ALUR PROGRAM

A.PERCABANGAN

- If
- if else
- switch

1. IF

Percabangan IF di Java menyatakan bahwa suatu statement (pernyataan) akan dieksekusi bila memenuhi syarat/kondisi tertentu

If *Ekspresi Statement*

```
public class If01
{
  public static void main (String args[])
  {
    int anInt = 0;
    if (anInt==0)
      {
      System.out.println("Variabel anInt bernilai Nol");
      }
  }
}
```

2. IF-ELSE

Percabangan ini untuk memilih salah satu dari dua kemungkinan kemunculan.

if *(Condition)*
Statement1
else
Statement2

```
public class IfElse01
{
  public static void main (String args[])
  {
    int anInt = 1;
    System.out.println("Nilai Variabel anInt : "+anInt);
    if (anInt==0){
      System.out.println("Variabel anInt bernilai Nol");
    }
    else{
      System.out.println("Variabel anInt tidak bernilai Nol");
    }
  }
}
```

Hasil :
Nilai Variabel anInt : 1
Variabel anInt tidak bernilai Nol

3. SWITCH

Percabangan *Switch* dimaksudkan untuk menangani banyak kemungkinan kemunculan :

switch *(ekspresi* {

case *Constant1:*

statementlist1

case *Constant1:*

statementlist1

default:

defaultstatementlist

}

```
public class Switch01 {
    public static void main (String args[]) {
        int a;
        a=5;
        switch(a) {
            case 0 : System.out.println("Nol");break;
            case 1 : System.out.println("Satu");break;
            case 2 : System.out.println("Dua");break;
            case 3 : System.out.println("Tiga");break;
            case 4 : System.out.println("Empat");break;
            case 5 : System.out.println("Lima");break;
            case 6 : System.out.println("Enam");break;
            case 7 : System.out.println("Tujuh");break;
            case 8 : System.out.println("Delapan");break;
            case 9 : System.out.println("Sembilan");break;
            default: System.out.println("Bukan Karakter Digit");
        }
    }
}
```

Hasil :
Lima

B.PERULANGAN

- for
- while
- do-while

1. FOR

Perulangan **for** menyediakan sarana mengulang kode dalam jumlah tertentu, bersifat terstruktur untuk mengulangi kode sampai tercapai batas tertentu.

for (*Initialization*Expression;*Loop*Condition;*Step*Expression)
Statement

- InitializationExpression :
  Digunakan untuk inisialisasi variabel kendali pengulangan
- LoopCondition :
  Membandingkan variabel kendali pengulangan dengan suatu nilai batas
- StepExpression :
  Menspesifikasikan cara variabel kendali dimodifikasi sebelum iterasi berikutnya dari perulangan.

Nested for

```
public class NestedFor {
   public static void main(String args[]){
      for(int i=1;i<=10;i++){
         for(int j=i;j<=10;j++){
            System.out.print(j);}
            System.out.println();

      }
   }
}
```

Hasil :

```
12345678910
2345678910
345678910
45678910
5678910
678910
78910
8910
910
10
```

VARIASI dari perulangan for :

```
public class VarianFor {
   public static void main(String args[]){
      boolean finish=false;

      int i=1;
      for (;!finish;){
         System.out.println("i= "+i);
         if(i==10) finish=true;
         i++;
      }
   }
}
```

```
i= 1
i= 2
i= 3
i= 4
i= 5
i= 6
i= 7
i= 8
i= 9
i= 10
```

2. WHILE

while (*Loop*Condition)
Statement

- Jika LoopCondition dievaluasi true, maka Statement dieksekusi dan proses berlanjut diulangi.
- Jika LoopCondition sejak semula dievaluasi false, maka statement tak pernah dieksekusi.

```
public class WhileLoop {
   public static void main(String args[]){
      int i=1;
      while(i<=10){
         int j=1;
         System.out.println("i = "+i);
         i++;
      }
   }
}
```

Hasil :

```
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
```

```
i = 8
i = 9
i = 10
```

3. DO-WHILE

Perulangan do-while serupa dengan perulangan while, hanya pemeriksaan kondisi yang dilakukan adalah setelah statement.

do
  Statement
while (LoopCondition)

```
public class DoWhileLoop {
    public static void main(String args[]){
        int i=1;
        do{
            System.out.println("i= "+i);
            i++;
        }while(i<11);
    }
}
```

Hasil :

```
i= 1
i= 2
i= 3
i= 4
i= 5
i= 6
i= 7
i= 8
i= 9
i= 10
```

CLASS Dan OBJECT

Beberapa orang pada awalnya, biasanya tidak memperhatikan perbedaan antara class dan object, mereka mencampuradukkan kedua istilah tersebut.
Mari kita simak kode berikut :

```
/*
Di sini kita mendefinisikan sebuah class bernama NiceGuy.
Simpan sebagai file NiceGuy.java dan compile file
Tersebut. Anda akan mendapatkan file bernama
NiceGuy.class
*/

public class NiceGuy {
        private String name;

        public NiceGuy(String name)
      {
         this.name=name;
            System.out.println("Instantiation of NiceGuy named " +name);
        }
        public void sayHello()
      {
         System.out.println("Hello Object Oriented World...!!!" ) ;
        }
        public void sayHelloOutLoud()
      {
         System.out.println("HELLLOOOO OBJECT ORIENTED WORLD!!!") ;
        }
        public String getName()
      {
         return name;
        }
}
```

Definisi class NiceGuy

```
/*
Di sini, kita membuat sebuah java application yang
bernama OurFirstCode.
Simpan source ini dalam file bernama OurFirstCode.java,
di direktori yang sama dengan tempat anda menyimpan
NiceGuy.java.
Kemudian compile file tersebut, Anda akan mendapatkan file OurFirstCode.class.
Lalu jalankan aplikasi ini dengan mengetikan java OurFirstCode pada command line
*/


public class OurFirstCode {
        public static void main(String args[])
      {
         NiceGuy ng= new NiceGuy ("ButtHead");
           ng.sayHello();
           ng.sayHelloOutLoud();

           //he's so cute…,who's he???
           String NiceGuyName = ng.getName();
           System.out.println("OH…!!!! HE'S "+NiceGuyName.toUpperCase());
       }
}
```

Aplikasi Java

Diatas merupakan contoh pendefinisian sebuah class bernama NiceGuy pada Definisi class NiceGuy. Pada Aplikasi Java merupakan kode dari sebuah aplikasi java yang bernama OurFirstCode. Pada aplikasi tersebut kita menggunakan class NiceGuy untuk membuet objek bertipe NiceGuy.

