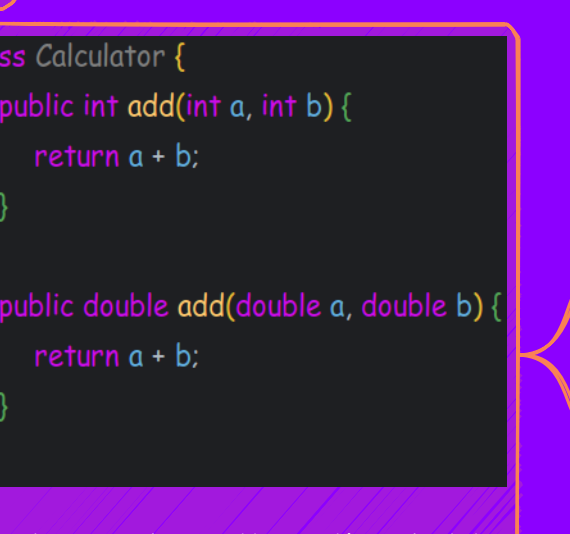
OOP_Second
created by
github.com/farzadafi

Polymorphism

- Polymorphism in Java is a concept by which we can perform a single action in different ways.
- Polymorphism is derived from 2 Greek words
  - poly
    - means many
  - morphs
    - means forms
  - So polymorphism means many forms.
- There are two types of polymorphism in Java
  - compile-time polymorphism
  - runtime polymorphism
- We can perform polymorphism in java by method overloading and method overriding.
- compile-time polymorphism, also known as
  - static polymorphism
  - early binding
  - refers to the ability of the compiler to determine which method to invoke based on the method signature at compile time. It is achieved through method overloading.
  - Method overloading allows you to define multiple methods with the same name in a class, but with different parameter lists.
  - The compiler resolves which method to call by matching the method name and the arguments provided during the method invocation to the appropriate method signature during compilation.
  - Consider the following example:

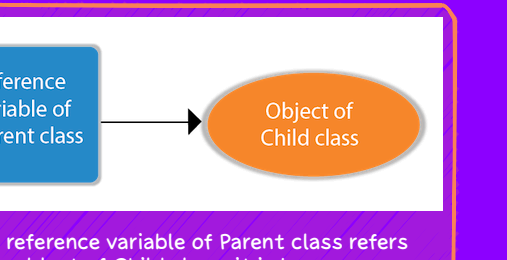


```
class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }
}
```

In the above example, the add method is overloaded with two versions: one that takes two integers and another that takes two doubles. The compiler determines which version of the method to call based on the argument types used in the method invocation.

- In this case, the compiler knows at compile time which version of the add method to invoke based on the argument types (int or double).
- Compile-time polymorphism allows for static binding, meaning the method resolution is determined during compilation rather than at runtime. It offers efficiency as the method invocation is resolved early, eliminating the need for dynamic method lookup during program execution.

- Runtime Polymorphism
  - or Dynamic Method Dispatch
  - is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
  - In this process, an overridden method is called through the reference variable of a superclass.
  - The determination of the method to be called is based on the object being referred to by the reference variable.
  - Upcasting

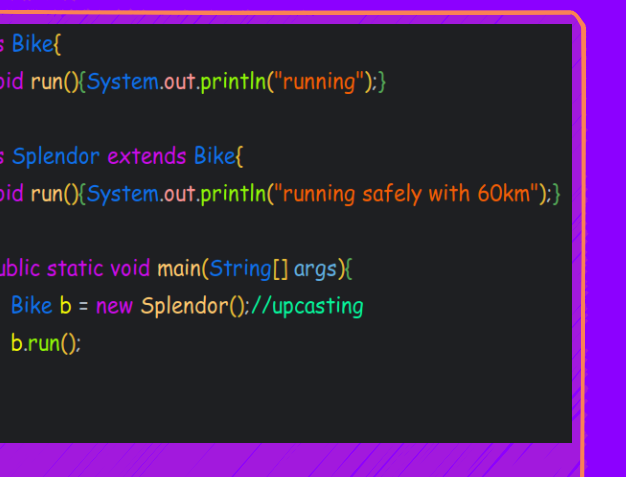


If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.

