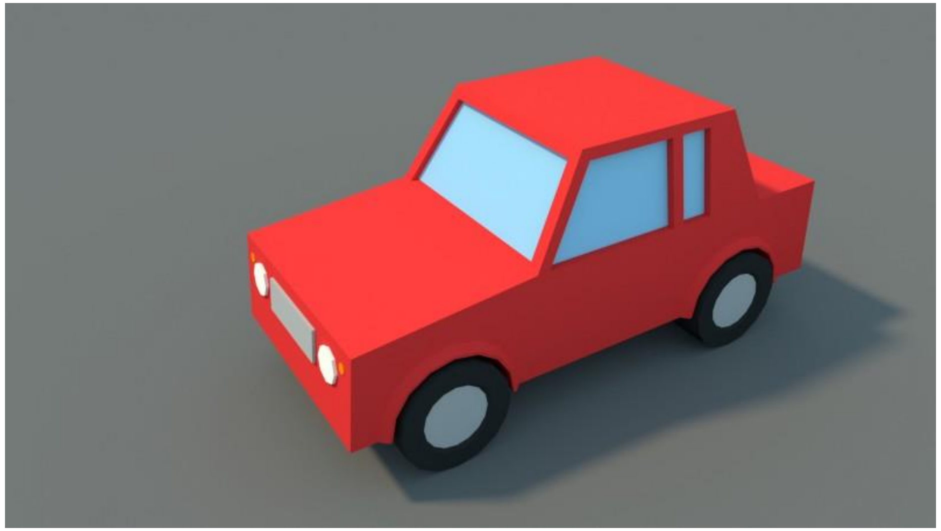
Lab 3

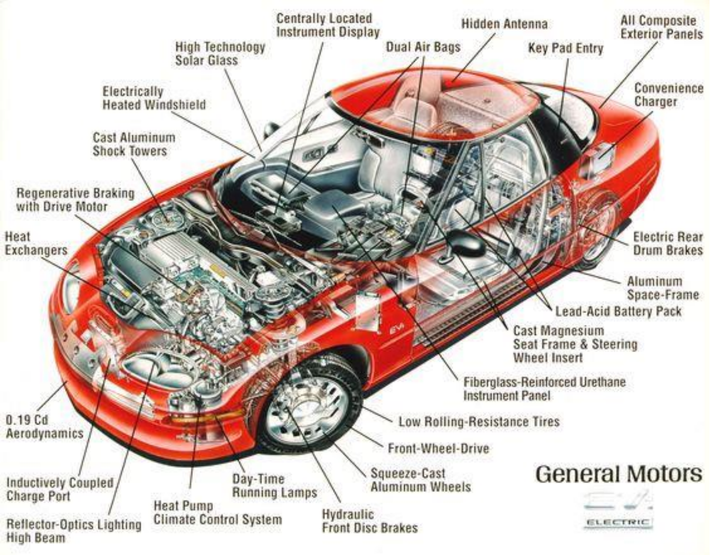
OOP – Abstraction, Polymorphism, Inheritance, Encapsulation

