

CS102

Programming II

CLASS (PART - II)

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

TOPICS

- Inheritance
 - **extends**
 - **super**
- **equals ()** and **instanceof**
- Polymorphism

CS102

Programming II

INHERITANCE

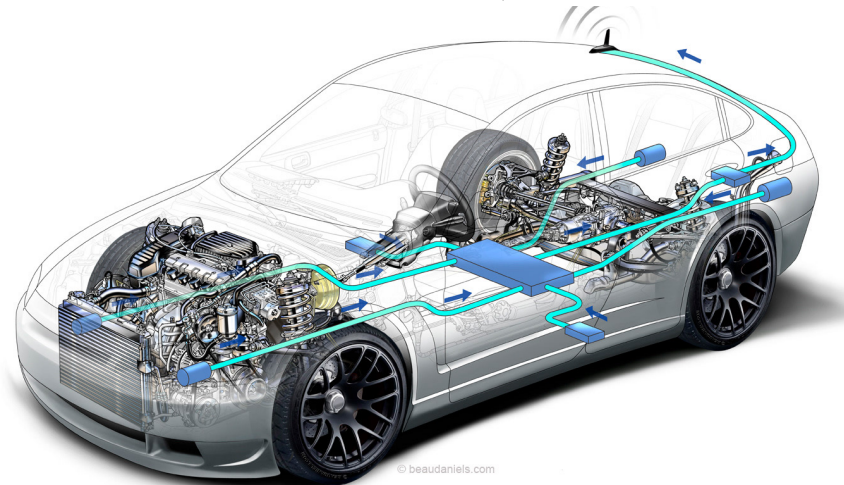
Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

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;  
    }  
}
```



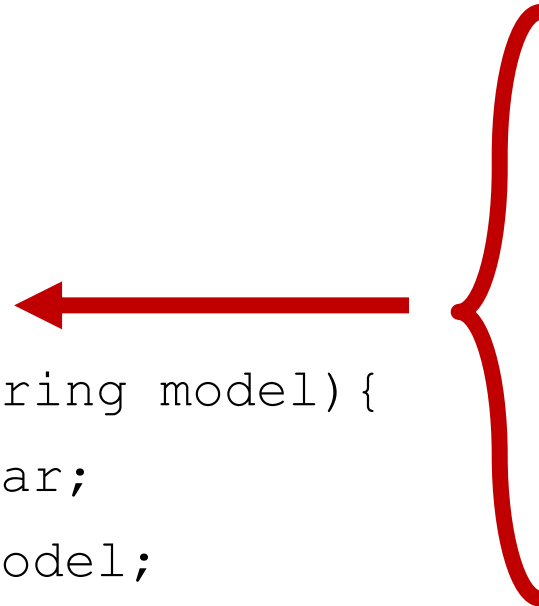
```
public class Engine {  
    int hp;  
    String fuel;  
    Engine() { //initialize attributes  
    }  
    Engine(int hp, String fuel){  
        this.hp = hp;  
        this.fuel = fuel;  
    }  
}
```



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

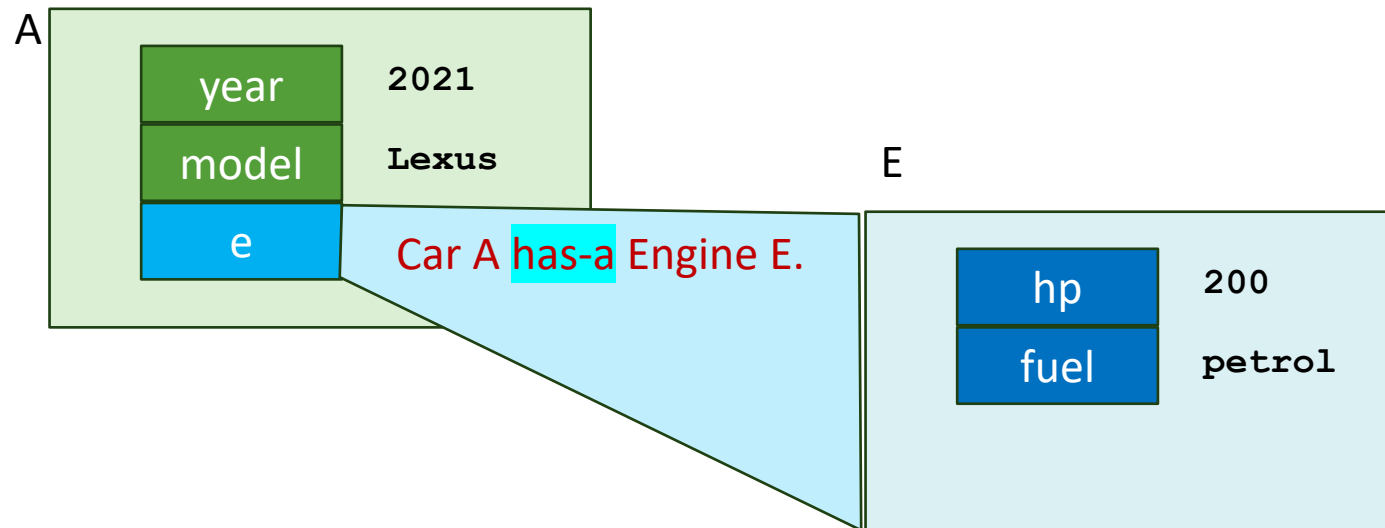
```
public static void main(String [] args){
```

