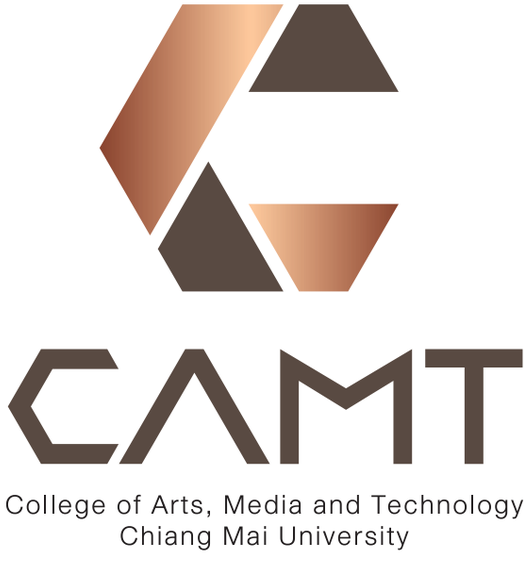
SE 233 Advanced Programming

Chapter II Exception handling



Lect Passakorn Phannachitta, D.Eng.

passakorn.p@cmu.ac.th

College of Arts, Media and Technology

Chiang Mai University, Chiangmai, Thailand

Agenda

- Catching exceptions
- Try-catch block
- The finally block
- Throwing exceptions
- Exceptions handling
- Categorization

Overview

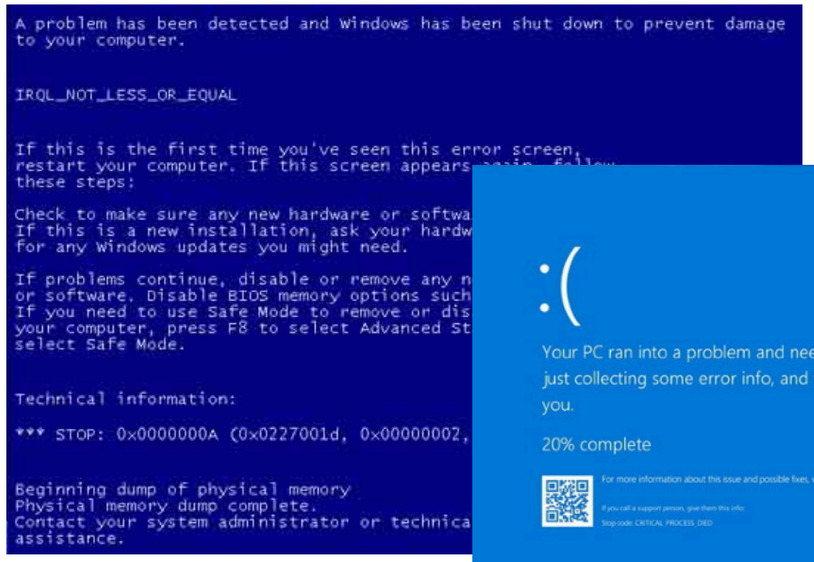
- Anyone heard of a classic adage:

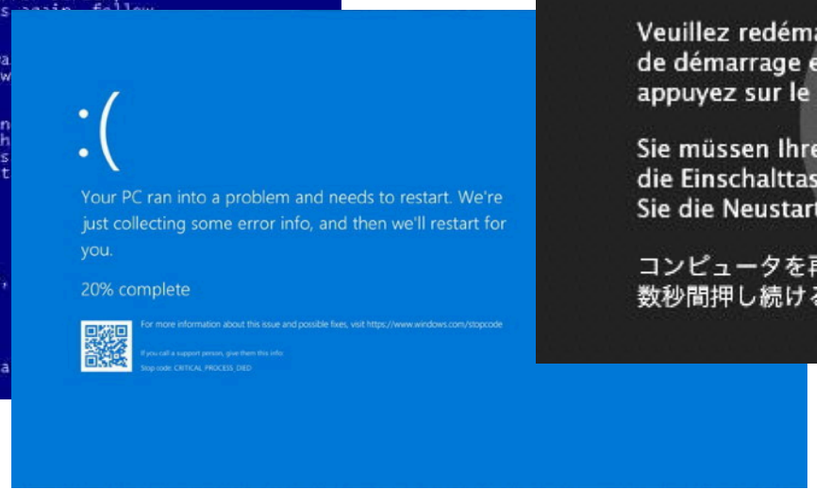
  “Anything that can go wrong will go wrong”

