DAYANANDA SAGAR COLLEGE OF ENGINEERING
Department of Information Science and Engineering

Java and J2EE

(18IS6DCJVA)

Unit 3

Packages and Interfaces

Introduction

- The main feature of OOP is its ability to support the reuse of code:
  - Extending the classes (via inheritance)
  - Extending interfaces
- The features in basic form limited to reusing the classes within a program.
- What if we need to use classes from other programs without physically copying them into the program under development ?
- In Java, this is achieved by using what is known as “packages”, a concept similar to “class libraries” in other languages.

Introduction

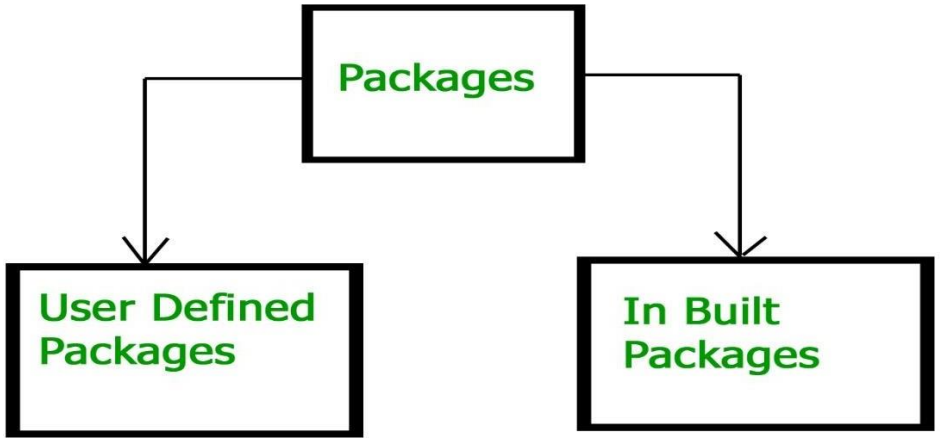
- Packages are Java's way of grouping a number of related classes and/or interfaces together into a single unit. That means, packages act as "containers" for classes.

- A package is a mechanism to group the similar type of classes, interfaces and sub-packages and provide access control. It organizes classes into single unit.

- The benefits of organising classes into packages are:
  - The classes contained in the packages of other programs/applications can be reused.
  - It helps in resolving naming collision when multiple packages have classes with the same name.
  - Classes in packages can be hidden if we don't want other packages to access them.
  - One package can be defined in another package.

Types of Packages

1. **Built-in packages :** are already defined in java API.
2. **User defined packages :** The package we create according to our need is called user defined package.



Java Foundation Packages

- Java provides a large number of classes groped into different packages based on their functionality.
- The six foundation Java packages are:
  - java.lang
    - Contains classes for primitive types, strings, math functions, threads, and exception
  - java.util
    - Contains classes such as vectors, hash tables, date etc.
  - java.io
    - Stream classes for I/O
  - java.awt
    - Classes for implementing GUI – windows, buttons, menus etc.
  - java.net
    - Classes for networking
  - java.applet
    - Classes for creating and implementing applets

Creating a Package

- We can create our own package by creating our own classes and interfaces together.
- The package statement should be declared at the beginning of the program.
- **Syntax:**

```
package <packagename>;
class ClassName
{
……..
……..
}
```

Example: Creating a Package

- **// Demo.java**

```
package p1;
class Demo
{
   public void m1()
   {
      System.out.println("Method m1..");
   }
}
```

How to comZpile and Run?

- **Syntax:** javac –d directory javafilename
- **For Example:** javac –d . demo.java
- **To run:** java demo

Example: Program to create and use a user defined package in Java.

Save as A.java

```
package pack;

public class A
{
public void show()
{
	System.out.println("A.class");

}

}
```

Save as B.java

```
package mypack;
import pack.A;
public class B {
public static void main(String args[])
{
	A obj = new A();
	obj.show();
	  }
}
```

The “import” keyword

- The import keyword provides the access to other package classes and interfaces in current packages.
- “import” keyword is used to import built-in and user defined packages in java program.
- There are different 3 ways to access a package from other packages.

1. Using full qualified name
2. import only single class

3. import all classes

Using fully qualified name

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example :

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
```

Output:

Hello

```
}
```

Import only single class

- If you import package.classname then only declared class of this package will be accessible.

Example :

```
//save by A.java

package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}                                                                    }
//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
```

Hello

Output:

}

Import all classes

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

Example :

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}                                                                    }
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
```

Hello

Output:

}

Protection and Packages

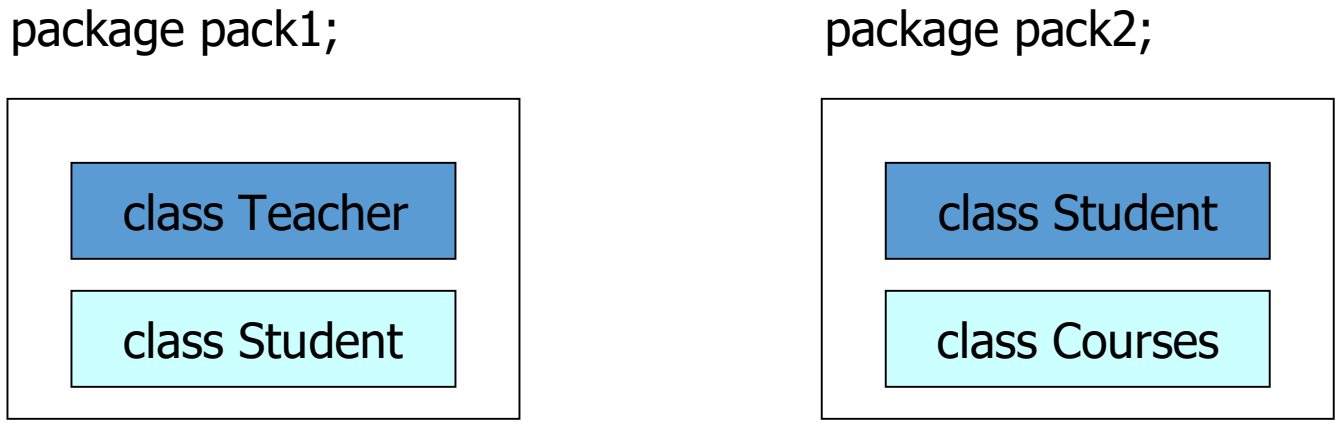
- All classes (or interfaces) accessible to all others in the same package.
- Class declared public in one package is accessible within another.
- Members of a class are accessible from a difference class, as long as they are not *private*
- *protected* members of a class in a package are accessible to subclasses in a different class

Visibility Modifiers

Accessible to:	public	protected	Package (default)	private
Within the class	Yes	Yes	Yes	Yes
Within package	Yes	Yes	Yes	No
Subclass in different package	Yes	Yes	No	No
Non-subclass different package	Yes	No	No	No

Packages and Name Clashing

- When packages are developed by different organizations, it is possible that multiple packages will have classes with the same name, leading to name classing.



