FILKOM

Polymorphism

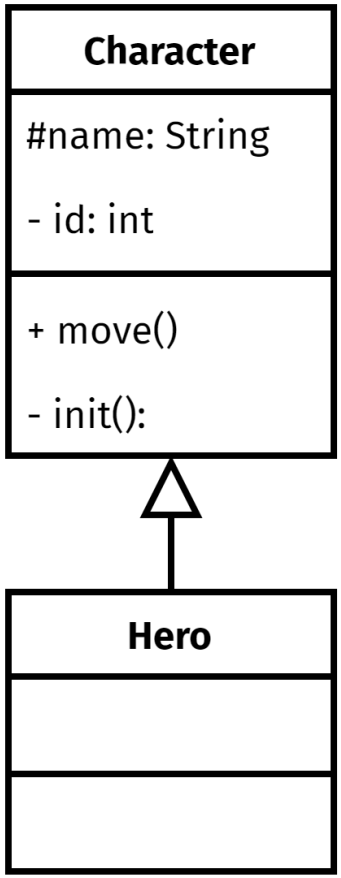
Object-oriented Programming

Aryo Pinandito, S.T., M.MT, Ph.D.

Review: Inheritance

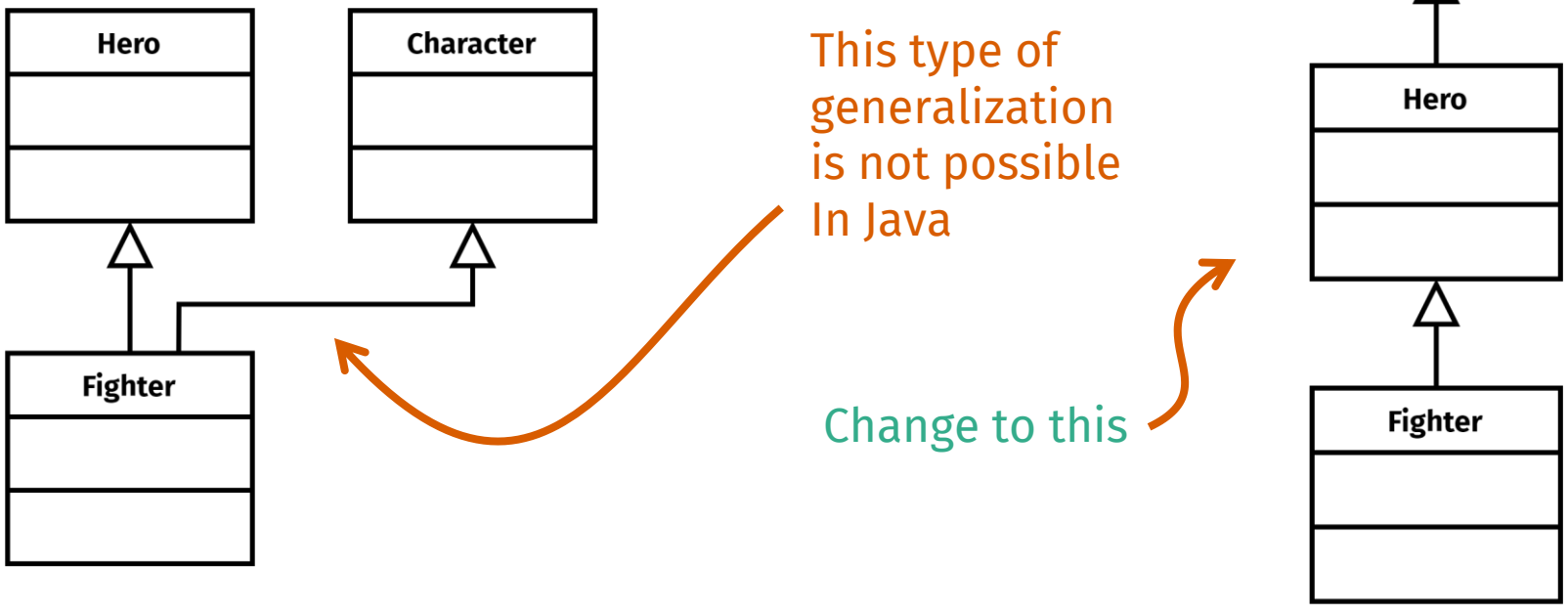
- Generalization relationship is also known as "inheritance" relationship
  - non-private members (attributes and methods) of superclass were inherited to the subclass.
- In generalization, a subclass inherits all the members of its superclass, except for those with private modifier.

```
public class Character {
  protected String name;
  private int id;
  public void move() { ... }
  private void init() { ... }
}

public class Hero extends Character {}

Hero hero = new Hero();
hero.move(); // inherited

hero.init();    // Error, not inherited
print(hero.id); // Error, not inherited
```



