/0EV6XXmlj/AMjZ4l/6/R/6CK1i71LmbVoWNWiiiuw5wooooAKKKKACud8H/wDIGm/6+5v/AEKuirnfB/8AyBpv+vub/wBCrmxX8L5o7cD/AB16M36y9W/4/LD/AK6P/wCgGtSsvVv+Pyw/66P/AOgGvHlsfRw3+/8AIk0j/U3P/X1L/wChVoVn6R/qbn/r6l/9CrQojsFT4mYOv/8AIa0H/r7P/oNdDXOeInWPV9CaRgqi7OSxwB8tbf2+z/5+4P8Av4K9rDfwUfNY3+PL5fkWKKr/AG+z/wCfuD/v4KPt9n/z9wf9/BW5yFiiq/2+z/5+4P8Av4KPt9n/AM/cH/fwUAWKKr/b7P8A5+4P+/go+32f/P3B/wB/BQBm6Zb+JtDt5LXStTs1tmmeVRJblm+Y561c/tDxr/0FdP8A/AU1N9vs/wDn7g/7+Cj7fZ/8/cH/AH8FZexiX7RkP9oeNf8AoK6f/wCApo/tDxr/ANBXT/8AwFNTfb7P/n7g/wC/go+32f8Az9wf9/BR7GI/aMh/tDxr/wBBXT//AAFNH9oeNf8AoK6f/wCApqb7fZ/8/cH/AH8FH2+z/wCfuD/v4KPYxD2jIf7Q8a/9BXT/APwFNH9oeNf+grp//gKam+32f/P3B/38FH2+z/5+4P8Av4KPYxD2jM/Ux4t1fTJ7C81SwMFwuxwtsQcfWtS3i8i2jizu8tAucdcDFR/b7P8A5+4P+/go+32f/P3B/wB/BVRpqLuiXNvcsUVX+32f/P3B/wB/BR9vs/8An7g/7+CrJLFZ2uabLqmniG3mWGVJUlRnXIypzzVn7fZ/8/cH/fwUfb7P/n7g/wC/gpNXVmCdnch/tDxr/wBBXT//AAFNH9oeNf8AoK6f/wCApqb7fZ/8/cH/AH8FH2+z/wCfuD/v4Kz9jE09oyH+0PGv/QV0/wD8BTR/aHjX/oK6f/4Cmpvt9n/z9wf9/BR9vs/+fuD/AL+Cj2MQ9oyH+0PGv/QV0/8A8BTR/aHjX/oK6f8A+Apqb7fZ/wDP3B/38FH2+z/5+4P+/go9jEPaMh/tDxr/ANBXT/8AwFNRz3fjK4t5IZNV0/ZIpRsWp6EYq19vs/8An7g/7+Cj7fZ/8/cH/fwUexiHtGR6RYnTNItrNnEhhQKWAxmrlV/t9n/z9wf9/BR9vs/+fuD/AL+CtErKxm3d3LFFV/t9n/z9wf8AfwUfb7P/AJ+4P+/gpgR6vYnU9IubNXEZmQqGIzg1DBd+Mre3jhj1XT9kahFzanoBirX2+z/5+4P+/go+32f/AD9wf9/BUSpqTuylNrYh/tDxr/0FdP8A/AU0f2h41/6Cun/+Apqb7fZ/8/cH/fwUfb7P/n7g/wC/gqfYxK9oyH+0PGv/AEFdP/8AAU0f2h41/wCgrp//AICmpvt9n/z9wf8AfwUfb7P/AJ+4P+/go9jEPaMh/tDxr/0FdP8A/AU0f2h41/6Cun/+Apqb7fZ/8/cH/fwUfb7P/n7g/wC/go9jEPaMh/tDxr/0FdP/APAU1FpGn3lrcX11qVxHPc3s3muYk2qOMdKt/b7P/n7g/wC/go+32f8Az9wf9/BTVJJ3QnNtWLFFV/t9n/z9wf8AfwUfb7P/AJ+4P+/grQgsUVX+32f/AD9wf9/BR9vs/wDn7g/7+CgCxRVf7fZ/8/cH/fwUfb7P/n7g/wC/goAsVzvg/wD5A03/AF9zf+hVtfb7P/n7g/7+CsXwcQdFlIOQbuXBH+9XNiv4XzO3A/x16M36y9W/4/LD/ro//oBrUrL1b/j8sP8Aro//AKAa8eWx9HDf7/yJNI/1Nz/19S/+hVoVn6R/qbn/AK+pf/Qq0KI7BU+JlW+06z1KJY7+3SdFO5Q46GqX/CK6H/0DIPyNa9FaqpOKsmYSpU5O8opmR/wiuh/9AyD8jR/wiuh/9AyD8jWvRT9rU/mf3k+wpfyr7jI/4RXQ/wDoGQfkaP8AhFdD/wCgZB+RrXoo9rU/mf3h7Cl/KvuMj/hFdD/6BkH5Gj/hFdD/AOgZB+RrXoo9rU/mf3h7Cl/KvuMj/hFdD/6BkH5Gj/hFdD/6BkH5Gteij2tT+Z/eHsKX8q+4yP8AhFdD/wCgZB+Ro/4RXQ/+gZB+RrXoo9rU/mf3h7Cl/KvuMj/hFdD/AOgZB+Ro/wCEV0P/AKBkH5Gteij2tT+Z/eHsKX8q+4yP+EV0P/oGQfkaP+EV0P8A6BkH5Gteij2tT+Z/eHsKX8q+4yP+EV0P/oGQfkaP+EV0P/oGQfka16KPa1P5n94ewpfyr7jI/wCEV0P/AKBkH5Gj/hFdD/6BkH5Gteij2tT+Z/eHsKX8q+4yP+EV0P8A6BkH5Gj/AIRXQ/8AoGQfka16KPa1P5n94ewpfyr7jI/4RXQ/+gZB+Ro/4RXQ/wDoGQfka16KPa1P5n94ewpfyr7jI/4RXQ/+gZB+Ro/4RXQ/+gZB+RrXoo9rU/mf3h7Cl/KvuMj/AIRXQ/8AoGQfkaP+EV0P/oGQfka16KPa1P5n94ewpfyr7jI/4RXQ/wDoGQfkaP8AhFdD/wCgZB+RrXoo9rU/mf3h7Cl/KvuMj/hFdD/6BkH5Gj/hFdD/AOgZB+RrXoo9rU/mf3h7Cl/KvuMj/hFdD/6BkH5Gj/hFdD/6BkH5Gteij2tT+Z/eHsKX8q+4yP8AhFdD/wCgZB+Ro/4RXQ/+gZB+RrXoo9rU/mf3h7Cl/KvuMj/hFdD/AOgZB+Ro/wCEV0P/AKBkH5Gteij2tT+Z/eHsKX8q+4yP+EV0P/oGQfkaP+EV0P8A6BkH5Gteij2tT+Z/eHsKX8q+4yP+EV0P/oGQfkaP+EV0P/oGQfka16KPa1P5n94ewpfyr7jI/wCEV0P/AKBkH5Gj/hFdD/6BkH5Gteij2tT+Z/eHsKX8q+4yP+EV0P8A6BkH5Gj/AIRXQ/8AoGQfka16KPa1P5n94ewpfyr7jI/4RXQ/+gZB+Ro/4RXQ/wDoGQfka16KPa1P5n94ewpfyr7jI/4RXQ/+gZB+Ro/4RXQ/+gZB+RrgdWm1q+i8frp+qX0U+m3EM1qI7hxsCx7mRQDwCM8dDWpruvT+IrTwlYaLdzW0utSpczSW8jRskCLukGVORk8VSqVHb3n0/FX/AM/uMeWjr7i69Ozt/l951X/CK6H/ANAyD8jWhaWdvYW4gs4UhiUkhEGBzXlfhjXtRtPiNr8uo6jdXGn/AGm9gjglnZkiMW2QbQTgfLvHHpVv4SX+r3Wsa/FrOoXV2yx2kyJPO