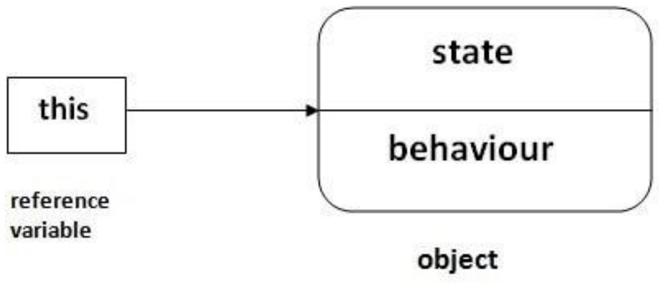
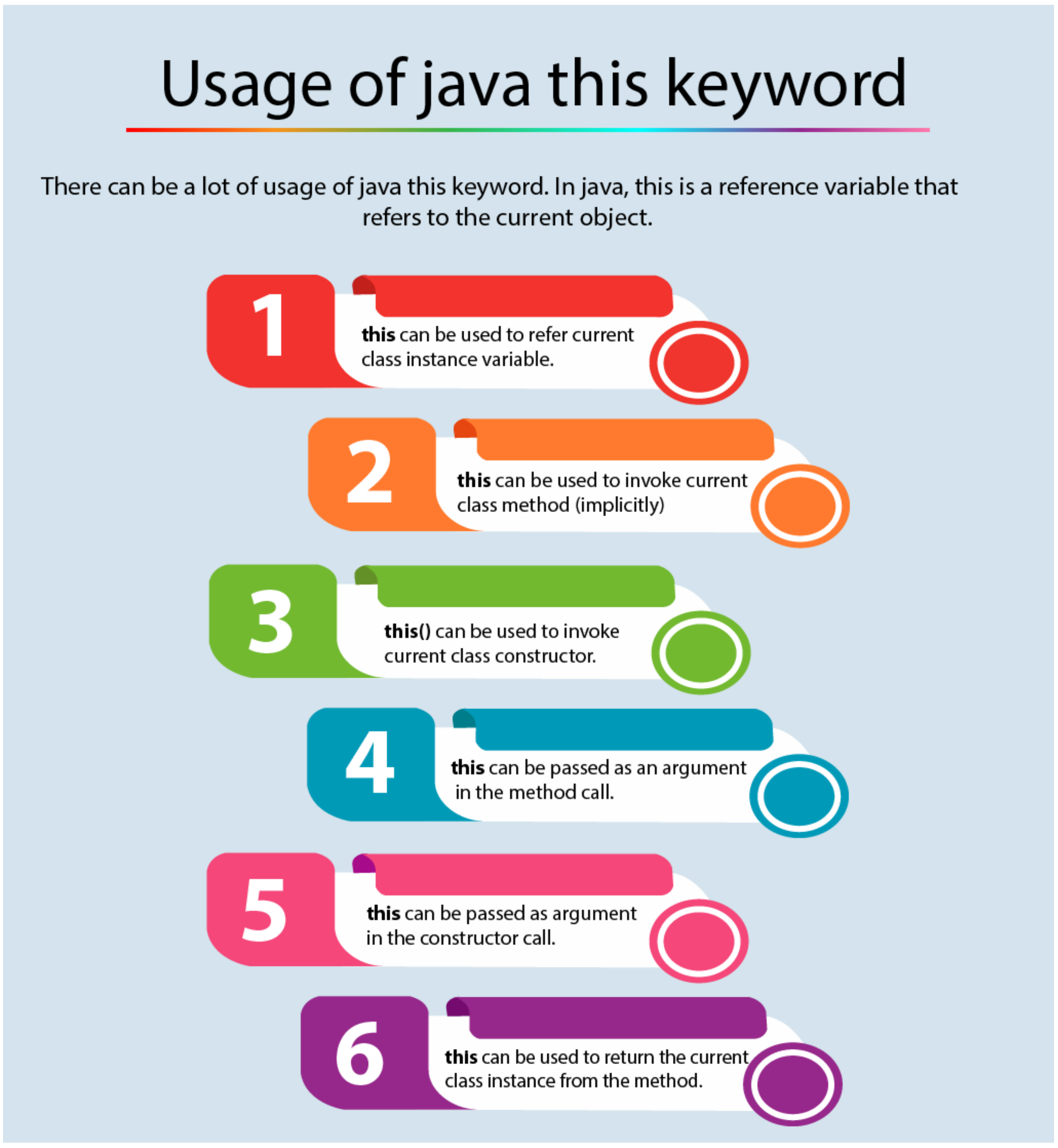
There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.





