Object Oriented Programming

Topics to be covered today

- Constructor
- Finalize() method
- this keyword
- Method Overloading
- Constructor Overloading
- Object As an Argument
- Returning Objects

```
class Box {
   double width;
   double height;
   double depth;
}
class BoxDemo {
  public static void main(String args[]) {
     Box mybox = new Box();
     double vol;

     mybox.width = 10;
     mybox.height = 20;
     mybox.depth = 15;

     vol = mybox.width * mybox.height * mybox.depth;
     System.out.println("Volume is " + vol);
  }
}
```

```
class Box {
    double width;
    double height;
    double depth;

    double volume() {
        return width * height * depth;
    }

    void setDim(double w, double h, double d) {
        width = w; height = h; depth = d;
    }
}
class BoxDemo5 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);

        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

Constructor

- A constructor initializes the instance variables of an object.
- It is called immediately after the object is created.
  - it is syntactically similar to a method:
  - it has the same name as the name of its class
  - it is written without return type; the default return type of a class constructor is the same class
- When the class has no constructor, the default constructor automatically initializes all its instance variables with zero.

Example: Constructor

```
class Box {
   double width;
   double height;
   double depth;

   Box() {
      System.out.println("Constructing Box");
      width = 10; height = 10; depth = 10;
   }

   double volume() {
      return width * height * depth;
   }
}
```

Example: Constructor

```
class BoxDemo6 {
   public static void main(String args[]) {
      Box mybox1 = new Box();
      Box mybox2 = new Box();
      double vol;

      vol = mybox1.volume();
      System.out.println("Volume is " + vol);

      vol = mybox2.volume();
      System.out.println("Volume is " + vol);
   }
}
```

- Output??
- Volume is= 1000.0
- Volume is =1000.0

Parameterized Constructor

- So far, all boxes have the same dimensions.
- We need a constructor able to create boxes with different dimensions:

```
class Box {
   double width;
   double height;
   double depth;

   Box(double w, double h, double d) {
      width = w; height = h; depth = d;
   }

   double volume() { return width * height * depth; }
}
```

Parameterized Constructor

```
class BoxDemo7 {
   public static void main(String args[]) {
      Box mybox1 = new Box(10, 20, 15);
      Box mybox2 = new Box(3, 6, 9);
      double vol;

      vol = mybox1.volume();
      System.out.println("Volume is " + vol);

      vol = mybox2.volume();
      System.out.println("Volume is " + vol);
   }
}
```

Finalize() method

- A constructor helps to initialize an object just after it has been created.
- In contrast, the finalize method is invoked just before the object is destroyed:
  - implemented inside a class as:

    protected void finalize() { … }
  - implemented when the usual way of removing objects from memory is insufficient, and some special actions has to be carried out
- How is the finalize method invoked?

Garbage Collection

- Garbage collection is a mechanism to remove objects from memory when they are no longer needed.
- Garbage collection is carried out by the garbage collector:
  - The garbage collector keeps track of how many references an object has.
  - It removes an object from memory when it has no longer any references.
  - Thereafter, the memory occupied by the object can be allocated again.
  - The garbage collector invokes the finalize method.

Keyword this

- Keyword this allows a method to refer to the object that invoked it.
- It can be used inside any method to refer to the current object:

```
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

- The above use of this is redundant but correct.
- When is this really needed?

Instance Variable Hiding

- Variables with the same names:
  - it is illegal to declare two local variables with the same name inside the same or enclosing scopes
  - it is legal to declare local variables or parameters with the same name as the instance variables of the class.
- As the same-named local variables/parameters will hide the instance variables, using this is necessary to regain access to them:

```
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

Method Overloading

- It is legal for a class to have two or more methods with the same name.
- However, Java has to be able to uniquely associate the incantation of a method with its definition relying on the number and types of arguments.
- Therefore the same-named methods must be distinguished:
  - by the number of arguments, or
  - by the types of arguments

Example: Method Overloading

```
class OverloadDemo {
   void test() {
      System.out.println("No parameters");
   }
   void test(int a) {
      System.out.println("a: " + a);
   }
   void test(int a, int b) {
      System.out.println("a and b: " + a + " " + b);
   }
   double test(double a) {
      System.out.println("double a: " + a); return a*a;
   }
}
```

Example: Method Overloading

```
class Overload {
  public static void main(String args[]) {
     OverloadDemo ob = new OverloadDemo();
     double result;
     ob.test();
     ob.test(10);
     ob.test(10, 20);
     result = ob.test(123.2);
     System.out.println("ob.test(123.2): " + result);
  }
}
```

Out Put

- No parameters
- a: 10
- a and b: 10 20
- double a: 123.25
- Result of ob.test(123.25): 15190.5625

Different Result Types

- Different return types are insufficient.
- The following will not compile:

```
double test(double a) {
   System.out.println("double a: " + a);
   return a*a;
}

int test(double a) {
   System.out.println("double a: " + a);
   return (int) a*a;
}
```

Overloading and Conversion

- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.
- When no exact match can be found, Java's automatic type conversion can aid overload resolution:

```
class OverloadDemo {
  void test() {
     System.out.println("No parameters");
  }
  void test(int a, int b) {
     System.out.println("a and b: " + a + " " + b);
  }
```

Overloading and Conversion

```
  void test(double a) {
    System.out.println("Inside test(double) a: " + a);
  }
}

class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    int i = 88;
    ob.test();
    ob.test(10, 20);
    ob.test(i);
    ob.test(123.2);
  }
}
```

Overloading and Polymorphism

- In the languages without overloading, methods must have a unique names:

```
int abs(int i)
long labs(int i)
float fabs(int i)
```

- Java enables logically-related methods to occur under the same name:

Constructor Overloading

- Why overload constructors? Consider this:

```
class Box {
   double width, height, depth;

   Box(double w, double h, double d) {
      width = w; height = h; depth = d;
   }
   double volume() {
      return width * height * depth;
   }
}
```

- All Box objects can be created in one way: passing all three dimensions.

Example: Overloading

- Three constructors: 3-parameter, 1-parameter, parameter-less.

```
class Box {
   double width, height, depth;
   Box(double w, double h, double d) {
      width = w; height = h; depth = d;
   }
   Box() {
      width = -1;  height = -1;  depth = -1;
   }
   Box(double len) {
      width = height = depth = len;
   }
   double volume() { return width * height * depth; }
}
```

Example: Overloading

```
class OverloadCons {
  public static void main(String args[]) {
     Box mybox1 = new Box(10, 20, 15);
     Box mybox2 = new Box();
     Box mycube = new Box(7);
     double vol;

     vol = mybox1.volume();
     System.out.println("Volume of mybox1 is " + vol);
     vol = mybox2.volume();
     System.out.println("Volume of mybox2 is " + vol);
     vol = mycube.volume();
     System.out.println("Volume of mycube is " + vol);
  }
}
```

Object Argument

- So far, all method received arguments of simple types.
- They may also receive an object as an argument. Here is a method to check if a parameter object is equal to the invoking object:

```
class Test {
   int a, b;
   Test(int i, int j) {
      a = i; b = j;
   }
   boolean equals(Test o) {
      if (o.a == a && o.b == b) return true;
      else return false;
   }
}
```

Object Argument

```
class PassOb {
  public static void main(String args[]) {
     Test ob1 = new Test(100, 22);
     Test ob2 = new Test(100, 22);
     Test ob3 = new Test(-1, -1);
     System.out.println("ob1==ob2: " + ob1.equals(ob2));
     System.out.println("ob1==ob3: " + ob1.equals(ob3));
  }
}
```

Passing object to Constructor

- A special case of object-passing is passing an object to the constructor.
- This is to initialize one object with another object:

```
class Box {
   double width, height, depth;

   Box(Box ob) {
      width = ob.width;
      height = ob.height;
      depth = ob.depth;
   }
```

```
  Box(double w, double h, double d) {
     width = w;
     height = h;
     depth = d;
  }

  double volume() {
     return width * height * depth;
  }
}
class OverloadCons2 {
  public static void main(String args[]) {
     Box mybox1 = new Box(10, 20, 15);
     Box mybox2 = new Box(mybox1);
     double vol;

     vol = mybox1.volume();
     System.out.println("Volume of mybox1 is " + vol);

     vol = mybox2.volume();
     System.out.println("Volume of mybox2 is " + vol);
  }
}
```

Argument Passing

- Two types of variables:
  - simple types
  - class types
- Two corresponding ways of how the arguments are passed to methods:
  - by value a method receives a copy of the original value;parameters of simple types
  - by reference a method receives the memory address of the original value, not the value itself; parameters of class types

Simple Type Argument Passing

- Passing arguments of simple types takes place by value:

```
class Test {
  void meth(int i, int j) {
    i *= 2;
    j /= 2;
  }
}
```

Simple Type Argument Passing

- With by-value argument-passing what occurs to the parameter that receives the argument has no effect outside the method:

```
class CallByValue {
  public static void main(String args[]) {
     Test ob = new Test();
     int a = 15, b = 20;
     System.out.print("a and b before call: “);
     System.out.println(a + " " + b);
     ob.meth(a, b);
     System.out.print("a and b after call: ");
     System.out.println(a + " " + b);
  }
}
```

Class Type Argument Passing

- Objects are passed to the methods by reference: a parameter obtains the same address as the corresponding argument:

```
class Test {
   int a, b;

   Test(int i, int j) {
      a = i; b = j;
   }

   void meth(Test o) {
      o.a *=  2; o.b /= 2;
   }
}
```

Class Type Argument Passing

- As the parameter hold the same address as the argument, changes to the object inside the method do affect the object used by the argument:

```
class CallByRef {
  public static void main(String args[]) {
     Test ob = new Test(15, 20);
     System.out.print("ob.a and ob.b before call: ");
     System.out.println(ob.a + " " + ob.b);
     ob.meth(ob);
     System.out.print("ob.a and ob.b after call: ");
     System.out.println(ob.a + " " + ob.b);
  }
}
```

Returning Objects

- So far, all methods returned no values or values of simple types.
- Methods may also return objects:

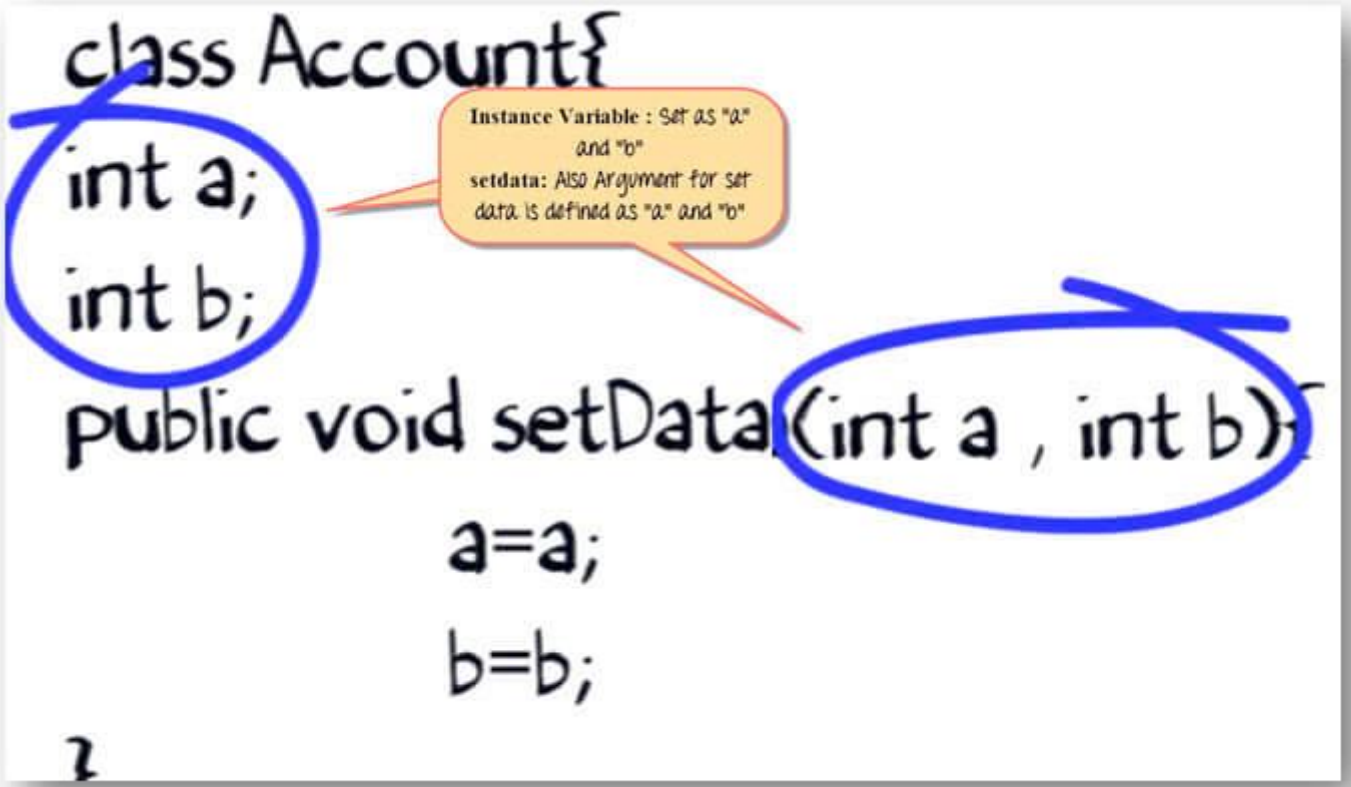
```
class Test {
   int a;
   Test(int i) {
      a = i;
   }
   Test incrByTen() {
      Test temp = new Test(a+10);
      return temp;
   }
}
```

Returning Objects

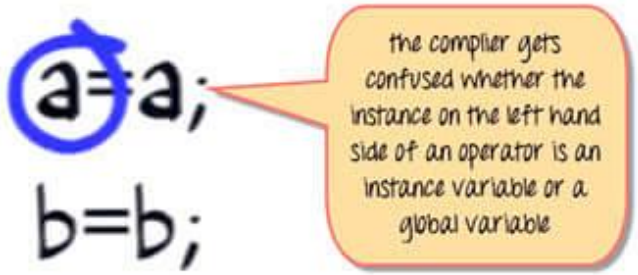
- Each time a method **incrByTen** is invoked a new object is created and a reference to it is returned:

```
class RetOb {
  public static void main(String args[]) {
     Test ob1 = new Test(2);
     Test ob2;
     ob2 = ob1.incrByTen();
     System.out.println("ob1.a: " + ob1.a);
     System.out.println("ob2.a: " + ob2.a);
     ob2 = ob2.incrByTen();
     System.out.print("ob2.a after second increase: ");
     System.out.println(ob2.a);
  }
}
```

This Keyword

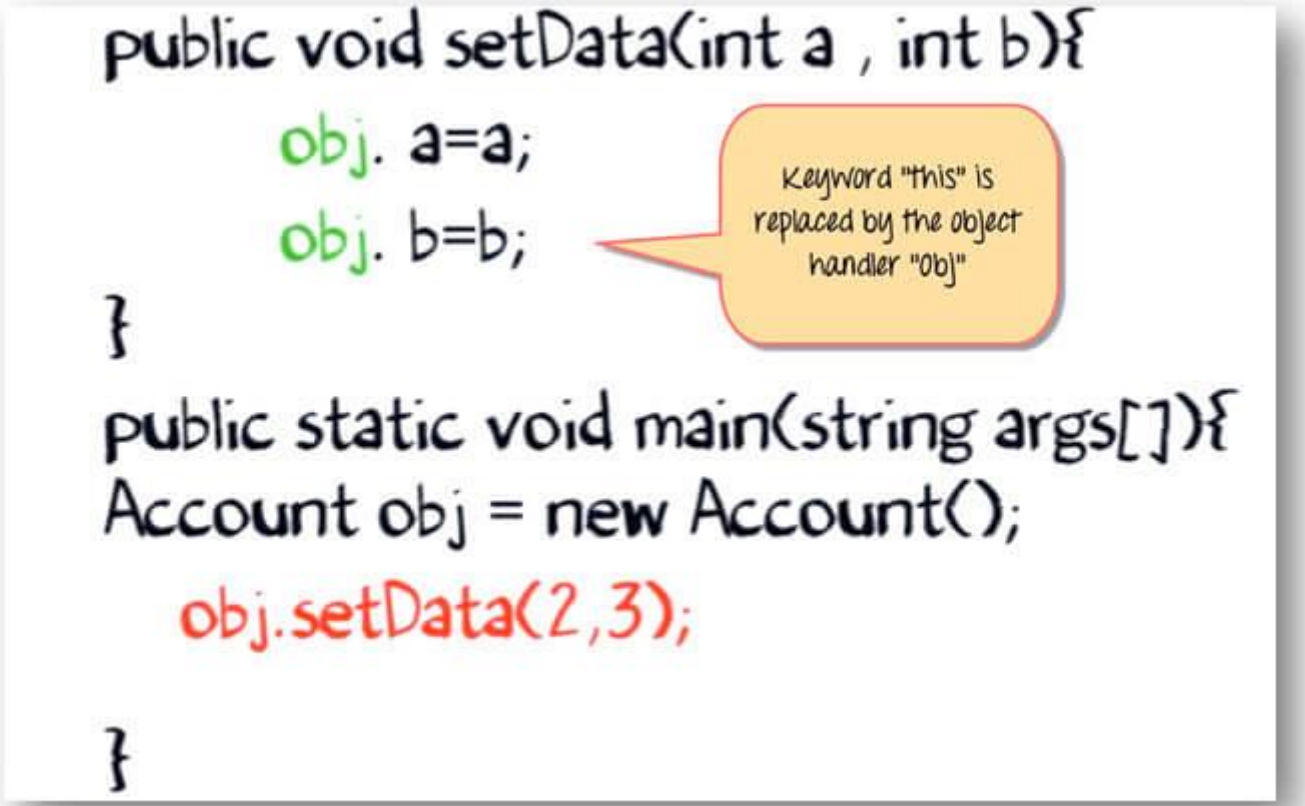


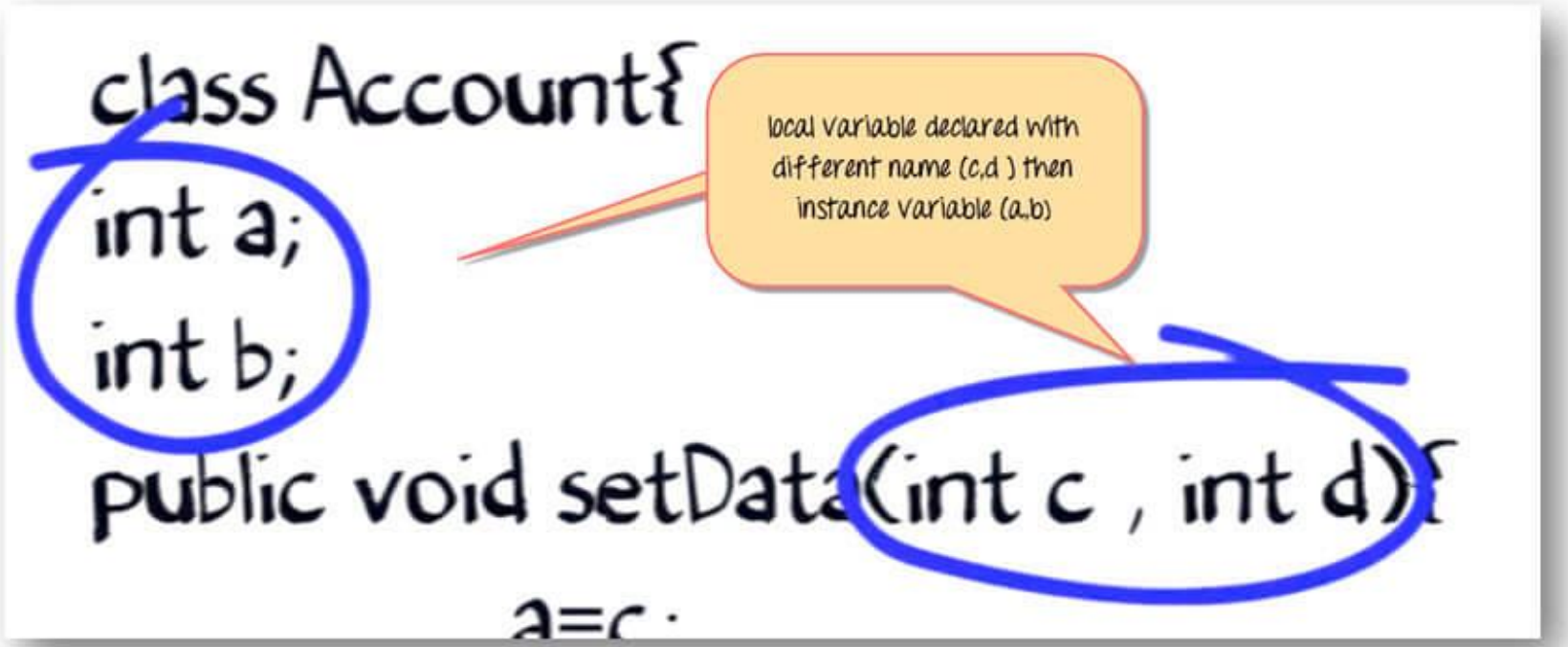
```
class Account{
int a;
int b;
public void setData(int a , int b){
    a=a;
    b=b;
}