```
    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:

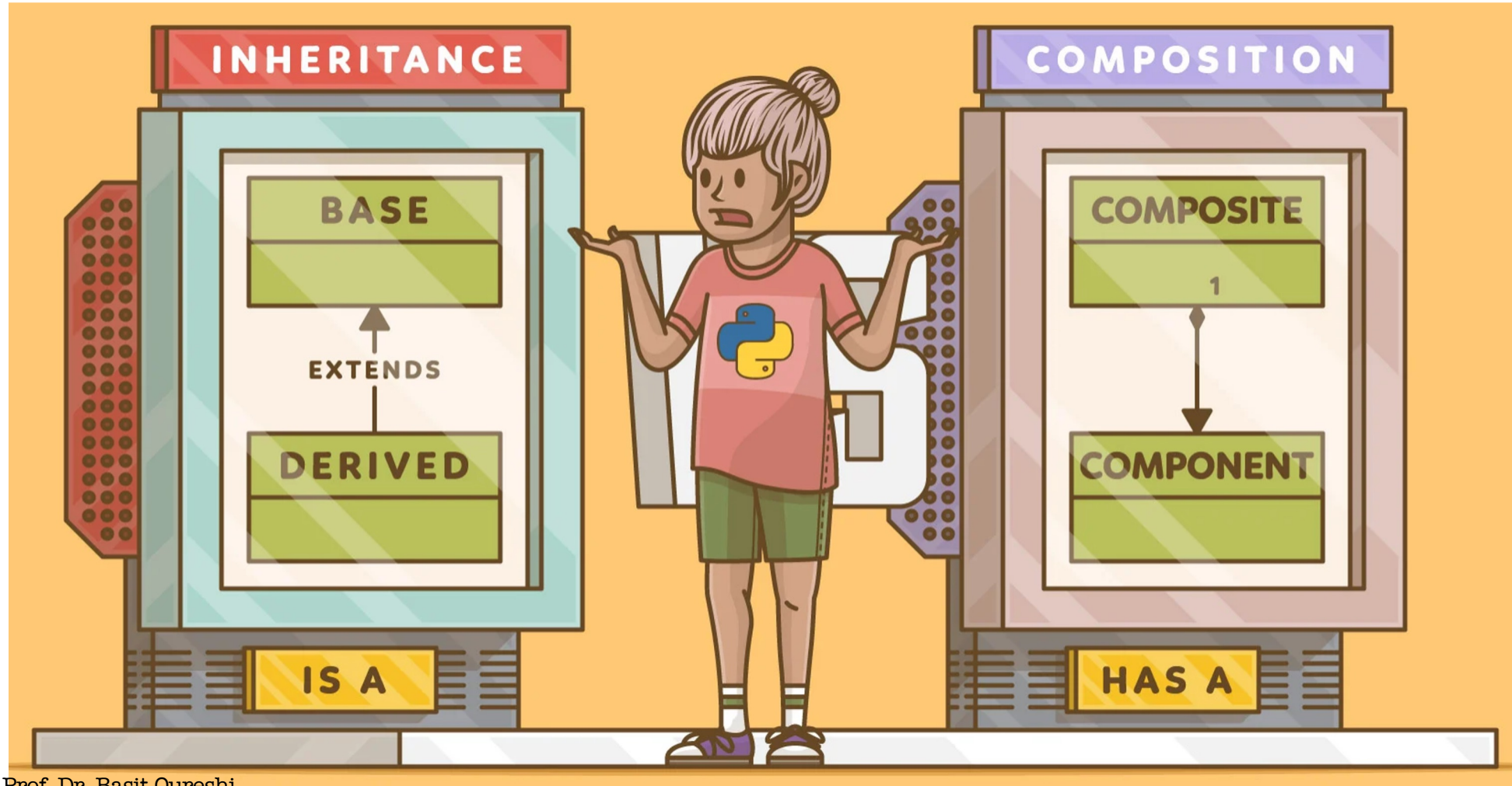
A **Student** is-a **Person**. A **Professor** is-a **Person**

