OBJECT ORIENTED PROGRAMMING METHODOLOGY

Dr. Ayesha Hakim

6	**Exception Handling & Packages, Multithreading**		**10**	**CO4**
	6.1	Packages: Creating Packages, Using Packages, Access Protection, Predefined packages		
	6.2	Exception handling: Exception as objects, Exception hierarchy, Try catch finally Throw, throws		
	6 .3	Multithreading: Thread life cycle, Multithreading advantages and issues, Simple thread program, Thread synchronization.		

Exception Handling

EXCEPTION HANDLING

- **Exception** is an event that occurs during the execution of a program and that disrupts the normal flow of instructions.

- Java exceptions are specialized events that indicate something bad has happened in the application, and the application either needs to recover or exit.

Why handle Java exceptions?

Java exception handling is important because it **helps maintain the normal, desired flow of the program** even when unexpected events occur.

If Java exceptions are not handled, **programs may crash or requests may fail**. This can be very frustrating for customers and if it happens repeatedly, you could lose those customers.

The worst situation is if your application crashes while the user is doing any important work, especially if their **data is lost.**

To make the user interface robust, it is important to handle Java exceptions to prevent the application from unexpectedly crashing and losing data. There can be many causes for a sudden crash of the system, such as incorrect or unexpected data input. For **example**, if we try to **add two users with duplicate IDs to the database**, we should throw an exception since the action would affect database integrity.

Tracking Exceptions in Java

Exceptions will happen in your application, no matter how well you write your program. You cannot predict the runtime environment entirely, especially that of your customer's.

The standard practice is to record all events in the application log file. The log file acts as a time machine helping the developer (or analyst) go back in time to view all the phases the application went through and what led to the Java exception.

For small-scale applications, going through a log file is easy. However, enterprise applications can serve thousands or even millions of requests per second. This makes manual analysis too cumbersome because errors and their causes can be lost in the noise.

This is where error monitoring software can help by grouping duplicates and providing summarized views of the top and most recent Java errors. They can also capture and organize contextual data in a way that's easier and faster than looking at logs.

How to handle Java exceptions?

This is accomplished using the keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

- You **try** to execute the statements contained within a block of code. *(A block of code is a group of one or more statements surrounded by braces.)*
- If you detect an exceptional condition within that block, you **throw an exception object** of a **specific type.**
- You **catch** and process the exception object using code that you have designed.
- You optionally execute a block of code, designated by **finally**, which **needs to be executed whether or not an exception occurs.** *(Code in the **finally** block is normally used to perform some type of **cleanup**.)*

How to handle exceptions in Java

The `try-catch` is the simplest method of handling exceptions. Put the code you want to run in the `try` block, and any Java exceptions that the code throws are caught by one or more `catch` blocks.

This method will catch any type of Java exceptions that get thrown.

```
try {
  // block of code that can throw exceptions
} catch (Exception ex) {
  // Exception handler

  // Push the handled error into Rollbar
  rollbar.error(ex, "Hello, Rollbar");
}
```

Note: You can't use a `try` block alone. The try block should be immediately followed either by a `catch` or `finally` block.

Exceptions in code written by others

There are also situations where you don't write the code to **throw** the exception object, but an exceptional condition that occurs in code written by someone else transfers control to exception-handling code that you write.

For example, the **read** method of the **InputStream** class throws an exception of type **IOException** if an exception occurs while the **read** method is executing. In this case, you are responsible only for the code in the **catch** block and optionally for the code in the **finally** block.

*(This Is The Reason That You Must Surround The Invocation Of **System.In.Read()** With A **Try** Block Followed By A **Catch** Block, Or Optionally Declare That Your Method **Throws** An Exception Of Type **IOException**.)*

Catching specific Java exceptions

You can also specify specific exceptions you would like to catch. This allows you to have **dedicated code** to recover from those errors.

A `try` block can be followed by one or more `catch` blocks, each specifying a different type.

```
try {
  // block of code that can throw exceptions
} catch (ExceptionType1 ex1) {
  // exception handler for ExceptionType1

  // Push the handled error into Rollbar
  rollbar.error(ex1, "Hello, Rollbar");
} catch (ExceptionType2 ex2) {
  // Exception handler for ExceptionType2

  // Push the handled error into Rollbar
  rollbar.error(ex2, "Hello, Rollbar");
}
```

If an exception occurs in the `try` block, the exception is thrown to the first `catch` block. If not, the Java exception passes down to the second `catch` statement. This continues until the exception either is caught or falls through all catches.

The finally block

Any code that must be executed irrespective of occurrence of an exception is put in a `finally` block. In other words, a `finally` block is executed in all circumstances. For example, if you have an open connection to a database or file, use the `finally` block to close the connection even if the `try` block throws a Java exception.

```
try {
  // block of code that can throw exceptions
} catch (ExceptionType1 ex1) {
  // exception handler for ExceptionType1

  // Push the handled error into Rollbar
  rollbar.error(ex1, "Hello, Rollbar");
} catch (ExceptionType2 ex2) {
  // Exception handler for ExceptionType2

  // Push the handled error into Rollbar
  rollbar.error(ex2, "Hello, Rollbar");
} finally {
  // finally block always executes
}
```

Handling uncaught or runtime Java exceptions

Uncaught or runtime exceptions happen unexpectedly, so you may not have a `try-catch` block to protect your code and the compiler cannot give you warnings either. Thankfully, Java provides a powerful mechanism for handling runtime exceptions. Every `Thread` includes a hook that allows you to set an `UncaughtExceptionHandler`. Its interface declares the method `uncaughtException(Thread t1, Throwable e1)`. It will be invoked when a given thread t1 terminates due to the given uncaught exception e1.

```
Thread.setDefaultUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler() {
  public void uncaughtException(Thread t1, Throwable e1) {
    // Exception handling code

    // Push the error into Rollbar
    rollbar.error(e1, "Hello, Rollbar");
  }
});
```

A thread in Java is the direction or path that is taken while a program is being executed.

EXCEPTION HIERARCHY

When an exceptional condition causes an exception to be *thrown*, that **exception is represented by an object instantiated from the class named Throwable or one of its subclasses.**

*"The **Throwable** Class Is The **Superclass Of All Errors** **And Exceptions** In The Java Language. Only Objects That Are Instances Of This Class (Or One Of Its Subclasses) Are Thrown By The Java Virtual Machine Or Can Be Thrown By The Java **Throw** Statement. Similarly, Only This Class Or One Of Its Subclasses Can Be The Argument Type In A **Catch** Clause."*

Defining your own exception types

You can define and **throw** exception objects of your own design.

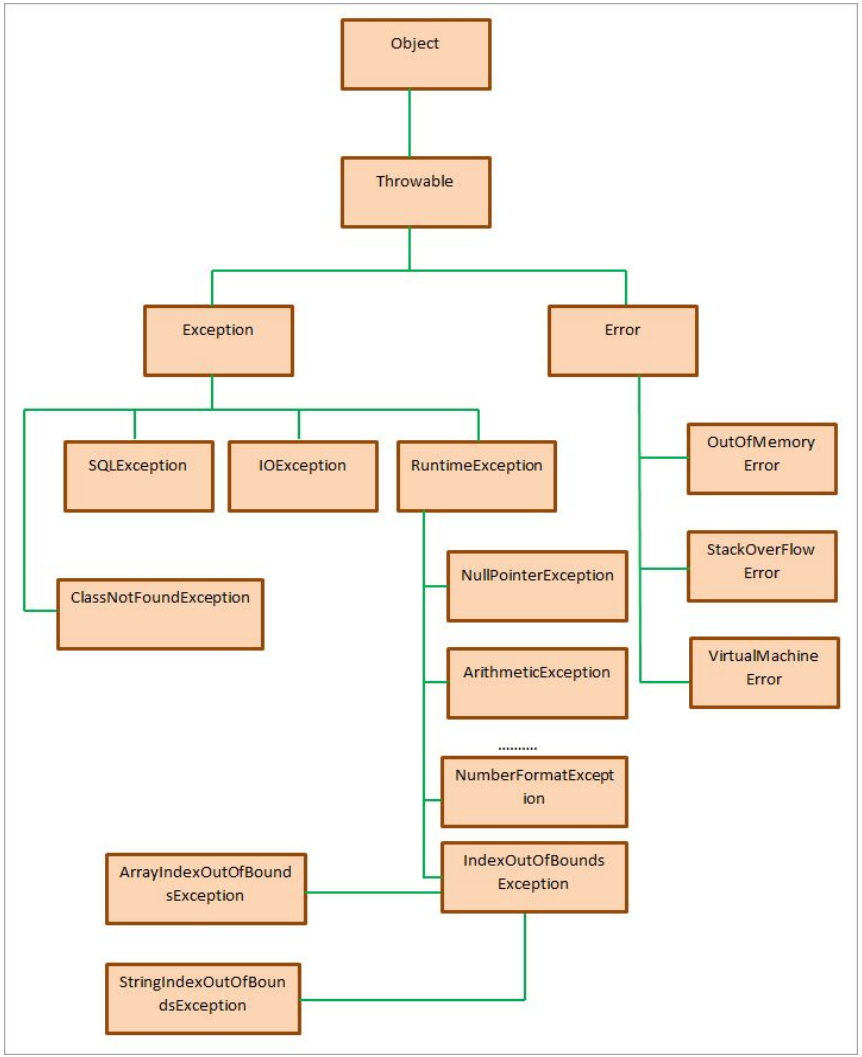
*Your new class must extend **Throwable** or one of its subclasses*.

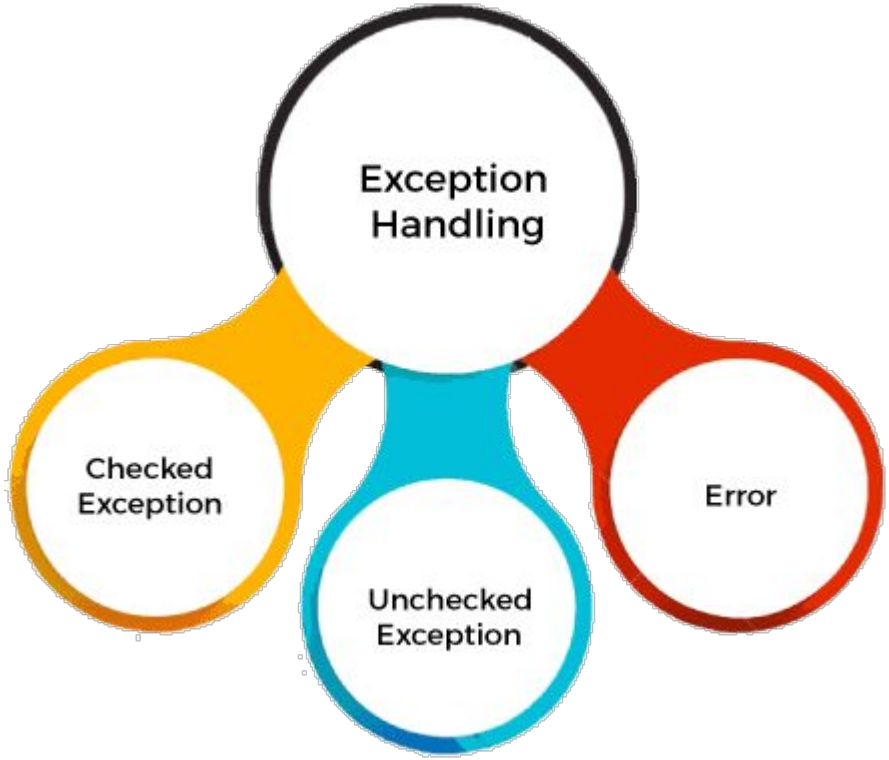
The difference between Error and Exception

Throwable class has two subclasses:

- **Error**
- **Exception**

EXCEPTION HIERARCHY





Java Exception Hierarchy

Exception Hierarchy – Following is the Exception Handling in Java handling hierarchy.

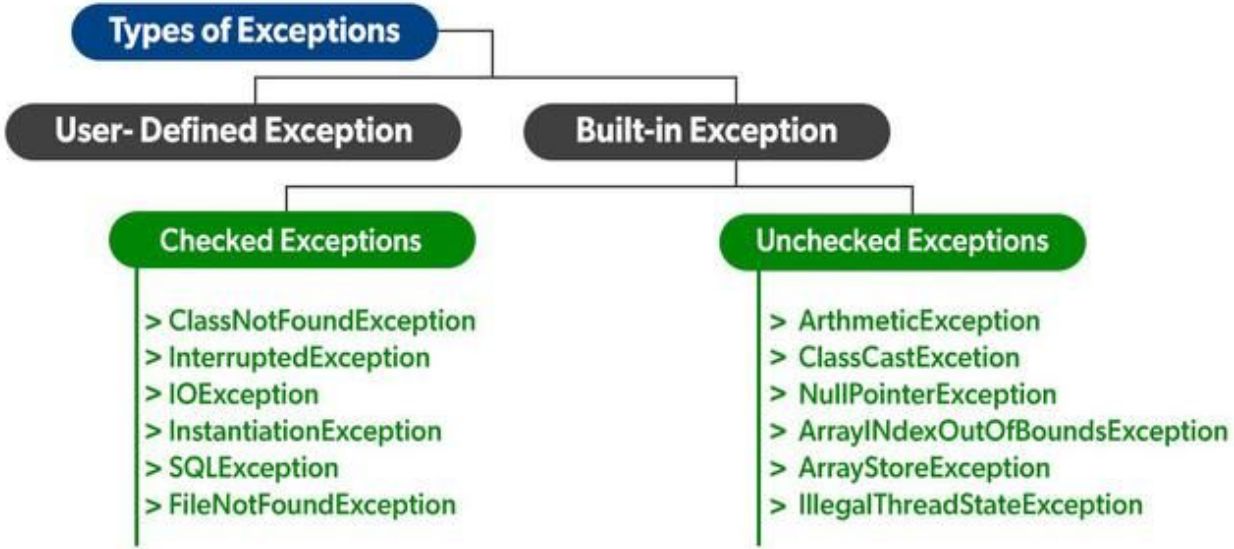
- **Throwable** –
  - It is the root class for the exception hierarchy in java.
  - It is in the java.lang package.
- **Error** –
  - Subclass of Throwable.
  - Consist of abnormal condition that is out of one's control and depends on the environment
  - They can't be handled and will always result in the halting of the program.
  - Eg: StackOverFlowError that can happen in infinite loop or recursion
- **Exception** –
  - Subclass of Throwable.
  - Consist of abnormal conditions that can be handled explicitly.
  - If one handles the exception then our code will continue to execute smoothly.

Difference between Checked and Unchecked Exception

Checked Exceptions	Unchecked Exceptions
Occur at compile time.	Occur at runtime.
The compiler checks for a checked exception.	The compiler doesn't check for exceptions.
Can be handled at the compilation time.	Can't be caught or handled during compilation time.
The JVM requires that the exception be caught and handled.	The JVM doesn't require the exception to be caught and handled.
Example of Checked exception- 'File Not Found Exception'	Example of Unchecked Exceptions- 'No Such Element Exception'

The ***advantages of Exception Handling in Java*** are as follows:

1. Provision to Complete Program Execution
2. Easy Identification of Program Code and Error-Handling Code
3. Propagation of Errors
4. Meaningful Error Reporting
5. Identifying Error Types



ERROR

Error indicates that a **non-recoverable error** (abnormal conditions) has occurred that should not be caught. Errors usually cause the Java virtual machine to **display a message and exit.**

*An **Error** Is A Subclass Of **Throwable***

For example, one of the subclasses of **Error** is named **VirtualMachineError**. This error is *"Thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating. "*

Exception indicates an abnormal condition that must be properly handled to prevent program termination.

*"The Class **Exception** And Its Subclasses Are A Form Of **Throwable** That Indicates Conditions That A Reasonable Application Might Want To Catch."*

As of JDK 1.4.0, there are **more than fifty known subclasses** of the **Exception** class. Many of these subclasses themselves have numerous subclasses.

The RuntimeException class

One **subclass of Exception** is the class named **RuntimeException**. As of JDK 1.4.0, this class has about **30 subclasses**, many which are further subclassed.

Unchecked exceptions

Exceptions instantiated from **RuntimeException** class and its subclasses as ***unchecked* exceptions.**

An unchecked exception is a type of exception that you can optionally handle, or ignore.

- ➔ If you elect to ignore the possibility of an unchecked exception, and one occurs, your program will **terminate** as a result.
- ➔ If you elect to handle an unchecked exception and one occurs, the **result will depend on the code** that you have written to handle the exception.

Checked exceptions

All exceptions instantiated from the **Exception** class, or from subclasses of **Exception** other than **RuntimeException** and its subclasses must either be:

- Handled with a **try** block followed by a **catch** block, or
- Declared in a **throws** clause of any method that can throw them

Checked exceptions *cannot be ignored* when you write the code in your methods.

