OPTUS

MACQUARIE University

Using Cryptographic Function Libraries

Outline

- Assignment 1
- Language and tools
- Structure of cryptographic libraries
- Online documentation
- Creating a key
- Choosing a cipher
- Setting the cipher mode and padding
- Creating a cipher
- This lecture uses Java for examples
  - But the key issues and concerns are language independent
  - And other languages (C, C++, C#, Python, etc.) have similar libraries

Objectives

- Gain exposure to real cryptographic code
- Observe the variety of options available and learn to navigate them safely
- Provide an opportunity to actually usefully encrypt something (probably a good outcome from an *applied* cryptography course!)
- I have a range of assignment exercises, involving encrypting and decrypting files, generating digests, signing messages, etc. and this lecture provides an overview of the Java Cryptography Architecture
  - The details are in the assignment instructions and the JCA online documentation

OPTUS

MACQUARIE University

Assignment 1

Assignment 1

- Due in Week 7
  - Edit a program, run it, answer a few questions, upload your source code and answers
- Allows you to explore
  - The capabilities of the Java Cryptography Architecture
  - How ciphers work in the real world
  - The effect of block modes
  - Hashes and digests
  - Signatures,
  - Useful crypto functions and their documentation
- Written in Java – useful for real-world enterprise applications involving crypto
- Make use of the Assignments Discussion Forum to discuss programming problems

OPTUS

MACQUARIE University

Some Object-Oriented Programming Basics

Some OO Architecture Stuff

- A *class* provides a bundle of functions (*methods*) and variables that can be used to instantiate an object and operate on it
  - Classes can extend other classes – inherit all the superclass's functionality, then extend or over-ride it:

```
Class Helicopter extends Aircraft {
    final double landingDistance = 0.0;
    …
}
```

- An *interface* defines a bundle of methods which are abstract (no code) that must then be implemented by other class(es)
  - Classes that provide implementations for interfaces or abstract superclasses are sometimes called providers
    - You find this in, e.g., Microsoft .Net and C# as well as Java
- Instantiate a new object of a class with the new keyword
- In practice, this exercise does not require you to implement any new classes, just create a few objects

Setting Up a Java Development Environment

- Download the Java SE Development Kit from https://www.oracle.com/au/java/technologies/javase-downloads.html
  - Install it
  - Optionally download the JavaDoc from the “Documentation Download” link on the same page
    - Install it into Eclipse for extra online help when coding
- Download Eclipse from https://www.eclipse.org/downloads/
  - Install it
  - It should find your installed JDK and automatically use it
- No other tools or libraries are necessary

OPTUS

MACQUARIE University

Eclipse

Toolbar: Debug, Run, New class, etc.