Hasil :

```
HELLLOOOO OBJECT ORIENTED WORLD!!!
OH…!!!! HE'S BUTTHEAD
```

PENDEFINISIAN CLASS

Sintaks dalam mendefinisikan class adalah sebagai berikut :

[modifier-modifier] **class** *NamaClass* [extends parentclass]
[implements interface] *{*
[deklarasi field-field]
[definisi method-method]
}

Yang tertera didalam kurung siku bersifat optional. Dengan demikian, definisi minimal dari sebuah class bisa jadi seperti berikut :

```
class Useless {

//….mmm….

}
```

Tidak ada yang bisa kita harapkan dari objek diatas. Contoh yang lebih baik adalah Definisi class NiceGuy. Disana kita mendeklarasikan 1 field dan 3 method. Pada baris 8 kita mendeklarasikan field yang bernama *name* yang bertipe string.
Secara umum sintaks untuk mendeklarasikan **Field** adalah :

type *namafield;*

Tipe dari field bisa primitif (seperti int, boolean, float, dan sebagainya), dan bisa juga Object (seperti String, Vector, Hashtable, dan sebagainya).

Method-method yang dimiliki class NiceGuy adalah method sayHello(), method sayHelloOutLoud(), dan method getName().
Secara umum sintaks dalam pendefinisian sebuah **Method** adalah :

[modifier-modifier] *return-type* **namaMethod**
([parameter1, [parameter2],… , [parameter N]) *{*
[statement-statement];
}

Tipe Return bisa void, tipe data primitif, atau tipe data object.
Method dengan tipe return non-void harus mencantumkan statement return <something> pada akhir deklarasi method. Kita juga dapat menggunakan kata return pada method void untuk keluar dari method tersebut.

Bagian penting lain dari definisi class adalah *constructor*.. **Constructor** digunakan pada saat penciptaan objek dari sebuah class. Pendeklarasian constructor mirip dengan pendeklarasian method, dengan satu pengecualian bahwa *constructor tidak mencantumkan tipe return.*

[modifier-modifier] **NamaConstructor** ([parameter 1], [parameter 2], …
, [parameter N]) *{*
[statement-statement];
}

Hal lain yang perlu dicatat tentang constructor adalah, ***nama constructor harus sama dengan classnya***. Constructor tanpa parameter disebut *default constructor*. Kalau kita sama sekali tidak mendeklarasikan constructor, compiler secara otomatis akan membuatkan sebuah default constructor.

Pada Apilkasi Java, aplikasi OurFirstCode memanfaatkan class NiceGuy untuk menciptakan sebuah objek bertipe NiceGuy (NiceGuy adalah suatu class) di memori. Setelah itu,

Reference **ng** dapat digunakan untuk memanggil method-method (mengirimkan pesan kepadanya) atau mengakses field-field dari objek yang bersangkutan.

- Reference adalah seperti alamat rumah
- sedangkan Objek adalah rumahnya.

dengan mengetahui alamat, kita bisa mencapai rumah yang dimaksud.

```
public class OurFirstCode {
        public static void main(String args[])
      {
         NiceGuy ng= new NiceGuy ("ButtHead");
           ng.sayHello();
           ng.sayHelloOutLoud();

           //he's so cute…,who's he???
           String NiceGuyName = ng.getName();
           System.out.println("OH…!!!! HE'S "+NiceGuyName.toUpperCase());
        }
}
```

INHERITANCE

Salah satu topik penting dalam OOP adalah inheritance (pewarisan sifat).

- Pengembangan software dapat bekerja lebih efisien dan lebih cepat.
- Dapat menggunakan definisi class yang pernah dibuat sebelumnya untuk membuat class-class lain yang menyerupai class tersebut.

```
public class KattWorld {
                public static void main (String args[]) {
                        Katt k = new Katt();
                        k.speak();
                        Anggora a = new Anggora();
                        a.speak(); a.jump();
                        Siam s = new Siam();
                        s.speak();
                        }
                }
        class Katt
      {
                public Katt()
            {
               System.out.println("Katt Constructor");
                }
                public void speak()
            {
               System.out.println("Miaww…");
                }
        }
        class Anggora extends Katt
      {
         public void jump()
         {
            System.out.println("Crash…Boom…");
          }
        }
        class Siam extends Katt
      {
         public Siam()
         {
            System.out.println("Siam Constructor");
          }
          public void speak()
         {
            System.out.println("Mmurrrr…mmurrrr…");
          }
}
```

Hasil :

```
Katt Constructor
Miaww…
Katt Constructor
Miaww…
Crash…Boom…
Katt Constructor
Siam Constructor
Mmurrrr…mmurrrr…
```

Hal pertama yang harus diperhatikan dari contoh di atas adalah cara melakukan inheritance. Contoh diatas mempunyai satu class bernama Katt. Kemudian dibut class lainnya yang mempunyai sifat seperti Katt, tetapi memiliki keunikan sendiri. Contohnya adalah class Anggora, kita dapat membuat subclass dari Katt.
"… Katt menjadi parentclass (baseclass atau superclass) dari class Anggora …"

Secara umum kita dapat mendefinisikan suatu class sebagai turunan dari base class dengan cara :

class *NamaClass* **extends** *BaseClass {*
[statement-statement];
}

METHOD OVERIDING

Kita membuat varian baru lain dari Katt dengan cara mengeong Murr….Murr…Murr…. Kita namai varian itu Siam. Kita definisikan Siam sebagai subclass dari Katt, tapi kini kita melakukan metode overiding, kita *mendefinisikan ulang method* ***speak()*** *didalam definisi class Siam*. Jika dijalankan akan menghasilkan output:

```
Katt constructor
Miaww…
Katt constructor
Miaww…
Crash…Boom…
Katt constructor
Siam constructor
Mmurrr…mmurrr…
```

Baris 6 dan 7 dimunculkan sewaktu berlangsung konstruksi/instantiasi objek bertipe Siam. ***Default constructor Katt dijalankan terlebih dahulu.*** Ini menunjukkan bahwa default, kecuali kita minta yang lain, dari baseclass secara otomatis dijalankan terlebih dahulu sebelum constructor dari class yang bersangkutan.