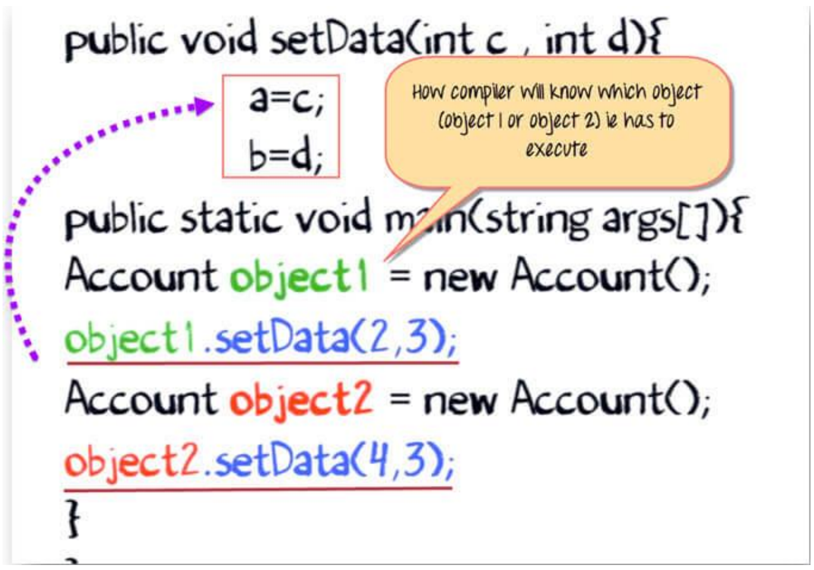
Usage of java this keyword

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

1. **this** can be used to refer current class instance variable.
2. **this** can be used to invoke current class method (implicitly)
3. **this()** can be used to invoke current class constructor.
4. **this** can be passed as an argument in the method call.
5. **this** can be passed as argument in the constructor call.
6. **this** can be used to return the current class instance from the method.

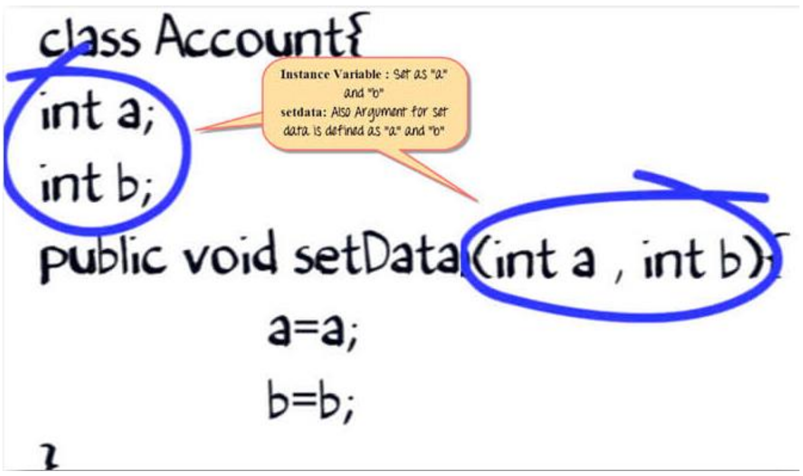
*1- Using **this** with a Field*

The most common reason for using this keyword is because a field is shadowed by a method or constructor parameter.