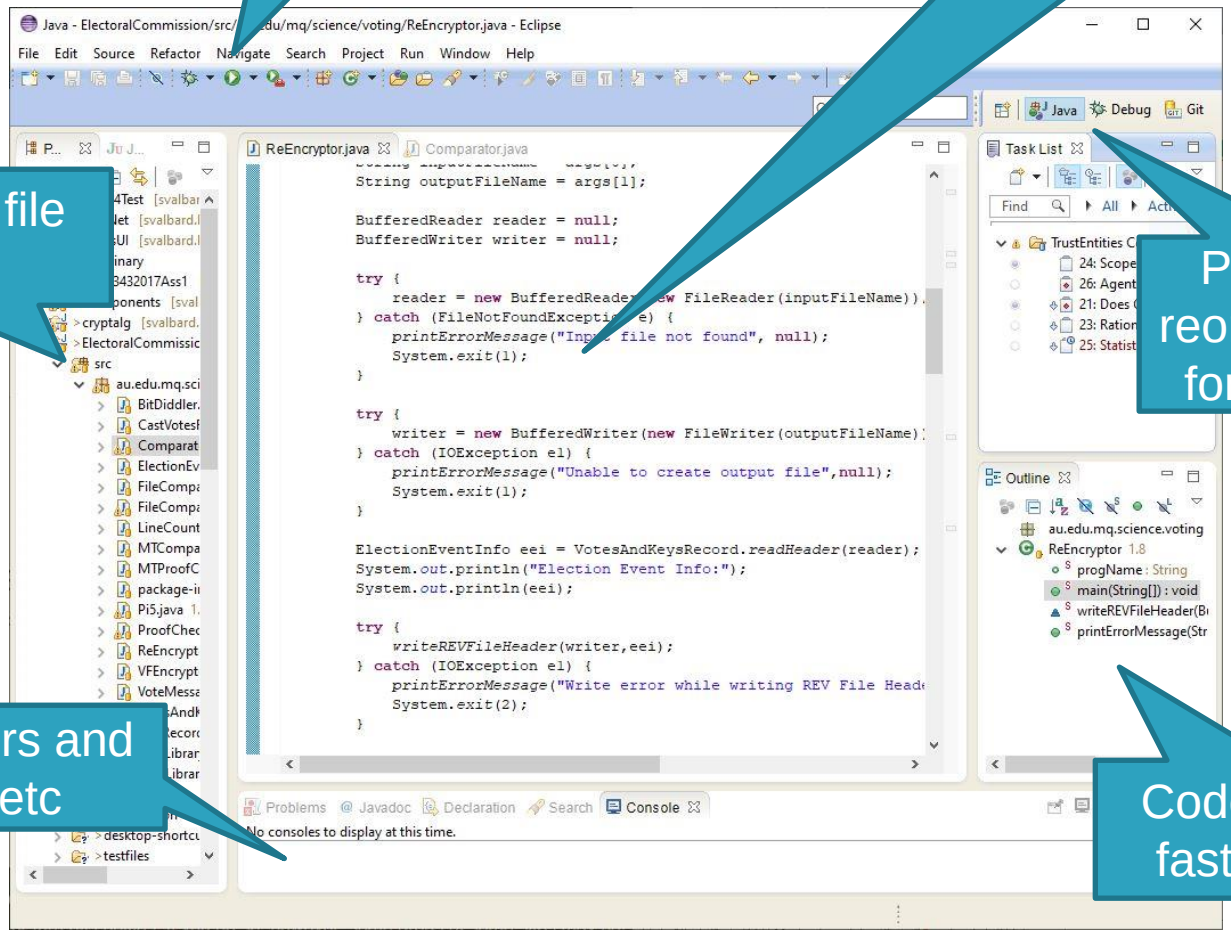
Syntax-aware editor with code completion, online help, etc

Source code file tree

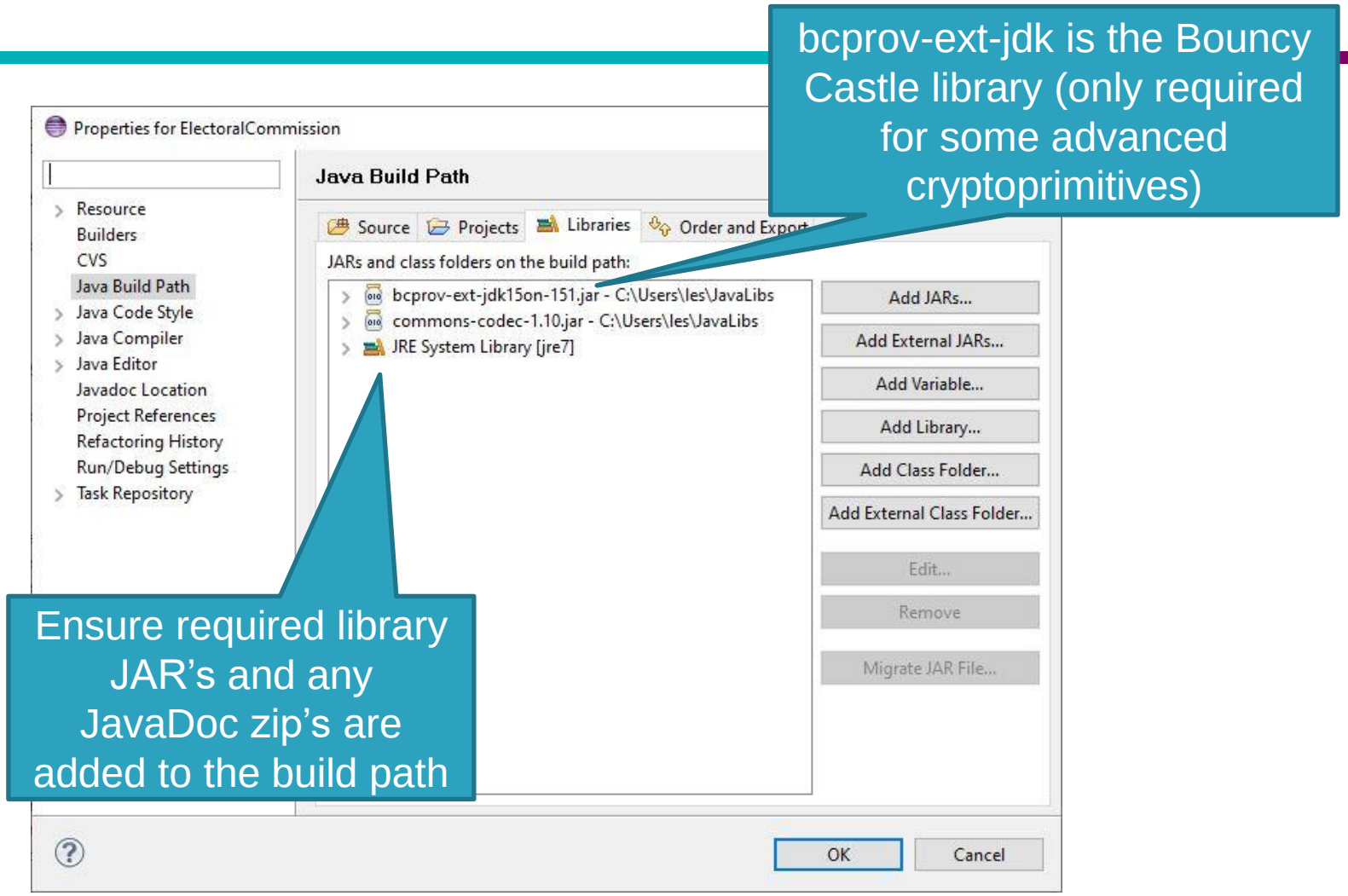
Perspectives: reorganize panes for current task

