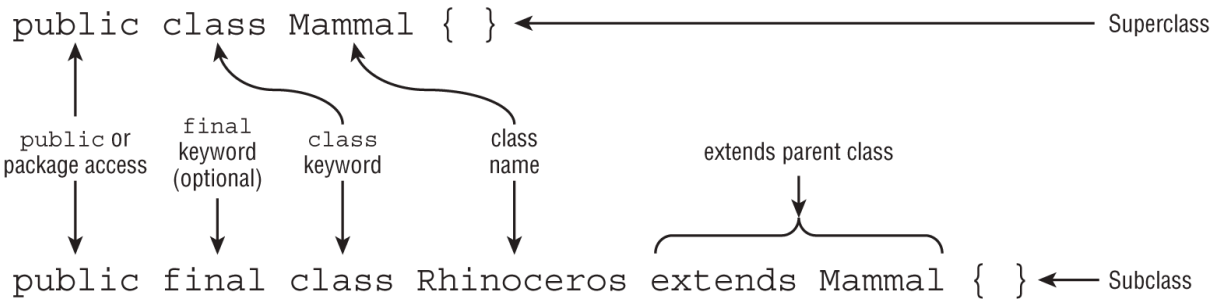
Capitolul 6

📌 Inheritance → process by which a subclass automatically includes certain members of the class, including primitives, objects or methods, defined in the parent class.



- When one class inherit from a parent class, all public and protected members are available as part of the child class. Not also the private members.

final	the class may not be extended
abstract	the class is abstract, may contain abstract methods, and requires a concrete subclass to instantiate
sealed	A subclass may only be extended by a specific list of classes
non-sealed	A subclass of a sealed class permits potentially unnamed subclasses
static	Used for static nested classes defined within a class

Java supports ***single inheritance*** → a class may inherit from only one direct parent class.

All classes inherit the ***Object class.***

PLS, use ***this.*** for the members of class.

SUPER → keyword to access the members from the parent class, can use also keyword ***this***

CONSTRUCTOR → special method that matches the name of the class and has no return type.

```
public class Bunny{
   public Bunny(){
      System.out.println("hello bunny");
   }
}
```

Like method parameters, constructor parameters can be any valid class, array, or primitive type, including generics, but no *var*.

Also, a class can have multiple constructors, as long as each constructor has an unique signature.

this() → when u have overloaded constructors, u can call other constructor. and be sure that u call first in your constructor.

```
public class Hamster{
   private int age;
   private String color;

   public Hamster(String color, int age){
      this.color=color;
      this.age=age;
   }

   public Hamster(int age){
      this("brown", age);
   }
}
```

super() → used to call parent constructor, same rule like this().

```
public class Animal {
  private int age;
  public Animal(int age) {
```

```
    super();     // Refers to constructor in java.lang.Object
    this.age = age;
  }
}

public class Zebra extends Animal {
  public Zebra(int age) {
    super(age);  // Refers to constructor in Animal
  }
  public Zebra() {
    this(4);     // Refers to constructor in Zebra with int argument
  }
}
```

`Overriding a method → occurs when a subclass declares a new implementation for an inherited method with the same signature and compatible return type.`

```
public class Marsupial {
  public double getAverageWeight() {
    return 50;
  }
}

public class Kangaroo extends Marsupial {
  public double getAverageWeight() {
    return super.getAverageWeight()+20;
  }
  public static void main(String[] args) {
    System.out.println(new Marsupial().getAverageWeight());  // 50.0
    System.out.println(new Kangaroo().getAverageWeight());   // 70.0
  }
}
```

`To override a method, you must follow a number of rules. The compiler performs the following checks when you override a method:`

1. The method in the child class must have the same signature as the method in the parent class.
2. The method in the child class must be at least as accessible as the method in the parent class.
3. The method in the child class may not declare a checked exception that is new or broader than the class of any exception declared in the parent class method.
4. If the method returns a value, it must be the same or a subtype of the method in the parent class, known as *covariant return types*.

Redeclaring *private* method → u can redeclare a new method in the class with the same or modified signature as the method in the parent class, being a separate and independent method.

Hiding static methods → occurs when a child class defines a static method with the same name and signature as an inherited static method defined in a parent class.

Hiding variables → occurs when a child class defines a variable with the same name as an inherited variable defined in the parent class. This creates two distinct copies of the variable within an instance of the child class: one instance defined in the parent class and one defined in the child class.

Abstract class

📌 Abstract class → is a class declared with the *abstract* modifier that can not be instantiated directly and may contain abstract methods.

```
public abstract class Canine {}

public class Wolf extends Canine {}
```

```
public class Fox extends Canine {}
public class Coyote extends Canine {}
```

An abstract class can have abstract methods.

> 📌 Abstract method → method declared with the abstract modifier that does not define a body. Put another way, abstract method = forces subclasses to override the method.

We use this for polymorphism → can guarantee that some version will be available on an instance without to specify what version is in the abstract parent class.

```
public abstract class Canine {
  public abstract String getSound();
  public void bark() { System.out.println(getSound()); }
}

public class Wolf extends Canine {
  public String getSound() {
    return "Wooooooof!";
  } }

public class Fox extends Canine {
  public String getSound() {
    return "Squeak!";
  } }

public class Coyote extends Canine {
  public String getSound() {
    return "Roar!";
  } }

-------------------------------------------
public static void main(String[] p) {
  Canine w = new Fox();
```

```
    w.bark();  // Squeak!
}
```

There are some rules you need to be aware of:

- Only instance methods can be marked `abstract` within a class, not variables, constructors, or `static` methods.
- An abstract class can include zero or more abstract methods, while a non-abstract class cannot contain any.
- A non-abstract class that extends an abstract class must implement all inherited abstract methods.
- Overriding an abstract method follows the existing rules for overriding methods that you learned about earlier in the chapter.

Declaring Abstract Methods → is always declared without a body.
It also includes a semicolon (`;`) after the method declaration.

Concrete class → A *concrete class* is a non-abstract class. The first concrete subclass that extends an abstract class is required to implement all inherited abstract methods.

Immutable objects

📌 The *immutable objects pattern* is an object-oriented design pattern in which an object cannot be modified after it is created

Declaring an immutable class:

1. Mark the class as `final` or make all of the constructors `private`.
2. Mark all the instance variables `private` and `final`.
3. Don’t define any setter methods.
4. Don’t allow referenced mutable objects to be modified.

5. Use a constructor to set all properties of the object, making a copy if needed.

```
public Animal(List<String> favoriteFoods) {
   if (favoriteFoods == null || favoriteFoods.size() == 0)
      throw new RuntimeException("favoriteFoods is required");
   this.favoriteFoods = new ArrayList<String>(favoriteFoods);
}
```

The copy operation is called a *defensive copy* because the copy is being made in case other code does something unexpected. It's the same idea as defensive driving: prevent a problem before it exists. With this approach, our `Animal` class is once again immutable.