A. P. I. E.

1. Abstraction

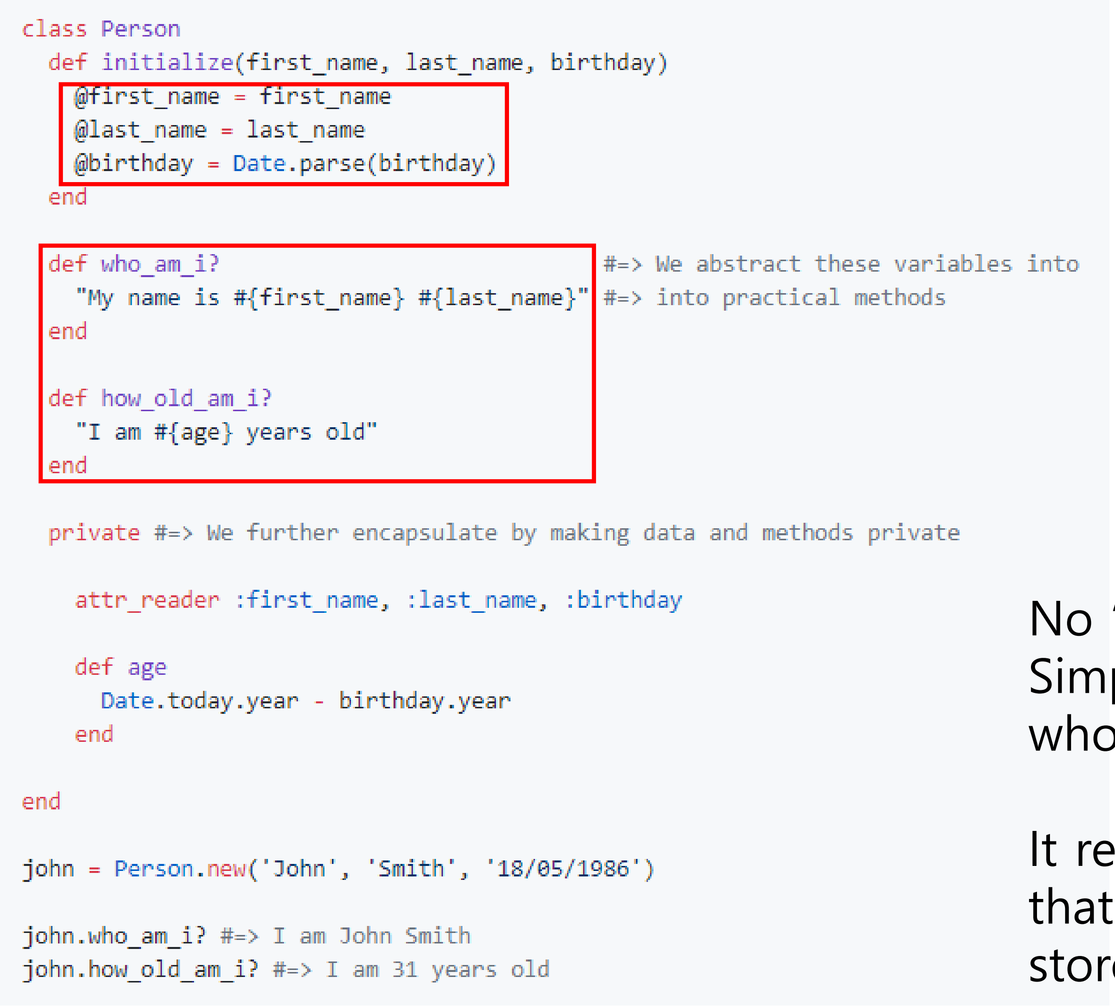
- Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.





Centrally Located Instrument Display
Hidden Antenna
All Composite Exterior Panels
High Technology Solar Glass
Dual Air Bags
Key Pad Entry
Electrically Heated Windshield
Convenience Charger
Cast Aluminum Shock Towers
Regenerative Braking with Drive Motor
Heat Exchangers
Electric Rear Drum Brakes
Aluminum Space-Frame
Lead-Acid Battery Pack
Cast Magnesium Seat Frame & Steering Wheel Insert
Fiberglass-Reinforced Urethane Instrument Panel
0.19 Cd Aerodynamics
Low Rolling-Resistance Tires
Front-Wheel-Drive
General Motors
Inductively Coupled Charge Port
Day-Time Running Lamps
Squeeze-Cast Aluminum Wheels
Heat Pump Climate Control System
Hydraulic Front Disc Brakes
Reflector-Optics Lighting High Beam
ELECTRIC

