Unit – IV Secure coding practices in C/C++ and Java: Potential Software Risks in C/C++, Defensive coding,Preventative Planning, Clean Code, Iterative Design, Assertions, PrePost Conditions, Low level design inspections, Unit Tests.

Java- Managing Denial of Service, Securing Information, Data Integrity, Accessibility and Extensibility, Securing Objects, Serialization Security

Potential Software Risks in C/C++

Potential for:

- **Bad data**

Programs written in C/C++ often interact with external sources such as user inputs, databases, or web services. If these sources provide unexpected or malicious data, it can lead to vulnerabilities.

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    string fileName;
    cout << "Enter the file name: ";
    cin >> fileName;

    ifstream file(fileName.c_str());
    if (!file.is_open()) {
        cerr << "Error opening file!" << endl;
        return 1;
    }

    // Process file contents (vulnerable to malicious payload)
    // ...

    file.close();
    return 0;
}
```

In this example, if a user provides a file with malicious content, it may lead to security vulnerabilities during the file processing.

- **Poor error handling**

C/C++ allows manual memory management and lacks built-in features like exceptions in languages such as Java or Python. Improper error handling can lead to memory leaks, crashes, or undefined behavior.

```cpp
#include <iostream>
using namespace std;

int main() {
    int* arr = new int[5];

    // Failure to check if memory allocation is successful
    // May lead to undefined behavior if allocation fails
    // and arr is a null pointer

    // ...

    delete[] arr;
    return 0;
}
```

In this example, there is no check for the success of dynamic memory allocation, which can lead to runtime errors.

- **Usability issues**

C/C++ programming may sometimes prioritize performance over user-friendly features. Poorly designed user interfaces or complex command-line interactions can lead to usability problems.

```cpp
#include <iostream>
using namespace std;

int main() {
    // Complex command-line interface
    cout << "Enter 1 for option A, 2 for option B, etc." << endl;

    int choice;
    cin >> choice;

    // Process the choice
    // ...

    return 0;
}
```

This simple program uses a less user-friendly command-line interface, which may lead to usability issues.

- **Poor runtime performance**

While C/C++ offer high performance, improper coding practices or inefficient algorithms can lead to suboptimal performance.

```cpp
#include <iostream>
using namespace std;

int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; ++i) {
        if (arr[i] == target) {
            return i;
        }
    }
    return -1;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int target = 3;

    // Using a linear search instead of a more efficient algorithm
    int result = linearSearch(arr, 5, target);

    // ...

    return 0;
}
```

In this example, a linear search is used on a small array, but for larger datasets, a more efficient algorithm should be employed.

- **Security vulnerabilities**

C/C++ provides low-level memory manipulation, which can lead to vulnerabilities like buffer overflows if not handled carefully.

```cpp
#include <iostream>
using namespace std;

int main() {
    char buffer[5];
    cout << "Enter your name: ";
    cin >> buffer;

    // Vulnerable to buffer overflow if the user enters more than 5

    cout << "Hello, " << buffer << "!" << endl;

    return 0;
}
```

In this example, if the user enters a name longer than the buffer size, it can lead to a buffer overflow.

Why do Risks Exist in C/C++ ?

1.Programs often rely on external data
Users,databases,web services can all provide “bad” data
2.Often a failure to account for and handle errors properly
3.Failure to implement proper software design
Time and budgets constraints

Defensive coding

1.Also referred to as “secure” coding
2.Attempts to ensure that software still functions under adverse or unforeseen circumstances
3.Assumes mistakes will be made
i.Tries to prevent them and plan for them
ii.Makes sure that errors are visible so they can be detected and dealt with

Program:

```
#include <iostream>

#include <vector>

// Function to safely access an element in a vector

// Returns true if successful, false if index is out of bounds

bool safeAccess(const std::vector<int>& vec, size_t index, int& result) {

    if (index < vec.size()) {

        result = vec[index];

        return true;

    } else {

        // Index is out of bounds, return false

        return false;

    }

}

int main() {

    std::vector<int> myVector = {1, 2, 3, 4, 5};
```

```
    // Attempt to access an element safely
    size_t indexToAccess;
    std::cout << "Enter the index to access: ";
    std::cin >> indexToAccess;

    int value;
    if (safeAccess(myVector, indexToAccess, value)) {
        std::cout << "Value at index " << indexToAccess << ": " << value << std::endl;
    } else {
        std::cout << "Index out of bounds!" << std::endl;
    }
    return 0;
}
```

In this program:

Secure Coding: The function safeAccess is designed to safely access an element in a vector by checking if the index is within bounds before attempting to access it.

Functioning Under Adverse Circumstances: The program anticipates that the user might enter an index that is out of bounds for the vector and handles this situation gracefully without crashing or causing undefined behavior.

Visible Errors: If the user enters an out-of-bounds index, the program prints a clear error message indicating that the index is out of bounds, making the error visible for detection and handling.

This simple example showcases the principles of defensive programming in C++, emphasizing secure coding practices and error prevention.

- **Strategies for Defensive Programming**
  1.Be wary of input
  2.Plan for success
  3.Stop errors before they happen
  4.Develop clean code
  5.Test early and test often

Preventative Planning

Proper design can prevent many potential problems

1.Iterative design
2.Use pseudo code before actually writing code
3.Avoid using code prone to vulnerabilites
4.Utilize test cases where possible
5.Perform low level design inspections
6.Use proper error handling techniques

Program:

```
#include <iostream>

#include <stdexcept>

// Function to perform division with proper error handling

double safeDivision(double numerator, double denominator) {

    if (denominator == 0) {

        // Throw an exception for division by zero

        throw std::invalid_argument("Error: Division by zero");

    }

    return numerator / denominator;