- We can import and use these packages like:
  - import pack1.*;
  - import pack2.*;

- Student student1; // Generates compilation error

Handling Name Clashing

- In Java, name clashing is resolved by accessing classes with the same name in multiple packages by their fully qualified name.

- Example:

```
import pack1.*;
import pack2.*;
pack1.Student student1;
pack2.Student student2;
Teacher teacher1;
```

Courses course1;

Important points

- Every class is part of some package.
- If no package is specified, the classes in the file goes into a special unnamed package (the same unnamed package for all files).
- All classes/interfaces in a file are part of the same package. Multiple files can specify the same package name.
- If package name is specified, (i.e., the directory name must match the package name).
- We can access public classes in another (named) package using: **package-name.class-name**

/Interfaces

Introduction

- An interface is basically a kind of class.
- Like classes, interfaces contains methods and variables.
- The difference is that interfaces define only abstract methods and final variables.
- Interface do not specify any code to implement the methods.
- Interfaces are designed to support dynamic method resolution at runtime.

Interface

- It is used to achieve abstraction and multiple inheritance in Java.
- There can be only abstract methods in the Java interface, not method body.
- It cannot be instantiated just like the abstract class.
- All the fields are public, static and final by default.
- All the interface methods are by default **abstract and public**.
- Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.
- A class that implements an interface must implement all the methods declared in the interface.
- Syntax:

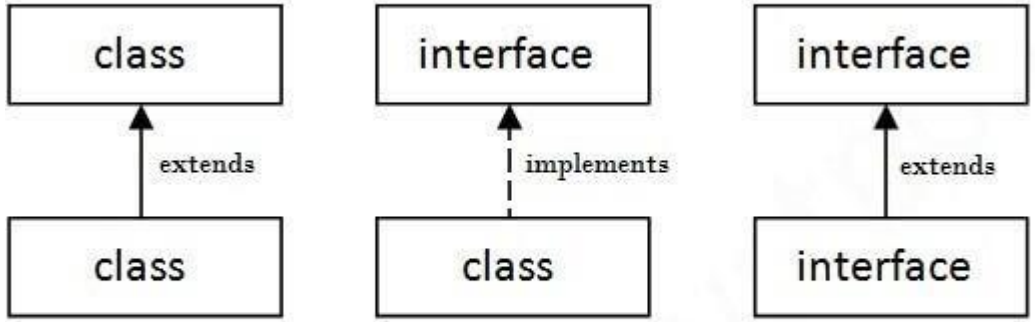
```
interface <interface_name>{

    // declare constant fields
    // declare methods that are abstract by default.
```

```
}
```

The relationship between classes and interfaces

- A class extends another class, an interface extends another interface, but a **class implements an interface**.



Example

```
interface Printable{
    void print();
}
interface Showable{
    void show();
}
class A implements Printable,Showable{
    public void print(){System.out.println("Hello");}
    public void show(){System.out.println("Welcome");}

    public static void main(String args[]){
        A obj = new A();
        obj.print();
        obj.show();
```

Output:
Hello
Welcome

```
        }
    }
```

Example

```
interface Callback
{
    void callback(int param);
}
```

```
class Client implements Callback
{
    // Implement Callback's interface
    public void callback(int p)
    {
            System.out.println("callback called with " + p);
    }
}
```

When we implement an interface method, it must be declared as public.

We can also define additional members.

```
class Client implements Callback
{
    // Implement Callback's interface
    public void callback(int p)
    {
        System.out.println("callback called with " + p);
    }
    void nonIfaceMeth()
    {
        System.out.println(“Added method");
    }
}
```

Accessing Implementations Through Interface References

```
class TestIface
 {
    public static void main(String args[])
    {
                Callback c = new Client();
                c.callback(42);
    }
}
```

c can be used to access the callback() method, it can not access any other members of the Client class.

```
// Another implementation of Callback.
class AnotherClient implements Callback
{
    // Implement Callback's interface
    public void callback(int p)
    {
            System.out.println("Another version of callback“+(p+10));
    }
}
```

```
class TestIface2
{
    public static void main(String args[])
    {
            AnotherClient ob = new AnotherClient();

            ob.callback(42);
    }
}
```

Interfaces Can Be Extended

```
// One interface can extend another.
interface A
 {
            void meth1();
}
// B now includes meth1() and meth2() .
interface B extends A
{
    void meth2();
}
```

```
// This class must implement all of A and B
class MyClass implements B
{
    public void meth1()
    {
            System.out.println("Implement meth1().");
    }

    public void meth2()
    {
            System.out.println("Implement meth2().");
    }
}
```

```
class prg
 {
    public static void main(String arg[])
    {
                MyClass ob = new MyClass();
                ob.meth1();
                ob.meth2();

    }
}
```

Any class that implements an interface must implement all methods defined by that interface.

Example: Sample program to implement multiple inheritance

```
interface Vehicle
{
  void run();
}
interface Bike extends Vehicle
{
  void stop();
}
public class Demo implements Bike
{
  public void run()
  {
    System.out.println("Vehicle is running.");
  }
  public void stop()
  {

    System.out.println("Bike is stopped.");
  }
  public static void main(String  args[])
  {
    Demo obj = new Demo();
    obj.run();
```

Output:

Vehicle is running.
Bike is stopped.

```
    obj.stop();
  }
}
```

Differences between Abstract class and Interface

Abstract class	Interface
It cannot support multiple inheritances.	Interface supports multiple inheritances.
It contains both abstract and non abstract method.	It contains only abstract method.
Abstract class is the partially implemented class.	Interface is fully unimplemented class.
It can have main method and constructor.	It cannot have main method and constructor.
It can have static, non-static, final, non-final variables.	It contains only static and final variable.

Exception Handling

Introduction

- The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

- What is Exception in Java ?
  - **Dictionary Meaning:** Exception is an abnormal condition.
  - In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

- What is Exception Handling
  - Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

- The core advantage of exception handling is **to maintain the normal flow of the application**.

- An exception normally disrupts the normal flow of the application that is why we use exception handling.

