THREADS IN JAVA

Java provides two ways to create a new thread.
– Extend the Thread class (java.lang.Thread)
– Implement the Runnable interface (java.lang.*Runnable*)

When creating a new thread by extending the Thread class, you should override the run() method.

```
public class FooThread extends Thread
{
     public FooThread()
     {
          // Initialize parameters
     }
     public void run()
     {
          // do something
     }
}
```

To start a new thread, use the inherited method start()

```
FooThread ft = new FooThread();
ft.start();
```

start() is responsible for two things:

– Instructing the JVM to create a new thread
– Call your Thread object's run() method in the new thread

You might think of run() as being similar to main()

Like main(), run() defines a starting point for the JVM.
What happens when the run() method exits?

– The thread 'ends'.

RUNNABLE INTERFACE

A thread can also be created by implementing the `Runnable` interface instead of extending the `Thread` class.

```
public class FooRunnable implements Runnable
{
     public FooRunnable()
     {
     }
     public void run()
     {
          // do something
     }
}
```

Pass an object that implements `Runnable` to the constructor of a `Thread` object, then start the thread as before.

```
FooRunnable fr = new FooRunnable();
new Thread(fr).start()
```

JOIN METHOD

The `join()` method is used to wait until a thread is done.

- The caller of `join()` blocks until thread finishes.
- Why might this be bad?
  - Blocking the main thread will make UI freeze.

Other versions of join take in a timeout value: where if thread doesn't finish before timeout, `join()` returns.

Alternatively, can check on a thread with "`isAlive()`" method

- Returns `true` if running, `false` if finished.

SLEEP METHOD

The sleep() method pauses execution of the current thread.

- Periodic actions: need to wait for some period of time before doing the action again.
- Allow other threads to run

Implemented as a class method, so you don't actually need a Thread object

```
Thread.sleep(1000);  // Pause here for 1 second.
```

ESERCIZIO 1

Si costruisca un programma che lancia due *thread*, il primo di nome Paolo, il secondo di nome Egidio. Il *thread* principale deve far partire Egidio 1 sec dopo Paolo, dunque mettersi in attesa di Paolo. Paolo e Egidio si limitano a contare fino a 5, aspettando 0,5 sec a ogni passo.

```
// file : ./javaapplication/Filo.java
package javaapplication;
import java.io.*;

public class Filo extends Thread {

// L'interfaccia java.lang.Runnable è anche implementata dalla classe Thread che si sta qui utilizzando,
// infatti l'intestazione della classe Thread è: """public class Thread extends Object implements Runnable"""

// Il metodo sleep(..) in Thread è definito come: """ public static void sleep() throws InterruptedException """
// mentre il metodo run() di Runnable è definito come: """ public void run( ) """
// Dovendo richiamare sleep(..) nel metodo run(); si è obbligati a gestire l'eccezione sollevata da sleep
// perché si sta facendo overriding e nell'intestazione di  run() non è previsto il sollevamento di eccezioni.

Filo(String nome){ super(nome); }
public void run() {
        for (int i = 1; i <= 5; i++) {
            try {
                 System.out.println(getName()+" : "+i);
                 Thread.sleep(500);
            } catch (InterruptedException e) {
               System.out.println(getName()+" interrotto!"+e.toString());
            }
        } // end for
    } // end run()
} // end class Filo

// file : ./javaapplication/demoFilo.java
package javaapplication;
import java.io.*;
import java.lang.*;
public class demoFilo {
    public static void main(String[] args) throws InterruptedException {
                                 // NON GESTISCO L'ECCEZIONE SOLLEVATA DA Thread.sleep(..)
        Filo p = new Filo("Paolo");
        Filo e = new Filo("Egidio");

        p.start();
        Thread.sleep(1000);
        e.start();
        p.join();
    }
}
```

ACCESS SYNCHRONIZATION

Every Thread object has a lock associated with it.

If you declare a method to be “`synchronized`”, then lock will be obtained before executing the method. Can use this to make sure that only one thread at a time is accessing an object.

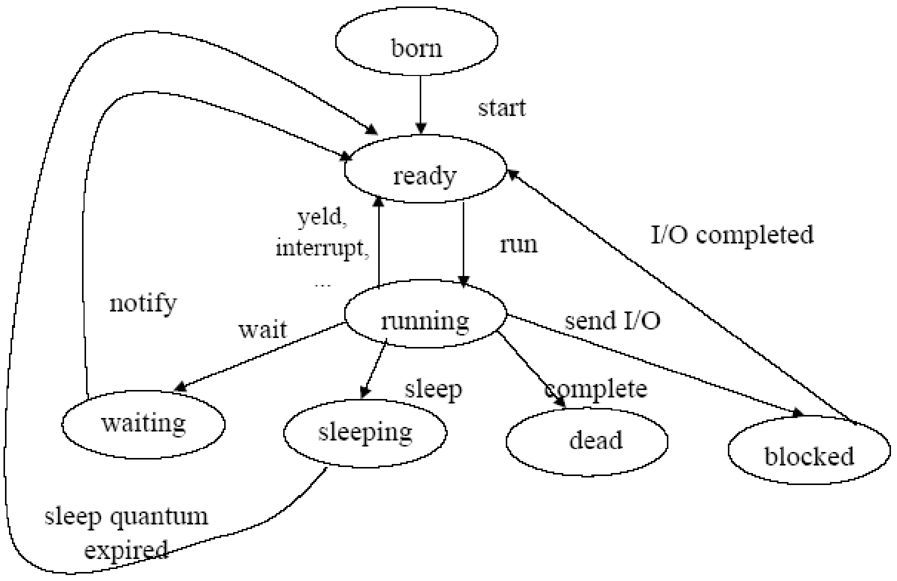
Example of synchronized method declaration:

```
synchronized int some_method (int param)
```

