ChatGPT

Master OOPs Notes (Merged: Hidden Traps + Deep Dive Concepts)

⚙ 1. Inheritance

```
class Parent {
    static { System.out.println("Parent static block"); } // Static block runs once during class loading
    { System.out.println("Parent instance block"); }       // Instance block runs before constructor each time an object is created
    private int secret = 42;                                // Private members are not inherited
    Parent() { System.out.println("Parent constructor"); }
}

class Child extends Parent {
    static { System.out.println("Child static block"); }
    { System.out.println("Child instance block"); }
    Child() { System.out.println("Child constructor"); }
    // Attempting to access 'secret' from Parent would fail
}

final class FinalClass {} // Cannot be extended

public class Test {
    public static void main(String[] args) {
        new Child(); // Demonstrates inheritance flow
    }
}
```

```
class Parent {
    static { System.out.println("Parent static block"); }
    { System.out.println("Parent instance block"); }
    Parent() { System.out.println("Parent constructor"); }
}

class Child extends Parent {
    static { System.out.println("Child static block"); }
    { System.out.println("Child instance block"); }
    Child() { System.out.println("Child constructor"); }
}

public class Test {
    public static void main(String[] args) {
        new Child();
    }
}
```

- **Single Inheritance** only; **Multiple Inheritance** via interfaces.
- **Private members** are not inherited.
- **Constructor chaining**: Always starts from parent → child.
- Static blocks run once per class loading.
- Instance blocks run every time object is created.

⚠ Hidden Traps

- No `super()` call? Java inserts implicit call to parent no-arg constructor.
- Can't call `this()` and `super()` in same constructor.
- Final class = cannot be extended.

---

2. Method Overriding vs Method Hiding

- **Overriding**: For instance methods. Runtime binding.
- **Hiding**: For static methods. Compile-time binding.
- `@Override` is invalid for static methods.

```
class Parent {
    static void m() { System.out.println("parent"); }
    void show() { System.out.println("parent show"); }
}
class Child extends Parent {
    static void m() { System.out.println("child"); }
    void show() { System.out.println("child show"); }
}
```

```
Parent p = new Child();
p.m();      // parent (compile-time binding)
p.show();   // child (runtime binding)
```

---

3. Final: Class, Method, Variable

```
final class FinalClass {}

class Demo {
    final int x;
    final void show() {
        System.out.println("Final method");
    }
    Demo() {
        x = 10;
        System.out.println("Final variable initialized to: " + x);
    }
}
```

- `final class` – No inheritance allowed.
- `final method` – Cannot be overridden.
- `final variable` – Must be initialized once.

Hidden Tip

- `final` reference variable can change object state.

---

4. Abstract Class vs Interface

```
interface I {
    void show();
    default void defaultMethod() {
        System.out.println("Default method in Interface");
    }
}

abstract class AbstractClass implements I {
    public void show() {
        System.out.println("Abstract class implementation");
    }
}

class ConcreteClass extends AbstractClass {}

public class Test {
    public static void main(String[] args) {
        I obj = new ConcreteClass();
        obj.show();
        obj.defaultMethod();
    }
}
```

Feature	Abstract Class	Interface (Java 8+)
Inheritance	Single	Multiple
Methods	Abstract + Concrete	Abstract + Default + Static
Variables	Instance + Static	public static final only
Constructors	Allowed	Not allowed
Access Mod	Any	public only by default

- Interface can have static and default methods (Java 8+).

- Cannot mark interface methods as `protected`.

📚 5. Method Overloading Priority

1. Exact Match
2. Widening
3. Autoboxing
4. Varargs

```
void m(int x) {...}      // Preferred
void m(long x) {...}     // Widening
void m(Integer x) {...}  // Autoboxing
void m(int... x) {...}   // Varargs
```

Trap

- Varargs is last-resort.
- Autoboxing ignored if widening is possible.

6. Boxing and Unboxing

- Auto-conversion between primitives and wrappers.