The exception classes in this category represent **routine abnormal conditions that should be anticipated and caught to prevent program termination.**

Checked by the compiler

Your code must anticipate and either handle or declare checked exceptions. Otherwise, your program won't compile. *(These are exception types that are checked by the compiler.)*

Throwable constructors and methods

All errors and exceptions are subclasses of the **Throwable** class. As of JDK 1.4.0, the **Throwable** class provides **four constructors** and about a dozen methods.

Four constructors:

1. `Throwable()`
2. `Throwable(String message)`
3. `Throwable(String message, Throwable cause)`
4. `Throwable(Throwable cause)`

When an exceptional condition occurs within a method, the method may instantiate an exception object and **hand it off to the runtime system** to deal with it. This is accomplished **using the 'throw' keyword (throwing an exception).** To be useful, the exception object should probably contain information about the exception, including its **type and the state of the program when the exception occurred.**

Handling the Exception

At that point, **the runtime system** becomes responsible for finding a block of code designed to handle the exception.

The runtime system begins its search with the method in which the exception occurred and **searches backwards through the call stack until it finds a method that contains an appropriate exception handler (catch block).**

An exception handler is appropriate if the type of the exception thrown is the same as the type of exception handled by the handler, or is a subclass of the type of exception handled by the handler. If no appropriate handler is found, the runtime system and the program terminate.

Catching an exception

A method catches an exception by providing an **exception handler (catch block) whose parameter type is appropriate for that type of exception object. ##**

The type of the parameter in the catch block must be the class from which the exception was instantiated, or a superclass of that class that resides somewhere between that class and the Throwable class in the inheritance hierarchy.

https://www.javatpoint.com/multiple-catch-block-in-java

Declaring an exception

If the code in a method can throw a checked exception, and the method **does not** provide an exception handler for the type of exception object thrown, **the method must *declare* that it can throw that exception.**

The **throws** keyword is used in the method declaration to declare that it **throws** an exception of a particular type.

Any checked exception that can be thrown by a method is part of the method's programming interface *(ex. the **read** method of the InputStream class throws **IOException**).*

Users of a method must know about the exceptions that a method can throw in order to be able to handle them. Thus, you must declare the exceptions that the method can throw in the method signature.

Checked exceptions

Checked exceptions are all exception objects instantiated from subclasses of the **Exception** class other than those of the **RuntimeException** class.

Exceptions of all **Exception** subclasses other than **RuntimeException** are **checked by the compiler and will result in compiler errors if they are neither *caught* or *declared*.**

NOTE: Whether your exception objects become checked or not depends on the class that you extend when you define your exception class.

(If You Extend A Checked Exception Class, Your New Exception Type Will Be A Checked Exception. Otherwise, It Will Not Be A Checked Exception.)

Exceptions that can be thrown within the scope of a method

The exceptions that can be thrown within the scope of a method include not only **exceptions which are thrown by code written into the method,** but also **includes exceptions thrown by methods called by that method, or methods called by those methods, etc.**

"This … Includes Any Exception That Can Be Thrown While The Flow Of Control Remains Within The Method. Thus, This … **Includes Both Exceptions That Are Thrown Directly By The Method With Java's Throw Statement, And Exceptions That Are Thrown Indirectly By The Method Through Calls To Other Methods."**

Sample program with no exception handling code

The first sample program shown in Listing 1 neither catches nor declares the **InterruptedException** which can be thrown by the **sleep** method of the **Thread** class.

An InterruptedException is thrown when a thread is interrupted while it's waiting, sleeping, or otherwise occupied.

```
import java.lang.Thread;

class Excep11{
  public static void main(
                          String[] args){
    Excep11 obj = new Excep11();
    obj.myMethod();
  }//end main
  //---------------------------------//

  void myMethod(){
    Thread.currentThread().sleep(1000);
  }//end myMethod
}//end class Excep11
```

Listing 1

The method sleep() is being used to halt the working of a thread for a given amount of time.

A possible InterruptedException

The code in the **main** method of Listing 1 invokes the method named **myMethod**. The method named **myMethod** invokes the method name **sleep** of the **Thread** class. The method named **sleep** declares that it throws **InterruptedException**.

InterruptedException is a checked exception. The program illustrates the failure to either catch or declare **InterruptedException** in the method named **myMethod**.

As a result, **this program won't compile. The compiler error is the output.**

OUTPUT

```
unreported exception
java.lang.InterruptedException;
must be caught or declared to be thrown

Thread.currentThread().sleep(1000);
```

The compiler detected a problem where the **sleep** method was called, because the method named **myMethod** failed to deal properly with an exception that can be thrown by the **sleep** method.

Sample program that fixes one compiler error

The next version of the program, shown in Listing 2, fixes the problem identified with the call to the **sleep** method, by declaring the exception in the signature for the method named **myMethod**. The declaration of the exception of type **InterruptedException** is highlighted in boldface in Listing 2.

```
import java.lang.Thread;

class Excep12{
  public static void main(
                          String[] args){
    Excep12 obj = new Excep12();
    obj.myMethod();
  }//end main
  //-----------------------------------//

  void myMethod()
           throws InterruptedException{
    Thread.currentThread().sleep(1000);
  }//end myMethod
}//end class Excep12
```

Listing 2

Another possible InterruptedException

As was the case in the previous program, this program also illustrates a failure to catch or declare an **InterruptedException**. However, in this case, the problem has moved up one level in the call stack relative to the problem with the program in Listing 1.

This program also fails to compile, producing a compiler error.. Note that the caret indicates that the problem is associated with the call to **myMethod**.

OUTPUT

```
unreported exception
java.lang.InterruptedException;
must be caught or declared to be thrown
    obj.myMethod();

        ^
```

Didn't solve the problem

Simply declaring a checked exception doesn't solve the problem. Ultimately, the exception must be handled if the compiler problem is to be solved.

(Note, However, That It Is Possible To Declare That The Main Method Throws A Checked Exception, Which Will Cause The Compiler To Ignore It And Allow Your Program To Compile.)

The program in Listing 2 eliminated the compiler error identified with the call to the method named **sleep**. This was accomplished by declaring that the method named **myMethod** *throws InterruptedException*.

However, this simply passed the exception up the call stack to the next higher-level method in the stack. **This didn't solve the problem**, it simply handed it off to another method to solve. The problem still exists, and is now identified with the call to **myMethod** where it will have to be handled in order to make the compiler error go away.

Sample program that fixes the remaining compiler error

The version of the program shown in Listing 3 fixes the remaining compile error.

This program illustrates both **declaring and handling a checked exception**. This program compiles and runs successfully.

```
import java.lang.Thread;

class Excep13{
  public static void main(
                          String[] args){
    Excep13 obj = new Excep13();
    try{//begin try block
      obj.myMethod();
    }catch(InterruptedException e){
      System.out.println(
              "Handle exception here");
    }//end catch block
  }//end main
  //-----------------------------------//

  void myMethod()
            throws InterruptedException{
    Thread.currentThread().sleep(1000);
  }//end myMethod
}//end class Excep13
```

Listing 3

The solution to the problem

This solution to the problem is accomplished by **surrounding the call to myMethod with a try block, which is followed immediately by an *appropriate* catch block.** In this case, an appropriate **catch** block is one whose parameter type is either **InterruptedException**, or a superclass of **InterruptedException**.

*(Note: However, That The Superclass Cannot Be Higher Than The **Throwable** Class In The Inheritance Hierarchy.)*

The myMethod method declares the exception

As in the previous version, the method named **myMethod** (declares the exception and passes it up the call stack to the method from which it was called.

The main method handles the exception

In the new version shown in Listing 3, the **main** method provides a **try** block with an *appropriate* **catch** block for dealing with the problem *(although it doesn't actually deal with it in any significant way).* This can be interpreted as follows:

- Try to execute the code within the **try** block.
- If an exception occurs, search for a **catch** block that matches the type of object thrown by the exception.
- If such a **catch** block can be found, immediately transfer control to the catch block without executing any of the remaining code in the **try** block. *(For simplicity, this program didn't have any remaining code. Some later sample programs will illustrate code being skipped due to the occurrence of an exception.)*

Not a method call

Note that this transfer of control is not a method call. It is an unconditional transfer of control.
There is no *return* from a catch block.

Matching catch block was found

In this case, there was a matching **catch** block to receive control. In the event that an **InterruptedException** is thrown, the program would execute the statement within the body of the **catch** block, and then transfer control to the code following the final **catch** block in the group of **catch** blocks *(in this case, there was only one **catch** block)*.

No output is produced

It is unlikely that you will see any output when you run this program, because it is unlikely that an **InterruptedException** will be thrown. *(I didn't provide any code that will cause such an exception to occur.)*

A sample program that throws an exception

This program illustrates the implementation of exception handling using the try/catch block structure.

The program in Listing 4 was written to throw an **ArithmeticException**. This was accomplished by trying to perform an integer divide by zero.

The try/catch structure is the same …

It is important to note that the *try/catch* structure illustrated in Listing 4 would be the same whether the exception is checked or unchecked. **The main difference is that you are not required by the compiler to handle unchecked exceptions and you are required by the compiler to either handle or declare checked exceptions.**

```
class Excep14{
  public static void main(
                          String[] args){
    try{
      for(int cnt = 2; cnt >-1; cnt--){
        System.out.println(
                "Running. Quotient is: "
                                + 6/cnt);
      }//end for-loop
    }//end try block
    catch(ArithmeticException e){
      System.out.println(
               "Exception message is:  "
               + e.getMessage()
               + "nStacktrace shows:");
      e.printStackTrace();
      System.out.println(
        "String representation isn " +
                             e.toString());
      System.out.println(
         "Put corrective action here");
    }//end catch block
    System.out.println(
                  "Out of catch block");
  }//end main

}//end class Excep14

Listing 4
```

Throwing an ArithmeticException

The code in Listing 4 executes a simple counting loop inside a **try** block. During each iteration, the counting loop **divides the integer 6 by the value of the counter.** When the value of the counter goes to zero, the runtime system tries to perform an integer **divide by zero operation**, which causes it to throw an **ArithmeticException**.

Transfer control immediately

At that point, **control is transferred directly to the catch block** that follows the **try** block. This is an *appropriate* **catch** block because the type of parameter declared for the **catch** block is

ArithmeticException. It matches the type of the object that is thrown.

(It Would Also Be Appropriate If The Declared Type Of The Parameter Were
A Superclass Of ArithmeticException, Up To And Including The Class
Named Throwable. Throwable Is A Direct Subclass Of
Object. If You Were To Declare The Parameter Type For The Catch
Block As Object, The Compiler Would Produce An Incompatible Type Error.)

Invoking methods inside the catch block

Once control enters the **catch** block, three of the methods of the **Throwable** class are invoked to cause information about the situation to be displayed on the screen.

OUTPUT

```
Running. Quotient is: 3
Running. Quotient is: 6
Exception message is:  / by zero
Stacktrace shows:
java.lang.ArithmeticException:
  / by zero
  at Excep14.main(Excep14.java:35)
String representation is
 java.lang.ArithmeticException:
 / by zero
Put corrective action here

Out of catch block
```

Key things to note

The key things to note about the program in Listing 5 are:

- The code to be protected is contained in a **try** block.
- The **try** block is followed immediately by an appropriate **catch** block.
- When an exception is thrown within the **try** block, control is transferred immediately to the **catch** block with the matching or appropriate parameter type.
- Although the code in the **catch** block simply displays the current state of the program, it could contain code that attempts to rectify the problem.
- Once the code in the **catch** block finishes executing, control is passed to the next executable statement following the **catch** block, which in this program is a print statement.

Java ‘Throw’ Keyword

It is a keyword that is used to explicitly throw an exception. It can be **used where according to our logic an exception should occur.**

Example:

```
public class ExceptionDemo {
     static void canVote(int age){
          if(age<18)
              try{
                  throw new Exception();
              }catch(Exception e){
                  System.out.println("you are not an adult!");
              }
          else
             System.out.println("you can vote!");
     }
     public static void main (String[] args) {
          canVote(20);
          canVote(10);
     }
}
```

Output:

```
you can vote!
you are not an adult!
```

Java ‘Throws’ Keyword

- Throws keyword is used **when callee doesn’t want to handle the exception rather it wants to extend this responsibility of handling the exception to the caller of the function.**
- Basically says what sort of exception the code can throw and relies on the caller to handle it.
- It is **used to handle checked Exceptions** as the compiler will not allow code to compile until they are handled.

Example:

```
public class ExceptionDemo {
      static void func(int a) throws Exception{
            System.out.println(10/a);
      }
      public static void main (String[] args) {
            try{
                func(10);
                func(0);
            }catch(Exception e){
               System.out.println("can't divide by zero");
            }

      }
}
```

Output:

```
   1
   can't divide by zero
```

Java Throw vs Throws

Throw	Throws
This keyword is used to explicitly throw an exception.	This keyword is used to declare an exception.
A checked exception cannot be propagated with throw only.	A checked exception can be propagated with throws.
The throw is followed by an instance and used with a method	Throws are followed by class and used with the method signature.
You cannot throw multiple exceptions.	You can declare multiple exceptions

User-defined Exceptions

You can create your own exceptions in Java. Note the following points when writing your own exception classes –

- All exceptions **must be a child of Throwable.**
- If you want to write a **checked exception** that is automatically enforced by the Handle or Declare Rule, you need to extend the **Exception** class.
- If you want to write a runtime exception, you need to e**xtend the RuntimeException class.**

Syntax to define own Exception class –

```
class MyException extends Exception {
}   // extend the predefined Exception class to create your own Exception.
```

Why use Custom Defined Exceptions?

Two drawbacks of predefined exception handling mechanism -

- Predefined exceptions of Java always generate the exception report in a predefined format.
- After generating the exception report, it immediately terminates the execution of the program.

The user-defined exception handling mechanism can generate the report of error message in any special format that we prefer. It will automatically resume the execution of the program after generating the error report.

Creating a Custom Exception

Reasons to use custom exceptions:

- To catch and provide specific treatment to a subset of existing Java exceptions. It is useful when we want to properly handle the cases that are **highly specific and unique to different applications.**
- Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

In order to create a custom exception, we need to extend the **Exception class** that belongs to **java.lang package.**

References:

https://www.tutorialspoint.com/how-to-create-a-user-defined-exception-custom-exception-in-java

https://www.javatpoint.com/custom-exception

https://www.educba.com/java-user-defined-exception/

https://www.prepbytes.com/blog/java/user-defined-exception-in-java/

How to Implement User-defined Exception in Java?

To implement a user-defined exception in Java, follow these steps:

- Create a custom exception class that extends the base exception class (java.lang.Exception).
- Define the constructor for the custom exception class. The constructor can accept parameters to provide more information about the error.
- Override the toString() method to provide a custom error message.
- Use the "throw" keyword to throw an instance of the custom exception when the error condition occurs.
- Use a "try-catch" block to catch the exception and handle it appropriately.

Example on User Defined Exception

```
// File Name InsufficientFundsException.java
import java.io.*;

public class InsufficientFundsException extends Exception {
  private double amount;

  public InsufficientFundsException(double amount) {
    this.amount = amount;
  }

  public double getAmount() {
    return amount;
  }
}
```

The following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

```
// File Name CheckingAccount.java
import java.io.*;

public class CheckingAccount {
   private double balance;
   private int number;

   public CheckingAccount(int number) {
      this.number = number;
   }

   public void deposit(double amount) {
      balance += amount;
   }

   public void withdraw(double amount) throws InsufficientFundsException {
      if(amount <= balance) {
         balance -= amount;
      }else {
         double needs = amount - balance;
         throw new InsufficientFundsException(needs);
      }
   }

   public double getBalance() {
      return balance;
   }

   public int getNumber() {
      return number;
   }
}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```
// File Name BankDemo.java
public class BankDemo {

   public static void main(String [] args) {
      CheckingAccount c = new CheckingAccount(101);
      System.out.println("Depositing $500...");
      c.deposit(500.00);

      try {
         System.out.println("\nWithdrawing $100...");
         c.withdraw(100.00);
         System.out.println("\nWithdrawing $600...");
         c.withdraw(600.00);
      } catch (InsufficientFundsException e) {
         System.out.println("Sorry, but you are short $" + e.getAmount());
         e.printStackTrace();
      }
   }
}
```

Compile all the above three files and run BankDemo.