WAIT & NOTIFY METHODS

It’s important to understand that `sleep()` *does not* release the lock when it is called. On the other hand, the method `wait()` does release the lock, which means that other `synchronized` methods in the thread object can be called during a `wait( )`.

Every object maintains a list of “waiting” threads.
A thread that is waiting at/on a particular object is suspended until some other thread wakes it.
`Notify()` or `NotifyAll()` awakens a thread that is waiting.

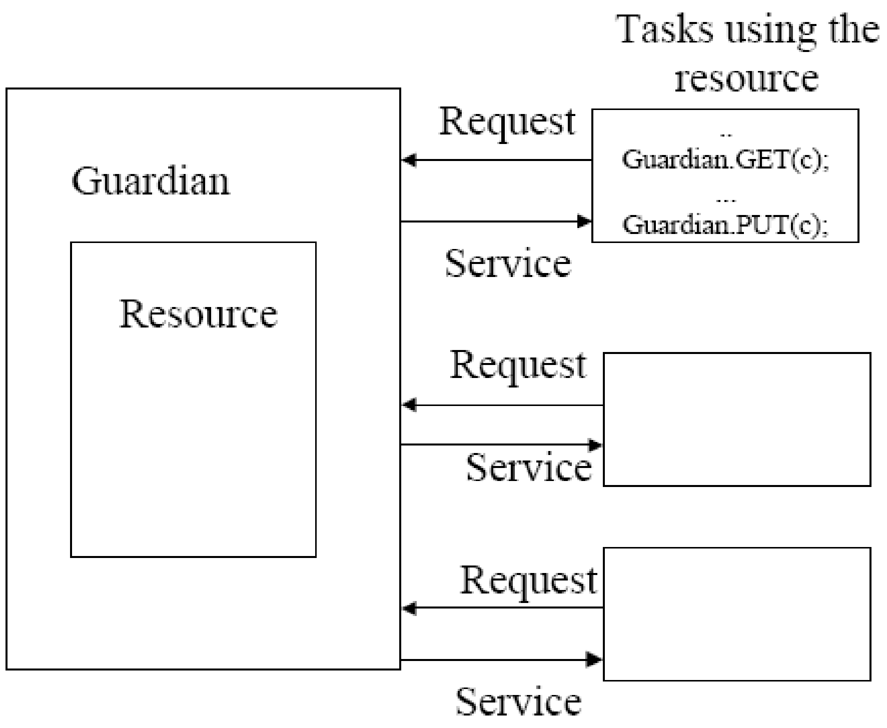


`wait(),notify(),notifyAll()` are methods associated with ANY object.

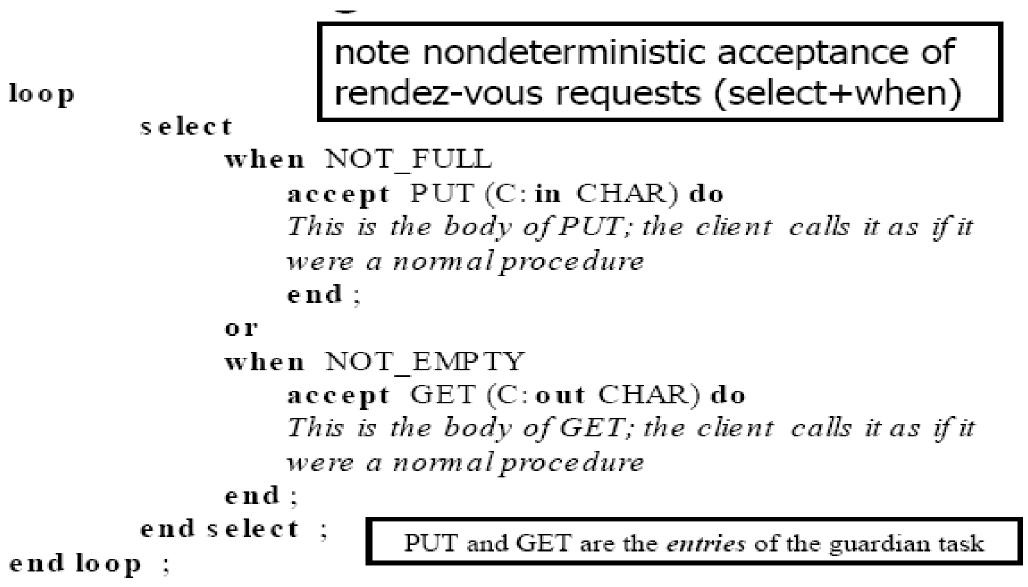
Deadlock: Because threads can become blocked and because objects can have synchronized methods that prevent threads from accessing that object until the synchronization lock is released, it's possible for one thread to get stuck waiting for another thread, which in turn waits for another thread, etc.You get a continuous loop of threads waiting on each other, and no one can move.
Starvation: Starvation occurs when a process waits for a resource that continually becomes available, BUT is NEVER assigned to that process because of priority or a flaw in the design of the scheduler.