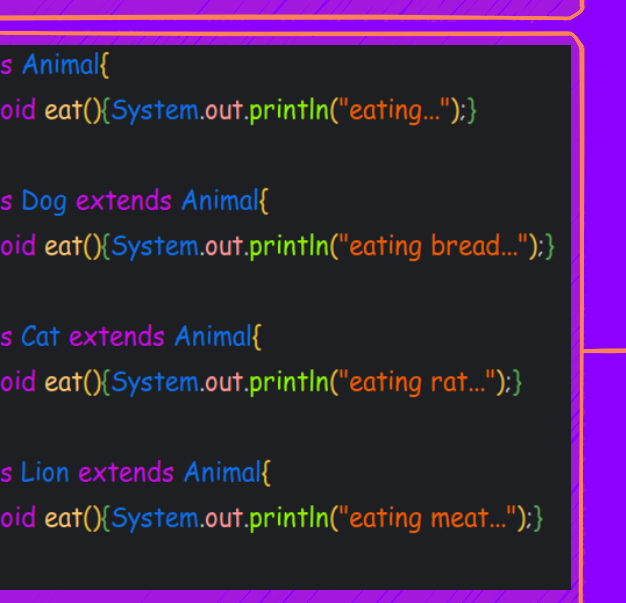
```
class A{}
class B extends A{}

A a=new B();//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type.

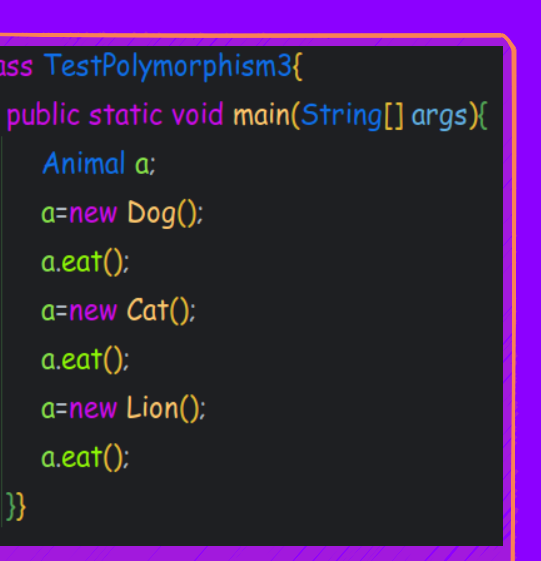
- Example of Java Runtime Polymorphism



```
class Bike{
    void run(){System.out.println("running");}
}
class Splendor extends Bike{
    void run(){System.out.println("running safely with 60km");}

    public static void main(String[] args){
        Bike b = new Splendor();//upcasting
        b.run();
    }
}
```

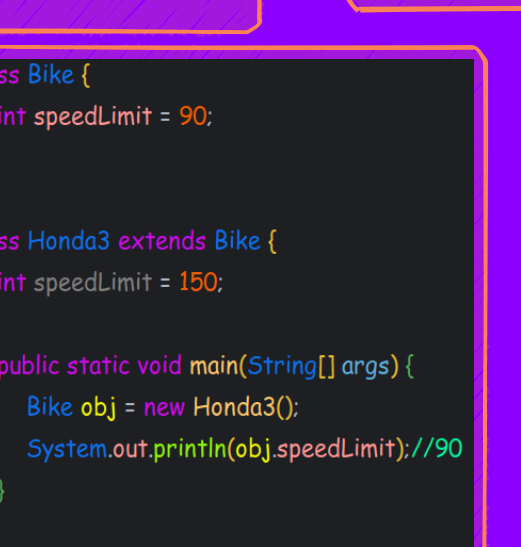


