

Unit-4

Types of errors

- Compile-time errors
- Run-time errors

Compile-Time Errors

- ✓ Missing semicolons.
- ✓ Missing brackets in classes and methods.
- ✓ Misspelling of identifiers and keywords.
- ✓ Missing double quotes in strings.
- ✓ Use of undeclared variables.
- ✓ Bad references to objects.
- ✓ And so on.

Run-Time Errors

- ✓ Dividing an integer by zero.
- ✓ Accessing an element that is out of the bounds of an array.
- ✓ Attempting to use a negative size of an array.
- ✓ Converting invalid string to a number.
- ✓ Accessing a character that is out of bounds of a string.
- ✓ Using a null reference to access a method or a variable.
- ✓ Missing input file.
- ✓ And so on.

Exception

- An exception is an abnormal condition that arises in a code at run time.
(or) An exception is a run-time error.
- When JRE encounters a run-time error, it creates an exception object and throws it (i.e., informs us about an error).
- If the exception object is not caught and handled properly, the JRE will display an error message as shown below and will terminate the program.

Ex:

```
class Error
{
    public static void main( String args[])
    {
        int a=10,b=5,c=5;
        int x= a/(b-c);
        System.out.println("x= " +x);
        int y=a/(b+c);
        System.out.println("y= " +y);
    }
}
```

Exception in thread "main" java.lang.ArithmeticException: / by zero
at Error.main(Error.java:6)

Benefits of exception handling

- It allows us to fix the error.
- It prevents program from automatically terminating.
- Separates Error-Handling Code from Regular Code.
 - Conventional programming combines error detection, reporting, handling code with the regular code, leads to confusion.

Exception-Handling

- When an exception arises, an object representing that exception is created and *thrown* in the method that caused the error.
- An exception can be caught to handle it or pass it on.
- *Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.*

- Java exception handling is managed by via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Program statements which might generate exceptions are kept inside a **try** block. (Program statements to monitor are contained within a **try** block.)
- If an exception occurs within the **try** block, it is thrown.
- Code within **catch** block catch the exception and handle it.
- System generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword **throw**
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Statements contained in the **finally** block will be executed, regardless of whether or not an exception is raised.

- 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*

}

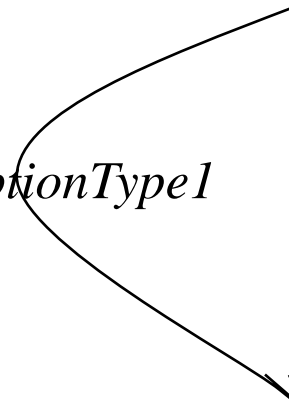
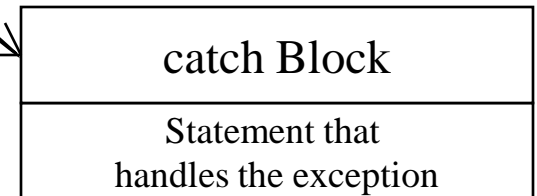
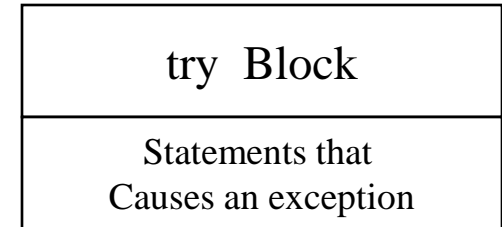
//...

finally

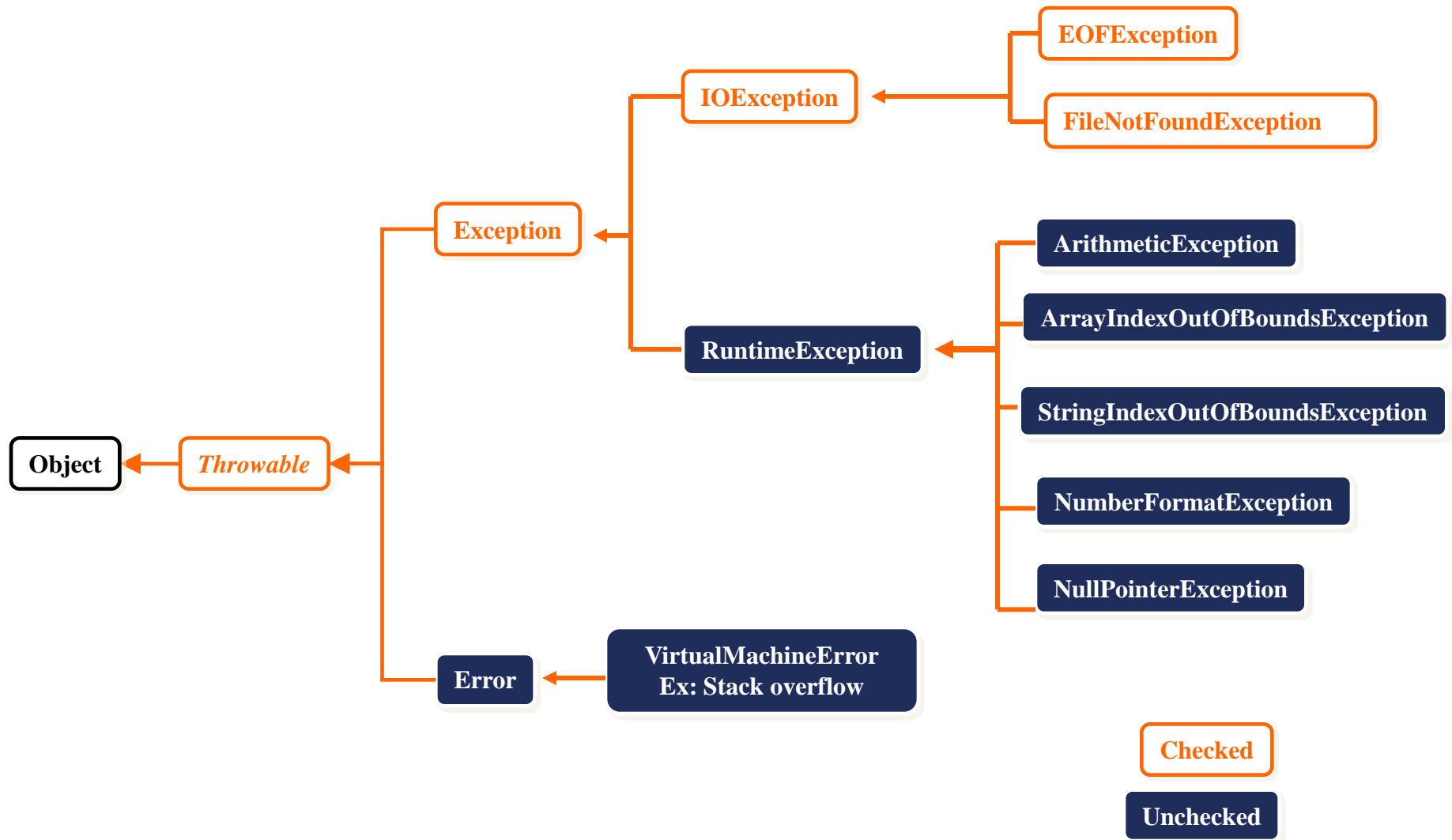
{

// block of code to be executed after try/catch block ends

}



Java Exception class hierarchy



- All exception types are subclasses of the built-in class **Throwable**
- Throwable has two subclasses, they are
 - **Exception**
 - Represents exceptional conditions that an user program might want to catch.
Ex:- IOExceptions, RuntimeExceptions etc.
 - **Error**
 - Represents exceptional conditions that are not expected to be caught by user program.
i.e. Stack overflow

- **IOExceptions:-**
 - The Subclasses of **IOException** represent errors that can occur during the processing of input and output statements.
- **RuntimeExceptions:-**
 - The subclasses of **RuntimeException** represent conditions that arise during the processing of the bytecode that represent a program.