0gi81GcqoJ4HIHHoKXNOa1fS5UfZwl7sbO7X3HqFZerf8flh/10f/0A1qVl6t/x+WH/AF0f/wBANYS2O2G/3/kSaR/qbn/r6l/9CrQrJD3uk3F1bTae7N9pkbKyp3Y+9O/ta4/6Bsn/AH9T/GhOyKlFt3X5o1KKy/7WuP8AoGyf9/U/xo/ta4/6Bsn/AH9T/Gi6J5H/AE0alFZf9rXH/QNk/wC/qf40f2tcf9A2T/v6n+NF0HI/6aNSisv+1rj/AKBsn/f1P8aP7WuP+gbJ/wB/U/xoug5H/TRqUVl/2tcf9A2T/v6n+NH9rXH/AEDZP+/qf40XQcj/AKaNSisv+1rj/oGyf9/U/wAaP7WuP+gbJ/39T/Gi6Dkf9NGpRWX/AGtcf9A2T/v6n+NH9rXH/QNk/wC/qf40XQcj/po1KKy/7WuP+gbJ/wB/U/xo/ta4/wCgbJ/39T/Gi6Dkf9NGpRWX/a1x/wBA2T/v6n+NH9rXH/QNk/7+p/jRdByP+mjUorL/ALWuP+gbJ/39T/Gj+1rj/oGyf9/U/wAaLoOR/wBNGpRWX/a1x/0DZP8Av6n+NH9rXH/QNk/7+p/jRdByP+mjUorL/ta4/wCgbJ/39T/Gj+1rj/oGyf8Af1P8aLoOR/00alFZf9rXH/QNk/7+p/jR/a1x/wBA2T/v6n+NF0HI/wCmjUorL/ta4/6Bsn/f1P8AGj+1rj/oGyf9/U/xoug5H/TRqUVl/wBrXH/QNk/7+p/jR/a1x/0DZP8Av6n+NF0HI/6aNSisv+1rj/oGyf8Af1P8aP7WuP8AoGyf9/U/xoug5H/TRqUVl/2tcf8AQNk/7+p/jR/a1x/0DZP+/qf40XQcj/po1KKy/wC1rj/oGyf9/U/xo/ta4/6Bsn/f1P8AGi6Dkf8ATRqUVl/2tcf9A2T/AL+p/jR/a1x/0DZP+/qf40XQcj/po1KKy/7WuP8AoGyf9/U/xo/ta4/6Bsn/AH9T/Gi6Dkf9NGpRWX/a1x/0DZP+/qf40f2tcf8AQNk/7+p/jRdByP8Apo1KKy/7WuP+gbJ/39T/ABo/ta4/6Bsn/f1P8aLoOR/00alFZf8Aa1x/0DZP+/qf40f2tcf9A2T/AL+p/jRdByP+mjUorL/ta4/6Bsn/AH9T/Gj+1rj/AKBsn/f1P8aLoOR/00alc94u0nWtStbOXw1qS2N9Z3CzBJXcQ3C90k28lfwP4ZyLv9rXH/QNk/7+p/jR/a1x/wBA2T/v6n+NHMhOm2rP80ZfhDw1faUuq3niCa2n1HV5/NuVtQ3koANqqu7k8evrjtk5XgjwHqHhvXZ7nUrq3ntbaF7bS0iZi0UTytI27IA3cqOM11P9rXH/AEDZP+/qf40f2tcf9A2T/v6n+NVz63/rt+RHsFa3z3Xr+ZwmpfDTWLuy1lba9tIbi91iS8gkDt8sMiNG6t8vDFWPTP1rpvDfha50Txd4g1J3gNpqP2cWyRsSyCNCp3AjA9sE1q/2tcf9A2T/AL+p/jR/a1x/0DZP+/qf40Kdlb+un+QOhd3+e68/8zUrL1b/AI/LD/ro/wD6AaP7WuP+gbJ/39T/ABqL/TNW1K0SOyaPy2ZiWkU/wn3qW76I1UeXVvv1XY//2Q==)

