CS102
Programming II

CLASS (PART - I)

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

جامعة الأمير سلطان
PRINCE SULTAN UNIVERSITY

TOPICS

- Inheritance
  - **`extends`**
  - **`super`**
- Polymorphism

CS102 Programming II

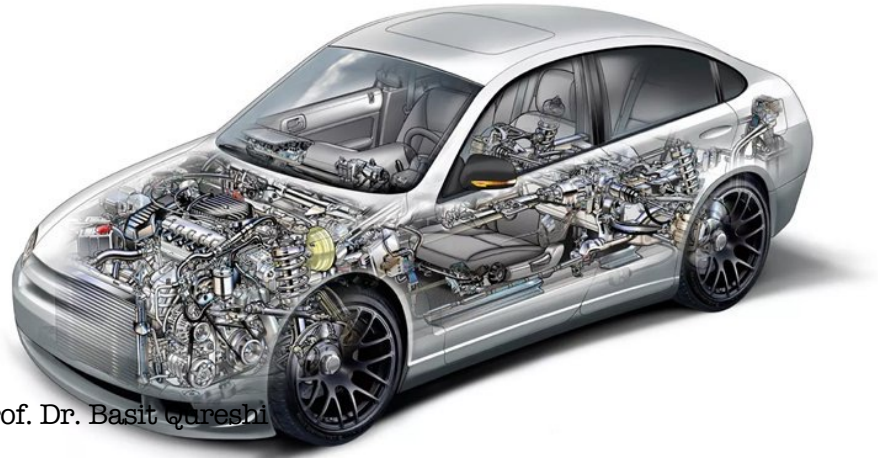
INHERITANCE

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

جامعة الأمير سلطان
PRINCE SULTAN UNIVERSITY

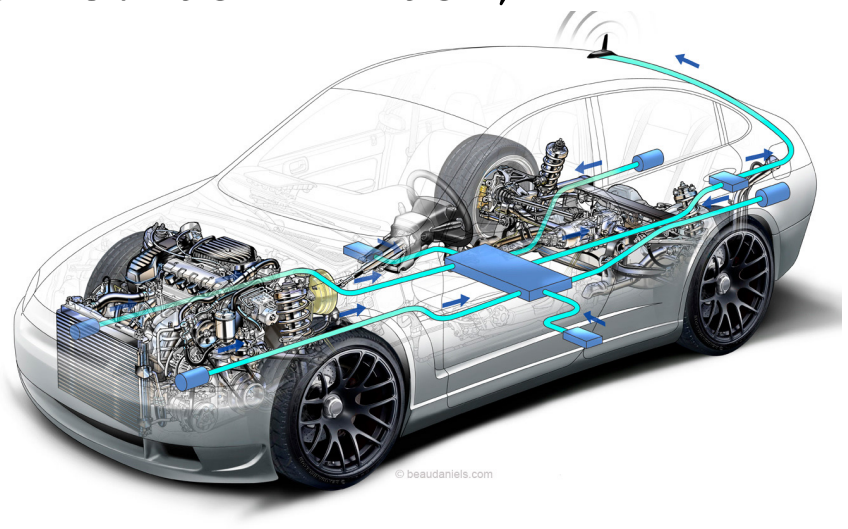
RECALL COMPOSITION

```
public class Car {
   int year;
   String model;
   Engine e;
   Car(){ //initialize attributes
   }
   Car(int year, String model){
      this.year = year;
      this.model = model;
   }
}
```



```
public class Engine {
   int hp;
   String fuel;
   Engine() {//initialize attributes


   }
   Engine(int hp, String fuel){
      this.hp = hp;
      this.fuel = fuel;
   }
}
```



RECALL COMPOSITION

```
public class Car {
   int year;
   String model;
   Engine e;
   Car(int year, String model){
      this.year = year;
      this.model = model;
   }
}
```

```
public class Engine {
   int hp;
   String fuel;
   Engine(int hp, String fuel){
      this.hp = hp;
      this.fuel = fuel;
   }
}
```

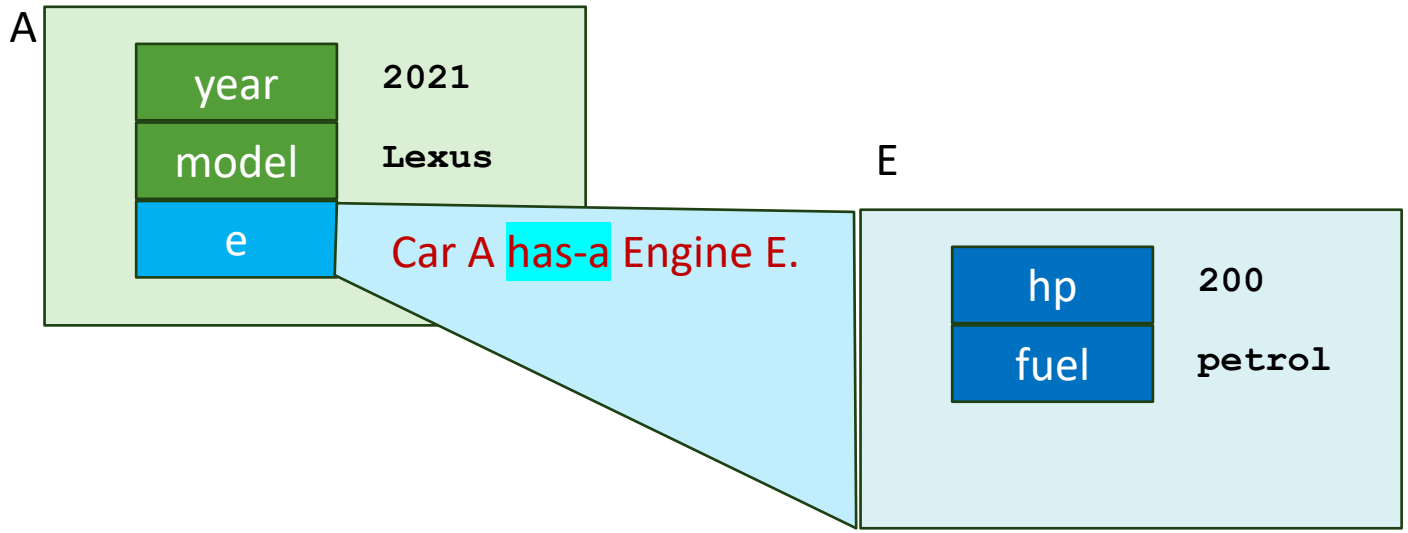
Every **Car** has-a **Engine**.