TASK SYNCRONIZATION IN ADA

- Guardians and rendez-vous
- The Ada style of designing concurrent systems
- In Ada a shared object is active (whereas a monitor is passive)
  - it is managed by a *guardian* process which can accept rendez-vous requests from tasks willing to access the object



A GUARDIAN TASK



note nondeterministic acceptance of rendez-vous requests (select+when)

```
loop
        select
            when NOT_FULL
                accept PUT (C: in CHAR) do
                This is the body of PUT; the client calls it as if it
                were a normal procedure
                end ;
            or
            when NOT_EMPTY
                accept GET (C: out CHAR) do
                This is the body of GET; the client calls it as if it
                were a normal procedure
            end ;
        end select ;
end loop ;
```

PUT and GET are the *entries* of the guardian task

SEMANTICA DEI RENDEZVOUZ

1) chiamata ad **ENTRY** (analoghe a chiamata di procedura)
2) il task chiamato deve fare una **ACCEPT** (*rendezvous)*

Di solito task body usa costrutto **SELECT** per decidere se e quando accettare un rendezvous.

```
SELECT
 WHEN C1 => PRG1
OR
 WHEN C2 => PRG2
OR
 PRG3 -- sempre aperta
ELSE
 PRG4
END SELECT;
```

- C1 e C2 sono alternative APERTE se e soltanto se
  C1 = true and C2 = true
- PRGk può contenere **ACCEPT** o **DELAY**

1) Se esiste PRGk con Ck aperta e contenente una **ACCEPT** esegui un PRGk scelto non deterministicamente.

Altrimenti

2) Se non esiste, allora esegui PRGk' con Ck' aperta e **DELAY** (il più piccolo **DELAY** aperto)

Altrimenti

3) Se tutto chiuso, esegui ramo **ELSE**.

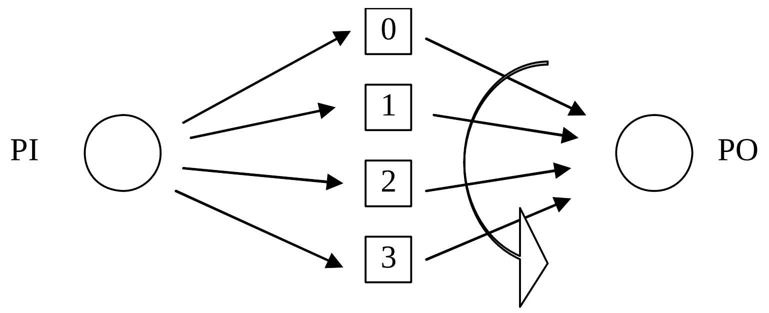
ESERCIZIO 2 (THREAD/TASK)

Si realizzi in Java e in Ada il seguente sistema.

- Un modulo PI esegue ripetutamente le seguenti operazioni: legge da tastiera una coppia di valori *<i, ch>*, dove *i* è un numero tra 0 e 3, *ch* un carattere, e inserisce il carattere *ch* nel buffer *i*-esimo (ognuno dei quattro buffer contiene al più un carattere).

- Un modulo PO considera a turno in modo circolare i quattro buffer e preleva il carattere in esso contenuto, scrivendo in uscita la coppia di valori *<i, ch>* se ha appena prelevato il carattere *ch* dal buffer *i*-esimo.

- L' accesso a ognuno dei buffer è in mutua esclusione; PI rimane bloccato se il buffer a cui accede è pieno, PO rimane bloccato se il buffer a cui accede è vuoto.



Data la seguente sequenza di valori letta da PI, scrivere la sequenza scritta in corrispondenza da PO.

<1, c> <0, b> <2, m> <0, f> <1, h> <3, n>

R. <0, b> <1, c> <2, m> <3, n> <0, f> <1, h>

Descrivere brevemente in quali casi si può verificare una situazione di *deadlock* tra PI e PO. Illustrare con un semplice esempio.