Thread stack and heap in JVM <src - http://tutorials.jenkov.com/java-concurrency/java-memory-model.html>

|  |  |
| --- | --- |
| Sleep | thread.sleep(x);  //x is time in milliseconds //throws InterruptedException |
| Interrupt | thread.interrupt();  //raises the InterruptedException in the thread (must be running) |
| Join | anotherThread.join();  //should be inside the run method of a thread //the thread will then wait for anotherThread to terminate  //throws InterruptedException |
| Join timeOut | anotherThread.join(timeOut) //the thread will wait until either the anotherThread terminates or timeOut milliseconds elapse |
| Set name | new Thread(“thread1”); thread.setName(“thread1”); |
| Get name | thread.getName(); |
| Current thread | Thread.currentThread();  //returns reference to the Thread instance executing the currentThread() |
| Set priority | thread.setPriority(x);  //x is an integer //setting priority does not guarantee thread execution order, it is a mere suggestion to OS |

### Race Condition

The interference between the threads accessing resources on the shared memory, and one or more of the threads write to the resource. Race conditions do not occur when multiple threads only read from the same resource.

**Slipped Condition**

Slipped conditions means, that from the time a thread has checked a certain condition until it acts upon it, the condition has been changed by another thread so that it is errornous for the first thread to act. It is special kind of race condition.

These conditions can be prevented using proper synchronization.

## Synchronization

The process of controlling the threads’ access. Instance methods and statements can be synchronized meaning they can be only access by one thread at a time. But we can not synchronize constructor.

public **synchronized** void methodX() { };

**synchronized**(var) { statementY; }

Here, statementY will be synchronized. And var should be an instance variable (stored on the heap) but not a local variable (stored on thread stack). If a local variable is used, any thread can gain access to the statement using their copy of the local variable.

Static method can also be synchronized and static object can be used as var in synchronized statements.

**Critical section** – shared region of code which should only be accessed by one thread at a time

**Thread safe** – the region where all the critical section has been synchronized

*Note – Only synchronized the code that must be synchronized. Synchronizing unnecessary code sections decreases the efficiency. Synchronized the code region that writes data. The code section where the data is only read does not need to be synchronized.*

Only one synchronized block can be executed at a time using thread in an object, since only one thread can hold the key for lock.

### Thread Signals

wait(), notify() and notifyAll()

A thread that calls wait() on any object becomes inactive until another thread calls notify() on that object. In order to call either wait() or notify() the calling thread must first obtain the lock on that object. In other words, the calling thread must call wait() or notify() from inside a synchronized block. The notifyAll() method will wake all threads waiting on a given object.

Once a thread calls wait() it releases the lock it holds on the monitor object. This allows other threads to call wait() or notify() too, since these methods must be called from inside a synchronized block. Once a thread is awakened it cannot exit the wait() call until the thread calling notify() has left its synchronized block. In other words: The awakened thread must reobtain the lock on the monitor object before it can exit the wait() call.

*Note: wait() releases the lock of the thread associated with the object that the wait() was called upon, whereas sleep() does not do so.*

**Spurious Wakeups**

For inexplicable reasons it is possible for threads to wake up even if notify() and notifyAll() has not been called. This is known as spurious wakeups. Wakeups without any reason.

To guard against spurious wakeups the signal member variable is checked inside a while loop instead of inside an if-statement. Such a while loop is also called a spin lock. The thread awakened spins around until the condition in the spin lock (while loop) becomes false.

### Drawbacks of synchronized block

* A thread waiting to acquire the lock for a synchronized block can not be interrupted and may cause problems.
* The synchronized block must be within the same method.
* We can not check if a lock of an object is available or check any other information about the lock.
* If multiple threads are waiting for the lock, there is no guaranteed order in which the lock will be assigned.

## java.util.Concurrent

Helps to properly synchronize codes and work with multiple threads.

Lock interface provides a way for threads to acquire lock using lock() method and pass the lock using unlock() method.

### ReentrantLock

It is an implementation of Lock interface. Reentrant means the thread holding the lock can enter another region that requires the same lock.

|  |  |
| --- | --- |
| ReentrantLock myLock = new ReentrantLock(); | Create the shared lock |
| myLock.lock(); | Gain the lock to access the critical region |
| myLock.unlock(); | Release the lock once done |
| myLock.tryLock(); myLock.tryLock(timeOut, TimeUnit); | Check and try to gain the lock – returns a boolean //tries for the given time |
| ReentrantLock myLock = new ReentrantLock(true); | Fairness – the thread waiting the longest will be given the lock |

Exiting the synchronized block automatically release the lock whereas here the lock has to be released manually by calling the unlock() method.

The number of successful lock() method calls for a thread must be equal to the number of unlock() method calls. In other words, the number of times a thread acquires the same lock must be equal to the number of times it will need to unlock it. The ReentrantLock class maintains a lock count for it.

To avoid missing out the unlock() method call, it is recommended to put unlock() method call in the **finally** block (try-finally) so that it is always executed.

**>>> Program 11.1\_Threads <<<**

### ExecutorService

The ExecutorService interface extends Executor interface which only has one method – execute(). Executor interface is a simple interface to support launching new tasks. ExecutorService adds features to manage the lifecycle, both of the individual tasks and of the executor itself.