Output

```
Depositing $500...

Withdrawing $100...

Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:25)
        at BankDemo.main(BankDemo.java:13)
```

Java Custom Exception Implementation

```
import java.util.*;

class WeightLimitExceeded extends Exception{
    WeightLimitExceeded(int x){
        System.out.print(Math.abs(15-x)+" kg : ");
    }
}
class Main {
    void validWeight(int weight) throws WeightLimitExceeded{
        if(weight>15)
            throw new WeightLimitExceeded(weight);
        else
            System.out.println("You are ready to fly!");
    }
      public static void main (String[] args) {
        Main ob=new Main();
        Scanner in=new Scanner(System.in);
        for(int i=0;i<2;i++){                //code will run 2 times
            try{
                ob.validWeight(in.nextInt());
            }catch(WeightLimitExceeded e){
                System.out.println(e);
            }
        }

     }
}
```

Input:

```
20
7
```

Output:

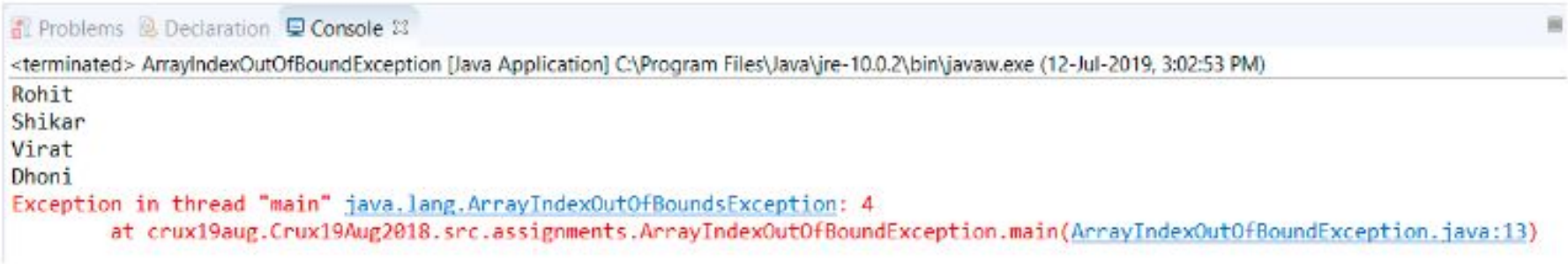
```
5 kg : WeightLimitExceeded
You are ready to fly!
```

Example of ArrayIndexOutOfBoundsException

```
public class ArrayIndexOutOfBoundException {

   public static void main(String[] args) {
String[] arr = {"Rohit","Shikar","Virat","Dhoni"};
//Declaring 4 elements in the String array

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

//Here, no element is present at the iteration number arr.length, i.e 4
          System.out.println(arr[i]);
//So it will throw ArrayIndexOutOfBoundException at iteration 4
      }

   }

}
```

The ArrayIndexOutOfBoundsException occurs whenever we are **trying to access any item of an array at an index which is not present in the array**. In other words, the index may be negative or exceed the size of an array.

Output:


Problems
Declaration
Console
<terminated> ArrayIndexOutOfBoundException [Java Application] C:\Program Files\Java\jre-10.0.2\bin\javaw.exe (12-Jul-2019, 3:02:53 PM)
Rohit
Shikar
Virat
Dhoni
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
at crux19aug.Crux19Aug2018.src.assignments.ArrayIndexOutOfBoundException.main(ArrayIndexOutOfBoundException.java:13)

Example

The following is an array declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;

public class ExcepTest {

   public static void main(String args[]) {
      try {
         int a[] = new int[2];
         System.out.println("Access element three :" + a[3]);
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception thrown  :" + e);
      }
      System.out.println("Out of the block");
   }
}
```

Output

```
Exception thrown
:java.lang.ArrayIndexOutOfBoundsException: 3
   Out of the block
```

ARITHMETIC EXCEPTION

1. When we perform a division where 0 is used as a divisor, i.e., 0 comes as the denominator.
2. A long decimal number that is non-terminating by Big Decimal.

FileName: ArithmeticException1.java

```
// import statement
import java.math.BigDecimal;
public class ArithmeticException1
{
// main method
public static void main(String[] argvs)
{
// creating two objects of BigDecimal
BigDecimal a1 = new BigDecimal(11);
BigDecimal a2 = new BigDecimal(17);
// 11 / 17 = 0.6470588235294118...
a1 = a1.divide(a2);
System.out.println(a1.toString());
}
}
```

Non-Terminating Big Decimal

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Non-terminating
decimal expansion; no exact representable decimal result.
        at java.base/java.math.BigDecimal.divide(BigDecimal.java:1766)
        at ArithmeticException1.main(ArithmeticException1.java:9)
```

Explanation: In the above program, the **Big Decimal class does not know the exact output, which comes after division, to display.** It is because ***the output is non-terminating decimal expansion***. One approach to solve it is to **provide the limit**. For example, we can tell explicitly in the program that the output should be limited to 6 decimal places.

```
// import statement
import java.math.BigDecimal;
public class HandleArithmeticException1
{
// main method
public static void main(String[] argvs)
{
// creating two objects of BigDecimal
BigDecimal a1 = new BigDecimal(11);
BigDecimal a2 = new BigDecimal(17);
try
{
// 11 / 17 = 0.6470588235294118...
a1 = a1.divide(a2);
System.out.println(a1.toString());
}
// handling the exception in the catch block
catch(ArithmeticException ex)
{
System.out.println("Should avoid dividing by an integer that leads to non-terminating decimal expansion. " + ex);
}
}
}
```

OUTPUT

```
Should avoid dividing by an integer that leads to non-terminating decimal expansion. java.lang.ArithmeticException: Non-terminating decimal expansion; no exact representable decimal result.
```

```java
// import statement
import java.math.BigDecimal;
public class ArithmeticException2
{
// main method
public static void main(String[] argvs)
{
// creating two objects of BigDecimal
BigDecimal a1 = new BigDecimal(11);
BigDecimal a2 = new BigDecimal(17);
// 11 / 17 = 0.6470588235294118...
// rounding up to 6 decimal places
a1 = a1.divide(a2, 6, BigDecimal.ROUND_DOWN);
System.out.println(a1.toString());
}
}
```

Output:

```
0.647058
```

The BigDecimal class provides operations for arithmetic, scale manipulation, rounding, comparison, hashing, and format conversion.

Java Exception Keywords

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

The Finally Block

- The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.
- Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.
- A finally block appears at the end of the catch blocks and has the following syntax –

Syntax

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

Example

```
public class ExcepTest {

  public static void main(String args[]) {
    int a[] = new int[2];
    try {
      System.out.println("Access element three :" + a[3]);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown  :" + e);
    }finally {
      a[0] = 6;
      System.out.println("First element value: " + a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

Output

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
   The finally statement is executed
```

```
/*File Excep15.java
/

class Excep15{
  public static void main(
                        String[] args){
    new Excep15().aMethod();
  }//end main
  //---------------------------------//

  void aMethod(){
    try{
      int x = 5/0;
    }//end try block
    catch(ArithmeticException e){
      System.out.println(
      "In catch, terminating aMethod");
      return;
    }//end catch block
finally{
      System.out.println(
            "Executing finally block");
    }//end finally block

    System.out.println(
                "Out of catch block");
  }//end aMethod

}//end class Excep15
```

Listing 5

The code in Listing 5 forces an **ArithmeticException** by attempting to do an integer divide by zero. Control is immediately transferred to the matching **catch** block, which prints a message and then executes a **return** statement.

Normally, execution of a **return** statement terminates the method immediately. In this case, however, before the method terminates and returns control to the calling method, the code in the **finally** block is executed. Then control is transferred to the **main** method, which called this method in the first place.

```
OUTPUT

In catch, terminating aMethod
Executing finally block
```

Java Nested Try

When there is another try block within the try block:

```
class Main {
     public static void main (String[] args) {
        try{
                try{
                    int[] a={1,2,3};
                    System.out.println(a[3]);
                }
   catch(ArrayIndexOutOfBoundsException e)
                {
                    System.out.println("Out of bounds");
                }
              System.out.println(4/0);
        }
       catch(ArithmeticException e)
        {
            System.out.println("ArithmeticException : divide by 0");
        }
     }
    }
```

Output:

```
Out of bounds
   ArithmeticException: Divide by 0
```

```
class Main {
     public static void main (String[] args) {
         try{
                System.out.println(4/0);
                try{
                     int[] a={1,2,3};
                     System.out.println(a[3]);
                 }
   catch(ArrayIndexOutOfBoundsException e)
                 {
                     System.out.println("Out of bounds");
                 }
         }
        catch(ArithmeticException e)
         {
             System.out.println("ArithmeticException : divide by 0");
         }
      }
     }
```

Output:

```
ArithmeticException: Divide by 0
```

Note – If we put code of outer try before inner try, then if an exception occurs, it will ignore the entire inner try and move directly to its catch block.

TUTORIAL CODING ACTIVITY

Q1. WAJP to demonstrate the occurrence of ArithmeticException caused by dividing a number by zero and handle the exception using a try and catch block.

Q2. WAJP to demonstrate a scenario where ArrayIndexOutOfBoundsException occurs.

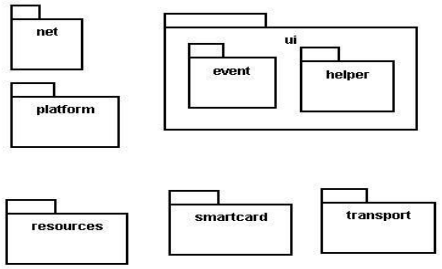
Q3. WAJP to demonstrate the occurrence of ArithmeticException caused by non terminating Big Decimal and handle the exception.

Q4. Predict the output (Refer **practice sheet)**

Packages in Java

Java packages

- **package**: A collection of related classes.
  - Can also "contain" sub-packages.
  - *Sub-packages* can have similar names, but are not actually contained inside.
    - `java.awt` does not contain `java.awt.event`



- Uses of Java packages:
  - group related classes together
  - as a *namespace* to avoid name collisions
  - provide a layer of access / protection
  - keep pieces of a project down to a manageable size

Packages

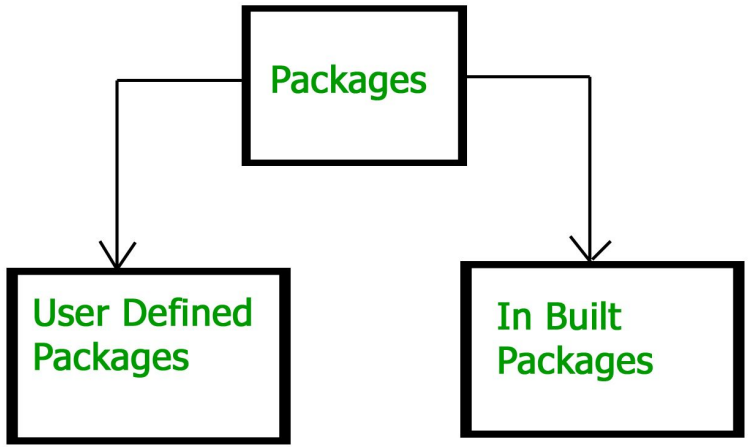
- Packages are Java's way of **grouping a number of related classes and/or interfaces** together into a single unit. That means, packages act as **"containers" for classes.**
- The **benefits** of organising classes into packages are:

- The classes contained in the packages of other programs/applications can be **reused**.
- In packages, classes can be **unique** compared with classes in other packages. That two classes in two different packages can have the same name. If there is a naming clash, then classes can be accessed with their fully qualified name.
- Classes in packages can be **hidden** if we don't want other packages to access them.
- Packages also provide a way for separating "design" from coding.

TYPES OF PACKAGES

There are basically only 2 types of java packages.
They are as follow :

- System Packages or Java API
- User Defined Packages.



System Packages or Java API

As there are built in methods , java also provides inbuilt packages which contain lots of classes & interfaces. These classes inside the packages are already defined & we can use them by importing relevant package in our program.

Java has an extensive library of packages, a programmer need not think about logic for doing any task. For everything, there are the methods available in java and that method can be used by the programmer without developing the logic on his own. This makes the programming easy.

JAVA SYSTEM PACKAGES & THEIR CLASSES

- **java.lang**

  Language Support classes. These are classes that java compiler itself uses & therefore they are automatically imported. They include classes for primitive types, strings, maths function, threads &exception.

- **java .util**

  Language Utility classes such as vector, hash tables ,random numbers, date etc.

- **java.io**

  Input /Output support classes. They provide facilities for the input & output of data

- **java.awt**

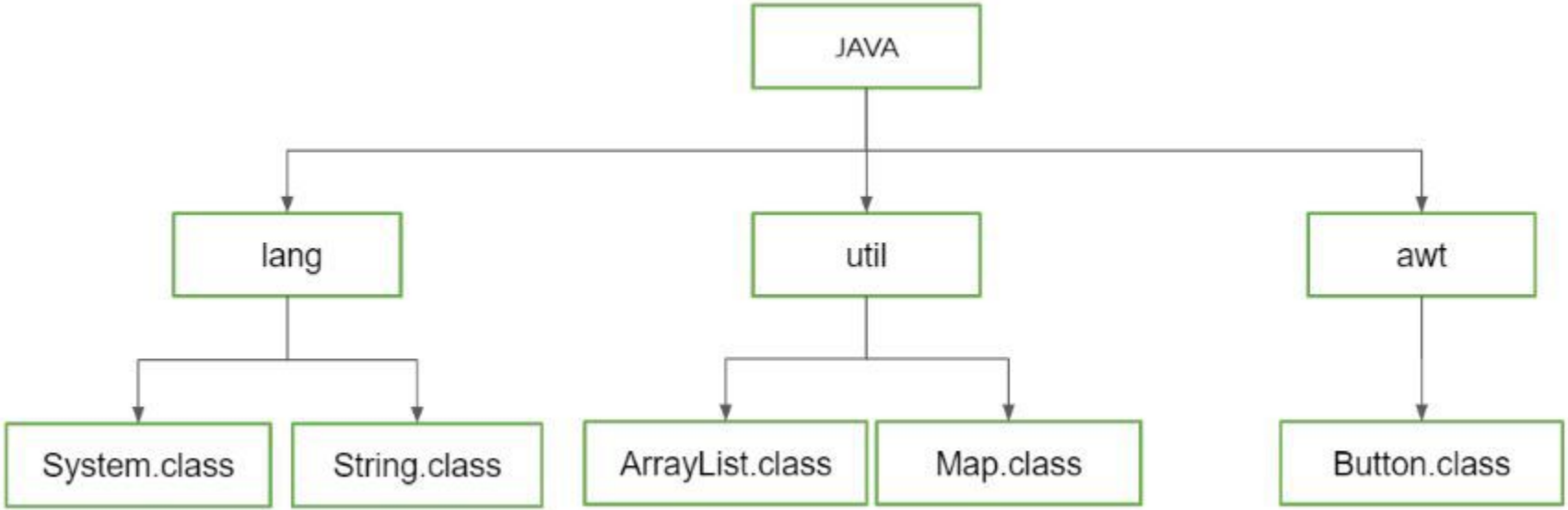
  Set of classes for implementing graphical user interface. They include classes for windows, buttons, list, menus & so on.

- **java.net**

  Classes for networking. They include classes for communicating with local computers as well as with internet servers.

- **java.applet**

  Classes for creating & implementing applets.


