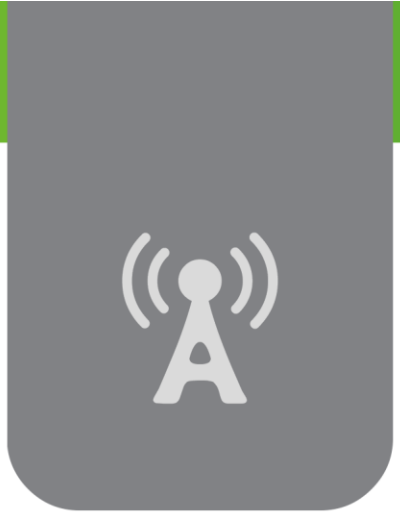


Chapter 2:
Object Oriented Programming in Java

Title:

Object Oriented Programming in Java

Key Words:

Classes and Objects, Inheritance, Polymorphism.

Summary:

This unit provides an overview of the object oriented programming in Java.

Outcomes:

Student will learn in this unit:

- Declaring classes.
- Using polymorphism and inheritance.

Plan:

3 Learning Objects

1. Object Oriented Programming in Java
2. Inheritance
3. Advanced Topics in OOP

1. Object Oriented Programming in Java

Learning outcomes:

Learning main object oriented programming using Java.

Classes and objects:

- **class**: A program entity that represents either:
  - A program / module, or
  - A template for a new type of objects.
  - The Point class is a template for creating Point objects
- object: An entity that combines state and behavior.
  - object-oriented programming (OOP): Programs that perform their behavior as interactions between objects.

Fields:

- **field**: A variable inside an object that is part of its state.
  - Each object has *its own copy* of each field.
  - **encapsulation**: Declaring fields `private` to hide their data.
- Declaration syntax:

```
private type name
```

- Example:

```
public class Student {
   private String name; // each object now has
   private double gpa;  // a name and gpa field
}
```

Instance methods:

- **instance method**: One that exists inside each object of a class and defines behavior of that object.

```
public type name(parameters) { statements;
}
```

- Example:

```
public void shout() {
    System.out.println("HELLO THERE!");
}
```

A Point class:

```
public class Point
{
   private int x;
   private int y;
   // Changes the location of this Point object.
    public void draw(Graphics g) {
        g.fillOval(x, y, 3, 3);
        g.drawString("(" + x + ", " + y + ")", x, y);
    }
}
```

- Each `Point` object contains data fields named `x` and `y`.
- Each `Point` object contains a method named `draw` that draws that point at its current `x/y` position.

The implicit parameter:

- implicit parameter:

  The object on which an instance method is called.

  - During the call `p1.draw(g);`

    the object referred to by p1 is the implicit parameter.

  - During the call `p2.draw(g);`

    the object referred to by `p2` is the implicit parameter.

  - The instance method can refer to that object's fields.

    We say that it executes in the *context* of a particular object.

    `draw` can refer to the `x` and `y` of the object it was called on

Kinds of methods:

- Instance methods take advantage of an object's state.
  - Some methods allow clients to access / modify its state.
- **accessor**: A method that lets clients examine object state.
  - Example: A distanceFromOrigin method that tells how far a Point is away from $(0, 0)$.
  - Accessors often have a non-void return type.
- **mutato**r: A method that modifies an object's state.
  - Example: A `translate` method that shifts the position of a `Point` by a given amount.

Constructors:

- **`constructor:`** Initializes the state of new objects.

```
public type(parameters) {
        statements;
}
```

- Example:

```
public Point(int initialX, int initialY)
        { x = initialX;
y = initialY;
```

  - runs when the client uses the new keyword
  - does not specify a return type; implicitly returns a new object
  - If a class has no constructor, Java gives it a *default constructor* with no parameters that sets all fields to 0.

toString method:

- Tells Java how to convert an object into a `String`

```
public String toString() {
        code that returns a suitable String;
}
```

- Example:

```
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

- Called when an object is printed / concatenated to a `String`:

```
    Point p1 = new Point(7, 2);
    System.out.println("p1: " + p1);
```

- Every class has a `toString`, even if it isn't in your code.
  - Default is class's name and a hex number: `Point@9e8c34`

this keyword:

- **`this`:** A reference to the implicit parameter.
  - *implicit parameter*: object on which a method is called
- Syntax for using `this`:
  - To refer to a field:
    `this.field`
  - To call a method:
    `this.method(parameters);`
  - To call a constructor from another constructor:
    `this(parameters);`

Static methods:

- **static method**: Part of a class, not part of an object.
  - shared by all objects of that class
  - good for code related to a class but not to each object's state
  - does not understand the *implicit parameter*; this; therefore, cannot access an object's fields directly
  - if public, can be called from inside or outside the class
- Declaration syntax:

```
public static type name(parameters) {
statements;
 }
```

2. Inheritance

Learning outcomes: using inheritance:

- **inheritance**: A way to form new classes based on existing classes, taking on their attributes / behavior.
  - a way to group related classes
  - a way to share code between two or more classes
- One class can *extend* another, absorbing its data / behavior.
  - **superclass**: The parent class that is being extended.
  - **subclass**: The child class that extends the superclass and inherits its behavior.
  - Subclass gets a copy of every field and method from superclass

Inheritance syntax:

```
public class name extends superclass {
```

- Example:

```
public class Secretary extends Employee {
...
}
```

Overriding methods

- **override**: To write a new version of a method in a subclass that replaces the superclass's version.
- No special syntax required to override a superclass method. Just write a new version of it in the subclass.