R: Deadlock: <1,a> <1,b>

VERSIONE JAVA

```
// riceve l'input via linea di comando,es."0:a 1:b 2:c")
import java.lang.*;
import java.io.*;

class Buf {
  private char ch;
  private boolean full;
  Buf(){ full = false;  }

  public synchronized void put (char item) throws InterruptedException {

     while (full) {
       wait(); //try{  wait(); } catch (InterruptedException ie) {};
     }
     ch = item;
     full = true;
     notify();
 }
 public synchronized char get() throws InterruptedException {

     while (!full) {
         wait(); // try{  wait(); } catch (InterruptedException ie) {};
     }
     full = false;
     notify();
     return ch;
 }
}
```

```
class Pi extends Thread {

   private Buf[] buff;
   private String[] commands;
   // commands = ["1,A", "2,B", "0,D"...]
   Pi (Buf[] b, String[] c) {
      buff = b;
      commands = c;
   }

   public void run() {
       int i, index;
       for(i=0; i < commands.length; i++) {
           try {
                 index = (int)(commands[i].charAt(0))-(int)'0';
                 buff[index].put(commands[i].charAt(2));
           } catch (InterruptedException ie) {
                  System.out.println("Monitor Problems -- wait()! ");
           }
       }
   }
}

class Po extends Thread {
    private Buf[] buff;
    Po (Buf[] b) { buff = b; }
    public void run() {
        while (true) {
           try {
               for (int i=0; i < buff.length; i++)
                  System.out.println("Buff "+i+":"+buff[i].get());
           } catch (InterruptedException ie) {
               System.out.println("Monitor Problems -- wait()! ");
           }
        }
    }
}

public class pi_po {

  public static void main(String[] args) throws InterruptedException {
        Buf[] bfs = new Buf[4];

        bfs[0] = new Buf();bfs[1] = new Buf();
        bfs[2] = new Buf();bfs[3] = new Buf();

        Pi pi0 = new Pi(bfs, args);
        Po po0 = new Po(bfs);

        pi0.start();
        po0.start();

    }

}
```

VERSIONE ADA

```
with TEXT_IO;
with Ada.Integer_text_IO;
use Ada.Integer_text_IO;
use TEXT_IO;
procedure PI_PO is
 task type BUF is
  entry PUT (X: in CHARACTER);
  entry GET (X: out CHARACTER);
 end BUF;
 BFS : array(0..3) of BUF;
 task PI;
 task PO;
begin

  null;

end PI_PO;
```

```
task body BUF is
      Q    : CHARACTER;
      FULL : BOOLEAN := FALSE;
begin
   loop
      select
         when not FULL =>
         accept PUT(X: in CHARACTER) do
            Q := X;
            FULL := TRUE;
         end PUT;
      or
         when FULL =>
         accept GET(X: out CHARACTER) do
            X := Q;
            FULL := FALSE;
         end GET;
      end select;
   end loop;
end BUF;
```

```
task body PI is
      B : INTEGER;
      V : CHARACTER;
begin
   loop
      TEXT_IO.PUT("Buff? >");
      Ada.Integer_Text_IO.GET(B);
      TEXT_IO.PUT("Char? >");
      TEXT_IO.GET(V);
      TEXT_IO.PUT_LINE(" ");
      BFS(B).PUT(V);
   end loop;
end PI;
```

```
task body PO is
   I : INTEGER := 0;
   V : CHARACTER;
begin
   loop
      for I in 0..3 loop
         BFS(I).GET(V);
         Ada.Integer_Text_IO.PUT(I);
         TEXT_IO.PUT(":");
         TEXT_IO.PUT(V);
         TEXT_IO.PUT_LINE(" ");
      end loop;
   end loop;
end PO;
```

ESERCIZIO 3 (TEMA D'ESAME)

Si consideri un conto corrente bancario a cui si può accedere in modo concorrente per effettuare operazioni di versamento e di prelievo.
Due possibilità:

a) È definito un importo massimo prelevabile, e l' accesso al conto corrente da parte di un utente che vuole effettuare un prelievo è possibile solo se, nel conto corrente è presente un importo tale da rendere possibile il prelievo della cifra massima senza “andare in rosso”.

b) L' accesso all' utente che vuole effettuare un prelievo è permesso se l' importo che si intende prelevare è inferiore all'attuale disponibilità.

Si dica quali delle soluzioni applicative (a) o (b) sono realizzabili con semplici programmi in Ada o in Java, motivando sinteticamente la risposta nel caso negativo.

Soluzione JAVA

```
class ContoCor{
  ………
  private int disponibile;
  private int soglia;
  ………
  ContoCor() { ……… disponibile=0; soglia=2000; ……… }
  ………
  public synchronized void deposita(int importo) {
         disponibile += importo;
         notifyAll();
  }
  // Soluzione a)
  public synchronized void ritira(int importo) {
                                        // Hp. importo è già verificato essere <=soglia
      while ( !(disponibile >= soglia) )
        try { wait(); }
        catch (InterruptedException e){System.out.println(e.toString());}
      disponibile -= importo;
  }
  ………
//Soluzione b)
//public synchronized void ritira(int importo) {
//         // Hp. importo è già verificato essere <=soglia
//  while ( !(disponibile >= importo) )
//    try { wait(); }
//    catch (InterruptedException e){System.out.println(e.toString());}
//  disponibile -= importo;
//}
}
```

Soluzione ADA

a)

```
task ContoCor is
     entry deposita(importo: in INTEGER);
     entry ritira(importo: in INTEGER);
end ContoCor;
task body ContoCor is
     disponibile: INTEGER := 0, soglia := 2000;
     loop
          select
               when (disponibile >= soglia) =>
                 accept ritira(importo: in INTEGER) do
                    disponibile := disponibile – importo;
                 end ritira;
          or
               when (true) =>
                 accept deposita(importo: in INTEGER) do
                    disponibile := disponibile + importo;
                 end deposita;
          end select;
     end loop;
end ContoCor;
```

b)

In Ada non esiste una semplice variante del caso (b) perché nella clausola *when* di una *select* non è possibile valutare il parametro della *entry* citata nella corrispondente *accept*; il valore del parametro risulta noto solo DOPO l'esecuzione della *accept* e quindi non può essere utilizzato per decidere l'esecuzione dell'*accept* stessa.