Categories Of Exceptions


- Unchecked exceptions
- Checked exception

Unchecked Exceptions

- The compiler doesn't force you to catch them if they are thrown.
 - No try-catch block required by the compiler
- Examples:
 - NullPointerException,
IndexOutOfBoundsException,
ArithmeticException...

NullPointerException example:

```
class UncheckExce
{
    public static void main(String args[])
    {
        int arr[] =null;
        arr[0] = 1;
        System.out.println(arr[0]);
    }
}
```



NullPointerException

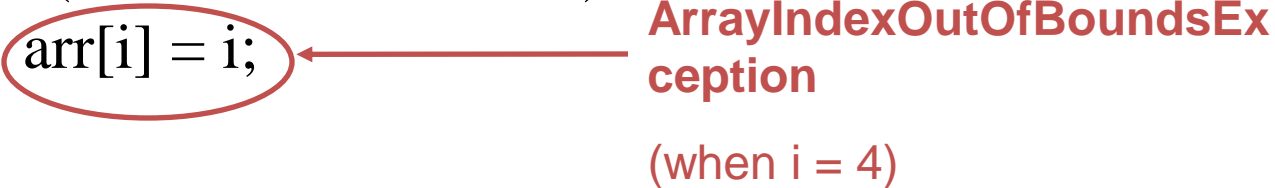
Output:

Exception in thread "main" java.lang.NullPointerException
at UncheckExce.main(UncheckExce.java:6)

IndexOutOfBoundsException example:

```
class UncheckExce
{
    public static void main(String args[])
    {
        int arr[] = new int[4];
        for (int i = 0; i <= 4; i++)
            arr[i] = i;
    }
}
```

ArrayIndexOutOfBoundsException
(when i = 4)



Output:

Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 4
at UncheckExce.main(UncheckExce.java:7)

Checked Exceptions

- The compiler gives an error if we do not catch these exceptions
- The compiler force you to catch them if they are thrown
- Must be handled
 - You must use a try-catch block or throws
- Example:
 - IOException etc.

IOException Example

```
import java.io.*;
public class KeyboardReading5
{
    public static void main(String args[])
    {
        DataInputStream dis = new DataInputStream(System.in);
        System.out.println("Enter name: ");
        String str = dis.readLine();
        System.out.println("Name:"+str);
    }
}
```

Error

KeyboardReading5.java:8: unreported exception
java.io.IOException; must be caught or declared to be thrown

Uncaught Exceptions

- If an exception is not caught by user program, then execution of the program stops and it is caught by the default handler provided by the Java run-time system
- Default handler prints a stack trace from the point at which the exception occurred, and terminates the program.

Ex:

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

Output:

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

```
Exception in thread "main"
```

The Java Stack Trace

- When there is a function call, that function name is placed in the stack, after execution of that function, that function name is removed from the stack.

Ex:

```
class Trace {  
    public static void main(String args[]) {  
        System.out.println("Starting Main method");  
        m1();  
        System.out.println("End Main method");  
    }  
    static void m1() {  
        System.out.println("Method One - m1");  
        m2();  
    }  
    static void m2() {  
        System.out.println("Method Two - m2");  
    }  
}
```

Output:

```
Starting Main Method  
Method One - m1  
Method Two - m2  
End Main method
```

Stack

2		2		2	m2	2		2		2	
1		1	m1	1	m1	1	m1	1		1	
0	main	0	main	0	main	0	main	0	main	0	

Ex: //previous example replace function m2 with the following code

```
static void m2( ) {
```

```
int x = 10;
```

```
int y = 0;
```

```
double z = x / y;
```

```
System.out.println( z );
```

```
System.out.println("Method Two - m2");
```

```
}
```

Output:

Starting Main method

Method One - m1

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Trace.m2(Trace.java:14)

at Trace.m1(Trace.java:9)

at Trace.main(Trace.java:4)

Using try and catch

- The **catch** block should follow immediately the **try** block.
- Once an exception is thrown, program control transfer out of the **try** block into the **catch** block.
- Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.
- Every **try** block should be followed by **at least one catch** statement or **finally** statement; otherwise compilation error will occur.

Example:

```
class Exc1
```

```
{  
    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.");  
    }  
}
```

Output: Division by zero.
 After catch statement.

Displaying Name & Description of an Exception

```
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(e);
        }
        System.out.println("After catch statement.");
    }
}
```

Output:

Division by zero.

java.lang.ArithmeticException: / by zero

After catch statement.

Multiple catch blocks

- If piece of code generates more than one exception, then we use multiple catch blocks.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one **catch** statement executes, the others are bypassed.

Example:

```
class MultiCatch
```

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

Output:

D:\javap>java MultiCatch

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

After try/catch blocks.

Example:

```
class MultiCatch
```

```
{  
    public static void main(String args[])  
    {  
        try  
        {  
            int b = 42 / 1;  
            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.");  
    }  
}
```

Output:

D:\javap>java MultiCatch

Array index oob:

java.lang.ArrayIndexOutOfBoundsException:
42

After try/catch blocks.

Caution

- Remember that, exception subclass must come before any of their superclasses
- Because, a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. So, the subclass would never be reached if it comes after its superclass
- For example, **ArithmeticException** is a subclass of **Exception**
- Moreover, unreachable code in Java generates error.

```

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.");
        }
    }
}

```

Output: SuperSubCatch.java:22: exception java.lang.ArithmeticException has already been caught catch(ArithmeticException e)

^

1 error

Nested try Statements

- A **try** statement can be inside the block of another try.
- Every **try** block should be followed by **at least one catch** statement or **finally** statement; otherwise compilation error will occur.

