SLIIT
Discover Your Future

Object Oriented Programming

Week 03

A few Things specific to Java

Learning Outcomes

At the end of the Lecture students should be able to get details of a few specific things related to the Java Programming Language.

- Primitive Data types
- Object Memory Allocation
- Object variables are reference type parameters
- Static properties and methods
- Final keyword
- Passing Objects as parameters
- Returning Objects
- Overloading vs Overriding

The Primitive Types

- Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**.
  - Integers This group includes byte, short, int, and long, which are for whole-valued signed numbers.
  - Floating-point numbers This group includes float and double, which represent numbers with fractional precision.
  - Characters This group includes char, which represents symbols in a character set, like letters and numbers.
  - Boolean This group includes boolean, which is a special type for representing true/false values.

SLIIT
Discover Your Future

Integer Data Type

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

- Unlike languages like C, C++ the sizes of Integers, Floats are fixed and are platform neutral.

Float Data Type

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

- Unlike languages like C, C++ the sizes of Integers, Floats are fixed and are platform neutral.

Type Conversion and Casting

- Auto Conversion happens in Java for Simple Data Types in situations
  - The two types are compatible.
  - The destination type is larger than the source type.
- Manual Casting is required when the destination is smaller than the source type.
  - e.g.

```
double d = 56.0;
int no = (int) d;
```

One Dimensional Arrays

- Slightly different from C/C++
  - Define Array
  - Allocate Memory

```
int data[]; // Array declaration
data = new int[10]; // Allocating Memory
```

Objects and Memory

- Lets consider how an Object is allocated in memory.

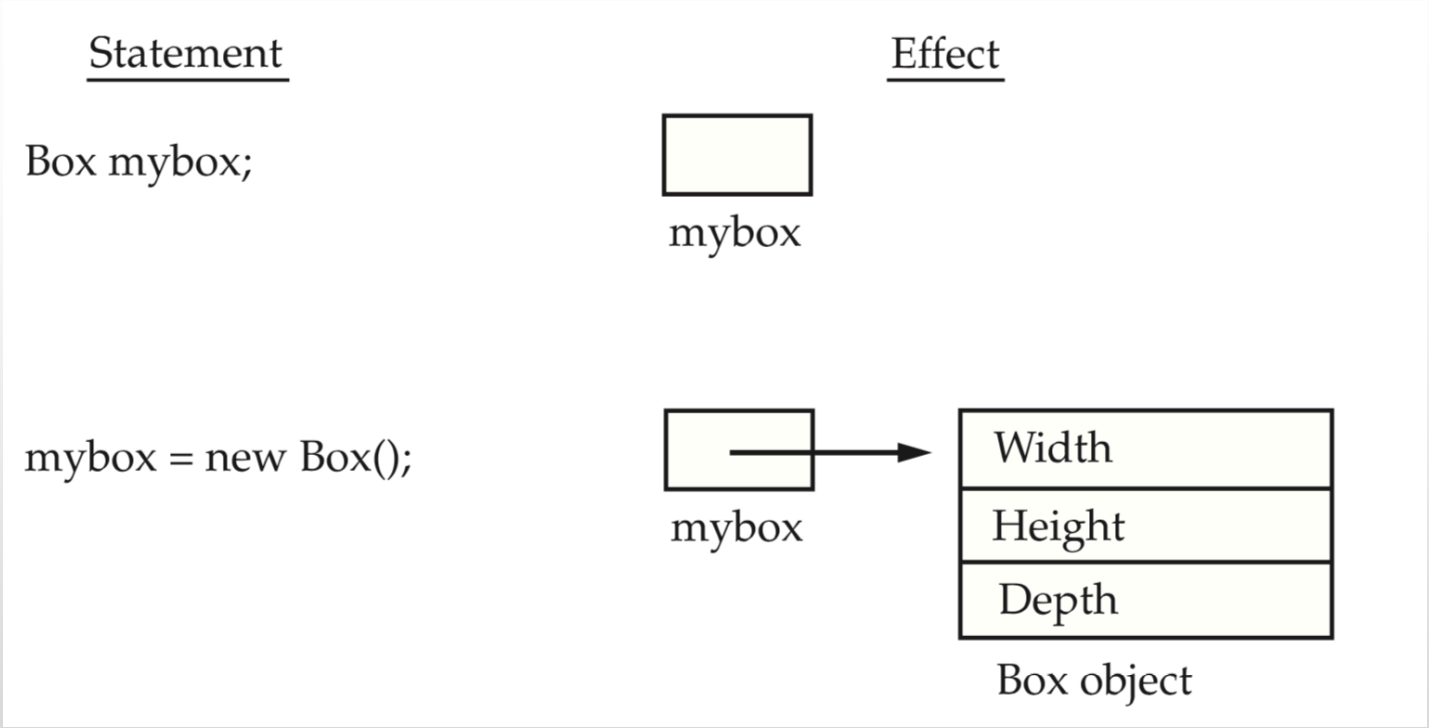
```
class Box {
  double width;
  double height;
  double depth;
}

class BoxDemo2 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol
```