}

int main() {

    // Prompt the user for input

    double numerator, denominator;

    std::cout << "Enter the numerator: ";

    std::cin >> numerator;

    std::cout << "Enter the denominator: ";

    std::cin >> denominator;

    try {

        // Attempt to perform division using safeDivision function

        double result = safeDivision(numerator, denominator);
```

```
        std::cout << "Result: " << result << std::endl;
    } catch (const std::exception& e) {
        // Catch and handle exceptions (e.g., division by zero)
        std::cerr << "Error: " << e.what() << std::endl;
    }
    return 0;
}
```

Explanation:

Iterative Design:

The program design is iterative, allowing for modifications and improvements based on evolving requirements.

Use Pseudo Code:

Before writing actual code, developers often use pseudo code to plan the logic and structure of the program. While pseudo code is not implemented in this example, it's a good practice to sketch out the algorithm before diving into coding.

Avoid Code Prone to Vulnerabilities:

The safeDivision function checks for division by zero, a common vulnerability. By using proper error handling, we avoid potential issues associated with division by zero.

Utilize Test Cases:

The program can be tested with various inputs, including edge cases like dividing by zero, to ensure that it behaves as expected and handles errors gracefully.

Perform Low-Level Design Inspections:

While this example is relatively simple, in a larger project, low-level design inspections involve reviewing the code for adherence to coding standards, identifying potential vulnerabilities, and ensuring proper error handling.

Use Proper Error Handling Techniques:

The safeDivision function uses exception handling to address errors, providing a clear indication of what went wrong and allowing for proper handling in the main function.

This example demonstrates how preventative planning principles can be incorporated into a C++ program to enhance design, reduce vulnerabilities, and improve error handling.

Clean Code

Clean coding in the context of secure coding techniques involves writing code that not only produces the desired functionality but also minimizes vulnerabilities, reduces the likelihood of security breaches, and facilitates the identification and resolution of security issues.

Program:

```
#include<iostream>

using namespace std;

// Function prototype

float triangle_area(float b, float h);

int main() {

   float base, height, area;

   // Get user input

   cout << "Please enter the base of the triangle: ";

   cin >> base;

   cout << "Please enter the height of the triangle: ";

   cin >> height;

   // Calculate the area using the function

   area = triangle_area(base, height);

   // Display the result

   cout << "The triangle's area is: " << area;

   return 0;

}

// Function definition to calculate the area of a triangle

float triangle_area(float b, float h) {

   float a;

   a = 0.5 * (b * h);

   return a;

}
```

Output:

Please enter the base of the triangle: 8.5

Please enter the height of the triangle: 12.3

The triangle's area is: 52.275

Iterative Design

- A cyclic process
- Involves prototyping
- Several models

Program:

```
#include <iostream>
using namespace std;
// Function prototypes
float add(float a, float b);
float subtract(float a, float b);
float multiply(float a, float b);
float divide(float a, float b);
int main() {
   float num1, num2;
   char operation;
   do {
      // Get user input
      cout << "Enter two numbers: ";
      cin >> num1 >> num2;
      cout << "Enter operation (+, -, *, /): ";
      cin >> operation;
      // Perform operation based on user input
      switch (operation) {
```

```
    case '+':
        cout << "Result: " << add(num1, num2) << endl;
        break;
    case '-':
        cout << "Result: " << subtract(num1, num2) << endl;
        break;
    case '*':
        cout << "Result: " << multiply(num1, num2) << endl;
        break;
    case '/':
        // Check for division by zero
        if (num2 != 0) {
            cout << "Result: " << divide(num1, num2) << endl;
        } else {
            cout << "Error: Division by zero!" << endl;
        }
        break;
    default:
        cout << "Invalid operation. Try again." << endl;
}
// Ask the user if they want to perform another calculation
cout << "Do you want to perform another calculation? (y/n): ";
char choice;
cin >> choice;
if (choice != 'y' && choice != 'Y') {
    cout << "Exiting the calculator." << endl;
    break;
```

```
        }

    } while (true);

    return 0;
}
// Function definitions
float add(float a, float b) {
    return a + b;
}
float subtract(float a, float b) {
    return a - b;
}
float multiply(float a, float b) {
    return a * b;
}
float divide(float a, float b) {
    return a / b;
}
```

This simple calculator program demonstrates an iterative design process.

The user is prompted to enter two numbers and choose an operation (+, -, *, /).

The program performs the operation and displays the result.

After each calculation, the user is given the option to perform another calculation or exit the calculator.

This process repeats in a loop, allowing for multiple iterations of user interactions.

In an iterative design process, each iteration involves presenting the design (in this case, the calculator functionality) to users, noting any problems, correcting issues, and repeating the cycle until all issues are addressed. This program provides a basic example, and in a real-world scenario,

additional features, user feedback mechanisms, and error handling would be incorporated in each iteration.

Typical Process

1.Complete an intial design
2.Present the design to several test users
3.Note any problems
4.Correct issues
5.Repeat until all issues are addressed

Assertions

Assertions in secure coding techniques (SCT) play a crucial role in validating assumptions about the state of a program. They are used to check conditions that should always be true at specific points in the code. When an assertion fails, it indicates a critical problem in the program, and typically, the program is terminated immediately.

/* assert example */

```
#include <stdio.h>

#include <assert.h>

#include <iostream>

#include <fstream>

using namespace std;

// Function prototypes

void print_number(int* myInt);

void read_file();

int main() {

   int a = 10;

   int* b = &a;

   int* c = NULL;


   print_number(b);

   print_number(c);  // This will trigger an assert failure

   read_file();

   return 0;
```

```
}
void print_number(int* myInt) {
    // Use assert to make sure the value is not null
    assert(myInt != NULL);
    printf("%d\n", *myInt);
}


void read_file() {
    long start, end;
    ifstream myfile("test.txt", ios::in | ios::binary);
    // Use assertion to ensure the file is open successfully
    assert(myfile.is_open());
    start = myfile.tellg();
    myfile.seekg(0, ios::end);
    end = myfile.tellg();
    myfile.close();
    cout << "Size of test.txt is " << (end - start) << " bytes.\n";
    return;
}
```

Output:

```
10
terminate called after throwing an instance of 'int'
Aborted (core dumped)
```

PrePost Conditions

Preconditions:

Definition:

Preconditions are conditions that must be true before a function or component of code is executed.

Purpose in SCT:

Input Validation:

Ensure that the inputs to a function or module are valid and meet the expected criteria.

Security Checks:

Verify that security-related conditions are satisfied before executing critical operations. For example, ensuring that access control conditions are met before granting access to sensitive data.

Avoiding Undefined Behavior:

Prevent undefined behavior by checking that assumptions about the state of the program are true before proceeding with an operation.

Assertion Checks:

Use assertions to explicitly state and check preconditions during development, providing a self-checking mechanism.

Failure Avoidance:

Detect potential failures or security vulnerabilities early in the execution of a function, reducing the likelihood of unexpected behaviors.

Postconditions:

Definition:

Postconditions are conditions that must be true after a function or component of code has executed.

Purpose in SCT:

Result Verification:

Verify that the result of an operation or computation meets the expected conditions or requirements.

Invariants:

Maintain program invariants, ensuring that certain properties hold true after the execution of a function.

State Restoration:

Ensure that the system or application is left in a consistent state after the execution of a function, particularly important in error-handling scenarios.

Security Assertions:

Make assertions about the security properties of the system after executing security-critical functions.

Avoidance of Side Effects:

Ensure that functions do not have unintended side effects that could compromise security or system integrity.

Example:

Consider a function transferFunds that transfers money from one account to another. Some SCT considerations might be:

Preconditions:

Ensure that the source account has sufficient funds.

Validate that the destination account exists and is accessible.

Check that the transfer amount is a positive value.

Postconditions:

Verify that the source account balance is decreased by the transfer amount.

Confirm that the destination account balance is increased by the transfer amount.

Ensure that the system remains in a consistent state, even in the event of failures (e.g., network errors, transaction failures).

Program:

```
#include<iostream>

#include<exception>

using namespace std;

float divide_numbers(float dividend, float divisor);

bool check_for_divisor(float number);

int main()

{

   try
```

```
    {
        float number, number2;
        cout << "Enter a number you wish to divide: ";
        cin >> number;
        cout << "Please enter your divisor: ";
        cin >> number2;
        float answer = divide_numbers(number, number2);
        cout << "Your answer is " << answer << "\n";
    }
    catch(int e)
    {
        cout << "An exception occurred. Exception Nr. " << e << '\n';
    }
    return 0;
}
float divide_numbers(float dividend, float divisor)
{
    bool precheck = check_for_divisor(divisor);
    if (precheck == false)
        throw 1;  // Throw an exception if divisor is zero
    float answer = 0.0f;
    answer = dividend / divisor;
    return answer;
}
bool check_for_divisor(float number)
{
    return (number != 0);
```

}

Output:

Enter a number you wish to divide: 10

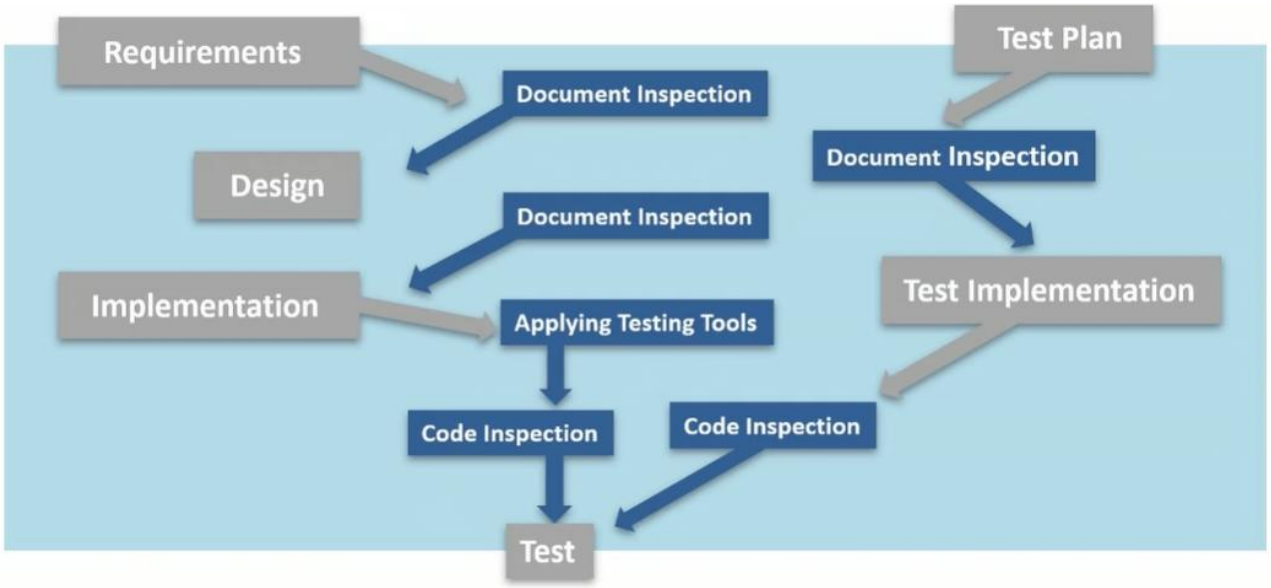
Please enter your divisor: 2

Your answer is 5

Low level design inspections

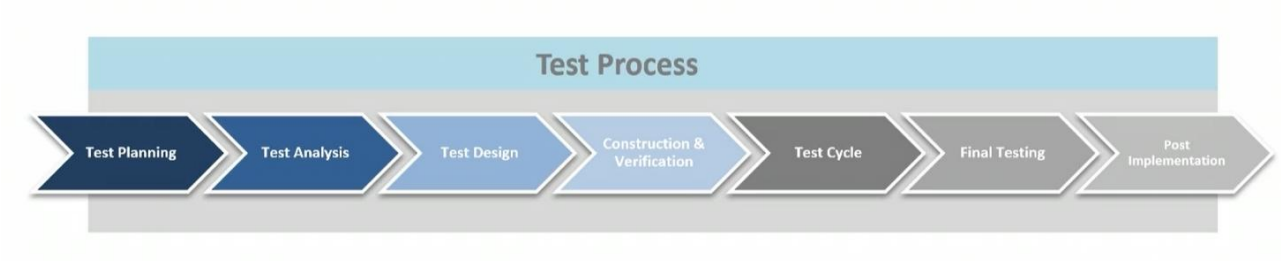
Inspections

1.Remove problems at a reduced cost
Cheaper to fix now than later
2.Do not prevent problems from happening
Gets rid of them as they occur
3.Provide value regardless of language or development methodology
4. Effective and efficient



Introduction to Testing for C/C++

1.Need software to be dependable
2.Testing let us asses its robustness
3.Prevent problems before deployment
4.Budget for testing until roll out
Failure to test more costly
5.Write testable code
6.Unit Testing



Testable Code

Test-driven development
An object should do exactly one thing
Interface Segregation Principle
Constructor should not do more than initialize
Readable

Bad Signs

Object not fully initialized after the constructor finishes
Anything more than field assignment in constructors
Static fields or static methods

Unit Testing for C/C++

Unit Testing

1.Tests to see if code does what it is supposed to do
2. Test functionality to discover discrete testable behaviors
Test these as individual units
3.Test the behaviors and report the results

Testing Tools Include

1.Google Test
Open source testing tool from Google
2.Boost Test Library
Provides Boost Test Library IUnit Test Framework
3.CppUnit
Port of Junit testing framework
4.Variety of open source testing tools exist

Java

Managing Denial of Service

The Denial of Service (DoS) attack is focused on making a resource (site, application, server) unavailable for the purpose it was designed. There are many ways to make a service unavailable for legitimate users by manipulating network packets, programming, logical, or resources handling vulnerabilities, among others. If a service receives a very large number of requests, it may cease to be available to legitimate users. In the same way, a service may stop if a programming vulnerability is exploited, or the way the service handles resources it uses.

Sometimes the attacker can inject and execute arbitrary code while performing a DoS attack in order to access critical information or execute commands on the server. Denial-of-service attacks significantly degrade the service quality experienced by legitimate users. These attacks introduce large response delays, excessive losses, and service interruptions, resulting in direct impact on availability.

NewFile.xml

