
Posted by [mirudom](#) on Sat, 17 Jun 2023 12:18:17 GMT
[View Forum Message](#) <> [Reply to Message](#)

```
package no.packeges;

public class HelloWorld {
    public static void main( String[] args){
        System.out.println("Hello World");
    }
}
```

```

package main

import (
    "fmt"
    "os"
)

type HelloWorld struct{}

func main() {

    var args []string = os.Args

    var hw HelloWorld = HelloWorld{}
    hw.HelloWorld_main(args)
}

/** generated method */
func (helloWorld *HelloWorld) HelloWorld_main(args []string) {
    fmt.Println("Hello World")
}

```

```

package no.packages;

public class TestInheritance {
    public static void main( String[] args){
        Inheritance inh=null;
        inh=new Second();
        inh.hello();
        inh=new Third();
        inh.hello();
    }
}

public interface Inheritance {
    public void hello();
}

class Second implements Inheritance {
    public void hello(){
        System.out.println("Second");
    }
}

class Third implements Inheritance {

```

```
public void hello(){
    System.out.println("Third");
}
}
```

```
package main
```

```
import (
    "fmt"
    "os"
)
```

```
type TestInheritance struct{}
```

```
func main() {
```

```
    var args []string = os.Args
```

```
    var ti TestInheritance = TestInheritance{}
    ti.TestInheritance_main(args)
```

```
}
```

```
/** generated method **/
```

```
func (testInheritance *TestInheritance) TestInheritance_main(args []string) {
```

```
    var inh Inheritance
    inh = AddressSecond(Second{})
    inh.hello()
    inh = AddressThird(Third{})
    inh.hello()
}
```

```
type Inheritance interface {
    hello()
}
```

```
type Second struct{}
```

```
func (second *Second) hello() {
    fmt.Println("Second")
}
```

```
type Third struct{}
```

```
func (third *Third) hello() {
    fmt.Println("Third")
}
```

```
func AddressSecond(s Second) *Second { return &s }
func AddressThird(t Third) *Third { return &t }
```

```
package no.packages;
```

```
public class TestExtension {
    public static void main( String[] args){
        TestExtension te=new TestExtension();
        te.hello();
        te=new Second();
        te.hello();
        te=new Third();
        te.hello();
        te=new Fourth();
        te.hello();
    }
    public void hello(){
        System.out.println("hello");
    }
}
class Second extends TestExtension {
    public void hello(){
        System.out.println("Second");
    }
}
class Third extends TestExtension {
    public void hello(){
        System.out.println("Third");
    }
}
class Fourth extends Third {
    public void hello(){
        System.out.println("Fourth");
    }
}
```

```
package main
```

```
import (
    "fmt"
    "os"
```

```

)

type TestExtension struct{}

func main() {

    var args []string = os.Args

    var te TestExtension = TestExtension{}
    te.TestExtension_main(args)
}
func (testExtension *TestExtension) hello() {
    fmt.Println("hello")
}

/** generated method **/
func (testExtension *TestExtension) TestExtension_main(args []string) {

    var te ITestExtension = AddressTestExtension(TestExtension{})
    te.hello()
    te = AddressSecond(Second{})
    te.hello()
    te = AddressThird(Third{})
    te.hello()
    te = AddressFourth(Fourth{})
    te.hello()
}

type Second struct {
    TestExtension
}

func (second *Second) hello() {
    fmt.Println("Second")
}

type Third struct {
    TestExtension
}

func (third *Third) hello() {
    fmt.Println("Third")
}

type Fourth struct {
    Third
}

```

```

func (fourth *Fourth) hello() {
    fmt.Println("Fourth")
}

type ITestExtension interface { /** Generated Method */

    hello()
}

func AddressSecond(s Second) *Second { return &s }
func AddressThird(t Third) *Third { return &t }
func AddressTestExtension(t TestExtension) *TestExtension
{ return &t }
func AddressFourth(f Fourth) *Fourth { return &f }

