Lecture 7

Dr. Umair Rehman

Agenda

- Dependency Injection
  - Where to inject →
    - Constructor (required at creation)
    - Setter (after construction, optional)
    - Method (per-call, one-off)
    - Interface (formal injection contract)
  - How to inject →
    - Lazy (provider/factory instead of direct object)
    - Hybrid (mix of required + optional)

Dependency Injection

- Dependency Injection is a technique where a class does not create its own dependencies, but instead they are supplied (injected) from the outside.
  - Dependency = any object a class relies on to do its work.
  - Injection = providing that object from outside, instead of hardcoding it inside.

Dependency Injection

Pattern	How It Works	Best Use Case	Pros	Cons
Constructor Injection	Dependencies are passed in via the class constructor.	Required dependencies (object cannot function without them).	Explicit, safe, object always fully initialized.	Constructor bloat if too many dependencies.
Setter Injection	Dependencies are set through mutator methods after construction.	Optional dependencies or ones that may change.	Flexible, easy to reconfigure.	Risk of null/forgotten dependencies.
Interface Injection	Class implements an interface with an inject() method. Injector calls it to supply dependencies.	When you want to enforce a formal injection contract.	Strong protocol, clear injection point.	Verbose, uncommon in C++.
Method Injection	Dependencies passed directly into methods that use them.	One-off or short-lived dependencies.	Simple, avoids storing dependencies.	Repetition if dependency is always needed.
Lazy Injection	Inject a **provider/factory** instead of a ready object. Dependency created only when needed.	Expensive/rarely used dependencies (e.g., DB connection).	Saves resources, avoids premature creation.	More complex (factory handling, lifetime).
Hybrid Injection	Constructor for required deps, setters for optional ones.	Balanced real-world cases.	Safety + flexibility combined.	Slightly more complex API.

Constructor Injection

```cpp
#include <iostream>
#include <string>

// ------------------------------
// 1) ABSTRACTIONS (interfaces)
// ------------------------------

// Payment gateway interface: high-level code will depend only on this.
struct IPaymentGateway {
    virtual bool charge(double amount) = 0;
    virtual ~IPaymentGateway() = default; // always virtual destructor for interfaces
};

// Logger interface: allows logging without tying to a specific logging system.
struct ILogger {
    virtual void log(const std::string& message) = 0;
    virtual ~ILogger() = default;
};

```

Constructor Injection

```
20. // ------------------------------
21. // 2) CONCRETE IMPLEMENTATIONS
22. // ------------------------------
23. 
24. // Example payment gateway: Stripe
25. class StripeGateway : public IPaymentGateway {
26. public:
27.     bool charge(double amount) override {
28.         std::cout << "[Stripe] Charging $" << amount << '\n';
29.         return true; // pretend the charge succeeded
30.     }
31. };
32. 
33. // Another payment gateway: PayPal
34. class PayPalGateway : public IPaymentGateway {
35. public:
36.     bool charge(double amount) override {
37.         std::cout << "[PayPal] Charging $" << amount << '\n';
38.         return true;
39.     }
40. };
```

Constructor Injection

```
42. // Simple console logger
43. class ConsoleLogger : public ILogger {
44. public:
45.     void log(const std::string& message) override {
46.         std::cout << "[LOG] " << message << '\n';
47.     }
48. };
```

Constructor Injection

```
50. // ----------------------------
51. // 3) HIGH-LEVEL SERVICE
52. // ----------------------------
53. // This class *depends* on a payment gateway and a logger.
54. // Instead of creating them inside, they are injected via the constructor.
55. class PaymentService {
56.     IPaymentGateway* gateway_; // required dependency (not owned here)
57.     ILogger* logger_;          // required dependency (not owned here)
58. 
59. public:
60.     // Constructor Injection: both dependencies must be passed in at creation time.
61.     PaymentService(IPaymentGateway* gateway, ILogger* logger)
62.         : gateway_(gateway), logger_(logger)
63.     {
64.         if (!gateway_ || !logger_) {
65.             throw std::invalid_argument("Dependencies cannot be null!");
66.         }
67.     }
68. 
69.     // Use the injected dependencies to perform work.
70.     bool pay(double amount) {
71.         logger_->log("Initiating payment of $" + std::to_string(amount));
72.         bool ok = gateway_->charge(amount);
73.         logger_->log(ok ? "Payment succeeded" : "Payment failed");
74.         return ok;
75.     }
76. };
77.
```