Example:

```
class NestTry
{
    public static void main(String args[])
    {
        try
        {
            int a = args.length;
            int b = 42 / a;
            System.out.println("a = " + a);
```

```
try
{ // nested try block
  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);
}
}
```

Output: **D:\javap>java NestTry**
Divide by 0: java.lang.ArithmeticException: / by zero
D:\javap>java NestTry Testarg1
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
D:\javap>java NestTry Testarg1 Testarg2
a = 2
Array index out-of-bounds: java.lang.ArrayIndexOutOfBoundsException:
42

throw

- So far, you have only been catching exceptions that are thrown by the Java run-time system.
- It is possible for your program to throw an exception explicitly. General form

throw ThrowableInstance

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

Example1:

```
class NestTry
{
    public static void main(String args[])
    {
        try
        {
            int a = args.length;
            int b;
            if(a==0) throw new ArithmeticException("Divide by Zero");
            else
            b = 42 / a;
            System.out.println(b);
        }
    }
}
```

```
catch(ArithmeticException e)
{
    System.out.println("Divide by 0: " + e);
}
}
```

Output:

C:\Documents and Settings\Admin\Desktop>java NestTry
Divide by 0: java.lang.ArithmeticException: Divide by Zero

C:\Documents and Settings\Admin\Desktop>java NestTry sree
42

- **new** is used to construct an instance of **ArithmeticException**.
- All of Java's built-in run-time exceptions have at least two constructors:
 - One with no parameter
 - Another one that takes a string parameter.
- The string parameter is used to describe exception.

throws

- An exception may occur inside the method
- Method does not handle the exception
- Calling method will handle the exception
type method-name(parameter-list) throws exception-list
{
 // body of method
}
- It is not applicable for **Error** or **RuntimeException**, or any of their subclasses

Example:

```
import java.io.*;
class MyException{
    public static void main(String args[]){
        try{
            checkEx();
        } catch( IOException ioe){
            System.out.println(" File Not Found ");
        }
    }
    public static void checkEx() throws IOException{

        FileInputStream fis = new FileInputStream(" myfile.txt ");
        //continue processing here.
    }
}
```

finally

- **finally** contains block of code that will be executed after **try/catch** block has completed.
- **finally** block will be executed, whether or not an exception is raised.
- The **finally** block is optional.
- Each **try** statement requires at least one **catch** or a **finally** block.

Syntax:

```
try
{
    .....
    .....
}
catch ( ..... )
{
    ....
    ....
}
catch ( ..... )
{
    ....
    ....
}
finally
{
    ...
    ...
}
```

Case 1

Let's see the java finally example where **exception doesn't occur**.

```
class TestFinallyBlock{  
    public static void main(String args[]){  
        try{  
            int data=25/5;  
            System.out.println(data);  
        }  
        catch(NullPointerException e){System.out.println(e);}  
        finally{System.out.println("finally block is always executed");}  
        System.out.println("rest of the code...");  
    }  
}
```

Test it Now

Output:5

```
finally block is always executed  
rest of the code...
```

Case 2

Let's see the java finally example where **exception occurs and not handled**.

```
class TestFinallyBlock1{  
    public static void main(String args[]){  
        try{  
            int data=25/0;  
            System.out.println(data);  
        }  
        catch(NullPointerException e){System.out.println(e);}  
        finally{System.out.println("finally block is always executed");}  
        System.out.println("rest of the code...");  
    }  
}
```

Test it Now

Output:finally block is always executed

Exception in thread main java.lang.ArithmeticException:/ by zero

Case 3

Let's see the java finally example where **exception occurs and handled**.

```
public class TestFinallyBlock2{  
    public static void main(String args[]){  
        try{  
            int data=25/0;  
            System.out.println(data);  
        }  
        catch(ArithmeticException e){System.out.println(e);}  
        finally{System.out.println("finally block is always executed");}  
        System.out.println("rest of the code...");  
    }  
}
```

Test it Now

```
Output:Exception in thread main java.lang.ArithmeticException:/ by zero  
        finally block is always executed  
        rest of the code...
```

Difference between final, finally and finalize

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to Release the resources held by the object just before the object is deleted by the garbage collector.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Java finalize example

```
class FinalizeExample{  
    public void finalize(){System.out.println("finalize called");}  
    public static void main(String[] args){  
        FinalizeExample f1=new FinalizeExample();  
        FinalizeExample f2=new FinalizeExample();  
        f1=null;  
        f2=null;  
        System.gc();  
    }  
}
```

Output : finalize called

Java's Built-in Exceptions

- Java defines several exception classes in **java.lang package** and **java.io package**.

Unchecked RuntimeException subclasses

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
ClassCastException	Invalid cast.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.

Checked exceptions defined in java.io

Exception	Meaning
FileNotFoundException	File is not found
EOFException	End of File Exception. (occurs when you try to read beyond EOF)
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.

- **Note:** `java.lang` is implicitly imported into all Java programs, most exceptions derived from **`RuntimeException`** are automatically available.

Creating Your Own Exception Subclasses

- Java's built-in exceptions handle most common errors.
- It is also possible to create your own exception types to handle situations specific to your applications.

- User-defined exceptions are created by extending **Exception** class.
- The **Exception** class does not define any methods of its own.
- It inherits all the methods provided by **Throwable**.
- Thus, all exceptions, including those that you create, contain the methods defined by **Throwable**.

The Methods Defined by Throwable

Method	Description
<code>void printStackTrace()</code>	Displays the stack trace.
<code>String toString()</code>	Returns a String object containing exception name and description of the exception. This method is called by println() when outputting a Throwable object.
<code>String getMessage()</code>	Returns a description of the exception.

Example:

```
class MyException extends Exception
{
    private String desc;
    MyException(String a)
    {
        desc = a;
    }
    public String toString()
    {
        return "MyException:" + desc;
    }
    public String getMessage() { return desc;}
}
class ExceptionDemo
{
```

```
public static void main(String args[])
{
    int a=0,d;
    try
    {
        if (a==0) throw new MyException("/ by zero");
        else
            d=42/a;
        System.out.println(d);
    }
    catch(MyException e)
    {
        System.out.println("Caught " + e);
        System.out.println(e.getMessage());
    }
}
}
```