The ExecutorService implementations provided by Java are used to create and manage the threads.

Thread pool   
It is a managed set of threads which reduces the overhead of the thread creation and optimization. It can also limit the number of threads that can be created within an application. It is used through an implementation of ExecutorService interface.

* **Constructor**  
  ExecutorService exe = Executors.newFixedThreadPool(x);  
  //x is the number of active threads
* **Execute**  
  exe.execute(Runnable r);  
  //r describes the task to perform
* **Orderly shutdown of the executive service** (waits for current tasks to complete)exe.shutdown();  
  //if not called, the application will not terminate
* **Return a result**   
  Future<T> submit(Callable<T> task)  
  //executes the task and returns a Future which represents the result of the task

A Java Future, java.util.concurrent.Future, represents the result of an asynchronous computation. When the asynchronous task is created, a Java Future object is returned. The result in the Future object can be accessed using get() method.

Runnable vs. Callable

The Runnable interface is very similar to the Callable interface. th interfaces represents a task that can be executed concurrently by a thread or an ExecutorService. Both interfaces only has a single method. The main difference between the Runnable run() method and the Callable call() method is that the call() method can return an Object from the method call. Another difference between call() and run() is that call() throws an exception, whereas run() does not.

*public interface Callable{  
 public Object call() throws Exception;  
}*

### A BlockingQueue with one thread putting into it, and another thread taking from it.Blocking Queues Interface

A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely. The blocking queues implementations are thread-safe.

A BlockingQueue <src - http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html>

**Implementations**

* ArrayBlockingQueue - a bounded FIFO blocking queue backed by an array
* DelayQueue - a time-based scheduling queue backed by a heap
* LinkedBlockingQueue - an optionally bounded FIFO blocking queue backed by linked nodes
* PriorityBlockingQueue - an unbounded blocking priority queue backed by a heap
* SynchronousQueue - a queue that can only contain a single element internally

*Note: The thread-safe queue methods guarantuee that the methods will complete before another thread can call it. However, the thread might be suspended between two different method calls by a thread and cause problems. So, synchronized blocks should be used to call a sequence of queue methods from a thread.*

## Deadlocks, Starvation, and Livelocks

### Deadlocks

Thread A -> has lock X -> wants lock Y  
Thread B -> has lock Y -> wants lock X

A deadlock is when two or more threads are blocked waiting to obtain locks that some of the other threads in the deadlock are holding.

Deadlock prevention

* **Lock ordering**If all the locks are always taken in the same order by any thread, deadlocks cannot occur.
* **Lock timeout**Put a timeout on lock attempts meaning a thread trying to obtain a lock will only try for so long before giving up.

### Starvation

If a thread is not granted CPU time because other threads grab it all and the thread is unable to make progress, it is called "starvation". In a deadlock, no process proceeds for execution. Meanwhile, in startvation, threads with higher priorities continuously uses the resources preventing low priority threads to acquire the resources.

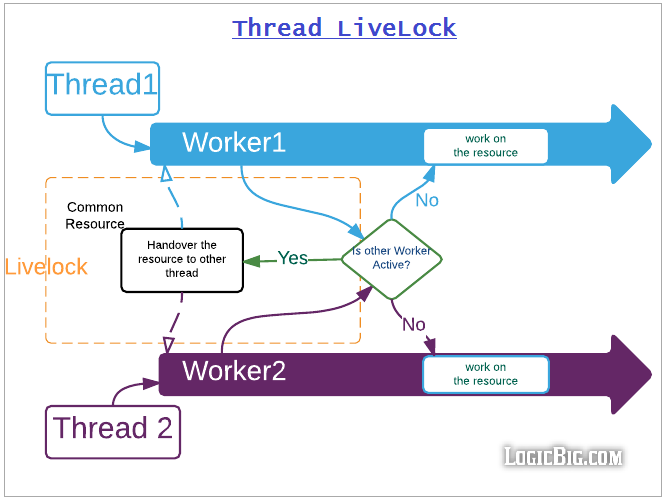
Main causes of Starvation in Java

* Threads with high priority swallow all CPU time from threads with lower priority.
* Threads are blocked indefinately waiting to enter a synchronized block, because other threads are constantly allowed access before it.
* Threads waiting on an object (called wait() on it) remain waiting indefinitely because other threads are constantly awakened instead of it.