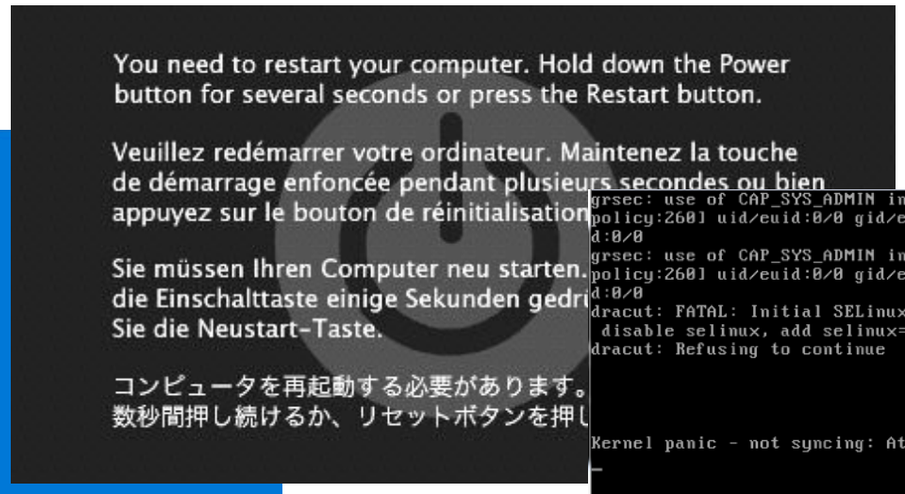
What can go wrong? E.g.,

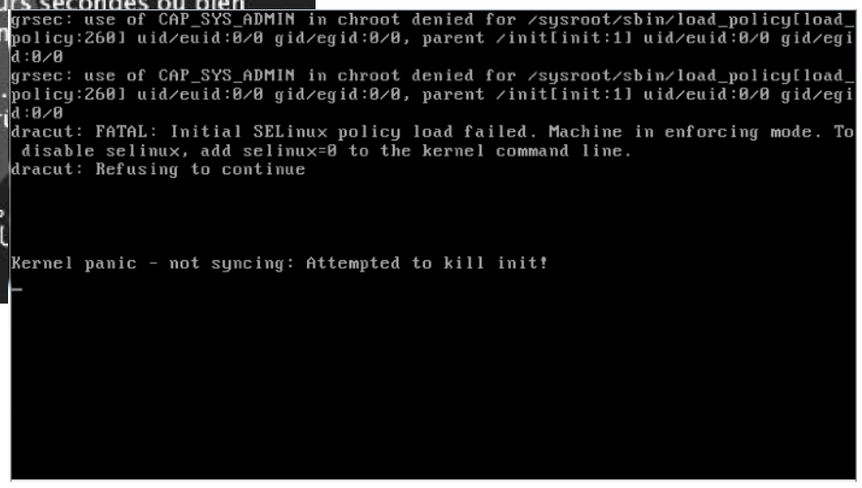
- Network cut off,
- Missing files,
- Attempting to compute what could not be computed,
- Etc.,

If the system does not know what to do









Exception

- The concept of Exception provides us resources to deal with the possible crash.
- Resources are guidelines, methodology, and other resources necessarily to plan for the possible problems systematically
- In short, Exception is all about thinking of alternative path of the normal flow can not be complete.

Principle

- The main principle of Exception is to attempt to detect the unusual problem when it occurs and handle it in the most sensible way possible.
- The successful detection and handling of the problem will make the pro- gram continue gracefully, rather than nothing but crash.

For example

- Encountering a “divide by zero”.
- System should not guess what it should do.
  - So, it just play safe by simply let the application crash and take no additional responsibility
- We are the one that have to tell it what it should do if this “divide by zero” is encountered — by coding.