Console, Errors and warnings, etc

Code outline for fast navigation



The Java Build Path



OPTUS

MACQUARIE University

Big Integers

Big Integers

- Programming languages generally support floating point numbers (which are limited in precision) as well as the integer types provided by the underlying processor (16-bit, 32-bit, 64-bit word sizes)
- These are not enough for cryptography!
  - Symmetric crypto: 56-bit, 128-bit, 256-bit key sizes
  - Public-key crypto: 2048-bit, 3072 and 4096-bit moduli
- Question: how many decimal digits are required to represent a 4096-bit integer?
  - Answer: $4096 / \log_2(10)$
    - That's 1,234 digits long!
- Clearly, numbers this large are well beyond the capabilities of your pocket calculator and most programming languages!
  - For symmetric crypto, we use arrays of bytes
  - But for public-key crypto, we need to do mathematics
    - So Java provides a special class: BigInteger

The Java BigInteger Class

- Provides lots of useful methods for number theory experiments
  - Arithmetic: add(), subtract(), multiply(), divide(), remainder(), divideAndRemainder(), pow(), abs(), max(), min(), negate(), compareTo(), gcd(), signum()
  - Modular Arithmetic: mod(), modInverse(), modPow()
  - Logical: and(), or(), xor(), not(), andNot()
  - Primality testing: isProbablePrime(), nextProbablePrime()
  - Bit manipulation: setBit(), clearBit(), flipBit(), bitLength(), bitCount(), shiftLeft(), shiftRight(), getLowestSetBit()
- Constructors
  - From byte array, String or as random
- Constants
  - ZERO, ONE, TEN

See https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html

Some BigInteger Fragments

```
BigInteger a = new BigInteger("4720272346345"),
  b = new BigInteger("FFFFFFFFFFFFFFFFFFFFFF",16);
System.out.println("Larger of the two is " + a.max(b) + "
decimal, " + (a.max(b).toString(16)) + " hex");

BigInteger atb = a.multiply(b);
System.out.println("Product = " + atb);

BigInteger one = new BigInteger("1");
if(one.compareTo(BigInteger.ONE) == 0) {
    System.out.println(one + " = ONE");
}

BigInteger p = b.nextProbablePrime();
System.out.println("p = " + p.toString(16) + " hex; a ^ b mod p =
" + a.modPow(b, p));
```

Cryptographic Libraries

Cryptographic Libraries

- General advice: do not try to roll your own cryptoprimitives!
- As a programmer, make use of standard, tried-and-tested cryptoprimitives
  - Ideally in certified libraries, e.g. to FIPS 140-2, FIPS 197, etc.
- Some libraries are provided at the operating system level
  - E.g. OpenSSL
- Some at the language level
  - E.g. Java Cryptography Architecture Providers (Sun/Oracle, Bouncy Castle, etc.)
- Especially for semi-interpreted languages like Java and C#
  - Low-level bit-manipulation is painful and slow, so the providers are implemented in assembly language with a Java interface