THIS

This sebenarnya adalah sebuah variable read-only (tidak dapat diubah nilainya).
Dengan variable ini, kita ***mendapatkan reference/pointer menuju objek terkini***.
Bayangkan aplikasi anda sedang berjalan, pada saat tertentu yang dijalankan oleh komputer anda adalah method aMethod() milik object anObject. Dari method aMethod() anda membutuhkan reference ke objek terkini yaitu object anObject itu sendiri. Untuk itulah kita menggunakan variabel this dalam method aMethod() milik object anObject.

```
public class BeavisAndButtheadStory {
   public static void main (String args[])
   {
        DivineBeing  beavis = new DivineBeing ("Beavis");
        DivineBeing  butthead = new DivineBeing ("Butthead");
        //guess what's gonna happen next…
        beavis.messWith (butthead);
        butthead.makeRevenge();
   }
}
```

```
class DivineBeing
{
   private String name;
   private DivineBeing baddDivineBeing;
   public DivineBeing (String name)
   {
      this.name = name ;
   }
   public void messWith (DivineBeing anotherDivineBeing)
   {
      anotherDivineBeing.kapow(this);
   }
   public void kapow (DivineBeing baddDivineBeing)
   {
      System.out.println(baddDivineBeing.getName()+"!!!!Damn You!!");
        //forgive but not forget…
        this.baddDivineBeing = baddDivineBeing;
   }
   public void makeRevenge()
   {
      if(baddDivineBeing != null)
      {
         baddDivineBeing.kapow(this);
        }
   }
   public String getName()
   {
      return name;
   }
}
```

A. **beavis.messWith (butthead);**
   anotherDivineBeing = butthead
   this = beavis
   anotherDivineBeing.kapow(this) = butthead.kapow(beavis)
   baddDivineBeing = beavis
   beavis.name = “Beavis”
   baddDivinebeing=beavis
B. **butthead.makeRevenge();**
   baddDivineBeing.kapow(this) = beavis.kapow(Butthead)
   baddDivineBeing = butthead
   butthead.name = “Butthead”

Hasil :
Beavis!!!!Damn You!!
Butthead!!!!Damn You!!

ABSTRACT

Abstract method adalah ***method yang belum mempunyai implementasi***.
Kita dapat menyatakan suatu method abstract dengan membubuhkan keyword abstract pada deklarasi method tersebut.
Secara umum sintaks dari pendeklarasian method abstract :

abstract return-type **namaMethod** ([daftar-parameter]);

```
public class ExplainAbstract
{
   public static void main (String args[])
   {
      Penyanyi joshua = new Penyanyi();
      joshua.berkesenian();
      Pemrogram raka = new Pemrogram();
      raka.berkesenian();
   }
}

abstract class Seniman
{
   public abstract void berkesenian();
   public void tidur()
   {
      System.out.println ("ZZZZ…");
   }
}

class  Pemrogram extends Seniman
{
   public void berkesenian()
   {
      System.out.println("tap…tap…click-click…tap…PLAK ! Click…DOR !!!");
   }
   public void tidur()
   {
      System.out.println("Buzzz…ngingggg….");
   }
}

class Penyanyi extends Seniman
{
   public void berkesenian()
   {
      System.out.println("Tralala-Trilili…");
   }
}
```

Hasil :
Tralala-Trilili…
tap…tap…click-click…tap…PLAK ! Click…DOR !!!

STATIC METHOD

Pada contoh-contoh sebelumnya kita harus menginstantiasi/membuat objek terlebih dahulu sebelum dapat menggunakan method-method atau mengakses field-field pada class yang bersangkutan. Tapi dengan mendefinisikan suatu field atau method sebagai static, kita dapat saja mengakses field method tersebut tanpa harus melakukan instantiasi terlebih dahulu.

```
public class MyGeomUtil {
        int iAmNotPopular = 13;     static int iAmCelebrity = 7;

    public static double luasSegiempat (float length, float width) {
        return length * width;
        }
    public static double luasSegitiga (float alas, float tinggi) {
        return 0.5 * alas * tinggi;
        }
    public static double luasSegitiga (float A, float B, float gamma) {
        //fungsi sin (double angel) menerima masukan sudut
        //dalam radian angle not angel, you prevert..!
        return 0.5 * A * B * Math.sin(gamma / Math.PI);
        }
    }
```

FINAL

Dengan menambahkan modifier final pada deklarasi sebuah method, kita menetapkan bahwa **method** pada class tersebut ***tidak bisa ditimpa pada subclass*** yang kelak mungkin akan dibuat. Salah satu alasan untuk membubuhkan final pada method di sebuah base class adalah karena method itu begitu fundamental bagi kerja instance class tersebut, sehingga jika di implementasikan secara berbeda oleh subclassnya (yang bisa saja ditulis oleh programer yang salah) *berpotensi menyebabkan kerja instance itu tidak benar.*

Bentuk umum final method :

[modifier-modifier] **final** *namaMethod*() {
//…mmmm…..
}

Final juga bisa diberlakukan bagi class, kita *tidak dapat membuat turunan dari class* final. Bentuk umum final class :

final class *NamaClass* **extends** *ParentClass* {
//…mmm…
}

INTERFACE

Pada Interface juga kita dapat mendeklarasikan method-method, field-field (dengan beberapa catatan tentunya), dan juga dapat membuat rantai inheritance seperti yang dilakukan pada class.

```
import java.awt.*;
import java.awt.event.*;

public class AustinPower extends Frame implements MouseListener
{
   Button behave;

   public static void main (String args[])
   {
      AustinPower spyWhoShaggedMe = new AustinPower();
        spyWhoShaggedMe.setSize(150,150);
        spyWhoShaggedMe.setVisible(true);
   }

   public AustinPower()
   {
      behave = new Button ("I Am a Button");
        behave.addMouseListener(this);
        add(behave);
   }

   public void mouseEntered(MouseEvent me)
   {
      behave.setLabel ("Click me…!!!");
   }

   public void mouseExited(MouseEvent me)
   {
      behave.setLabel ("Come to  me…!!!");
   }
   public void mousePressed(MouseEvent me)
   {
            behave.setLabel ("You've Pressed me…!!!");
   }
   public void mouseReleased(MouseEvent me)
   {
            behave.setLabel ("You've Released me…!!!");
   }
   public void mouseClicked(MouseEvent me)
   {
            behave.setLabel ("You've Clicked me…!!!");
   }
}
```