Of course, we have to instruct it before the problem is happened.

Usually seen exceptions

- I/O error
  - Can generate bad consequence to the system
  - Error that occur with the hardware (I/O, memory) access
    - Require the exception handler
- Runtime error
  - Error during runtime, which may not effect the system
  - Hence exception handler is an optional

For example

Class IOException

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.io.IOException

All Implemented Interfaces:

`Serializable`

Direct Known Subclasses:

`ChangedCharSetException, CharacterCodingException, CharConversionException, ClosedChannelException, EOFException, FileLockInterruptionException, FileNotFoundException, FilerException, FileSystemException, HttpRetryException, IIOException, InterruptedByTimeoutException, InterruptedIOException, InvalidPropertiesFormatException, JMXProviderException, JMXServerErrorException, MalformedURLException, ObjectStreamException, ProtocolException, RemoteException, SaslException, SocketException, SSLException, SyncFailedException, UnknownHostException, UnknownServiceException, UnsupportedDataTypeException, UnsupportedEncodingException, UserPrincipalNotFoundException, UTFDataFormatException, ZipException`

The entire family

Direct Known Subclasses:

AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CertificateException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalClassFormatException, InterruptedException, IntrospectionException, InvalidApplicationException, InvalidMidiDataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException, IOException, JAXBException, JMException, KeySelectorException, LambdaConversionException, LastOwnerException, LineUnavailableException, MarshalException, MidiUnavailableException, MimeTypeParseException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NotBoundException, NotOwnerException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, ReflectiveOperationException, RefreshFailedException, RemarshalException, RuntimeException, SAXException, ScriptException, ServerNotActiveException, SOAPException, SQLException, TimeoutException, TooManyListenersException, TransformerException, TransformException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URIReferenceException, URISyntaxException, UserException, XAException, XMLParseException, XMLSignatureException, XMLStreamException, XPathException

Catching exceptions

- The logic of Exception can be viewed as a particular form of an if-statement:

```
+ 1 | /*
+ 2 |    if(is no problem occurs) {
+ 3 |       normal flow
+ 4 |    } else {
+ 5 |       exception flow
+ 6 |    }
+ 7 | */
```

Catching exceptions

- The logic of Exception can be viewed as a particular form of an if-statement:

```
+ 8  public class Launcher {
+ 9     public static void main(String[] args) {
+10        try {
+11           normalFlow();
+12        } catch (Exception e) {
+13           exceptionFlow();
+14        }
+15     }
+16  }
```

Anatomy

- If ~/dummy.text does not exist!

```
+  1 public class FileReader {
+  2    public String retrieve(string path) { ... }
+  3 }
+  4 public class Document {
+  5    public void init() {
+  6       FileReader fr = new FileReader();
+  7       String content = fr.retrieve('~/dummy.txt');
+  8     }
+  9 }
+ 10 public class Launcher {
+ 11    public static void main(String[] args) {
+ 12       Document d = new Document();
+ 13       d.init();
+ 14    }
   15 }
```

Try-catch block

- If ~/dummy.text does not exist!

```
  1 | public class FileReader {
- 2 |    public String retrieve(string path) { ... }
+ 3 |    public String retrieve(string path) {
+ 4 |    try {
+ 5 |       ...
+ 6 |    } catch (FileNotFoundException e) {
+ 7 |       System.out.println("File not found!");
+ 8 |    }
  9 | }
 10 | public class Document {
 11 |    public void init() {
 12 |       FileReader fr = new FileReader();
 13 |       String content = fr.retrieve('~/dummy.txt');
 14 |    }
 15 | }
 16 | public class Test {
 17 |    public static void main(String[] args) {
 18 |    Document d = new Document();
 19 |    d.init(); }
 20 | }
```

Try-catch block

- The FileNotFoundException is planned for its handling by catching it.
- Two steps are considered: try and catch.
  - Try: If we know or see that the result of any operations in our program is likely to cause an exception, we surround the code with a try block.
  - Catch: It is where we code one or more exception handlers. Note that a handler is implemented in out catch block.