```

Repeater.

```

package main;

public class Speaker {
    String message;
    public static void main( String[] args){
        Speaker sp = new Speaker("Say hello !");
        System.out.println(sp.amount());
        sp.speak();
        Repeater rp = new Repeater("Say hello !",3);
        System.out.println(rp.amount());
        rp.speak();
    }
    public Speaker( String message){
        this.message=message;
    }
    public void speak(){
        for (int i=0; i < amount(); i++) {
            System.out.println(this.message);
        }
    }
    public int amount(){
        return 1;
    }
}

class Repeater extends Speaker {
    int to_repeat=0;

```

```

public Repeater( String message, int amount){
    super(message);
    this.to_repeat=amount;
}
public int amount(){
    return this.to_repeat;
}
}

```

```

package main

```

```

import (
    "fmt"
    "os"
)

```

```

type Speaker struct {
    message string
}

```

```

func main() {

```

```

    var args []string = os.Args

```

```

    var s_dummy Speaker = NewSpeaker("")
    s_dummy.Speaker_main(args)
}

```

```

func NewSpeaker(message string) Speaker {

```

```

    var speaker Speaker = Speaker{message}
    return speaker
}

```

```

func (speaker *Speaker) speak() {
    for i := 0; i < speaker.amount(); i++ {
        fmt.Println(speaker.message)
    }
}

```

```

func (speaker *Speaker) amount() int {
    return 1
}

```

```

/** generated method */

```

```

func (speaker *Speaker) Speaker_main(args []string) {

```

```

    var sp ISpeaker = AddressSpeaker(NewSpeaker("Say hello !"))
    fmt.Println(sp.amount())
}

```

```
sp.speak()
```

```
var rp ISpeaker = AddressRepeater(NewRepeater("Say hello !", 3))
fmt.Println(rp.amount())
rp.speak()
}
```

```
type Repeater struct {
    Speaker
    to_repeat int
}
```

```
func NewRepeater(message string, amount int) Repeater {
    var repeater Repeater = Repeater{NewSpeaker(message), amount}
    return repeater
}
func (repeater *Repeater) amount() int {
    return repeater.to_repeat
}
func (repeater *Repeater) speak() {
    for i := 0; i < repeater.amount(); i++ {
        fmt.Println(repeater.message)
    }
}
```

```
type ISpeaker interface {

    /** Generated Method */

    amount() int
    /** Generated Method */

    speak()
}
```

```
func AddressRepeater(r Repeater) *Repeater { return &r }
func AddressSpeaker(s Speaker) *Speaker { return &s }
```

```
package com.builder.start.here;
```

```
public class CatchException {

    public static void main(String[] args) {
        try {
            new ThrowException().runme();
        } catch (Exception e) {
```



```

    System.out.println("yes, I caught it");
} finally {
    System.out.println("finally processing");
}

}

}

```

```

class ThrowException{
    public void runme() throws Exception{
        throw new Exception();
    }
}

```

```

package main

```

```

import (
    "fmt"
    "os"
)

```

```

type CatchException struct{}

```

```

func main() {
    var args []string = os.Args
    var ce CatchException = CatchException{}
    ce.CatchException_main(args)
}

```

```

/** generated method */

```

```

func (catchException *CatchException) CatchException_main(args []string) {
    defer func() {
        if err := recover(); err != nil {
            exc := err.(Exception)
            switch exc.msg {
            case "Exception":
                fmt.Println("yes, I caught it")
            default:
                fmt.Println("No, something is not right")
            }
        }
    }()
    fmt.Println("finally processing")
}()

```

```

(&ThrowException{}).runme()

```

```

}

type ThrowException struct{}

func (throwException *ThrowException) runme() {
    panic(Exception{"Exception"})
}

type Exception struct {
    msg string
}

```

```

package run.me;
public class One {
    String msg;
    public static void main( String[] args)
        throws InterruptedException {
        One one=new One();
        one.msg="Initial message";
        System.out.println("main:before start:" +
            one.msg + " second part of");
        new Second(one).start();
        one.msg="after go start";
        Thread.sleep(2000);
        System.out.println("main:about to end:" + one.msg);
    }
}
class Second extends Thread {
    One one;
    public Second( One one){
        this.one=one;
    }
    public void run(){
        try {
            sleep(1000);
        } catch ( InterruptedException e) {
        }
        System.out.println("run:after sleep:" + one.msg);
        one.msg="try to change msg";
    }
}