Review: Multilevel Inheritance

- Java supports only single-parent relationship in generalization



The code below is OK

```
public class Character {}
public class Hero extends Character {}
public class Fighter extends Hero {}
// Fighter inherits Hero and Character
```

But this one is not OK

```
public class Character {}
public class Hero {}
public class Fighter extends Character, Hero {}
// Not allowed, a class cannot extends
// two or more classes at a time
```

Review: Overload vs Override

Overload

- Two or more method have the same name, but different parameter signature
  - different type, number, and/or order
- Can be implemented in a single class
- Can be implemented in two or more different class having generalization relationship

Override

- Two or more method have the same name and have the same parameter signature
  - same type, number, and order
- Can only be implemented in generalization relationship, abstract class, or implementation of interface

FILKOM

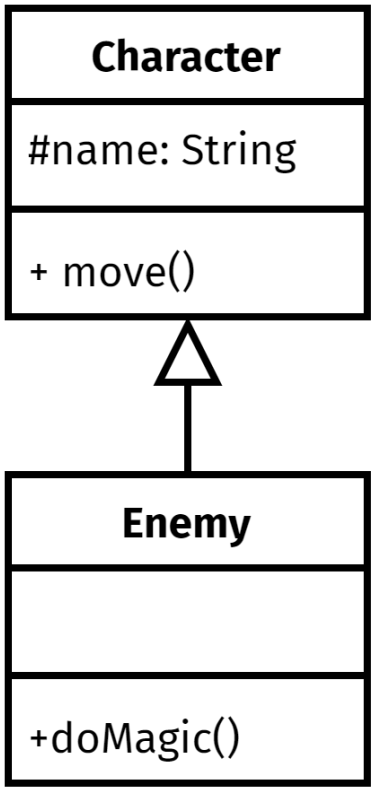
Polymorphism

Polymorphism

- Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by generalization.
- Each class in generalization may have methods of the same name and signature, but have different implementation body
- Polymorphism occur when variable of supertype refers to object of subtype

Subtype and Supertype

- A class is defining a (data) type
- A subtype is a (data) type defined by a subclass
- A supertype is a (data) type defined by a superclass



Character is a supertype of Enemy
Enemy is a subtype of Character

Type of Polymorphism

- Static Polymorphism
  - Uses overloading technique
  - Compile-time polymorphism, compiler decides which method to execute by the signature of the method call
- Dynamic Polymorphism
  - Uses overriding technique
  - Run-time polymorphism, JVM decides which method to execute by resolving the overridden method during run-time

Type of Polymorphism

- Static Polymorphism
  - Uses overloading technique
  - Compile-time polymorphism, the compiler decides which method to execute by the signature of the method call
- Dynamic Polymorphism
  - Uses overriding technique
  - Run-time polymorphism

```
public class Hero {
  public void say() {
    print("Hello!");
  }

  // same name but different parameter signature!
  public void say(String words) {
    print("Yo! " + words);
  }
}

Hero h =  new Hero();
// Compiler decides which method to call
h.say(); // Hello!
h.say("Bruh..."); // Yo! Bruh...
```