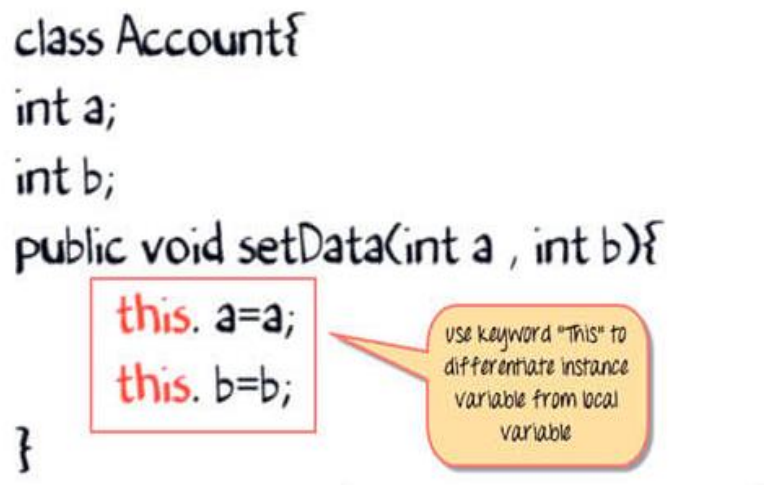
```
public void setData(int c , int d){
        a=c;
        b=d;
public static void main(string args[]){
Account object1 = new Account();
object1.setData(2,3);
Account object2 = new Account();
object2.setData(4,3);
}
```

How compiler will know which object (object 1 or object 2) ie has to execute



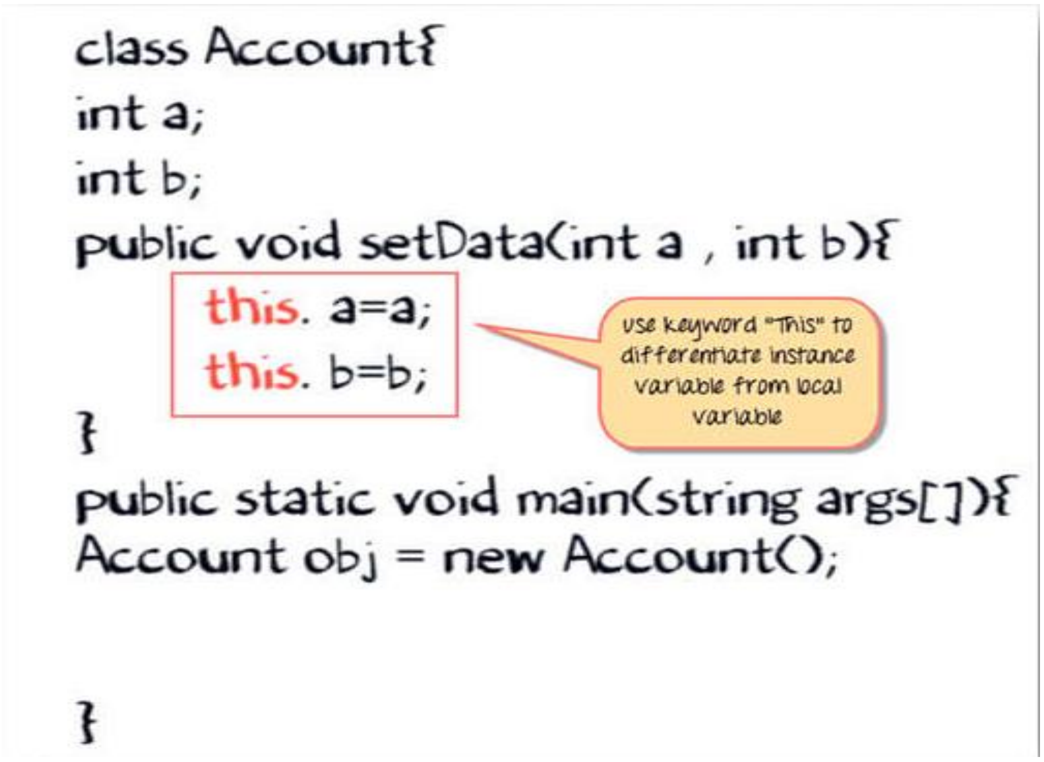
```
class Account{
int a;
int b;
public void setData(int a , int b){
        a=a;
        b=b;
}
```