Sedangkan interface MouseListener seperti tertera dibawah ini :

```
package java.awt.event;

import java.util.EventListener;
/**
* The listener interface for receiving mouse event on a
* Component.
* @version 1.7 07/01/98
* @author Carl Quinn
*/
public interface MouseListener extends EventListener {
        public void mouseClicked (MouseEvent e);
        public void mousePressed (MouseEvent e);
        public void mouseReleased (MouseEvent e);
        public void mouseEntered (MouseEvent e);
        public void mouseExited (MouseEvent e);
        }
```

Pada contoh diatas, kita mengatakan bahwa ***AustinPower mengimplementasikan*** MouseListener. Secara umum interface dapat dideklarasikan sebagai berikut :

interface *NamaInterface* [extends parentinterface] *{*
    [deklarasi field-field (konstanta)]
    [deklarasi method-method]
}

Mendeklarasikan method dalam interface mirip dengan mendeklarasikan method abstract, dimana deklarasi method tidak diakhiri pasangan kurung kurawal, melainkan tanda titik koma :

return-type *namaMethod* ([daftar parameter]);

Ada satu catatan, field-field yang didefinisikan pada interface secara otomatis bersifat static dan final, yang berarti mereka hanya bertindak sebagai konstanta.

sedangkan secara umum, suatu **class** dibuat agar mengimplementasikan suatu interface dengan cara sebagai berikut :

class *NamaClass* [extends parentclass] [implements interface1,interface2,…]
{ [deklarasi field-field]
    [deklarasi method-method milik class ini]

    [deklarasi implementasi method-method yang terdefinisikan pada
        interface-interface yang di implementasi oleh class tersebut]
    }

Pada contoh diatas dengan mengimplementasikan MouseListener, class AustinPower dikatakan menandatangai suatu kontrak yang menyatakan bahwa ***ia mampu menunjukkan kelakuan-kelakuan seorang MouseListener*** (yaitu bereaksi terhadap event-event yang berhubungan dengan Mouse).
Semua method yang terdefinisi pada interface adalah betul-betul abstract, tidak ada implementasinya.
Salah satu konsekuensi dari totalitas abstraksi pada interface itu adalah ***semua method yang tertera pada interface harus di implementasikan oleh implementor.***

POLYMORPHISM

Polymorphism didefinisikan sebagai kemampuan beberapa objek bertipe sama bereaksi secara berbeda terhadap message yang sama.

```
public class KattParty {
        public static void main (String args[]) {
                Katt[] kandangKatt = new Katt[3];
                kandangKatt[0] = new Anggora();
                kandangKatt[1]  = new Siam();
                kandangKatt[2] = new Katt();
                for (int I = 0 ; I < 3 ; I++) {
                        kandangKatt[I].speak();
                        }
                }
        }
```

Hasil :

```
Katt Constructor
Katt Constructor
Siam Constructor
Katt Constructor
Miaww…
Mmurrrr…mmurrrr…
Miaww…
```

Disetiap iterasi pada loop diatas, pada dasarnya kita mengirimakan pesan yang sama ke objek-objek yang bertipe sama (yaitu Katt, sesuai dengan definisi array pada baris 3), speak(). Walaupun begitu, ternyata masing-masing bereaksi dengan caranya sendiri-sendiri.

Contoh lain dari polymorphism yaitu, misalkan kita membuat aplikasi client/server, dimana client mengirimkan objek-objek Katt (dan tentunya bisa juga apapun turunan Katt) ke server.
Pada kasus tersebut, server bersedia menerima Katt-Katt itu tanpa terlalu mempermasalahkan perbedaan spesifikasinya.
Untuk itu kode diserver bisa jadi seperti berikut:

```
…
Katt k = null;
while (true) {
        k = receive();
        }
…
```

Tentunya untuk kasus ini, tidaklah tepat jika, secara static kita mengasosiasikan k dengan class Siam (misal dengan pendeklarasian Siam k = null).
Pada kode diatas, pengasosiasian k dengan clas spesifik (Anggora, Siam atau lainnya) dilakukan pada saat run-time, yaitu ketika return Value dari pemanggilan receive() di-assign ke k. Itulah yang disebut *dynamic/late binding.*

OPERATOR INSTANCEOF

Bagaimana seandainya pihak server ingin mengetahui type spesifik dari objek Katt yang diterimanya ?

```
public class KattParty2
{
   public static void main (String args[])
   {
      Katt[] kandangKatt = new Katt[3];
        kandangKatt[0] = new Anggora();
        kandangKatt[1]         = new Siam();
        kandangKatt[2] = new Katt();
        for (int I = 0 ; I < 3 ; I++)
      {
          kandangKatt[I].speak();
            if (kandangKatt[I] instanceof Siam)
          {
             System.out.println("Wow, It's   a Siam…");
            } else if (kandangKatt[I] instanceof Anggora)
          {
             System.out.println("Look, they sent us Anggora…");
            }else
          {
             System.out.println("Well…It could be Lucifer," + " or Alleys or Regular Katt");
                  System.out.println("But one thing for sure…it's Katt" + " Praise the Lord");
            }
        }
   }
}
```

Hasil :

```
Katt Constructor
Katt Constructor
Siam Constructor
Katt Constructor
Miaww…
Look, they sent us Anggora…
Mmurrrr…mmurrrr…
Wow, It's   a Siam…
Miaww…
Well…It could be Lucifer, or Alleys or Regular Katt
But one thing for sure…it's Katt Praise the Lord
```

Operator yang kita pakai adalah instanceof.
A instanceof B, mengevaluasi **apakah objek A bertipe B** (B adalah nama class).
Evaluasi itu menghasilkan nilai ***true jika nama class dari objek A adalah B*** atau nama parent/grandparent/greatgrandparent /seterusnya dari class dari objek A adalah B.

Evaluasi juga akan bernilai benar ***jika B adalah nama interface dan A mengimplementasikan B***. Begitu pula jika parent/grandparent/greatgrandparent/seterusnya dari class objek A mengimplementasi B.

Singkat kata. jika A instanceof B menghasilkan true, berarti objek A dapat menunjukkan behavior yang disyaratkan oleh class atau interface B.