JAVA
lang
util
awt
System.class
String.class
ArrayList.class
Map.class
Button.class

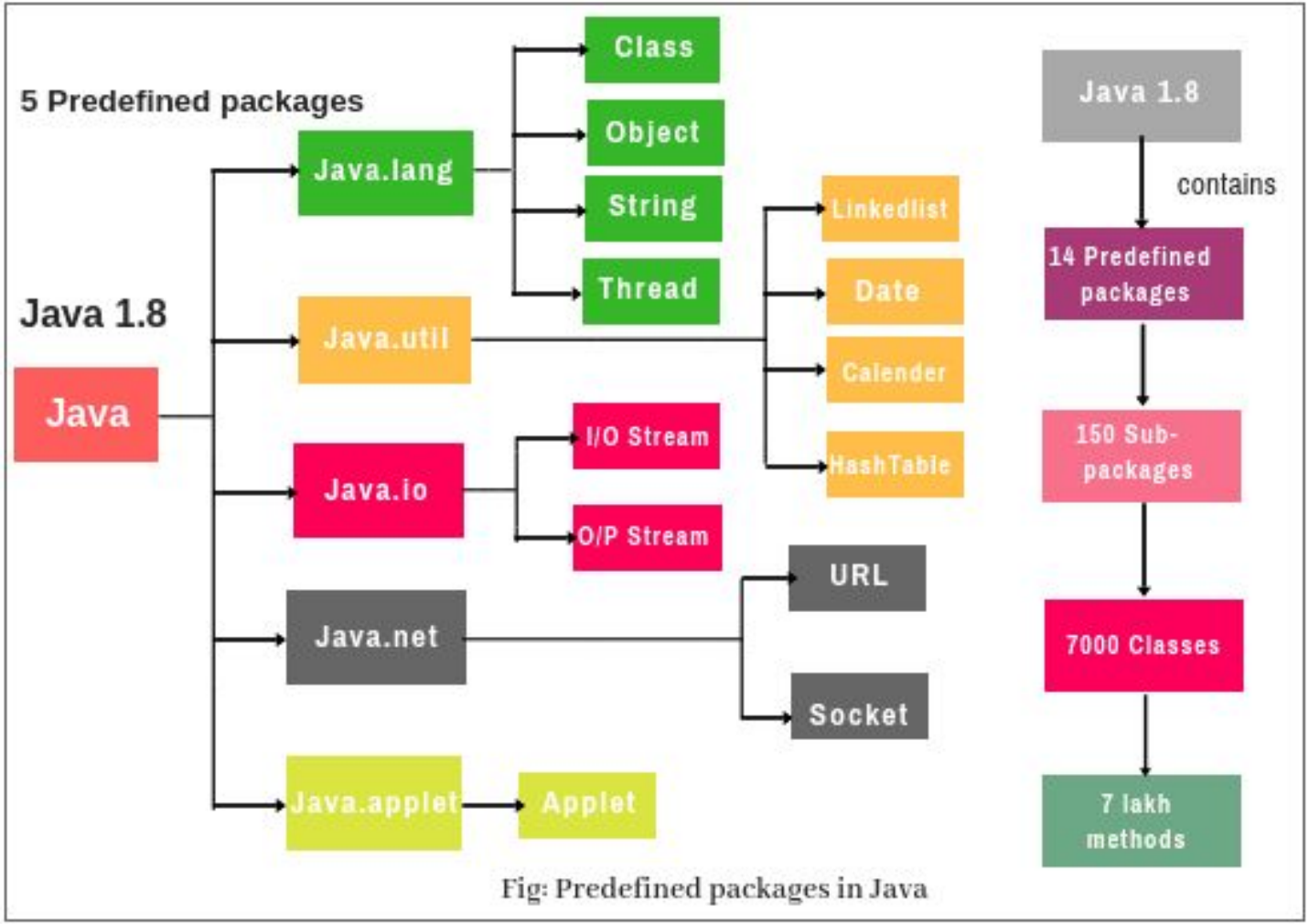
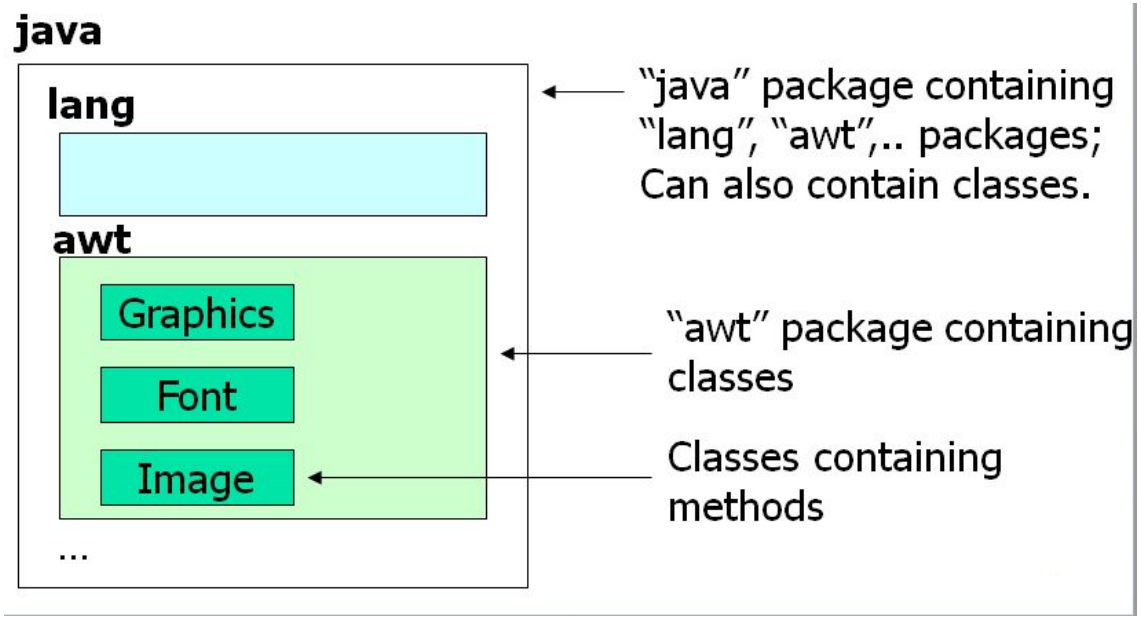


Fig: Predefined packages in Java

Using System Packages

- The packages are organised in a hierarchical structure .For example, a package named “java” contains the package “awt”, which in turn contains various classes required for implementing GUI (graphical user interface).



2. User Defined Packages :

The users of the Java language can also create their own packages. They are called user-defined packages. User defined packages can also be imported into other classes & used exactly in the same way as the Built in packages.

i) Creating User Defined Packages

Syntax :

```
package packageName;
public class className
{
        - - - - - - - - - - - - -
        // Body of className
        - - - - - - - - - - - - -
}
```

We must first declare the name of the package using the **package** keyword followed by the package name. This must be the first statement in a Java source file. Then define a classes as normally as define a class.

Example :

```
package myPackage;
public class class1
{
- - - - - - - - - - - - - -
// Body of class1
}
```

In the above example, myPackage is the name of the package. The class class1 is now considered as a part of this package.

This listing would be saved as a file called class1.java & located in a directory named mypackage. When the source file is compiled, java will create a .class file & store it in the same directory.

The .class files must be located in a directory that has the same name as the package & this directory should be a subdirectory of the directory where classes that will import the package are located.

STEPS FOR CREATING PACKAGE :

To create a user defined package the following steps should be involved :-

1: Declare the package at the beginning of a file using the syntax :-

```
package packageName;
```

2: Define the class that is to be put in the package & declare it public.

3: Create a subdirectory under the directory where the main source files are stored.

4: Store the listing as the classname.java file in the subdirectory created.

5: Compile the file. This create .class file in the subdirectory.

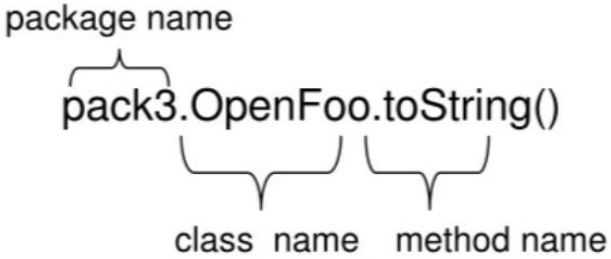
Java also supports the concept of package hierarchy. This is done by specifying multiple names in a package statement, seprated by dots (.).

Ex :- package firstPackage.secondPackage;

This approach allows us to group related classes into a package and their group related package into a larger package. Store this package in a subdirectory named firstpackage/secondPackage.

A java package file can have more than one class definition. In such cases, only one of the classes may be declared public & that class name with .java extension is the source file name. When a source file with more than one class definition is compiled, java creates independent .class files for those classes.

Fully Qualified Names: Includes Package



pack3.ClosedFoo.toString()

Importing Packages

Importing all classes from a package

Format

import <*package name*>.*;

Example

import java.util.*;

Importing a single class from a package

Format

import <package name>.<class name>;

Example

import java.util.Vector;

Importing Packages (2)

When you do not need an import statement:

- When you are using the classes in the java.lang package.
- You do not need an import statement in order to use classes which are part of the same package

Excluding the import requires that the full name be provided:

```
java.util.Random generator = new java.util.Random ();
```

Vs.

```
import java.util.Random;
Random generator = new Random ();
```

ACCESSING A PACKAGE

Java package can be accessed either using a fully qualified class name or using a shortcut approach through the import statement.

Syntax :

```
import package1[.package2][.package3].classname;
```

Here, package1 is the name of the top level package, package2 is the name of the package that is inside the package & so on. We can have any number of packages in a package hierarchy. Finally the explicit classname is specified. The import statement must end with a semicolon (;). The import startment should appear before any class definitions in a source file. Multiple import statements are allowed.

Ex :

```
import firstpackage.secondPackage.Myclass;
                    or
import firstpackage.*;
```

Advantages Of Packages

There are several advantages of package some of them are as follow :-

1: Packages are useful to arrange related classes and interface into a group.This makes all the classes & interface performing the same task to put together in the same package.

2: Packages hide the classes & interfaces in a seprate subdirectory, so that accidental deletion of classes & interfaces will not take place.

3: The classes & interfaces of a packages are isolated form the classes & interfaces of another packages. This means that we can use same names for classes of two different classes.

4: A group of packages is called a library. The classes & interface of a package are like books in a library & can be reused several times. This reusability nature of packages makes programming easy.

Accessing Classes from Packages

There are two ways of accessing the classes stored in packages:

- Using fully qualified class name

  java.lang.Math.sqrt(x);

- Import package and use class name directly

  import java.lang.Math

  Math.sqrt(x);

- Selected or all classes in packages can be imported:

```
import package.class;
import package.*;
```

- Implicit in all programs: import java.lang.*;
- package statement(s) must appear first

Creating Packages

Java supports a keyword called “package” for creating user-defined packages. The package statement must be the first statement in a Java source file (except comments and white spaces) followed by one or more classes.

```
package myPackage;
public class ClassA {
// class body
}
class ClassB {
}
```

Package name is “myPackage” and classes are considered as part of this package; The code is saved in a file called “ClassA.java” and located in a directory called “myPackage”.

Accessing a Package

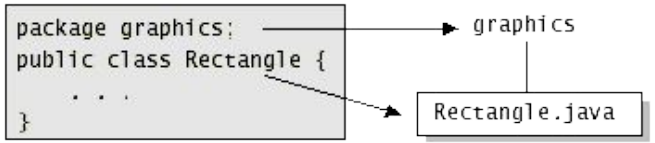
- Classes in packages can be accessed using a fully qualified name or using a short-cut as long as we import a corresponding package.
- The general form of importing package is:
  import package1[.package2][…].classname
- Example:
  import myPackage.ClassA;
  import myPackage.secondPackage
- All classes/packages from higher-level package can be imported as follows:
  import myPackage.*;
  - Package java.lang is implicitly imported in all programs by default.
    - `import java.lang.*;`

Packages and directories

- package □□ directory (folder)
- class □□ file

- A class named `D` in package `a.b.c` should reside in this file:

`a/b/c/D.class`



  - (relative to the root of your project)
- The "root" directory of the package hierarchy is determined by your *class path* or the directory from which `java` was run.

Classpath

- **class path**: The location(s) in which Java looks for class files.
- Can include:
  - the current "working directory" from which you ran javac / java
  - other folders
  - ...
- Can set class path manually when running java at command line:
  - `java -cp /home/stepp/libs:/foo/bar/jbl MyClass`

Using a Package

- Let us store the code listing below in a file named“ClassA.java” within subdirectory named “myPackage”within the current directory (say “abc”).

```
package myPackage;
public class ClassA {
// class body
public void display()
{
System.out.println("Hello, I am ClassA");
}
}
class ClassB {
// class body
}
```

Using a Package

Within the current directory (“abc”) store the following code in a file named“ClassX.java”

```
import myPackage.ClassA;
public class ClassX
{
public static void main(String args[])
{
ClassA objA = new ClassA();
objA.display();
}
}
```

Compiling and Running

- en ClassX.java is compiled, the compiler compiles it and places .class file in current directory. If .class of ClassA in subdirectory “myPackage” is not found, it compiles ClassA also.
- Note: It does not include code of ClassA into ClassX
- When the program ClassX is run, java loader looks for ClassA.class file in a package called “myPackage” and loads it.

Using a Package

Let us store the code listing below in a file named “ClassA.java” within subdirectory named“secondPackage” within the current directory (say“abc”).

```
public class ClassC {
// class body
public void display()
{
System.out.println("Hello, I am ClassC");
}
    }
```

Using a Package

Within the current directory (“abc”) store the following code in a file named“ClassX.java”

```
import myPackage.ClassA;
import secondPackage.ClassC;
public class ClassY
{
public static void main(String args[])
{
ClassA objA = new ClassA();
ClassC objC = new ClassC();
objA.display();
objC.display();
}
}
```

https://www.guru99.com/java-packages.html (see video)

Output

- Hello, I am ClassA
- Hello, I am ClassC

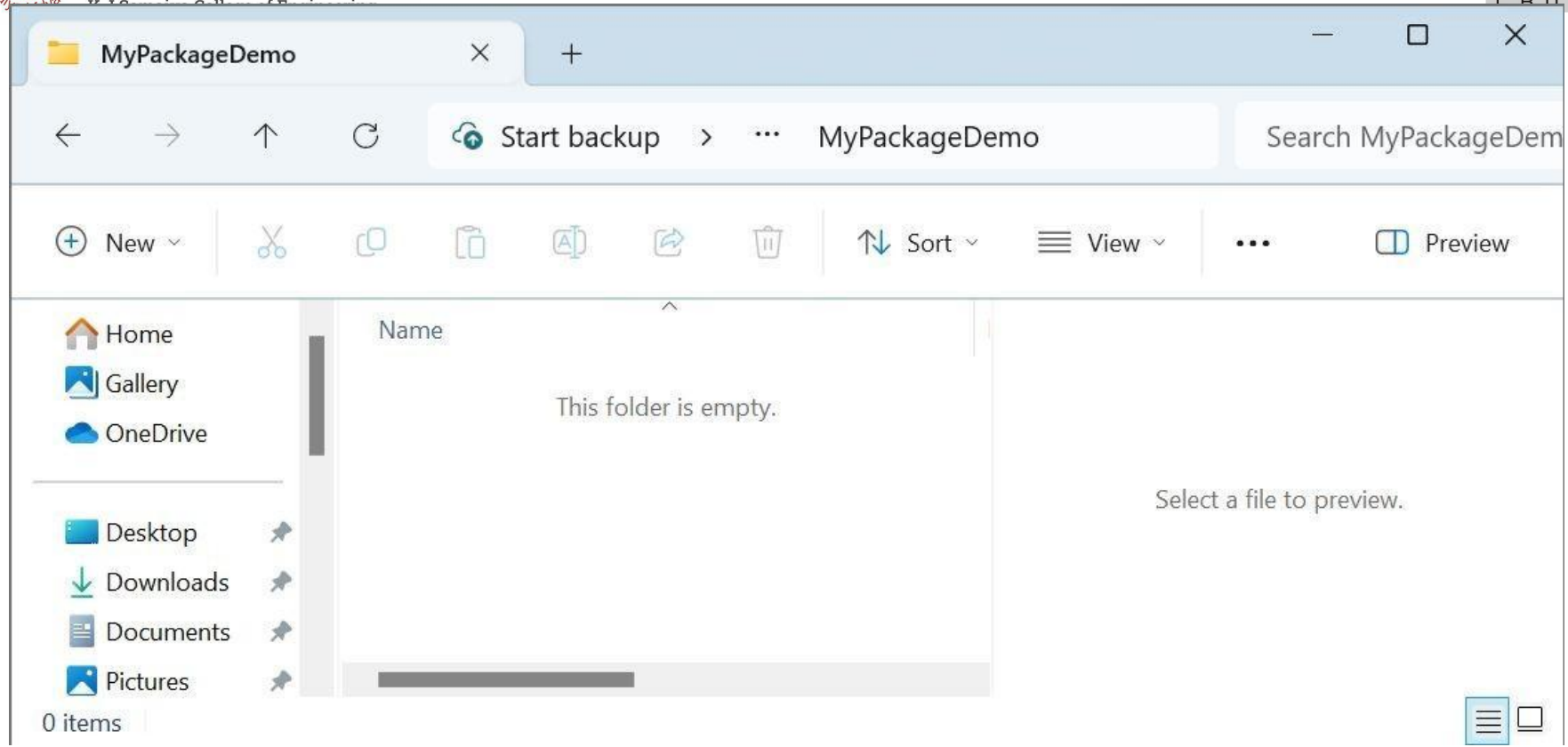
Package access

- Java provides the following access modifiers:
  - **`public`** : Visible to all other classes.
  - **`private`** : Visible only to the current class (and any nested types).
  - **`protected`** : Visible to the current class, any of its subclasses, and any other types within the same package.
  - **default** (package): Visible to the current class and any other types within the same package.
- To give a member default scope, do not write a modifier:

```
package pacman.model;
public class Sprite {
    int points;       // visible to pacman.model.*
    String name;      // visible to pacman.model.*
```