Instance Variable : Set as "a" and "b"
setdata: Also Argument for set data is defined as "a" and "b"



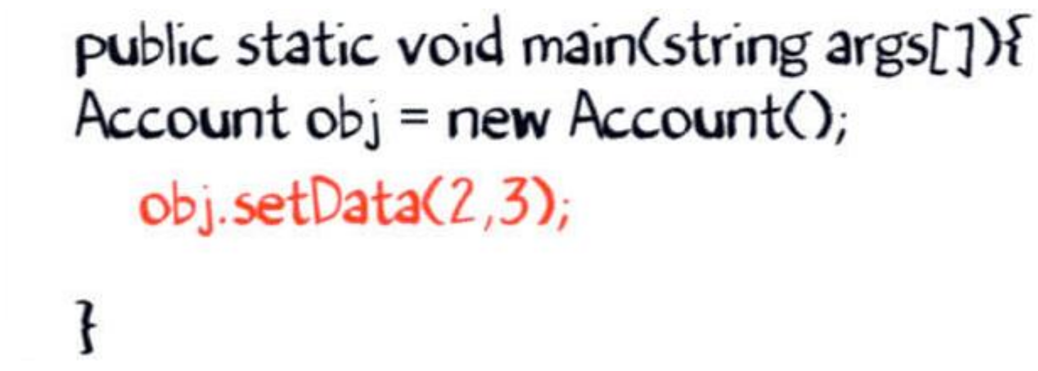
```
class Account{
int a;
int b;
public void setData(int a , int b){
     this. a=a;
     this. b=b;
}
```

use keyword "This" to differentiate instance variable from local variable

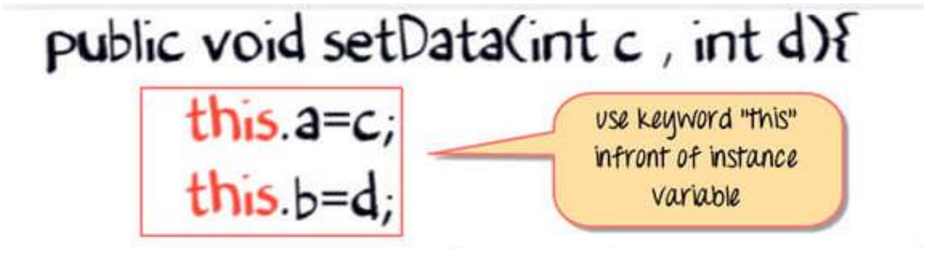


```
class Account{
int a;
int b;
public void setData(int a , int b){
        this. a=a;
        this. b=b;
}
public static void main(string args[]){
Account obj = new Account();


}
```

use keyword "This" to differentiate instance variable from local variable



```
public static void main(string args[]){
Account obj = new Account();
   obj.setData(2,3);

}
```



```
public void setData(int c , int d){
        this.a=c;
        this.b=d;
```

use keyword "this" infront of instance variable

For example, the Point class was written like this:

Example1 (Trace):

```
public class Point {
public int x = 0;
public int y = 0;
public Point(int a, int b) {
    x = a;        y = b;     }}
```

but it could have been written like this:

```
public class Point {
 public int x = 0;
 public int y = 0;
 public Point(int x, int y) {
       this.x = x;        this.y = y;
   }}
```

Each argument to the constructor shadows one of the object's fields — inside the constructor x is a local copy of the constructor's first argument. To refer to the Point field x, the constructor must use this.x.