QUALCHE DOMANDA

1. Quale politica seguono ADA e JAVA per la gestione dei task sospesi ?
2. E’ possibile imporre in Java di usare la stessa politica di ADA ?

1.R Ada segue una politica FIFO mentre Java lascia la scelta non deterministica

2.R Si potrebbe definire un oggetto CODA (di tipo FIFO) da associare al monitor. Prima di ogni `wait`, viene inserito un riferimento al task che viene sospeso. Dopo ogni risveglio (`NotifyAll()`) ogni task verifica se esso è il primo task di coda. Se si, si toglie dalla coda e procede, altrimenti si rimette in `wait`

ESERCIZIO 4

Un impianto può portarsi dallo stato di funzionamento normale N in uno stato di gestione di malfunzionamento M. Entrato in tale stato, entro 5 s deve essere aperta una valvola di scarico. Se non si apre, l'impianto passa in uno stato di fermo (F). Se la valvola viene aperta, essa rimane in tale stato per un tempo non inferiore a 20 s e non superiore a 30 s, poi l'impianto ritorna nello stato N.

```
public class Impianto extends Thread {
     private int stato; // 1 = N, -1 = M, 0 = F
     private Valvola valvola;
     public Impianto() { stato=1; }
     public void run(){
       while (stato!= 0){
         System.out.println("sto lavorando!");
         while (Math.random()<.8){ // affidabilita`
            System.out.println("tutto bene!");
         }
         valvola = new Valvola("Valvola Di Sfogo");
         stato=-1;
         valvola.start();
         try{
             System.out.println("Aspetto la valvola di sfogo...");
             synchronized(valvola)
             {
             // Cedo il lock sulla valvola, che quindi può cambiare stato e poi aspetto massimo 5
             // secondi. Se entro questo periodo non c'è stata alcuna notify (cioè il thread valvola è
             // ancora in sleep mode), mi sveglio. La parola chiave synchronized indica che questo
             // pezzo di codice è synchronized rispetto all'oggettovalvola. Questo è necessario
             // perché ogni wait o notify (vedi Valvola.java) deve trovarsiall'interno di un metodo
             // synchronized oppureall'interno di un blocco synchronized come in questo caso

                 valvola.wait(5000);
             }
         } catch (InterruptedExceptionie){ ie.printStackTrace(); }
         System.out.println("Check valvola aperta!");
         // Controllo se è cambiato lo stato della valvola
         if (valvola.getStato() == 0) {
           System.out.println("La valvola non si e' aperta!");
           stato=0;
         } else try {
                     System.out.println("La valvola si e' aperta”
                        +”, aspetto che finisca di sfogarsi...");
                     valvola.join();// Aspetto che la valvola abbia finito
                } catch(InterruptedException ie){
                                         ie.printStackTrace();
                  }
        } // end while
     } //end run()
```

```
    public static void main(String[] args){
        Impianto imp = new Impianto();
        imp.start();
    }
}



public class Valvola extends Thread{
    private String name;
    private int stato; // 0 chiuso, 1 aperto
    public Valvola(String name){
        this.name=name;
        this.stato= 0;
    }

    public void run(){
        long t = 0;
        System.out.println("sono la "+name);
        try {
            t = (long)(5500*Math.random());
            sleep(t);
        } catch (InterruptedException ie) {ie.printStackTrace();}
        System.out.println(name+ ": c'ho messo" + t + "ms");
        // Appena sveglio, cambio il mio stato e poi notifico il cambiamento usando una notify
        stato= 1;
        synchronized(this) {notify();}
        try {
            System.out.println("Ora mi sfogo...");
            t = (long)(20000+10000*Math.random());
            sleep(t);
        } catch (InterruptedException ie) {ie.printStackTrace();}
        System.out.println(name+": sfogata per " + t + "ms");
        stato= 0;
    }

    public int getStato() { return stato; }
}
```

```
procedure MAIN is

 task IMPIANTO;

 task VALVOLA is
  entry APRI;
 end VALVOLA;

task body IMPIANTO is

 type STATO_T is ( N , M , F );

 STATO : STATO_T;
 begin

  loop
     ......
     determinazione dello stato [ (M)alfunzioname
     ......
   if STATO = M then

   select
      VALVOLA.APRI;
      STATO := N;
   or
      delay 5.0;
      -- la valvola non ha risposto entro 5 secondi
      STATO := F;
   end select;

   end if;
    if STATO = F then
       ......
       azioni da eseguire in stato di fermo
       ......
    end if;
   end loop;
  end IMPIANTO;
```

```
task body VALVOLA is
    begin
     loop
      select
          accept APRI do

            -- apri valvola

            delay ( tempo variabile tra 20 e 30 secondi )

          end APRI;
        else
           -- azione eseguita nel caso non vi sia nessuna chiamata pendente
           -- per APRI:
           -- ritardo variabile durante il quale non può essere accettata
           -- alcuna chiamata per APRI. Se questo tempo è maggiore di
           -- 5 secondi e la sospensione ha inizio subito prima della chiamata
           -- ad APRI da parte dell'impianto, questo dopo 5 secondi entrerà
           -- nello stato di malfunzionamento.

           delay (tempo scelto casualmente tra 0 e N secondi)

           -- N è un parametro che caratterizza la risposta della valvola:
           -- quanto più N supera i 5 secondi maggiore sarà la probabilità che
           -- la valvola non risponda entro l'intervallo di tempo richiesto.
        end select;

       end loop;

      end VALVOLA;
```

