

# **Laporan Tugas Kecil IF2210 Pemrograman Berorientasi Objek Eksplorasi Bahasa Pemrograman Berorientasi Objek**

Oleh:

Tony (13516010)

Harry Setiawan Hamjaya (13516079)



PROGRAM STUDI TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
BANDUNG  
2018

# Ruby

## I. Pengenalan Bahasa Ruby

Ruby adalah bahasa berorientasi objek murni dan semuanya tampak sebagai objek Ruby. Setiap nilai dalam Ruby adalah objek, bahkan hal yang paling primitif: string, angka dan bahkan benar dan salah. Bahkan kelas itu sendiri adalah objek yang merupakan instance dari kelas Kelas. Bab ini akan membawa Anda melalui semua fungsi utama yang terkait dengan Object Oriented Ruby.

Ruby pertama kali dibuat oleh seorang programmer Jepang bernama Yukihiro Matsumoto. Pada tahun 1993 Yukihiro ingin membuat sebuah bahasa skripting yang memiliki kemampuan orientasi objek. Pada saat itu pemrograman berorientasi objek sedang berkembang tetapi belum ada bahasa pemrograman scripting yang mendukung pemrograman objek.

Penulisan Ruby dimulai pada Februari 1993 dan pada Desember 1994 dirilis versi alpha dari Ruby. Pada awal perkembangan Ruby, Yukihiro menulis Ruby sendiri sampai pada tahun 1996 sudah terbentuk komunitas Ruby yang banyak mengontribusikan perkembangan Ruby.

Saat ini Ruby telah berkembang tidak hanya di Jepang, tetapi diseluruh dunia. Bulan Agustus tahun 2006, Macintosh telah melakukan kerja sama dengan mengintegrasikan Ruby on Rails pada Mac OS X v10.5 Leopard telah diluncurkan bulan Oktober 2007.

## II. Translasi Penulisan Kelas dan Empat Sekawan dari Java ke Ruby

### 1. Penulisan Kelas

Saat Anda mendefinisikan kelas, Anda menetapkan cetak biru untuk tipe data. Ini tidak benar-benar mendefinisikan data apa pun, tetapi mendefinisikan apa arti nama kelas, yaitu, apa objek dari kelas akan terdiri dari dan operasi apa yang dapat dilakukan pada objek tersebut.

Nama harus dimulai dengan huruf kapital dan dengan nama konvensi yang mengandung lebih dari satu kata yang dijalankan bersama dengan setiap kata yang ditulis huruf kapital dan tidak ada karakter yang memisahkan (CamelCase).

Contoh pada Java:

```
class Box{  
    //code  
}
```

Contoh pada Ruby:

```
class Box  
    //code  
end
```

## 2. Constructor

### *initialize method*

*initialize method* adalah metode kelas Ruby standar dan bekerja dengan cara yang hampir sama dengan konstruktor bekerja dalam bahasa pemrograman berorientasi objek lainnya. Metode inisialisasi berguna ketika Anda ingin menginisialisasi beberapa variabel kelas pada saat pembuatan objek. Metode ini dapat mengambil daftar parameter dan seperti metode ruby lainnya itu akan didahului oleh kata kunci *def* seperti yang ditunjukkan di bawah ini

Contoh pada Java:

```
class Box{
    private double width;
    private double height;
    public Box(double w, double h){
        width = w;
        height = h;
    }
    public double getArea(){
        return width*height;
    }
}
```

Contoh pada Ruby:

```
class Box
  def initialize(w,h)
    @width, @height = w, h
  end
end
```

## 3. Copy Constructor

Pada Java dan Ruby keduanya tidak memerlukan copy constructor karena keduanya hanya menunjuk (*me-reference*) objek jadi ketika ingin melakukan copy constructor lebih baik untuk membuat objek baru dibandingkan dengan melakukan copy constructor karena keduanya akan menunjuk objek yang sama

## 4. Destructor

Pada Java dan Ruby juga keduanya tidak perlu diimplementasikan destructor secara khusus karena telah disediakan *garbage* khusus untuk menampung objek yang sudah tidak digunakan

## 5. Object Assignment

Pada Java dan Ruby keduanya tidak memerlukan copy constructor karena keduanya hanya menunjuk (me-*reference*) objek jadi ketika ingin melakukan copy constructor lebih baik untuk membuat objek baru dibandingkan dengan melakukan copy constructor karena keduanya akan menunjuk objek yang sama

### III. Translasi kelas Inheritance Java ke Ruby

Salah satu konsep paling penting dalam pemrograman berorientasi objek adalah *inheritance*. *Inheritance* memungkinkan kita untuk mendefinisikan kelas dalam hal kelas lain, yang membuatnya lebih mudah untuk membuat dan memelihara aplikasi.