PACKAGES DEMO

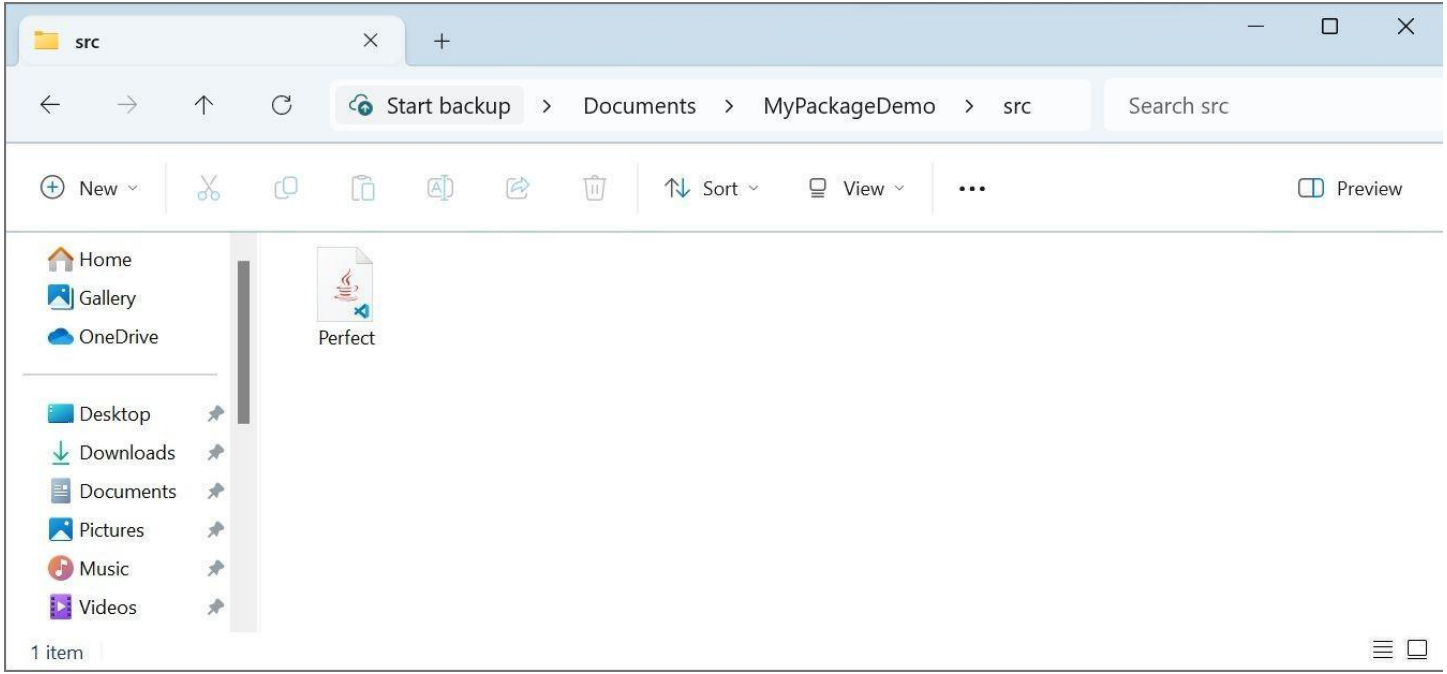
Create your project directory



Create your java file in this directory

```
package com.mypackage;


public class Perfect {
   public static boolean checkPerfect(int num) {
      int sum = 0;
      for (int i = 1; i < num;
         i++) {  if (num % i == 0)
         {
            sum += i;
         }
      }
      return sum == num;
   }
}
```

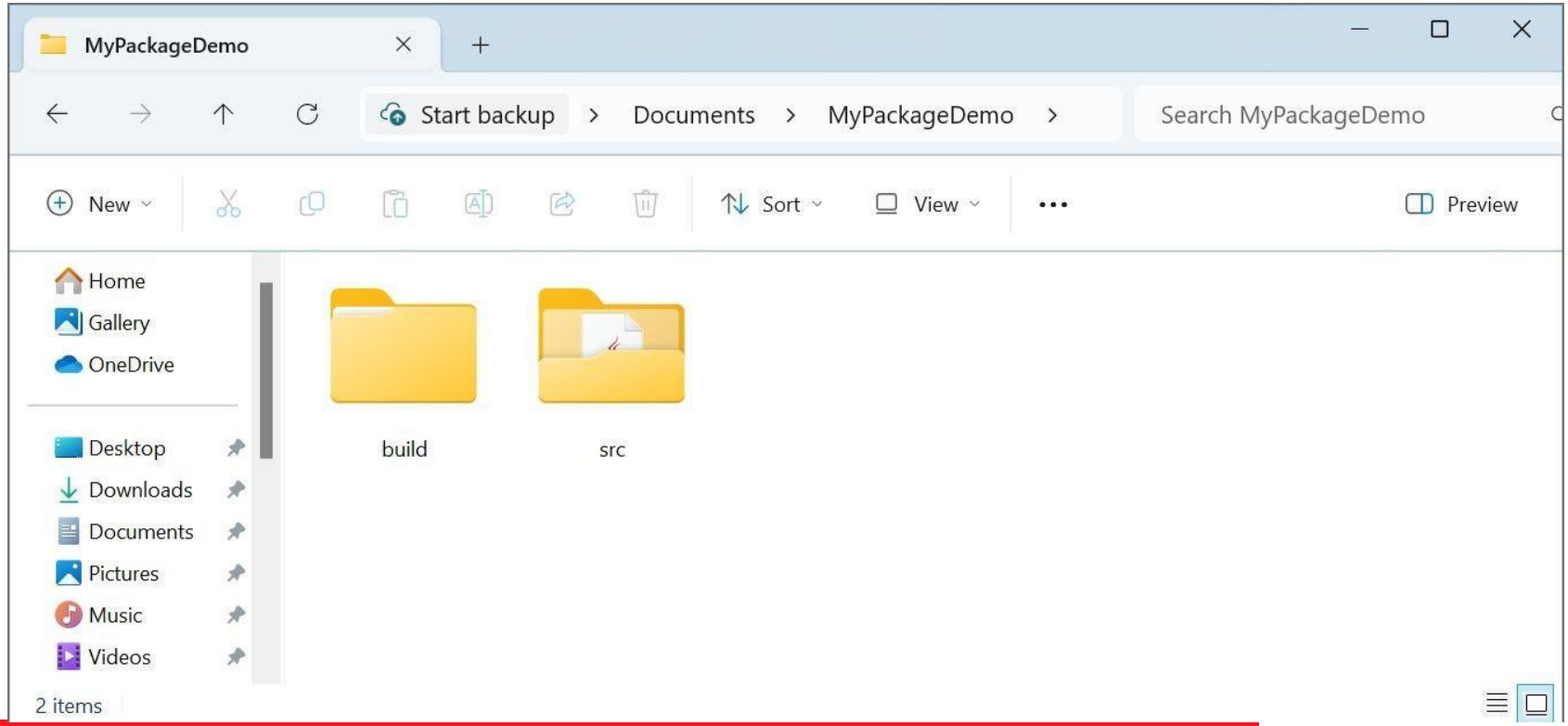
Create your source file

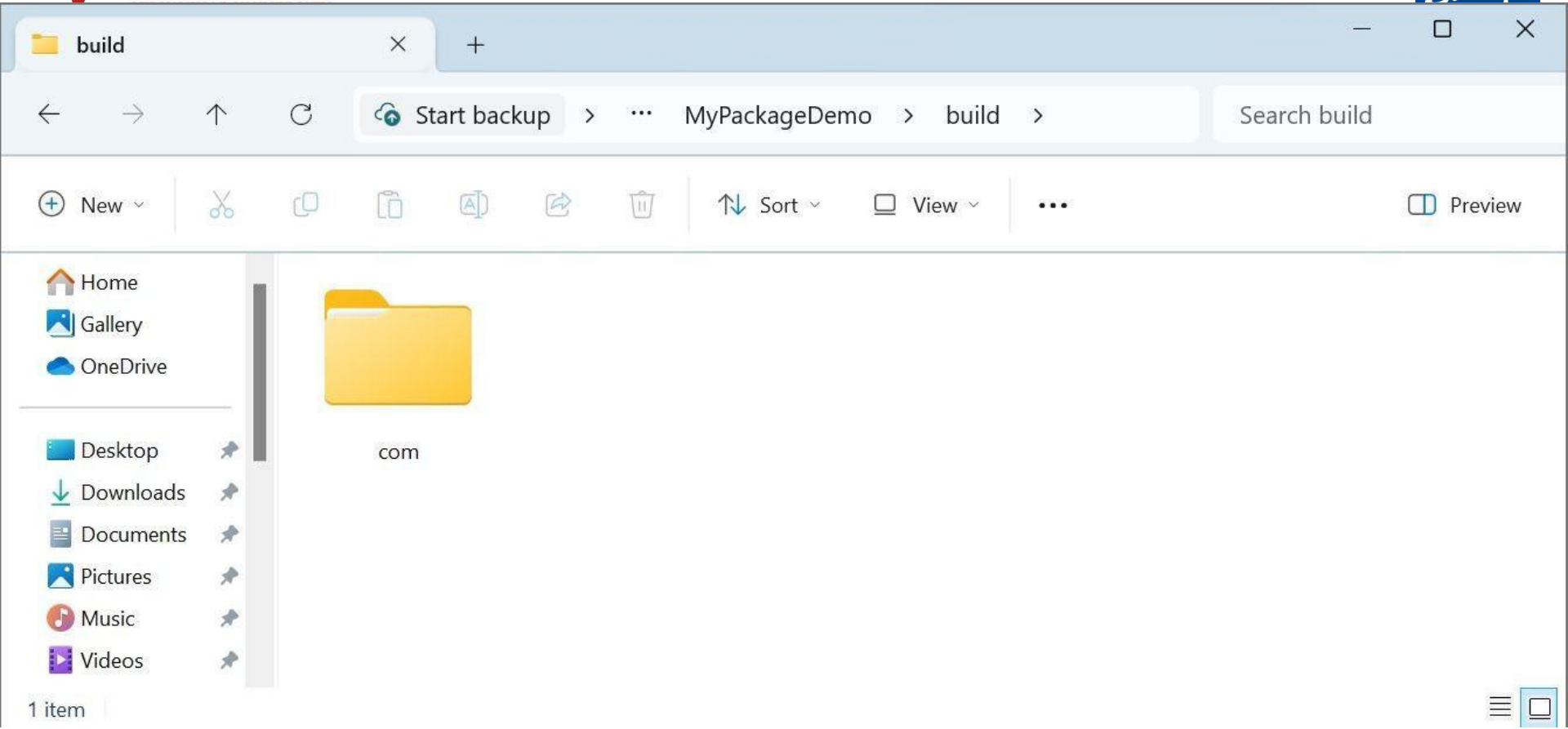


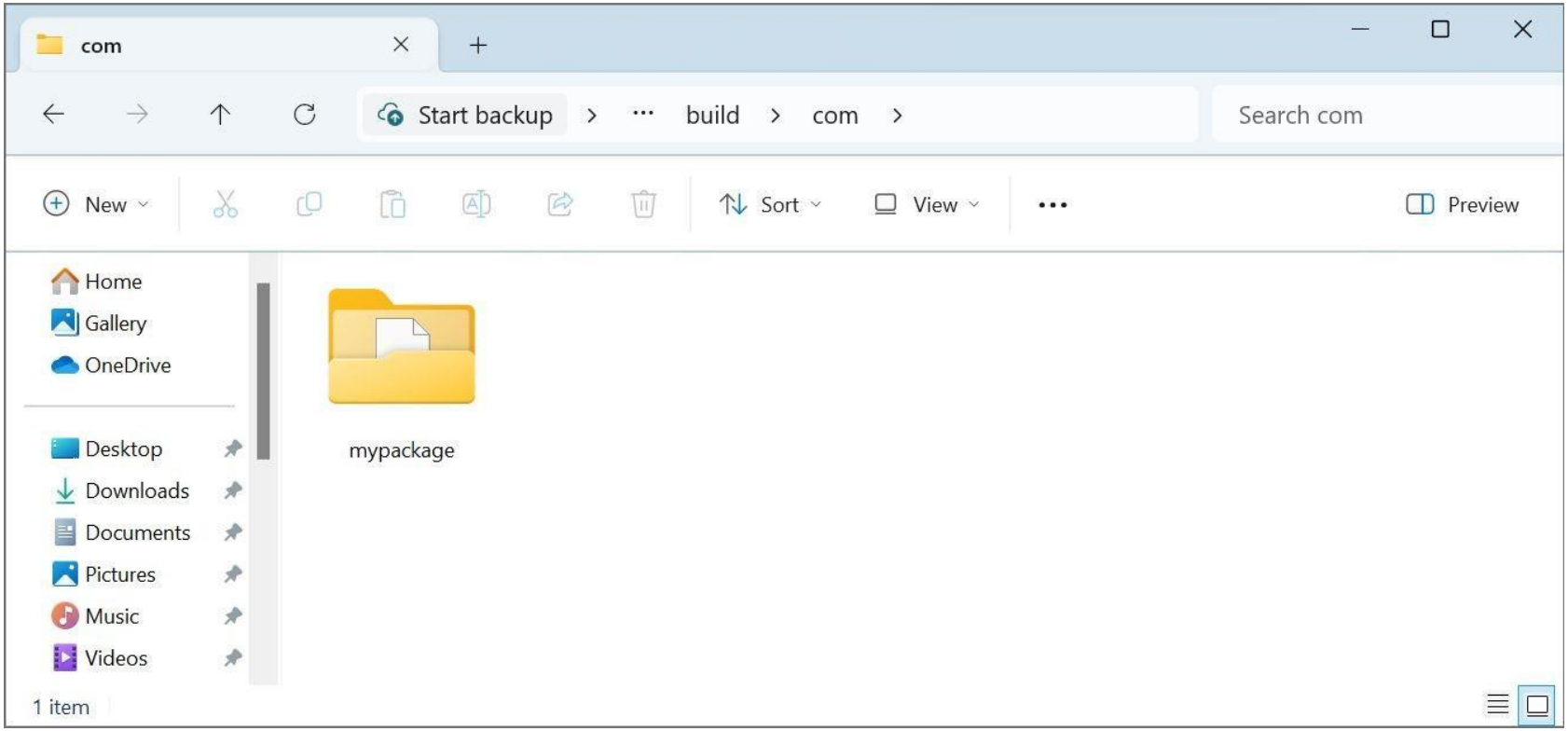
Compile your source file

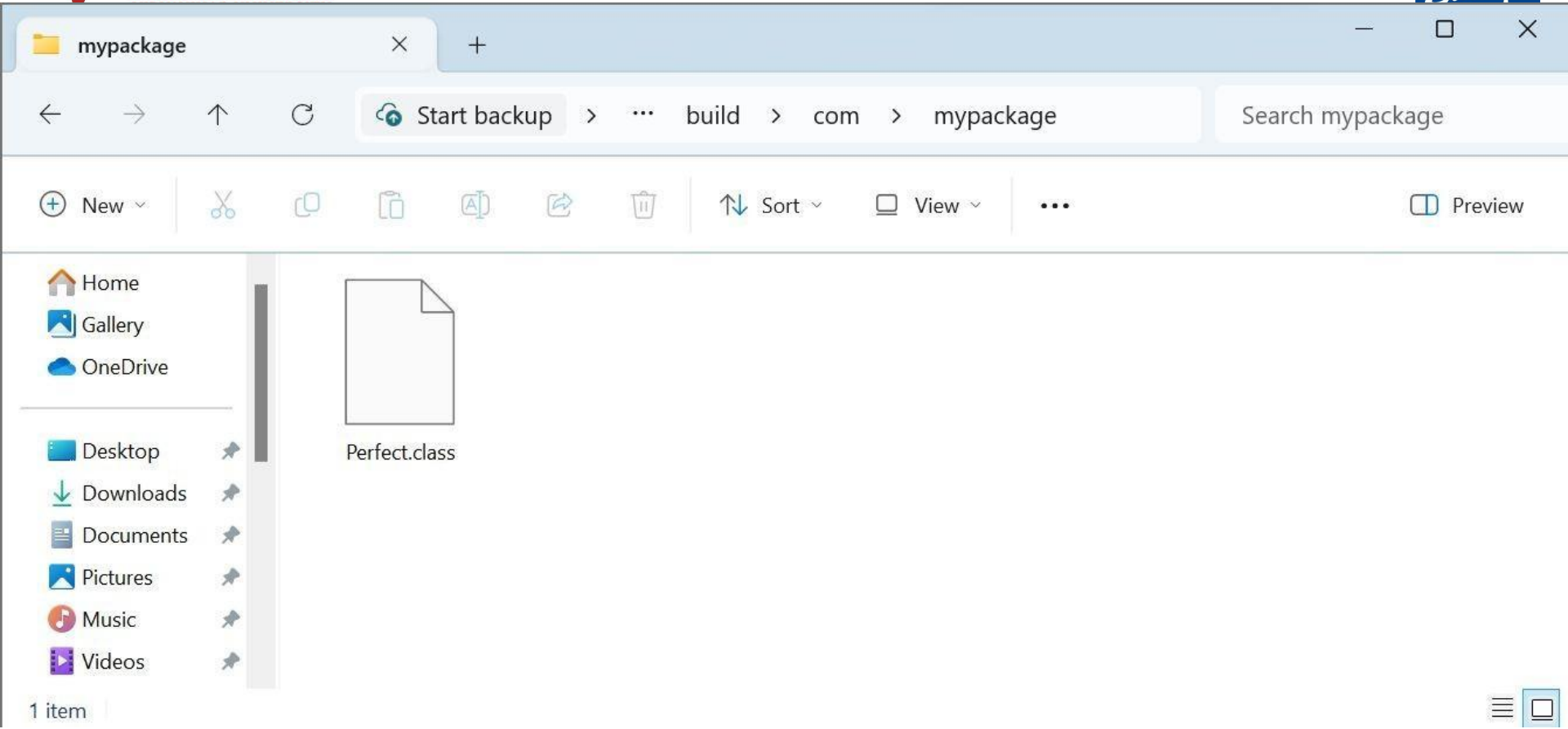
```
C:\Users\KRK\Documents\MyPackageDemo\src>javac -d C:\Users\KRK\Documents\MyPackageDemo\build Perfect.java

C:\Users\KRK\Documents\MyPackageDemo\src>
```