PACKAGE

Kita biasannya mengelompokkan class-class (dan juga interface-interface) yang terkait (karena jenisnya/fungsinya/alasan lain) dalam sebuah package. Mungkin ada baiknya jika class-class geometris yang kita buat dikumpulkan dalam sebuah package bernama com.raka.geoms

```
/*simpan sebagai Graphic.java */
package com.raka.geoms;
public abstract class Shape {
        …….
        }
```

Package---1

```
/*  Simpan sebagai Rectangle java */
package com.raka.geoms;
public class Rectangle extends Shape {
        ……………
        }
```

Package---2

```
/* Simpan sebagai Triangle.java */
package com.raka.geoms;
public class Triangle extends Shape {
        ……………
        }
```

Package---3

Beberapa keuntungan mengorganisasi class-class buatan kita dalam sebuah package adalah :

1. **Terhindar dari konflik penamaan**.
   Mungkin saja ada orang lain, dibelahan dunia lain, membuat class yang bernama Rectangle juga. Yang membedakan antara Rectangle kita dengan yang lain adalah *fully qualified name.* Fully qualified name dari class Rectangle kita adalah com.raka.geoms.Rectangle.
2. **Teratur**.
   Mendapatkan suatu class tertentu akan mudah dengan mengetahui nama package-nya.

Pada contoh diatas, ***masing-masing anggota dari package com.raka.geoms disimpan di file terpisah***, dan ***dideklarasikan sebagai public***.
Dengan demikian class/program lain (yang bukan anggota com.raka.geoms) dapat mengimport clas-class kita diatas.
Kita juga dapat mendefinisikan beberapa class sekaligus dalam sebuah file java :

```
/* Simpan sebagai NuclearWarHead.java */
package com.raka.lethal;

public class NuclearWarHead {
        ……………
        }
class Sarin     {
        …….
        }
class Uzi       {
        …….
        }
```

Jika kita kompilasi NuclearWarHead.java diatas, kita akan mendapatkan tiga file class, yaitu :

- NuclearWarHead.class
- Sarin.class
- Uzi.class.

Kesemuanya akan menjadi anggota dari package com.raka.lethal.

Akan tetapi *hanya NuclearWarHead yang bisa digunakan secara langsung dari luar package com.raka.lethal*, karena hanya NuclearWarHead yang dideklarasikan sebagai *public*, selebihnya dideklarasikan sebagai protected (defaultnya), sehingga hanya dapat digunakan oleh class-class yang tergabung didalam com.raka.lethal.

Sekarang untuk menggunakan class-class yang terdapat pada com.raka.geoms dari luar com.raka.geoms, kita mengenal kata kunci **import.** Seperti contoh dibawah ini :

```
package com.drawworks.threedee;
import com.raka.geoms.*;
……

public class Cube {
        ……
        Rectangle r = new Rectangle ();
        …..
        }
…..
```

Jika pemrogram com.drawworks.threedee bermaksud menggunakan class-class yang terdapat pada java.awt juga dimana terdapat juga class Rectangle, maka saat inilah digunaka fully qualified name :

```
package com.drawworks.threedee;
import com.raka.geoms.*;
import java.awt.*;
……

public class ThreeWorld extends Canvas {
        ……
        com.raka.geoms.Rectangle r = new com.raka.geoms.Rectangle ();
        …..
        }
…..
```

Fully Qualified Name

Package-package didalam harddisk dapat kita organisasikan dalam suatu struktur direktori. File-file class anggota com.raka.geoms terkumpul dalam struktur direktori. Anda boleh mengambil kesimpulan bahwa tiap-tiap suku kata pada nama package adalah nama sebuah direktori.

PENANGANAN EKSEPSI

Eksepsi adalah ***keadaan tidak normal*** yang muncul pada suatu bagian program pada saat dijalankan. Penanganan eksepsi pada java membawa pengelolaan kesalahan program saat dijalankan kedalam orientasi-objek. Eksepsi java adalah ***objek yang menjelaskan suatu keadaan eksepsi yang muncul pada suatu bagian program.***
Saat suatu keadaan eksepsi muncul, suatu ***objek exception dibuat dan dimasukkan ke dalam method yang menyebabkan eksep***si. *Method tersebut dapat dipilih untuk menangani eksepsi berdasarkan tipe tertentu*. Method ini juga menjaga agar tidak keluar terlalu dini melalui suatu eksepsi, dan memiliki suatu blok program yang dijalankan tepat sebelum suatu eksepsi menyebabkan metodenya kembali ke pemanggil.
Eksepsi dapat ***muncul tidak beraturan*** dalam suatu method, atau ***dapat juga dibuat secara manual*** dan nantinya melaporkan sejumlah keadaan kesalahan ke method yang memanggil.

DASAR-DASAR PENANGANAN EKSEPSI

Penanganan eksepsi pada java diatur dengan lima kata kunci :

1. try
2. catch
3. throw
4. throws
5. dan finally.

Pada dasarnya *try* digunakan untuk mengeksekusi suatu bagian program, dan jika muncul kesalahan, sistem akan melakukan *throw* suatu eksepsi yang dapat anda *catch* berdasarkan tipe eksepsinya, atau yang anda berikan *finally* dengan penangan default.

```
try {
        //  Block of Code
        }
catch (ExceptionType1 e) {
        //      Exception Handler for ExceptionType1
        }       catch (ExceptionType2 e) {
                // Exception Handler for ExceptionTYpe2
                throw (e); // re-throw the Exception…
                }
finally {
                }
```

TIPE EKSEPSI

Dipuncak hirarki class eksepsi terdapat satu class yang disebut ***throwable***. Class ini digunakan untuk merepresentasikan semua keadaan ekasepsi. Setiap *ExceptionType* pada bentuk umum diatas adalah subclass dari *throwable*.
Dua subclass langsung *throwable* didefinisikan untuk membagi class *throwable* menjadi dua cabang yang berbeda. Satu, class ***Exception,*** digunakan untuk keadaan eksepsi yang harus ditangkap oleh program yang kita buat. Sedangkan yang lain diharapkan dapat menangkap class yang kita subclasskan untuk menghasilkan keadaan eksepsi.
Cabang kedua *throwable* adalah class ***error,*** yang mendefinisikan keadaan yang tidak diharapkan untuk ditangkap dalam lingkungan normal.