```
public class Secretary extends Employee {
        // overrides getVacationForm in
        Employee public String
        getVacationForm() {
                return "pink";
        }
        ...
}
```

super keyword:

- Subclasses can call overridden methods with super

  `super.`**method(parameters)**

- Example:

```
public class LegalSecretary extends Secretary {
    public double getSalary() {
          double baseSalary =
          super.getSalary();
          return baseSalary + 5000.0;
    }
    ...
```

Polymorphism:

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.
- Example: `System.out.println` can print any type of object.
  - Each one displays in its own way on the console.
- A variable of type *T* can hold an object of any subclass of T.

  `Employee ed` `= new LegalSecretary();`
  - You can call any methods from `Employee` on ed.
  - You can *not* call any methods specific to `LegalSecretary`.
- When a method is called, it behaves as a `LegalSecretary`.

```
System.out.println(ed.getSalary()); //55000.0
System.out.println(ed.getVacationForm()) //pink
```

3. Advanced Topics in OOP

Learning outcomes: Learning some advanced topics in OOP:

- **inheritance**: Forming new classes based on existing ones.
  - **superclass**: Parent class being extended.
  - **subclass**: Child class that inherits behavior from superclass.
  - gets a copy of every field and method from superclass
- **override**: To replace a superclass's method by writing a new version of that method in a subclass.

```
public class Lawyer extends Employee {
     // overrides getSalary in Employee; a raise!
     public double getSalary() {
            return 55000.00;
     }
}
```

The super keyword:

- Syntax:

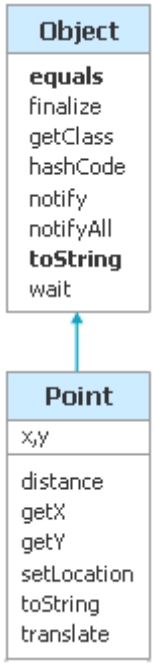
  `super.method(parameters)`

  `super (parameters);`

- Subclasses can call overridden methods/constructors with `super`

```
public class Lawyer extends Employee
   { private boolean passedBarExam;
   public Lawyer(int vacationDays, boolean bar)
        { super(vacationDays * 2);
          this.passedBarExam = bar;
   }
   public double getSalary() {
          double baseSalary = super.getSalary();
          return baseSalary + 5000.00;    // $5K raise
   }
   ...
}
```

The class Object:



- The class Object forms the root of the overall inheritance tree of all Java classes.
  - Every class is implicitly a subclass of `Object`
- The `Object` class defines several methods that become part of every class you write. For example:
  - public String `toString()`

  Returns a text representation of the object, usually so that it can be printed.

Object methods:

method	description
`protected Object clone()`	creates a copy of the object
`public boolean equals(Object o)`	returns whether two objects have the same state
`protected void finalize()`	used for garbage collection
`public Class<?> getClass()`	info about the object's type
`public int hashCode()`	a code suitable for putting this object into a hash collection
`public String toString()`	text representation of object

`public void notify()`
`public void notifyAll()`
`public void wait()`
`public void wait()`
	methods related to concurrency and locking

Using the Object class:

- You can store any object in a variable of type Object.

```
Object o1 = new Point(5, -3);
Object o2 = "hello there";
```

- You can write methods that accept an Object parameter.

```
public void
      checkNotNull(Object
      o) { if (o != null) {
            throw new IllegalArgumentException();
    }
```

- You can make arrays or collections of `Objects`.

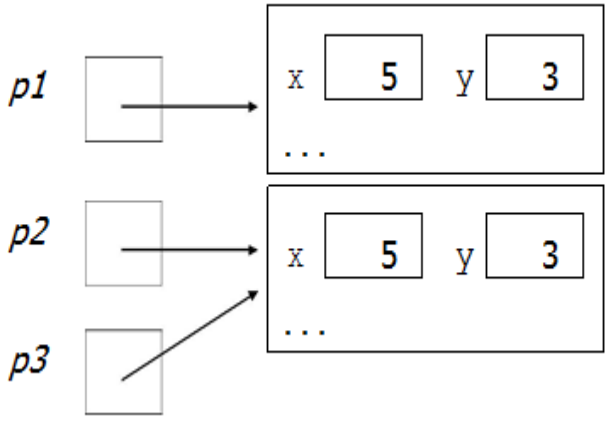
```
Object[] a = new
Object[5]; a[0] = "hello";
a[1] = new Random();
List<Object> list = new ArrayList<Object>();
```

Recall: comparing objects:

- The == operator does not work well with objects.
  - It compares references, not objects' state.
  - It produces true only when you compare an object to itself.