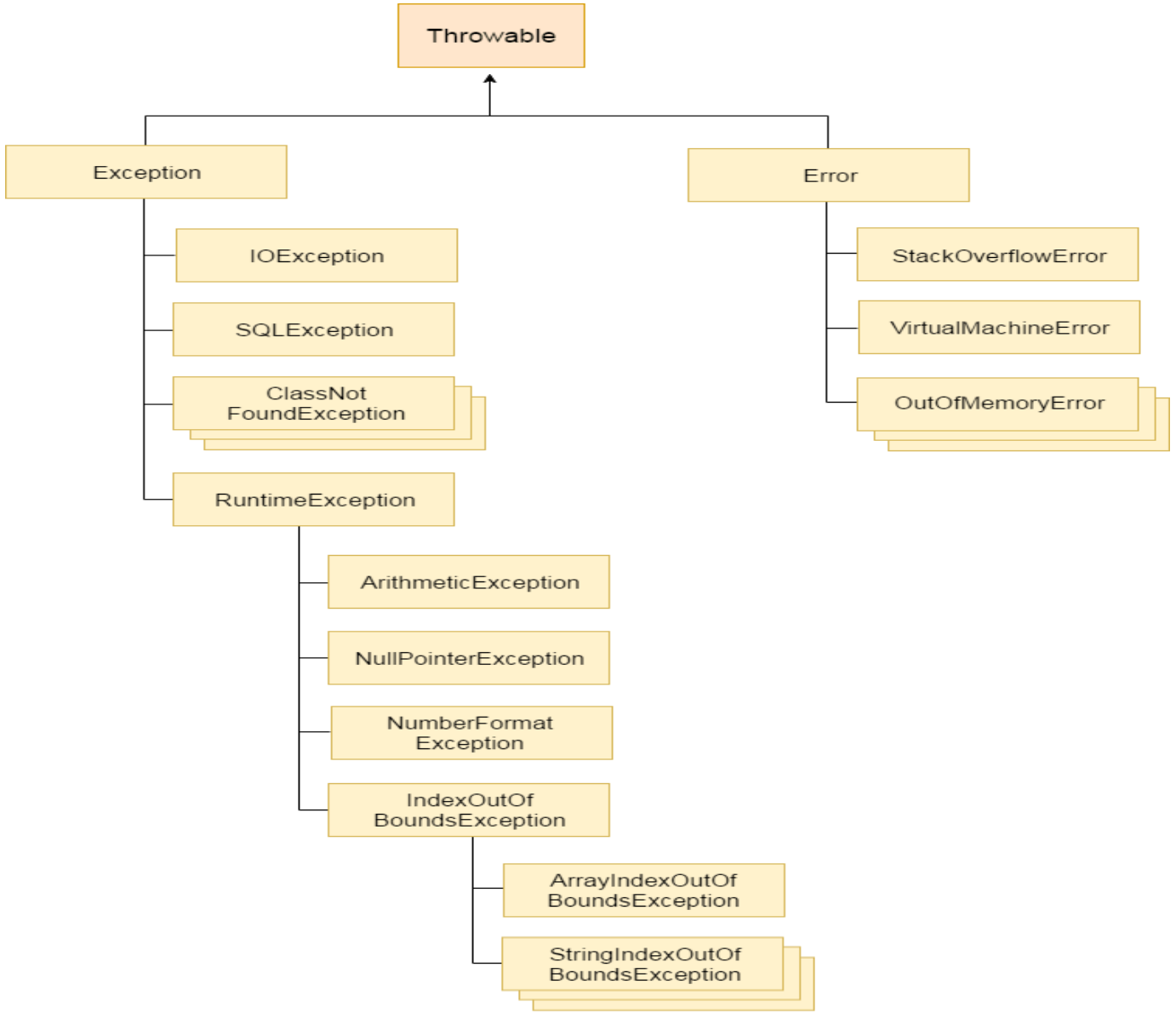
- Let's take a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Note: Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed

i.e. statement 6 to 10 will not be executed.If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes


Throwable
Exception
Error
IOException
SQLException
ClassNot FoundException
RuntimeException
StackOverflowError
VirtualMachineError
OutOfMemoryError
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOf BoundsException
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException

Types of Java Exceptions

- There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception.

- Types of exceptions:
  - Checked Exception
  - Unchecked Exception

Difference between Checked and Unchecked Exceptions

1) Checked Exception

- The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc.
- Checked exceptions are checked at compile-time.

2) Unchecked Exception

- The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
- Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Java's Built-in Exceptions

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.

Java's Unchecked RuntimeException *Subclasses*

Exception	Meaning
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Java's Unchecked RuntimeException *Subclasses* (continued)

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Java's Checked Exceptions Defined in java.lang

Java Exception Keywords

- There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
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 important 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 doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

An *exception is an* abnormal condition that arises in a code sequence at run time.

An exception is a run-time error.

In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on.

Java's exception handling avoids these problems.

This is the general form of an exception-handling block:

```
try
 {
        // block of code to monitor for errors
}
catch (ExceptionType1    exOb)
{
        // exception handler for ExceptionType1
}
catch (ExceptionType2    exOb)
{
        // exception handler for ExceptionType2
}
// ...
finally
{
        // block of code to be executed before try block ends
}
```

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.

When an exceptional condition arises, an object representing that exception is created and *thrown in the method that caused the error.*

Exceptions can be generated by the Java run-time system, or they can be manually generated by our code.

Example:

```
class Exc0
    {
            public static void main(String args[])
            {
                    int d = 0;
                    int a = 42 / d;
                System.out.println(“This code is safe”);
            }
    }
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws this exception.*

This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.

Any exception that is not caught by our program will ultimately be processed by the default handler.

The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the output generated when this example is executed.

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Example:

```
class Exc1
    {
            static void subroutine()
            {
                    int d = 0;
                    int a = 10 / d;
            }
            public static void main(String args[])
            {
                    Exc1.subroutine();
            }
    }
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

Using try and catch

```
class Exc2
{
    public static void main(String  args[])
    {       int d, a;
            try
            {           // monitor a block of code.
                d = 0;
                a = 42 / d;
                System.out.println("This will not be printed.");
            }
            catch (ArithmeticException   e)
            {           // catch divide-by-zero error
                System.out.println("Division by zero.");
            }
            System.out.println("After catch statement.");
    }
```

```
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Multiple catch Clauses

```
class MultiCatch
{       public static void main(String  args[])
        {       try
                {               int a = args.length;
                                System.out.println("a = " + a);
                                int b = 42 / a;
                                int c[] = { 1 };
                                c[42] = 99;
                  }
                catch(ArithmeticException   e)
                  {             System.out.println("Divide by 0: " +e);
                  }
                catch(ArrayIndexOutOfBoundsException   e)
                   {            System.out.println("Array index oob: " +  e);
                  }
                  System.out.println("After try/catch blocks.");
```

```
    }
}
```

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
```

```
C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException
After try/catch blocks.
```

When we use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclass.

This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error.

For example, consider the following program:

```
class SuperSubCatch
{
  public static void main(String  args[])
   {
              try
            { int a = 0;
                        int b = 42 / a;
              }

               catch(Exception e)
               {            System.out.println("Generic Exception catch.");
               }
               /* This catch is never reached because ArithmeticException is a subclass of Exception. */
               catch(ArithmeticException e)
               {            // ERROR - unreachable
                           System.out.println("This is never reached.");
               }
   }
```

```
}
```

If we try to compile this program, we will receive an error message stating that the second catch statement is unreachable because the exception has already been caught.