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

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

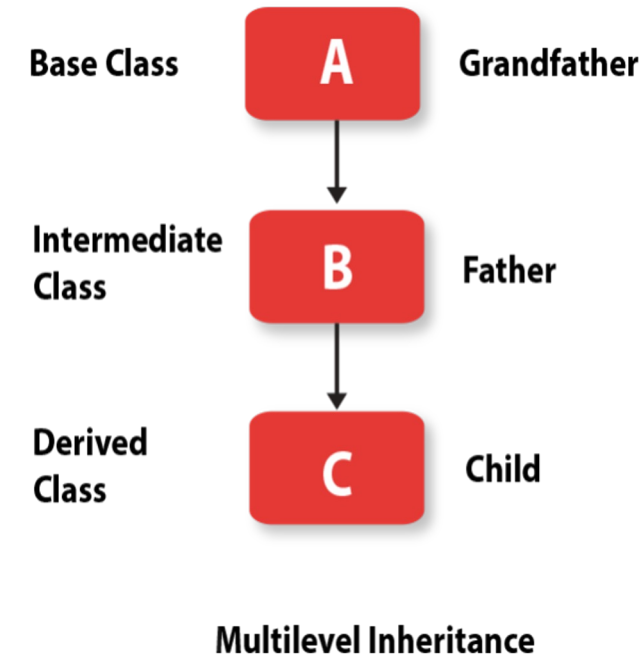
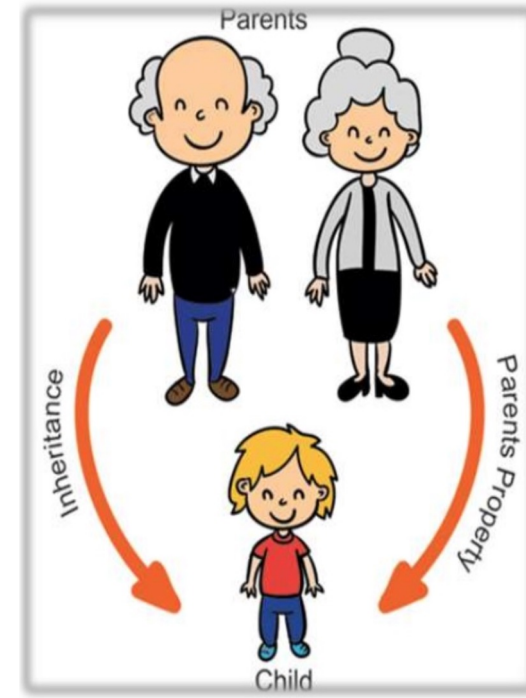
INHERITANCE

- 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 **superclass**, or **baseclass**
- The derived class is called the **child class** or **subclass**
- 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