Engine is a class. **e** is object of class **Engine**, which is a attribute of class **Car**.

RECALL COMPOSITION

- Make a Car object and initialize the composed object of Engine class

```
public static void main(String [] args){
   Car A = new Car(2021, "Lexus");
   Engine E = new Engine(200, "petrol");
   A.e = E;
  }
```



Methods of class Engine can be available to object A.e

INHERITANCE

- Recall Composition:

Every **`Car`** has-a **`Engine`**

- Inheritance defines the is-a relationship between two classes

Examples:

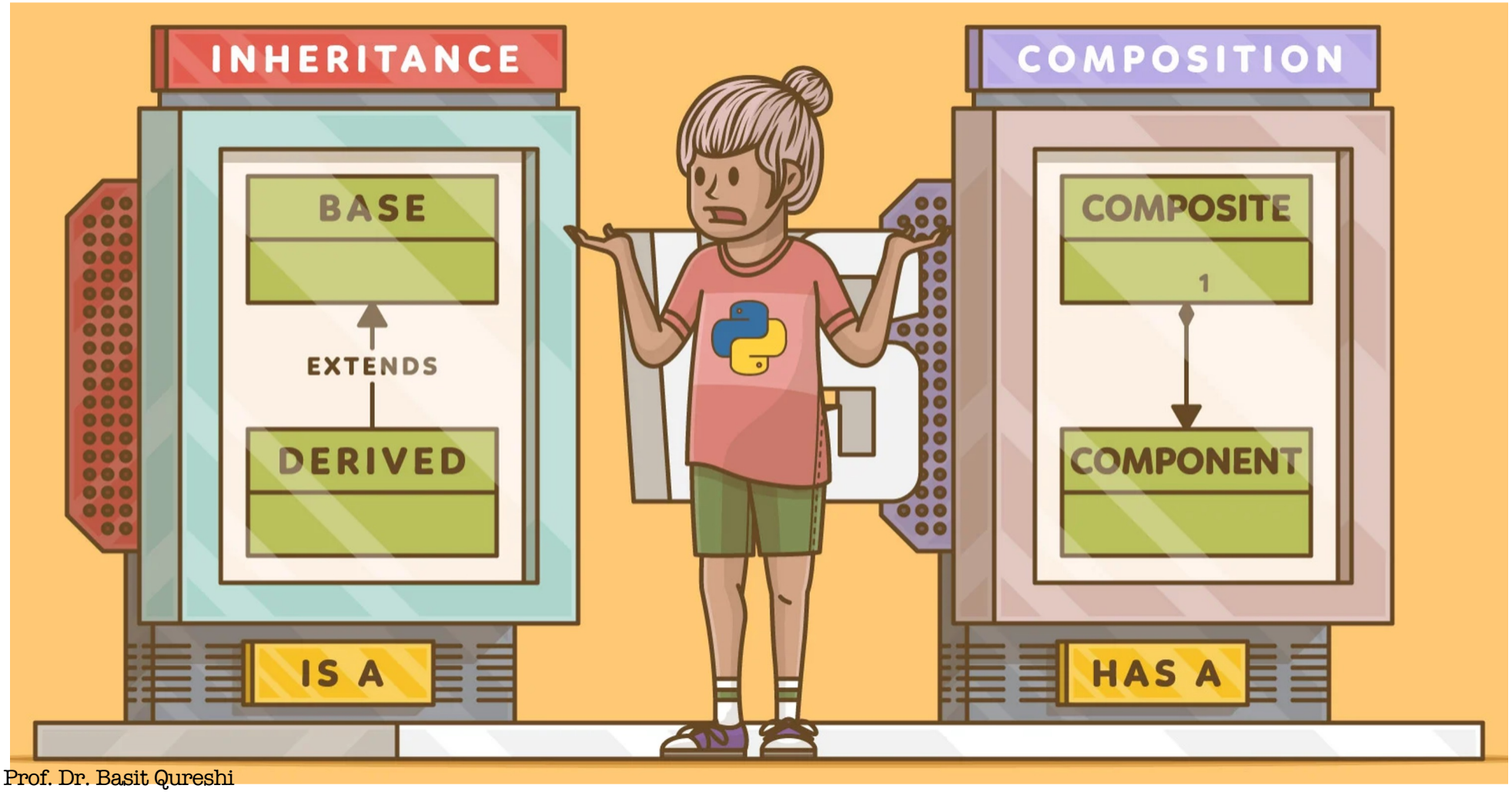
A **`Student`** is-a **`Person`**. A **`Professor`** is-a **`Person`**