Since ArithmeticException is a subclass of Exception, the first catch statement will handle all Exception-based errors, including ArithmeticException.

This means that the second catch statement will never execute.

To fix the problem, reverse the order of the catch statements.

Nested try Statements: The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

```
class NestTry
{     public static void main(String args[])
    {       try
            {         int a = args.length;
                      int b = 42 / a;
                     System.out.println("a = " + a);
                      try
                     {       if(a==1)
                             a = a/(a-a); // division by zero
                             if(a==2)
                             {
                                      int c[] = { 1 };
                                     c[42] = 99; // generate an out-of-bounds exception
                            }
                     }
                     catch(ArrayIndexOutOfBoundsException  e)
```

```
                {       System.out.println("Array index out-of-bounds: " + e);
                }
        }
        catch(ArithmeticException  e)
       {       System.out.println("Divide by 0: " + e);
       }
  }
```

Whenever a try block does not have a catch block for a particular exception, then the catch blocks of parent try block are inspected for that exception, and if a match is found then that catch block is executed.C:\>java NestTry

Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One

a = 1

Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two

a = 2

Array index out-of-bounds: java.lang.ArrayIndexOutOfBoundsException

/* Try statements can be implicitly nested via calls to methods. */

```
class MethNestTry
{       static void nesttry(int  a)
      {       try
              {               if(a==1)
                            a = a/(a-a); // division by zero
                              if(a==2)
                              {               int c[] = { 1 };
                                              c[42] = 99; // generate an out-of-bounds exception
                              }
              }
              catch(ArrayIndexOutOfBoundsException e)
              {               System.out.println("Array index out-of-bounds: " + e);              }
      }
      public static void main(String  args[])
      {       try {           int a = args.length;
                             int b = 42 / a;
                            System.out.println("a = " + a);
                            nesttry(a);
                 }
                 catch(ArithmeticException e)
                {               System.out.println("Divide by 0: " +e);
                }
    }
}
```

throw

It is possible to throw an exception explicitly, using the **throw** statement.

Till now, we have seen catching the exceptions that are thrown by the Java run-time system. It is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

throw ThrowableInstance;

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.

There are two ways you can obtain a Throwable object:

– using a parameter in a catch clause, or
– creating one with the new operator.

The general form of throw is shown here:

throw *ThrowableInstance;*

// Demonstrate throw.

```
class Excep1
{
	static void validate(int age)
	{
		if(age<18)
		{
			throw new ArithmeticException("Not valid to vote");
		}
		else
		{
			System.out.println("Welcome to vote");
		}

	}
	public static void main(String args[])
	{
		try
		{
		validate(21);
		}
		catch(ArithmeticException e)
		{
			System.out.println(e);
		}
		System.out.println("Testing Complete");
	}
}
```

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.

We can do this by including a throws clause in the method's declaration.

A throws clause lists the types of exceptions that a method might throw.

This is the general form of a method declaration that includes a throws clause:

type method-name(parameter-list) throws exception-list

```
{
        // body of method
}
```

Here, *exception-list is a comma-separated list of the exceptions that a method can throw.*

```
// Example
class ThrowsDemo
{
    static void throwOne() throws  IllegalAccessException
    {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
    }
    public static void main(String  args[])
    {           try
                {               throwOne();
                }
                 catch (IllegalAccessException e)
                 {
                                System.out.println("Caught " + e);
                }
    }
}
```

Here is the output generated by running this example program:

inside throwOne

caught java.lang.IllegalAccessException: demo

```
package exh;
class Excep2
{
    static void validate(int age) throws ArithmeticException
    {
           if(age<18)
           {
                  throw new ArithmeticException("Not valid to vote");
           }
           else
           {
                  System.out.println("Welcome to vote");
           }

    }
    public static void main(String args[])
    {
           try
           {
           validate(16);
           }
           catch(ArithmeticException e)
           {
```

```
                System.out.println(e);
            }
            System.out.println("Testing Complete");
        }
    }
```

finally

finally executes a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block.

The **finally** block will execute whether or not an exception is thrown.

If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.

Used to perform certain house-keeping operations such as closing files and releasing system resources.

Example:

```
class prg
{
    public static void main(String args[])
   {
          try
        {
              int a = 8/0;
          }

          catch(Exception e)
         {
              System.out.println("error"+e);
         }
        finally
        {
              System.out.println("finally");
        }
      System.out.println(“successful");
   }
```

```
}
```

```
// Demonstrate finally.
class FinallyDemo
{       static void procA()
        {       try
                  {             System.out.println("inside procA");
                                throw new RuntimeException("demo");
                }
                 finally
                {               System.out.println("procA's finally");
                }
       }
       // Return from within a try block.
      static void procB()
      {       try
                {                               System.out.println("inside procB");
                                                 return;
                }
                finally
                {               System.out.println("procB's finally");
                }
```

```
}
```

```
    // Execute a try block normally.
    static void procC()
    {           try
                {           System.out.println("inside procC");
                }
                finally
                {           System.out.println("procC's finally");
                }
    }
    public static void main(String args[])
    {           try
                {
                            procA();
                }
                catch (Exception e)
                {           System.out.println("Exception caught");
                }
                procB();
                procC();
    }
}
```

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Creating our Own Exception Subclasses

```
// This program creates a custom exception type.
class MyException extends Exception
{
    private int detail;
    MyException(int a)
    {
            detail = a;
    }
    public String toString()
    {
            return "MyException[" + detail + "]";
    }
}
```

```
class ExceptionDemo
{       static void compute(int a) throws  MyException
        {           System.out.println("Called compute(" + a +  ")");
                    if(a > 10)
                    throw new MyException(a);
                    System.out.println("Normal exit");
         }
        public static void main(String  args[])
        {           try
                {       compute(1);
                            compute(20);
                    }
                     catch (MyException e)
                    {                   System.out.println("Caught " + e);
                    }
      }
}
```

Output:

Called compute(1)

Normal exit

Called compute(20)

Caught MyException[20]

MULTITHREADED PROGRAMMING

Introduction

- A multithreaded program contains two or more parts that can run concurrently.
- Each part of a multithreaded program is called a thread.
- Each thread defines a separate path of execution.
- Java provides built in support for multithreaded programming.
- Multithreading is a specialized form of multitasking.

- The two types of multitasking are
    1. Process-based
    2. Thread based

Process-based multitasking is the feature that allows our computer to run two or more programs concurrently.

Here program is the smallest unit of code that can be dispatched by the scheduler.

For ex: It allows us to run the Java compiler at the same time that we are using a text editor.

Thread based multitasking:

Thread is the smallest unit of dispatchable code.

