The SOLID principles are like golden rules for building strong foundations in object-oriented programming. Let’s look at each through simple analogies and Java examples.

**🧱 1. Single Responsibility Principle (SRP)**

**"A class should do one thing and do it well."**

🧩 **Analogy**:  
Imagine a coffee machine that brews coffee, repairs itself, and even handles payments. If any of those parts fail, the whole machine becomes complicated to fix. Wouldn’t it be better to separate those roles?

🧪 **Bad Design**

java

CopyEdit

public class Report {

public void generate() {}

public void print() {}

public void email() {}

}

🛠 **Better Design**

java

CopyEdit

public class Report {

public void generate() {}

}

public class ReportPrinter {

public void print(Report report) {}

}

public class ReportMailer {

public void email(Report report) {}

}

Each class now has **one job**—easy to manage and test.

**🪄 2. Open/Closed Principle (OCP)**

**"Code should be extendable, not rewriteable."**

🧩 **Analogy**:  
Think of a power strip where you can plug in new devices without opening it up. Similarly, your code should support adding features without modifying existing logic.

🧪 **Bad Design**

java

CopyEdit

public class Payment {

public void pay(String method) {

if (method.equals("credit")) { /\* logic \*/ }

else if (method.equals("paypal")) { /\* logic \*/ }

}

}

🛠 **Better Design**

java

CopyEdit

interface PaymentMethod {

void pay();

}

class CreditCard implements PaymentMethod {

public void pay() {

System.out.println("Paid with credit card");

}

}

class PayPal implements PaymentMethod {

public void pay() {

System.out.println("Paid with PayPal");

}

}

New payment types? Just **extend**, don’t **edit** the original code.

**🦅 3. Liskov Substitution Principle (LSP)**

**"Subtypes must behave like their parent."**

🧩 **Analogy**:  
If you rent a car, you expect it to drive. Imagine getting a bulldozer instead that can't go on the road. That’s what LSP warns us about.

🧪 **Bad Design**

java

CopyEdit

class Vehicle {

public void startEngine() {}

}

class Bicycle extends Vehicle {

@Override

public void startEngine() {

throw new UnsupportedOperationException("No engine!");

}

}

🛠 **Better Design**

java

CopyEdit

class Vehicle {}

class MotorVehicle extends Vehicle {

public void startEngine() {}

}

class Car extends MotorVehicle {

public void startEngine() {

System.out.println("Car engine started");

}

}

class Bicycle extends Vehicle {

// No engine method here

}

Now, every subclass behaves **consistently** with expectations.

**🪓 4. Interface Segregation Principle (ISP)**

**"Don’t force classes to implement what they don’t use."**

🧩 **Analogy**:  
You wouldn't want a smartphone manual to include steps for setting up a microwave. Similarly, an interface should only expose relevant actions.

🧪 **Bad Design**

java

CopyEdit

interface Machine {

void start();

void stop();

void refuel();

}

class ElectricFan implements Machine {

public void start() {}

public void stop() {}

public void refuel() {

throw new UnsupportedOperationException("No refueling needed!");

}

}

🛠 **Better Design**

java

CopyEdit

interface Startable {

void start();

void stop();

}

interface Refuelable {

void refuel();

}

class ElectricFan implements Startable {

public void start() {}

public void stop() {}

}

Interfaces are now **purpose-specific** and flexible.

**🧲 5. Dependency Inversion Principle (DIP)**

**"Depend on abstractions, not concrete classes."**

🧩 **Analogy**:  
Using an electric plug lets you connect to any appliance—fan, TV, toaster. You don’t hardwire one specific device.

🧪 **Bad Design**

java

CopyEdit

class SMS {

public void send(String msg) {

System.out.println("SMS sent: " + msg);

}

}

class NotificationService {

private SMS sms = new SMS();

public void notify(String msg) {

sms.send(msg);

}

}

🛠 **Better Design**

java

CopyEdit

interface MessageService {

void send(String msg);

}

class SMS implements MessageService {

public void send(String msg) {

System.out.println("SMS sent: " + msg);

}

}

class Email implements MessageService {

public void send(String msg) {

System.out.println("Email sent: " + msg);

}

}

class NotificationService {

private MessageService service;

public NotificationService(MessageService service) {

this.service = service;

}

public void notify(String msg) {

service.send(msg);

}

}

Now the service is **flexible** and depends on an interface.

**🧠 SOLID Cheat Sheet**

| **Principle** | **Motto** | **Key Idea** |
| --- | --- | --- |
| SRP | One class, one job | Don’t multitask in classes |
| OCP | Open to grow, closed to break | Add features without rewriting existing |
| LSP | Respect the parent’s contract | Subtypes must behave like base types |
| ISP | Less is more | Don’t overload interfaces |
| DIP | Talk to abstractions | Break tight coupling |