A **`Car`** is-a **`Vehicle`**. A **`Bus`** is-a **`Vehicle`**

A **`Laptop`** is-a **`Device`**. A **`Tablet`** is-a **`Device`**

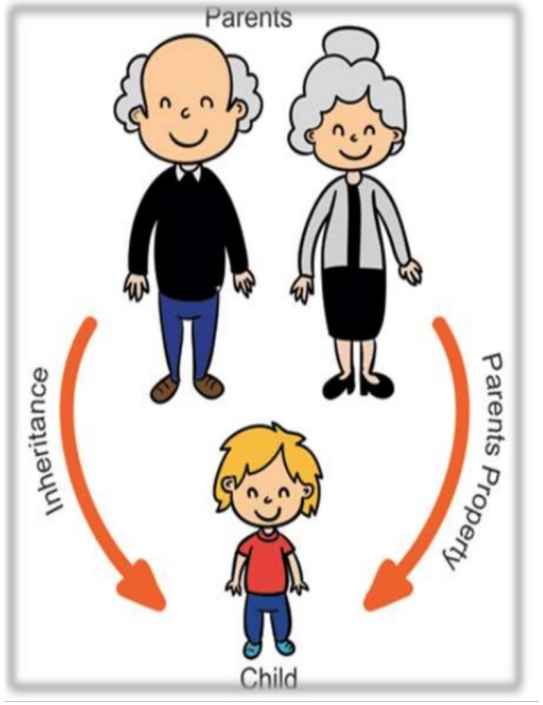
INHERITANCE

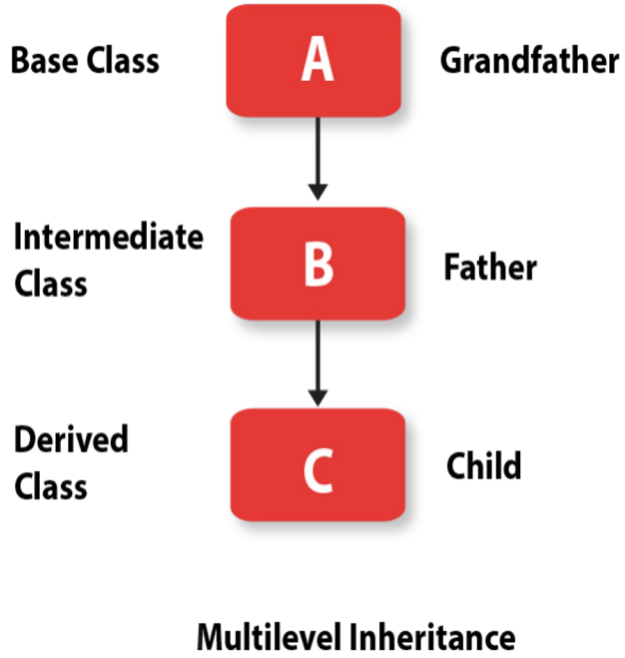
- We use **`extends`** to define the **is-a** relationship between classes.



INHERITANCE

- *Inheritance* allows a software developer to *derive* a new class from an existing one
- The existing class is called the ***parent*** *class,* or ***super****class*, or ***base****class*
- The derived class is called the ***child*** *class* or ***sub****class*
- The ***child*** inherits characteristics of the ***parent*** including methods and variables/attributes defined by the parent class





WHY INHERITANCE? REUSABILITY

- Save time: No need to implement the common methods/behaviors again.

  The child class can derived/reuse the parent methods/behaviors.

  No need to re-implement from the scratch.

- Save space: No need to declare the common attributes again.

  - The child class can derived the common attribute from the parent class.