- A single program can perform two or more tasks simultaneously.
- **Ex:** a text editor can format text at the same time that is printing, as long as these two actions are being performed by two separate threads.
- Thus process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

Difference between process-based and thread-based multitasking processes:

- Multitasking threads require **less overhead** than multitasking processes.

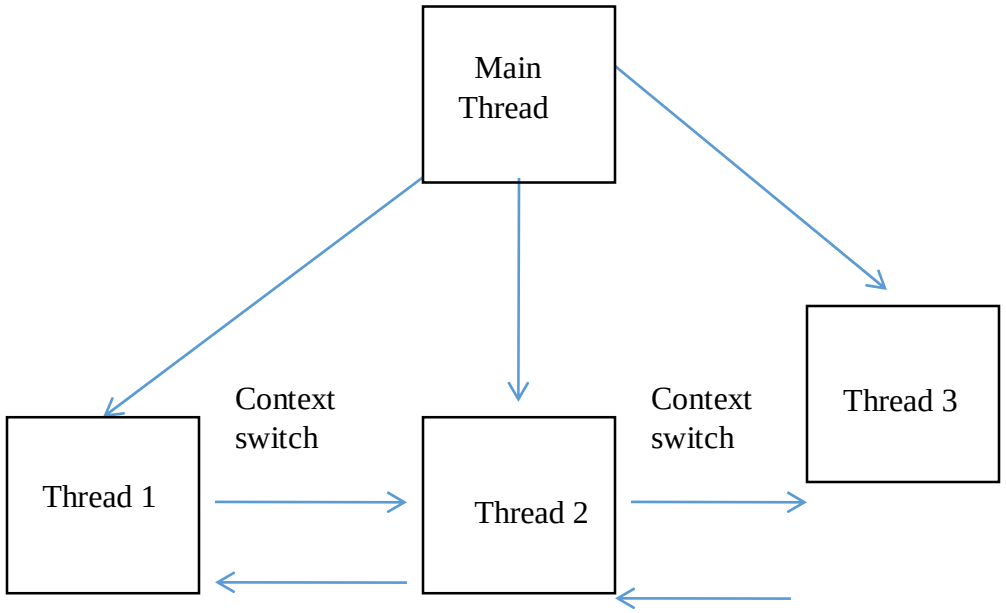
- Processes are **heavyweight tasks** that require their own **separate address spaces**. But the threads are **lightweight processes** and they **share the same address space**

- **Context switching** from one process to other process is **costly**. But, context switching from one thread to the next is **low cost**.

- **Interprocess communication is expensive** and limited in process-based multitasking. But, **interthread communication is inexpensive.**

- Process-based multitasking is **not under the control of Java**. But, the multithreaded multitasking is **under the control of Java**.

Multithreaded program



- Multithreading enables us to write very efficient programs that make the maximum use of the CPU, because the idle time can be kept to a minimum.

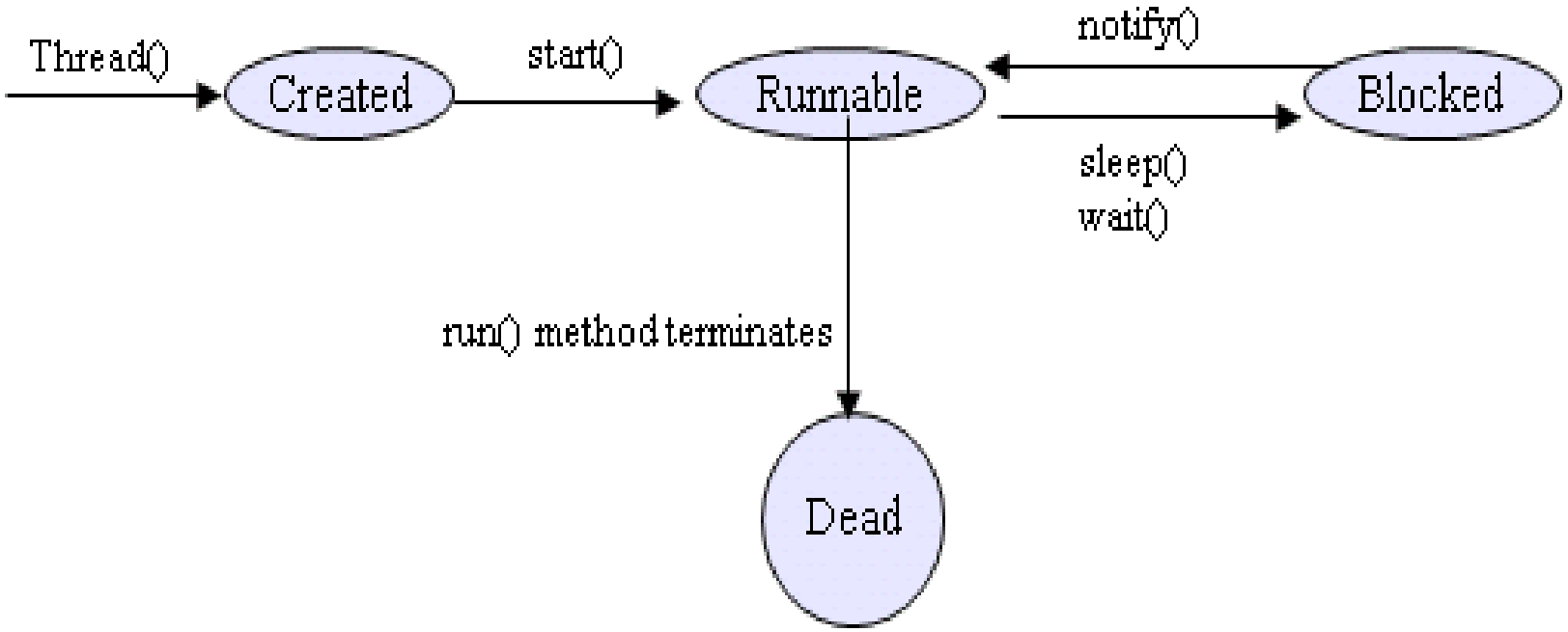
- In a single threaded environment, our program has to wait for each of these tasks to finish before it can proceed to the next one – even though the CPU is sitting idle most of the time.

- Multithreading allows us to gain access to the idle time and put it to good use.

Threads exist in several states

- Threads can be in one of four states
  - Created, Running, Blocked, and Dead

- A thread's state changes based on:
  - Control methods such as start, sleep, wait, notify
  - Termination of the run method


Thread()
Created
start()
Runnable
notify()
Blocked
sleep()
wait()
run() method terminates
Dead

• **New state:** After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.
• **Runnable (Ready-to-run) state:** A thread starts its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
• **Running state:** A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.
• **Dead state:** A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
• **Blocked:** A thread can enter in this state because of waiting the resources that are hold by another thread.

Thread priorities:

A thread's priority is used to decide when to switch from one running thread to the next.