*Inheritance* juga memberikan kesempatan untuk menggunakan kembali fungsionalitas kode dan waktu implementasi yang cepat tetapi sayangnya Ruby tidak mendukung berbagai level pewarisan tetapi Ruby mendukung *mixin*. Sebuah mixin seperti implementasi khusus dari multiple inheritance di mana hanya bagian interface yang diwariskan.

Saat membuat kelas, alih-alih menulis anggota data dan anggota yang benar-benar baru, programmer dapat menunjuk bahwa kelas baru harus mewarisi anggota kelas yang ada. Kelas yang ada ini disebut kelas dasar atau superclass, dan kelas baru disebut sebagai kelas turunan atau sub-kelas.

Ruby juga mendukung konsep subclassing, yaitu, pewarisan dan contoh berikut menjelaskan konsepnya. Sintaks untuk memperluas kelas itu sederhana. Cukup tambahkan <karakter dan nama superclass ke pernyataan kelas Anda. Sebagai contoh, berikut mendefinisikan kelas *BigBox* sebagai subkelas *Box*

Contoh pada Java:

```
class Box{
    private double width;
    private double height;
    public Box(double w, double h){
        width = w;
        height = h;
    }
    public double getArea(){
        return width*height;
    }
}

class BigBox extends Box{
    public BigBox(double w, double h){
        super(w,h);
    }
    public void printArea(){
        System.out.println("Big box area is : "+super.getArea());
    }
}

public class Test
{
    public static void main(String args[])
    {
        BigBox box = new BigBox(10, 20);
```

```

        box.printArea();
    }
}

```

Contoh pada Ruby:

```

class Box
  def initialize(w,h)
    @width, @height = w, h
  end
  def getArea
    @width * @height
  end
end

class BigBox < Box
  def printArea
    @area = @width * @height
    puts "Big box area is : #@area"
  end
end

box = BigBox.new(10, 20)
box.printArea()

```

## IV. Keunikan dari bahasa Ruby

### 1. Sistem Boolean

Di Ruby hanya nil dan salah evaluasi ke nilai false. Ini berarti bahwa segala sesuatu yang lain bernilai benar! Oleh karena itu bahkan nilai 0 mengevaluasi true di ruby. Mengikuti cuplikan kode akan menampilkan “Hello World” di konsol,

```

if (0) then
  print "Hello World"
end

```

### 2. Method *attr\_accessor*

Bayangkan memiliki banyak variabel instan dan metode penyetel dan pengambil mereka. Kode itu akan sangat panjang. Ruby memiliki cara built-in untuk secara otomatis membuat metode pengambil dan penyetel ini menggunakan metode *attr\_accessor*.

Metode *attr\_accessor* mengambil simbol nama variabel instance sebagai argumen, yang digunakan untuk membuat metode pengambil dan penyetel. Sebagai contoh:

```

class Person

  attr_accessor :name, :age

```

```
def initialize(name, age)
  @name = name
  @age = age
end

p = Person.new("David", 20)
p.name = "Bob"
puts p.name #result Bob
puts p.age #result 20
```

### 3. Assignement yang bersifat parallel

Ruby mendukung assignment paralel - Ada kemungkinan untuk mengubah banyak variabel dalam satu penugasan. Contoh terbaik adalah menukar dua variabel seperti yang diberikan di bawah ini,

```
a,b = b,a
```

# Rust

## I. Pengenalan Bahasa Rust

Rust adalah bahasa pemrograman system yang multi paradigma, yang dideskripsikan aman, konkuren, dan bahasa yang muda. Secara sintaks, Rust mirip dengan C++, tapi didesain supaya lebih aman dalam mengakses memori. Rust adalah bahasa pemrograman yang open source. Rust pertama kali muncul pada tahun 2010.

