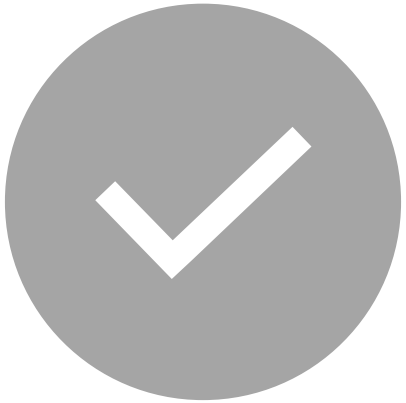
Factory Pattern Method

CSCI 2020

Edmond Lee

Overview

VERY COMMONLY USED DESIGN PATTERN IN JAVA



OFFERS AN EFFICIENT WAY TO CREATE OBJECTS

CAN CREATE MULTIPLE OBJECTS USING THE SAME INTERFACE/ABSTRACT CLASS

Why Use it

Simple to implement

Helps to avoid any duplications when creating objects and classes

Program is more modular as objects/sub-classes not dependent on specific implementations of its interfaces

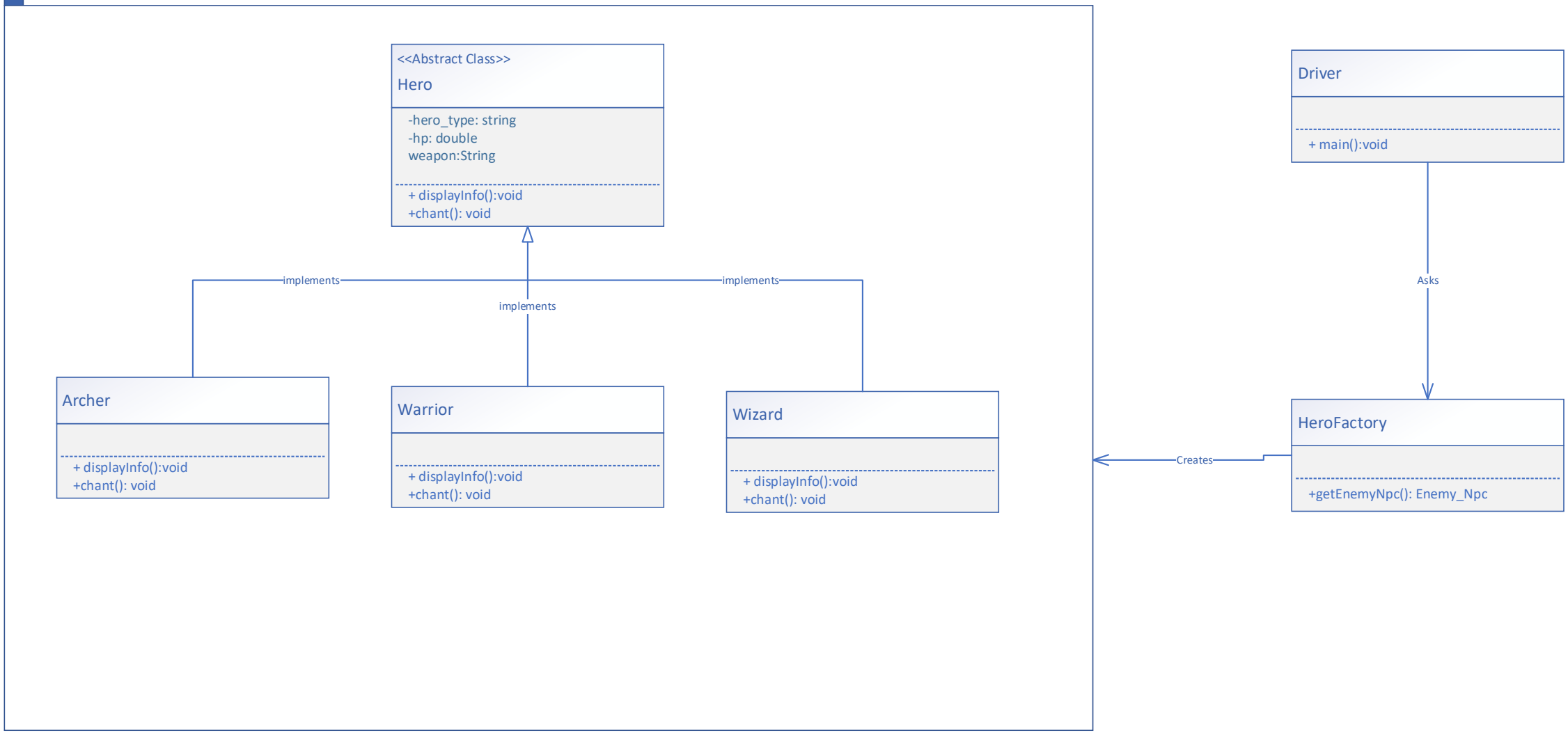
When To Use

- When a Class does not know what sub-class/objects will be created beforehand
- When a Class needs its subclasses to specify the object/subclass to be created