```
class Animal{
    void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
    void eat(){System.out.println("eating bread...");}
}
class Cat extends Animal{
    void eat(){System.out.println("eating rat...");}
}
class Lion extends Animal{
    void eat(){System.out.println("eating meat...");}
}
```



```
class TestPolymorphism3{
    public static void main(String[] args){
        Animal a;
        a=new Dog();
        a.eat();
        a=new Cat();
        a.eat();
        a=new Lion();
        a.eat();
    }}
```

- Data Member
  - A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.

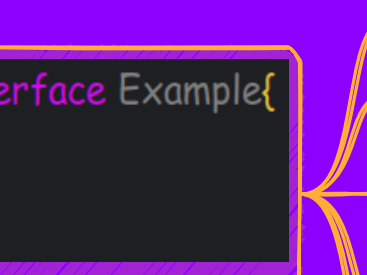


```
class Bike {
    int speedLimit = 90;
}

class Honda3 extends Bike {
    int speedLimit = 150;

    public static void main(String[] args) {
        Bike obj = new Honda3();
        System.out.println(obj.speedLimit);//90
    }
}
```

- dynamic byinding VS static byinding
  - Dynamic Binding (Runtime Binding)
  - Static Binding (Compile-Time Binding)
  - Invocation Resolution
    - Resolved at runtime based on the actual type of the object.
    - Resolved at compile time based on the reference type.
  - Method Selection
    - Selects the appropriate method implementation based on the actual object type.
    - Selects the appropriate method based on the method signature known at compile time.
  - Type Dependency
    - Depends on the actual type of the object.
    - Depends on the reference type of the object.
  - Overriding vs. Overloading
    - Associated with method overriding, where a subclass provides its own implementation of a method defined in its superclass.
    - Associated with method overloading, where multiple methods with the same name but different parameters are defined in a class.
  - Flexibility vs. Efficiency
    - Provides flexibility for runtime method invocation but incurs a small performance overhead due to late binding.
    - Offers efficiency as the method resolution is determined early during compilation, eliminating the need for dynamic method lookup at runtime.

interface

- description
  - An interface in Java is a blueprint of a behaviour.
  - A Java interface contains static constants and abstract methods.
  - The interface in Java is a mechanism to achieve abstraction.
  - There can be only abstract methods in the Java interface, not the method body.
  - It is used to achieve abstraction and multiple inheritance in Java.
  - In other words, you can say that interfaces can have abstract methods and variables.
  - It cannot have a method body
  - Interfaces specify what a class must do and not how. It is the blueprint of the behaviour.
  - Interface do not have constructor.
  - If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- Syntax:



```
interface Example{

}
```

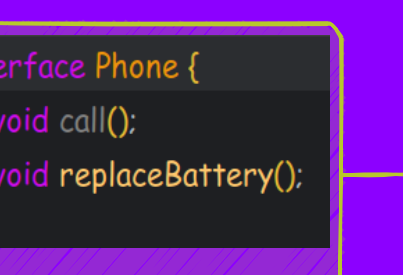
  - To declare an interface, use the interface keyword.
  - It is used to provide total abstraction.
  - That means all the methods in an interface are declared with an empty body and are public and all fields are public, static, and final by default.
  - A class that implements an interface must implement all the methods declared in the interface.
  - To implement interface use implements keyword.
- Why do we use an Interface?
  - It is used to achieve total abstraction.
  - Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
  - Any class can extend only 1 class but can any class implement infinite number of interface.
  - It is also used to achieve loose coupling.
  - Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?
    - The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public and static.
- Difference Between Class and Interface
  - -
    - class
    - interface
  - 1
    - In class, you can instantiate variables and create an object.
    - In an interface, you can't instantiate variables and create an object.
  - 2
    - Class can contain concrete(with implementation) methods
    - The interface cannot contain concrete(with implementation) methods
  - Purpose
    - A class is a blueprint or a template for creating objects
    - whereas an interface is a contract that defines a set of methods that a class must implement.
  - Implementation
    - A class can contain both properties and methods along with their implementation
    - whereas an interface only defines method signatures without any implementation.
  - Accessibility
    - A class can have different levels of accessibility, such as public, private, and protected
    - whereas all members in an interface are implicitly public.
  - Instantiation
    - A class can be instantiated to create an object
    - whereas an interface cannot be instantiated as it only defines a set of methods that a class must implement.
  - Multiple implementation
    - A class can only inherit from one class
    - whereas a class can implement multiple interfaces.
- Real Example
  - Shape
  - store(Tractor :), Motor, Car)
- Extending Interfaces
  - One interface can inherit another by use of keyword extends.
  - When a class implements an interface that inherit another interface, it must provide implementation for all method required by the interface inheritance chain.
- New Features Added in Interfaces in JDK 8
  - implementation
    - Prior to JDK 8, the interface could not define the implementation.
    - We can now add default implementation for interface methods.
    - This default implementation has a special use and does not affect the intention behind interfaces.
    - Suppose we need to add a new function in an existing interface. Obviously, the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.
  - static methods