EKSEPSI YANG TIDAK DAPAT DITANGKAP

Obyek eksepsi secara otomatis dihasilkan oleh runtime java untuk menanggapi suatu keadaan eksepsi :

```
class Exc0 {
public static void main (String args[]) {
        int d = 0;
        int a = 42 / d;
        }
}
```

Hasil :

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

Saat runtime java mencoba meng-eksekusi pembagian, akan terlihat bahwa pembaginya adalah nol, dan akan membentuk objek eksepsi baru yang menyebabkan program terhenti dan harus berurusan dengan keadaan kesalahan tersebut. Kita ***belum mengkodekan suatu penangan eksepsi****, sehingga penanganan eksepsi default akan segera dijalankan.*

```
class Exc1 {
static void subroutine() {
        int d = 0;
        int a = 42 / d;
        }
public static void main (Stinr args[]) {
        Exc1.subroutine();
        }
}
```

Hasil :

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

TRY Dan CATCH

Kata kunci try digunakan untuk menentukan suatu blok program yang harus dijaga terhadap semua eksepsi, setelah blok try masukkan bagian catch, yang menentukan tipe eksepsi yang akan ditangkap.

```
class Exc2 {
public static void main (String args[]) {
        try {
                int d = 0;
                int a = 42 / d;
                }
        catch (ArithmeticException e) {
                System.out.println("Division By Zero);
                }
        }
}
```

THROW

Pernyataan throw digunakan untuk secara eksplisit ***melemparkan suatu eksepsi***.
Pertama kita harus mendapatkan penanganan dalam suatu instance throwable, melalui suatu parameter kedalam bagian catch, atau dengan membuatnya menggunakan operator ***new***. Bentuk umum pernyataan throw :

throw *ThrowableInstance;*

Aliran eksekusi akan segera terhenti setelah pernyataan throw, dan pernyataan selanjutnya tidak akan dicapai. Blok try terdekat akan diperiksa untuk melihat jika telah memiliki bagian catch yang cocok dengan tipe instance Throwable. *Jika ditemukan yang cocok, maka pengaturan dipindahkan ke pernyataan tersebut*. Jika tidak, maka blok pernyataan try selanjutnya diperiksa, begitu seterusnya sampai penanganan eksepsi terluar menghentikan program dan mencetak penelusuran semua tumpukan sampai pernyataan throw.

```
class throwDemo
{
   static void demoProc()
   {
      try
      {
         throw new NullPointerException("demo");
        }
        catch (NullPointerException e)
      {
         System.out.println("caught inside demoproc…");
        throw e;
        }
   }

   public static void main (String args[])
   {
      try
      {
         demoProc();
      }
        catch (NullPointerException e)
      {
         System.out.println("recaugt : " + e);
        }
   }
}
```

Hasil :

```
caught inside demoproc…
recaugt : java.lang.NullPointerException: demo
```

THROWS

Kata kunci throws digunakan untuk ***mengenali daftar eksepsi yang mungkin di-throw*** oleh suatu method.
Jika tipe eksepsinya adalah **error**, atau **RuntimeException**, atau suatu subclassnya, ***aturan ini tidak berlaku***, karena tidak diharapkan sebagai bagian normal dari kerja program.
Jika suatu method ***secara eksplisit men-throws suatu intans dari Exception atau subclassnya***, diluar **RuntimeException**, kita harus mendeklarasikan tipenya dengan pernyataan **throws**.
Ini mendefinisikan ulang deklarasi method sebelumnya dengan sintaks sbb:

type *methodName* (arg-list) **throws** *exception-list* { }

```
class ThrowsDemo
{
   static void procedure () throws IllegalAccessException
   {
      System.out.println("Inside Procedure");
        throw new IllegalAccessException("demo");
   }

   public static void main(String args[])
   {
      try
      {
         procedure();
        }
        catch (IllegalAccessException e)
      {
         System.out.println("caught "+ e);
        }
   }
}
```

Hasil :

```
Inside Procedure
caught java.lang.IllegalAccessException: demo
```

FINALLY

Saat suatu eksepsi dilemparkan, alur program dalam suatu method membuat jalur yang cenderung tidak linier melalui method tersebut, melompati baris-baris tertentu, bahkan mungkin akan keluar sebelum waktunya pada kasus dimana tidak ada bagian **catch** yang cocok.
Kadang-kadang ***perlu dipastikan bahwa bagian program yang diberikan akan berjalan, tidak perduli eksepsi apa yang terjadi dan ditangkap.***
Kata kunci **finally** dapat digunakan untuk menentukan bagian program seperti itu.

Setiap **try** membutuhkan sekurang-kurangnya satu bagian **catch** atau **finally** yang cocok. Jika kita tidak mendapatkan bagian catch yang cocok, maka bagian finally akan dieksekusi sebelum akhir program, atau setiap kali suatu method akan kembali ke pemanggilnya, melalui eksepsi yang tidak dapat ditangkap, atau melalui pernyataan **return**, bagian finally akan dieksekusi sebelum kembali ke method kembali.

```
public class FinallyDemo2
{
   public FinallyDemo2()
   {
   }
   public static void main (String args[])
   {
      int zeroInt=0;
      int anInt=10;

      try
      {
         int divResult=anInt/zeroInt;
         System.out.println("Hasilnya adalah : "+divResult);
      }
      catch (ArithmeticException e)
      {
         System.out.println("Terjadi pembagian dengan nol");
         System.out.println("Exception Handling secara manual");
      }
      finally
      {
         System.out.println("Kalimat ini berada di dalam finally");
      }
      System.out.println("Kalimat ini berada di luar try-catch-finally");
   }
}
```

Hasil :
Terjadi pembagian dengan nol
Exception Handling secara manual
Kalimat ini berada di dalam finally
Kalimat ini berada di luar try-catch-finally

MULTITHREADING

Banyak persoalan dalam pemrograman membutuhkan kemampuan suatu program untuk melakukan beberapa hal sekaligus, atau memberikan penanganan segera terhadap suatu kejadian/ event tertentu dengan menunda aktivitas yang sedang dijalankan untuk menangani event tersebut dan akhirnya kembali melanjutkan aktivitas yang tertunda.