```
<?xml version=”1.0” encoding =”UTF-8”?>

<dos>

<blacklistentries>
        <blacklist>
                <subnet>abcd::123:4567/48</subnet>
                <description>an IPv6 subnet</description>
        </blacklist>
</blacklistentries>
<static-error-handlers>

        <handler error-code=”dos_api_denial” file=”dos-response.html”/>
<static-error-handlers>

</dos>
```

Securing Information in Java

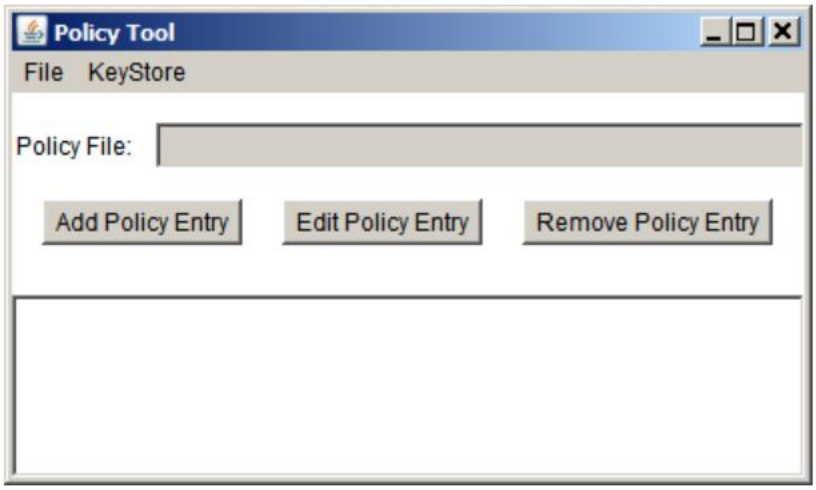
Start Policy Tool

To start Policy Tool, simply type the following at the command line:

policytool

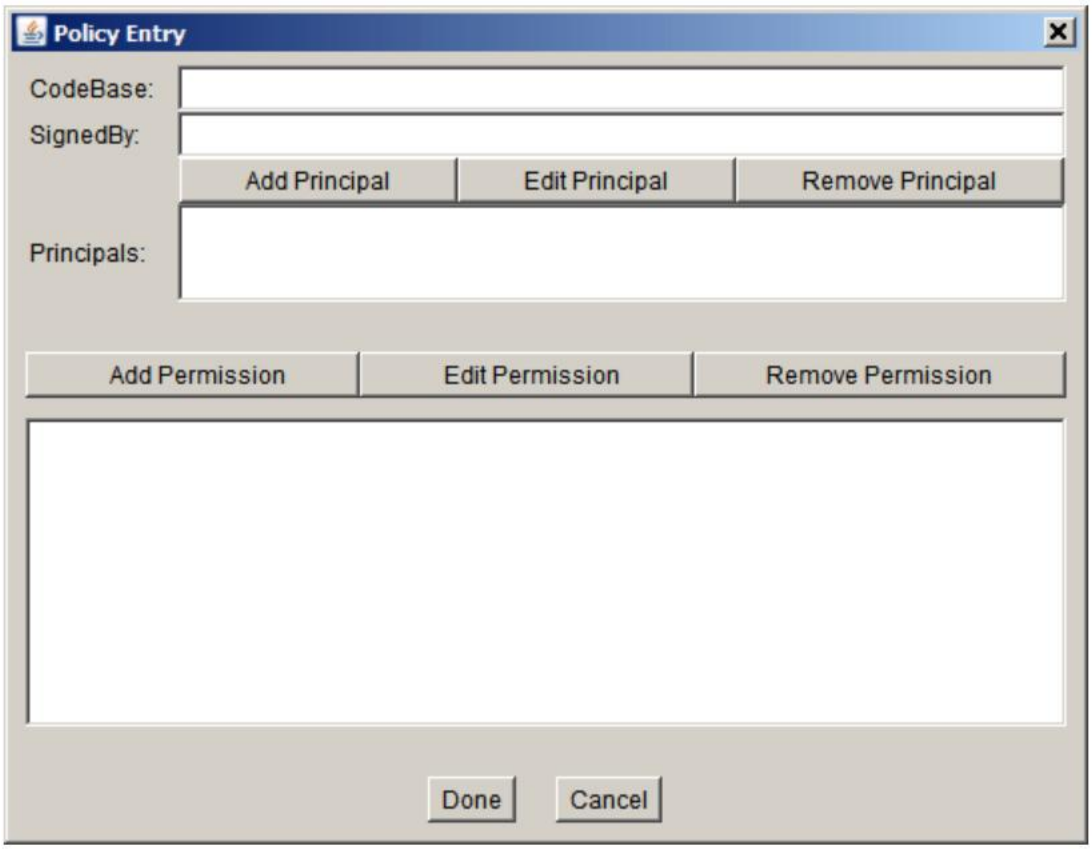
This brings up the Policy Tool window.

Whenever Policy Tool is started, it attempts to fill in this window with policy information from the user policy file. The user policy file is named .java.policy by default in your home directory. If Policy Tool cannot find the user policy file, it issues a warning and displays a blank Policy Tool window (a window with headings and buttons but no data in it), as shown in the following figure.



Grant the Required Permission

To create a new entry, click the Add Policy Entry button in the main Policy Tool window. This displays the Policy Entry dialog box as shown in the following figure.



A policy entry specifies one or more permissions for code from a particular code source - code from a particular location (URL), code signed by a particular entity, or both.

The CodeBase and the SignedBy text boxes specify which code you want to grant the permission(s) you will be adding in the file.

A CodeBase value indicates the code source location; you grant the permission(s) to code from that location. An empty CodeBase entry signifies "any code" -- it does not matter where the code originates.

A SignedBy value indicates the alias for a certificate stored in a keystore. The public key within that certificate is used to verify the digital signature on the code. You grant permission to code signed by the private key corresponding to the public key in the keystore entry specified by the alias. The SignedBy entry is optional; omitting it signifies "any signer" -- it does not matter whether the code is signed, or by whom.