```

```
}  
}
```

```
package main
```

```
import (  
    "fmt"  
    "os"  
    "time"  
)
```

```
type One struct {  
    msg *string  
}
```

```
func NewOne() *One {  
    var one One = One{}  
    return &one  
}
```

```
func main() {  
    var args []string = os.Args  
    var o_dummy *One = NewOne()  
    o_dummy.One_main(&args)  
}
```

```
/** generated method */  
func (one_one *One) One_main(args *[]string) {  
    var one *One = NewOne()  
    s := "Initial message"  
    one.msg = &s  
    fmt.Println("main:before start:" +  
        *one.msg + " second part of")  
    go NewSecond(one).run()  
    s_s := "after go start"  
    one.msg = &s_s  
    time.Sleep(2000)  
    fmt.Println("main:about to end:" + *one.msg)  
}
```

```
type Second struct {  
    one *One  
}
```

```
func NewSecond(one *One) *Second {  
    var second Second = Second{one}  
    return &second  
}
```

```
}  
func (second *Second) run() {  
    time.Sleep(1000)  
    fmt.Println("run:after sleep:" + *second.one.msg)  
    s_s_s := "try to change msg"  
    second.one.msg = &s_s_s  
}
```

channels.

```
package com.builder.start.here;  
  
import java.util.Random;  
  
public class RunMeDrop {  
    public static void main(String[] args) {  
        Drop drop = new Drop();  
        (new Thread(new Producer(drop))).start();  
        (new Thread(new Consumer(drop))).start();  
    }  
}  
  
class Drop {  
    // Message sent from producer  
    // to consumer.
```

```

private String message;
// True if consumer should wait
// for producer to send message,
// false if producer should wait for
// consumer to retrieve message.
private boolean empty = true;

public synchronized String take() {
    // Wait until message is
    // available.
    while (empty) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    // Toggle status.
    empty = true;
    // Notify producer that
    // status has changed.
    notifyAll();
    return message;
}

public synchronized void put(String message) {
    // Wait until message has
    // been retrieved.
    while (!empty) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    // Toggle status.
    empty = false;
    // Store message.
    this.message = message;
    // Notify consumer that status
    // has changed.
    notifyAll();
}
}

class Consumer implements Runnable {
    private Drop drop;

    public Consumer(Drop drop) {
        this.drop = drop;
    }

    public void run() {

```

```

Random random = new Random();
for (
    String message = drop.take();
    ! message.equals("DONE");
    message = drop.take()) {
    System.out.format(
        "MESSAGE RECEIVED: %s%n", message);
    try {
        Thread.sleep(random.nextInt(5000));
    } catch (InterruptedException e) {}
}
}
}
class Producer implements Runnable {
    private Drop drop;

    public Producer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };
        Random random = new Random();

        for (int i = 0; i < importantInfo.length; i++) {
            drop.put(importantInfo[i]);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
        drop.put("DONE");
    }
}
}

```

```
package main
```

```

import (
    "fmt"
    "math/rand"
    "os"
    "sync"

```

```

"time"
)

type RunMe struct{}

func NewRunMe() *RunMe {
    var runMe RunMe = RunMe{}
    return &runMe
}
func main() {

    var args []string = os.Args
    var rm *RunMe = NewRunMe()
    rm.RunMe_main(&args)
}

/** generated method */
func (runMe *RunMe) RunMe_main(args *[]string) {
    wg := &sync.WaitGroup{}
    var drop *Drop = NewDrop()
    wg.Add(1)
    go (&Thread{NewProducer(drop)}).run(wg)
    wg.Add(1)
    go (&Thread{NewConsumer(drop)}).run(wg)
    wg.Wait()
}

type Drop struct {
    /** generated field */
    cond *sync.Cond
    message *string
    empty *bool
}

func NewDrop() *Drop {

    var drop Drop = Drop{}
    drop.cond = sync.NewCond(&sync.Mutex{})
    empty := true
    drop.empty = &empty
    return &drop
}
func (drop *Drop) take() string {
    drop.cond.L.Lock()
    for *drop.empty {
        drop.cond.Wait()
    }
}