Constructor Injection

```
77.
78. // ------------------------------
79. // 4) DEMO USAGE
80. // ------------------------------
81. int main() {
82.     // Create concrete objects (they live in main and are passed by pointer).
83.     StripeGateway stripe;
84.     PayPalGateway paypal;
85.     ConsoleLogger logger;
86.
87.     // Inject Stripe + Logger into PaymentService
88.     PaymentService service1(&stripe, &logger);
89.     service1.pay(50.0);
90.
91.     // Inject PayPal + Logger into PaymentService
92.     PaymentService service2(&paypal, &logger);
93.     service2.pay(75.0);
94.
95.     return 0;
96. }
97.
```

Constructor Injection

- Dependencies are **required** at construction time.
- `PaymentService` **does not create** `StripeGateway` **or** `PayPalGateway` **inside itself.**
- Swapping dependencies (`Stripe` ↔ `PayPal`, **or even a test mock) requires no changes to** `PaymentService`.
- Ownership is managed by the caller (`main` **here). The service just uses the injected pointers.**

Setter Injection

```cpp
#include <iostream>
#include <string>

// ------------------------------
// 1) ABSTRACTIONS (interfaces)
// ------------------------------

// Logger interface — defines a contract for logging.
struct ILogger {
    virtual void log(const std::string& message) = 0;
    virtual ~ILogger() = default;
};

// ------------------------------
// 2) CONCRETE IMPLEMENTATION
// ------------------------------

// Console logger (prints to terminal).
class ConsoleLogger : public ILogger {
public:
    void log(const std::string& message) override {
        std::cout << "[LOG] " << message << '\n';
    }
};

```

Setter Injection

```
25.
26. // ------------------------------
27. // 3) HIGH-LEVEL SERVICE
28. // ------------------------------
29. // This service can work with or without a logger.
30. // Instead of constructor injection, the dependency is set later.
31. class PaymentService {
32.     ILogger* logger_; // optional dependency (not owned)
33.
34. public:
35.     // Default constructor: service can be created without a logger.
36.     PaymentService() : logger_(nullptr) {}
37.
38.     // Setter Injection: dependency is provided after construction.
39.     void setLogger(ILogger* logger) {
40.         logger_ = logger;
41.     }
42.
43.     void pay(double amount) {
44.         // If a logger is available, use it. Otherwise, just work silently.
45.         if (logger_) {
46.             logger_->log("Processing payment of $" + std::to_string(amount));
47.         }
48.         std::cout << "Charged $" << amount << '\n';
49.     }
50. };
```

Setter Injection

```
51.
52. // ------------------------------
53. // 4) DEMO USAGE
54. // ------------------------------
55. int main() {
56.     PaymentService service; // created without dependencies
57.
58.     // Service still works, but no logging
59.     service.pay(20.0);
60.
61.     // Now inject a dependency via setter
62.     ConsoleLogger logger;
63.     service.setLogger(&logger);
64.
65.     // Now it uses the logger too
66.     service.pay(30.0);
67.
68.     return 0;
69. }
70.
```

Setter Injection

- Good for **optional dependencies**.
- Object can be created first, and dependencies supplied later.
- Drawback: risk of calling methods before the dependency is set (null pointer).
- Contrasts with Constructor Injection, which enforces required dependencies at compile-time.

Interface Injection

```cpp
#include <iostream>
#include <string>

// ------------------------------
// 1) DEPENDENCY CLASS (Configuration)
// ------------------------------
// Let's say our service needs some configuration data.
struct Config {
    int retries;   // how many times to retry a payment
};

// ------------------------------
// 2) CONFIGURABLE INTERFACE
// ------------------------------
// Any class that needs configuration implements this interface.
// The injector will call `setConfig()` on it.
struct IConfigurable {
    virtual void setConfig(const Config& c) = 0;
    virtual ~IConfigurable() = default;
};

```

Interface Injection

```
21.
22. // ------------------------------
23. // 3) HIGH-LEVEL SERVICE (implements IConfigurable)
24. // ------------------------------
25. // PaymentService declares that it can accept configuration
26. // through the IConfigurable interface.
27. class PaymentService : public IConfigurable {
28.     Config cfg_; // holds configuration
29.
30. public:
31.     // Implementation of the injection method from IConfigurable
32.     void setConfig(const Config& c) override {
33.         cfg_ = c;
34.     }
35.
36.     void pay(double amount) {
37.         std::cout << "Processing $" << amount
38.                   << " with retries=" << cfg_.retries << '\n';
39.     }
40. };
```