CS102 Programming II

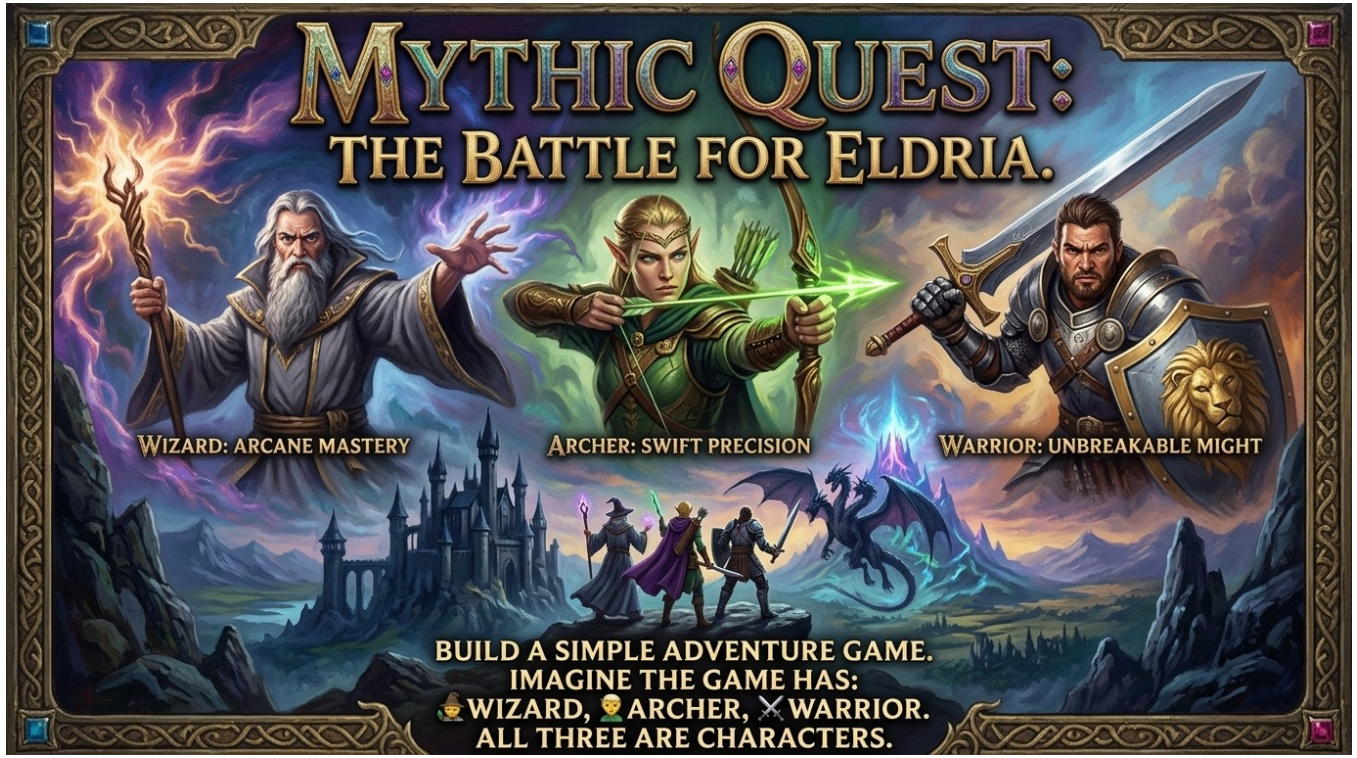
EXTENDS

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

جامعة الأمير سلطان
PRINCE SULTAN UNIVERSITY

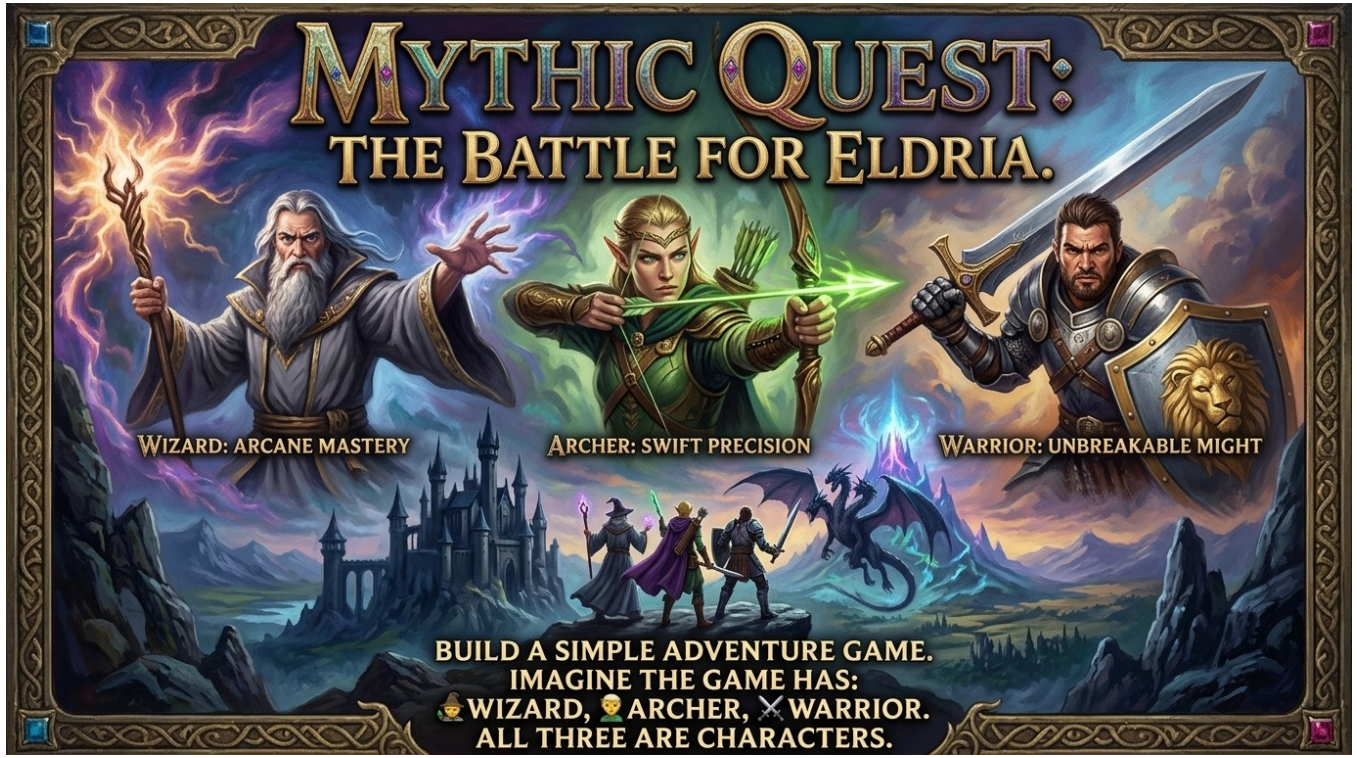
IDEATING INHERITANCE

- Let's build a **Adventure Game**.
- Imagine the game has:
- 🧙‍♂️ **Wizard**
- 🧝‍♂️ **Archer**
- ⚔️ **Warrior**
- All three are **characters**, so they share some common attributes and methods. But each character attacks differently. This is exactly what **inheritance** is designed for.