1. Abstraction



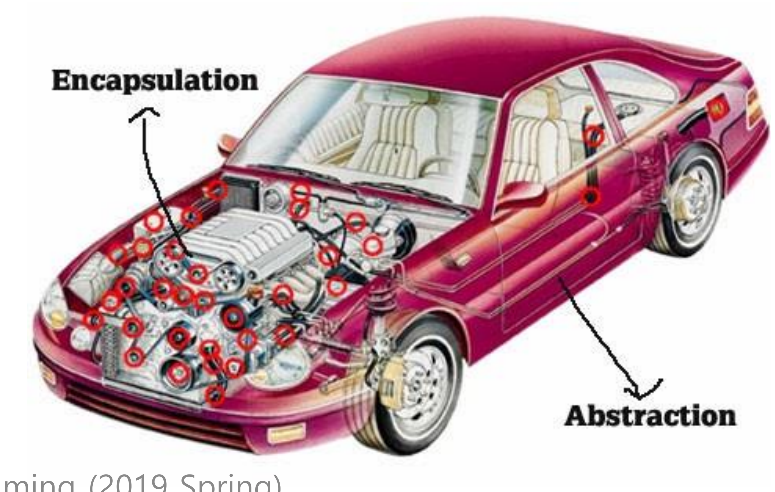
```ruby
class Person
  def initialize(first_name, last_name, birthday)
    @first_name = first_name
    @last_name = last_name
    @birthday = Date.parse(birthday)
  end

  def who_am_i?                             #=> We abstract these variables into
    "My name is #{first_name} #{last_name}" #=> into practical methods
  end

  def how_old_am_i?
    "I am #{age} years old"
  end

  private #=> We further encapsulate by making data and methods private

    attr_reader :first_name, :last_name, :birthday

    def age
      Date.today.year - birthday.year
    end

end

john = Person.new('John', 'Smith', '18/05/1986')

john.who_am_i? #=> I am John Smith
john.how_old_am_i? #=> I am 31 years old
```

No “first_name” and “last_name”,
Simply ask the object
who it is or how old it is.

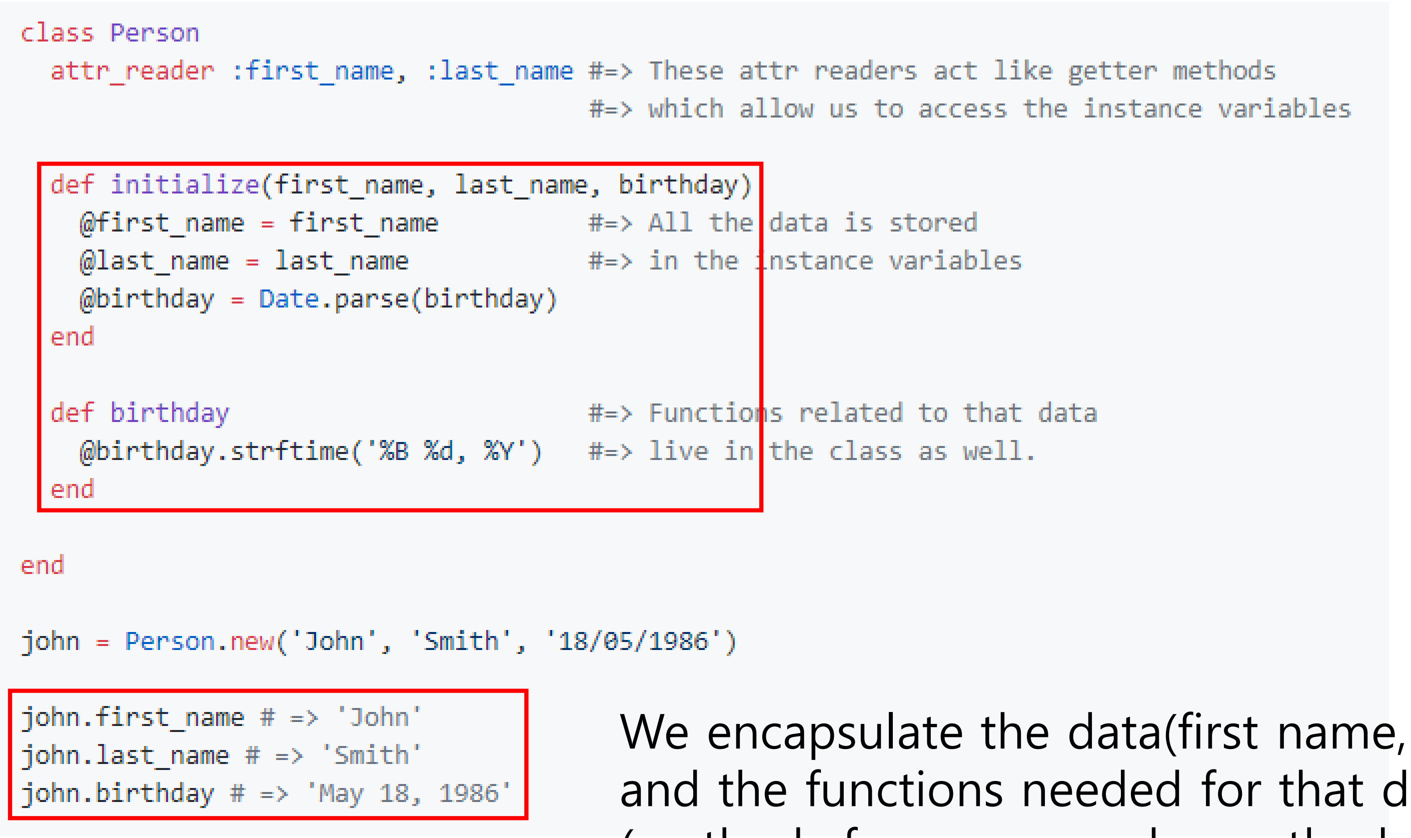
It returns nicely formatted strings
that interpolate the data we have
stored or data we have calculated.