```
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
Point p3 = p2;


// p1 == p2 is false;
// p1 == p3 is false;
// p2 == p3 is true
// p1.equals(p2)?
// p2.equals(p3)?
```



Default equals method:

- The `Object` class's `equals` implementation is very simple:

```
public class Object {
        ...
        public booleanequals(Object
                o) { return this ==
                o;
        }
}
```

- However:
  - When we have used equals with various objects, it didn't behave like == .Why not? `if (str1.equals(str2)) { ...`
  - The Java API documentation for equals is elaborate. Why?

Implementing equals:

- Syntax:

```
public boolean equals(Object name) {
        statement(s) that return a boolean value ;
}
```

- The parameter to `equals` must be of type `Object`.
- Having an `Object` parameter means *any* object can be passed.
  - If we don't know what type it is, how can we compare it?

Casting references:

```
Object o1 = new Point(5, -3);
Object o2 = "hello there";

((Point) o1).translate(6, 2);          // ok
int len = ((String) o2).length();      // ok
Point p = (Point) o1;
int x = p.getX();                  // ok
```

- Casting references is different than casting primitives.
  - Really casting an `Object` reference into a `Point` reference.
  - Doesn't actually change the object that is referred to.
  - Tells the compiler to assume that `o1` refers to a `Point` object.

The instanceof keyword:

- Syntax:

```
if (variable instanceof type) {
        statement(s);
        }
```

  - Asks if a variable refers
  - to an object of a given type.
    - Used as a `boolean` test.

```
String s = "hello";
Point p = new Point();
```

expression	result
`s instanceof Point`	false
`s instanceof String`	true
`p instanceof Point`	true
`p instanceof String`	false
`p instanceof Object`	true
`s instanceof Object`	true
`null instanceof String`	false
`null instanceof Object`	false

equals method for Points:

```
//   Returns whether o refers to a Point object with
//   the same (x, y) coordinates as this Point.
boolean equals(Object o) { if (o instanceof Point) {
             //   o is a Point; cast and compare it Point
             other = (Point) o;
             return x == other.x && y == other.y; } else {
             //   o is not a Point; cannot be equal
             return false;
       }
}
```

More about equals:

- Equality is expected to be reflexive, symmetric, and transitive:

  `a.equals(a)` is true for every object

  `a a.equals(b)` ↔ `b.equals(a)`

  `(a.equals(b) && b.equals(c))` ↔ `a.equals(c)`

- No non-null object is equal to null:

  `a.equals(null)` is false for every object a

- Two sets are equal if they contain the same elements:

```
Set<String> set1 = new HashSet<String>();
Set<String> set2 = new TreeSet<String>();
for(String s: "hi how are you".split(" ")) {
   set1.add(s); set2.add(s);
}
System.out.println(set1.equals(set2));   // true
```

The hashCode method:

`public int hashCode()`

Returns an integer hash code for this object, indicating its preferred to place it in a hash table / hash set.

- Allows us to store non-`int` values in a hash set / map:

```
public static int hashFunction(Object o) {
   return Math.abs(o.hashCode()) % elements.length;
}
```

- How is `hashCode` implemented?
  - Depends on the type of object and its state.

    Example: a String's `hashCode` adds the ASCII values of its letters.
  - You can write your own `hashCode` methods in classes you write.

    All classes come with a default version based on memory address.

Polymorphism:

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.
- A variable or parameter of type T can refer to any subclass of T.

```
Employee ed = new Lawyer();
Object otto = new Secretary();
```

- When a method is called on `ed`, it behaves as a `Lawyer`.
- You can call any `Employee` methods on `ed`. You can call any `Object` methods on `otto`.
  - You can *not* call any `Lawyer`- only methods on `ed` (e.g. `sue`).
  - You can *not* call any `Employee` methods on `otto` (e.g. `getHours`).

Polymorphism examples:

- You can use the object's extra functionality by casting.

```
Employee ed = new Lawyer();
ed.getVacationDays();                    // ok
otto.getVacationDays();                  // compiler error
((Employee) otto).getVacationDays();     // ok
((Lawyer) otto).sue();                   // runtime error
```

- You can't cast an object into something that it is not.

```
Object otto= new Secretary();
System.out.println(otto.toString());    // ok
otto.getVacationDays();                  // compiler error
((Employee)  otto).getVacationDays();    // ok
((Lawyer) otto).sue();                   // runtime error
```