Dynamic Polyorphism: Upcasting in Generalization

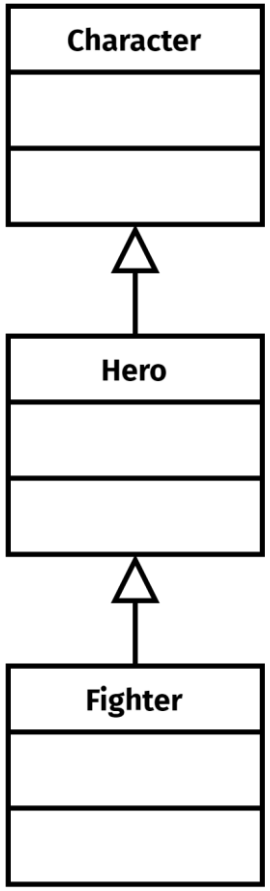
- Upcasting refers to the process where an object of the subclass is referred to by a reference variable of the superclass.
- The approach to this type of polymorphism is called dynamic binding

```java
public class Character {
  public void say(String words) {
    print("Hello!");
  }
}

public class Hero extends Character {
  @Override // Same name and signature
  public void say(String words) {
    print("Yo! " + words);
  }
}
            // A character is morphing to Hero
            // Hero upcasted to Character
            Character ch = (Character) new Hero();
            ch.say("Bruh..."); // Yo! Bruh...
```

Dynamic Binding

- Dynamic binding occurs when the declaration of a supertype variable is instantiated by a subtype (upcasting).

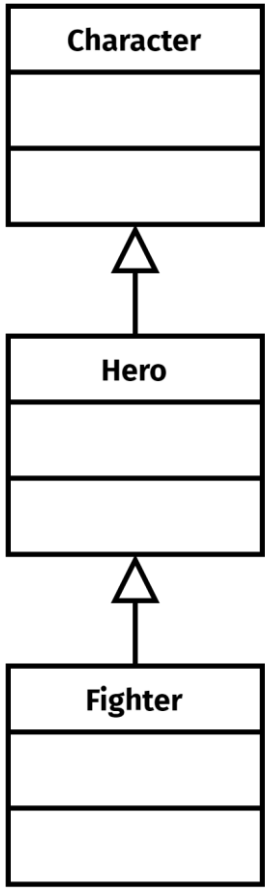
```
// Polymorphism of a Character
// Hero is upcasted to Character
// ch is now a Hero
Character ch = (Character) new Hero();
// Fighter is upcasted to Character
// ch is now morphing to a Fighter
ch = (Character) new Fighter();
```



Dynamic Binding

- Casting in upcasting may be written implicitly, automatically determined by the declaration of the supertype variable.

```
Character ch;
ch = (Character) new Hero();
// is the same with
ch = new Hero();

ch = (Character) new Fighter();
// is also the same with
ch = new Fighter();
```

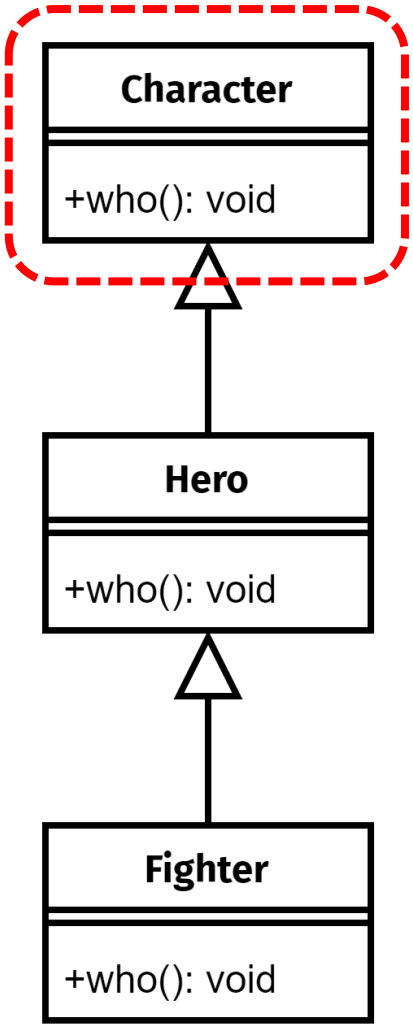


```
public class Character {
  public void who() { print("I am a Character"); }
}

public class Hero extends Character {
  @Override
  public void who() { print("I am a Hero); }
}

public class Fighter extends Hero {
  @Override
  public void who() { print("I am a Fighter"); }
}
```