Rules to decide context switching are:

1. **A thread can voluntarily relinquish control:**

   It is done by explicitly yielding, sleeping, or blocking on pending I/O. Here all the threads are examined and the highest-priority thread that is ready to run is given the CPU.

2. **A thread can be preempted by a higher –priority thread:**

   In this case, a lower priority thread that does not yield the processor is simply preempted no matter what it is doing by a higher priority thread. This is called preemptive multitasking.

The Thread class and the Runnable Interface

Multithreading in Java is facilitated using,

1. **Thread class** and its methods
2. The interface **Runnable**

- To create a new thread our program will have to extend either the **Thread class** or implement the **Runnable interface.**

Methods defined by Thread class

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

The Main Thread

When a Java program is started the **Main thread runs immediately**. ie, it starts execution.

Importance of Main Thread:

1. It is the thread from which other **"child"** threads will be spawned.
2. Often, it must be the **last thread** to finish execution because it performs various shutdown actions.

The main thread can be controlled through a **Thread** object. It is done by obtaining a reference to it by calling the method **currentThread()**

The currentThread() is a public static member of thread .

General form :

static Thread currentThread()

It returns a reference to the thread in which it is called.

By using a reference to the main thread, we can control it like any other thread.

Example

```
// Controlling the main Thread.
class  CurrentThreadDemo
{
      public   static   void   main(String    args[])
   {              Thread t = Thread.currentThread();
                  System.out.println("Current thread: " + t);
                  t.setName("My Thread");
                  System.out.println("After name change: " + t);
                  try  {            for(int n = 5;n>0;n--)
                                    {                 System.out.println(n);
                                                      Thread.sleep(1000);
                                    }
                  } catch(InterruptedException    e)
                  {                 System.out.println("Main thread interrupted");
                  }
    }
}
```

```
}
```

Current thread: Thread [main, 5, main]

After name change: Thread [My Thread, 5, main]

5

4

3

2

1

- The first line in the output displays the name of the thread, its priority and the name of its group.
- The second line displays the output after changing the thread name.
- Thread group is a data structure that controls the state of a collection of threads as a whole.
- The sleep() method causes the thread from which it is called to suspend execution for the specified period of milliseconds.

General form:

1. static void sleep(long milliseconds) throws InterruptedException

Note: Sleep and join has to be written in try block, because they may through the exception

We can set the name of a thread by using setName().

General form:

final void setName(String threadName)

Here threadName specifies the name of the thread.

We can obtain the name of a thread by calling getName().

General form:

final String getName()

Creating a Thread

A thread can be created by **instantiating an object** of type Thread.

The 2 ways defined by Java for the creation of thread are:
Implementing the single thread:

1. By **Extending the Thread class.**

2. By **Implementing the Runnable interface.**

1. Extending Thread

- A class that **extends Thread** is another way of **creating a thread** and **creating an instance** of that class.

- The **extending class** must **override the run()** method, which is the entry point for the new thread.

- It must also call the **start()** method to **begin the execution** of the **new thread.**

A new thread is created by the following statement:

Thread t = new Thread();

After the creation of a new thread it will not start running until the start() method declared inside the Thread is called.

start() executes a call to run().

General form:

void start();

You can inherit the Thread class as another way to create a thread in your program. When you declare an instance of your class, you'll also have access to members of the Thread class. Whenever your class inherits the Thread class, you must override the run() method, which is an entry into the new thread. The following example shows how to inherit the Thread class and how to override the run() method.

```
package exthreads;
import java.util.*;
class Single extends Thread
{
	public void run()
	{
		System.out.println("Child thread started");
		for(int p =0;p<5;p++)
		{

		System.out.println(p);
		}
		System.out.println("Child thread terminated");
	}


}

public class St
{
	public static void main(String args[])
	{

		System.out.println("Main thread started");

		Single t=new Single();
		t.start();
```

```
        }
}
```

2. Implementing Runnable:

We can construct a thread on any object that implements **Runnable.**

To implement Runnable, a class need to implement a single method called **run().**

General form:

public void run()

- Inside run(), we will define the code that constitutes the new thread.
- After creating a class that implements Runnable ,we can instantiate an object of type Thread from within that class.

Steps to create a new Thread **using** Runnable **:**
1. Create a Runnable implementer and implement run() method.
2. Instantiate Thread class and pass the implementer to the Thread, Thread has a constructor which accepts Runnable instance.
3. Invoke start() of Thread instance, start internally calls run() of the implementer. Invoking start(), creates a

new Thread which executes the code written in run().

Constructors defined by Thread:

Thread()

Thread(Runnable threadOb)

Thread(String threadName)

Thread(Runnable threadOb, String threadName)

Here,

- threadOb is an instance of a class that implements Runnable interface. It defines where the execution of the thread will begin.
- threadName is the name of the new thread.

```
class Singl implements    Runnable
{
   public void run()
       {
              System.out.println("Child thread started");
              for(int p =0;p<5;p++)
              {

              System.out.println(p);
              }
              System.out.println("Child thread terminated");
       }


}

public class Strunnable
{
       Public static void main(String args[])
       {

       System.out.println("Main thread started");

       Singl s=new Singl();
       Thread t=new Thread(s);
       t.start();

 }
```

Choosing an Approach

- The **Thread** class defines several methods that can be overridden by a derived class. Of these methods, the only one that *must* be overridden is **run( )**.

- Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**.

Creating multiple threads

```
class NewThread implements Runnable
{
    String name; // name of thread
    Thread t;
     NewThread(String threadname)
     {      name = threadname;
            t = new Thread(this, name);
           System.out.println("New thread: " +t);
           t.start(); // Start the thread
     }
      public void run()
      {     try
           {     for(int i = 5; i > 0; i--)
                 {       System.out.println(name + ": " + i);
                         Thread.sleep(1000);
                  }
            }
           catch (InterruptedException e)
            {              System.out.println(name + "Interrupted");
              }
             System.out.println(name + " exiting.");
```

```
        }
    }
```

```
class  MultiThreadDemo
{
    public   static   void   main(String   args[])
  {
                new   NewThread("One"); // start threads
                new   NewThread("Two");
                new   NewThread("Three");
                try
                {             // wait for other threads to end
                                Thread.sleep(10000);
                }
                catch (InterruptedException   e)
                {               System.out.println("Main thread  Interrupted");
                }
                System.out.println("Main thread exiting.");
   }
```

```
}
```

New thread: Thread[One, 5, main]

New thread: Thread[Two, 5, main]

New thread: Thread[Three, 5, main]

One: 5

Two: 5

Three: 5

One: 4

Two: 4

Three: 4

One: 3 **// Here once started, all three child threads share the CPU and the call to sleep(10000)**

Three: 3 **// in main() causes the main thread to sleep for ten seconds and ensures thatit will finish last**.

Two: 3

One: 2

Three: 2

Two: 2

One: 1

Three: 1

Two: 1

One exiting.