```

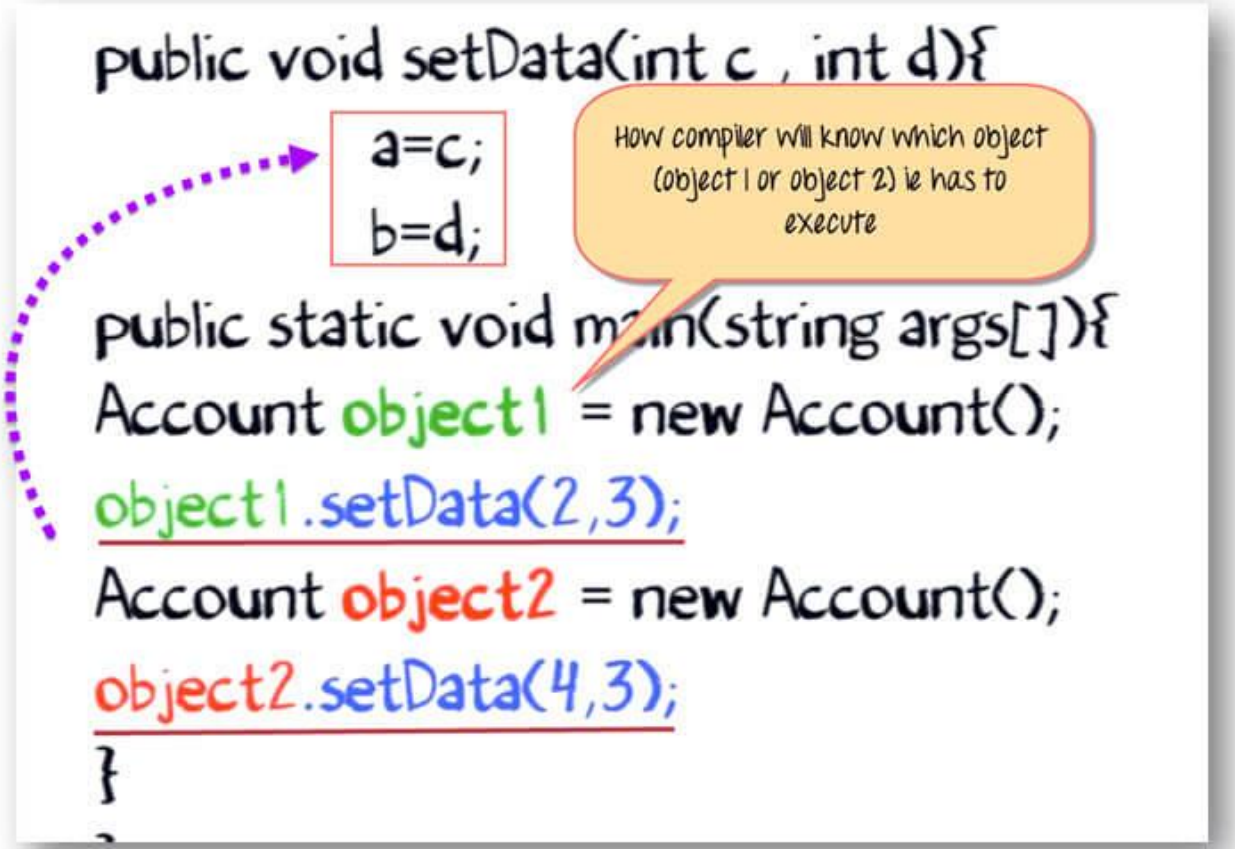


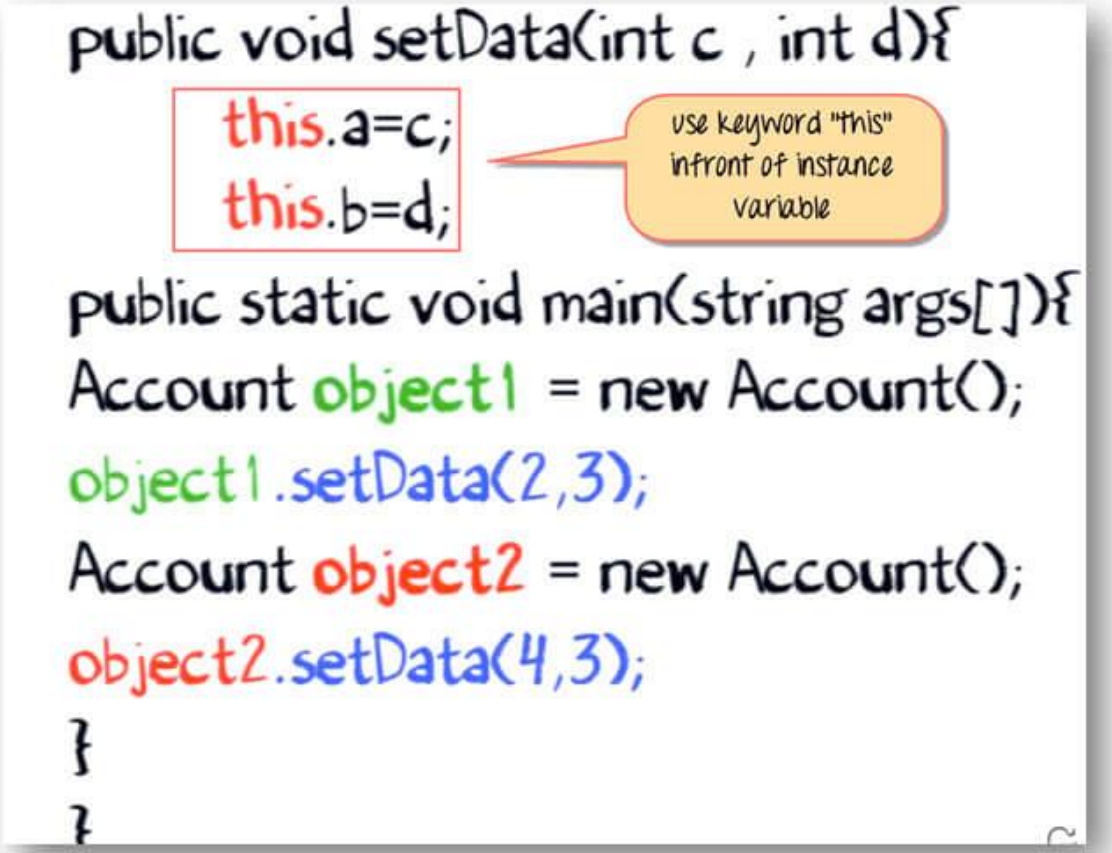
```
class Account{
int a;
int b;
public void setData(int a , int b){
        this. a=a;
        this. b=b;
}
public static void main(string args[]){
Account obj = new Account();