    - Another feature that was added in JDK 8 is that we can now define static methods in interfaces that can be called independently without an object. Note:
    - Note: these methods are not inherited.
- New Features Added in Interfaces in JDK 9
  - Private methods
    - Java 9 allows interfaces to have private methods, which are methods that can only be used within the interface itself. Private methods can be used to reduce code duplication and improve code readability.
  - Default methods enhancements
    - Java 9 introduced several enhancements to default methods, including allowing private methods to be used within default methods, allowing default methods to be overridden in implementing classes, and allowing diamond inheritance of default methods.
  - Static methods

Abstraction

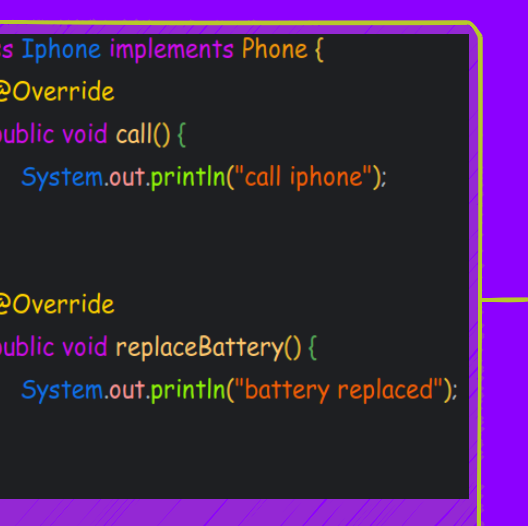
- allows the programmer to represent complex systems in a simplified manner.

ATM
Sitesbay.com
Real Life Example of Abstraction

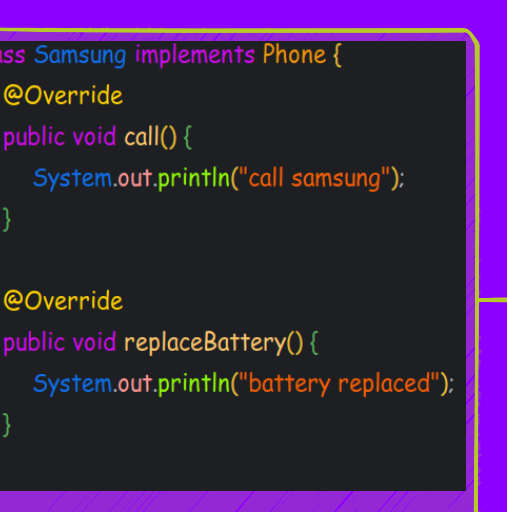
  - dReal-Life Example
    - Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of a car or applying brakes will stop the car, but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.
- In Java, abstraction is implemented through abstract classes and interfaces.
- abstract classes
  - An abstract class is a class that cannot be instantiated directly and is designed to be subclassed by other classes.
  - Java Abstract classes and Java Abstract methods
    - An abstract class is a class that is declared with an abstract keyword.
    - An abstract method is a method that is declared without implementation.
    - An abstract class may or may not have all abstract methods. Some of them can be concrete methods
    - A method-defined abstract must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.
    - Any class that contains one or more abstract methods must also be declared with an abstract keyword.
    - There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator.
    - An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.
- Interface
  - An interface is a collection of abstract methods and constants that can be implemented by any class.
  - We can achieve 100% abstraction using interfaces.
  - Interfaces are similar to abstract classes in that they cannot be instantiated directly, but they differ in that they can be implemented by multiple classes.
  - A class can implement multiple interfaces, but can only inherit from one superclass.
- Abstraction is important because it allows the programmer to focus on the essential features of an object while hiding the implementation details.
- This reduces complexity and makes code easier to maintain and modify.
- Abstraction also promotes code reusability, as classes can be designed to be reused in different contexts without modification.
- Abstraction is primarily for other programmers who work on the project
- real example



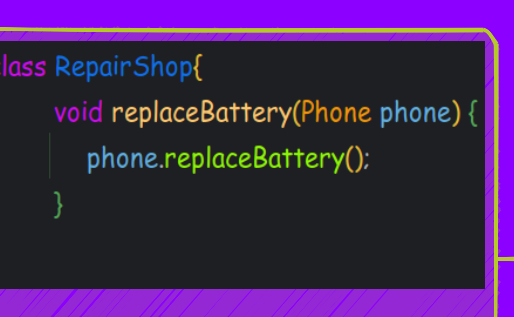
```
interface Phone {
    void call();
    void replaceBattery();
}
```