Two exiting.

Three exiting.

Main thread exiting.

```
package exthreads;
	class Thh extends Thread
	{
	public void run()
	{
		System.out.println("Child thread started");
		Thread t=currentThread();
		System.out.println("alive status="+ t.isAlive());
		for(int p =0;p<5;p++)
		{
	    try
	    {
	     //Thread.sleep(1000);
	     t.sleep(1000);
	    }
	    catch(Exception e)
	    {

	    }
		System.out.println(p);
		}
		System.out.println("Child thread terminated");
	}


}

public class TTmethods
{
	public static void main(String args[])
	{
```

```
		System.out.println("Main thread started");

		Thh t1=new Thh();
		Thh t2=new Thh();
		System.out.println("Id="+t1.getId());
		System.out.println("Name="+t1.getName());
		t1.setName("Rahul");
		System.out.println("NewName="+t1.getName());
		System.out.println("get the priority="+t1.getPriority());
		t1.setPriority(1);
		System.out.println("get the priority="+t1.getPriority());*/
		t1.start();

		try
		{
			t1.join();
		}
		catch(Exception e)
		{

		}
		System.out.println("alive status="+t1.isAlive());
		t2.start();

	}
}
```

```
Main thread started
Child thread started
alive status=true
0
1
2
```

```
3
4
Child thread terminated
alive status=false
Child thread started
alive status=true
0
1
2
3
4
Child thread terminated
```

```
class MyThread extends Thread
{
     public void run()
     {

     }
}
class prg
{
    public static void main(String args[])
   {
          MyThread   t = new MyThread();
          t.start();
    }
}
```

```
class MyThread extends Thread
{
    MyThread()
    {
          start();
     }
     public void run()
     {

     }
}
class prg
{
    public static void main(String args[])
   {
          MyThread   t = new MyThread();
     }
```

```
}
```

```
class MyThread implements Runnable
{
    public void run()
    {

    }
}
class prg
{
    public static void main(String args[])
   {
        MyThread   t = new MyThread();
        Thread     t2 = new Thread(t)
        t2.start();
    }
}
```

```
class MyThread implements Runnable
{
    MyThread()
    {
    Thread t2 = new Thread(this)
          t2.start();
     }
     public void run()
     {

     }
}
class prg
{    public static void main(String args[])
   {
        MyThread   t = new MyThread();
    }
}
```

Constructors defined by Thread:

Thread()

Thread(String threadName)

Thread(Runnable threadOb)

Thread(Runnable threadOb, String threadName)

Methods : start()

run()

setName()

getName();

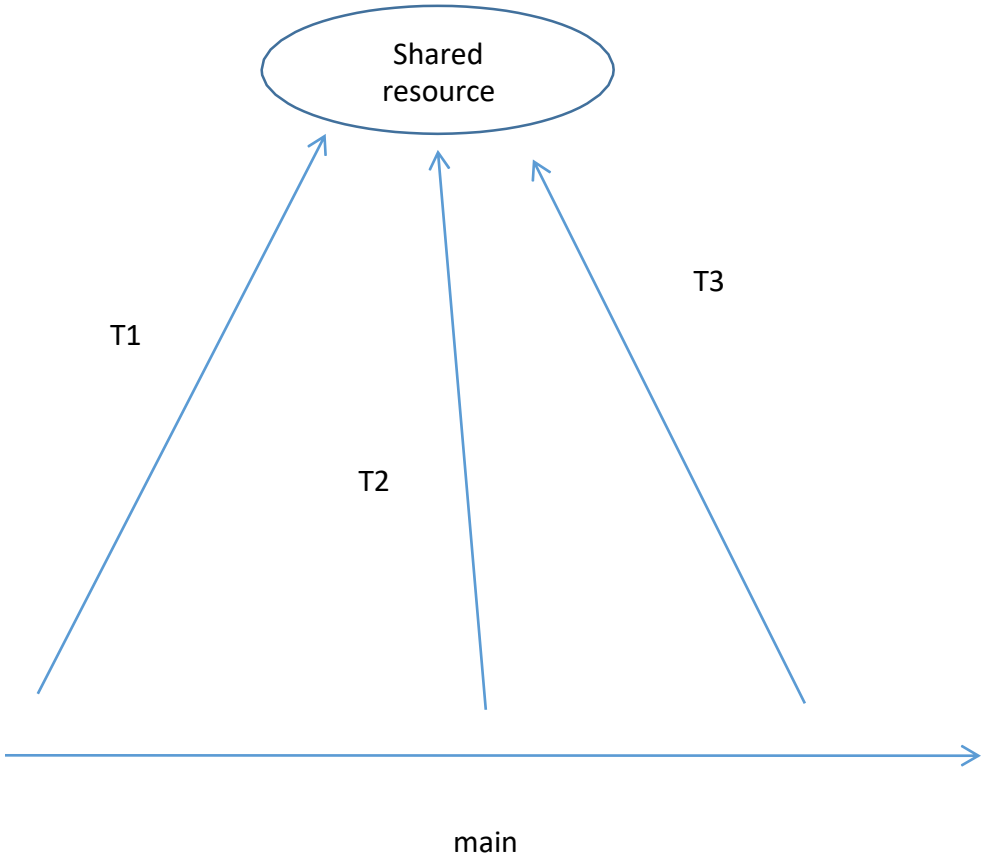
Static methods: currentThread()

sleep()

Synchronization

- When two or more threads need concurrent access to a shared data resource, they need to take care to only access the data one at a time.
- For example, one thread may try to read a record from a file while another is still writing to the same file. Depending on the situation, we may get strange results.
- Java enables us to overcome this problem using a technique known as synchronization.
- Keyword synchronized helps to solve such problem by keeping a watch on such locations.

- For example the method that will read information from a file and the method that will update the same file may be declared as synchronized.


Shared
resource
T1
T2
T3
main

```
synchronized void update()
{

}
```

- When we declare a method synchronized, java creates a “monitor” and hands it over to the thread that calls the method first time.
- As long as the thread holds the monitor, no other thread can enter the synchronized section of the code.
- A monitor is like a key and the thread that holds the key can only open the lock.