2. Encapsulation

- Encapsulation is a process where you keep all the inner works of the system together hidden. The important works are stored hidden to keep it safe from the average user which ensures the integrity of the system as it was designed.



2. Encapsulation

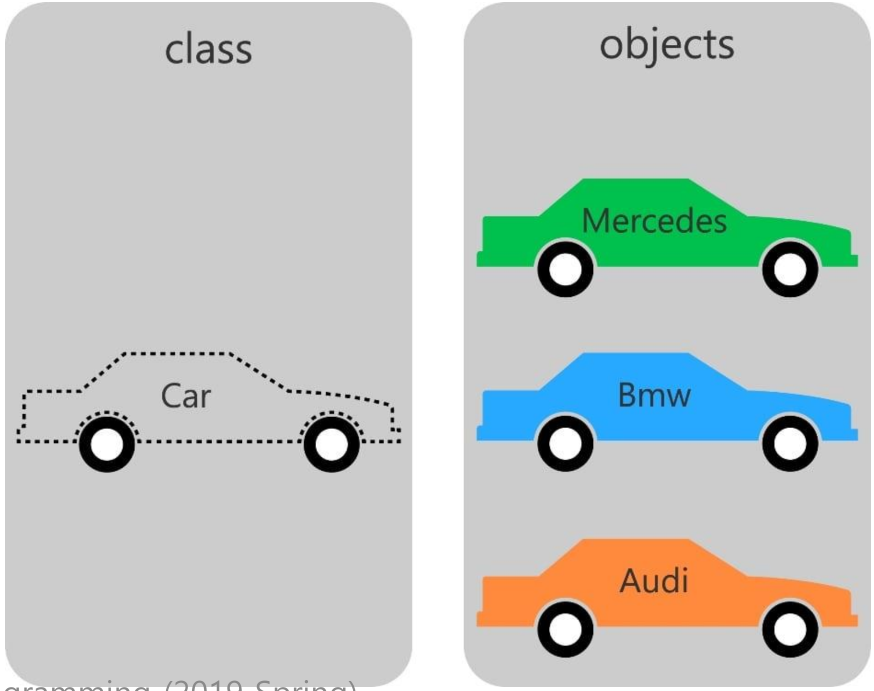


```ruby
class Person
  attr_reader :first_name, :last_name #=> These attr readers act like getter methods
                                      #=> which allow us to access the instance variables

  def initialize(first_name, last_name, birthday)
    @first_name = first_name          #=> All the data is stored
    @last_name = last_name            #=> in the instance variables
    @birthday = Date.parse(birthday)
  end

  def birthday                        #=> Functions related to that data
    @birthday.strftime('%B %d, %Y')   #=> live in the class as well.
  end

end

john = Person.new('John', 'Smith', '18/05/1986')

john.first_name # => 'John'
john.last_name # => 'Smith'
john.birthday # => 'May 18, 1986'
```