```

```

*drop.empty = true
drop.cond.L.Unlock()
drop.cond.Broadcast()
return *drop.message
}
func (drop *Drop) put(message *string) {
drop.cond.L.Lock()
for !*drop.empty {
drop.cond.Wait()
}

*drop.empty = false
drop.message = message
drop.cond.L.Unlock()
drop.cond.Broadcast()
}

type Consumer struct {
drop *Drop
}

func NewConsumer(drop *Drop) *Consumer {
var consumer Consumer = Consumer{drop}
return &consumer
}
func (consumer *Consumer) run(wg *sync.WaitGroup) {
defer wg.Done()

var random *Random = NewRandom()
for message := consumer.drop.take();
message != "DONE";
message = consumer.drop.take() {
fmt.Printf("MESSAGE RECEIVED: %s\n", message)
time.Sleep(random.nextInt(5000))
}
}

type Producer struct {
drop *Drop
}

func NewProducer(drop *Drop) *Producer {
var producer Producer = Producer{drop}
return &producer
}
func (producer *Producer) run(wg *sync.WaitGroup) {
defer wg.Done()

```



```

var importantInfo [4]string = [4]string{
    "Mares eat oats",
    "Does eat oats",
    "Little lambs eat ivy",
    "A kid will eat ivy too",
}

var random *Random = NewRandom()
for i := 0; i < len(importantInfo); i++ {
    producer.drop.put(&importantInfo[i])
    time.Sleep(random.nextInt(5000))
}
s := "DONE"
producer.drop.put(&s)
}

type Random struct{}

func NewRandom() *Random {
    var random Random = Random{}
    return &random
}

func (r *Random) nextInt(n int) time.Duration {
    return time.Duration(rand.Intn(n))
}

type Thread struct {
    Runnable
}

func (t *Thread) run(wg *sync.WaitGroup) {
    t.Runnable.run(wg)
}

type Runnable interface {
    run(wg *sync.WaitGroup)
}

```

```

package com.builder.start.here;

import java.util.Random;

public class RunMe {
    public static void main(String[] args) {
        Drop drop = new Drop();
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
}

class Drop {
    // Message sent from producer
    // to consumer.
    private String message;
    // True if consumer should wait
    // for producer to send message,
    // false if producer should wait for
    // consumer to retrieve message.
    private boolean empty = true;

    public synchronized String take() {
        // Wait until message is
        // available.
        while (empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = true;
        // Notify producer that
        // status has changed.
        notifyAll();
        return message;
    }

    public synchronized void put(String message) {
        // Wait until message has
        // been retrieved.
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
    }
}

```

```

    // Toggle status.
    empty = false;
    // Store message.
    this.message = message;
    // Notify consumer that status
    // has changed.
    notifyAll();
}
}
class Consumer implements Runnable {
    private Drop drop;

    public Consumer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        Random random = new Random();
        for (
            String message = drop.take();
            ! message.equals("DONE");
            message = drop.take()) {
            System.out.format(
                "MESSAGE RECEIVED: %s%n", message);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {}
        }
    }
}
class Producer implements Runnable {
    private Drop drop;

    public Producer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };
        Random random = new Random();

        for (int i = 0; i < importantInfo.length; i++) {
            drop.put(importantInfo[i]);
        }
    }
}

```

```

        try {
            Thread.sleep(random.nextInt(5000));
        } catch (InterruptedException e) {}
    }
    drop.put("DONE");
}
}

```

```
package main
```

```
import (
    "fmt"
    "math/rand"
    "os"
    "sync"
    "time"
)
```

```
type RunMe struct{}
```

```
func NewRunMe() *RunMe {
    var runMe RunMe = RunMe{}
    return &runMe
}