Objects and Memory

- Declaring Objects in Java is a two step process.
  - First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object.
  - Second, you must use the **new** operator to dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**.
  - This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

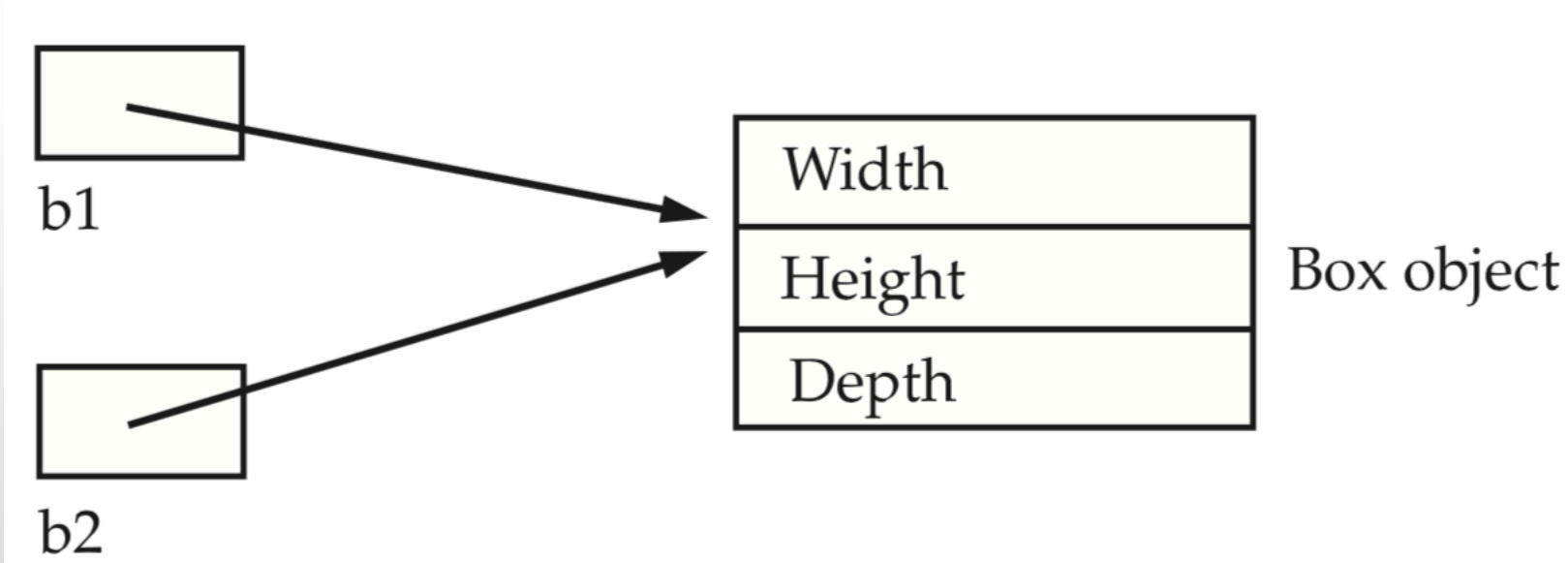
Objects and Memory

- Lets consider how an Object is allocated in memory.



Objects and Memory

```
Box b1 = new Box();
Box b2 = b1;
```



Static Members

- Attributes and methods (members of a class) can be defined as static.
- Static members do not belongs to an individual object.
- Static members are common to all the instances (objects of the same class).
- Static members are stores in static memory (a common memory location which can by everybody)

Static Members

```
class Student {
    private String ditno;
    private String name;
    private static String batchId;

    public Student(String mditno, String mname) {
        ditno = mditno;
        name = mname;
    }
    public void setBatchId(String mbatchId) {
        batchId = mbatchId;
    }
```

StaticDemo.java

Static Members

```
public static void setBatchId2(String mbatchId) {
    batchId = mbatchId;
}
```

- setBatchId2() is a static method
- Static Methods can be called directly using the class name.

```
Student.setBatchId2("Metro Y1B1");
```