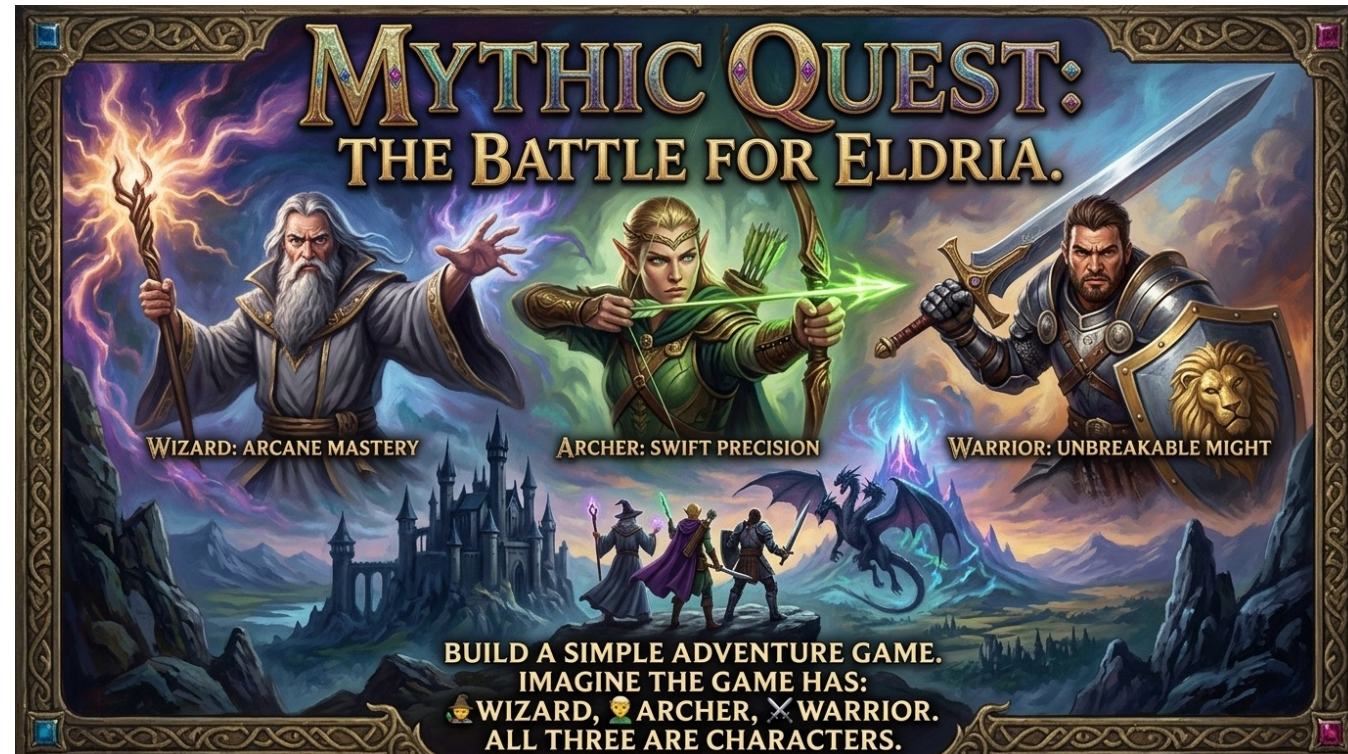
Programming II

EXTENDS

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

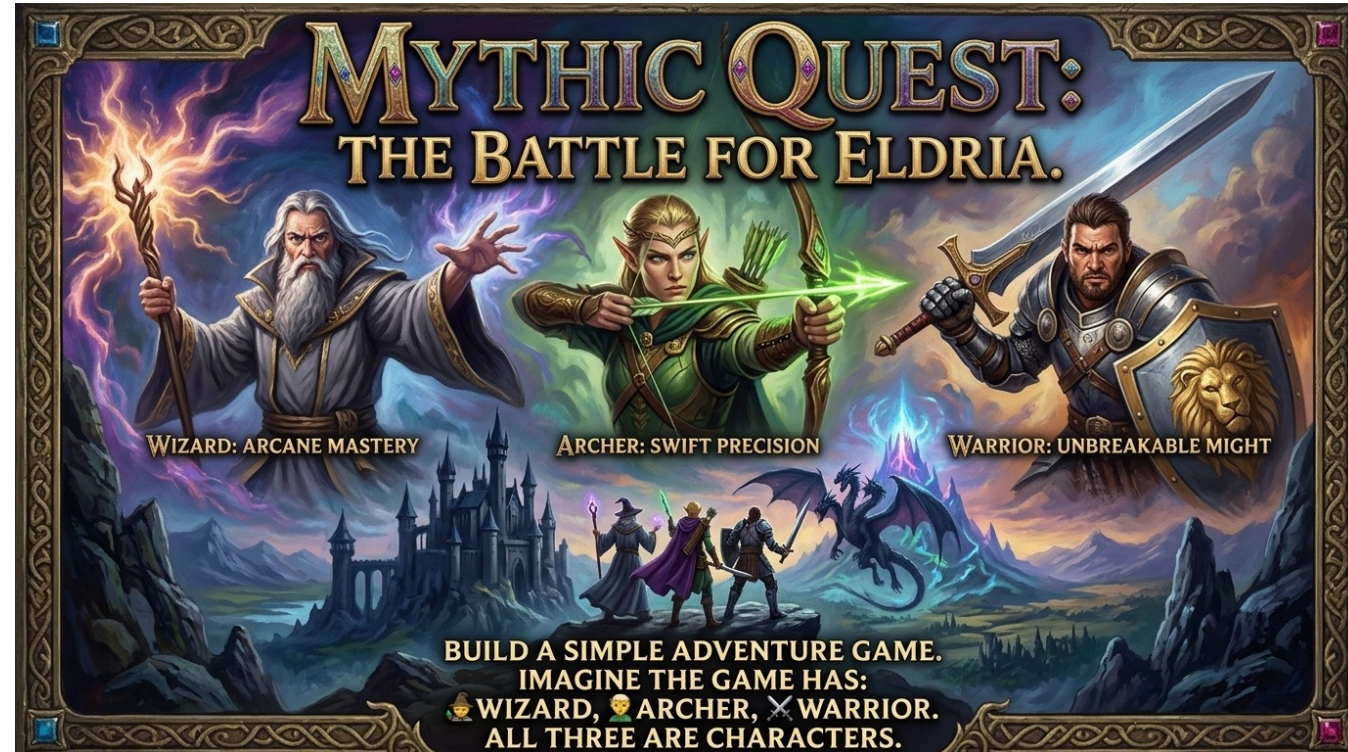
IDEATING INHERITANCE

- 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!");  
    }  
}
```



Character

Base / Parent class

Warrior