JAVA GENERICS (JVM 5.0)

Sources:
Brian Goetz, Java theory and practice: Generics gotchas, IBM e-library
Oscar Nierstrasz, Java Lectures at Universitat Bern
Gilad Bracha, *Generics in the Java Programming Language, 2004*

Generic types (or generics) bear a superficial resemblance to templates in C++, both in their syntax and in their expected use cases (such as container classes).
But the similarity is only skin-deep -- generics in the Java language are implemented almost entirely in the compiler, which performs type checking and type inference, and then generates ordinary, non-generic bytecodes.

This implementation technique, called *erasure* (where the compiler uses the generic type information to ensure type safety, but then erases it before generating the bytecode),

Why do I need generics?
How do I use generics?
Can I subtype a generic type?

- Generics allow you to abstract over types.
  The most common examples are container types,
  the **Collection** hierarchy.
- Generics are a big step forward in the type-safety of the Java language, but the design of the generics facility, and the *generification* of the class library, were not without compromises. Extending the virtual machine instruction set to support generics was deemed unacceptable, because that might make it prohibitively difficult for JVM vendors to update their JVM.
- Accordingly, the approach of ***"erasure"***, which could be implemented entirely in the compiler, was adopted.
- Similarly, in *generifying* the Java class libraries, the desire to maintain backward compatibility placed many constraints on how the class libraries could be *generified*, resulting in some confusing and frustrating constructions. These are not problems with generics per se, but with the practicality of language evolution and compatibility.

Motivating Example – Old Style

The compiler can only guarantee that an Object is returned by the iterator. To ensure the assignment to a variable of type Stone is type safe we need the cast.
The cast introduces clutter and also increases the risk of error, since the programmer may be mistaken.

```
List stones = new LinkedList();
stones.add(new Stone(RED));
stones.add(new Stone(GREEN));
stones.add(new Stone(RED));
Stone first = (Stone) stones.get(0);
```

The cast is annoying but essential!

```
public int countStones(Color color) {
    int tally = 0;
    Iterator it = stones.iterator();
    while (it.hasNext()) {
        Stone stone = (Stone) it.next();
        if (stone.getColor() == color)
           tally++;
    }
    return tally;
}
```

Motivating Example – New Style using Generics

What if the programmer could actually express their intent and mark a list as being restricted to contain a particular data type - in this case a Stone.

We say that the list is a generic interface that takes a type parameter.

The compiler can now check the correctness of the program at compile-time. The declaration tells us about the List that holds true whenever the list is used, and the compiler will guarantee it.

In contrast the cast tells us what the programmer thinks is true at a single point in the code.

Improved readability in large programs and robustness.

List is a generic interface that takes a type as a parameter.

```java
List<Stone> stones = new
LinkedList<Stone>();
stones.add(new Stone(RED));
stones.add(new Stone(GREEN));
stones.add(new Stone(RED));
Stone first = /*no cast*/ stones.get(0);
```

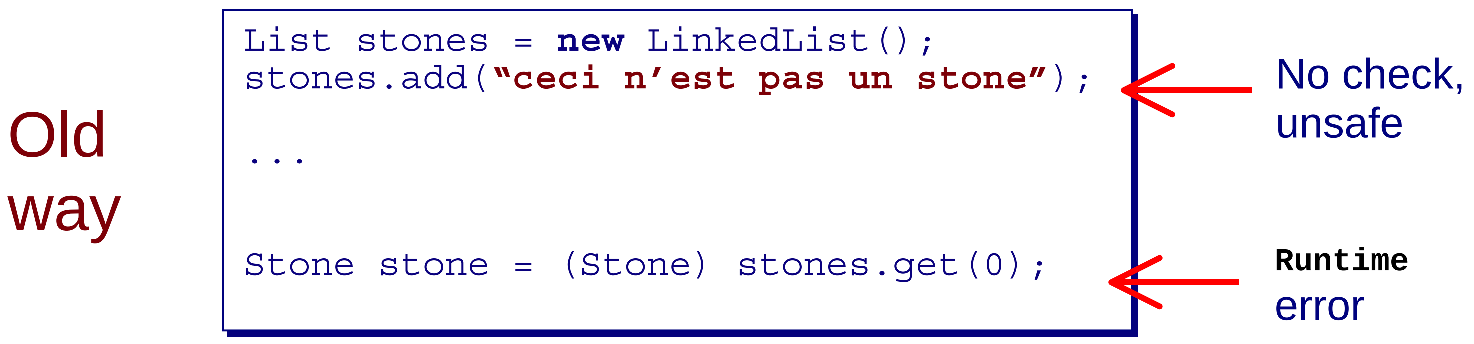
```java
public int countStones(Color color) {
    int tally = 0;
    /*no temporary*/
    for (Stone stone : stones) {
        /*no temporary, no cast*/
        if (stone.getColor() == color)
            tally++;
    }
    return tally;
}
```

Compile Time vs. Runtime Safety

When you take an element out of a Collection, you must cast it to the type of element that is stored in the collection. Besides being inconvenient, this is unsafe. The compiler does not check that your cast is the same as the collection's type, so the cast can fail at run time.