Character ch

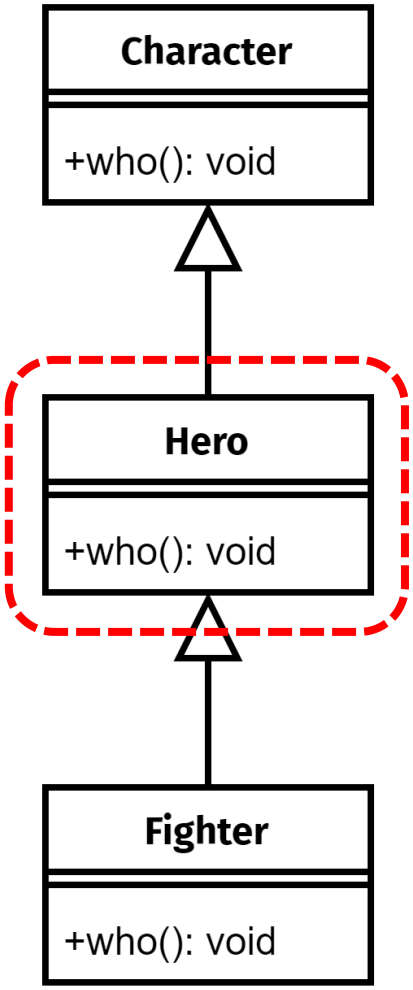
```
Character ch = new Character();
ch.who(); // I am a Character
```

```
// Morphing to Hero
ch = new Hero();
ch.who(); // I am a Hero

// Morphing to Fighter
ch = new Fighter();
ch.who(); // I am a Fighter
```



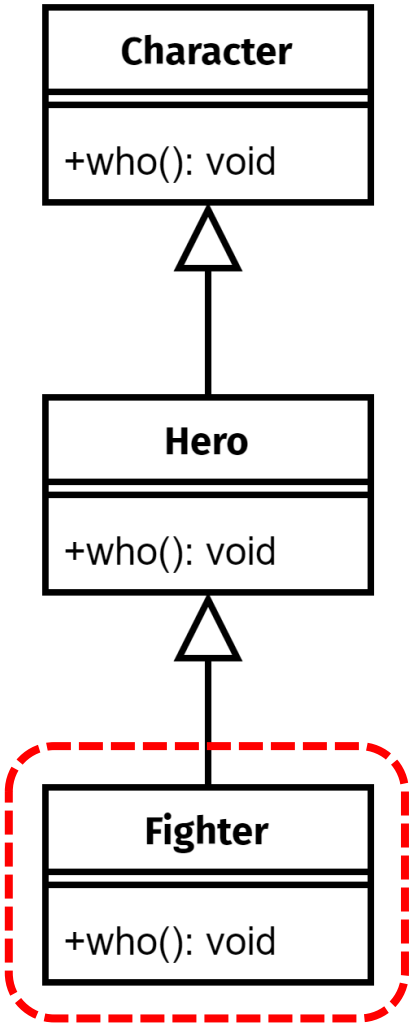
```
Character ch = new Character();
ch.who(); // I am a Character

// Morphing to Hero
ch = new Hero();
ch.who(); // I am a Hero

// Morphing to Fighter
ch = new Fighter();
ch.who(); // I am a Fighter
```

Character ch



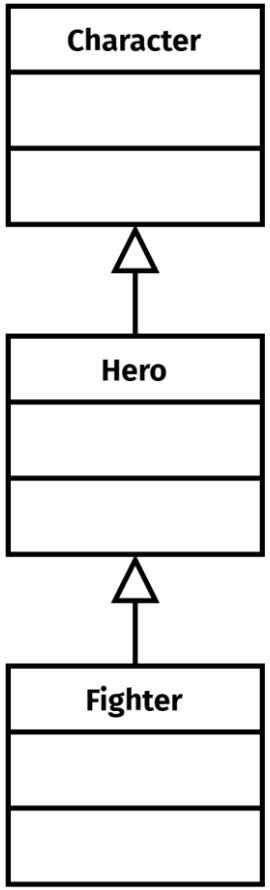
```
Character ch = new Character();
ch.who(); // I am a Character

// Morphing to Hero
ch = new Hero();
ch.who(); // I am a Hero

// Morphing to Fighter
ch = new Fighter();
ch.who(); // I am a Fighter
```