If you have both a CodeBase and a SignedBy entry, the permission(s) are granted only to code that is both from the specified location and signed by the named alias.

You can grant permission to all code from the location (URL) where examples are stored.

Type the following URL into the CodeBase text box of the Policy Entry dialog box:

https://docs.oracle.com/javase/tutorial/security/tour1/examples/

Note: This is a URL. Therefore, it must always use slashes as separators, not backslashes.

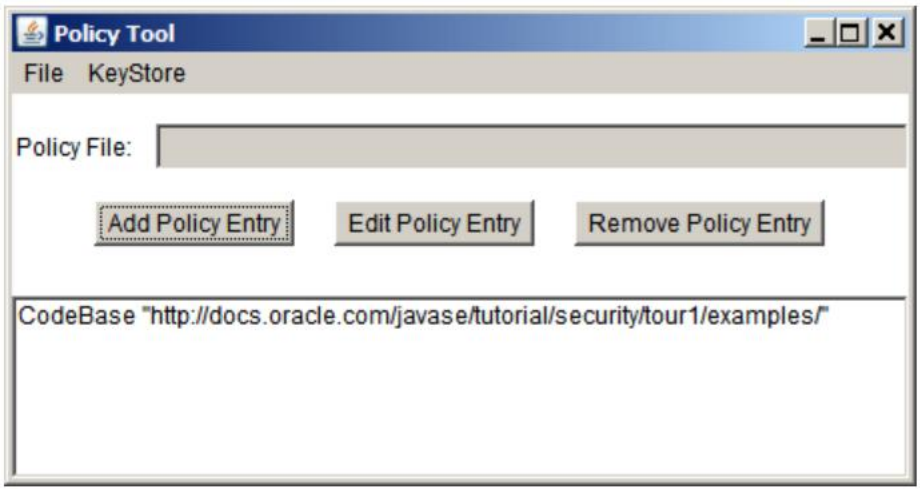
Leave the SignedBy text box blank, since you aren't requiring the code to be signed.

Note: To grant the permission to any code (.class file) not just from the directory specified previously but from the security directory and its subdirectories, type the following URL into the CodeBase box:

https://docs.oracle.com/javase/tutorial/security/

You have specified where the code comes from (the CodeBase), and that the code does not have to be signed (since there is no SignedBy value).

You have now specified this policy entry, so click the Done button in the Policy Entry dialog. The Policy Tool window now contains a line representing the policy entry, showing the CodeBase value.

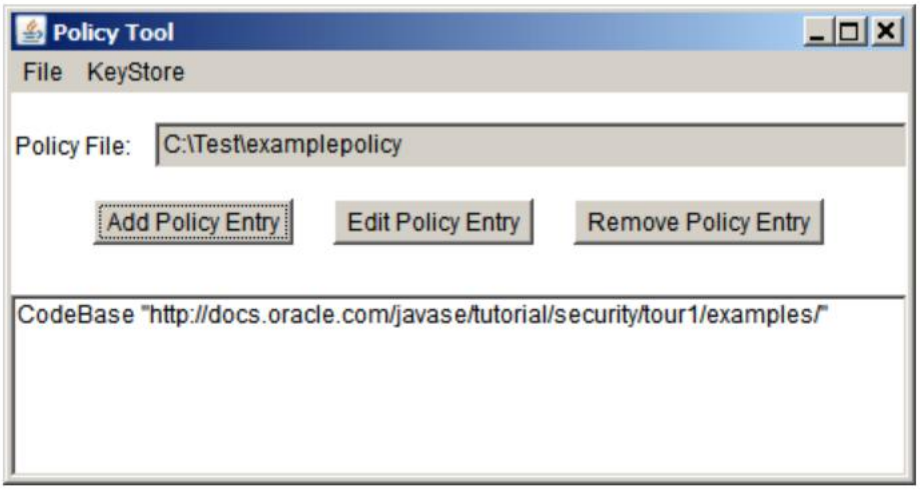


Save the Policy File

To save the new policy file you've been creating, choose the Save As command from the File menu. This displays the Save As dialog box.

Navigate to the Test directory. Type the file name examplepolicy and click Save.

The policy file is now saved, and its name and path are shown in the text box labeled Policy File.

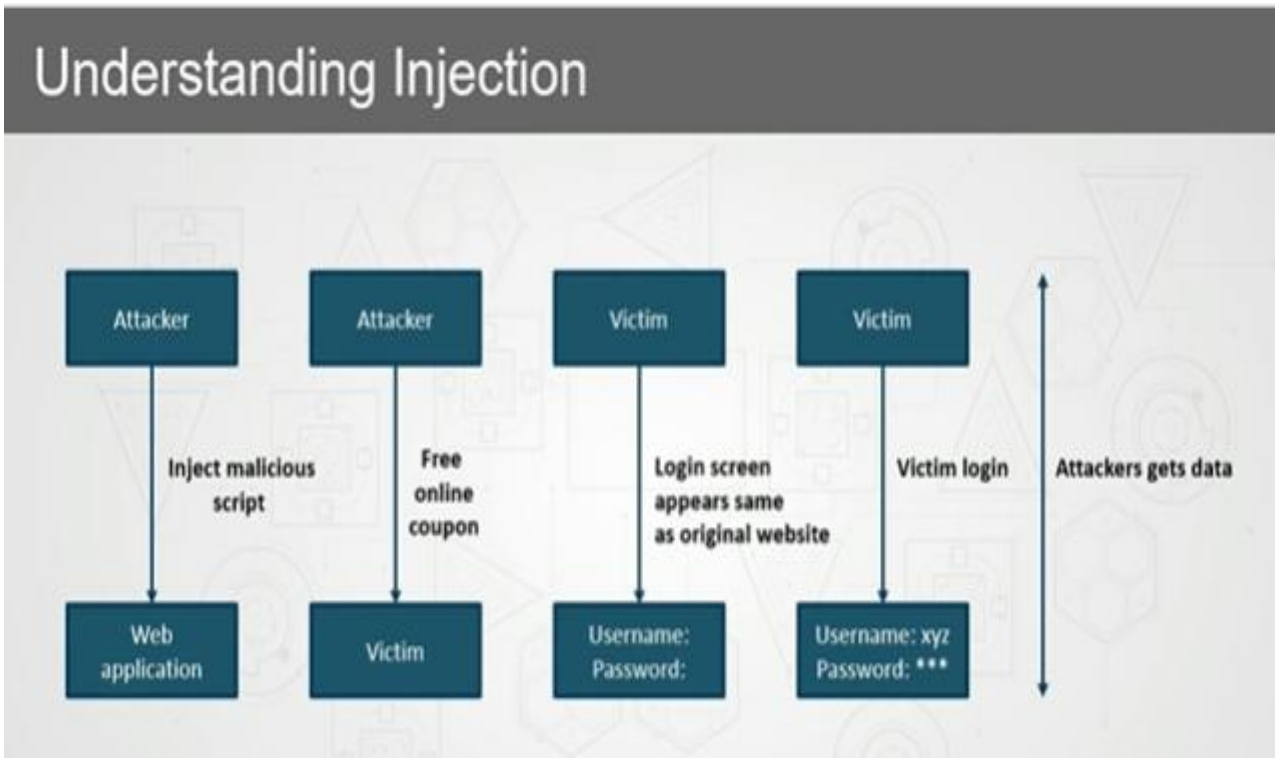


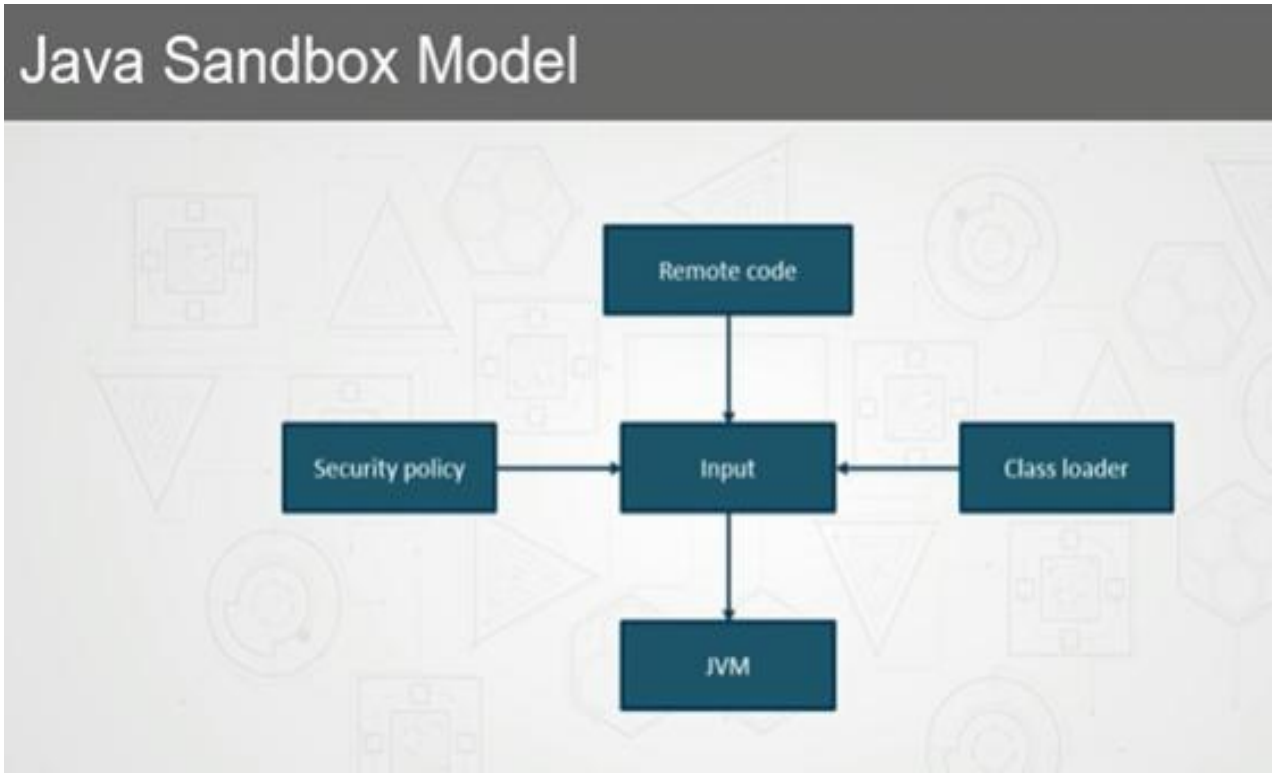
Exit Policy Tool by choosing Exit from the File menu.

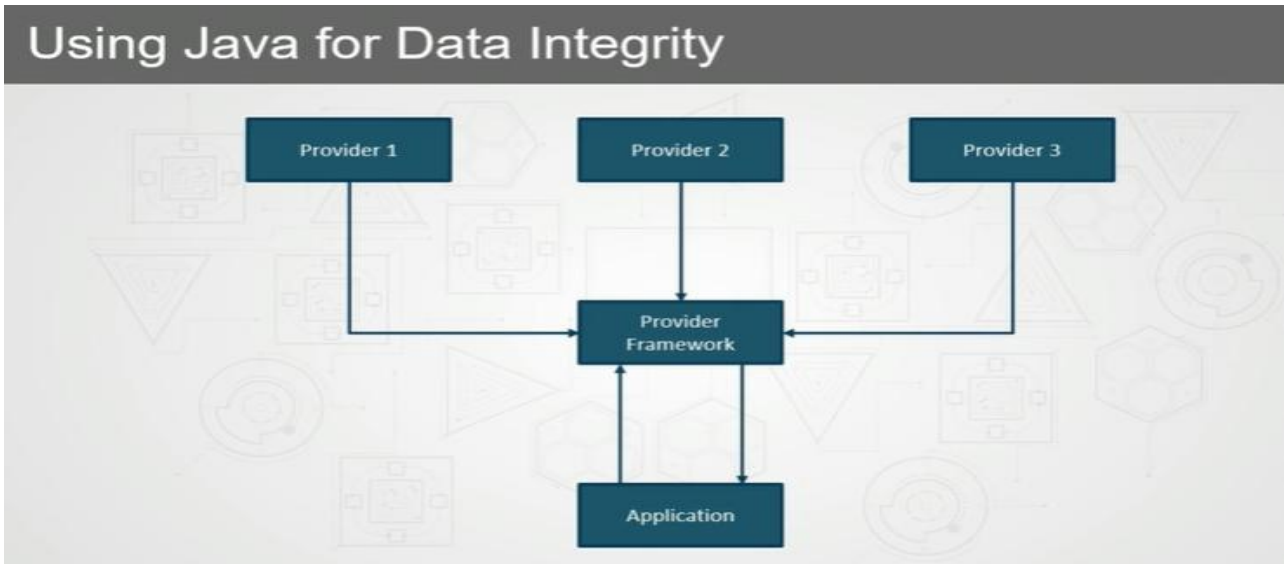