Example2 (Trace):

```
public class Account{
private int a; int b;

public void set(int a ,int b){
a = a;
b = b; }

public void show(){
System.out.println("Value of A ="+a); System.out.println("Value of B ="+b); }
}

public class Main{
public static void main(String args[]){
Account obj = new Account();
obj.set(2,3);
obj.show(); } }
```

Example3 (Trace):
Replae the set method within the previous program to :

```
public void set(int a ,int b){
this. a = a;
this.b = b; }
```

Example4 (Trace):
Replae the set method within the previous program to :

```
public void set(int x ,int y){
this. a = x;
this.b = y; }
```

Example5 (Trace):
Replae the set method within the previous program to :

```
public void set(int x ,int y){
a = x;
b = y; }
```

2- Using this with a Constructor

From within a constructor, you can also use the this keyword to call another constructor in the same class. Doing so is called an explicit constructor invocation. Here's another Rectangle class, with a different implementation from the one in the Objects section.

Example6 (Trace):

```
public class Rectangle {
    private int x, y;
    private int width, height;
            public Rectangle() { this(0, 0, 1, 1); }

    public Rectangle(int width, int height) { this(0, 0, width, height); }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;    }
    ... }  }
```

This class contains a set of constructors. Each constructor initializes some or all of the rectangle's member variables. The constructors provide a default value for any member variable whose initial value is not provided by an argument. For example, the no-argument constructor creates a 1x1 Rectangle at coordinates 0,0. The two-argument constructor calls the four-argument constructor, passing in the width and height but always using the 0,0 coordinates. As before, the compiler determines which constructor to call, based on the number and the type of arguments.

3- Shadowed and hided Variables

One variable *shadows* another if they have the same name and are accessible in the same place.

Example7 (Trace):

```
public class One {
   private int x = 1000;
   public void set(){
       int x;
       x=100;
   System.out.println(x);

   System.out.println(((One)this).x);

((One)this).x=2000;
 System.out.println(((One)this).x);

} }

public class JavaMain {
public static void main(String[] args) {
       One obj=new One();
        obj.set();
    }}
```

The output is:

```
100
1000
2000
```

What happens if an inherited variable has the same name as a variable of the subclass? The variable of the subclass is said to hide (sometimes called shadow) the inherited variable with the same name. The inherited variable is visible in the subclass, yet it cannot be accessed by the same name.
We can said that a field is said to *hide* all fields with the same name in super classes.

But what if you need to use the inherited variable in the subclass; how can it be accessed? The answer is to use the reserved word super.

For example, if class B is a subclass of class A, and both contain a variable named **x** as follows.

Example8(Trace):

```
class A  {
protected int x; ………………. }

class B extends A {
// hide (shadow) the inherited variable common from class A
protected int x;
…………………………… }
```

Then in class B, the variable common may be referred to by either **x** or **this.x**. However, the inherited variable common is referred to by **super.x** or by **((A)this).x**. Notice that the keyword this may be cast to refer to the appropriate class, in this case class A. This technique is useful if you want to refer to a variable in a class beyond (وراء) the immediate superclass higher up the class hierarchy. Although you may refer to shadowed variables by casting an object to the appropriate type, this technique cannot be used to refer to overridden methods.

The Special Variable super ,Java also defines another special variable, named “super”, for use in the definitions of instance methods (methods within a class). The variable super is for use in a subclass. Like **"this"**, super refers to the object that contains the method. But it’s forgetful. It forgets that the object belongs to the class you are writing, and it remembers only that it belongs to the superclass of that class. The point is that the class can contain additions and modifications to the superclass. **"super"** doesn’t know about any of those additions and modifications; it can only be used to refer to methods and variables in the superclass.