IDEATING INHERITANCE

- What things do all characters in a game have in common?
  - `Name`
  - `Health`
  - `Level`
  - `Position`
  - `Attack`
  - `Move`

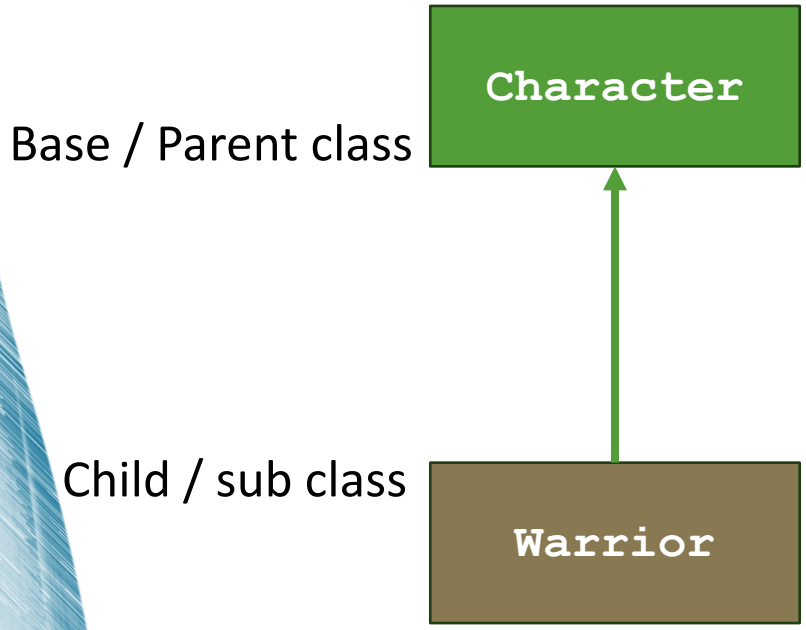


THE CHARACTER CLASS

```
class Character {
  int health;
  String name;
  void move(){
    System.out.println(name + " is moving");
  }
  void attack() {
    System.out.println(name + " is attacking");
  }
}
```

THE WARRIOR CLASS


```
class Warrior extends Character {
  void useSword() {
    System.out.println(name + " attacks with a sword!");
  }
}
```



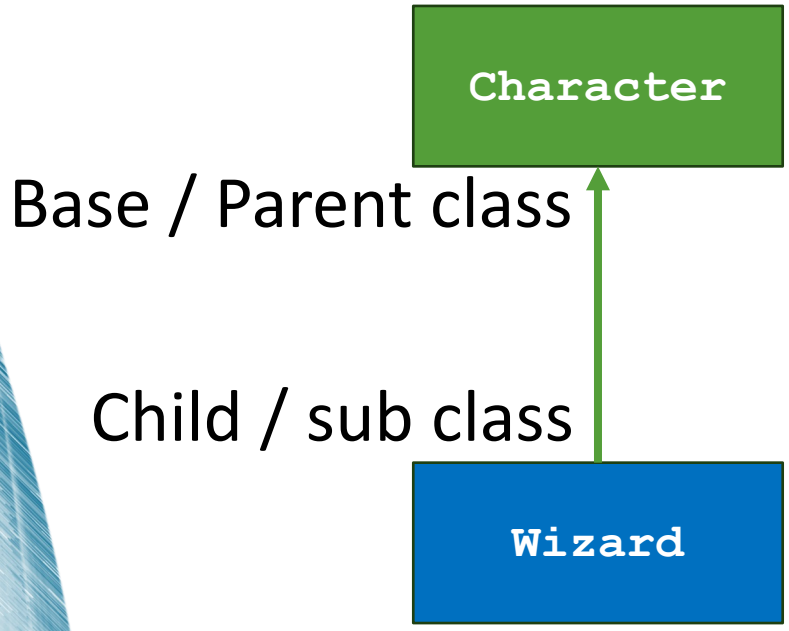
`Warrior` is ***inherited*** from **`Character`**.
Note the keyword **`extends`**

Warrior automatically get **health** and **name** variables from Character class.
It also gets the `move()` and `attack()` methods from Character class.

Warrior has its own `useSword()` method.

THE WIZARD CLASS


```
class Wizard extends Character {
  void fireBall() {
    System.out.println(name + " throws a fireball!");
  }
}
```

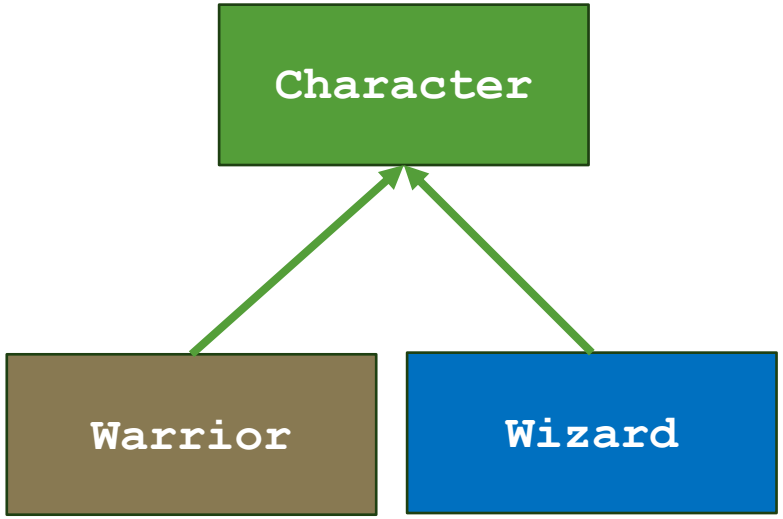


`Wizard` is ***inherited*** from **`Character`**.
Note the keyword **`extends`**

Wizard automatically get **health** and **name** variables from Character class.
It also gets the `move()` and `attack()` methods from Character class.