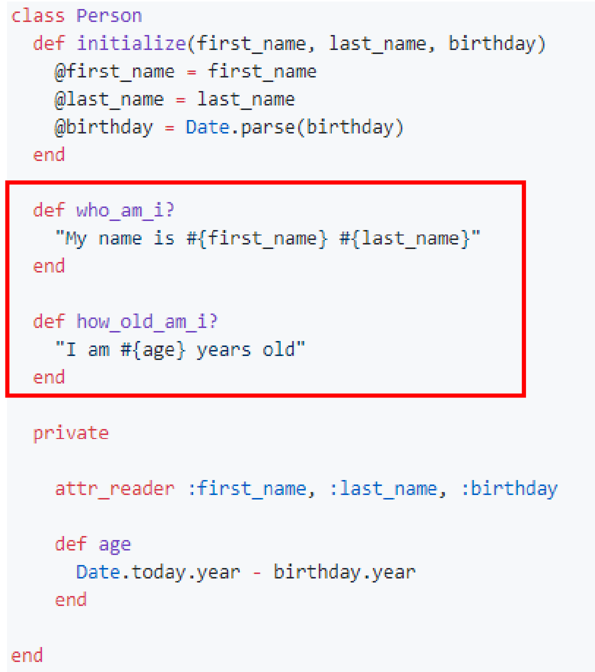
We encapsulate the data(first name, last name, age) and the functions needed for that data (methods for names and a method for date) inside the class. We have packaged all relevant Information together and only the methods within that package can directly manipulate that information.

3. Inheritance

- Inheritance is the mechanism by which an object acquires the some/all properties of another object.
- It supports the concept of hierarchical classification.



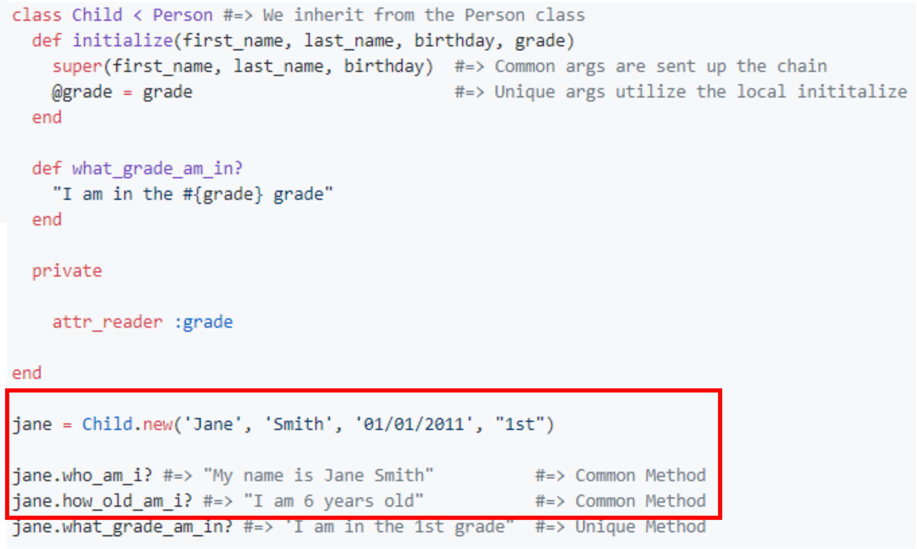
3. Inheritance



```
class Person
  def initialize(first_name, last_name, birthday)
    @first_name = first_name
    @last_name = last_name
    @birthday = Date.parse(birthday)
  end

  def who_am_i?
    "My name is #{first_name} #{last_name}"
  end

  def how_old_am_i?
    "I am #{age} years old"
  end

  private

    attr_reader :first_name, :last_name, :birthday

    def age
      Date.today.year - birthday.year
    end

end
```

As you can see in the example Jane can tell you who she is and how old she is, but she can also tell you what grade she is in. We are able to extend the functionality of a child class without duplicating all the code from the parent.