Fair Locks

All these occur because the order of thread execution is not guaranteed in Java (no first-come-first-serve). The solution to starvation is called "fairness" - that all threads are fairly granted a chance to execute. For example – a fair reentrant lock. It guarantees first-come-first-serve order when getting a lock. But there is a performance cut-off.

### Live Locks

Livelock in Java is a situation where two or more threads are acting on a response to an action of each other and not able to make any progress because of that.

Live lock <src - https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/thread-livelock.html>

This is comparable to two people attempting to pass each other in a corridor: Ram moves to his left to let Shyam pass, while Shyam moves to his right to let Ram pass. Seeing that they are still blocking each other, Ram moves to his right, while Shyam moves to his left. They're still blocking each other, and they repeat.

Unlike deadlock, threads are not blocked when livelock occurs. They are simply too busy responding to each other to resume work.

## Other thread issues

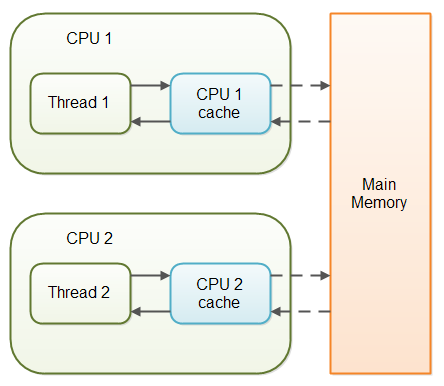
### Atomic action

An atomic action can’t be suspended in the middle of being executed. The System.println() method is not atomic action as a thread can be suspended in the middle of executing it. It might be during the evaluation of the arguments or before printing the result.

**Operations that are atomic:**

* Reading and writing reference variables  
  myObject1 = myObject2;
* Reading and writing primitive variables (except long and double)
* Reading and writing all variables declared volatile

### Volatile

Threads running on different CPU might have different values for the same variable in their CPU cache. And this might have to undesired result.

The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

Threads running on the volatile variables can still be suspended if the action takes multiple steps to complete. For example, x++ requires reading the variable from main memory, incrementing it, and then writing it back to the memory. The thread can be suspended in between these steps. So, the code must be synchronized despite using the volatile keyword.

<src - http://tutorials.jenkov.com/java-concurrency/volatile.html>

A common use of volatile is with variables of type long and double.   
private volatile double num;

**Alternative** to using volatile for an integer, an AtomicInteger class from java.util.concurrent.atomic package can be used which support lock-free thread-safe programming on single variables. There are Atomic classes for the following data types:  
boolean, integer, integer array, long, long array, object reference, and double

# LAMBDA EXPRESSIONS

Lambda expressions provide easier way to implement interfaces with only one method. Such interfaces with single abstract method are called functional interfaces. For examples:   
 - java.lang.Runnable interface which only has run() method  
 - java.util.concurrent.Callable interface which only has call() method  
 - java.util.Comparator<T> which only has compare(T,T) method

A Java lambda expression can be passed around as if it was an object and executed on demand. It facilitates functional programming, and simplifies the development a lot. Lamda expression was introduced in Java 8.

**Starting thread with Runnable implementation**

//without lambda expression*new Thread(new Runnable(){  
 @Override  
 public void run(){  
 System.out.println(“Runnable it is!”);  
 }  
}).start();*

//with lambda expression *new Thread( () -> System.out.println(“Runnable it is!”) ).start();*

Syntax(parameters) -> { body }

*() -> System.out.println(“printing...”);*

*x -> x+1; //returns x+1*

*(x, y) -> {  
 int z = x + y;  
 System.out.println(z);  
 return z;  
}*

The data type of parameters are not required to be specied, but they should match the expected parameter types in the method of the interface.

The lamda expressions are similar to anonymous class, but can be stored and resused as follows:

*Runnable r = () -> System.out.println(“Runnable running!”) ;*

*new Thread(r).start();  
new Thread(r).start();*

|  |  |
| --- | --- |
| **Anonymous Class** | **Lambda Expression** |
| It is a class without a name | It is a nested block of code |
| It can be created for any interface | It can only be created for interface with single method |
| ‘this’ keyword resolves to anonymous class itself | ‘this’ keyword resolves to the enclosing class where the lambda expression is written |
| It can access local variables only if they are final | It can access local variables only if they are final or effectively final |
| It is can be stored and reused | |

//Incorrect way *Employee employee; //Employee is a class  
for(int i=0; i<employees.size(); i++) { //employees is an ArrayList of Employee  
 employee = employees.get(i);  
 new Thread( () -> System.out.println(employee.getName()) ).start();   
}*

Here, a single employee object is used to iterate over employees and its value changes in each iteration. Since it is neither final nor effectively final, it can not be used in the lamda expression.