Output: Caught MyException: / by zero
/ by zero

Multithreaded Programming

Thread is a part of the program

Multitasking: Executing two or more programs concurrently

Multithreading: Executing multiple parts of a single program concurrently

Types of Multitasking

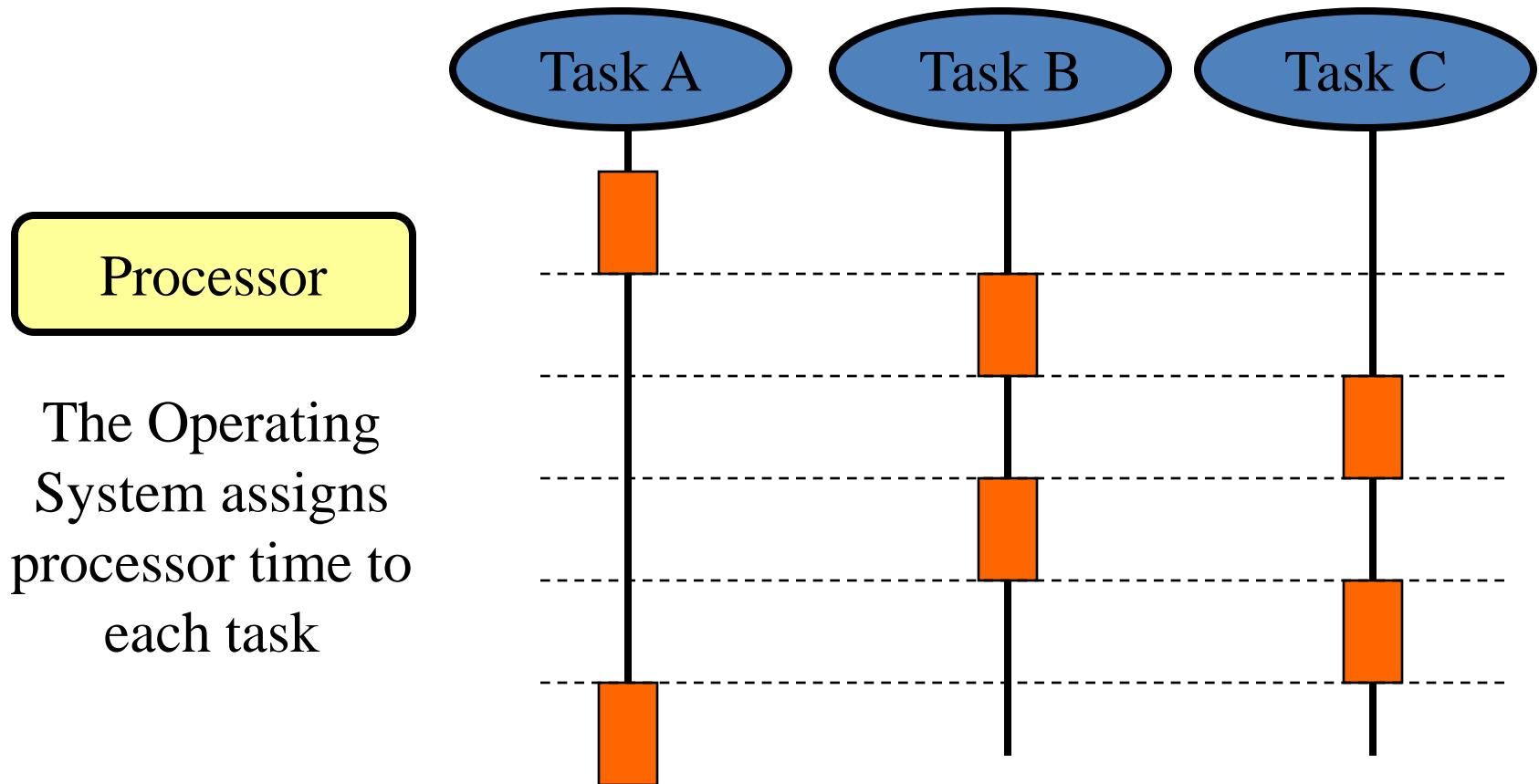
- Process Based

- Thread Based

Process Based Multitasking

- The *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently.
 - For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor.
- A *process* is a program in execution.
- A program becomes process when an executable file is loaded into memory.
- Each process contains its own address space.