Result of compiling the source file




build
Start backup
MyPackageDemo
build
Search build
New
Sort
View
Preview
Home
Gallery
OneDrive
Desktop
Downloads
Documents
Pictures
Music
Videos
com
1 item


com
Start backup
build
com
Search com
New
Sort
View
Preview
Home
Gallery
OneDrive
Desktop
Downloads
Documents
Pictures
Music
Videos
mypackage
1 item


mypackage
Start backup
build
com
mypackage
Search mypackage
New
Sort
View
Preview
Home
Gallery
OneDrive
Desktop
Downloads
Documents
Pictures
Music
Videos
Perfect.class
1 item

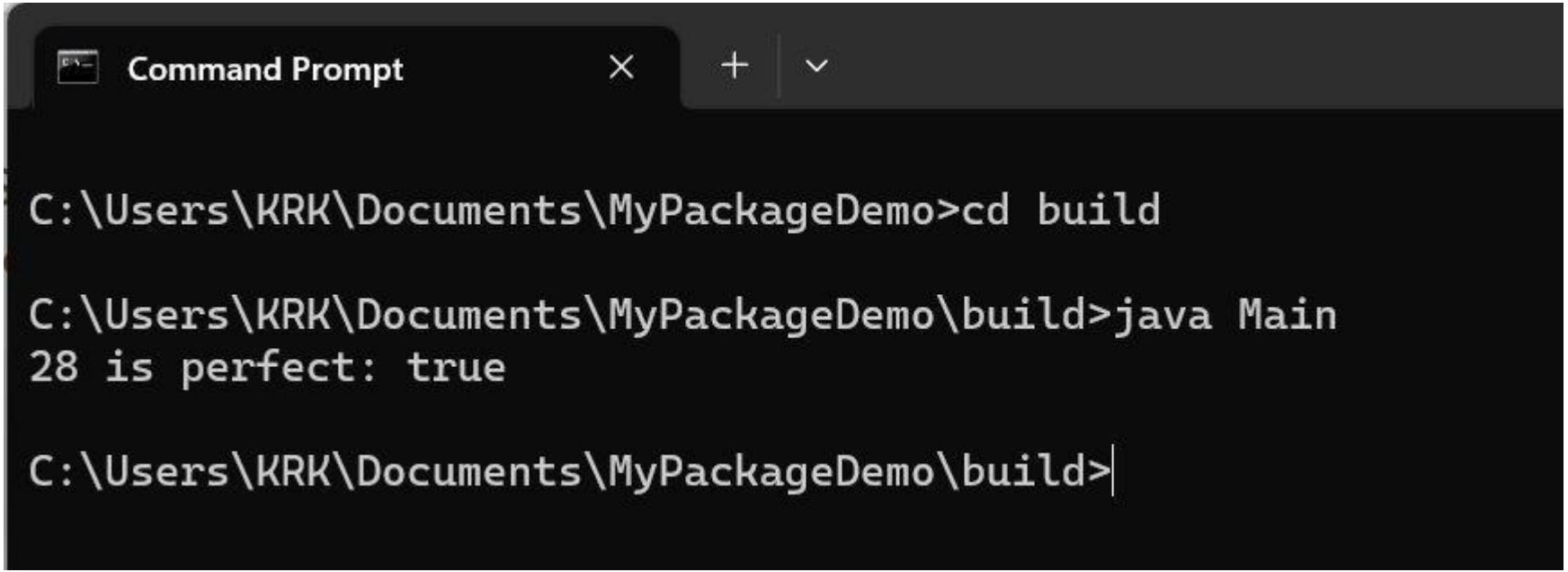
Create your Main / driver code

```
import com.mypackage.Perfect; // Import the package


public class Main {
    public static void main(String[] args) {
        int number = 28; // Example number to check for perfection
        boolean isPerfect = Perfect.checkPerfect(number);
        System.out.println(number + " is perfect: " + isPerfect);

    }
}
```

Compile the driver code


```
C:\Users\KRK\Documents\MyPackageDemo>javac -cp build -d build src/Main.java

C:\Users\KRK\Documents\MyPackageDemo>
```

Execute the driver code



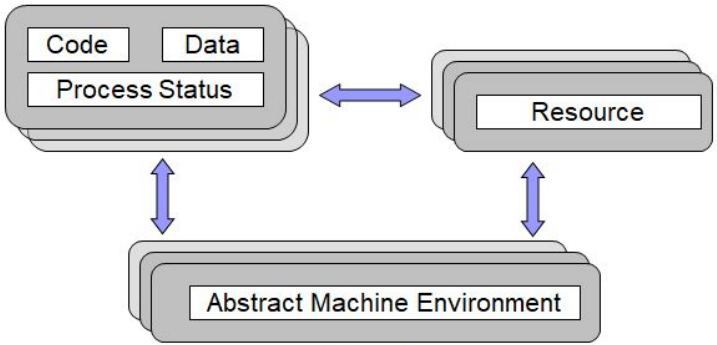
Multithreading in Java

Process

- It is executable program in memory
- Process Properties
  - Address space
  - Own program counter, execution context
  - Each process may execute a different program
  – May contain multiple threads

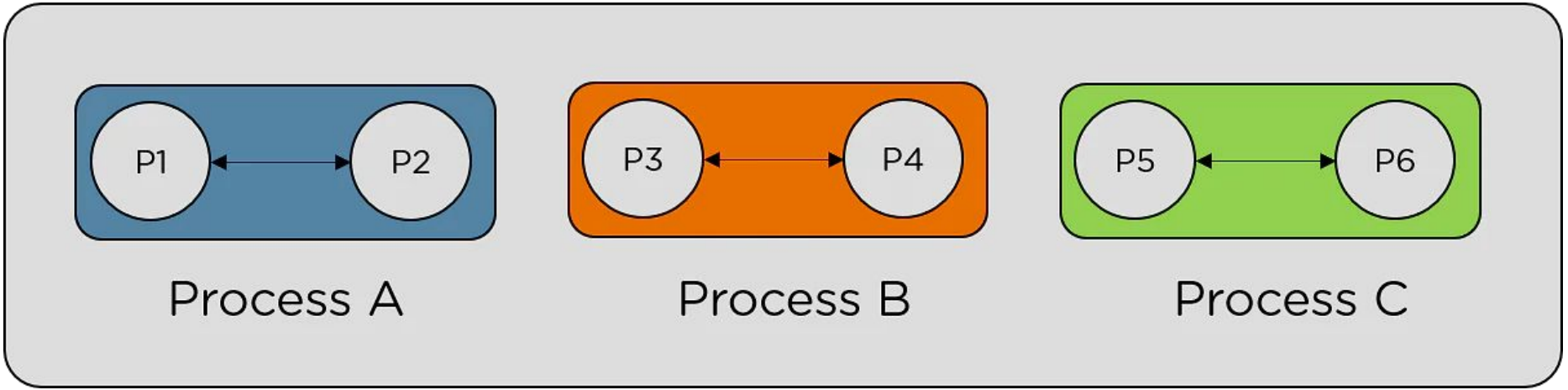
Process Model

- A **process** is a sequential program in execution.
- A **process** is a unit of computation.
- **Process components:**
  - The program (code) to be executed.
  - The data on which the program will execute.
  - Resources required by the program.
  - The status of the process execution.
- **A process runs in an abstract machine environment (could be OS) that manages the sharing and isolation of resources among the community of processes.**



Thread

- It is a lightweight process, a single sequentially executed stream of instructions
- Properties
  - Shares address space with other threads
  - Communicate via shared access to data
  - Multiple threads in process execute same program



A thread is the smallest segment of an entire process. A thread is an independent, virtual and sequential control flow within a process. In process execution, it involves a collection of threads, and each thread shares the same memory. Each thread performs the job independently of another thread.

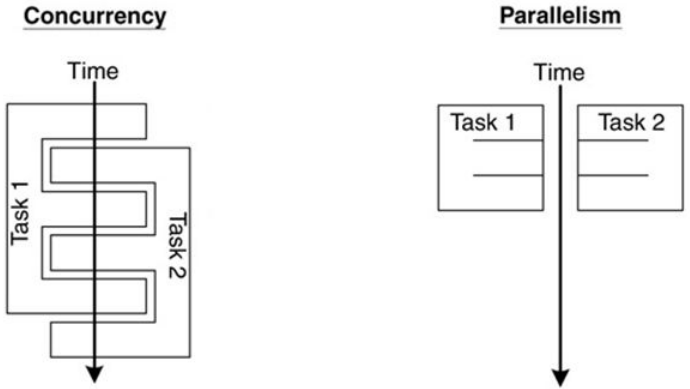
Program and Process

Program and process – distinction?

- A program is a **static entity** made up of program statements. The latter define the run-time behavior.
- A process is a **dynamic entity** that executes a program on a particular set of data.
- **Two or more processes could execute the same program**, each using their own data and resources.

Concurrency and Parallelism

- Concurrent multithreading systems **give the appearance of several tasks executing at once**, but these tasks are actually split up into chunks that share the processor with chunks from other tasks.
- In parallel systems, two tasks are actually performed simultaneously. **Parallelism requires a multi-CPU system.**

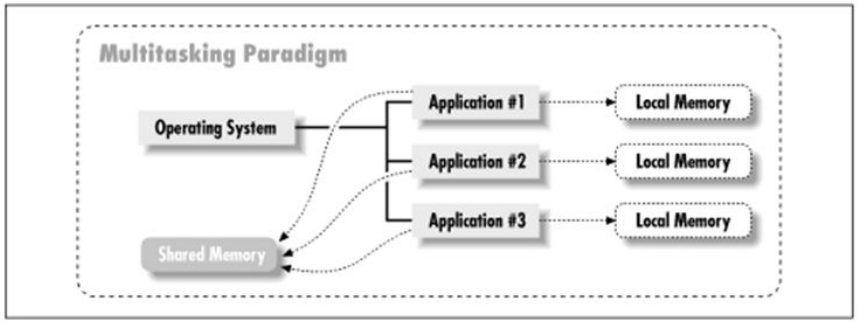


Multitasking

Multitasking operating systems **run multiple programs simultaneously**.

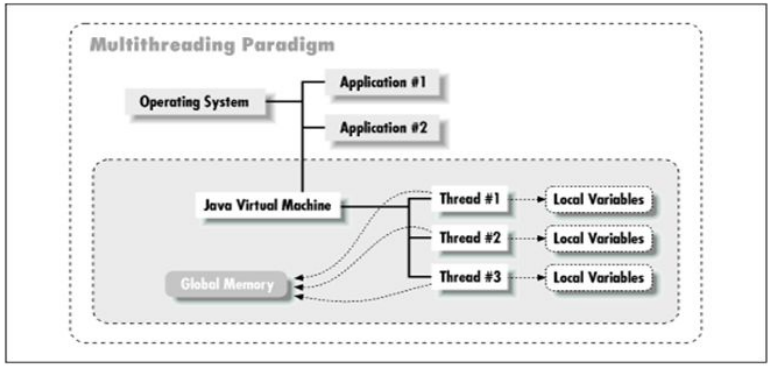
Each of these programs has at least one thread within it - **single-threaded** process:

- **The process begins execution** at a **well-known** point. In Java, C# or C++, the process begins execution at the first statement of the function called ***main()***.
- **Execution of the statements** follows in a completely **ordered, predefined sequence** for a given set of inputs.
- While executing, the process has access to certain data – local, global, static etc.

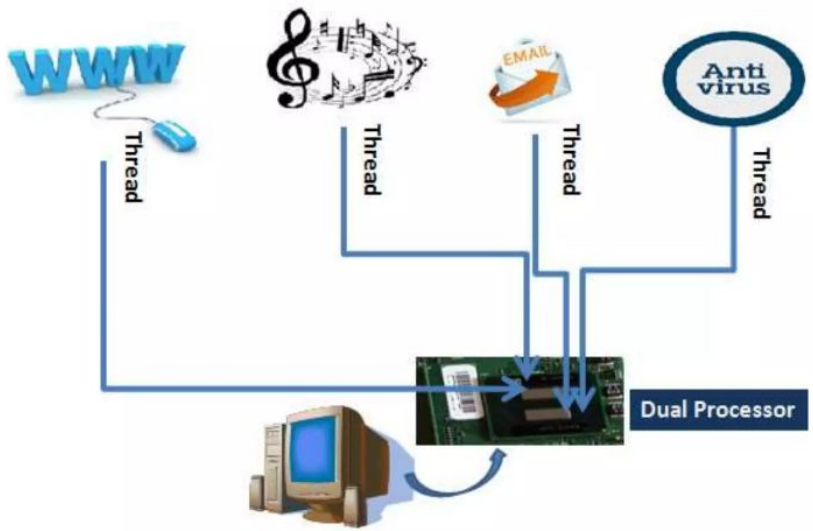


Multithreading

- **A program with multiple threads** running within a single instance **could be considered** as a multitasking system within an OS.
- In a multithreading program, threads have the following properties:
  - A thread begin execution at a predefined, well-known location. For one of the threads in the program, that location is the ***main()*** method; for the rest of the threads, it is a **particular location** the programmer decides on when the code is written.
  - A thread executes code in an **ordered, predefined sequence**.
  - A thread executes its code **independently** of the other threads.
  - The threads appear to have a **certain degree of simultaneous execution**.



MULTITHREADING Contd.