Child / sub class

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

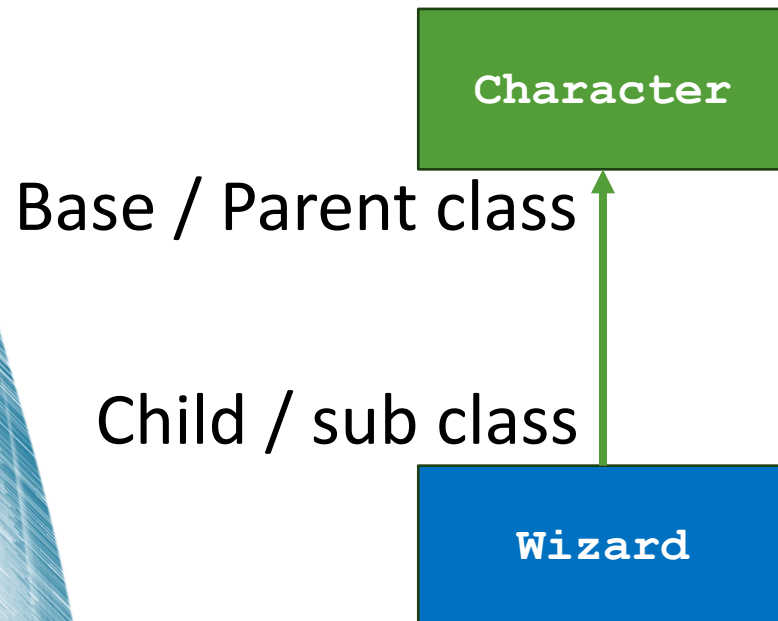
Note the keyword **extends**

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!");  
    }  
}
```



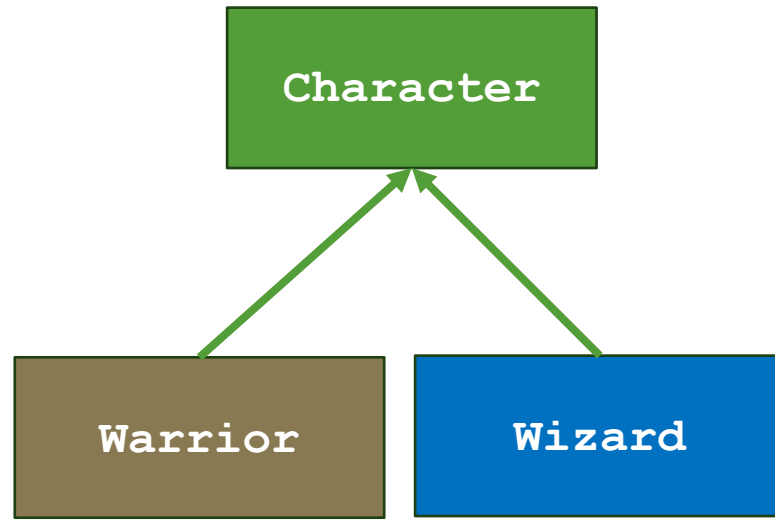
Wizard is *inherited* from **Character**.