```
class Iphone implements Phone {
    @Override
    public void call() {
        System.out.println("call iphone");
    }

    @Override
    public void replaceBattery() {
        System.out.println("battery replaced");
    }
}
```

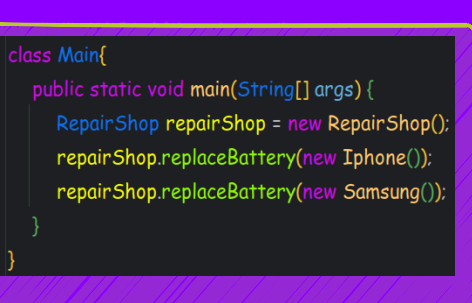


```
class Samsung implements Phone {
    @Override
    public void call() {
        System.out.println("call samsung");
    }

    @Override
    public void replaceBattery() {
        System.out.println("battery replaced");
    }
}
```



```
class RepairShop{
        void replaceBattery(Phone phone) {
            phone.replaceBattery();
        }
}
```

if we dont use from abstraction here, we have to create some of method for each phone, but when we use from Phone interface, each phone class that implement phone, can pass to this method, and replaceBattery method dont know any thing about that phone, this is abstraction :)



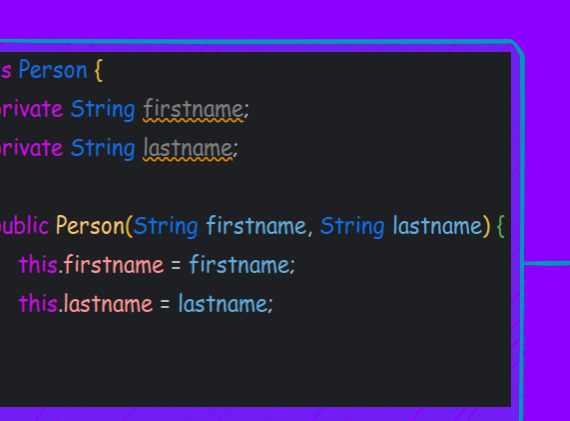
```
class Main{
    public static void main(String[] args) {
        RepairShop repairShop = new RepairShop();
        repairShop.replaceBattery(new Iphone());
        repairShop.replaceBattery(new Samsung());
    }
}
```

The abstraction in this code is useful for other programmers who are working on the project, as it provides a common interface for different types of phones that can be used interchangeably in the RepairShop class.

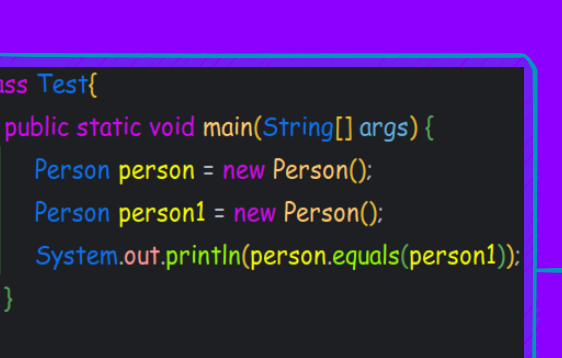
- Here are some reasons why we use abstract in Java:
  - Abstraction
    - Abstract classes are used to define a generic template for other classes to follow. They define a set of rules and guidelines that their subclasses must follow. By providing an abstract class, we can ensure that the classes that extend it have a consistent structure and behavior. This makes the code more organized and easier to maintain.
  - Polymorphism
    - Abstract classes and methods enable polymorphism in Java. Polymorphism is the ability of an object to take on many forms. This means that a variable of an abstract type can hold objects of any concrete subclass of that abstract class. This makes the code more flexible and adaptable to different situations.
- Encapsulation vs Data Abstraction
  - Encapsulation is data hiding(information hiding) while Abstraction is detailed hiding(implementation hiding).
  - Encapsulation is a procedure that takes place at the implementation level, while abstraction is a design-level process.
- Advantages of Abstraction
  - It reduces the complexity of viewing things.
  - Avoids code duplication and increases reusability.
  - Helps to increase the security of an application or program as only essential details are provided to the user.
  - It improves the maintainability of the application.
  - It improves the modularity of the application.
  - Improves code reusability and maintainability.
  - Hides implementation details and exposes only relevant information.
  - Provides a clear and simple interface to the user.
  - Increases security by preventing access to internal class details.
- Disadvantages of Abstraction
  - Abstraction can make it more difficult to understand how the system works.
  - It can lead to increased complexity, especially if not used properly.
  - This may limit the flexibility of the implementation.
  - Abstraction can add unnecessary complexity to code if not used appropriately, leading to increased development time and effort.
  - Abstraction can make it harder to debug and understand code, particularly for those unfamiliar with the abstraction layers and implementation details.

equals() and hashCode()

- equals()
  - The Object class defines both the equals() and hashCode() methods
  - which means that these two methods are implicitly defined in every Java class,

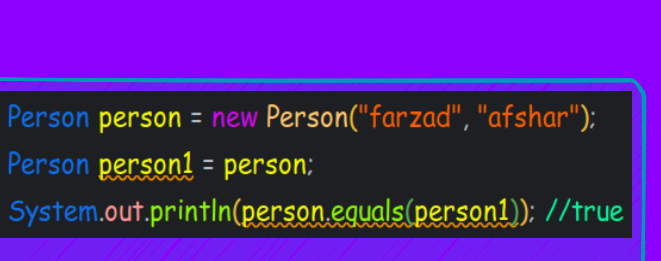


```
class Person {
    private String firstname;
    private String lastname;

    public Person(String firstname, String lastname) {
        this.firstname = firstname;
        this.lastname = lastname;
    }
}
```