Multithreading On a Dual Processor Desktop System

Need for Multithreading

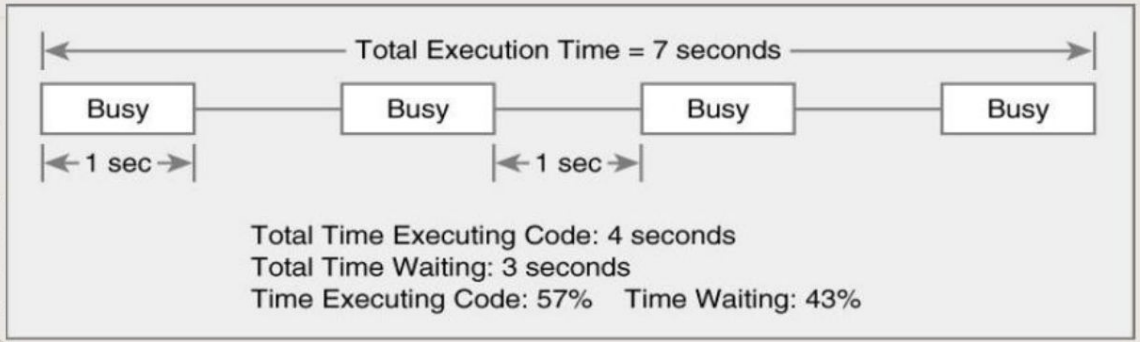
- Speed up the computation
  - Two threads each solve half of the problem and them combine their results
  - Speedup happens only on multiprocessors
- Faster Response
  - One thread computes while another handles user interface
  - One thread loads an image from Internet while another performs computation
- Performing multiple tasks simultaneously
  - One thread does garbage collection while other performs computation
  - Several thread performs animation simultaneously

Programming with Threads

- Concurrent programming
  - Writing programs divided into independent tasks
  - Tasks may be executed in parallel on multiprocessors
- Multithreading
  - Executing program with multiple threads in parallel
  - Special form of multiprocessing

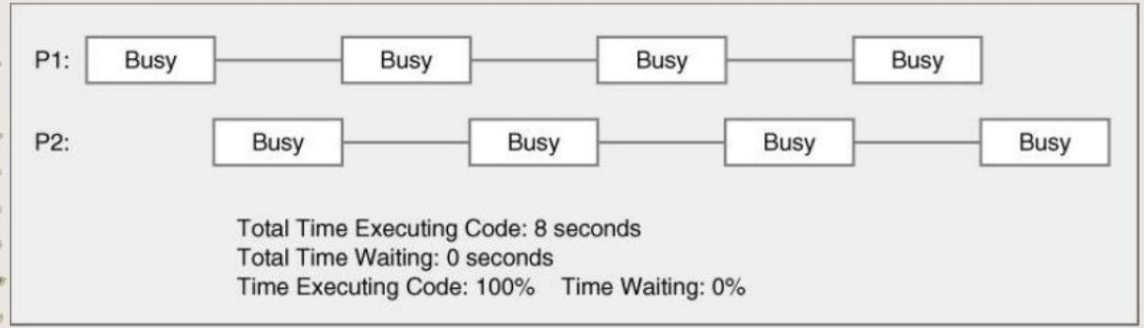
Multithreading

Single Thread



Total Time Executing Code: 4 seconds
Total Time Waiting: 3 seconds
Time Executing Code: 57% Time Waiting: 43%

Two Thread



Total Time Executing Code: 8 seconds
Total Time Waiting: 0 seconds
Time Executing Code: 100% Time Waiting: 0%

Thread Support in Java

- Two approaches
  - Thread class
  - Runnable interface

Thread Class

```
public class Thread extends Object
                                implements Runnable {
    public Thread();
    public Thread(String name);   // Thread name
    public Thread(Runnable R);    // Thread ⇒ R.run()
    public Thread(Runnable R, String name);

    public void run();
    public void start();        // begin thread execution
    ...
}
```

More Thread Class Methods

```
public class Thread extends Object {
    …
    public static Thread currentThread()
    public String getName()
    public void interrupt()
    public boolean isAlive()
    public void join()
    public void setDaemon()
    public void setName()
    public void setPriority()
    public static void sleep()
    public static void yield()
}
```

Thread Creations in Java

1. Thread class
   - Extend Thread class and override the run method

Example

```
public class newThread extends Thread {
   public void run() {
      …                          // code for each thread
                              }
   }
newThread T = new newThread() ;// To create a new thread
T.start();                 // begin running the new thread
…                          // thread executing in parallel
```

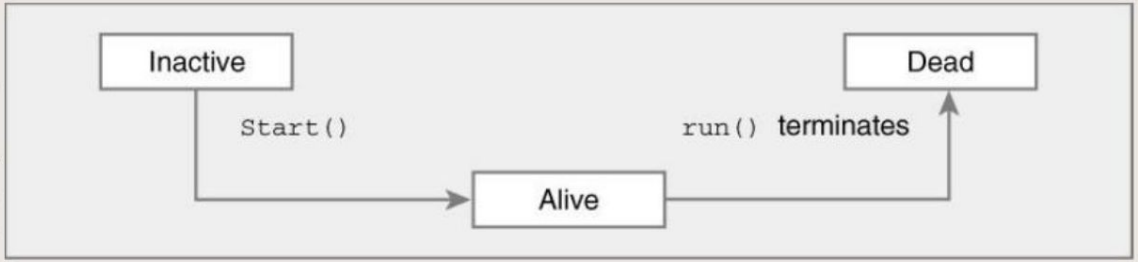
Thread Creations in Java

2. Runnable interface
   - Create object implementing Runnable interface
   - Pass it to Thread object via Thread constructor

Example

```
public class newThread implements Runnable {
   public void run() {
      …   // code for each thread
   }
}
Thread T = new Thread(new newThread);  // To create a new thread
T.start(); // begin running the new thread
…          // thread executing in parallel
```

Thread Creations in Java



- Runnable is interface
  - So it can be multiply inherited
  - Required for multithreading in applets

**Applet is a special type of program that is embedded in the webpage to generate the dynamic content.

Threads in Java

- There are two ways to create a java thread:
  - By extending the **java.lang.Thread** class.
  - By implementing the **java.lang.Runnable** interface.
- The ***run()*** method is where the action of a thread takes place.
- The execution of a thread starts by calling its ***start()*** method.

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime; }
    public void run() {
// compute primes larger than minPrime . . .
    }
}
```

- The following code would then create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

Implementing the Runnable Interface

- In order to create a new thread we may also provide a class that implements the **java.lang.Runnable** interface.
- Preferred way in case our class has to subclass some other class.
- A Runnable object can be wrapped up into a Thread object:
  - **Thread**(Runnable target)
  - **Thread**(Runnable target, String name)
- The thread's logic is included inside the **run()** method of the **runnable** object.

```
class ExClass
  extends ExSupClass
  implements Runnable {
    …
    public ExClass (String name) {
    }
    public void run() {
              …
    }
}
```

```
class A {
    …
    main(String[] args) {
        …
        Thread mt1 = new Thread(new ExClass("thread1”));
        Thread mt2 = new Thread(new ExClass("thread2”));
        mt1.start();
        mt2.start();
    }
}
```

Implementing the Runnable Interface

- Constructs a new thread object associated with the given ***Runnable*** object.
- The new Thread object's ***start()*** method is called to begin execution of the new thread of control.
- The reason we need to pass the runnable object to the thread object's constructor is that the thread must have some way to get to the ***run()*** method we want the thread to execute. Since we are no longer overriding the ***run()*** method of the ***Thread*** class, the default ***run()*** method of the ***Thread*** class is executed:

```
public void run() {
        if (target != null) {
                target.run();
        }
}
```

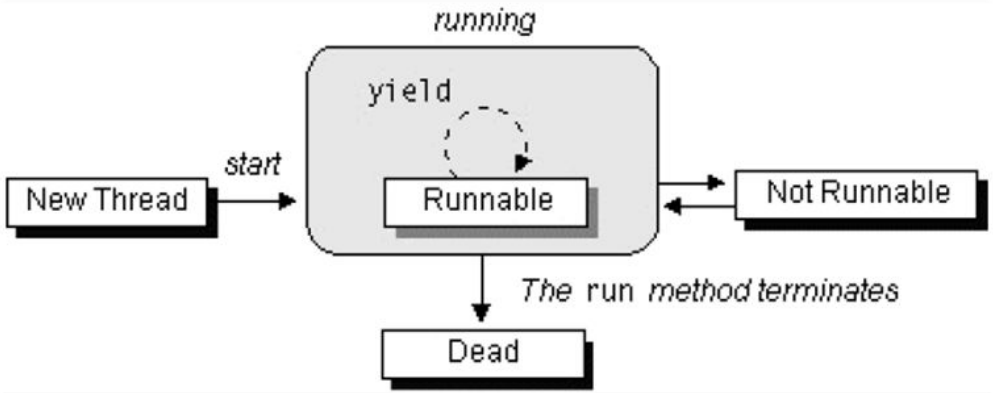
- Here, target is the runnable object we passed to the thread's constructor. So the thread begins execution with the ***run()*** method of the ***Thread*** class, which immediately calls the ***run()*** method of our runnable object.

Sleep, Yield, Notify & Wait Thread's Functions

- ***sleep(long millis)*** - causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- ***yield()*** - causes the currently executing thread object to temporarily pause and allow other threads to execute.
- ***wait()*** - causes current thread to wait for a condition to occur (*another thread invokes the* ***notify()*** *method or the* ***notifyAll()*** *method for this object*). This is a method of the ***Object*** class and must be called from within a **synchronized** method or block.
- ***notify()*** - notifies a thread that is waiting for a condition that the condition has occurred. This is a method of the ***Object*** class and must be called from within a **synchronized** method or block.
- ***notifyAll()*** – like the ***notify()*** method, but notifies all the threads that are waiting for a condition that the condition has occurred.

The Lifecycle of a Thread

- The ***start()*** method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's ***run()*** method.
- A thread becomes **Not Runnable** when one of these events occurs:
  - Its ***sleep()*** method is invoked.
  - The thread calls the ***wait()*** method.
  - The thread is blocked on I/O operations.
- A thread dies naturally when the ***run()*** method exits.



Multithreading

:: Thread Life Cycle

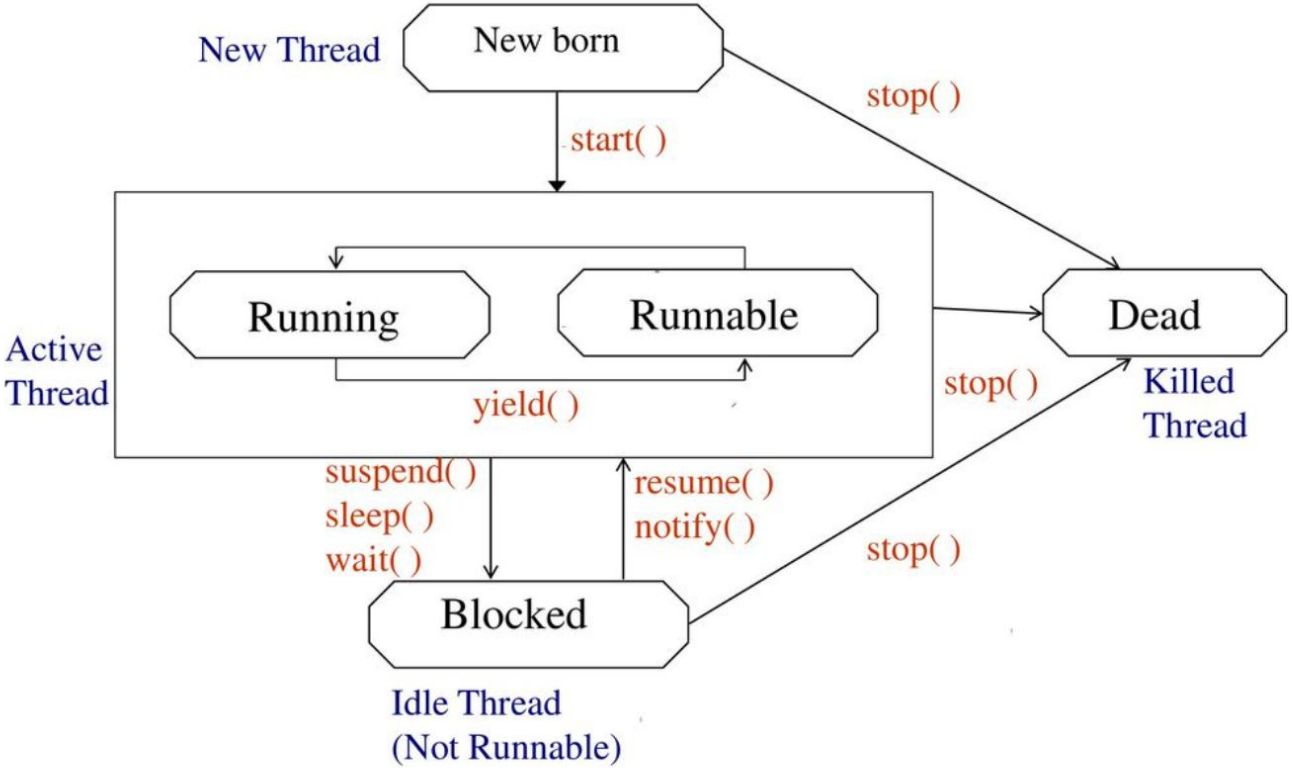


Fig:- State Transition diagram of a thread

Thread States

- Java thread can be in one of these states
  - New – thread allocated & waiting for start()
  - Runnable – thread can begin execution
  - Running – thread currently executing
  - Blocked – thread waiting for event (I/O, etc.)
  - Dead – thread finished
- Transitions between states caused by
  - Invoking methods in class Thread
    - new(), start(), yield(), sleep(), wait(), notify()…
  - Other (external) events
    - Scheduler, I/O, returning from run()…

Lifecycle of a Thread in Java

New

The first stage is "New". This stage is where it initiates the thread. After that, every thread remains in the new state until the thread gets assigned to a new task.

Runnable

Here, a thread gets assigned to the task and sets itself for running the task.

Running

The third stage is the **execution** stage. Here, the thread gets triggered as control enters the thread, and the thread performs a task and continues the execution until it finishes the job.

Waiting

At times, there is a possibility that one process as a whole might depend on another. During such an encounter, the thread might **halt for an intermediate result** because of its **dependency** on a different process. This stage is called the Waiting Stage.

Dead

The **final stage** of the process execution with Multithreading in Java is thread termination. After it terminates the process, the JVM automatically declares the thread dead and terminates the thread. This stage is known as the dead thread stage.

Thread Priority

- On a single CPU, threads actually run one at a time in such a way as to provide an **illusion of concurrency**.
- Execution of multiple threads on a single CPU, in some order, is called **scheduling**.
- The Java runtime supports a very simple **scheduling algorithm** (fixed priority scheduling). This algorithm schedules threads based on their priority relative to other runnable threads.
- The runtime system chooses the runnable thread with the **highest** priority for execution.

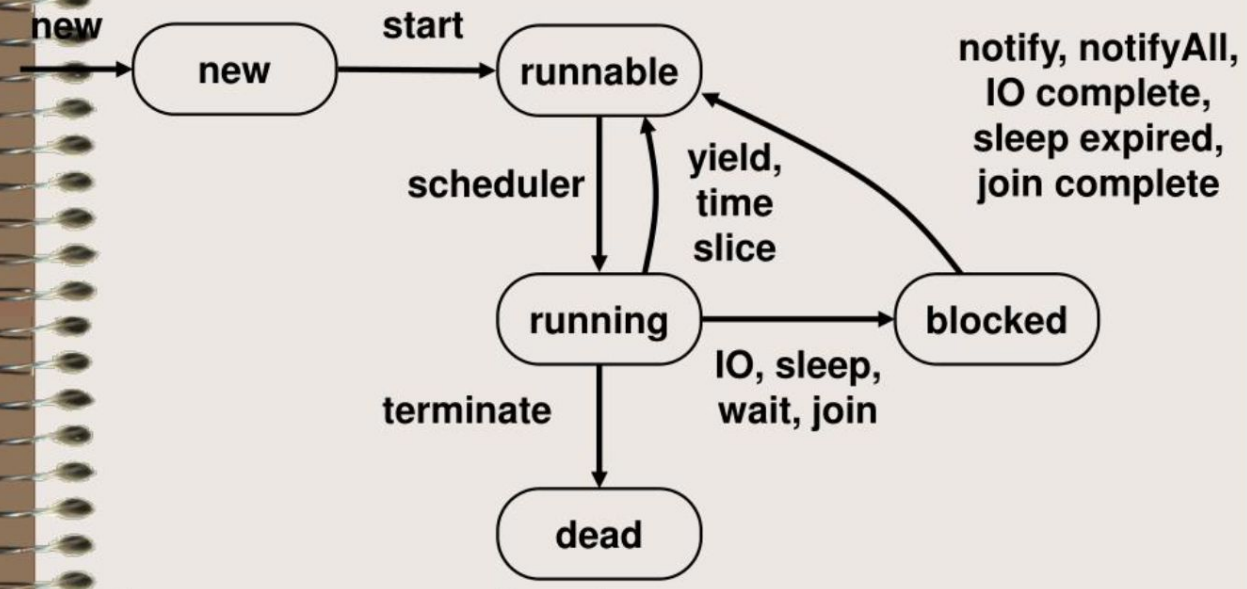
Thread Types

- Java threads types
  - User
  - Daemon
    - Provide general services
    - Typically never terminate
    - Call setDaemon() before start()
- Program termination
  1. All user threads finish
  2. Daemon threads are terminated by JVM
  3. Main program finishes

In Java, **daemon threads** are low-priority threads that run in the background to perform tasks such as garbage collection or provide services to user threads.

The life of a daemon thread depends on the mercy of user threads, meaning that when all user threads finish their execution, the Java Virtual Machine (JVM) automatically terminates the daemon thread.

Thread States



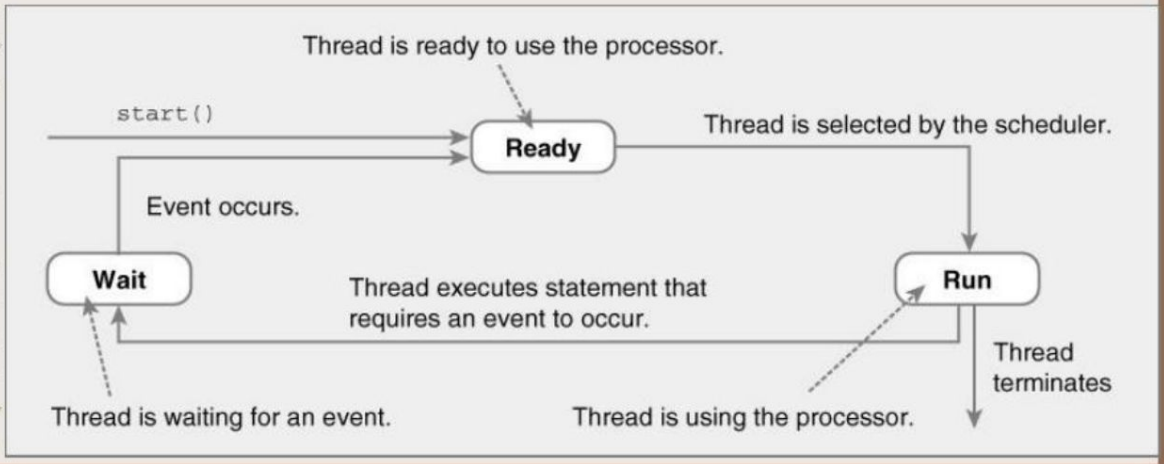
State diagram

Thread – Scheduling

- Scheduler
  - Determines which runnable threads to run
  - Can be based on thread priority
  - Part of OS or Java Virtual Machine (JVM)
- Scheduling policy
  - Nonpreemptive (cooperative) scheduling
  - Preemptive scheduling

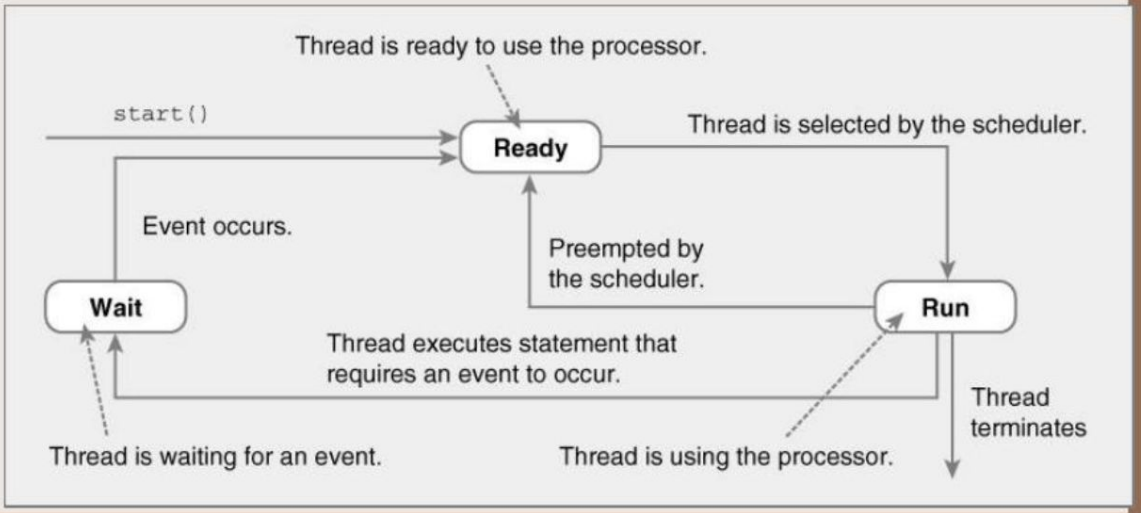
Non-preemptive Scheduling

- Threads continue execution until
  - Thread terminates
  - Executes instruction causing wait (e.g., IO)
  - Thread volunteering to stop (invoking yield or sleep)



Preemptive Scheduling

- Threads continue execution until
  - Same reasons as non-preemptive scheduling
  - Preempted by scheduler



Methods of Multithreading in Java

start()	The start method initiates the execution of a thread
currentThread()	The currentThread method returns the reference to the currently executing thread object.
run()	The run method triggers an action for the thread
isAlive()	The isAlive method is invoked to verify if the thread is alive or dead
sleep()	The sleep method is used to suspend the thread temporarily
yield()	The yield method is used to send the currently executing threads to standby mode and runs different sets of threads on higher priority

Suspend()	The suspend method is used to instantly suspend the thread execution
resume()	The resume method is used to resume the execution of a suspended thread only
interrupt()	The interrupt method triggers an interruption to the currently executing thread class
destroy()	The destroy method is invoked to destroy the execution of a group of threads
stop()	The stop method is used to stop the execution of a thread

Advantages of Multithreading in Java

- Multithreading in Java improves the performance and reliability
- Multithreading in Java minimizes the execution period drastically
- There is a smooth and hassle-free GUI response while using Multithreading in Java
- The software maintenance cost is lower
- The CPU and other processing resources are modes judiciously used

Disadvantages of Multithreading in Java

- Multithreading in Java poses complexity in code debugging
- Multithreading in Java increases the probability of a deadlock in process execution
- The results might be unpredictable in some worst-case scenarios
- Complications might occur while code is being ported

Java Thread Example