Character ch



Generic Method Parameter

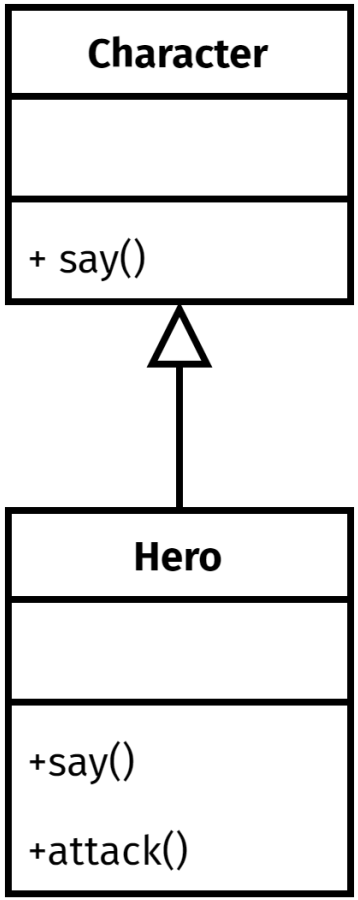
- Polymorphism allows passing an object of a subtype to a method with a supertype parameter



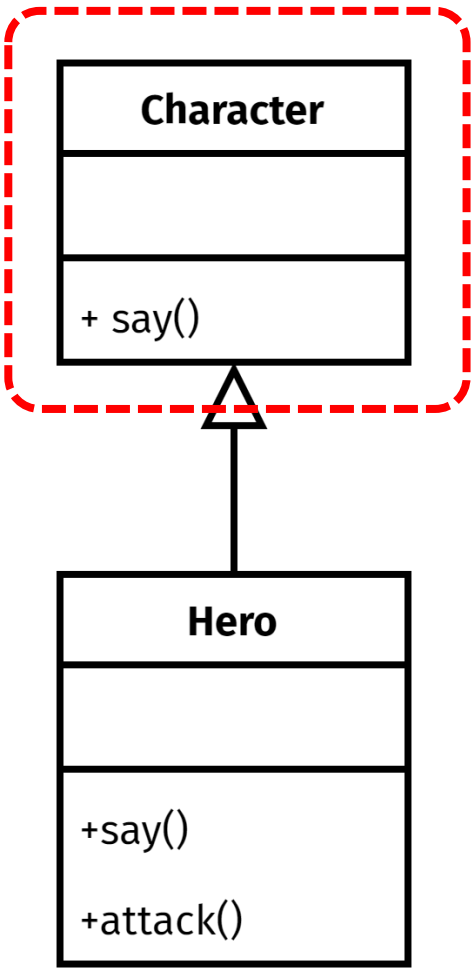
```
public class Game {
  public static void introduce(Character c) {
    c.who();
  }
}

Character character = new Character();
Hero hero           = new Hero();
Fighter fighter     = new Fighter();

Game.introduce(character); // I am a Character
Game.introduce(hero);      // I am a Hero
Game.introduce(fighter);   // I am a Fighter
```

Downcasting

- When a supertype variable refer to a subtype object, the variable cannot "see" methods of subtype that are not overridden from the current supertype.
- To call non-override methods of subtype, the supertype variable must be downcasted to a target subtype
- In Java, downcasting a variable of supertype must be written explicitly

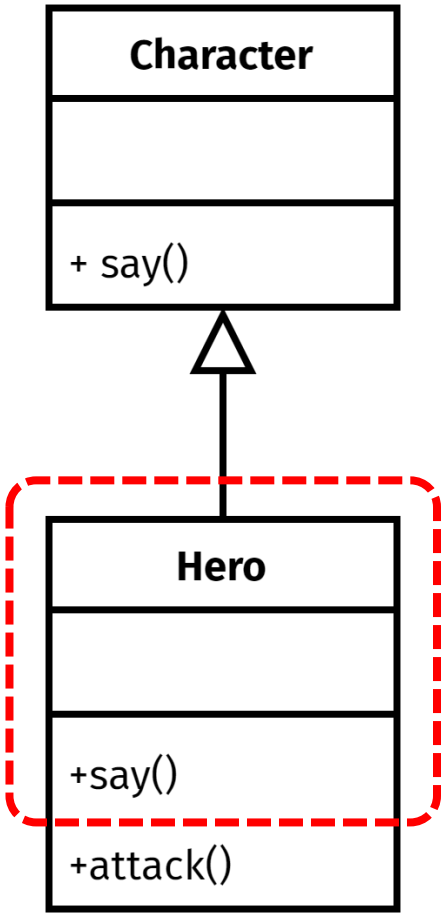


```
Character ch = new Character();
ch.say(); // Calling say() of Character

ch = new Hero();
ch.say(); // Calling say() of Hero

// ch.attack(); // Error
// Character cannot "see" attack() method

Hero h = (Hero) ch; // Downcasting to Hero

// Character: ch can now see Hero's method
// through new reference variable: h
h.say();    // Calling say() of Hero
h.attack(); // Calling attack() of Hero
```