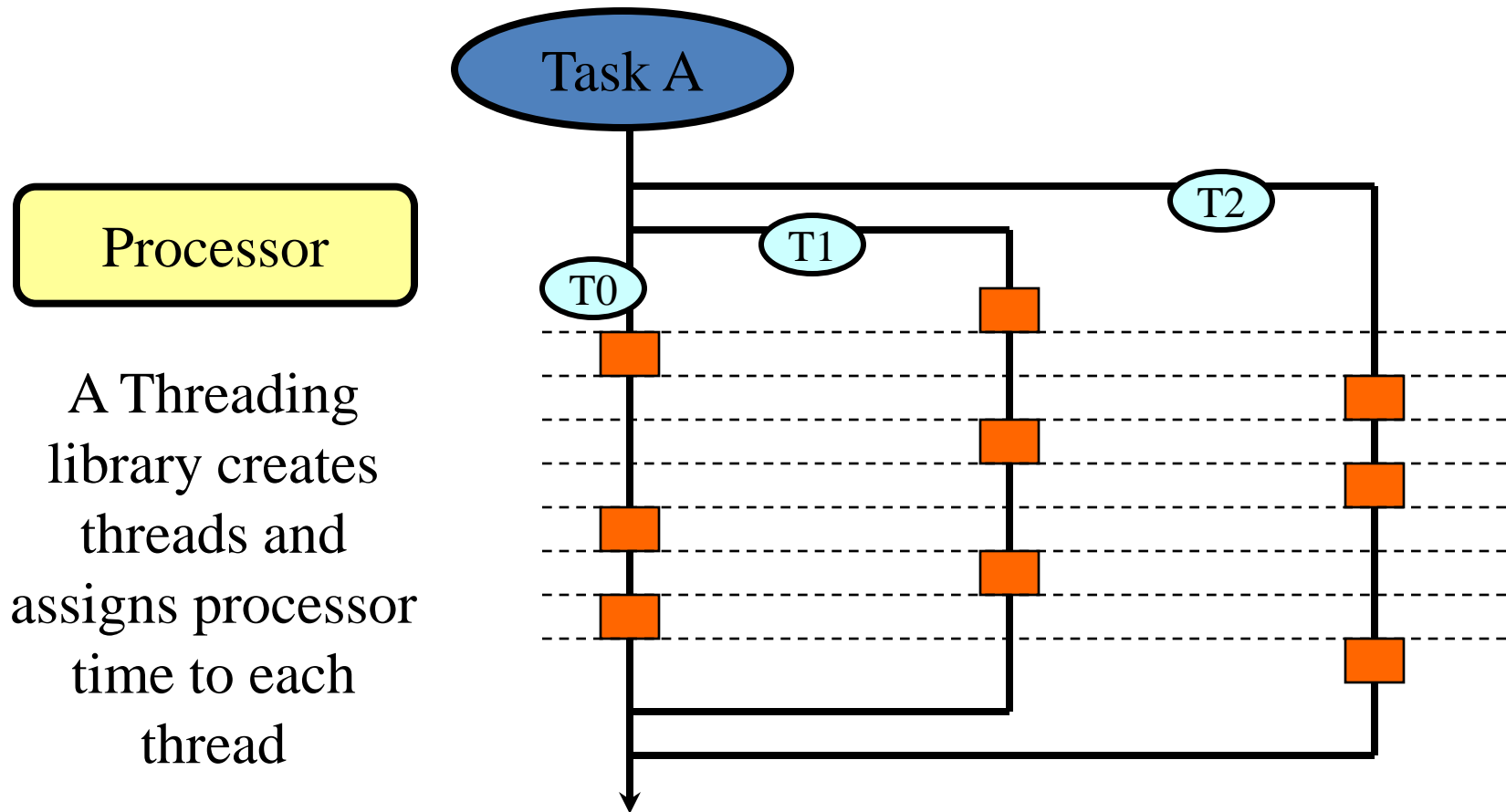
Process Based Multi Tasking



Thread Based Multitasking

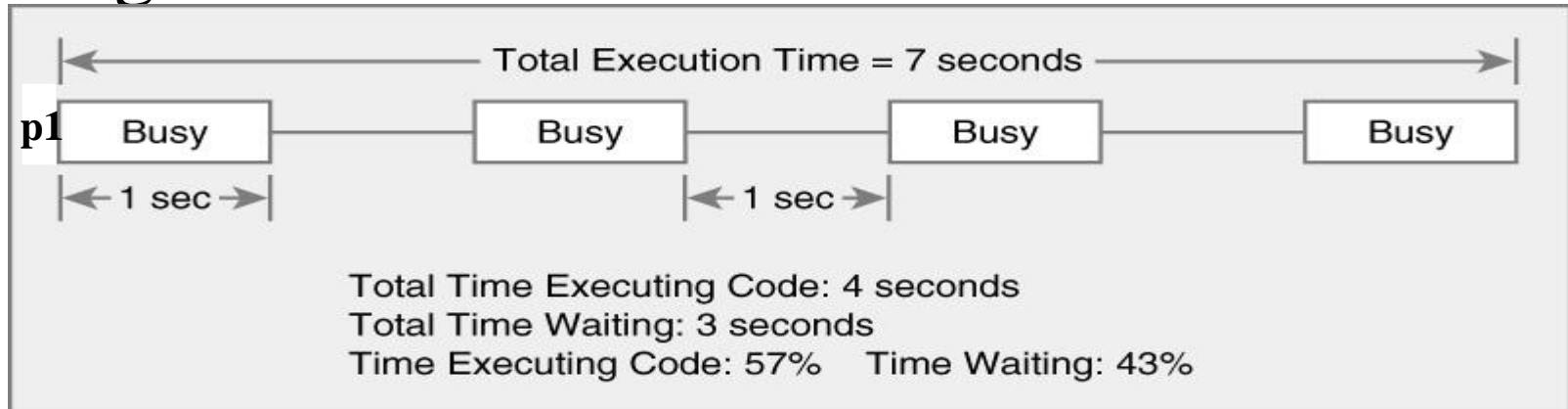
- The *thread-based* multitasking is the feature that allows your computer to run two or more parts of a program concurrently.
 - For example, a text editor can format text at the same time that it is printing.
- **Thread** is a part of the program that is executed independently of other parts of a program.

Thread Based Multitasking

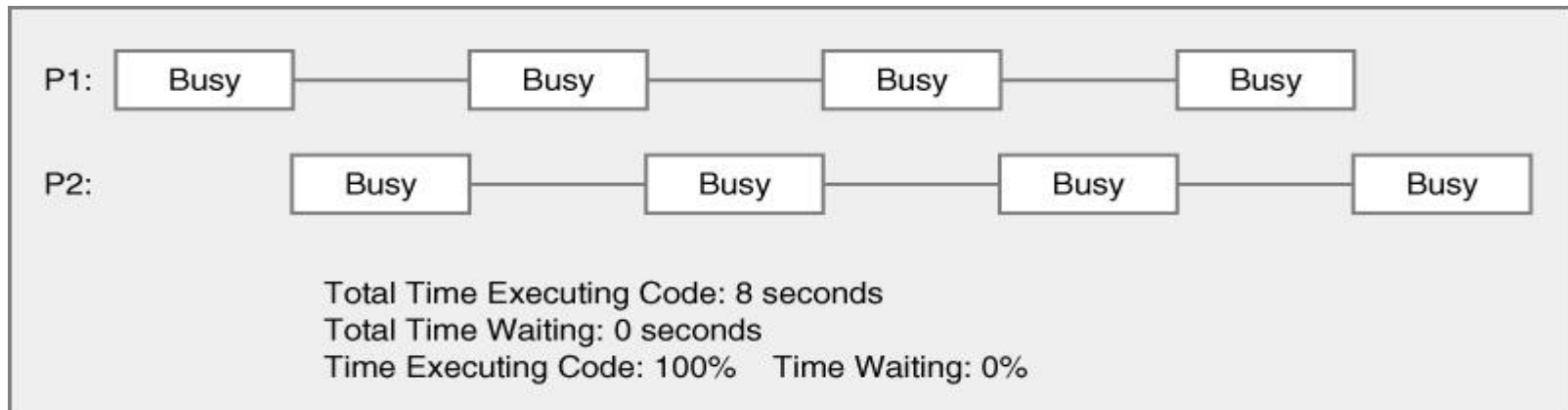


Multitasking Improves Performance

■ Single task



■ Two tasks



Context switch

- *Context switch* moves CPU from one process to another process.
- Switching the CPU to another process requires a state save of the current process and a state restore of a different process.
- While switching, the state of the process will be stored in PCB(Process Control Block).