SLIIT
Discover Your Future

Static Members

Metro Y1B1

IT15123412
Tharidi

IT15132343
Kumudu

```
public class Static {
    public static void main(String args[]) {
        Student s1 = new Student("IT15123412", "Tharidi");
        Student s2 = new Student("IT15132343", "Kumudu");
        s1.setBatchId("Malabe - Y1B2");
        System.out.println(s1.getBatchId() + " -  " + s1.getDitNo()
        System.out.println(s2.getBatchId() + " -  " + s2.getDitNo()
        Student.setBatchId2("Metro Y1B1");
        System.out.println(s1.getBatchId() + " -  " + s1.ge
        System.out.println(s2.getBatchId() + " -  " + s2.ge
```

StaticDemo.java

In this code the methods setBatchId() and setBatchId2() both change the static variable batchId. This is common to both objects s1 and s2 and stored in the stack where as the instance variables name and ditno are different for each object.

Static Modifiers

- The static modifier indicates that the attributes and methods are common to all the object in the whole class rather than to an individual object.
- A static method does not operate on an object:
- ClassName.methodName(parameterList)
- Many attributes and methods we use are static:

System.out	Integer.parseInt()
System.in	Character.isLetter()
Color.RED	Math.random()
Math.PI	

Understanding Static

- There will be times when you will want to define a class member that will be used independently of any object of that class.
- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be **static**.
- The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.

Understanding Static

- Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.
- Methods declared as **static** have several restrictions:
  - They can only directly call other **static** methods.
  - They can only directly access **static** data.
  - They cannot refer to **this** or **super** in any way.

SLIIT
Discover Your Future

Static Demo (JTCR pg 145)

```
class Main {
  static int a = 3;
  static int b;
  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }
  static {
    System.out.println("Static block initialized.");
    b = a * 4;
  }
  public static void main(String args[]) {
    meth(42);
  }
}
```

StaticDemo2.java

The static variables and the code is executed when the class is loaded.

final properties

- Final is used to declare constants.

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

- These can be declared as above in the class or initialized in the constructor.
- A static final variable is a global constant.

SLIIT
Discover Your Future

Passing Object as a Parameter (JTCR pg 138)

```
class Test {
   int a, b;
  Test(int i, int j) {
    a = i;
    b = j;
  }
  // return true if o is equal to the invoking object
  boolean equalTo(Test o) {
    if(o.a == a && o.b == b) return true;
    else return false;
  }
}
```

PassingObjects.java

Here a Test object is passed as a parameter to the equalTo() method

SLIIT
Discover Your Future

Object Parameter to a Constructor (JTCR pg 135)

```
// Here, Box allows one object to in
class Box {
  double width;
  double height;
  double depth;
  // Notice this constructor. It take
  Box(Box ob) { // pass object to co
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
}
```

ObjectConstructor.java

Here a Box type object is passed to the overloaded Box constructor.

SLIIT
Discover Your Future

Returning an Object (JTCR pg 138)

```
// Returning an object.
class Test {
    int a;
    Test(int i) {
      a = i;
    }
    Test incrByTen() {
      Test temp = new Test(a+10);
      return temp;
    }
}
```

ReturnObjects.java

A Test object is created and returned by the method incrByTen()

SLIIT Discover Your Future

Passing Objects as References (JTCR pg 175)

```
// Objects are passed throu
class Test {
  int a, b;
  Test(int i, int j) {
    a = i;
    b = j;
  }
  // pass an object
  void meth(Test o) {
    o.a *= 2;
    o.b /= 2;
  }
}
```

ObjectReference.java

In the meth() method the parameter o is an object, any changes will affect the arguments that are sent.

Overloading

- Overloading occurs when there are methods with different signatures.

Overloading Demo (JTCR pg 129)

```
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }
  // Overload test for one integer parameter.
  void test(int a) {
    System.out.println("a: " + a);
  }
  // Overload test for two integer parameters.
  void test(int a, int b) {
     System.out.println("a and b: " + a + " " + b);
  }
  // Overload test for a double parameter
  double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}
```

Overloading.java

The test() method is overloaded.

Overriding

- Overriding occurs in inheritance when a descendant class replaces a method with the same signature.

SLIIT
Discover Your Future

Overriding Demo (JTCR pg 175)

```
class B extends A {
        int k;
        B(int a, int b, int c) {
          super(a, b);
          k = c;
        }
        // display k – this overrides show() in A
        void show() {
          System.out.println("k: " + k);
        }
}
```

Override.java

The show() method in class B overrides the show() method in the class A

SLIIT
Discover Your Future

References

- JTCR – Java the Complete Reference, Herbert Schildt, 9th Edition, Oracle Press.