Generics provides a way for you to communicate the type of a collection to the compiler, so that it can be checked. Once the compiler knows the element type of the collection, the compiler can check that you have used the collection consistently and can insert the correct casts on values being taken out of the collection.

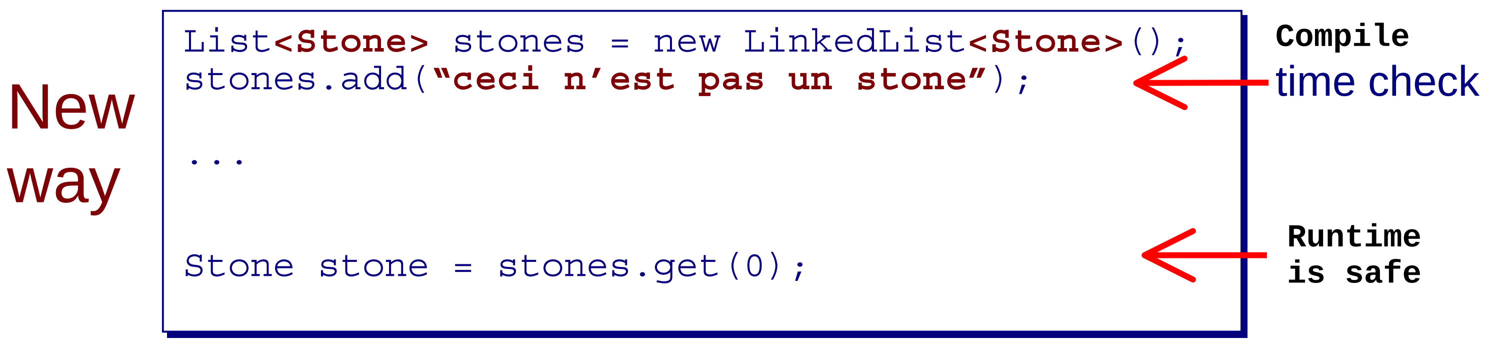
More importantly, we have moved part of the specification of the method from a comment to its signature, so the compiler can verify at compile time that the type constraints are not violated at run time. Because the program compiles without warnings, we can state with certainty that it will not throw a ClassCastException at run time. The net effect of using generics, especially in large programs, is improved readability and robustness.



```
List stones = new LinkedList();
stones.add("ceci n'est pas un stone");

...

Stone stone = (Stone) stones.get(0);
```



```
List<Stone> stones = new LinkedList<Stone>();
stones.add("ceci n'est pas un stone");

...

Stone stone = stones.get(0);
```

Stack Example

```
public interface StackInterface {
   public boolean isEmpty();
   public int size();
   public void push(Object item);
   public Object top();
   public void pop();
}
```

Old way

```
public interface StackInterface<E> {
   public boolean isEmpty();
   public int size();
   public void push(E item);
   public E top();
   public void pop();
}
```

New way: we define a generic interface that takes a type parameter

Linked Stack Example

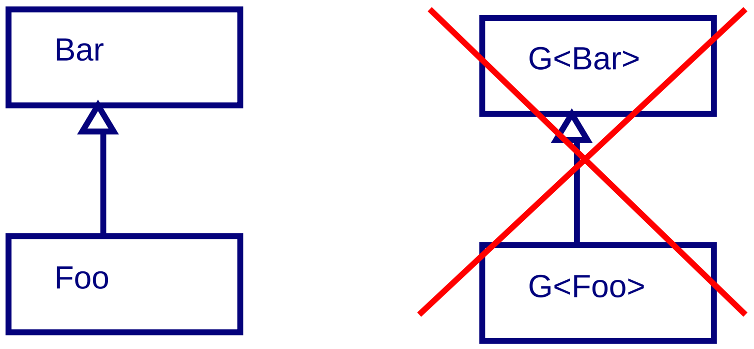
```
public class LinkStack<E> implements StackInterface<E> {
…
    public class Cell {
        public E item;
        public Cell next;
        public Cell(E item, Cell next) {
            this.item = item;
            this.next = next;
        }
    }
…
    public E top() {
        assert !this.isEmpty();
        return _top.item;
    }
```

Creating a Stack of Integers

When a generic declaration is invoked, the actual type parameters are substituted for the formal type paramters.

```
Stack<Integer> myStack = new LinkedStack<Integer>();
myStack.push(42); // autoboxing
```

Generics and Subtyping



G is some generic type declaration

```
List<String> ls = new Array<String>();
List<Object> lo = ls;
```

Compile time error as it is not type safe

Here we have aliased ls and lo. By accessing a list of String ls through the Alias lo we can insert arbitrary objects into it. As a result, ls does not hold just strings anymore and when we try to get something out of it we get a rude surprise.