```
public class newThread extends Thread {
   public void run() {
      for (int i = 0; i < 3; i++)
         System.out.println(i);
         try {
            sleep((int)(Math.random() * 8000)); // 8 secs
         } catch (InterruptedException e) { }
   }
   public static void main(String[] args) {
      new newThread().start();
      new newThread().start();
      System.out.println("Done");
   }
}
```

Java Thread Example – Output

- Possible outputs
  - 0,1,2,0,1,2,Done // thread 1, thread 2, main()
  - 0,1,2,Done,0,1,2 // thread 1, main(), thread 2
  - Done,0,1,2,0,1,2 // main(), thread 1, thread 2
  - 0,0,1,1,2,Done,2 // main() & threads interleaved

Summary

- Thread is a lightweight process
- Faster in execution
- Saves memory
- Reduces the response time
- Scheduling a big problem

Thread creation by extending the Thread class

We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

```
// Java code for thread creation by extending
// the Thread class
class MultithreadingDemo extends Thread {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
public class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            MultithreadingDemo object
                = new MultithreadingDemo();
            object.start();
        }
    }
}
```

Output

```
Thread 15 is running
Thread 14 is running
Thread 16 is running
Thread 12 is running
Thread 11 is running
Thread 13 is running
Thread 18 is running
Thread 17 is running
```

Thread creation by implementing the Runnable Interface

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

```
// Java code for thread creation by implementing
// the Runnable Interface
class MultithreadingDemo implements Runnable {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            Thread object
                = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

Output

```
Thread 13 is running
Thread 11 is running
Thread 12 is running
Thread 15 is running
Thread 14 is running
Thread 18 is running
Thread 17 is running
Thread 16 is running
```

Thread Class vs Runnable Interface

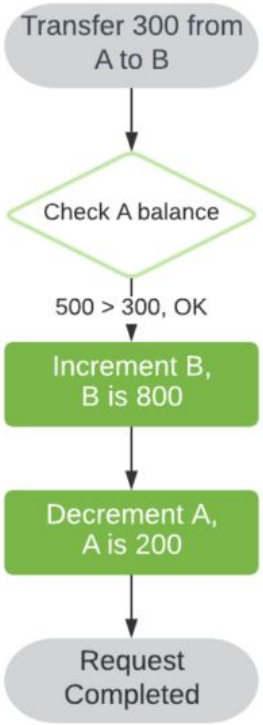
1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
3. Using runnable will give you an object that can be shared amongst multiple threads.

Codes:
https://docs.google.com/document/d/1tM6La9VpnZNpBbQ9tcV11IMaogQTkrpxjKrWp1_O1EU/edit?usp=sharing

Race Condition

A race condition is a condition of a program where its behavior depends on relative timing or interleaving of multiple threads or processes. One or more possible outcomes may be undesirable, resulting in a bug. We refer to this kind of behavior as **nondeterministic**.

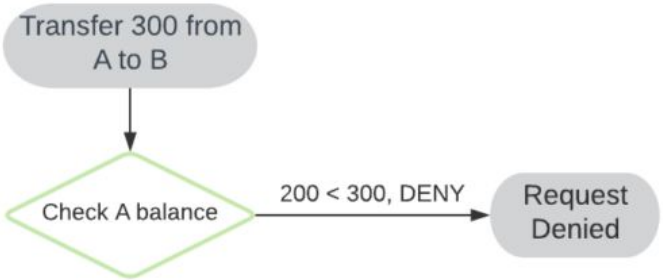
Thread-safe is the term we use to describe a program, code, or data structure free of race conditions when accessed by multiple threads.

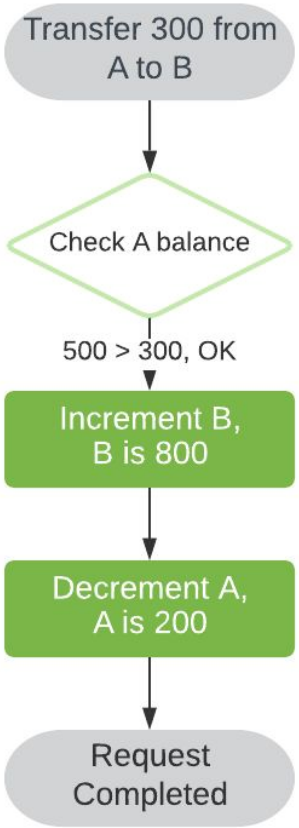


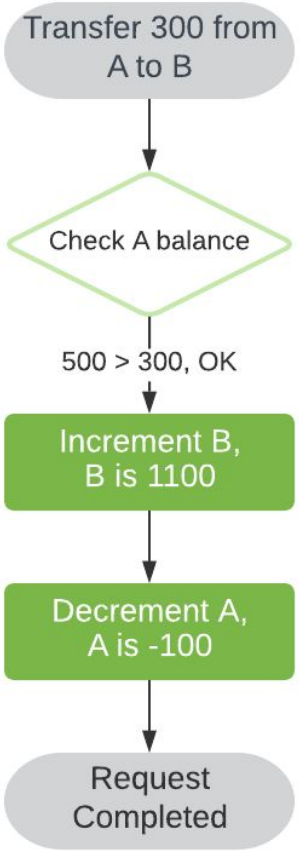
Let's consider a simple function for performing a funds transfer between two bank accounts:

Let's say we have accounts A and B, each having a balance of 500, and we perform two attempts to transfer 300 from A to B:

However, if these two attempts are kicked off simultaneously, in different processes or threads, we may observe some undesired behavior:







Given unpredictable thread scheduling, the order of specific steps is arbitrary. We will encounter a race condition due to the **interleaving** of our execution flows.

Thread Synchronization

- To solve the problem of race condition (data races)
- Two ways in Java
  - Synchronized methods
  - Synchronized statements

Synchronization

Background

- **Concurrent access** to shared data may result in **data inconsistency**.
- Maintaining data consistency requires mechanisms to ensure the **orderly execution** of cooperating processes (or threads).

When do we need synchronization?

When two or more processes (or threads) work on the same data simultaneously.

Synchronization

Example:

Two threads are trying to **update the same shared variable simultaneously:**

- The result is **unpredictable**.
- The result depends on which of the two threads was the last one to change the value.
- The competition of the threads for the variable is called **race condition.**
- The **first** thread is the one who **wins the race to update** the variable.

Java Thread Synchronization

In multithreading, there is the asynchronous behavior of the programs. If one thread is writing some data and another thread which is reading data at the same time, might create inconsistency in the application.

When there is a need to access the shared resources by two or more threads, then synchronization approach is utilized.

Java has provided synchronized methods to implement synchronized behavior.

In this approach, once the thread reaches inside the synchronized block, then no other thread can call that method on the same object. All threads have to wait till that thread finishes the synchronized block and comes out of that.

In this way, the synchronization helps in a multithreaded application. One thread has to wait till other thread finishes its execution only then the other threads are allowed for execution.

It can be written in the following form:

```
Synchronized(object)
{
        //Block of statements to be synchronized
}
```

Example

```
public class Counter{
    private int count = 0;
    public int getCount(){
            return count;
    }

    public setCount(int count){
            this.count = count;
    }
}
```

- In this example, the counter tells how many an access has been made.
- If a thread is accessing setCount and updating count and another thread is accessing getCount at the same time, there will be inconsistency in the value of count.

Fixing the example

```
public class Counter{
    private static int count = 0;
    public synchronized int getCount(){
            return count;
    }

    public synchoronized setCount(int count){
             this.count = count;
    }
}
```

- By adding the synchronized keyword we make sure that when one thread is in the setCount method the other threads are all in waiting state.
- The synchronized keyword places a lock on the object, and hence locks all the other methods which have the keyword synchronized. The lock does not lock the methods without the keyword synchronized and hence they are open to access by other threads.

Synchronization of Java Threads

- In many cases **concurrently** running threads share data and must consider the state and activities of other threads.
- If two threads can both execute a method that modifies the state of an object then the method should be declared to be ***synchronized***, those allowing only one thread to execute the method at a time.
- If a class has at least one ***synchronized*** method, each instance of it has a **monitor**. A monitor is an object that can **block** threads and **notify** them when the method is available.

Example:

```
public synchronized void updateRecord() {
//**** critical code goes here …
}
```

- Only one thread may be inside the body of this function. A second call will be blocked until the first call returns or ***wait()*** is called inside the synchronized method.

Synchronization of Java Threads

- If you don't need to protect an entire method, you can synchronize on an object:

```
public void foo() {
        synchronized (this) {
        //critical code goes here …
        }
        …
}
```

- There are **two syntactic forms** based on the **synchronized** keyword - blocks and methods.
- **Block synchronization** takes an argument of which object to lock. This allows **any method to lock any object.**
- The most common argument to synchronized blocks is **this.**
- Block synchronization is considered more fundamental than method synchronization.

Synchronization of Java Threads

- To program the synchronization behavior we use the Object class' methods ***wait()***, ***notify()*** and ***notifyAll()***.

- With these methods we allow objects to wait until another object notifies them:

```
synchronized( waitForThis ) {
        try { waitForThis.wait();}
        catch (InterruptedException ie) {}
}
```

- To wait on an object, you must first **synchronize** on it.
- ***InterruptedException*** is thrown when a thread is waiting, sleeping, or otherwise paused for a long time and another thread interrupts it using the interrupt method in class Thread.

Synchronization of Java Threads

- A thread may call ***wait()*** inside a **synchronized** method. A timeout may be provided. If missing or zero then the thread waits until either ***notify()*** or ***notifyAll()*** is called, otherwise until the timeout period expires.

- ***wait()*** is called by the thread owning the lock associated with a particular object.

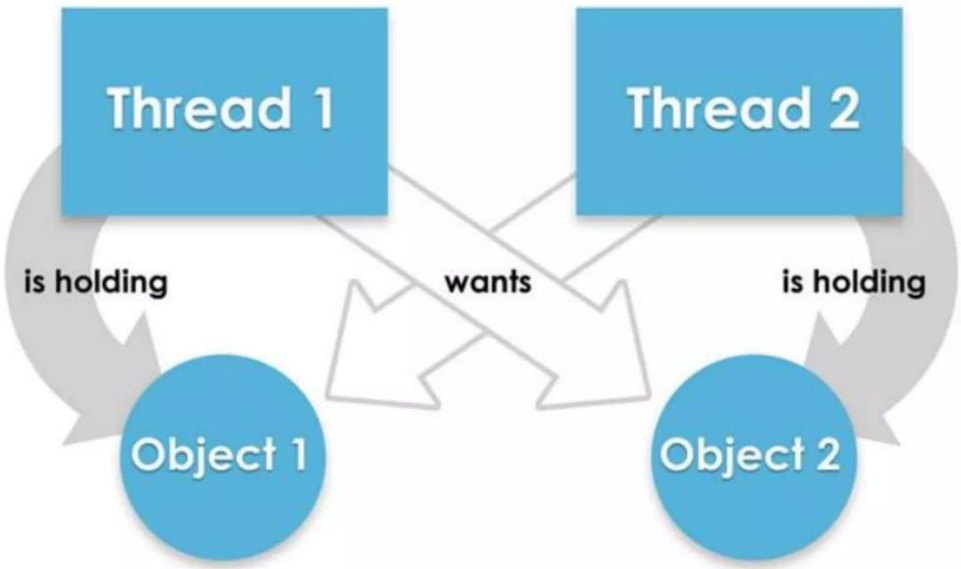
- ***notify()*** or ***notifyAll()*** are only called from a **synchronized** method. One or all waiting threads are notified, respectively. It's probably better (safer) to use ***notifyAll().*** These methods don't release the lock. The threads awakened will not return from their ***wait()*** call immediately, but only when the thread that called ***notify()*** or ***notifyAll()*** finally relinquishes ownership of the lock.

deadlock

- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other.
- when two or more threads are waiting to gain control on a resource.

For example, assume that the thread A must access Method1 before it can release Method2, but the thread B cannot release Method1 until it gets holds of Method2.

deadlock


Thread 1
Thread 2
is holding
wants
is holding
Object 1
Object 2

Coding Activity

- WAJP to create a Thread, and use **atleast 4** methods that represent its life cycle; like, 'yield', 'sleep', 'join', start', 'interrupt', 'run', etc.
- WAJP to demonstrate the use of 'synchronized' keyword in multithreading.

Example codes of demonstration of *state of thread* and *different methods of Thread class:*
https://docs.google.com/document/d/1tM6La9VpnZNpBbQ9tcV11IMaogQTkrpxjKrWp1_O1EU/edit?usp=sharing

QUESTIONS?