Wizard has its own `fireBall()` method.

INHERITANCE



Warrior is-a **Character**
Wizard is-a **Character**

Wizard is ***inherited*** from **Character**.
Warrior is ***inherited*** from **Character**.

Wizard and Warrior automatically get **health** and **name** variables from Character class. They also gets the `move()` and `attack()` methods from Character class.

Warrior has its own `useSword()` method.
Wizard has its own `fireBall()` method.

We can define variables/attributes/methods for each subclass.

TESTING THE GAME

```
class Character {
 public class Game {
   public static void main(String[] args) {
    Warrior w = new Warrior();
    w.name = "Thor";
    w.health = 100;
    w.move();
    w.attack(); //inherited from Character
    w.useSword();
    Wizard a = new Wizard();
    a.name = "Robin";
    a.health = 80;
    a.move();
    a.attack(); // inherited from Character
    a.fireBall();
   }
}
```

Output:

```
Thor is moving
Thor is attacking
Thor attacks with a sword!
Robin is moving
Robin is attacking
Robin throws a fireball!
```

THE WARRIOR CLASS


```
class Warrior extends Character {
  void useSword() {
    System.out.println(name + " attacks with a sword!");
  }
  @Override
  void attack() {
    System.out.println(name + " swings a sword!");
  }
}
```

```
class Character { ...attributes/methods...
void attack() {
    System.out.println(name + " is attacking");
  }
}
```

Warrior is ***inherited*** from **Character**.

Warrior defines `attack()` method.
Character already has `attack()` method.

We use the keyword `@Override`

THE WIZARD CLASS


```
class Wizard extends Character {
  void fireBall() {
    System.out.println(name + " throws a fireball!");
  }
 @Override
  void attack() {
    System.out.println(name + " throws fireBall!");
  }
}
```

```
class Character { ...attributes/methods...
void attack() {
    System.out.println(name + " is attacking");
  }
}
```

Wizard is ***inherited*** from **Character**.

Wizard defines `attack()` method.
Character already has `attack()` method.

We use the keyword `@Override`

TESTING THE GAME

```
class Character {
 public class Game {
   public static void main(String[] args) {
    Warrior w = new Warrior();
    w.name = "Thor";
    w.attack(); //using @override attack() from Warror
    Wizard a = new Wizard();
    a.name = "Robin";
    a.attack(); //using @override attack() from Wizard
  }
}
```

Expected output:

```
Thor is attacking
Robin is attacking
```

Actual output:

```
Thor swings a sword!
Robin throws fireball!
```

SUPER

- To call a parent class method or attribute in Java, we use **`super.`**
- **`super`** means go to my parent.

```
class Character {
void attack() {
    System.out.println(name + " is attacking");
  }
}

class Warrior extends Character {
@Override
  void attack() {
    System.out.println(name + " swings a sword!");
  }
}
```

```
//Tester program
Warrior w = new Warrior();
w.attack();

//should print
Warrior swings a sword!
```

SUPER

- To call a parent class method or attribute in Java, we use **`super.`**
- **`super`** means go to my parent.

```
class Character {
void attack() {
    System.out.println(name + " is attacking");
  }
}

class Warrior extends Character {
@Override
  void attack() {
    super.attack();
    System.out.println(name + " swings a sword!");
  }
}
```

```
//Tester program
Warrior w = new Warrior();
w.attack();

//would print
Warrior is attacking
Warrior swings a sword!
```

Note

`this.attack()` → call my (Warrior) method

`super.attack()` → call parent's (Character) method

`Super` can be used with attributes of parent class

SUPER

- Using **super** we can modify the attributes of the parent class.

```
class Character {
   int health;
   String name;
   void attack() {
     System.out.println(name + " is attacking");
   }
}
class Warrior extends Character {
@Override
   void attack() {
     super.name = "Alan";
     super.health = 99;
   }
}
```

Character
name = Alan
health = 99

Warrior
name = Thor
Health 85;