It is also possible to mark a block of code as synchronized as follows.

```
synchronized(lock-object)
{
        // code here
}
```

Whenever a thread has completed its work of using synchronized method(or block of code), it will handover the monitor to the next thread that is ready to use the same resources.

```
class  Callme
{
    void call(String msg)
    {
                System.out.print("[" + msg);
                try
                {               Thread.sleep(1000);
                }
                 catch(InterruptedException   e)
                 {
                                System.out.println("Interrupted");
                }
                 System.out.print("]" );
    }
}
```

```
class Caller extends Thread
 {
       String msg;
       Callme  obj;
       public Caller(Callme targ, String s)
      {
                msg = s;
                obj=targ;
     }
      public void run()
     {
                obj.call(msg);
     }
}
```

```
class Synch3
{       public static void main(String args[])
        {         Callme target = new Callme();

                    Caller c1= new Caller(target, " dsce “);
                  c1.start();

                   Caller c2= new Caller(target, “ise“);
                   c2.start();

                   Caller c3= new Caller(target, “engineering“);
                  c3.start();

                   new Caller(target, " college ").start();
     }
}
```

OUTPUT

[dsce [ise [engineering [college]]]]

```
class   Callme
{
   synchronized  void   call(String   msg)
   {
            System.out.print("[" + msg);
            try
            {            Thread.sleep(1000);
            }
             catch(InterruptedException    e)
             {
                        System.out.println("Interrupted");
            }
             System.out.print(“]“ );
   }
}
```

OUTPUT

[dsce] [ise] [engineering] [college]

Marking block of code as synchronized

```
public void run()
{
        synchronized(obj)
        {
                obj.call(msg);
        }
 }
```

• What is a Join Method in Java?

- Join method in Java allows one thread to wait until another thread completes its execution. In simpler words, it means it waits for the other thread to die. It has a ***void*** type and throws ***InterruptedException***.

- Joining threads in Java has three functions namely,

Method	Description
join()	Waits for this thread to die
join(long millis)	Waits at most millis milliseconds for this thread to die
join(long millis, int nanos)	Waits at most millis milliseconds plus nano nanoseconds for this thread to die

Example

```
import java.io.*;
import java.util.*;

public class Threadjoiningmethod extends Thread{
        public void run(){
         for(int i=1;i<=4;i++){
         try{
         Thread.sleep(500);
          }catch(Exception e){System.out.println(e);}
         System.out.println(i);
         }
        }
       public static void main(String args[]){
          Threadjoiningmethod th1=new Threadjoiningmethod ();
          Threadjoiningmethod th2=new Threadjoiningmethod ();
          Threadjoiningmethod th3=new Threadjoiningmethod ();
        th1.start();
        try{
        th1.join();
        }
        catch(Exception e){
           System.out.println(e);
           }

        th2.start();
        th3.start();
                                                                    }
                                                                }
```

```
O          1
           2
  u        3
           4
  t        1
           1
  p        2
           2
  u        3
           3
  t        4
           4
  :
```

Suspending, Resuming and Stopping Threads

The **suspend()** method of thread class puts the thread from running to waiting state.
This method is used if you want to stop the thread execution and start it again when a certain event occurs. This method allows a thread to temporarily cease execution. The suspended thread can be resumed using the resume() method.

Java Thread stop() method
The stop() method of thread class terminates the thread execution. Once a thread is stopped, it cannot be restarted by start() method.

The following methods defined by Thread are used to suspend and resume threads

in the Java 1.1 and other versions before Java 2.

```
final void suspend( )
final void resume( )
```

final void stop()

Once a thread has been stopped it cannot be restarted using **resume()**.

```
public class JavaSuspendExp extends Thread
{
   public void run()
   {
      for(int i=1; i<5; i++)
      {
         try
         {
            // thread to sleep for 500 milliseconds
             sleep(500);
             System.out.println(Thread.currentThread().getName());
         }catch(InterruptedException e){System.out.println(e);}
         System.out.println(i);
      }
```

```
    }
    public static void main(String args[])
    {
        // creating three threads
        JavaSuspendExp t1=new JavaSuspendExp ();
        JavaSuspendExp t2=new JavaSuspendExp ();
        JavaSuspendExp t3=new JavaSuspendExp ();
        // call run() method
        t1.start();
        t2.start();
        // suspend t2 thread
        t2.suspend();
        // call run() method
        t3.start();

    }
}
```

```
Thread-0
1
Thread-2
1
Thread-0
2
Thread-2
2
Thread-0
3
Thread-2
3
Thread-0
4
Thread-2
4
```

Suspend(), resume() and stop() methods in Java 2

- The suspend(), resume() and stop() methods of the Thread class are deprecated in Java 2 since they may result in serious system failures.

- To facilitate the above said operation in Java 2 a thread must be designed so that the r**un( ) method periodically checks to determine whether that thread should suspend, resume, or stop its own execution**. This is accomplished by **establishing flag** variable that indicates the execution state of the thread. As long as this flag is to **"running,"** the run( ) method must continue to let the thread execute. **If this variable is set** to **"suspend,"** the thread must pause. If it is set to **"stop,"** the thread must **terminate**.

```
class NewThread implements Runnable
{
String name;
boolean suspendFlag;// name of thread Thread t;
NewThread(String threadname)
{
  name = threadname;
  Thread t = new Thread(this, name);
  System.out.println("New thread: " + t);
  suspendFlag = false;
```

```
  t.start(); // Start the thread
}
public void run()
{
  try
  {
            for(int i = 15; i > 0; i--)
            {
                        System.out.println(name + ": " + i);
                        Thread.sleep(200);
                        synchronized(this)
                        {
                                    while(suspendFlag)
                                    {
                                                wait();
                                    }
                        }

            }
  }
  catch (InterruptedException e)
  {
            System.out.println(name + " interrupted.");
  }
  System.out.println(name + " exiting.");
}

void mysuspend()
{
  suspendFlag = true;
}
```

```
synchronized void myresume()
{
  suspendFlag = false;
  notify();
}
}
class  SuspendResume
{
public static void main(String args[])
{
  NewThread ob1 = new NewThread("One");
  NewThread ob2 = new NewThread("Two");
  try
  {
        Thread.sleep(1000);
        ob1.mysuspend();
        System.out.println("Suspending thread One");
        Thread.sleep(1000);
        ob1.myresume();
        System.out.println("Resuming thread One");
        ob2.mysuspend();
        System.out.println("Suspending thread Two");
        Thread.sleep(1000);
        ob2.myresume();
        System.out.println("Resuming thread Two");
  }
        catch (InterruptedException e)
        {
              System.out.println("Main thread Interrupted");
        }
```

```
                    //System.out.println("Main thread exiting.");

                }
}
```

```
      One: 15
Two: 15
One: 14
Two: 14
One: 13
Two: 13
One: 12
Two: 12
One: 11
Two: 11
Suspending thread One
Two: 10
Two: 9
Two: 8
Two: 7
Two: 6
Resuming thread One
Suspending thread Two
One: 10
One: 9
One: 8
One: 7
One: 6
Resuming thread Two
Two: 5
One: 5
```

```
Two: 4
One: 4
Two: 3
One: 3
Two: 2
One: 2
Two: 1
One: 1
Two exiting.
One exiting.
```

Inter-thread communication in Java

- **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:
  - wait()
  - notify()
  - notifyAll()

1) wait() method

- Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
- The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

2) notify() method

- Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

3) notifyAll() method

- Wakes up all threads that are waiting on this object's monitor.

Difference between wait and sleep?

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.