```

```
func main() {
    var args []string = os.Args
    var rm *RunMe = NewRunMe()
    rm.RunMe_main(&args)
}

```

```
/** generated method **/
```

```
func (runMe *RunMe) RunMe_main(args *[]string) {
    wg := &sync.WaitGroup{}
    var drop chan string = make(chan string)
    wg.Add(1)
    go (&Thread{NewProducer(drop)}).run(wg)
    wg.Add(1)
    go (&Thread{NewConsumer(drop)}).run(wg)
    wg.Wait()
}

```

```
type Consumer struct {
    drop chan string
}

```

```
func NewConsumer(drop chan string) *Consumer {
    var consumer Consumer = Consumer{drop}
    return &consumer
}

```

```

func (consumer *Consumer) run(wg *sync.WaitGroup) {
    defer wg.Done()
    var random *Random = NewRandom()
    for message := <-consumer.drop;
        message != "DONE";
        message = <-consumer.drop {
        fmt.Printf("MESSAGE RECEIVED: %s\n", message)
        time.Sleep(random.nextInt(5000))
    }
}
type Producer struct {
    drop chan string
}
func NewProducer(drop chan string) *Producer {
    var producer Producer = Producer{drop}
    return &producer
}
func (producer *Producer) run(wg *sync.WaitGroup) {
    defer wg.Done()
    var importantInfo [4]string = [4]string{
        "Mares eat oats",
        "Does eat oats",
        "Little lambs eat ivy",
        "A kid will eat ivy too"}
    var random *Random = NewRandom()
    for i := 0; i < len(importantInfo); i++ {
        producer.drop <- (importantInfo[i])
        time.Sleep(random.nextInt(5000))
    }
    producer.drop <- ("DONE")
}

type Random struct{}
func NewRandom() *Random {
    var random Random = Random{}
    return &random
}
func (r *Random) nextInt(n int) time.Duration { return time.Duration(rand.Intn(n)) }
type Thread struct {
    Runnable
}
func (t *Thread) run(wg *sync.WaitGroup) {
    t.Runnable.run(wg)
}
type Runnable interface {
    run(wg *sync.WaitGroup)
}

```

Posted by [javadev](#) on Wed, 01 Nov 2023 08:01:10 GMT

[View Forum Message](#) <> [Reply to Message](#)

```
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

func main() {
    drop := NewDrop()
    go Producer(drop)
    go Consumer(drop)
    select{}
}

type Drop struct {
    message string
    empty bool
    mutex sync.Mutex
    notEmpty *sync.Cond
    notFull *sync.Cond
}

func NewDrop() *Drop {
    drop := &Drop{
        empty: true,
        notEmpty: sync.NewCond(new(sync.Mutex)),
        notFull: sync.NewCond(new(sync.Mutex)),
    }
    return drop
}

func (d *Drop) Take() string {
```

```
d.mutex.Lock()
for d.empty {
    d.notEmpty.Wait()
}
message := d.message
d.empty = true
d.notFull.Signal()
d.mutex.Unlock()
return message
}
```

```
func (d *Drop) Put(message string) {
    d.mutex.Lock()
    for !d.empty {
        d.notFull.Wait()
    }
    d.message = message
    d.empty = false
    d.notEmpty.Signal()
    d.mutex.Unlock()
}
```

```
func Consumer(drop *Drop) {
    rand.Seed(time.Now().UnixNano())
    for {
        message := drop.Take()
        if message == "DONE" {
            break
        }
        fmt.Printf("MESSAGE RECEIVED: %s\n", message)
        time.Sleep(time.Duration(rand.Intn(5000)) * time.Millisecond)
    }
}
```

```
func Producer(drop *Drop) {
    importantInfo := []string{
        "Mares eat oats",
        "Does eat oats",
        "Little lambs eat ivy",
        "A kid will eat ivy too",
    }
    rand.Seed(time.Now().UnixNano())
    for _, info := range importantInfo {
        drop.Put(info)
        time.Sleep(time.Duration(rand.Intn(5000)) * time.Millisecond)
    }
    drop.Put("DONE")
}
```