Finally block

```
   1 public class FileReader {
   2    public String retrieve(string path) {
   3    try {
   4       ...
   5    } catch (FileNotFoundException e) {
   6       System.out.println("File not found!");
-  7    }
+  8    } finally {
+  9       System.out.println("Read file done!");
+ 10    }
  11 }
  12 public class Document {
  13    public void init() {
  14       FileReader fr = new FileReader();
  15       String content = fr.retrieve('~/dummy.txt');
  16    }
  17 }
  18 public class Test {
  19    public static void main(String[] args) {
  20    Document d = new Document();
  21    d.init(); }
  22 }
```

Finally block

- The default exit for a try-catch block.
- It is optional.
- If defined, it always be executed no matter if the exception occurs or not.
- Particularly useful for implementing a clean up code that guarantee its execution.
  - Closing file or freeing memory

If we don’t want to catch exceptions

- Throwing exception is the option.
- It is to delay the catching by passing the exception back to the caller method.
- And leave the caller method to handle it
- Of course, we have to instead catch the exception in the caller method. Otherwise, we will face the same problem as that we do not handle the exception.

Throwing exception

```
   1  public class FileReader {
-  2     public String retrieve(string path) {
+  3     public String retrieve(string path) throws FileNotFoundException {
-  4        try {
              ...
-  6        } catch (FileNotFoundException e) {
-  7           System.out.println("File not found!");
-  8        } finally {
-  9           System.out.println("Read file done!");
- 10        }
- 11     }
  12  }
  13  public class Document {
         public void init() {
            FileReader fr = new FileReader();
+ 16        try {
  17           String content = fr.retrieve('~/dummy.txt');
+ 18        } catch (FileNotFoundException e) {
+ 19           System.out.println("File not found!");
+ 20        } finally {
+ 21           System.out.println("Read file done!");
+ 22        }
  23     }
  24  }
```

Categorization

- Checked and Unchecked Exceptions
- Checked exceptions: a Java compiler can help us check whether there are missing handlers for their handling
- Unchecked exceptions: a compiler is unable to know whether an unchecked exception will be raised or not because it does not always happen — Runtime error.

Exception handling

- In programming design, we have to plan ahead for all the possible types of exception and decide where and how to handle each of which in the most sensible way.
- Principles
  - If the Exception is more likely to be caused by user error, then it must be better to give the user a chance to correct it;
  - Take a sensible default action;
  - Display a popup dialog to facilitate a quick action that can prevent any further exceptions due to misunderstanding.

Benefits of Exception handling

- We will not forget to prepare the flow for common failure.
- The predefined Exception objects in Java provide us a high-level summary of Exception, including a stack trace for every time an Exception is encountered.
- We will develop a kind of consistent logical thought where we will more frequently think about separating normal flows from exceptional flows

Summary

- Exception handing is to plan ahead for preventing the application to crash due to that it does not know how to handle a specific problem that it should not simply guess what to do next.
- We can choose to throw an exception just to delay its catching.

Bonus topic — Lambda and Stream

- Agenda
  - Lambda
  - Stream

What is Lambda

- In λ calculus
  - Everything is a function
  - It treats functions "**anonymously**" without giving them explicit names, e.g., square_sum(x,y) = $x^2 + y^2$ can be rewritten as (x,y) -> $x^2 + y^2$

Anonymous class

```
this.watch.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        AllEventHandlers.onWatch(currency.getShortCode());
    }
});
```

```
imageViewList[i].setOnDragDone(new EventHandler<DragEvent>() {
    @Override
    public void handle(DragEvent event) {
        onEquipDone(event);
    }
});
```

Ways to use it

```
imageViewList[i].setOnDragDone(new EventHandler<DragEvent>() {
    @Override
    public void handle(DragEvent event) {
        onEquipDone(event);
    }
});
```

```
imageViewList[i].setOnDragDone(event -> onEquipDone(event));
```

Lambda in Java

- Anonymous class has been available since Java 1.1
- An explicit implementation of Lambdas has been available in Java 8