It makes our code reusable and keeps us from repeating ourselves.



```
class Child < Person #=> We inherit from the Person class
  def initialize(first_name, last_name, birthday, grade)
    super(first_name, last_name, birthday)  #=> Common args are sent up the chain
    @grade = grade                          #=> Unique args utilize the local inititalize
  end

  def what_grade_am_in?
    "I am in the #{grade} grade"
  end

  private

    attr_reader :grade

end

jane = Child.new('Jane', 'Smith', '01/01/2011', "1st")

jane.who_am_i? #=> "My name is Jane Smith"          #=> Common Method
jane.how_old_am_i? #=> "I am 6 years old"           #=> Common Method
jane.what_grade_am_in? #=> "I am in the 1st grade"  #=> Unique Method
```

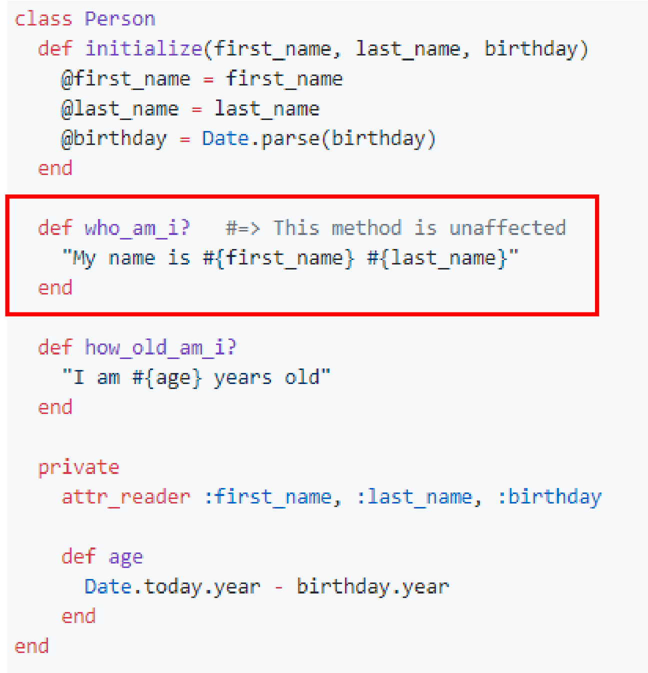
4. Polymorphism

- Polymorphism means to process objects differently based on their data type.
- One method with multiple implementation, for a certain class of action.

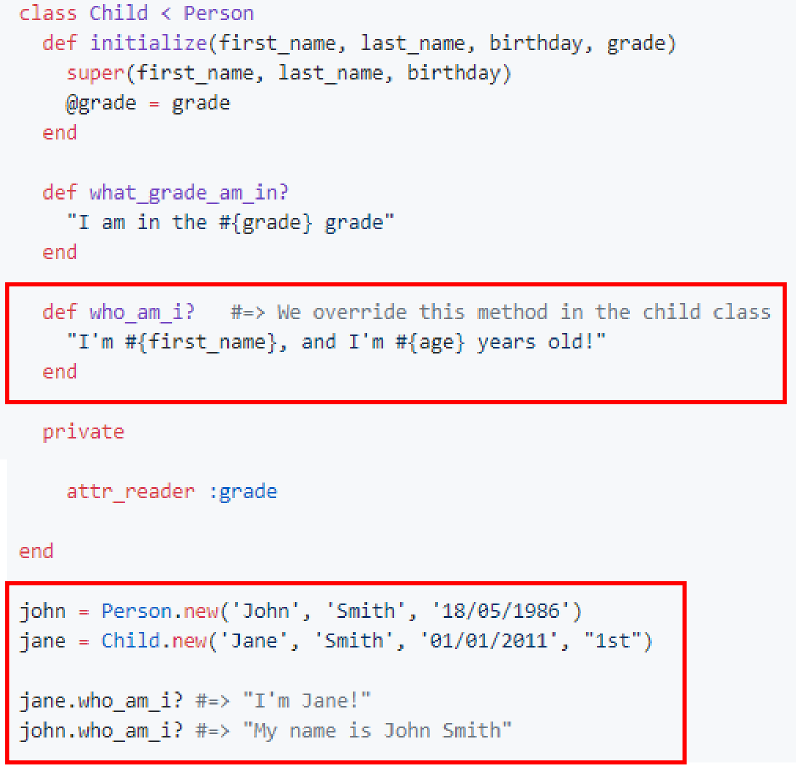
"Now Speak!"
"Quack"
"Woof"
"Meow"