}
```

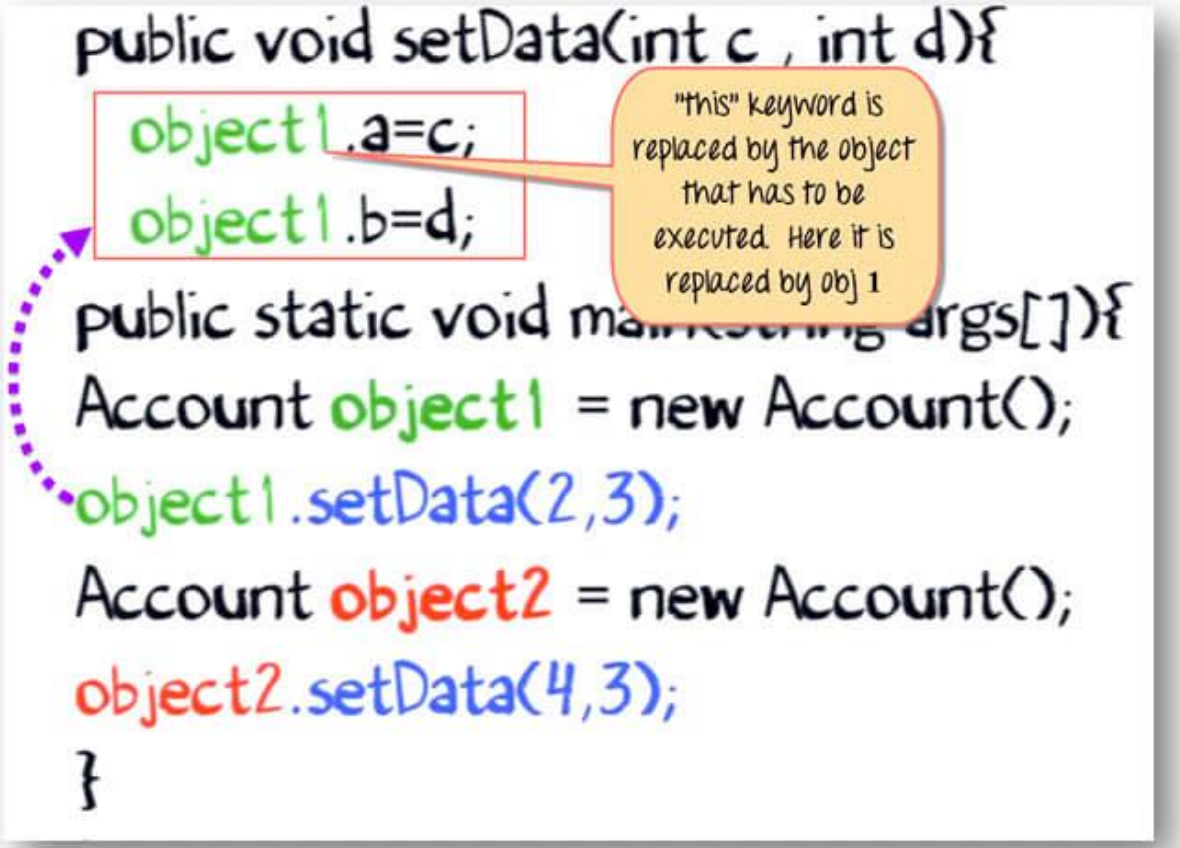
use keyword "This" to differentiate instance variable from local variable

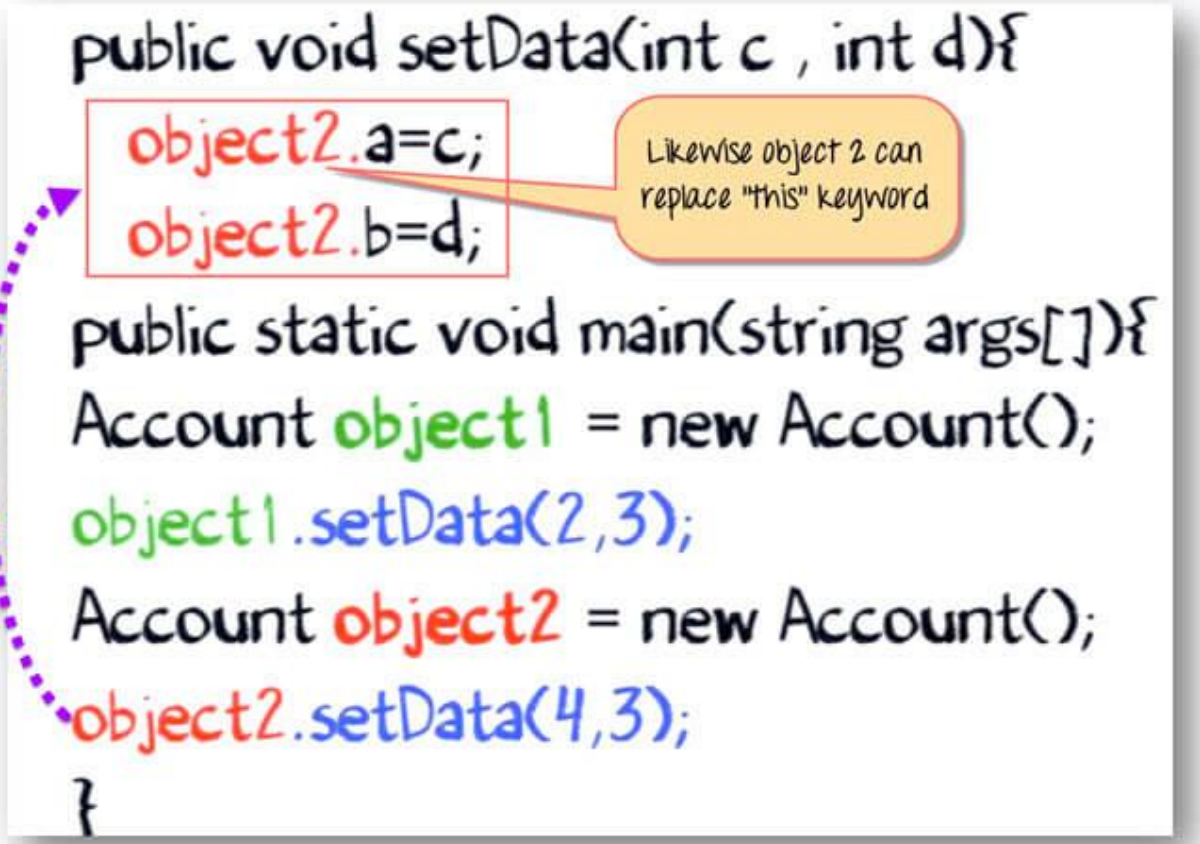

public void setData(int a , int b){
obj. a=a;
obj. b=b;
}
public static void main(string args[]){
Account obj = new Account();
obj.setData(2,3);
}
Keyword "this" is replaced by the object handler "obj"


class Account{
int a;
int b;
local variable declared with different name (c,d) then instance variable (a,b)
public void setData(int c , int d){
a=c ;


public void setData(int c , int d){
a=c;
b=d;
How compiler will know which object (object 1 or object 2) ie has to execute
public static void main(string args[]){
Account object1 = new Account();
object1.setData(2,3);
Account object2 = new Account();
object2.setData(4,3);
}


public void setData(int c , int d){
this.a=c;
this.b=d;
use keyword "this" infront of instance variable
public static void main(string args[]){
Account object1 = new Account();
object1.setData(2,3);
Account object2 = new Account();
object2.setData(4,3);
}
}


public void setData(int c , int d){
object1.a=c;
object1.b=d;
"this" keyword is replaced by the object that has to be executed. Here it is replaced by obj 1
public static void ma
rgs[]){
Account object1 = new Account();
object1.setData(2,3);
Account object2 = new Account();
object2.setData(4,3);
}


public void setData(int c , int d){
object2.a=c;
object2.b=d;
Likewise object 2 can replace "this" keyword
public static void main(string args[]){
Account object1 = new Account();
object1.setData(2,3);
Account object2 = new Account();
object2.setData(4,3);
}