How it Works

- An interface/ abstract class is created and used as a template
- Specific Classes are created that implement the parent abstract class/interface
- A “Factory Class” is used generate objects from the Specific Classes

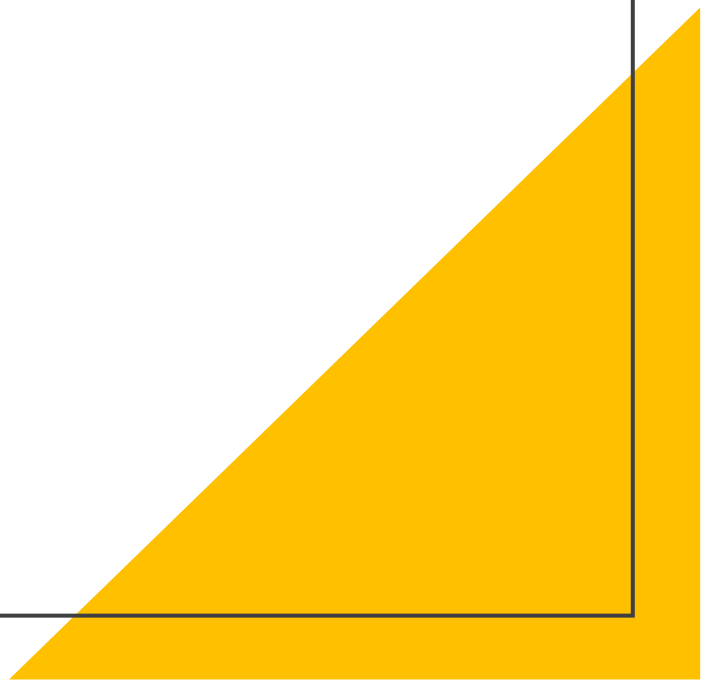
Sample Structure



Still Confused?

Example

- We are creating the next best RPG game The Final Final Fantasy
- The user gets to choose what type of Hero to create
- How does the program know which Hero type to create?



Lets Find Out

<<Abstract Class>>

Hero

-hero_type: string
-hp: double
weapon:String

+ displayInfo():void
+chant(): void

```java
public abstract class Hero
{
    /***
     *  GENERIC attrtibues for each enemey NPC
     */

    private String hero_type;
    private String weapon;
    double hp;

    public abstract void chant();


    /*** ...*/
    public void displayInfo()
    {...}

    /***
     * GETTERS
     */
    public String getHeroType() { return hero_type; }
    public String getWeapon() { return weapon; }
    public double getHp() { return hp; }

    /***
     * Setters
     */
    public void setHeroType(String enemy_type) { this.hero_type = enemy_type; }
    public void setWeapon(String weapon) { this.weapon = weapon; }
    public void setHp(double hp) { this.hp = hp; }
}
```