4. Polymorphism

If we override the “who_am_i?” method
in the child class to only offer a first name
but additionally offer an age,
we still have not changed the interface
with the code from the user perspective.



```ruby
class Person
  def initialize(first_name, last_name, birthday)
    @first_name = first_name
    @last_name = last_name
    @birthday = Date.parse(birthday)
  end

  def who_am_i?    #=> This method is unaffected
    "My name is #{first_name} #{last_name}"
  end

  def how_old_am_i?
    "I am #{age} years old"
  end

  private
    attr_reader :first_name, :last_name, :birthday

    def age
      Date.today.year - birthday.year
    end
end
```



```ruby
class Child < Person
  def initialize(first_name, last_name, birthday, grade)
    super(first_name, last_name, birthday)
    @grade = grade
  end

  def what_grade_am_in?
    "I am in the #{grade} grade"
  end

  def who_am_i?    #=> We override this method in the child class
    "I'm #{first_name}, and I'm #{age} years old!"
  end

  private

    attr_reader :grade

end

john = Person.new('John', 'Smith', '18/05/1986')
jane = Child.new('Jane', 'Smith', '01/01/2011', "1st")

jane.who_am_i? #=> "I'm Jane!"
john.who_am_i? #=> "My name is John Smith"
```

Constructor

- Form of the Constructor
    - The class has the same name as the function.
    - Return type not declared, not actually returned.
    - A kind of function that allows default values to be set for overload and parameter.

```
#include <iostream> using  namespace std;

class Constructor
{
    int num1;
    int num2;

public:
    Constructor()
    {
        num1=0;
        num2=0;
    }
    Constructor(int n)
    {
        num1=n;
        num2=0;
    }
    Constructor(int  n1,  int n2)
    {
        num1=n1;
        num2=n2;
    }

     /* default parameter constructor
    Constructor(int  n1=0,  int n2=0)
    {
        num1=n1;
        num2=n2;
    }
    */

    void  ShowData() const
    {
        cout<<num1<<' '<<num2<<endl;
    }
};

int main(void) {
    Constructor sc1;
    sc1.ShowData();

    Constructor sc2(100);
    sc2.ShowData();

    Constructor sc3(100, 200);
    sc3.ShowData();
    return 0;
}
```

When sc1, sc2, sc3 objects are being made, they pass overloaded constructor. If you use the defualt parameter constructor, then you erase other contructors. the result is same.

Constructor

- The initialization using member initializer
  - Use member initializer when you call constructors of the objects which is declared as member variable.
  - Not initialize at the body, initialize at the next of the parameters.

```
#include <iostream>
using namespace std;
class Constructor
{
        int num1;
        int num2;
public:
        Constructor(int n1, int n2) : num1(n1), num2(n2)
        {

        }

        void ShowData() const
        {
                cout<<num1<<' '<<num2<<endl;
        }
};


int main(void)
{
        Constructor sc(100,200);
        sc.ShowData();
        return 0;
}
```

Destructor

- Destruct the resources which is allocated by constructor.