```
class Account{
int a;
int b;
 public void setData(int a ,int b){
  a = a;
  b = b; }
 public void showData(){
   System.out.println("Value of A ="+a);
   System.out.println("Value of B ="+b);}
 public static void main(String args[]){
   Account obj = new Account();
   obj.setData(2,3);
   obj.showData();
  }
 }
```

- **this** Keyword in Java is a reference variable that refers to the current object.
- One of the use of this keyword in Java is to refer current class instance variable
- It can be used to invoke or initiate current class constructor
- It can be passed as an argument in the method call
- this pointer in Java can be passed as argument in the constructor call
- this operator in Java can be used to return the current class instance
- this in Java is a reference to the current object, whose method is being called upon.
- You can use "this" keyword to avoid naming conflicts in the method/constructor of your instance/object

Java final keyword

```
class A {
final int a;
void f(final int b) {
        a=2;b=5;
}}
```

final Variable
Can't be Modified

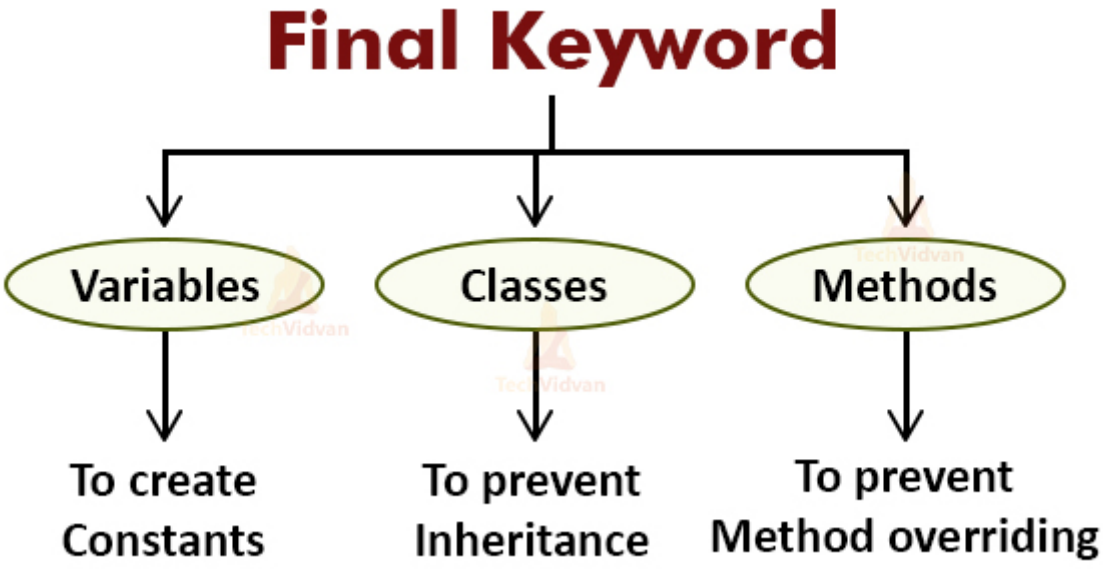
```
final class A{}
class B extends A{}
```

final class
Can't be Extended

```
class A{
        final void f() {}
}
class B extends A{
        void f() {}
}
```

final method
Can't be Overridden


Java Final Keyword
Stop Value Change
Stop Method Overridding
Stop Inheritance
javatpoint.com


Final Keyword
Variables
Classes
Methods
To create Constants
To prevent Inheritance
To prevent Method overriding

Questions