Wildcards

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (k=0; k<c.size();k++){
        System.out.println(i.next());
    }
}
```

We want a method that prints us all the elements of a collection

```
void printCollection(Collection<Object> c) {
    for (Object e: c){
        System.out.println(e);
    }
}
```

Here is a naïve attempt at writing it using generics

The problem is that the new version is less useful than the old one.
The old one could be called for any kind of collection, the new one for Collection<Object> only, which is not a supertype of any kind of collection.

What is the supertype of all kinds of collections?

```
Collection<?>
```

"collection of unknown" is a collection whose elementtype matches anything - **a wildcard type**

```
void printCollection(Collection<?> c) {
    for (Object e: c){
        System.out.println(e);
    }
}
```

Now we can call it with any type of collection.

Pitfalls with the Collection of Unknown

```
Collection<?> c = new ArrayList<String>();
c.add(new Object());
```

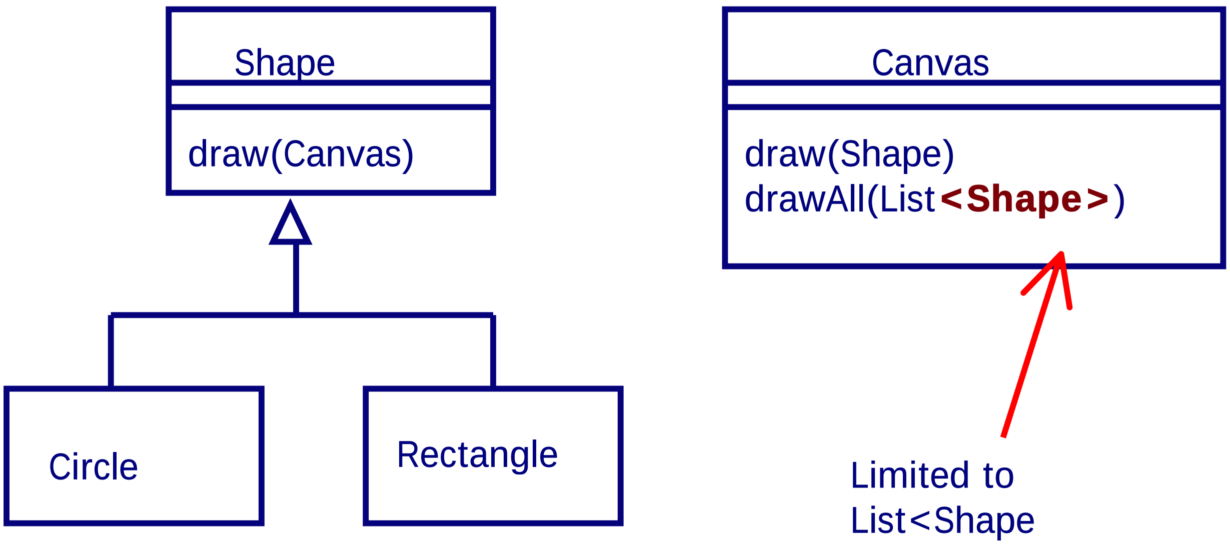
Compile time error

```
Collection<?> c = new ArrayList<String>();
…
Object myObject = c.get();
```

Since we do not know what the element type of c stands for, we cannot add Objects.
However we can get objects as we always know the type is of type Object.

Bounded Wildcards

Consider a simple drawing application to draw shapes (circles, rectangles,...)
Any drawing will typically contain a number of shapes. Assuming that
They are represented in a list, it would be convenient to have a method
In canvas that draws them all.
However drawAll can only be called for list of Shape not list of circles.
We want the method to accept a list of any shape.



A Method that accepts a List of any kind of Shape…

```
public void drawAll(List<? extends Shape>) {…}
```

a bounded wildcard

Shape is the ***upper bound*** of the wildcard

Wildcards are designed to support flexible subtyping, which is
what we're trying to express here.

Generic methods allow type parameters to be used to express dependencies among the types of one or more arguments to a method and/or its return type. If there isn't such a dependency, a generic method should not be used.

It is possible to use both generic methods and wildcards in tandem.
Here is the method Collections.copy():

```
class Collections {
   public static <T>
          void copy(List<T> dest, List<? extends T>src){
          ...
   }
}
```

Note the dependency between the types of the two parameters. Any object copied from the source list, src, must be assignable to the element type T of the destination list, dst.
So the element type of src can be any subtype of T - we don't care which.
The signature of copy expresses the dependency using a type parameter, but uses a wildcard for the element type of the second parameter.

We could have written the signature for this method another way, without using wildcards at all:

```
    class Collections {
        public static <T, S extends T>
               void copy(List<T> dest, List<S> src){...}
    }
```

This is fine, but while the first type parameter is used both in the type of dst and in the bound of the second type parameter, S, S itself is only used once, in the type of src - nothing else depends on it. This is a sign that we can replace S with a wildcard.

Using wildcards is clearer and more concise than declaring explicit type parameters, and should therefore be preferred whenever possible.

Wildcards also have the advantage that they can be used outside of method signatures, as the types of fields, local variables and arrays.