Inheritance

- Inheritance: is the process by which one new class, called the derived class, is created from another class, called the base class.
- The derived class is also called: subclass or child class.
- The base class is also called: superclass or parent class.

Motivation: In real life objects have a hierarchical structure:

- We want to do the same with our program objects.

Inheritance cont.

- What are some properties of a Person?
  - name, height, weight, age

- How about a Student?
  - ID, major

- Does a Student have a name, height, weight, and age?
  - Student inherits these properties from Person.
The is-a relationship

- This inheritance relationship is known as an is-a relationship
- A Doctoral student is a Grad student
- A Grad student is a Student
- A Student is a Person

Is a Person a Student? – Not necessarily!

Base Class and Derived class

- General class is called a base class
  - Also called a parent class or a superclass
  - Examples:
    - Person, Animal
- A specialized class that inherits properties from a base class is called a derived class
  - Also called a child class or a subclass
  - Examples:
    - Student is-a Person
    - Employee is-a Person
    - Car is-a vehicle

Derived class

- Derived (Child) class can be base (parent) class
- Example:
  - Student is a child class of Person
  - Student is also the parent class:

Why inheritance is useful

- Enables you to define shared properties and actions once
- Derived classes can perform the same actions as base classes without having to redefine the actions
- If desired, the actions can be redefined – method overriding
Dissecting the Student Class

- **Extends**: To specify that Student is a derived class (subclass) of Person we add the descriptor "extends" to the class definition:
  ```java
  public class Student extends Person {
      ...}
  ``

- Notice that a Student class
  - Inherits everything from the Person class
  - A Student IS-A Person (wherever a Person is needed, we can use a Student).

- **super()**: When initializing a new Student object, we need to initialize its base class (or superclass). This is done by calling `super(...)`. For example, `super(name)` invokes the constructor `Person(name)`
  - `super(...)` must be the first statement of your constructor
  - If you **do not** call `super()`, Java will automatically invoke the base class's default constructor
  - What if the base class's default constructor is **undefined**? Errors
  - You must use "super(...)", not "Person(...)".

Student Class

```java
public class Student extends Person {
    public Student() {
        super();
    }
    public Student(String name, int id) {
        super(name);
    }
    public void setId(int id) {
        id = id;
    }
    public int getId() {
        return id;
    }
    @Override
    public String toString() {
        return "Id: " + id + "\nName: " + getName();
    }
}
```

Memory Layout and Initialization Order

- When you create a new derived class object:
  - Java allocates space for both the base class instance variables and the derived class variables
  - Java initializes the base class variables first, and then initializes the derived class variables

- **Example**:
  ```java
  Person ted = new Person("Ted Goodman");
  Student bob = new Student("Bob Goodstudent", 100);
  ```

- The `Person` constructor finishes it off.
Inheritance

- **Inheritance:** Since Student is derived from Person, a Student object can invoke any of the Person methods, it **inherits** them

```java
Student bob = new Student("Bob Goodstudent", 100);
String bobName = bob.getName();
bob.setName("Robert Goodstudent");
System.out.println("Bob's new info: " + bob.toString());
```

- **A Student "is a" Person:**
  - By inheritance a Student object is also a Person object. We can use a Student reference anywhere that a Person reference is needed
  ```java
  Person robert = bob; // Okay: A Student is a Person
  ```
  - We cannot reverse this. (A Person need not be a Student.)
  ```java
  Student bob2 = robert; // Error! Cannot convert Person to Student
  ```

Overriding Methods

- **New Methods:** A derived class can define **entirely new** instance variables and new methods (e.g. gpa and getGpa)
- **Overriding:** A derived class can also **redefine existing** methods

```java
public class Person {
  public String toString() {
    ...}
}
public class Student extends Person {
  public String toString() {
    ...}
}
Student bob = new Student("Bob Goodstudent", 100);
System.out.println("Bob's info: " + bob);
```

Overriding and Overloading

- **Don't confuse method overriding with method overloading.**
  - **Overriding:** occurs when a derived class defines a method with the **same name and parameters** as the base class.
  - **Overloading:** occurs when two or more methods have the **same name**, but have **different parameters** (different signature).

Example:

```java
public class Person {
  public void setName(String n) {
    name = n;
  }
  ...
}
public class Faculty extends Person {
  public void setName(String n) {
    super.setName("The Evil Professor " + n);
  }
  public void setName(String first, String last) {
    super.setName(first + " " + last);
  }
}
```

Overriding Variables: Shadowing

- **We can override methods, can we override instance variables too?**
- **Answer:** Yes, it is possible, but **not recommended**
  - Overriding an instance variable is called **shadowing**, because it makes the base instance variables of the base class inaccessible. (We can still access it explicitly using `super.varName`).

```java
public class Person {
  String name;
}
public class Staff extends Person {
  String name;
  // ...
  // name refers to Staff's name
}
```

- This can be **confusing** to readers, since they may not have noticed that you redefined name. Better to just pick a new variable name
super and this

- **super**: refers to the base class object
  - We can invoke any base class constructor using `super(...)`.
  - We can access data and methods in the base class (Person) through `super`. E.g., `toString()` and `equals()` invoke the corresponding methods from the Person base class, using `super.toString()` and `super.equals()`.

- **this**: refers to the current object
  - We can refer to our own data and methods using "this," but this usually is not needed.
  - We can invoke any of our own constructors using `this(...). As with the super constructor, this can only be done within a constructor, and must be the first statement of the constructor.

Example:
```java
public Toy(Toy toy) {
    this(toy.name, toy.releasedYear);
}
```