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 get 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

```
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 Warrior  
            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;
--

THIS()

- Using **this()** we can call the constructor of the same class.

```
class Character {  
    int health;  
    String name;  
    Character() {  
        this(100, "unknown");  
    }  
    Character(int health, String name) {  
        this.health = health;  
        this.name = name;  
    }  
}
```

Note: `this(100,"unknown")` will call the overloaded constructor in the same class. It will initialize the values of `health=100` and `name="unknown"`

TYPES OF INHERITANCE IN JAVA

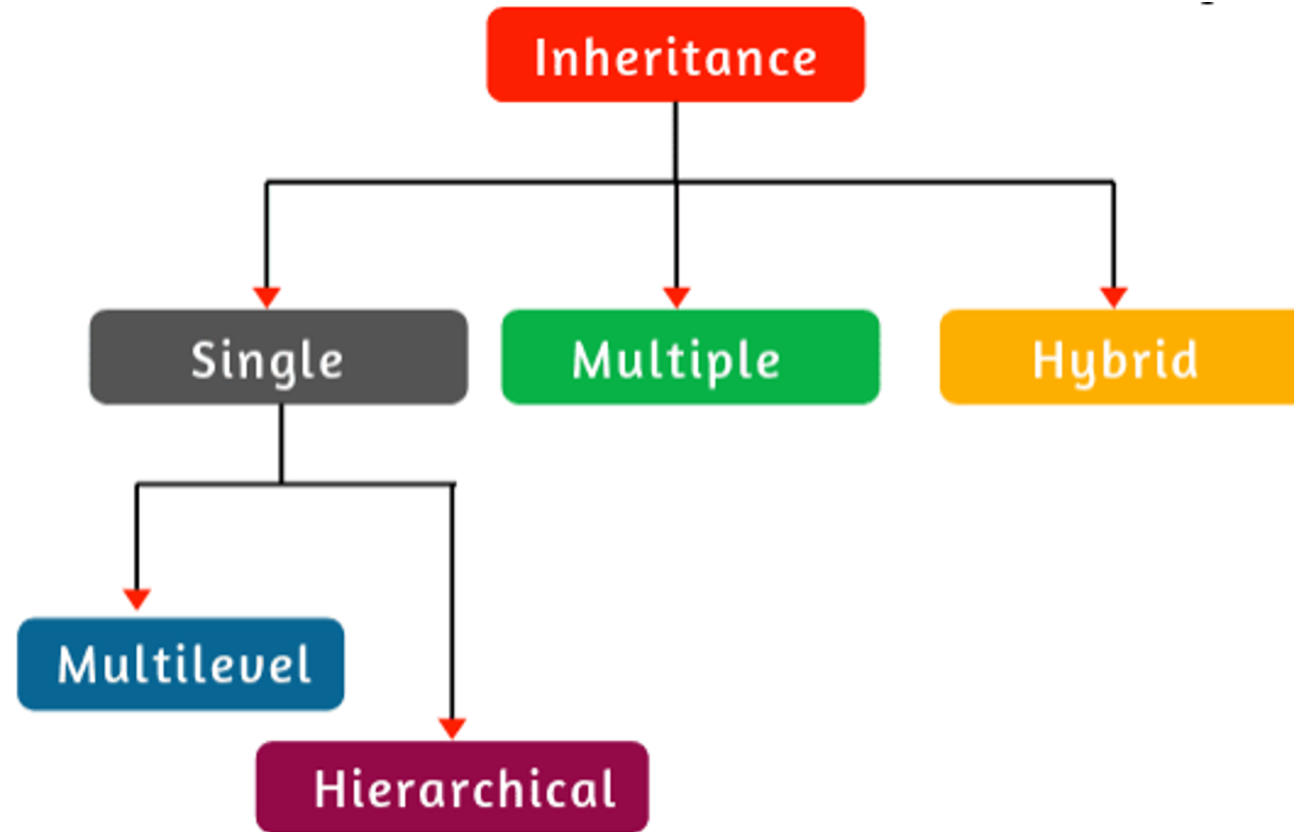
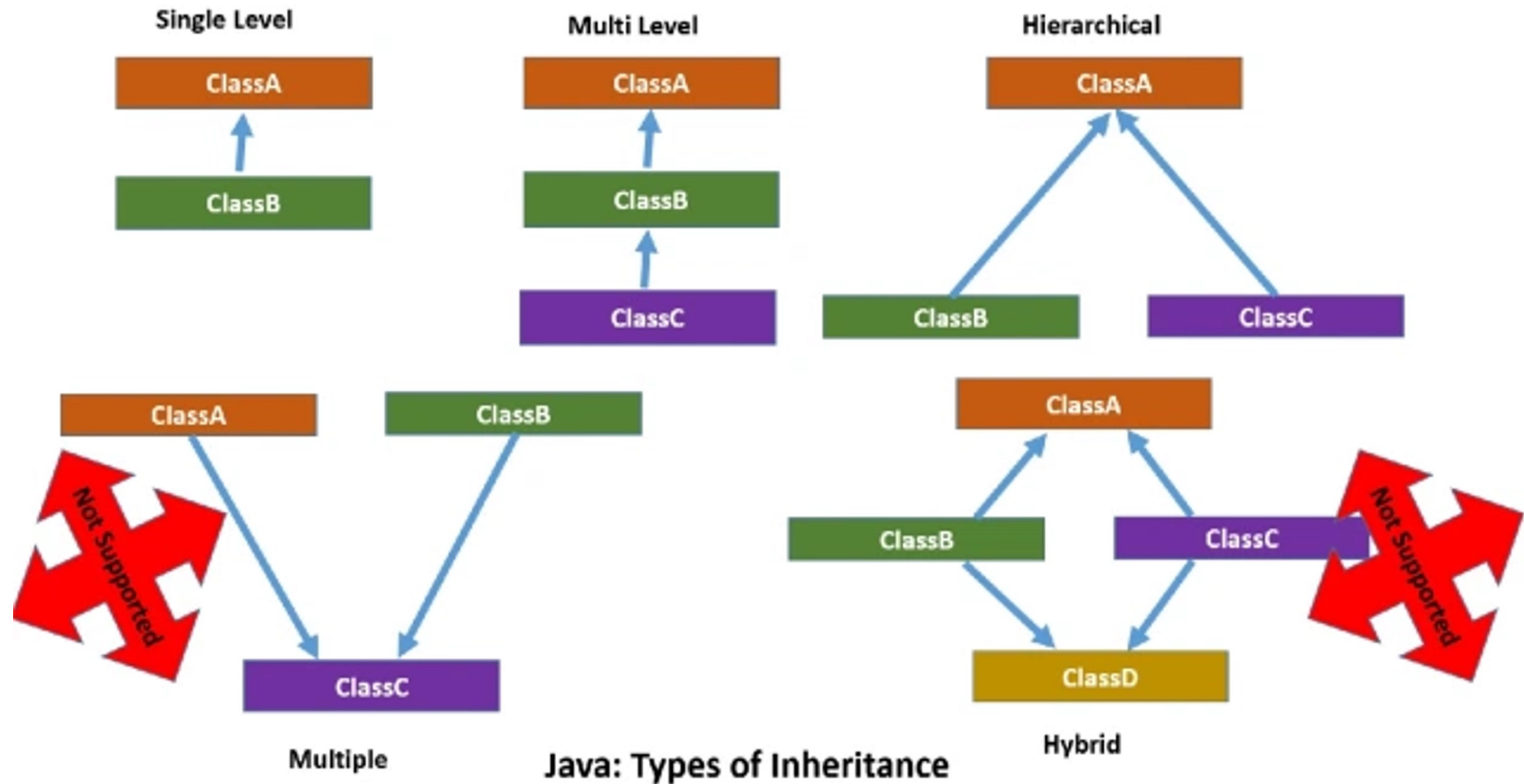


Fig: Classification of Inheritance in Java

TYPES OF INHERITANCE IN JAVA



CS102

Programming II

EQUALS AND INSTANCEOF

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

CLASS **Object**

- In java, all types of objects have a **superclass** named **Object**.
 - Every class implicitly extends `Object`.
// we study extends later!!
- The `Object` class defines several methods:
 - `public String toString()`
Used to print the object.
 - `public boolean equals(Object other)`
Compare the object to any other for equality.

CLASS **Object**

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