```
class Test{
    public static void main(String[] args) {
        Person person = new Person();
        Person person1 = new Person();
        System.out.println(person.equals(person1));
    }
}
```

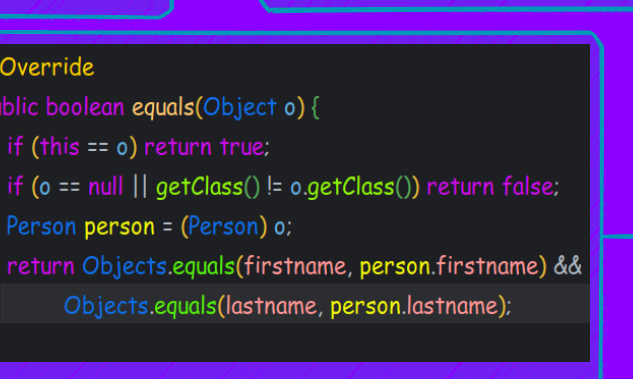
We would expect person.equals(person1) to return true, but with the Person class in its current form, it won't.



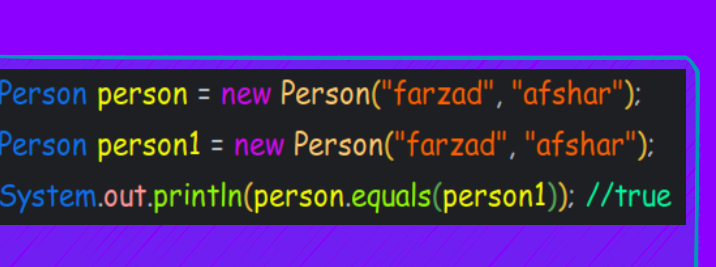
```
Person person = new Person("farzad", "afshar");
Person person1 = person;
System.out.println(person.equals(person1)); //true
```

The default implementation of equals() in the Object class says that equality is the same as object identity, and income and expenses are two distinct instances.

  - Overriding equals()
    - Let's override the equals() method so that it doesn't consider only object identity, but also the value of the two relevant properties:



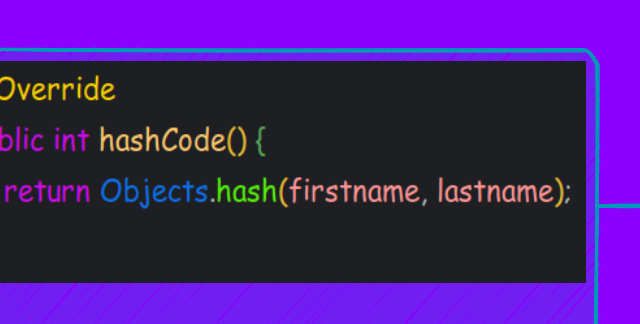
```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Person person = (Person) o;
    return Objects.equals(firstname, person.firstname) &&
            Objects.equals(lastname, person.lastname);
}
```



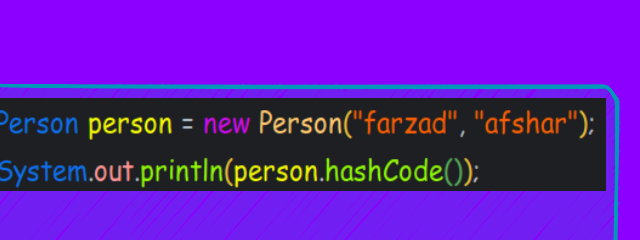
```
Person person = new Person("farzad", "afshar");
Person person1 = new Person("farzad", "afshar");
System.out.println(person.equals(person1)); //true
```

because Override equals mthod :)

  - equals() Contract
    - Java SE defines the contract that our implementation of the equals() method must fulfill. Most of the criteria are common sense. The equals() method must be:
      - reflexive
        - an object must equal itself
      - symmetric
        - x.equals(y) must return the same result as y.equals(x)
      - transitive
        - if x.equals(y) and y.equals(z), then also x.equals(z)
      - consistent
        - consistent: the value of equals() should change only if a property that is contained in equals() changes (no randomness allowed)
- hashCode()
  - hashCode() returns an integer representing the current instance of the class.
  - We should calculate this value consistent with the definition of equality for the class.
  - Thus, if we override the equals() method, we also have to override hashCode().



```
@Override
public int hashCode() {
    return Objects.hash(firstname, lastname);
}
```



```
Person person = new Person("farzad", "afshar");
System.out.println(person.hashCode());
```

  - hashCode() Contract
    - internal consistency
      - the value of hashCode() may only change if a property that is in equals() changes
    - equals consistency
      - objects that are equal to each other must return the same hashCode
    - collisions
      - unequal objects may have the same hashCode

Presented with xmind