TYPES OF INHERITANCE IN JAVA

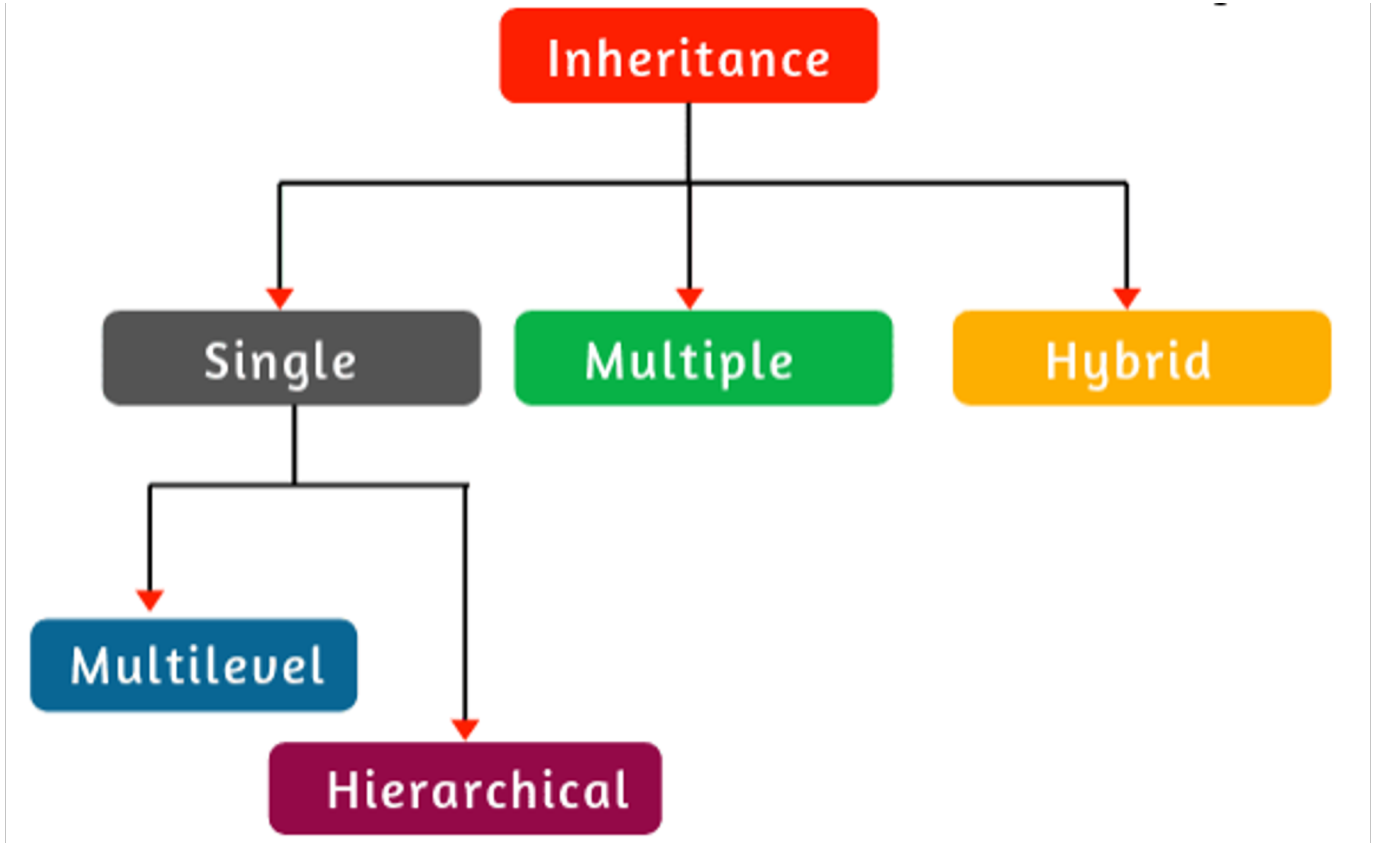
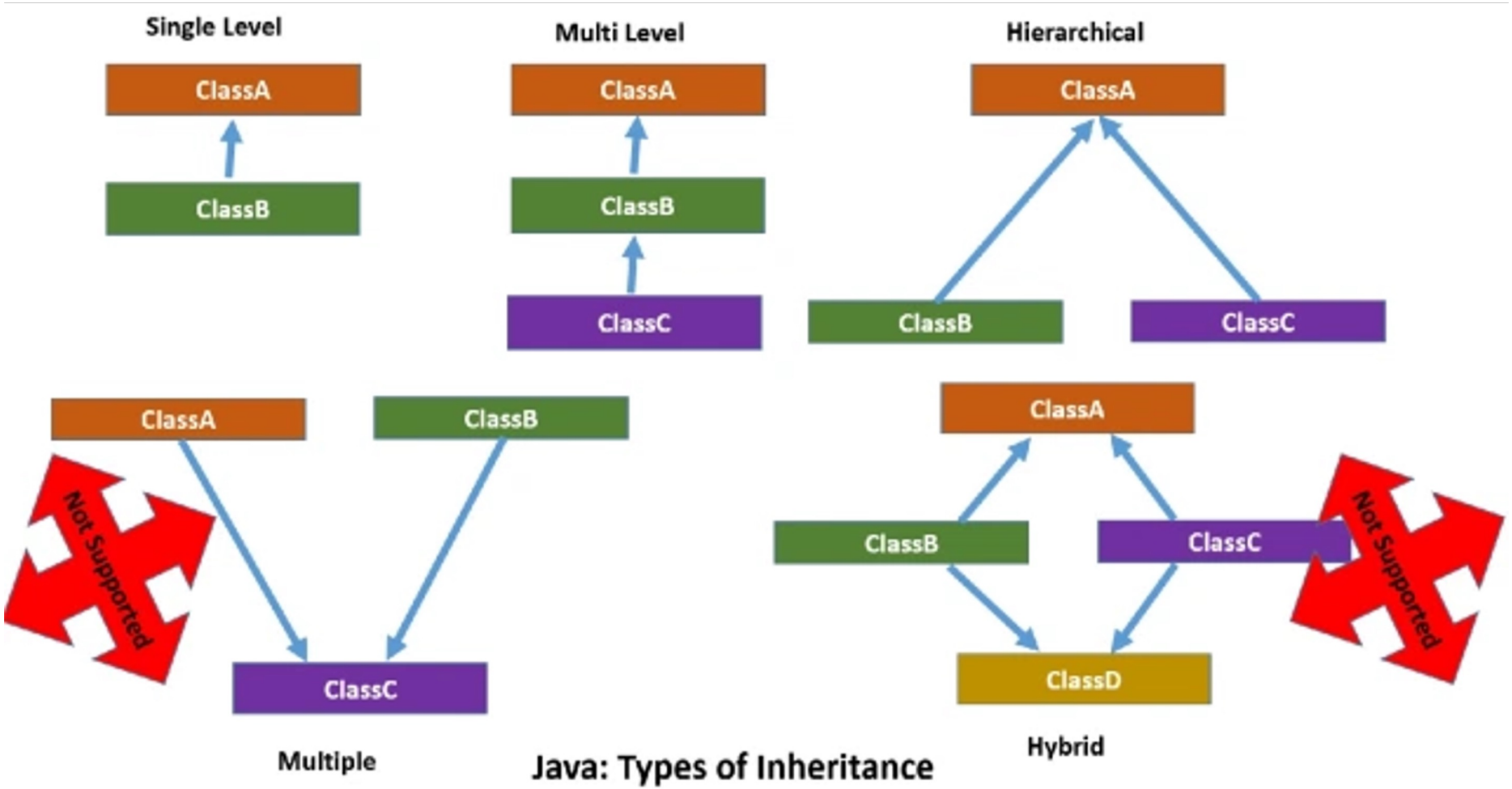