Coding Security with Java

- Java provides extensive class libraries for security-related functionality
- Java Certification Path API
  - For creating and validating certificate chains
- Java GSS-API
  - Generic Security Services (see RFC 2743) – really Kerberos, etc.
  - RFC 2744 for C bindings and RFC 2853 for Java
- Java Authentication and Authorization Service
  - Generic authentication and authorization policy language
- Java Secure Socket Extension (SSL/TLS)
- Java Cryptography Architecture – today's topic

Java Cryptography Architecture

- A comprehensive set of API's for
  - Using symmetric (both stream and block) and public key ciphers
  - Computing message digests and hashes
  - Computing and verifying digital signatures
  - Generating and validating digital certificates and certificate chains
  - Cryptographic random number generation
  - Key generation and management
- Algorithm independence
  - Achieved through pluggable components called engines which define generic interfaces
- Implementation independence
  - Via API's open to third-party developers
    - Allows implementation of proprietary schemes

Pluggable Components

HANDLING MULTIPLE PROVIDERS

- The JCA provides generic interfaces which should suit almost any cryptoprimitive
- Actual implementations can be provided by anybody
  - Java Cryptography Extensions (Oracle)
  - Bouncy Castle (Legion of the Bouncy Castle)
  - Apple
- Collections of implementations are called Cryptographic Service Providers
  - These are written to the Service Provider Interfaces
    - Names of these interfaces typically end in Spi
    - Application programmers don't use these directly – rather, implementers do

Standard Algorithm Names

- The Java Cryptography Architecture uses a consistent set of names to identify algorithms, modes and padding mechanisms
  - These are used, rather than class names (which would tend to bind to specific provider classes)
    - The names are just strings, which allow for any future algorithms to be easily added
    - They are also called transformations:
    - "A *transformation* is a string that describes the operation (or set of operations) to be performed on the given input, to produce some output. A transformation always includes the name of a cryptographic algorithm (e.g., *AES*), and may be followed by a feedback mode and padding scheme. " (Cipher class Javadoc)
  - See the Standard Algorithm Name documentation for details
  - Some popular algorithms are implemented by multiple providers
    - Providers can be specified at run-time, if necessary
- You will see similar names being passed in the startup messages of protocols like SSH and TLS
  - Use Wireshark to view these

Loading Algorithms

- Load algorithms by name – but not all providers supply all possible combinations
  - During development use a construction like that below
    - Some trial-and-error required to find supported algorithms

```
try {
        cipher = Cipher.getInstance(“AlgorithmName");
}
catch (NoSuchAlgorithmException e) {
        System.err.println("No Such Algorithm Exception when
creating Cipher",e);
        System.exit(2);
}
catch (NoSuchPaddingException) {
        System.err. . . . (etc.)
}
```

Generators and Factories

- Generators are used to create objects with brand-new contents
  - For example, the KeyPairGenerator class can create a public and private key pair for use with a specified algorithm
  - May allow the programmer to specify parameters (e.g. p and g for Diffie-Hellman) or can automatically select values
- Factories create objects from existing material
  - For example, convert a KeySpec into a Key or vice versa, or use a CertificateFactory to convert the byte array of a certificate into a X509Certificate object

Dealing with Arrays of Bytes

- Cryptoprimitives deal with arrays of bytes
  - Not Strings or other more complex Java types and classes
    - A String object in Java is a more complex data structure with a size and other information. It is also immutable
  - Use `String.getBytes()` to turn a String into just the array of bytes if you need them
- Remember, encryption, decryption, hash functions, etc. need to deal with the full range of binary values – this is not classical crypto like the Caesar Cipher
  - For the same reason, we don't assume we're reading and writing lines of text

Encryption/Decryption with JCA

- Uses the Cipher class – for both symmetric and public-key crypto
- Create with the static generator method
  `Cipher.getInstance("AlgorithmName");`

Keys

- Key is really an interface, extended by lots of other interfaces:
  - SecretKey, PrivateKey, PublicKey
- Implemented by opaque key objects
  - Get keys from
    - A key factory
    - A key generator
    - A key store
      - An in-memory database for secret keys, private keys and certificates
  - These have three characteristics
    - An algorithm the key will work with, e.g. AES or DSA
    - An external coded form, which is the actual key value that can be extracted from the key for use with a cipher
      - Use the `byte[] getEncoded()` method to extract the key
    - A format – the name of the encoded key format (use `String getFormat()` if you need this)