//Correct way *for(int i=0; i<employees.size(); i++) {   
 Employee employee = employees.get(i);  
 new Thread( () -> System.out.println(employee.getName()) ).start();   
}*

Now, new employee object is created in each iteration whose value does not change in the iteration. Even though it is not final, it can be used in the lamda expression because it is effectively final.

## Functional Interfaces

Any interface with a SAM(Single Abstract Method) is a functional interface, and can be implemented using lambda expression.

All functional interfaces are recommended to have an informative @FunctionalInterface annotation. This not only clearly communicates the purpose of this interface, but also allows a compiler to generate an error if the annotated interface does not satisfy the conditions.

A functional interface may still have multiple default methods. These methods have default implementations and do not violate SAM property.

java.util.function

This package contains functional interfaces designed for commonly occurring use cases, so you don’t have to create your own functional intefaces for every little use case. Some of them are as follows:

### Consumer<T>

Accepts a single input argument and returns no result. Unlike most other functional interfaces, Consumer is expected to operate via side-effects.

**void accept(T t);** //performs this operation on the given argument

* DoubleConsumer - void accept(double value)
* IntConsumer - void accept(int value)

It is always recommended to use specific Predicate interface provided in the java.util.function package if possible.

Example: *Consumer<Integer> consumer = (value) -> System.out.println(value);  
consumer.accept(10); //prints 10*

void forEach(Consumer<? super T> action)forEach() is used to iterate over items in a class that implements iterable<T> interface. For example: all the classes in Collections framework implement iterable.

*myArrayList.forEach( x -> System.out.println(x) );*

Here, we are iterating over each elements in myArrayList and printing them out. forEach() method accepts an implementation of Consumer interface which is achieved using lambda expression. And the accept() method in Consumer class is implementation using the body of the lambda expression which only prints the value of x.

### Predicate<T>

Takes a single value as parameter, and returns true or false.

**boolean test(T t);** //performs specified operation and returns a boolean

The Predicate interface contains more methods than the test() method, but the rest of the methods are default or static methods which you don't have to implement.

*//A method that prints the elements of a given list using a predicate  
printNumber(List<Integer> list, Predicate<Integer> condition) {  
 for(Integer i : list) {  
 if(condition.test(i))  
 System.out.println(i);  
 }  
}*

*//prints even number [Using Lamda Expression]  
printNumber(myArrayList, element -> element%2 == 0 );*

*//prints odd number [Using Anonymous Class]   
printNumber(myArrayList, new Predicate<Integer>() {  
 @Override()  
 public boolean Test(Integer element) {  
 return element%2 != 0;  
 }   
 );*

Predicate chaining

Multiple predicates can be used to test values using and() method.

*IntPredicate greaterThan5 = i -> i > 5;  
IntPredicate lesserThan10 = i -> i < 10;  
System.out.println( greaterThan5.and(lesserThan10).test(15) ); //returns false*

### Supplier<T>

Takes no argument and supplies some value.

**T get();** //performs operation to generate T

* int getAsInt(); - IntSupplier
* double getAsDouble(); - DoubleSupplier
* boolean getAsBoolean(); - BooleanSupplier

//A random number (0-10) supplier  
Supplier<Integer> supplier = () -> (int) (Math.random() \* 10);  
System.out.println( supplier.get() );

### Funtion<T, R>

Takes one argument and returns one argument.

**R apply(T t);** //performs operation on T and returns R

* DoubleFunction<R> - R apply(double value)
* IntFunction<R> - R apply(int value)
* ToDoubleFunction<T> - double applyAsDouble(T value)
* ToIntFunction<T> - int applyAsInt(T value)
* DoubleToIntFunction - int applyAsInt(double value)
* IntToDoubleFunction - double applyAsDouble(int value)

Function chaining

Functions can also be chained together using andThen() method if the return type of first function is same as the parameter type of second function.

function1.andThen(function2).andThen(function3).apply(T value);

BiFunction<T, U, R>

Takes in two arguments T & U, and returns one argument R.