Traps

- NPE if you unbox null.
- Integer cache: -128 to 127 → reuse from cache.

```
Integer a = 100, b = 100;
System.out.println(a == b); // true (cached)
Integer c = 200, d = 200;
System.out.println(c == d); // false
```

7. Typecasting: Upcasting vs Downcasting

- **Upcasting**: Child → Parent (Safe)
- **Downcasting**: Parent → Child (Explicit & Risky)

```
A a = new B();         // Upcast
B b = (B) a;           // Downcast OK
A x = new A();
B y = (B) x;           // Runtime Error
```

8. instanceof Keyword

- Checks actual object type.
- Returns false if object is `null`.

```
Object o = null;
System.out.println(o instanceof String); // false
```

---

9. Shadowing (Variables) vs Hiding (Static Methods)

```
class A { static void m(){} int x = 10; }
class B extends A { static void m(){} int x = 20; }
A obj = new B();
System.out.println(obj.x);  // 10
obj.m();                    // A.m()
```

⚖ 10. Object Class Methods

```
class Emp {
    int id;
    Emp(int id) { this.id = id; }
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Emp)) return false;
        Emp e = (Emp) o;
        return this.id == e.id;
    }
    public int hashCode() {
        return id;
    }
}

public class Test {
    public static void main(String[] args) {
        Emp e1 = new Emp(101);
        Emp e2 = new Emp(101);
        System.out.println(e1.equals(e2));  // true
        System.out.println(e1.hashCode() == e2.hashCode());  // true
    }
}
```

- `.equals()` compares content (override it).
- `.hashCode()` must be consistent with `.equals()`.
- Default `.equals()` = reference comparison.

Tip

- Always override both for proper behavior in HashMap/HashSet.

---

11. Polymorphism Runtime Traps

- Static, final, private methods = not polymorphic.
- Fields are resolved by reference type.

```
class A {
    int x = 10;
    static void show() {}
    void print() {}
}
class B extends A {
    int x = 20;
    static void show() {}
    void print() {}
}
A obj = new B();
System.out.println(obj.x);   // 10
obj.show();                  // A.show()
obj.print();                 // B.print()
```

⛓ 12. Constructor & Block Execution Order

Object Creation Flow:

1. Static block (super → sub, only once)
2. Instance blocks & fields (super → sub)
3. Constructors (super → sub)

```
class A {
    static { System.out.println("A static"); }
    { System.out.println("A instance"); }
    A() { System.out.println("A constructor"); }
}
```

INTERACTIONS: Constructor + Abstract + Interface + Normal

```
interface I {
    void method();
}
```

```java
abstract class A implements I {
    A() {
        System.out.println("A constructor");
        method(); // Risky: calls subclass method before init
    }
}

class B extends A {
    int x = 10;
    B() { System.out.println("B constructor"); }
    public void method() {
        System.out.println("B method, x = " + x); // x not initialized yet
    }
}

public class Test {
    public static void main(String[] args) {
        new B();
    }
}
```

Scenario	Behavior Summary
Abstract class with constructor	Runs during instantiation via subclass
Interface + Abstract class	Abstract class implements interface, must define/ forward abstract methods
Constructor calling abstract method	Risky: calls child method before child fields are initialized
Constructor calling interface method	If default/static — works; otherwise abstract — requires implementation
Normal class extending abstract class	Must implement all abstract methods or be abstract itself
Interface with default method	Can be overridden, else inherited
Abstract class vs Interface constructor	Abstract classes have constructors; interfaces don't
Multiple interfaces with same method	Class must override method to resolve ambiguity
Constructor chaining across abstract + normal class	Follows usual constructor chaining rules (super → sub)

Scenario	Behavior Summary
Abstract class with constructor	Runs during instantiation via subclass
Interface + Abstract class	Abstract class implements interface, must define/ forward abstract methods
Constructor calling abstract method	Risky: calls child method before child fields are initialized

Scenario	Behavior Summary
Constructor calling interface method	If default/static — works; otherwise abstract — requires implementation
Normal class extending abstract class	Must implement all abstract methods or be abstract itself
Interface with default method	Can be overridden, else inherited
Abstract class vs Interface constructor	Abstract classes have constructors; interfaces don't
Multiple interfaces with same method	Class must override method to resolve ambiguity
Constructor chaining across abstract + normal class	Follows usual constructor chaining rules (super → sub)

Let me know if you'd like this exported as a beautifully formatted PDF with diagrams and traps highlighted.