PBKDF2

PASSWORD BASED KEY DERIVATION FUNCTION 2

- A widely-used technique for turning passphrases into keys
  - In 802.11 wifi WPA2 (Wifi Protected Access 2), Veracrypt, etc. etc.
- Hashes the password, then hashes the hash, then …
- Can use thousands of iterations to slow the process
  - Why?
    - Genuine user runs the algorithm once – it might add a few seconds
    - Attacker has to run it millions of times – high work factor
- Can generate a key of the required size
- Can optionally *salt* the password
  - e.g. WPA2 salts the password with the network SSID
  - JCA hint: the JCA methods do not like null parameters
    - (And for a password-based file encryption program, what would you use as salt anyway? You want the file to be decryptable anywhere, by anyone)

SecretKeyFactory Class

- All the symmetric (secret-key) ciphers can use keys supplied by this class
- Get one by calling the static method
  - `kf = SecretKeyFactory.getInstance(`*`AlgorithmName`*`)`
- The important method:
  - `key = kf.generateSecret(KeySpec ks)`
- Returns a key of the required type
  - But requires a `KeySpec` as argument

Creating a KeySpec for PBKDF2

- You need to identify the correct class from the Sun JCA documentation
  - This may take some trial-and-error
- Required arguments:
  - The passphrase to be converted into a key
  - Salt
  - Number of iterations
  - Size of key required (128, 192 or 256 bits for AES)
- Once you have created the KeySpec, you can pass it as an argument to the KeyFactory

The Cipher Class

- Used to do the actual encryption/decryption
- See https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#Cipher for a full description
- The process is essentially:
  - Create the cipher, specifying the *transformation* (algorithm, block mode, padding, e.g. “AES/ECB/NoPadding” – see Standard Algorithm Name document)
  - Initialize it, specifying encrypt or decrypt, the keyspec and ivspec
  - Repeatedly call one of the `update()` methods
  - For the last block, call `doFinal()`
- Notice: the Cipher implementation may well buffer data that you pass to it
  - Pass it less than a full block, it will wait for more
  - Pass it more than a full block, it will encrypt and return as much as it can
- The `doFinal()` call will deal with any final plaintext and perform padding as specified

Padding

- When using AES-128 to encrypt an arbitrarily-long file, you'll need to specify a padding method
- As discussed earlier, the most common technique is PKCS#7
  - PKCS#5 is technically inapplicable to AES, but a lot of crypto libraries still use the name PKCS#5 to refer to it. Check the Standard Algorithm Name document and see what you can discover

Digests

- The MessageDigest class can compute a digest over its input, using any of many popular algorithms (SHA-2, SHA-3, etc.)

- See https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/security/MessageDigest.html

- The process is essentially:
  - Create the digest, which initializes it (use `reset()` to reinitialize)
  - Repeatedly call one of the `update()` methods to feed it input
  - Once it has eaten everything, call one of the `digest()` methods

Signatures

- The Signature class makes use of the functionality of message digests and public cryptography
  - E,g, `SHA256withRSA`, etc.
- To generate a signature, the Signature object is
  - Created with the `getInstance()` generator method
  - Initialized with the `initSign()` method, passing a `PrivateKey`
  - Fed data with one of the `update()` methods
  - And the signature (a byte array) obtained with the `sign()` method
- To verify a signature, the Signature object is
  - Created with the `getInstance()` generator method
  - Initialized with the `initVerify()` method, passing a PublicKey
  - Fed data with one of the `update()` methods
  - And a Boolean obtained with the `verify()` method, which is passed the signature

Random Numbers

- The JCA provides the SecureRandom class
  - As opposed to java.util.Random, which is a linear congruential generator
- Used to fill an array of bytes with random values
  - `public void nextBytes(byte[] bytes)`
- Useful for generating
  - Session keys
  - Initialization vectors
- Hint: In programs which use lots of random numbers in a short time, never create more than one random number generator
  - Use the Singleton design pattern and make multiple calls on the singleton
  - Otherwise, your multiple generators may share the same seed!

Clearing Cryptographic Material

- When finished, don’t forget to destroy any keys or other cryptographic material in memory by overwriting them
- Do not rely on Java finalizers – the garbage collector may not call them at program termination
- (This may be overkill, but is good – and recommended practice)