Lecture topics

- Class design for security
  - Visibility of classes, fields, and methods
  - Implications of using inner classes
  - Mutability
  - Design for sending objects across JVMs (serialization)

Visibility modifiers in Java

- Visibility of interfaces and classes can be
  - Public
    - All parts of the application can use it
  - Package
    - Can be used by name only in the same package
  - Inner classes and interfaces

- Visibility of fields and methods of a class can be
  - Public
    - All parts of the application can use it
  - Protected
    - Can be used only in the same package and from subclasses in other packages
  - Default (no modifier)
    - Can be used only in the same package
  - Private
    - Can be used only in the same class
Lecture 7: Class design for security

What’s the purpose of visibility modifiers?

- Information hiding
  - Visibility of a class (interface, field, method) should be as restricted as possible
- Clear abstraction of program components
- Does visibility of program components affect security?
  - Intuitively, it should
    - A secret may be embedded in an object as its private field
    - Untrusted code can be prevented from executing sensitive code by placing it in a package-local class

A concrete example of visibility modifiers affecting program security

```java
public class Auction {
    public List bids;
    public Auction() {
        this.bids = new ArrayList();
    }
    public void bid(int amount) {
        if (!this.bids.isEmpty()) {
            int lastBid = ((Integer) this.bids.get(this.bids.size() - 1)).intValue();
            if (lastBid >= amount) {
                throw new RuntimeException("Bids must be higher than previous ones");
            }
        }
        this.bids.add(new Integer(amount));
    }
    public int getHighestBid() {
        if (this.bids.isEmpty()) {return 0;}
        return ((Integer) this.bids.get
            (this.bids.size() - 1)).intValue();
    }
}
```

It’s not a problem if a client does the following:
```
... auction.bid(newBid);
... auction.getHighestBid();
```

But it is a problem if a client does the following:
```
... auction.bids.add(newBid);
... auction.bids.get(i);
```
Lecture 7: Class design for security

So, how protected are protected and default visibility object members?

- By default, not that much
  - A malicious client can have a class that claims to have the same package
  - A malicious client can have a class that extends a trusted class and accesses protected fields
- It would be nice to have a way to prohibit packages from being added to

Protecting packages from getting joined: sealed JAR archives

- Can add a simple entry to the manifest of a JAR file, saying that packages (all or some) in this JAR cannot be joined
  
  Name: edu.poly.glebssecurepackage
  Sealed: true

- The JAR file should contain the complete package if this package is sealed
- Does not protect if the JAR is given to clients
Protecting packages from getting joined: via local security policy

- Can insert in the java.security file:
  package.definition=
edu.poly.glebssecurepackage

- Need additional assignments of permissions to join this package to classes that are allowed to

- A big gotcha: no class loaders from Sun support this at present!

Situations arise where different parts of the application should not use the same type

- Often, because different access levels exist for accessing information in the type
  - E.g., Auction administrators may need to get access to history information about users, but users themselves should not

    package auction.common;

    public interface User {
        public String getHistory();
        public void appendHistory(String entry);
        public void setHistory(String history);
    }
Lecture 7: Class design for security

**Solutions?**

- **Sacrifice simplicity of design for security**
  - Need two different interfaces and two different classes to represent users
    - Having two different interfaces and one class that implements both of them does not work for the same reasons private fields do not protect data on the client machine
  - It is important not to throw good design principles out the window, blaming security
    - Two different user implementation classes should do code reuse
    - Several design patterns are useful in such situations
      - E.g decorator, proxy

---

**Java inner classes**

- Classes defined as members of other classes
- Inner classes are allowed to access private members of the enclosing class and vice versa
- For each instance of the outer class there is a corresponding instance of the inner class
- Useful especially for defining in-line implementations of simple interfaces

```java
class A {
    private int a;

    class B {
        private int b;
        private void f() {
            b = a*2;
        }
    }

    public g() {
        B bObj = new B();
        bObj.f();
        bObj.b = 2;
    }
}
```

- B accesses a private field of A
- A accesses a private field of B
Lecture 7: Class design for security

Inner classes are not understood by JVM

- No notion of inner classes during run-time
- Java compilers must transform inner classes into top-level classes
  - But JVM prohibits access to private members from outside the class!
  - Compiler provides access to private fields accessed by inner (outer) classes via package-local methods

Security implications of this transformation

- Private data members get exposed through non-private methods
- Other classes from the same package can call these methods and tamper with object state
But so what --- attacker classes will be in other packages, right?