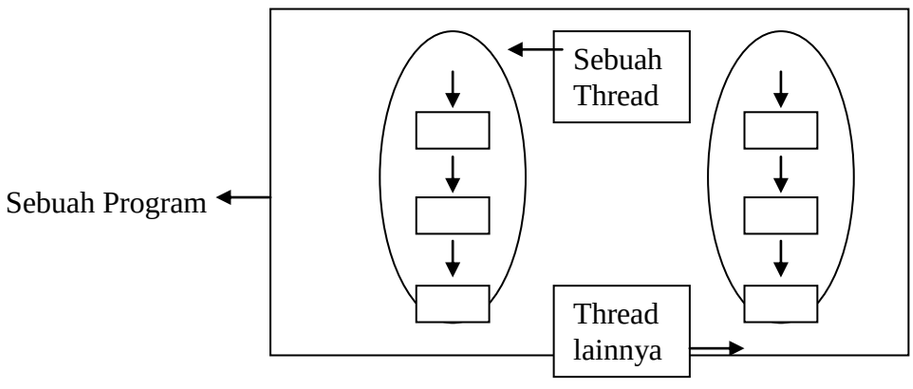
Contoh, dalam sistem aplikasi jaringan, kita dapat membuat suatu program melakukan komputasi lokal dengan data yang sudah didapat dari jaringan, pada saat program tersebut menunggu datangnya tambahan data dari jaringan. Tanpa multithreading, program tersebut harus melakukannya secara sekuensial dalam sebuah alur program tunggal (yaitu alur control utama), yang diawali dengan penantian tibanya keseluruhan data, baru kemudian komputasi. Pada masa penantian tersebut, komputer berada pada keadaan idle yang menyebabkan ketidakefisienan pada keseluruhan program.

Dengan multithreading kita dapat **menciptakan dua thread secara dinamis**, yaitu thread yang berjaga dipintu gerbang, menunggu masuknya data., dan thread yang melakukan komputasi lokal atas data yang sudah tersedia.

MULTITHREADING Dan JAVA

Thread (seringkali disebut juga ***lightweight process (LWP)*** **atau** ***execution context***) adalah sebuah *singlesequentialflow of control* didalam sebuah program. Secara sederhana, thread adalah sebuah ***subprogram yang berjalan didalam sebuah program***.

Seperti halnya sebuah program, sebuah thread mempunyai awal dan akhir. Sebuah program dapat mempunyai beberapa thread didalamnya. Jadi perbedaannya program yang *multithreaded* ***mempunyai beberapa flow of control yang berjalan secara konkuren*** atau paralel sedangkan program yang *singlethreaded* hanya mempunyai satu flow of control.

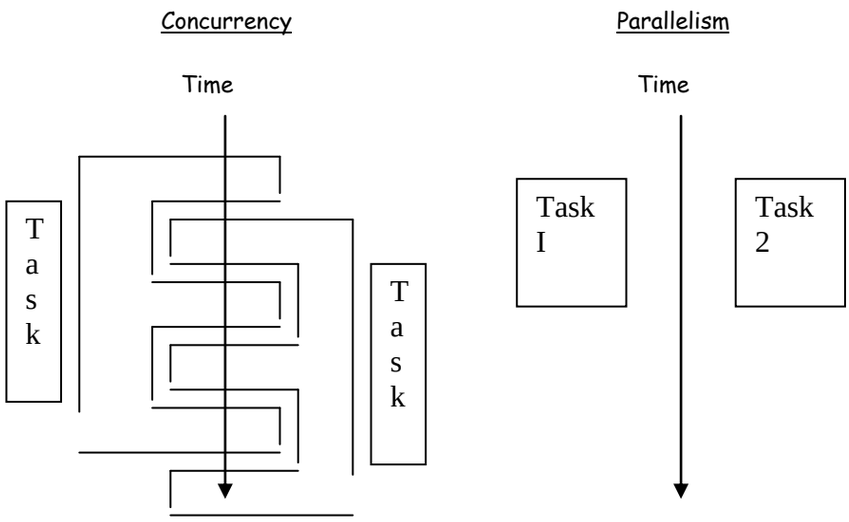


Gb.1. Dua thread dalam satu program

Dua program yang dijalankan secara terpisah (dari command line secara terpisah), berada pada dua address space yang terpisah.
Sebaliknya, ***kedua thread pada gambar diatas berada pada address space yang sama*** (address space dari program dimana kedua thread tersebut dijalankan).

Kalau program itu berjalan diatas mesin dengan single processor, maka thread-thread itu dijalankan secara konkuren(dengan mengeksekusi secara bergantian dari satu thread ke thread yang lainnya).

Jika program itu berjalan diatas mesin dengan multiple processor, maka thread-thread itu bisa dijalankan secara paralel (masing-masing thread berjalan di processor yang terpisah).