Difference b/w Multithreading and Multitasking

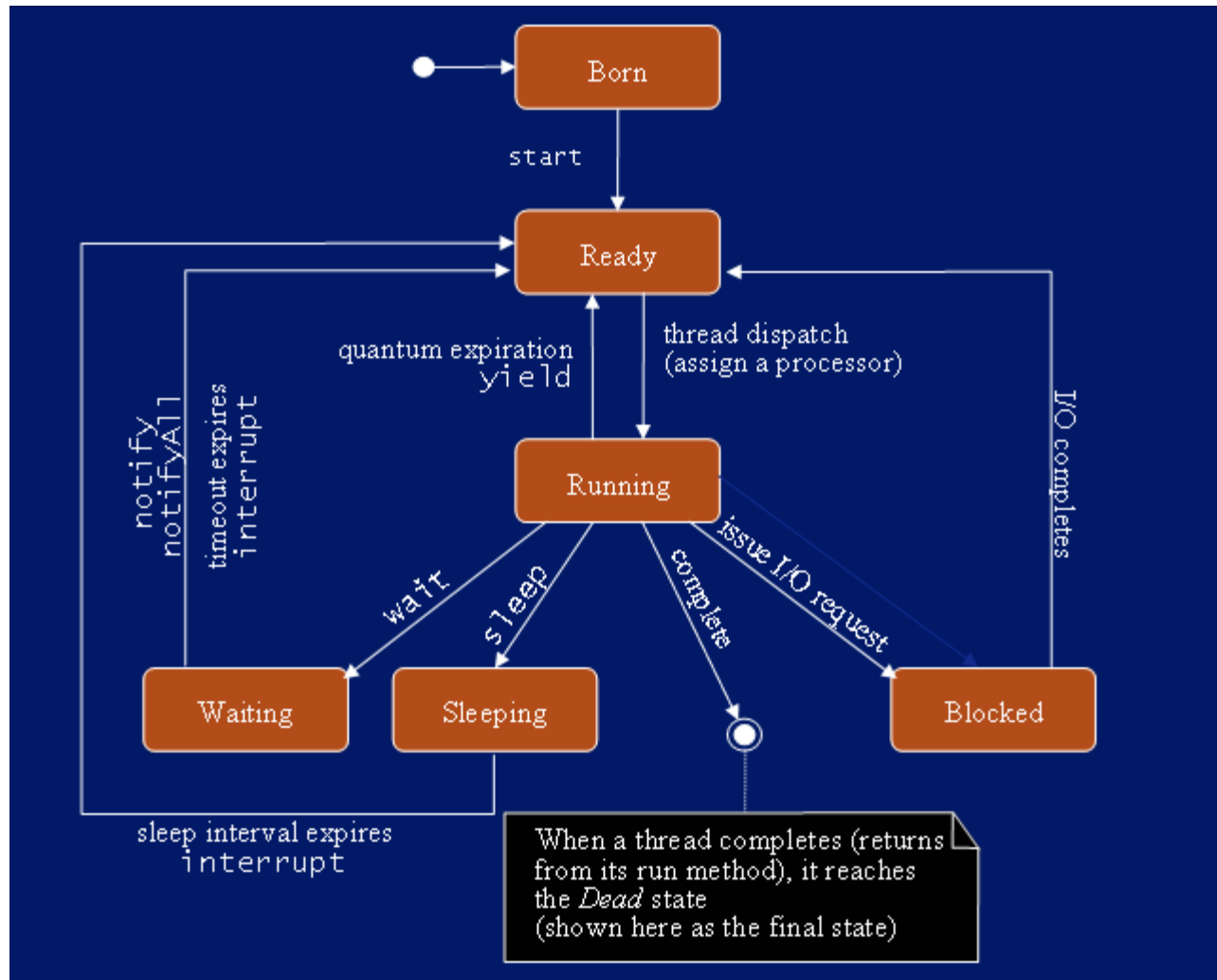
Multithreading

- It supports execution of multiple parts of a single program simultaneously
- All threads share common address space
- Context switching is low cost
- Interthread communication is inexpensive
- Thread is light weight task
- It is under the control of java

Multitasking

- It supports execution of multiple programs simultaneously.
- Each process has its own address space
- Context switching is high cost
- Interprocess communication is expensive
process is heavy weight task
- It is not under the control of java

Thread Life Cycle(States)



Thread States

- When a thread is first created, it does not exist as an independently executing set of instructions. Instead, it is a template/structure from which an executing thread will be created. This state is referred to as *born* state.
- When the thread start method is called, the thread enters into the *ready* state.
- When the system assigns a processor to the thread, the thread enters into the *running* state.
- A thread enters into the *dead* state when it finishes its execution (when its run method completes or terminates).

- The running thread will enter into the *block* state when it issues an input or output request.
- The running thread will enter into the *waiting* state when wait method is called.
 - The thread in the waiting state will become ready when notify method is called.
 - Every thread in the waiting state will become ready when notifyAll method is called.
- The running thread will enter into the *sleeping* state when sleep method is called.

Thread priorities

- Every thread has a *priority*(*integer value*) which can be increased or decreased by calling the *setPriority()* method.
- Thread priority is used to decide when to switch from one running thread to the next.
 - *Thread.MIN_PRIORITY* is the minimum priority (defined as 1)
 - *Thread.MAX_PRIORITY* is the maximum priority (defined as 10)
 - When a thread is created, it is given a default priority of 5 – defined as *Thread.NORM_PRIORITY*.

The Thread Class and the Runnable Interface

- Java's multithreading is built upon the **Thread** class, its **methods**, and **Runnable** interface.
- To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.
- The **Thread** class defines several methods that help manage threads.

Methods Defined by the Thread class

```
public class Thread {  
  
    public Thread(Runnable R); // Thread  $\Rightarrow$  R.run()  
    public Thread(Runnable R, String threadName);  
  
    public void start(); // begin thread execution  
    public void run(); // Entry point for the thread.  
  
    public String getName(); // obtain a thread's name  
    public boolean isAlive(); // Determine if a thread is still running.  
    final public void join(); // Wait for a thread to terminate.  
  
    public void setName(String name); // sets name to the thread  
    public void setPriority(int level); // sets priority to the thread  
  
    public static Thread currentThread(); //references to a main thread  
    public static void sleep(long milliseconds); //Suspend a thread for a  
                                                period of time.  
  
    ...  
}
```

The Main Thread

- When a Java program starts up, one thread begins running immediately.
- This is usually called the *main thread* of your program, because it is the one that is executed when your program begins.
- The main thread is important for two reasons:
 - It is the thread from which other “child” threads will be generated.
 - Often it must be the last thread to finish execution.

- Although the main thread is created automatically, it can be controlled through **Thread** object.
- We can obtain a reference to a main thread by calling the method **currentThread()**, which is a **public static** member of **Thread**.
 - Its general form is:

`static Thread currentThread()`
- This method returns a reference to the thread in which it is called.
- Once we have a reference to the main thread, you can control it just like any other thread.

Ex:-

```
class CurrentThreadDemo {  
    public static void main(String args[]) {  
  
        Thread t = Thread.currentThread();  
        System.out.println("Name of the thread: " + t.getName());  
  
        System.out.println(" Priority : " + t.getPriority());  
  
        // change the name of the thread  
        t.setName("MyThread");  
        System.out.println("After name change: " + t.getName());  
    }  
}
```

```
D:\>javac CurrentThreadDemo.java  
D:\>java CurrentThreadDemo  
        Name of the thread: main  
        Priority : 5  
        After name change: MyThread  
D:\>
```

Creating Threads in java

- Threads are implemented in the *form of objects* that contain a method called *run()*.

The run method contains the code that is to be executed in the thread.

- Java defines two ways to create threads
 - Create a class that implements the **Runnable** interface.
 - Create a class that extends **Thread** class.

I Method: Threads by implementing Runnable interface

1. Declare a class that implements Runnable interface.

```
class MyThread implements Runnable
{
    ....
    public void run()
    {
        // thread body of execution
    }
}
```

2. Create your thread class instance :

```
MyThread myObject = new MyThread();
```

3. Create **Thread** class instance by passing your thread class instance to the **Thread** constructor :

```
Thread thr1 = new Thread( myObject );
```

4. Start Execution:

```
thr1.start();
```


Example1:

```
class MyThread implements Runnable
{
    public void run()
    {
        System.out.println(" this thread is running ... ");
    }
}
class ThreadTest {
    public static void main(String [] args )
    {
        MyThread myObject = new MyThread();
        Thread thr1 = new Thread(myObject);
        // due to implementing the Runnable interface
        // I can call start(), and this will call run().
        thr1.start();
    }
}
```

Output:

this thread is running ...