Example9 (Trace):

```
 public class One {
   protected  int x ;
           public void set(){
           x=1000;
    System.out.println(x);
    System.out.println(this.x);} }

public class Two extends One {
 private int x ;
   public void print(){
   set();
   x=2000;
       System.out.println(this.x);
       System.out.println(x);
       System.out.println(((One)this).x);
       System.out.println(super.x);

       super.x=10;
       this.x=20;
   System.out.println(this.x);
   System.out.println(x);
   System.out.println(((One)this).x);
   System.out.println(super.x);

   ((One)this).x=30;
   x=40;
   System.out.println(((One)this).x);
   System.out.println(super.x);
   System.out.println(this.x);
   System.out.println(x);  }  }

public class Main {
        public static void main(String[] args) {
        Two obj=new Two();
        obj.print();     } }
```

The output will be:

```
1000
1000
2000
2000
1000
1000
20
20
10
10
30
30
40
40
```

4- Overriding Method (ألمهيمنة)

A subclass can override an inherited method by providing a new method declaration that has the same name, the same number and types of parameters and the same result type as the one inherited.

The inherited method is hidden (Shadowed) in the scope of the subclass. When the method is called for an object of the subclass, the overriding method is executed using dynamic binding.

The override method is completely re-declaring the method in the subclass but recognizing that it is a new version of the inherited method, rather than just another method. Constructors cannot be overridden. Overriding should not be confused with overloading.

Example 10 (Trace):
Consider the following program…what will be the output and why?

```
public class One {
protected int a,b;
public void set(){
   a=50;b=500;}
public void print(){
   System.out.println(a+"   "+b);}  }

public class Two extends One{
   private int x,y;
   public void set(){
      super.set();
      x=100; y=200;    }
   public void print(){
      super.print();
      System.out.println(x+"   "+y);     }
  }

public class Main {
public static void main(String[] args) {
Two t=new Two();
   t.set();      t.print();}
 }
```

The output:

```
50   500
100   200
```

When the super method call for both print and set methods are cancelled the output will be:

```
   100    200
```

Private methods cannot be overridden, so a matching method declared in a subclass is considered completely separate. Set and print methods in One class are not overridden).

Example 11(Trace):

```
public class One {
protected  int a,b;
private void set(int a,int b){
   this.a=a;this.b=b;}
private void print(){
   System.out.println(a+"   "+b);}
}
public class Two extends One{
   private int x,y;
  public  void set(int x,int y){
    this.x=x;this.y=y;           }
   public void print(){
    set(100,200);
    System.out.println(x+"   "+y);    }}

public class Main {
       public static void main(String[] args) {
   Two t=new Two();
   t.print();} }
```

The output:

```
100   200
```

If we override the set method and set the modifier to "protected" in sub class while it is public in super class, then the compiler will detect an error.

Access to the overridden method using **public, protected or the default** if no modifier, must be either the same as that of the super class method or made more accessible (change it from protected to public). An overriding method cannot be made less accessible (e.g change from public to protected).

Static method cannot be overridden. Instance methods cannot be overridden by static method. The final instance method cannot be overridden also a final static method cannot be re-declared in a subclass(**this point will be explained later**).

Example 12 (Trace):

```
public class One {
    protected int x,y;
    public One(int a,int b){
       x=a;y=b;
    System.out.println("One"+x+"   "+y);}
    public One(){
       System.out.println("One One");}
    }

public class Two extends One{
   private int x;
   public Two(){
      super(3,4);
      x=100;
           System.out.println("Two"+x+"   "+y+"    "+super.x);
           System.out.println("Two"+this.x+"   "+y+"    "+super.x);    }
}

public class Main {
   public static void main(String[] args) {
     Two t=new Two();  // TODO code application logic here
   }
}
```

Lab
Why we got the following output:

```
10   20
100   200
```

```
public class One {
protected  int a,b;
protected void set(int a,int b){
    this.a=a;this.b=b;}
protected void print(){
    System.out.println(a+"   "+b);}
}
public class Two extends One{
    private int a,b;
   public  void set(int a,int b){
     this.a=a;this.b=b;
   ((One)this).a=10;
   //or super.a=10;
   super.b=20;}
    public void print(){
       this.set(100,200);
      //or  set(100,200);
       super.print();
        System.out.println(a+"   "+b);
    }}
public class Main {
    public static void main(String[] args) {
    Two t=new Two();
    t.print();}}
```