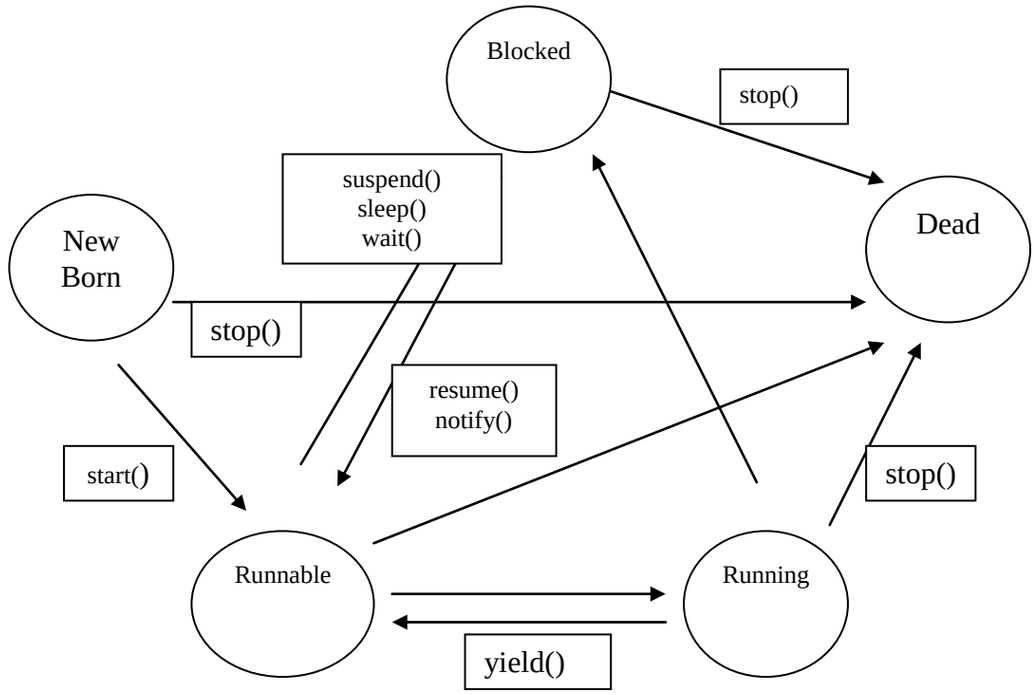


Gb.2. Konkurensi dan parallelism

Gambar 2 dapat menjelaskan perbedaan antara konkurensi dan parallelism.
Bahasa Java mempunyai kemampuan multithreading built-in, pada Java Virtual Machine terdapat thread scheduler yang menentukan thread mana yang beraksi pada selang waktu tertentu. Scheduler pada JVM mendukung ***preemptive multithreading***, yaitu suatu thread dengan prioritas tinggi dapat menyeruak masuk dan menginterupsi thread yang sedang beraksi, kemampuan ini sangat menguntungkan dalam membuat aplikasi real-time.

Scheduler pada JVM juga mendukung ***non-preemptive multithreading*** I(atau sering disebut juga cooperative multithreading), yaitu thread yang sedang beraksi tidak dapat diinterupsi, ia akan menguasai waktu CPU, sampai ia menyelesaikan tugasnya atau secara eksplisit merelakan diri untuk berhenti dan memberi kesempatan bagi thread yang lain.

DAUR HIDUP SEBUAH THREAD



Gb.3. State-state dari thread

Newborn
Sebuah thread berada pada state ini ketika dia di instantiasi. Sebuah ruangan dimemori telah dialokasikan untuk thread itu,dan telah menyelesaikan tahap inisialisasinya.

```
……
Thread timerThread = new TimerThread();
……..
```

Pada state ini, timeThread belum masuk dalam skema penjadwalan thread scheduler.

Runnable
Pada state ini, sebuah thread berada dalam skema penjadwalan, akan tetapi dia tidak sedang beraksi. Kita bisa membuat timerThread yang kita buat sebelumnya masuk ke state runnable dengan :

```
…..
timerThread.start();
……
```

Kapan tepatnya timerThread beraksi, ditentukan oleh thread scheduler.

Running
Pada state ini, thread sedang beraksi.
Jatah waktu beraksi bagi thread ini ditentukan oleh thread scheduler.
Pada kasus tertentu, thread scheduler berhak meng-interupsi kegiatan dari thread yang sedang beraksi (misalnya ada thread lainnya dengan prioritas yang lebih tinggi).
Thread dalam keadaan running bisa juga lengser secara sukarela, dan masuk kembali ke state runnable, sehingga thread lain yang sedang menunggu giliran(runnable) memperoleh kesempatan untuk beraksi. Tindakan thread yang lengser secara sukarela itu biasanya disebut **yield**-ing.

```
public void run() {
        …….
        Thread.yield();
```

…….
}
Blocked

Pada tahap ini thread sedang tidak beraksi dan diabaikan dalam penjadwalan thread scheduler. Thread yang sedang terblok menunggu sampai syarat-syarat tertentu terpenuhi, sebelum ia kembali masuk kedalam skema penjadwalan thread scheduler (masuk state runnable lagi).

Suatu thread menjadi terblok karena hal-hal berikut :

a. Thread itu ***tidur*** untuk jangka waktu tertentu, seperti berikut :

```
public void run() {
        ……
        try {
                thread.slepp(3000);
                //thread yg sedang beraksi akan tidur selama 3000 milisecond=3menit
                }
        catch (InterruptedException e) {
                …….
                }
```

b. Thread itu ***di-suspend***(). Thread yang ter-suspend() itu bisa masuk kembali ke state runnable bila ia *resume*().
seperti hal berikut :

```
……
//timerThread akan segera memasuki state blocked
timerThread.suspend();
……
timerThread.resume();
//timerThread kembali masuk state runnable
……
```

c. Bila thread tersebut ***memanggil method wait***() dari suatu object yang sedang ia kunci. Thread tersebut bisa kembali memasuki state runnable bila ada thread lain yang memanggil method notify() atau notifyAll() dari object tersebut.

d. Bila thread ini ***menunggu selesainya aktifitas yang berhubungan dengan I/O***.
Misalnya, jika suatu thread menunggu datangnya bytes dari jaringan komputer maka secara otomatis thread tersebut masuk ke state blocked.

e. Bila suatu thread ***mencoba mengakses critical section dari suatu object yang sedang dikunci oleh*** *thread lain*. Critical section adalah method/blok kode yang ditandai dengan kata ***synchronized.***

Dead

Suatu thread secara otomatis disebut mati bila *method run() – nya sudah dituntaskan* (return dari method run()). Contoh dibawah ini adalah thread yang akan mengecap state running hanya sekali saat thread scheduler memberinya kesempatan untuk running, ia akan mencetak " I'm doing something….something stupid….but I'm proud of It"… kemudian mati.

```
public class MyThread extends Thread {
        …..
        public void run() {
                System.out.print("I'm doing something…");
                System.out.print("something stupid…");
                System.out.println("but I'm proud of It…");
                // MyThread akan mati begitu baris diatas selesai dieksekusi
                }
        ……
        }
```

```
public class MultiTToy {

    public MultiTToy() {
    }

  public static void main (String args[]) {
     new NewThread("One");
     new NewThread("Two");
     new NewThread("Three");

     try {
        Thread.sleep(10000);
     } catch (InterruptedException e) {
        System.out.println("Main thread interrupted.");
     }
  }
}
class NewThread implements Runnable {
  String name;
  Thread t;

  NewThread(String threadName) {
     name = threadName;
     t = new Thread(this, name);
     System.out.println(" New thread : "+t);
     t.start();
  }
  public void run() {
     try {
        for (int i=5; i>0; i--) {
           System.out.println(name + " : " + i);
           Thread.sleep (1000);
        }
     } catch (InterruptedException e) {
        System.out.println(name+ " interrupted.");
     }
     System.out.println(name + " existing.");
  }
}
```

Hasil :

```
One : 4
Three : 4
Two : 4
Two : 3
Three : 3
One : 3
Two : 2
One : 2
Three : 2
Three : 1
One : 1
Two : 1
Two existing.
One existing.
Three existing.
```