Template

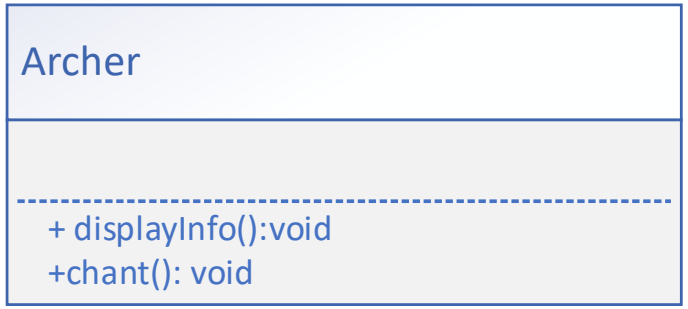
First Create Abstract Class/Interface

In our case it’s Generic Hero

Specific Classes

Create Specific Classes that implement/extend the template

In our case we will create 3 Specific Hero Classes: Archer, Warrior, Wizzard

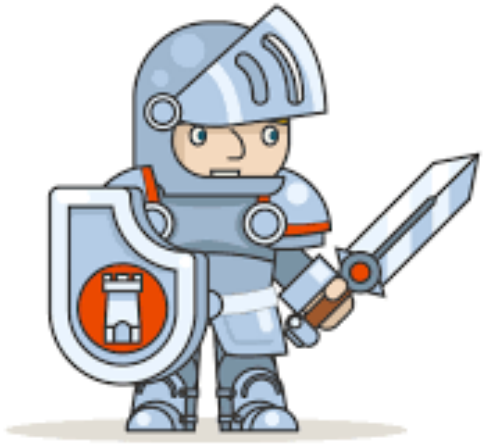



```java
public class Archer extends Hero
{
    /***
     * Default constructor for an Archer EnemyNPC type
     */
    public  Archer()
    {
        setHeroType("Archer");
        setWeapon("Bow and Arrow");
        setHp(50);
    }

    /***
     * Chant method for Archer
     */
    @Override
    public void chant()
    {
        System.out.println("Archer Chant: I AM ARCHER I LIKE ARROWS!!");

    }
```

Specific Classes: Archer



Warrior
+ displayInfo():void +chant(): void

```java
public class Warrior extends Hero
{
    /***
     * Default constructor for an Wizzard EnemyNPC type
     */
    public  Warrior()
    {
        setHeroType("Warrior");
        setWeapon("Sword and Shield");
        setHp(80);
    }

    /***
     * Displays Warrior Chat
     */

    @Override
    public void chant()
    {
        System.out.println("Warrior Chant: I AM WARRIOR I Like Sword!!");

    }
```

Specific Classes: Warrior

Wizard
+ displayInfo():void +chant(): void

```java
public class Wizzard extends Hero
{
    /**
     * Default constructor for an Wizzard EnemyNPC type
     */
    public  Wizzard()
    {
        setHeroType("Wizzard");
        setWeapon("Wand and Spell Book");
        setHp(40);
    }

    /*** ...*/
    @Override
    public void chant()
    {
        System.out.println("Wizzard Chant: I AM WIZZARD I Like Magic!!");

    }
}
```

Specific Classes: Warrior

HeroFactory

+getEnemyNpc(): Enemy_Npc

```
public class HeroFactory
{
    // Factory class
    public Hero HeroFactory(String hero)
    {
        if (hero==null)
        {
            return null;
        }

        // Used to create an Archer NPC
        if(hero.equalsIgnoreCase( anotherString: "Archer"))
        {
            return new Archer();
        }
        // Used to create  WIZZARD NPC
        else if (hero.equalsIgnoreCase( anotherString: "wizzard"))
        {
            return new Wizzard();
        }
        // Used to create  warrior NPC
        else if (hero.equalsIgnoreCase( anotherString: "warrior"))
        {
            return new Warrior();
        }
```

Factory Class

A Factory Class is created and used to determine which class to use when creating the object

```
public static void main(String[] args)
{
    int user_hero=0;
    // Creating the factory
    HeroFactory npcFactory = new HeroFactory();

    System.out.println("FACTORY DEMO");

    System.out.println("Select Hero Type to create");
    System.out.println("1. Archer");
    System.out.println("2. Wizzard");
    System.out.println("3. Warrior");
    System.out.print("Choose from Options (1-3): ");
    Scanner input = new Scanner(System.in);
    user_hero= input.nextInt();

    System.out.println();


    if(user_hero==1)
    {
        // using factory to create Archer Hero type
        Hero hero1= npcFactory.HeroFactory("archer");
        hero1.displayInfo();
        hero1.chant();

    }
    else if (user_hero==2)
    {...}
    else if (user_hero==3)
    {...}
```

Final Product

- User can decide which Hero Type to build
- The factory will choose the appropriate class to call when creating the object based on users input

```
FACTORY DEMO
Select Hero Type to create
1. Archer
2. Wizzard
3. Warrior
Choose from Options (1-3): 1

Hero Info:
Hero: Archer
Hero Weapon: Bow and Arrow
Hero HP: 50.0
Archer Chant: I AM ARCHER I LIKE ARROWS!!
```

Final Product

Resources Cited

- Java Point. (n.d.). *Factory Method Pattern*. Retrieved March 1, 2022, from https://www.javatpoint.com/factory-method-design-pattern
- Refactoring Guru. (n.d.). *Factory Method*. Retrieved March 1, 2022, from https://refactoring.guru/design-patterns/factory-method
- Teo, I. (2011, December 16). *Understanding the Factory Method Design Pattern*. Retrieved March 1, 2022, from Sitepoint : https://www.sitepoint.com/understanding-the-factory-method-design-pattern/
- Tutorials point. (n.d.). *Design Pattern - Factory Pattern*. Retrieved March 1, 2022, from https://www.tutorialspoint.com/design_pattern/factory_pattern.htm