Interface Injection

```
41.
42. // ------------------------------
43. // 4) INJECTOR CLASS
44. // ------------------------------
45. // This class knows how to inject configuration into any IConfigurable object.
46. class ConfigInjector {
47. public:
48.     static void inject(IConfigurable& obj) {
49.         Config c;
50.         c.retries = 5; // provide dependency (here it's just hardcoded)
51.         obj.setConfig(c);
52.     }
53. };
54.
```

Interface Injection

```
55. // ------------------------------
56. // 5) DEMO USAGE
57. // ------------------------------
58. int main() {
59.     PaymentService service;
60.
61.     // The injector provides the configuration via the interface method.
62.     ConfigInjector::inject(service);
63.
64.     // Now the service can use the injected config.
65.     service.pay(100.0);
66.
67.     return 0;
68. }
69.
```

Interface Injection

- **Interface Injection** makes the *client class* (`PaymentService`) explicitly say:
  "I accept dependencies through this interface."
- The **injector** (`ConfigInjector`) knows about that interface and supplies the dependency.
- It's less common in C++ (used more in Java frameworks), but it's a neat way to show how dependency injection can be formalized.
- Compared to Constructor / Setter Injection, this pattern enforces the *protocol* via an interface.

Method Injection

```cpp
#include <iostream>
#include <string>

// ------------------------------
// 1) ABSTRACTION
// ------------------------------
struct ILogger {
    virtual void log(const std::string& message) = 0;
    virtual ~ILogger() = default;
};

// ------------------------------
// 2) CONCRETE IMPLEMENTATION
// ------------------------------
class ConsoleLogger : public ILogger {
public:
    void log(const std::string& message) override {
        std::cout << "[LOG] " << message << '\n';
    }
};

```

Method Injection

```
22. // ------------------------------
23. // 3) HIGH-LEVEL SERVICE
24. // ------------------------------
25. // No member variables for dependencies here.
26. // Instead, dependencies are passed as parameters to methods.
27. class PaymentService {
28. public:
29.     void pay(double amount, ILogger* logger) {
30.         if (logger) {
31.             logger->log("Processing payment of $" + std::to_string(amount));
32.         }
33.         std::cout << "Charged $" << amount << '\n';
34.     }
35. };
36.
```

Method Injection

```
36.
37. // ------------------------------
38. // 4) DEMO USAGE
39. // ------------------------------
40. int main() {
41.     ConsoleLogger logger;
42.     PaymentService service;
43.
44.     // Inject logger *per method call* (Method Injection)
45.     service.pay(25.0, &logger);
46.
47.     // Can also call without a logger (null pointer)
48.     service.pay(40.0, nullptr);
49.
50.     return 0;
51. }
52.
```

Method Injection

- **Method Injection** is good for *one-off or short-lived* dependencies.
- No need to store dependencies in the object → keeps class state simpler.
- **Trade-off:** If the method needs the dependency *every time*, passing it repeatedly can clutter code.
- Works well when the dependency is needed **only in one method** (not across the whole class).

Lazy Injection

```
1. #include <iostream>
2. #include <string>
3. #include <functional> // for std::function
4. 
5. // ------------------------------
6. // 1) ABSTRACTION
7. // ------------------------------
8. struct IPaymentGateway {
9.     virtual bool charge(double amount) = 0;
10.     virtual ~IPaymentGateway() = default;
11. };
12. 
13. // ------------------------------
14. // 2) CONCRETE IMPLEMENTATION
15. // ------------------------------
16. class StripeGateway : public IPaymentGateway {
17. public:
18.     bool charge(double amount) override {
19.         std::cout << "[Stripe] Charging $" << amount << '\n';
20.         return true;
21.     }
22. };
23. 
```

Lazy Injection

```
23.
24. // ------------------------------
25. // 3) HIGH-LEVEL SERVICE
26. // ------------------------------
27. // Instead of holding a gateway directly, this class holds a *provider*
28. // that can create gateways on demand.
29. class PaymentService {
30.     std::function<IPaymentGateway*()> provider_; // factory function
31.     IPaymentGateway* gateway_;                   // cached instance (lazy init)
32.
33. public:
34.     // Constructor takes a provider instead of a ready-made dependency
35.     explicit PaymentService(std::function<IPaymentGateway*()> provider)
36.         : provider_(provider), gateway_(nullptr) {}
37.
38.     bool pay(double amount) {
39.         // Lazy initialization: create the dependency only when first needed
40.         if (!gateway_) {
41.             gateway_ = provider_(); // use the factory
42.         }
43.
44.         return gateway_->charge(amount);
45.     }
46. };
```