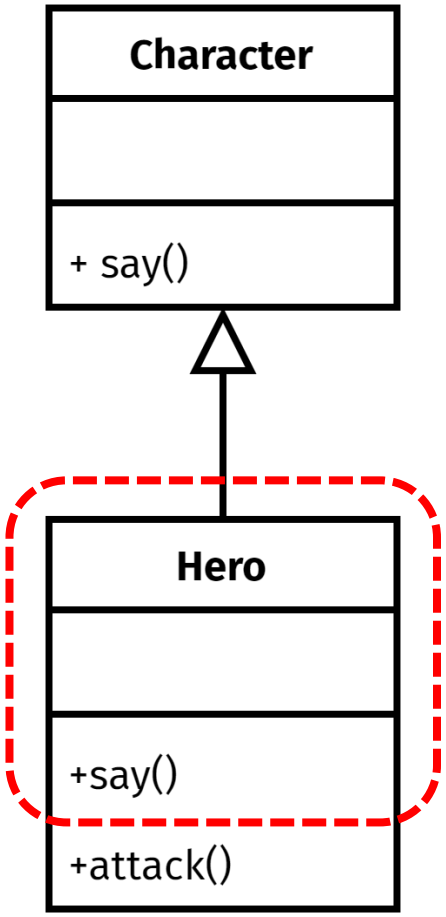


```
Character ch = new Character()
ch.say(); // Calling say() of Character

ch = new Hero();
ch.say(); // Calling say() of Hero

// ch.attack(); // Error
// Character cannot "see" attack() method

Hero h = (Hero) ch; // Downcasting to Hero

// Character: ch can now see Hero's method
// through new reference variable: h
h.say();    // Calling say() of Hero
h.attack(); // Calling attack() of Hero
```

Character ch



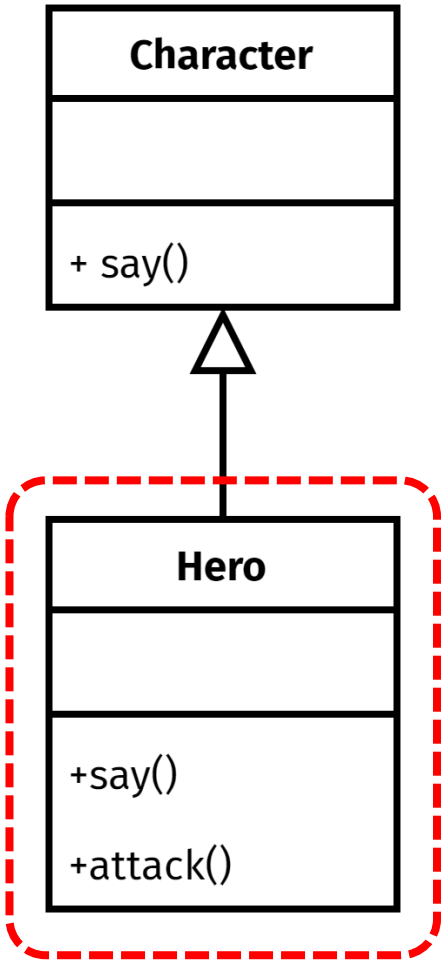
```
Character ch = new Character()
ch.say(); // Calling say() of Character

ch = new Hero();
ch.say(); // Calling say() of Hero

// ch.attack(); // Error
// Character cannot "see" attack() method

Hero h = (Hero) ch; // Downcasting to Hero

// Character: ch can now see Hero's method
// through new reference variable: h
h.say();    // Calling say() of Hero
h.attack(); // Calling attack() of Hero
```



```
Character ch = new Character()
ch.say(); // Calling say() of Character

ch = new Hero();
ch.say(); // Calling say() of Hero

// ch.attack(); // Error
// Character cannot "see" attack() method
```

