Threads in Java

1. Threads that each display a string, character by character.

**class** Typist **extends** Thread {

String string;

**public** Typist(String string) {

**this**.string = string;

start(); // start the thread

}

// pause the thread for specified time

**public** **void** sleepThread() {

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

@Override

**public** **void** run() {

// display string character by character

// with a pause between each character

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

System.err.print(string.charAt(i)); // write in error output stream for different color

sleepThread();

}

}

}

// public class

**public** **class** Main {

// main method

**public** **static** **void** main(String[] args) {

System.*out*.println("Start of main...");

**new** Typist("I am a machine");

**new** Typist("HELLO EVERYBODY");

**new** Typist("12345678901234567890");

**new** Typist("...---... ...---... ...---...");

System.*out*.println("End of main.");

}

}

1. Simultaneous access to unsynchronized and synchronized data structures. With an unsynchronized data structure we can obtain the exception java.util.ConcurrentModificationException

**import** java.util.ArrayList;

**import** java.util.Vector;

**class** Proc **extends** Thread {

**int** min, max;

**public** Proc(**int** min, **int** max) {

**this**.min = min;

**this**.max = max;

}

**public** **void** run() {

**try** {

**for** (**int** i = min; i <= max; i++) {

Main.*list*.add(i);

Thread.*sleep*((**long**) (100 + Math.*random*() \* 300));

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

// public class

**public** **class** Main {

**static** ArrayList<Integer> *list* = **new** ArrayList<Integer>(); // unsynchronized

//static Vector<Integer> list = new Vector<Integer>(); // synchronized

// main method

**public** **static** **void** main(String[] args) **throws** InterruptedException {

System.*out*.println("start of main : " + *list*);

**new** Proc(1, 10).start();

**new** Proc(11, 20).start();

Thread.*sleep*(3000);

System.*out*.println("end of main : " + *list*);

}

}

1. Communication between 2 threads.

All resources that can be shared by multiple threads need to be synchronized: (**synchronized** (<monitor>)). The **wait**() method causes the current thread to wait until another thread calls the **notify**() or **notifyAll**() method for the object. It needs to be synchronized on the monitor of the object on which it is called. While waiting, the monitor is liberated. Subsequently, to wake up the thread, call **notify**() or **notifyAll**(). Note that the **notify**() method will wake up one of the threads waiting on this monitor. You cannot specify or know which one will be woken up.

Example of a Producer thread that sends data to a Consumer thread:

Version 1 : In this version, the product is “deposited” directly into the consumer.

**class** Producer **extends** Thread {

Consumer theConsumer;

**public** Producer(Consumer theConsumer) {

**this**.theConsumer = theConsumer;

start();

}

**public** **void** run() {

System.***out***.println("Producer : started");

**for** (**int** product = 0; product < 20; product++) {

**synchronized** (theConsumer) {

**if** (theConsumer.productReady)

**try** {

System.***out***.println("Producer : waiting...");

theConsumer.wait(); // wait for the consumer

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println("Producer : sending product " + product);

theConsumer.product = product;

theConsumer.productReady = **true**;

theConsumer.notify();

}

}

System.***out***.println("Producer : ended...");

}

}

**class** Consumer **extends** Thread {

**int** product;

**boolean** isReady, productReady;

**public** Consumer() {

productReady = **false**;

isReady = **true**;

start();

}

**public** **void** run() {

**boolean** endProduction = **false**;

**while** (!endProduction) {

**synchronized** (**this**) {

**if** (!productReady)

**try** {

System.***out***.println("Consumer : waiting...");

**this**.wait(1000); // wait for the producer, or for 1 second

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**if** (productReady) {

System.***out***.println("Consumer : receiving product " + product);

productReady = **false**; // no product ready for the consumer

notify(); // wakes the producer up so that it can give another product

}

**else**

endProduction = **true**; // product not ready => the producer has ended

}

}

System.***out***.println("Consumer : ended...");

}

}

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Consumer aConsumer = **new** Consumer();

**new** Producer(aConsumer);

}

}

Version 2 : Here we are using an array list to simulate a conveyor.

**import** java.util.ArrayList;

**class** ConveyorException **extends** Exception {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

String message;

**public** ConveyorException(String message) {

**this**.message = message;

}

**public** String toString() {

**return** message;

}

}

**class** Conveyor {

**int** capacity;

ArrayList<Integer> list;

**public** Conveyor(**int** capacity) {

**this**.capacity = capacity;

list = **new** ArrayList<Integer>();

}

**public** **synchronized** **boolean** isFull() {

**return** list.size() == capacity;

}

**public** **synchronized** **boolean** isEmpty() {

**return** list.isEmpty();

}

**public** **synchronized** **void** addProduct(**int** product) **throws** ConveyorException {

**if** (isFull())

**throw** **new** ConveyorException("The conveyor cannot contain more than " + capacity + " products.");

list.add(product);

}

**public** **synchronized** Integer removeProduct() **throws** ConveyorException {

**if** (isEmpty())

**throw** **new** ConveyorException("The conveyor is empty.");

**return** list.remove(0);

}

}

**class** Producer **extends** Thread {

Conveyor conveyor;

**int** delay;

**public** Producer(Conveyor conveyor, **int** delay) {

**this**.conveyor = conveyor;

**this**.delay = delay;

start();

}

**public** **void** run() {

System.***out***.println("Producer : started");

**for** (**int** product = 0; product < 20; product++) {

// processing time for preparing new product

System.***out***.println("Producer : preparing product...");

**try** {

Thread.*sleep*(delay);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

// wait while the conveyor is full

**while** (conveyor.isFull())

**try** {

System.***out***.println("Producer : conveyor full... sleeping...");

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

// deposit the product (the conveyor cannot be full, because the

// consumer diminishes the number of products)

System.***out***.println("Producer : adding product product " + product);

**try** {

conveyor.addProduct(product);

} **catch** (ConveyorException e) {

e.printStackTrace();

}

}

System.***out***.println("Producer : ended...");

}

}

**class** Consumer **extends** Thread {

Conveyor conveyor;

**int** delay;

**public** Consumer(Conveyor conveyor, **int** delay) {

**this**.conveyor = conveyor;

**this**.delay = delay;

start();

}

**public** **void** run() {

**boolean** endProduction = **false**;

**while** (!endProduction) {

// sleep while conveyor is empty

**int** counter = 10;

**while** (counter-- > 0 && conveyor.isEmpty()) {

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**if** (counter > 0) {

// remove the product from the conveyor (the conveyor cannot be empty

// because the producer increases the number of products)

**int** product = 0;

**try** {

product = conveyor.removeProduct();

} **catch** (ConveyorException e) {

e.printStackTrace();

}

System.***out***.println("Consumer : receiving product " + product);

// product processing time

System.***out***.println("Consumer : processing product " + product);

**try** {

Thread.*sleep*(delay);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

} **else**

endProduction = **true**;

}

System.***out***.println("Consumer : ended...");

}

}

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Conveyor theConveyor = **new** Conveyor(10);

**new** Producer(theConveyor, 700);

**new** Consumer(theConveyor, 1000);

}

}