Fig: Classification of Inheritance in Java

TYPES OF INHERITANCE IN JAVA



CS102 Programming II

POLYMORPHISM

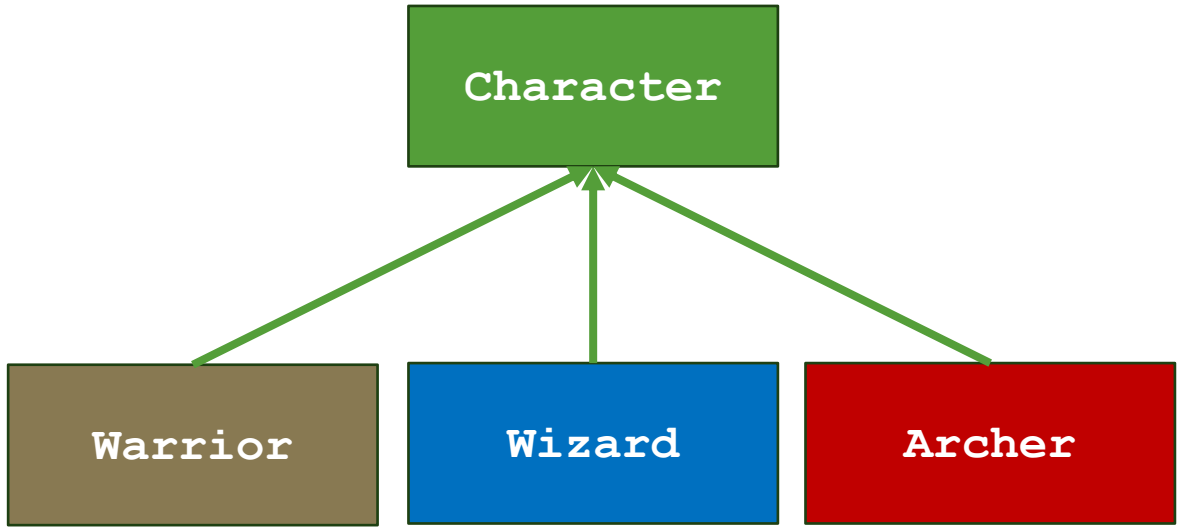
Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

جامعة الأمير سلطان
PRINCE SULTAN UNIVERSITY

POLYMORPHISM

"One thing, many forms"

In Java, polymorphism allows a **parent-class reference** to refer to different child objects, and Java automatically calls the appropriate overridden method.



We can use the `Character` object to refer to child classes!

POLYMORPHISM

```
class Character {
  void attack() {
    System.out.println(”Character attacks”);
  }
}
```

```
class Warrior extends Character {
  @Override
  void attack() {
    System.out.println(”Warrior swings a sword!");
  }
}
```

POLYMORPHISM

```
Class Wizard extends Character {
  @Override
  void attack() {
    System.out.println(”Wizard throws a fireball!”);
  }
}
```


```
class Archer extends Character {
  @Override
  void attack() {
    System.out.println(”Archer shoots a arrow!");
  }
}
```

POLYMORPHISM

```
Class Tester {
  public static void main(String [] args){
    Character c1 = new Warrior();
    Character c2 = new Wizard();
    Character c3 = new Archer();
    c1.attack();
    c2.attack();
    c3.attack();
  }
}
```

Observe:

Warrior object is stored in c1

Wizard object is stored in c1

Archer object is stored in c1

```
//Output
Warrior swings a sword!
Wizard throws a fireball!
Archer shoots a arrow!
```

Java looks at the **actual object**, not just the reference type.

We wrote the **same code** three times; But each object performs the operation differently
That's **polymorphism**.

POLYMORPHISM

```
Class Tester {
  public static void main(String [] args){
    Character [] Players = { new Warrior(), new Wizard(), new Archer() };
    for(int i=0; i<Players.length; i++)
      Players[i].attack();
  }
}
```

Same Output!

We don't need to write:

```
if (player is Warrior)
    ...
else if (player is Archer)
    ...
else if (player is Wizard)
    ...
```

Java automatically chooses the correct attack()

Polymorphism allows us to use a common parent type to work with different child objects, while each child provides its own implementation of the same method.

POLYMORPHISM

- **Static Polymorphism**
  - Method Overloading
  - Compile-time
  - Same method name, different parameters
- **Dynamic Polymorphism**
  - Method Overriding
  - Runtime
  - Parent reference → Child object
  - Actual object's method is executed
- To note: **constructors cannot be overridden**, although they can be overloaded.

CS102 Programming II

POLYMORPHISM

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

جامعة الأمير سلطان
PRINCE SULTAN UNIVERSITY