Data Integrity

Managing data to retain

- Data completeness
- Consistency
- Accuracy


Understanding Injection
Attacker
Attacker
Victim
Victim
Inject malicious script
Free online coupon
Login screen appears same as original website
Victim login
Attackers gets data
Web application
Victim
Username: Password:
Username: xyz Password: ***


Java Sandbox Model
Remote code
Security policy
Input
Class loader
JVM

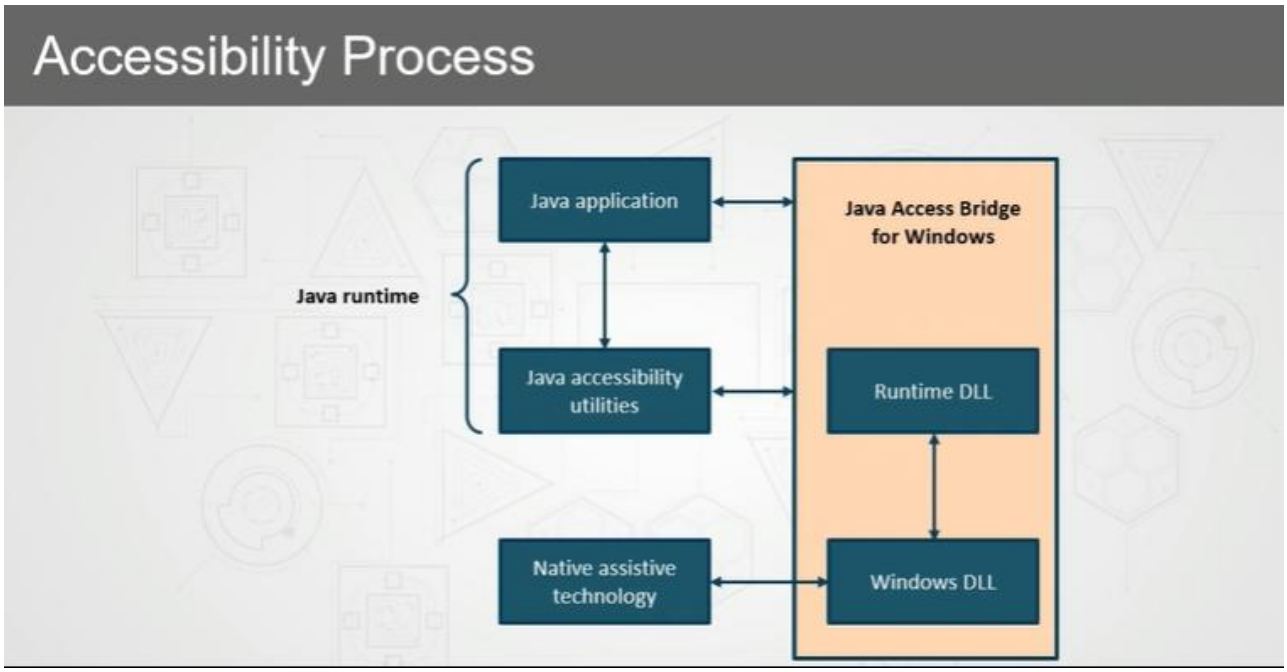


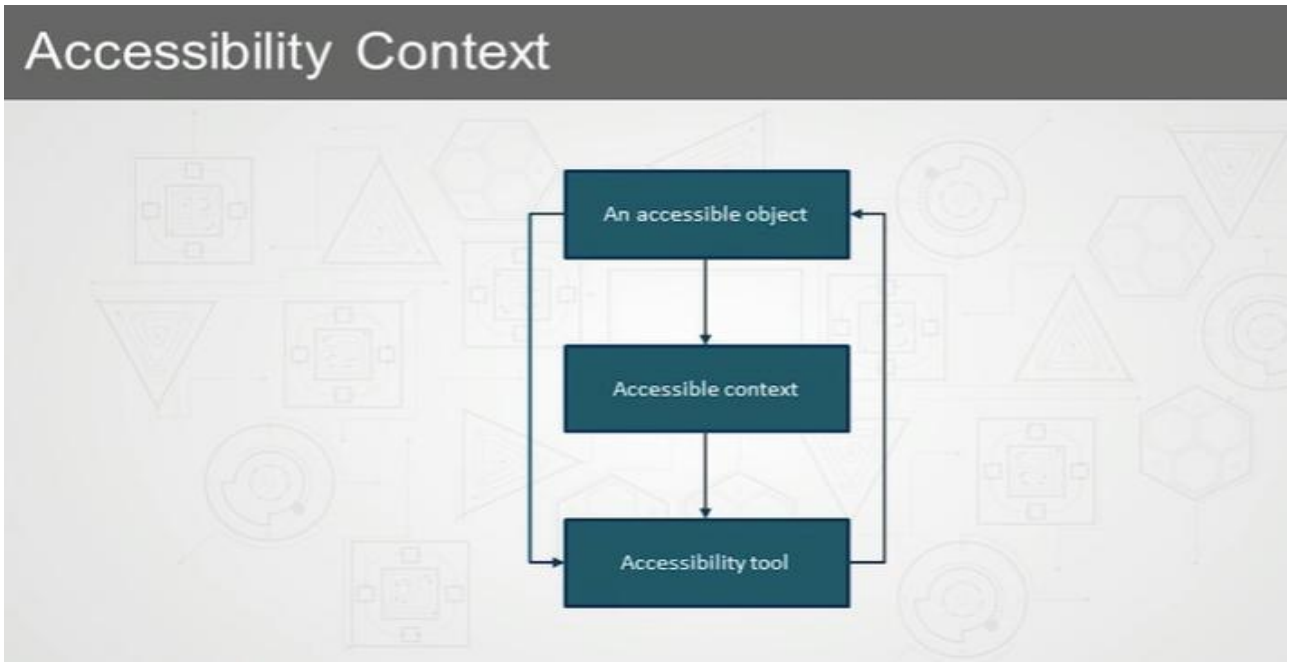
Accessibility and Extensibility

Capability of plugging in alternative looks and feels for applications
Independent Java Accessibility API toolkit providing boilerplate interfaces
Provides utilities supporting assistive technologies

API Interfaces and Classes

Interface Accessible
Interface AccessibleContext
Interface AccessibleAction
Interface AccessibleComponent





Securing Objects

Securing objects in Java involves implementing practices to ensure the integrity, confidentiality, and reliability of object-oriented code. In the given code, we have an Employee class and a Driver class that creates and prints details of an Employee object. Here are some considerations for securing objects in Java:

1. Encapsulation:

Explanation: Encapsulation is one of the fundamental principles of object-oriented programming. It involves bundling data (attributes) and methods (functions) that operate on the data into a single unit (class). In the Employee class, encapsulation is used to protect the internal state of the object by making the attributes private and providing public methods to access and modify them (getters and setters).

Security Benefit: Encapsulation helps in controlling access to the internal state of objects, preventing direct manipulation and enforcing proper usage through well-defined interfaces.

2. Serializable Interface:

Explanation: The Serializable interface is implemented by the Employee class. This allows instances of the class to be serialized into a byte stream, which is useful for tasks like object persistence or network communication.