Example2:

```
class MyThread implements Runnable
{
    public void run()
    {
        System.out.println(" \t\t\tchild thread is running");

        try
        {
            for(int i=1;i<10;i=i+2)
            {
                Thread.sleep(1000);

                System.out.println("\t\t\t"+i);
            }
        } catch ( InterruptedException e ) {}

        System.out.println("\t\t\tExit from child");
    }
}
```

```
class ThreadTest
{
    public static void main( String args[ ] ) throws
        InterruptedException
    {
        System.out.println("Main thread is running");

        Thread t = new Thread(new MyThread());

        t.start();

        for(int i=2;i<10;i=i+2)
        {
            Thread.sleep(1000);
            System.out.println(i);
        }

        System.out.println("Exit from main ");
    }
}
```

Output:

Main thread is running

child thread is running

1

2

3

4

5

6

7

8

Exit from main

9

Exit from child

II method: Extending Thread class

1. Declare a class that extends a **Thread** class

```
class MyThread extends Thread  
{  
    public void run()  
    {  
        // thread body of execution  
    }  
}
```

2. Create your thread class instance in main:

```
MyThread thr1 = new MyThread();
```

3. Start Execution of threads:

```
thr1.start();
```

Ex:-

```
class MyThread extends Thread
```

```
{  
    public void run()  
    {  
        System.out.println(" this thread is running ... ");  
    }  
}
```

```
class ThreadTest
```

```
{  
    public static void main(String [] args )  
    {  
        MyThread thr1 = new MyThread();  
        // due to extending the Thread class (above)  
        // I can call start(), and this will call  
        // run(). start() is a method in class Thread.  
        thr1.start();  
    }  
}
```

Output:

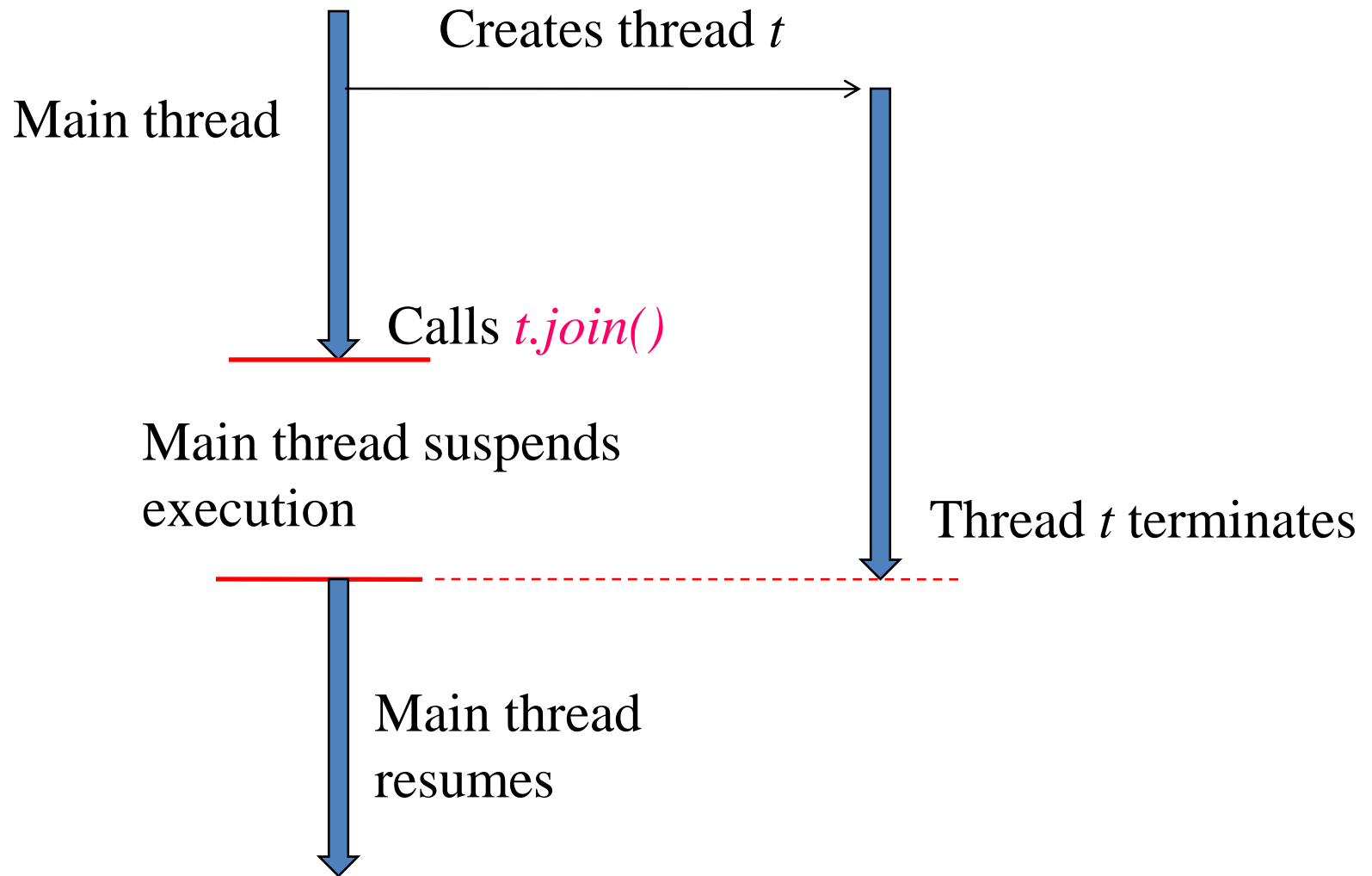
this thread is running ...

join()

- Often we want the main thread to finish last.
- It does not happen automatically.
- We use **join()** to ensure that the main thread is the last to stop.

final void join() throws InterruptedException

- This method waits until the thread on which it is called terminates.



Example:

```
class MyThread implements Runnable
{
    public void run()
    {
        System.out.println(" \t\t\tchild thread is running");

        try
        {
            for(int i=1;i<10;i=i+2)
            {
                Thread.sleep(1000);
                System.out.println("\t\t\t\t"+i);
            }
        } catch ( InterruptedException e ) {}

        System.out.println("\t\t\tExit from child");
    }
}
```

```
class ThreadTest
{
    public static void main( String args[ ] ) throws
        InterruptedException
    {
        System.out.println("Main thread is running");

        Thread t = new Thread(new MyThread());

        t.start();
        t.join();
        for(int i=2;i<10;i=i+2)
        {
            Thread.sleep(1000);
            System.out.println(i);
        }

        System.out.println("Exit from main ");
    }
}
```


Output:

Main thread is running

child thread is running

1

3

5

7

9

Exit from child

2

4

6

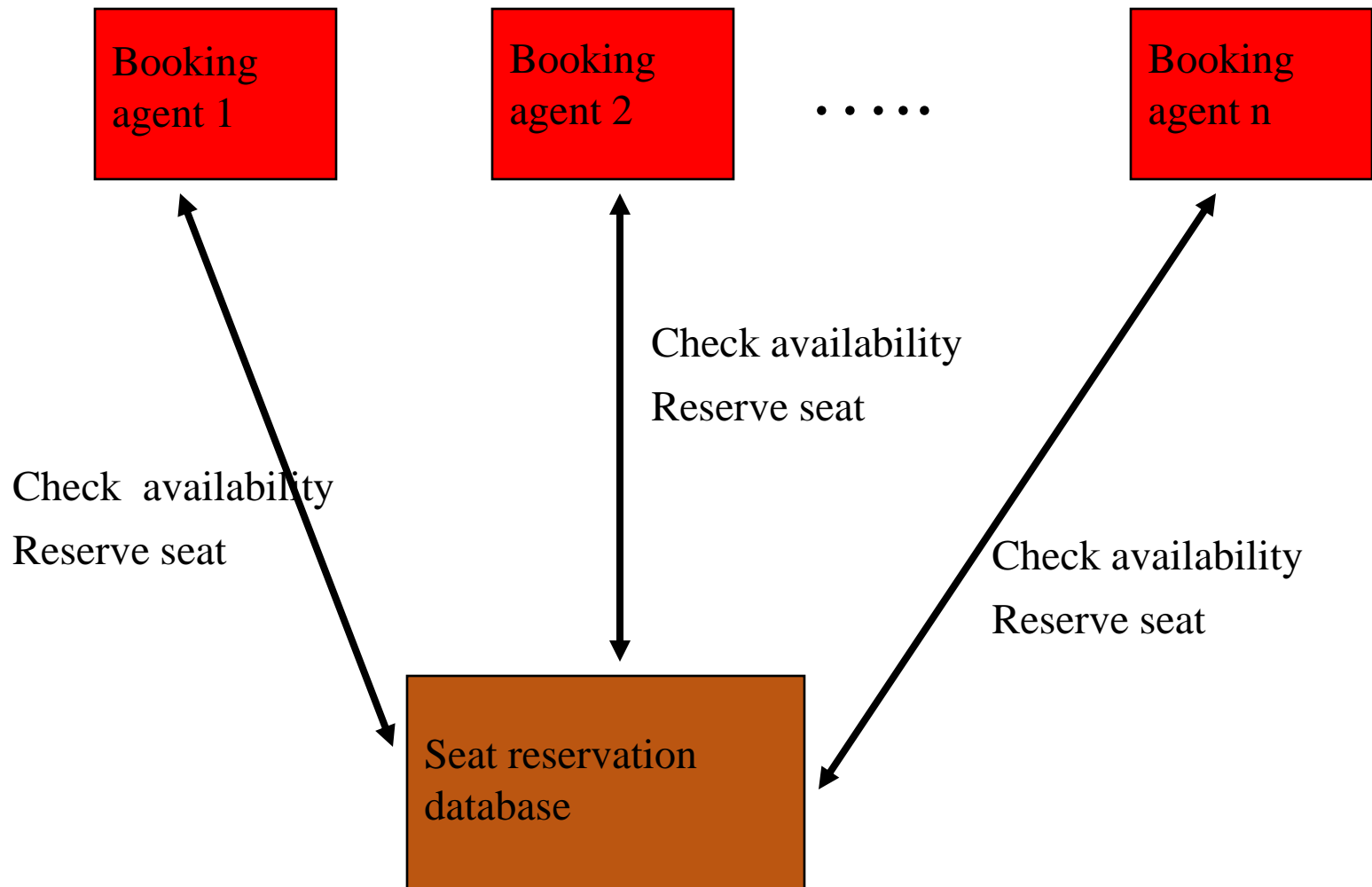
8

Exit from main

Synchronization

- When two or more threads need access to a shared resource, the resource will be used by only one thread at a time is called *synchronization*.
- The process by which this is achieved is called *synchronization*.
- Java uses *synchronized* keyword to provide this feature.

Ex:-



Using Synchronized Methods

- Every object with *synchronized* method is a *monitor*.
- Java allows *one thread at a time* to execute a synchronized method on the monitor.
- When the **synchronized method** is invoked, the object is *locked*. All other threads attempting to invoke synchronized methods must wait.
- When a synchronized method finishes executing, the *lock on the monitor is released*. The highest priority ready thread that invoke a synchronized method will proceed.

Understanding the problem without Synchronization
In this example, there is no synchronization, so output is inconsistent

```
class Table
{
    void printTable(int n)
    {
        for(int i=1; i<=5; i++)
        {
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){ System.out.println(e);}
        }
    }
}
```

```
class MyThread1 extends Thread
{
    Table t;
    MyThread1(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(5);
    }
}
```

```
class MyThread2 extends Thread
{
    Table t;
    MyThread2(Table t)
    {
        this.t=t;
    }
    public void run()
    {
        t.printTable(100);
    }
}
```

```
class TestSynchronization1
{
    public static void main(String args[])
    {
        Table obj = new Table(); //only one object
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);
        t1.start();
        t2.start();
    }
}
```

- Output: *5 100 10 200 15 300 20 400 25 500*
- Output with synchronized method
5 10 15 20 25 100 200 300 400 500

Inter-thread communication in Java

- **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.
- It is implemented by following methods of **Object class**:
 - wait()
 - notify()
 - notifyAll()

- **wait()** - Tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
- **notify()** - Wakes up the first thread that called wait() on the same object.
- **notifyAll()** - Wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

Example:

```
class Test {  
    boolean flag = false;  
    public synchronized void question(String msg) {  
        if (flag) {  
            try {  
                wait();  
            } catch (InterruptedException e) { }  
        }  
        System.out.println(msg);  
        flag = true;  
        notify();  
    }  
}
```

```
public synchronized void answer(String msg) {  
    if (!flag) {  
        try {  
            wait();  
        } catch (InterruptedException e) { }  
    }  
    System.out.println(msg);  
    flag = false;  
    notify();  
}  
}
```

```
class T1 extends Thread
{
    Test m;
    String[] s1 = { "q1.....?", "q2..... ?",
    "q3.....?" };
    public T1(Test m) { this.m = m; }
    public void run() {
        for (int i = 0; i < s1.length; i++) {
            m.question(s1[i]);
        }
    }
}
```

```
class T2 extends Thread
{
    Test m;
    String[ ] s2 = { "a1.....", "a2.....",
    "a3....." };
    public T2(Test m) { this.m = m; }
    public void run() {
        for (int i = 0; i < s2.length; i++) {
            m.answer(s2[i]);
        }
    }
}
```

```

public class TestThread {
    public static void main(String[] args) {
        Test m = new Test();
        T1 q=new T1(m);
        T2 a=new T2(m);
        q.start();
        a.start();
    }
}

```

Output:

```

q1.....?
a1.....
q2..... ?
a2.....
q3.....?
a3.....

```

Daemon Threads

- Threads that run in the background and provide services to an application are called *daemon threads*.
 - There are many java daemon threads running automatically, example , the *garbage collector*.
- Normally when a thread is created in Java, by default it is an user defined thread.
- JVM terminates the daemon thread if there are no user threads.
- Any thread can be converted into a daemon thread simply by passing *true* to the *setDaemon()* method.

Example:

```
class DThread extends Thread
{
    public void run()
    {
        while (true)
        {
            try {
                Thread.sleep(500);
            } catch (InterruptedException x) { }
            System.out.println("\t\t\t Daemon Thread Running");
        }
    }
}
```

```
class DaemonThread
{
    public static void main(String[] args)
    {
        System.out.println("Entering main Method");
        DThread dt = new DThread();
        dt.setDaemon(true);
        dt.start();

        try {
            Thread.sleep(3000);
        } catch (InterruptedException x) {    }

        System.out.println("Leaving main method");
    }
}
```

Output:

Entering main Method

Daemon Thread Running

Daemon Thread Running

Daemon Thread Running

Daemon Thread Running

Daemon Thread Running

Daemon Thread Running

Leaving main method