```
Object o1 = new Car(2020, "G.M.C.");  
Object o2 = "hello there";  
Object o3 = new Scanner(System.in);
```

- An `Object` variable only knows how to do general things.

```
String s = o1.toString();           // ok  
int len = o2.length();              // error  
String line = o3.nextLine();        // error
```

- You can write methods that accept an `Object` parameter.

```
public void example(Object o) {  
    if (o != null) {  
        System.out.println("You passed " + o.toString());  
    }  
}
```

CLASS **Object**

- The == operator does not work well with objects.
 - == compares references to objects, not their state.
- Example:

```
Car c1 = new Car(2020, "GMC");  
Car c2 = new Car(1999, "Ford");  
if (c1 == c2) {    // false  
    System.out.println("equal");  
}
```

equals

- The `equals` method compares the state of objects.

```
if (str1.equals(str2)) {  
    System.out.println("the strings are equal");  
}
```

- But if you write a class, its `equals` method behaves like `==`

```
if (c1.equals(c2)) {    // false :- (  
    System.out.println("equal");  
}
```

This is the behavior we inherit from class **Object**. Java doesn't understand how to compare `Car` by default.

TYPE-CASTING OBJECTS

- Solution: *Type-cast* the object parameter to a `Car`.

```
public boolean equals(Object o) {  
    Car other = (Car) o; ←type casting  
    return year == o.year && model == o.model;  
}
```

- Casting objects is different than casting primitives.
 - Tells the compiler to *assume* that `o` refers to a `Car` object.

instanceof

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

- Asks whether a variable refers to an object of a given type.
 - Used as a boolean test.
 - Examples:

```
String s = "hello";  
Car c1 = new Car();
```

expression	result
s instanceof Car	false
s instanceof String	true
c1 instanceof Car	true
c1 instanceof String	false
c1 instanceof Object	true
s instanceof Object	true
null instanceof String	false

instanceof

```
public boolean equals(Object o) {  
    if (o instanceof Car) {  
        // o is a Car; cast and compare it  
        Car other = (Car) o;  
        return year==o.year && model == o.model;  
    } else {  
        // o is not a Car; cannot be equal  
        return false;  
    }  
}
```

CORRECT USE OF EQUALS

- The == operator compares:
 - Memory addresses (references)
 - NOT object content

```
Car c1 = new Car(2020, "GMC");  
Car c2 = new Car(1999, "Ford");  
if (c1 == c2) { // false  
    System.out.println("Same memory");  
}  
if (c1.equals(c2)) { // true  
    System.out.println("Different memory");  
}
```

CS102

Programming II

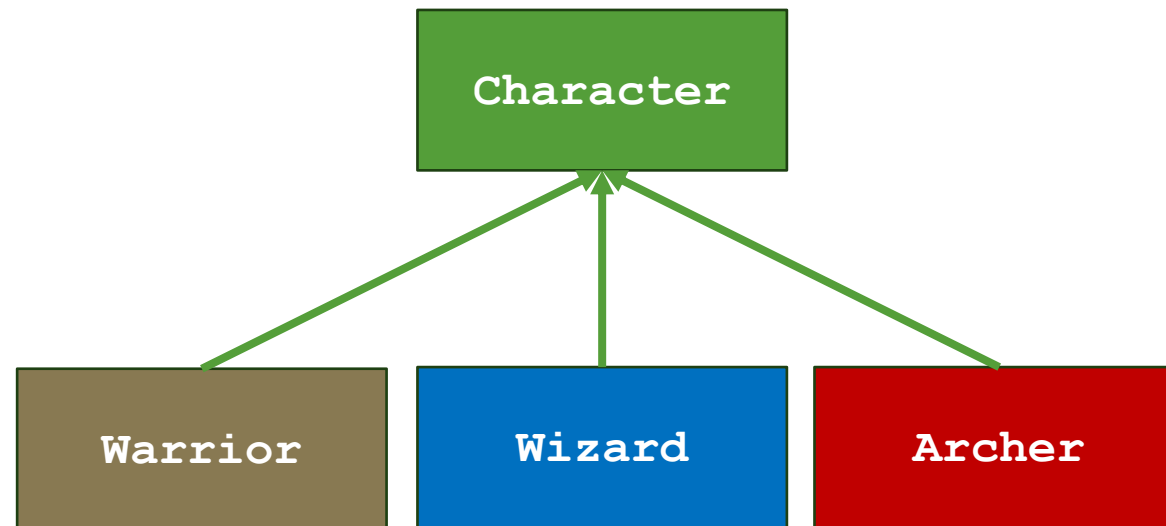
POLYMORPHISM

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

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.**

CS 102

Programming II

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