- If there is memory space allocated by new operator, then destructor destruct this memory space

- reference>> new and delete
  - They are compared to malloc and free respectively.
  - When you generate objects, you have to use "new".

```
#include <iostream>
#include <cstring>
using namespace std;
class Book
{
private:
	char * bookName;
	int bookNum;
public:
	Book(char * tempName, int tempNum)
	{
		int len=strlen(tempName)+1;
		bookName=new char[len];
		strcpy(bookName, tempName);
		bookNum=tempNum;
	}
	void ShowBookInfo() const
	{
		cout<<"Book Name : "<<bookName<<endl;
		cout<<"Book Number : "<<bookNum<<endl;
	}
	~Book()
	{
		delete []bookName;
		cout<<"destructor"<<endl;
	}
};

int main(void)
{
	Book book1("Computer Programming", 2001001);
	Book book2("This is C++", 400010);
	book1.ShowBookInfo();
	book2.ShowBookInfo();
	return 0;
}
```

Copy Constructor

- Copy Constructor
  - When you recall name which is generated in parameter, copy the object.
  - If you are not definite copy constructor, default copy constructor insert automatically.
- Kinds of copy constructor through conversions
  - implicit conversion : = , explicit conversion : (object)
  - you have to use explict to prevent implicit conversion

Copy Constructor

- Call point of copy generator
  - 1. Initialize a new object using a already generated object.

    ```
    Point x2(x1);
    ```

  - 2. Call-by-value : pass the object as a parameter during the function calling

    ```
    Point copyFunc(Point obj)
    {
    return obj;
    }
    ```

Copy Constructor

- Call point of copy generator
  - 3. return the object which is not returned by the references.

```
Point copyFunc(Point obj)
{
return obj;
}
```

```
#include <iostream>
#include <cstring>
using namespace std;
class Book
{
private:
    char * bookName;
    int bookNum;
public:
    Book(char * tempName, int tempNum)
    {
        int len=strlen(tempName)+1;
        bookName=new char[len];
        strcpy(bookName, tempName);
        bookNum=tempNum;
    }
    void ShowBookInfo() const
    {
        cout<<"Book Name : "<<bookName<<endl;
        cout<<"Book Number : "<<bookNum<<endl;
    }
    ~Book()
    {
        delete []bookName;
        cout<<"destructor"<<endl;
    }
};
```

```
int main(void)
{
    Book book1("Computer Programming", 2001001);
    Book book2("This is C++", 400010);
    Book book3(book2);
    book1.ShowBookInfo();
    book2.ShowBookInfo();
    book3.ShowBookInfo();
    return 0;
}
```

Copy Constructor

- if you not define any copy constructor, a default copy constructor copies member to member.
- Upper code has an error, the default copy constructor points same book name part, destructor destruct at book2, and destructor destruct at book3, too. But there is nothing to destruct. because string already destructed. so, error appears.
- To solve this problem, it needs to copy this book name part into another memory. this is called "deep copy".

```
Book(const Book& copy) : bookNum(copy.bookNum)
{
bookName = new char[strlen(copy.bookName)+1];
strcpy(bookName, copy.bookName);
}
```

Vector

- An array-based container that supports a random access iterator.
- Elements are stored consecutively in one memory block

```
template<typename T, typename Allocator = allocator<T>>
class vector
```

v.pop_back() : Remove the last element of v.
v.push_back() : Add the element to the end of v

```
#include <iostream>
#include <vector>
using namespace std;
int main(void)
{
	vector<int> v;
	v.push_back(5);
	v.push_back(2);
	v.pop_back();
}
```

Polymorphism

- Same sentence but different result.
- Polymorphism : the method of implementing all of the super -class' member. Sub-class has its own member and super class' member.
- Is-a relation.

```
class Person
{
private:
	int age;
	char name[50];
public:
	Person(int myage, char * myname) : age(myage)
	{
	strcpy(name, myname);
	}
	void ShowName() const
	{
	cout<<"My name is"<<name<<endl;
	}
	void ShowAge() const
	{
	cout<<"My age is"<<age<<endl;
	}
};
```

```
class Student : public Person
{
private:
	char major[50];
public:
Student(char * myname, int myage, char * mymajor) : Person(myage, 
myname)
	{
	strcpy(major, mymajor);
	}
	void ShowStudent() const
	{
	ShowName();
	ShowAge();
	cout<<"My major is"<<major<<endl<<endl;
	}
};
```

Student is a person. (is-a relation), student class inherits person class.
Student is implemented by 'public Person’.
And Student is inherited from Person's member.