- Defense in depth is one of the important principles of security
- Using inner classes removes one of Java security barriers --- private visibility
- In principle, fixing the way inner classes in Java are handled wouldn't be too difficult
  - Proposal by Bill Pugh
  - Based on sharing a secret key among all classes that need access to private members of a class
    Does this sound familiar?
      - C++ friends

Illustration of Pugh’s proposal

```java
Class A {
    static private final Object sharedSecret = new Object();
    static {
        A$B.receiveSecretForA(sharedSecret);
    }
    private int x;
    int access$1(Object secretForA) {
        if (secretForA != sharedSecret) throw new SecurityException();
        return x;
    }
}
Class A$B {
    private A this$0;
    static private Object sharedSecret;
    static void receiveSecretForA(Object secretKey) {
        if (sharedSecret != null) throw new VerifyError();
        sharedSecret = secretKey;
    }
    ... invoke this$0.access$1(sharedSecret)...
}```
Changing visibility of fields when subclassing

- Java allows increasing visibility of a member

```java
public class C {
    protected void m() {...}
    ...}
}
public class D extends C {
    public void m() {...}
    ...}
}
```

- Situations where this is desirable are rare
- Java does not allow decreasing visibility of a member

Object immutability

- An object is immutable if its state cannot be modified
  - Once created, none of the fields can be changed
  - Example: String

- Advantages of immutability:
  - Sometimes, good API design requires it
  - Simplicity --- if you want to change an object, create a new one
  - Can be shared freely --- no side-effects are possible
  - In some cases, to check equality, instead of checking equality of fields, can check whether two variables of immutable types are a reference to the same object

- Disadvantages of immutability:
  - Many objects may need to be created
    - Can be solved by having a mutable counterpart for each immutable class
    - E.g. StringBuffer for String
Rules of defining immutable classes

- Don’t provide any methods that can change class fields
  - Constructors are allowed to modify fields
- Make sure that no methods can be overridden
- Make all fields final
- Make all fields private
- Ensure exclusive access to mutable fields
  - If the class has a mutable field, the reference to this field cannot be returned via a method call
  - Defensive copies must be made

Quiz: is this class immutable?

```java
public class Student {
    private final String name;
    private final Calendar birthDate;
    private final int id;

    public Student(String name, Calendar birthDate, int id) {
        this.name = name;
        this.birthDate = birthDate;
        this.id = id;
    }