## II. Translasi Penulisan Kelas dan Empat Sekawan dari c++ ke Rust

### 1. Penulisan Kelas

C++	Rust
<pre>class Kelas{     // Atribut dan Method Kelas };</pre>	<pre>struct Kelas{     // Field/Atribut Kelas }  impl Kelas{     // Method Kelas }</pre>

### 2. Constructor

Pada dasarnya semua struct pada rust memiliki konstruktor dasar yaitu

[Nama Kelas]{ [Nama Filed 1] : [Nilai Field 1], ..., [Nama Filed n] : [Nilai Field n]}

Tapi dapat dibuat alternatif dengan mengimplementasi method new sehingga penggunaan konstrukornya menjadi

[Nama Kelas]::new([Nama Filed 1], ..., [Nama Filed n])

C++	Rust
<pre>class Kelas{ private:     type1 var1;     type2 var2; public:     Kelas(type1 var1, type2 var2)){         this-&gt;type1 = type1;         this-&gt;type2 = type2;     } };</pre>	<pre>struct Kelas{     var1 : type1,     var2 : type2 }  impl Kelas{     // Konstruktor     fn new(var1 : type1, var2 : type2) -&gt; self{         Kelas{var1 : var1, var2 : type2}     } }</pre>

<pre>int main(){     type1 var1;     type2 var2;     Kelas a(var1, var2); }</pre>	<pre>fn main(){     let a = Kelas::new(var1, var2); }</pre>
---	---

### 3. Copy Constructor dan Assignment

Pada rust Copy Constructor dan Assignment dapat ditambahkan ke struct hanya dengan menambahkan kode

```
#[derive (Clone, Copy)]
```

Diatas kelas yang diinginkan. Atau dengan menimplementasi trait Clone yang memiliki method clone() dan/atau trait Copy yang memiliki method copy()

C++	Rust
<pre>class Kelas{ private:     type1 var1;     type2 var2; public:     // Copy Constructor     Kelas(const Kelas &amp;other){         type1 = other.type1;         type2 = other.type2;     }     // Operator Assignment     Kelas&amp; operator=(const Kelas &amp;other){         type1 = other.type1;         type2 = other.type2;         return *this;     } };  int main(){     Kelas a;     Kelas b = a; // cctor     Kelas c;     c = a; // assignment operator }</pre>	<pre>// Copy Constructor dan assignment #[derive (Clone, Copy)] struct Kelas{     var1 : type1,     var2 : type2 }  impl Kelas{     // Konstruktor     fn new(var1 : type1, var2 : type2) -&gt; self{         Kelas{var1 : var1, var2 : var2}     } }  fn main(){     let a = Kelas::new(var1, var2);     let b = a;     let c = a;     b = c; }</pre>

### 4. Destructor

Destructor pada rust dapat diimplementasikan dengan mengimplementasi method drop(&self) pada trait Drop.



C++	Rust
<pre>class Kelas{ private:     type1 var1;     type2 var2; public:     // Destructor     ~Kelas(){         // Delete pointer     } };</pre>	<pre>struct Kelas{     var1 : type1,     var2 : type2 }  // Destructor impl Drop for Kelas{     fn drop(&amp;mut Self){         // Dealloc Field     } }</pre>

### III. Translasi Penulisan Kelas Inheritance dari C++ ke Rust

C++	Rust
<pre>class Parent{ private:     ... public:     void testfunc(){         ...     }     ... };  class Child : public Parent{ private:     ... public:     void testfunc(){         Parent::testfunc();// call parent method         ...     }     ... };</pre>	<pre>// Inheritance trait Function {     fn testfunc(&amp;Self);     ... }  struct Parent {     ... }  impl Function for Parent{     fn testfunc(&amp;self){         // do something     }     ... }  struct Child {     parent: Parent,     ... }  impl Function for Child{     fn testfunc(&amp;self){         self.parent.testfunc(); // call parent method         // do something     }     ... }</pre>

### IV. Contoh kelas pada C++ dan Rust

C++	Rust
<pre>class Point2D{ private:     double x, y; public:     Point2D(double x, double y);     Point2D(const Point2D &amp;other);     Point2D&amp; operator= (const Point2D&amp; other);     double getX() const{</pre>	<pre>trait PointFunction{     fn getX(&amp;self) -&gt; f64;     fn getY(&amp;self) -&gt; f64;     fn to_origin(&amp;mut self);     fn print(&amp;self);     fn setX(&amp;mut self, x : f64);     fn setY(&amp;mut self, y : f64); }</pre>

```

        return x;
    }
    double getY() const{
        return y;
    }
    void setX(double x){
        this->x = x;
    }
    void setY(double y){
        this->y = y;
    }
    void to_origin();
    void print();
    double distance_to(Point2D &other);
};

Point2D::Point2D(double x, double y){
    this->x = x;
    this->y = y;
}

Point2D::Point2D(const Point2D &other){
    x = other.x;
    y = other.y;
}

Point2D& Point2D::operator= (const
Point2D& other){
    x = other.x;
    y = other.y;
    return *this;
}

void Point2D::to_origin(){
    x = 0;
    y = 0;
}

void Point2D::print(){
    cout<<"x : "<<x<<endl;
    cout<<"y : "<<y<<endl;
}

double Point2D::distance_to(Point2D
&other){
    double dx = x - other.x;
    double dy = y - other.y;
    return dx*dx + dy*dy;
}

class Point3D : public Point2D{
private:

```

```

#[derive (Clone, Copy)]
struct Point2D {
    x : f64,
    y : f64
}

impl Point2D{
    fn new(x : f64, y : f64) -> Self{
        Point2D{x : x, y : y}
    }
    fn distance_to(&self, other :
&Point2D) -> f64 {
        let dx = self.x - other.x;
        let dy = self.y - other.y;
        dx*dx - dy*dy
    }
}

impl PointFunction for Point2D {
    fn getX(&self) -> f64 {
        self.x
    }
    fn getY(&self) -> f64 {
        self.y
    }
    fn to_origin(&mut self) {
        self.x = 0.0;
        self.y = 0.0;
    }
    fn print(&self){
        println!("x : {:?}", self.x);
        println!("y : {:?}", self.y);
    }
    fn setX(&mut self, x : f64){
        self.x = x;
    }
    fn setY(&mut self, y : f64){
        self.y = y;
    }
}

#[derive (Clone, Copy)]
struct Point3D {
    parent : Point2D,
    z : f64
}

```

```

    double z;
public:
    Point3D(double x, double y, double
z);
    Point3D(const Point3D &other);
    Point3D& operator= (const Point3D&
other);
    double getZ() const{
        return z;
    }
    void setZ(double z){
        this->z = z;
    }
    void to_origin();
    void print();
    double distance_to(Point3D &other);
};

Point3D::Point3D(double x, double y,
double z): Point2D(x, y){
    this->z = z;
}
Point3D::Point3D(const Point3D
&other):Point2D(other.getX(),
other.getY()){
    z = other.z;
}
Point3D& Point3D::operator= (const
Point3D& other){
    setX(other.getX());
    setY(other.getY());
    z = other.z;
    return *this;
}
void Point3D::to_origin(){
    Point2D::to_origin();
    z = 0;
}
void Point3D::print(){
    Point2D::print();
    cout<<"z : "<<z<<endl;
}
double Point3D::distance_to(Point3D
&other){
    double dx = getX() - other.getX();
    double dy = getY() - other.getY();
    double dz = z - other.z;

```

```

impl Point3D{
    fn new(x : f64, y : f64, z : f64)
-> Self{
        Point3D{parent :
Point2D::new(x, y), z : z}
    }
    fn getZ(&self) -> f64 {
        self.z
    }
    fn distance_to(&self, other :
&Point3D) -> f64 {
        let dx = self.parent.getX() -
other.parent.getX();
        let dy = self.parent.getY() -
other.parent.getY();
        let dz = self.z - other.z;
        dx*dx + dy*dy + dz*dz
    }
    fn setZ(&mut self, z : f64){
        self.z = z;
    }
}

impl PointFunction for Point3D {
    fn getX(&self) -> f64 {
        self.parent.getX()
    }
    fn getY(&self) -> f64 {
        self.parent.getY()
    }
    fn to_origin(&mut self) {
        self.parent.to_origin();
        self.z = 0.0;
    }
    fn print(&self){
        self.parent.print();
        println!("z : {:?}", self.z);
    }
    fn setX(&mut self, x : f64){
        self.parent.setX(x);
    }
    fn setY(&mut self, y : f64){
        self.parent.setY(y);
    }
}

```

<pre>    return dx*dx + dy*dy + dz*dz; }</pre>	
--	--

## V. Hal yang unik dari Rust

### 1. Memory Management

Rust dapat secara otomatis men-dealokasi variabel yang telah digunakan tanpa menggunakan sistem *Garbage Collector*, tapi rust manajemen sumber daya dengan menggunakan sistem *resource acquisition is initialization* (RAII).

### 2. Sistem Ownership

Rust memiliki sistem *Ownership* dimana semua nilai memiliki pemilik yang unik dimana setiap scope dari suatu nilai sama dengan scope dari pemilik. Nilai dapat dipassing menggunakan *immutable reference* dengan `&T`, *mutable reference* dengan `&mut T`, atau hanya nilai itu sendiri dengan `T`. Semua aturan itu di cek dan saat waktu *compile*.

### 3. Memory Safety

Rust didesain sehingga sistem itu *memory safe* (artinya adalah status yang terlindungi oleh *software bugs* dan *security vulnerabilities* ketika berurusan dengan pengaksesan memori), sehingga rust tidak membiarkan adanya *null pointer*, *dangling pointer*, ataupun *data races*.