Security Benefit: Serialization allows for secure storage and transmission of objects, but it also introduces potential security risks. Ensure that sensitive information is properly handled and marked as transient if it should not be serialized.

3. Data Validation:

Explanation: The Employee class provides a constructor and a static method (createEmployee) for creating an Employee object. Data validation should be implemented in these methods to ensure that only valid data is used to create an object.

Security Benefit: Validating input data helps prevent the creation of objects with invalid or maliciously crafted data, reducing the risk of security vulnerabilities.

4. Immutable Objects:

Explanation: The Employee class, as presented, is mutable (attributes can be changed after object creation). Consider making the class immutable by removing the setters and ensuring that all attributes are set through the constructor.

Security Benefit: Immutable objects are inherently thread-safe and less prone to unintended modifications, enhancing the reliability and predictability of code.

5. Access Modifiers:

Explanation: The Employee class uses private access modifiers for its attributes, limiting direct access to them from outside the class. The public methods (getters and setters) provide controlled access to these attributes.

Security Benefit: Restricting direct access to internal attributes enhances security by preventing unintended modifications or unauthorized access.

6. Code Reviews and Static Analysis:

Explanation: Regularly review code for security issues and consider using static analysis tools to identify potential vulnerabilities.

Security Benefit: Code reviews help identify and address security issues early in the development process, reducing the likelihood of security vulnerabilities in the final code.

// **Employee.java**