**R apply(T t, U u);**

* BiConsumer <T, U>
* BiPredicate <T, U>

These can only be the first function in method chaining.

### UnaryOperator<T>

Takes in one argument and returns one result of same type. It extends from Function<T, T>. So, it can be chained using andThen().

**T apply(T t);**

* IntUnaryOperator - int applyAsInt(int value)
* DoubleUnaryOperator - double applyAsDouble(double value)

BinaryOperator<T>

It takes two arguments of same type and returns one result of same type. It is equivalent to BiFunction<T,T,T>

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Interface** | **Functional method** | **Arguments** | **Return** | **Can be chained** |
| Consumer | accept() | T | - | YES |
| Supplier | get() | - | R | NO |
| Predicate | test() | T | boolean | YES |
| Function | apply() | T | R | YES |
| UnaryOperator | apply() | T | T | YES |
| Bi [Prefix] | - | T, U | R | Depends |
| Int [Prefix] | - | int | R | YES |
| Double [Prefix] | - | double | R | YES |

*Note: Runnable interface does not fall under any of these interfaces because its lambda does not take any argument and does not return any value.*

## Stream (java.util.stream)

Stream represents a sequence of objects from a source, which supports sequential and parallel aggregate operations.

Stream properties

* **Source** – It takes Collections, Arrays, or I/O resources as input source.
* **Non interfaring** – Stream operations do not modify the original source
* **Pipelining** − Most of the stream operations return stream itself so that their result can be pipelined.
* **Automatic iterations** − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

**Generating stream**

With Java 8, Collection interface has two methods to generate a Stream:

* stream() − Returns a sequential stream considering collection as its source.
* parallelStream() − Returns a parallel Stream considering collection as its source.

Stream of any object:

* of()  
  Stream<String> strStream = Stream.of(“abc”, “xyz”, “ooo”);

Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

**Stream Pipeline**

*myList.stream()  
 .map(String::toUpperCase)  
 .filter(s -> s.startsWith(“A”))  
 .sorted()  
 .forEach(System.out::println);*

Each stream operation works on the stream returned from previous step. And each of them are lazily evaluated. That means, they are not executed until a terminal operation is found.

### Method Reference

Lambda expressions can be used to create anonymous methods. However, we can replace lambda expression with method reference to call to an existing method by name. Method references enable this. In the above example, map and forEach method both use method reference.   
To pass a constructor – Object::new (Example - ArrayList::new)

### Intermediate Stream Operations

These are the stream operations that return a stream and can not be the last operation in the stream operation chain.

1. **map( function )**It maps each element in the stream based on the function passed to it. The resulting stream can be of different type
2. **filter( predicate )**It selects the elements from the stream based on the predicate passed to it
3. **sorted( )**It sorts the stream based on the natural ordering
4. **distinct( )**It removes all the duplicates from the stream
5. **limit( int )**It is used to limit the number of elements in the stream
6. **peek( consumer )**It is used to evaluate the items in the stream and returns the stream unchanged. It is mainly used for debugging the stream operation chain
7. **flatmap( function )**It maps a single object to more than one objects. It takes a function that returns a stream.

### Terminal Stream Operations

These are the stream operations that do not return a stream and are the last operation in the stream operation chain.

1. **forEach( consumer )**It is used to consume the elements in the stream
2. **collect( collector )**It is used to return the result stream of the intermediate stream operations  
   It takes collector as an argument  
   List<String> result = myStream.distinct().collect(Collectors.toList());  
   **collect( supplier, biconsumer, biconsumer )**first biconsumer – accumulator & second biconsumer – combiner ArrayList<String> result = myStream.distinct()  
    .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
3. **reduce( function )**It is used to reduce the elements in the stream to a signle value and return it. Optional<T> is returned by the reduce() method. This Optional contains the value (if any) returned by the lambda expression passed to the reduce() method. The value can be obtained by calling the Optional get() method.  
   *Optional<Integer> num = myStream.reduce( (n1, n2) -> n1 > n2? n1 : n2 );  
   System.out.println(num.get());*  //prints the max number obtained from the stream
4. **count( )**It returns the total number of elements in the stream

**Combining two streams**Stream<String> combinedStream = Stream.concat(stream1, stream2);

**>>> Program 12.2\_Streams <<<**

# REGULAR EXPRESSION