    public final String getName() {
        return this.name;
    }

    public final Calendar getBirthDate() {
        return this.birthDate;
    }

    public final int getId() {
        return this.id;
    }
}
```
Answer: NO!

- The reason: a reference to a mutable field is returned
- Important: in Java, if a field (variable, parameter) is final, it does not mean that it’s fields cannot change, only that the reference cannot change
- A way to mutate a Student:
  ```java
  Calendar stolenBirthDate = student.getBirthDate();
  stolenBirthDate.set(Calendar.YEAR, 1900);
  ```
- See cs916.immutability

The fix

- Make a copy of the date before returning it
  ```java
  public final Calendar getBirthDate() {
      return (Calendar) this.birthDate.clone();
  }
  ```
Lecture 7: Class design for security

**Immutable forms of Java collections**

- What if you want to return a large collection (list, set, map, etc) field reference from a method?
  - Can clone
    - Computationally expensive for large collections
  - **Class Collections** has methods
    - `public static List unmodifiableList(List list)`
    - `public static Set unmodifiableSet(Set s)`
    - `public static Collection unmodifiableCollection(Collection c)`
    - Wait, but if it returns a List, the client code can still call methods that modify the list, e.g. add?
    - All methods that *would* modify the list throw exception `UnsupportedOperationException`
    - This method is not as good as having a special immutable type
      - E.g. `UnmodifiableList` interface without `add`, `insert`, `remove`, ... methods
      - Problems with client code are detected at compile time with the immutable type and only at run time with throw exception

---

**Sending objects between JVMs**

- J2EE programs allow remote method calls (calls across JVMs). We sure can’t do argument passing by reference...
  - Intuitively, need to write data in an object as a stream of bits
  - Tedious if we have to do it for every class
  - Can get complicated and error-prone if objects are complex (lots of references to other objects)
- In Java, the **object serialization** mechanism already does this work for you
  - Serialization can also be used to save objects persistently locally
Lecture 7: Class design for security

The basic recipe for serialization

- Make sure the class implements the `Serializable` interface
  - It's a *mixin* interface: does not have any methods, just marks the class as being allowed to use the serialization mechanism
- Make sure that all fields of the class are either
  - Primitive types
    - The serialization mechanism knows how to deal with them
  - Serializable reference types
    - Lots of standard classes are serializable (String, Date, …)
  - Marked as transient
    - Their values are not written during serialization and not restored during de-serialization
- See `course916.serialization.Student`

What is actually written when an object is serialized?

1. Class of the object
2. Signature of the class
3. Values of object fields
   - static fields are not written
   - transient fields are not written
   - Non-transient fields must be of primitive type or of classes that are serializable

The process of serialization follows references:
Lecture 7: Class design for security

**What about recursive data structures?**

![](image)

- See course916.serialization.linked
- What prevents the process of serialization from saving the same object a number of times?
- The answer will explain where the term serialization comes from
  - Each object is assigned a serial number
  - The ObjectOutputStream checks, for each object passed to it, if this object has already been written. If yes, it ignores this object
  - This means that if you change the object and try to write it again, it will not be written
  - See the modified course916.serialization.linked example

**Can I customize object serialization?**

- Yes
  - By creating private writeObject and readObject methods in the class whose objects are serialized
    - These methods describe how the object is serialized/de-serialized
      - It’s a good idea to call defaultWriteObject of ObjectOutputStream from writeObject and defaultReadObject of ObjectInputStream from readObject
      - See course916.serialization.StudentFixed
  - Wait, but if writeObject and readObject are private, how can they be called outside of the class where they are defined?
    - Implementation of object streams is native (in C or C++), so it is not subject to visibility rules
Default serialization mechanism is often unacceptable for performance reasons

- See course916.serialization.IntegerMap
- Although HashMap is a serializable class, its objects often are very expensive to serialize
  - Holds true for most complex data structures
  - In many cases, it is faster to re-construct the data structure given its elements
  - See course916.serialization.IntegerMapImproved
    - Note that defaultReadObject and defaultWriteObject are still called
    - It is important for compatibility with future versions of this class that may add non-transient fields
      - E.g., a serialized object of a later version can be de-serialized in the earlier version

Another important benefit of customizing serialization in this example

- It is not only performance that is improved here...
  - Abstraction is improved too
    - Serialization format in IntegerMap includes implementation details
      - A HashMap is used internally
    - Serialization format in IntegerMapImproved does not!
      - Later versions may use any data structure that stores pairs
- Hash tables are generally tricky to serialize
  - Even the same version of JVM may fail to de-serialize a hash table
    - Hash table implementations sometimes make placement of key-value pairs non-deterministic
Lecture 7: Class design for security

Can I make my class non-serializable?

- Yes, just override methods writeObject and readObject, in the following way:
  private void writeObject(ObjectOutputStream out) throws IOException {
    throw new NotSerializableException("Objects of this class are not serializable!");
  }

- See course916.serialization.nonserializable

Serialization and potential for security holes

- Objects obtained from client applications should be validated for well-formedness
  - The serialization format is well known and it's not too hard to construct serialized data by hand
- Using default serialization is tempting, but
  - Can expose sensitive data fields
  - Can reveal implementation details
    - E.g. use of HashSet
- Developers often have to reconcile objects already on the server with objects received from clients
  - Often causes bugs
- Serialization of objects should be an important part of design of Web applications