```
import java.io.Serializable;

public class Employee implements Serializable {

 int id;

String name;

 int age;

 public int getId() {

return id;

 }
```

```
public void setId(int id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public Employee(int id, String name, int age) {
    super();
    this.id = id;
    this.name = name;
    this.age = age;
}
```

```
    public static Employee createEmployee(int id, String name, int age) {
        return new Employee(id, name, age);
    }
}
```

// Driver.java

```
public class Driver {
    public static void main(String[] args) {
        Employee obj = Employee.createEmployee(100, "cody", 25);

        // Print the details of the created Employee object
        System.out.println("Employee ID: " + obj.getId());
        System.out.println("Employee Name: " + obj.getName());
        System.out.println("Employee Age: " + obj.getAge());
    }
}
```

Output:

Employee ID: 100

Employee Name: cody

Employee Age: 25

Conclusion:

Securing objects in Java involves adopting best practices such as encapsulation, data validation, and access control. By following these practices, developers can create robust and secure object-oriented code. Additionally, consider ongoing code reviews, adherence to secure coding standards, and awareness of potential security risks introduced by specific language features, such as serialization.

Serialization Security

Serialization in Java is a mechanism that allows objects to be converted into a stream of bytes, making it possible to save the object's state, send it over a network, or store it in a persistent storage. While serialization is a powerful and flexible feature, it introduces security concerns that developers need to be aware of.

Program:

//Employee.java

```
import java.io.Serializable;
public class Employee implements Serializable
{
int id;
String name;
int age;
public int getId()
{
return id;
}
public void setId(int id)
{
this.id=id;
}
public String getName()
{
return name;
}
public void setName(String name)
{
this.name=name;
}
public int  getAge()
{
return age;
```

```
}
public void setAge(int age)
{
this.age=age;
}
public Employee (int id,String name, int age)
{
super();
this.id=id;
this.name=name;
this.age=age;
}
public static Employee createEmployee(int id,String name,int age)
{
return new Employee(1,”blackwell”,30);
}
}
```

//SerializationWithTransient.java

```
import java.io.*;
public class SerializationWithTransient {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        final Employee employee = new Employee();
        employee.setId(100);
        employee.setName("cody");
        employee.setAge(100);
        serializeProcessing(employee);
        deserializeProcessing();
    }
```

```
    private static void serializeProcessing(Employee employee) throws IOException {
        final FileOutputStream fout = new FileOutputStream("F.txt");
        final ObjectOutputStream out = new ObjectOutputStream(fout);
        out.writeObject(employee);
        out.flush();
        out.close();
        System.out.println("Serialization success");
    }

    private static void deserializeProcessing() throws IOException, ClassNotFoundException {
        final FileInputStream fin = new FileInputStream("F.txt");
        final ObjectInputStream in = new ObjectInputStream(fin);
        final Employee employee = (Employee) in.readObject();
        in.close();
        System.out.println("Deserialization success");
        System.out.println("Employee ID: " + employee.getId());
        System.out.println("Employee Name: " + employee.getName());
        System.out.println("Employee Age: " + employee.getAge());
    }
}
```

Output:

Serialization success

Deserialization success

Employee ID: 100

Employee Name: cody

Employee Age: 100