Lazy Injection

```
47.
48. // ------------------------------
49. // 4) DEMO USAGE
50. // ------------------------------
51. int main() {
52.     // Provider: a lambda that knows how to build a StripeGateway
53.     auto stripeProvider = []() {
54.         std::cout << "(Factory creates StripeGateway)\n";
55.         return new StripeGateway(); // for teaching simplicity (manual delete omitted)
56.     };
57.
58.     // Inject provider instead of the object
59.     PaymentService service(stripeProvider);
60.
61.     // First call → provider creates dependency lazily
62.     service.pay(100.0);
63.
64.     // Second call → uses cached gateway (no new creation)
65.     service.pay(200.0);
66.
67.     return 0;
68. }
69.
```

Lazy Injection

- **Lazy Injection** means you delay actual creation until it's needed.
- Useful for:
  - Heavy resources (database, API connections).
  - Conditional or rarely used dependencies.
- Trade-off: adds complexity — you need to manage lifetime (`new`/`delete` or smart pointers).

Hybrid Injection

```cpp
#include <iostream>
#include <string>

// ------------------------------
// 1) ABSTRACTIONS (interfaces)
// ------------------------------
struct IPaymentGateway {
    virtual bool charge(double amount) = 0;
    virtual ~IPaymentGateway() = default;
};

struct ILogger {
    virtual void log(const std::string& message) = 0;
    virtual ~ILogger() = default;
};

// ------------------------------
```

Hybrid Injection

```
17. // ------------------------------
18. // 2) CONCRETE IMPLEMENTATIONS
19. // ------------------------------
20. class StripeGateway : public IPaymentGateway {
21. public:
22.     bool charge(double amount) override {
23.         std::cout << "[Stripe] Charging $" << amount << '\n';
24.         return true;
25.     }
26. };
27.
28. class ConsoleLogger : public ILogger {
29. public:
30.     void log(const std::string& message) override {
31.         std::cout << "[LOG] " << message << '\n';
32.     }
33. };
34.
```

Hybrid Injection

```
35. // ------------------------------
36. // 3) HIGH-LEVEL SERVICE
37. // ------------------------------
38. // - Requires a payment gateway (injected via constructor).
39. // - Optionally accepts a logger (injected via setter).
40. class PaymentService {
41.     IPaymentGateway* gateway_; // required dependency
42.     ILogger* logger_;          // optional dependency
43. 
44. public:
45.     // Constructor Injection: gateway is required
46.     explicit PaymentService(IPaymentGateway* gateway)
47.         : gateway_(gateway), logger_(nullptr) {
48.         if (!gateway_) {
49.             throw std::invalid_argument("Gateway cannot be null!");
50.         }
51.     }
52. 
```

Hybrid Injection

```
53.     // Setter Injection: logger is optional
54.     void setLogger(ILogger* logger) {
55.         logger_ = logger;
56.     }
57.
58.     bool pay(double amount) {
59.         if (logger_) {
60.             logger_->log("Starting payment of $" + std::to_string(amount));
61.         }
62.
63.         bool ok = gateway_->charge(amount);
64.
65.         if (logger_) {
66.             logger_->log(ok ? "Payment succeeded" : "Payment failed");
67.         }
68.
69.         return ok;
70.     }
71. };
72.
```

Hybrid Injection

```
73. // ------------------------------
74. // 4) DEMO USAGE
75. // ------------------------------
76. int main() {
77.     StripeGateway stripe;
78.     ConsoleLogger logger;
79. 
80.     // Required dependency injected via constructor
81.     PaymentService service(&stripe);
82. 
83.     // Service works even without a logger
84.     service.pay(25.0);
85. 
86.     // Optional dependency injected later
87.     service.setLogger(&logger);
88.     service.pay(50.0);
89. 
90.     return 0;
91. }
92. 
```

Hybrid Injection

- **Hybrid Injection** is practical in real-world apps:
  - Required dependencies → constructor injection.
  - Optional dependencies → setter injection.
- This balances **safety** (can't forget required deps) with **flexibility** (can add optional features later).
- Very common in enterprise frameworks (Java Spring, .NET DI, etc.).

Conclusion

- Dependency Injection
  - Where to inject →
    - Constructor (required at creation)
    - Setter (after construction, optional)
    - Method (per-call, one-off)
    - Interface (formal injection contract)
  - How to inject →
    - Lazy (provider/factory instead of direct object)
    - Hybrid (mix of required + optional)