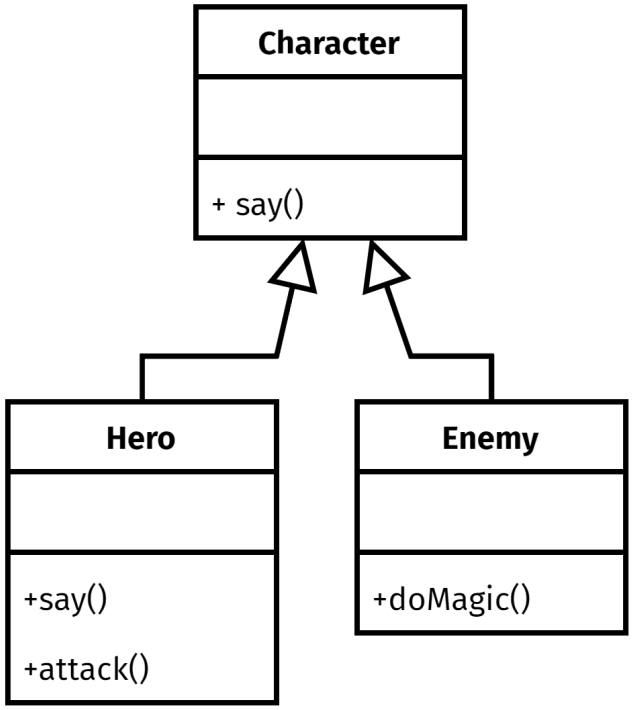
Character -> Hero

```
Hero h = (Hero) ch; // Downcasting to Hero

// Character: ch can now see Hero's method
// through new reference variable: h (Hero)
h.say();    // Calling say() of Hero
h.attack(); // Calling attack() of Hero
```

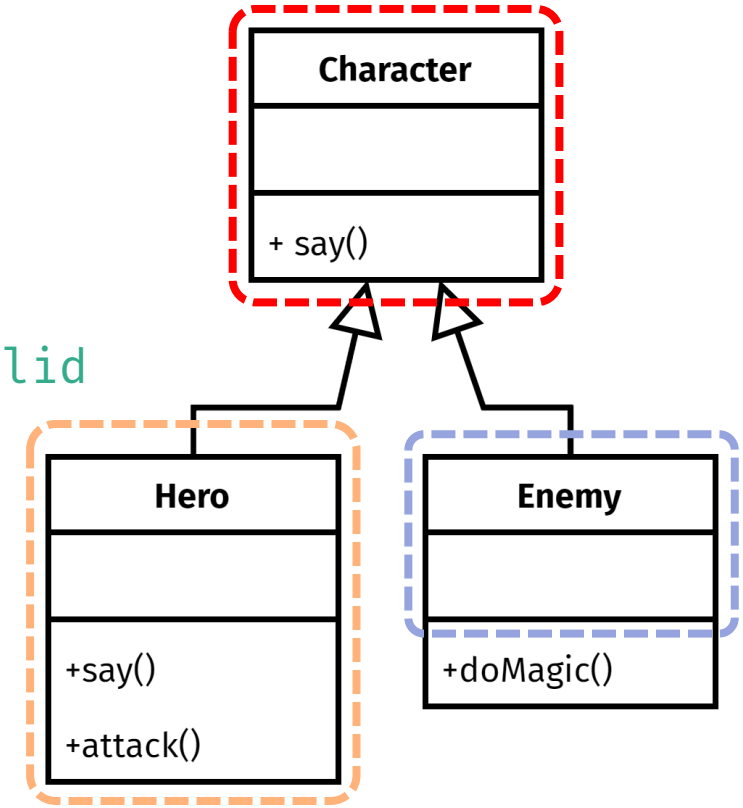


Checking Object of a Type

- Java provides an `instanceof` operator to check whether an object is an instance of a type/class
- This operator is useful to check whether an object is the correct type before downcasting a supertype to subtype.



```
Character ch = new Hero();
Enemy e = (Enemy) ch; // valid

// e.doMagic(); // runtime error
// the actual object
// is not of Enemy type

// perform a check
if (ch instanceof Hero) {
  Hero h = (Hero) ch; // also valid
  h.attack(); // ok
}
```



Questions?