Example, before Java 1.1

```
public class StringLengthComparator implements Comparator {
    private StringLengthComparator() { }

    public static final StringLengthComparator INSTANCE = new StringLengthComparator();
    public int compare(Object o1, Object o2) {
        String s1 = (String) o1, s2 = (String) o2;
        return s1.length() - s2.length();
    }
}
```

```
String[] arr = {"ABC","AB","A","ABCD"};
Arrays.sort(arr, StringLengthComparator.INSTANCE);
System.out.println(Arrays.toString(arr));
```

In Java 1.1, Anonymous class allows us to do

```
String[] arr = {"ABC","AB","A","ABCD"};
Arrays.sort(arr, new Comparator() {
    public int compare(Object o1, Object o2) {
        String s1 = (String) o1, s2 = (String) o2;
        return s1.length() - s2.length();
    }
});

System.out.println(Arrays.toString(arr));
```

This is also called Class Instance Creation Expression (CICE)

Lambda!

```
String[] arr = {"ABC","AB","A","ABCD"};
Arrays.sort(arr,(s1, s2) -> s1.length() - s2.length());
System.out.println(Arrays.toString(arr));
```

- In terms of functionality, it is simply the same anonymous function as that of the Java 1.1 CICE

Lambda syntax

Syntax	Example
parameter -> expression	x->x+1
parameter -> block	s -> { System.out.println(s); }
(parameters) -> expression	(x, y) -> x+y
(parameters) -> block	(s1, s2) -> { System.out.println(s1 + "," + s2); }
(parameters declares) -> expression	(double x, double y) -> Math.sqrt(x*x + y*y)
(parameters declares) -> block	(List<?> list) ->{ Arrays.sort(list); }

Ways to use it

- In most cases, event handling where we need to create an anonymous class and @override the handle function can simply implement as lambda, for example:

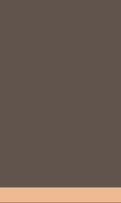
```
imageViewList[i].setOnDragDone(new EventHandler<DragEvent>() {
    @Override
    public void handle(DragEvent event) {
        onEquipDone(event);
    }
});
```

If we click the *suggestion* lightbulb

Ways to use it

- It is usable for many interfaces from old libraries, such as Runnable, Comparator, etc.

```
imageViewList[i].setOnDragDone(new EventHandler<DragEvent>() {
    @Override
    public void handle(DragEvent event) {
        onEquipDone(event);
    }
});
```



```
imageViewList[i].setOnDragDone(event -> onEquipDone(event));
```

Shorter code, and usually has better readability

Some suggestions on its usage

1. Keep Lambda expressions short and self-explained

e.g., don't do —

```
onClick( () -> foofoofoofoo(param) );
```

2. Avoid block of code in Lambda's body

e.g., don't do —

```
Foo foo = parameter -> {

    String result = "blah blah blah";

    //many lines of code
    return result;
};
```

Some suggestions on its usage

3. Avoid specifying parameter types

e.g., don't do —

```
(String a, String b) -> a.toUpperCase() + b.toLowerCase();
```

4. Avoid parenthesis around single parameter

e.g., don't do —

```
(a) -> a.toLowerCase();
```

5. Avoid return statement and braces

e.g., don't do —

```
x -> {return x.toLowerCase()};
```

What is Stream

- A bunch of data objects, typically from a collection, array, or input device, for bulk data processing
- Example
  - One Stream generator +
  - Zero or more intermediate stream operations +
  - One terminal stream operation

Stream — example

```
List<String> stringList = Arrays.asList("Hello", "World!", "How", "Are", "You");
stringList.stream().forEach(a->System.out.println(a));
```

```
IntStream.range(0, 10).forEach(a->System.out.println(a));
```

```
"Hello world!".chars().forEach(x -> System.out.print((char) x));
```

Applicable for object ref types, int, long, and double

Intermediate stream operations

- Mapping, filtering, checking for duplicates, and sorting are the commonly used operations
- E.g., mapping

```
List<String> stringList = Arrays.asList("Hello", "World!", "How", "Are", "You");
List<String> mappedList = stringList.stream()
        .map(s -> s.substring(0,1))
        .collect(Collectors.toList());
System.out.println(mappedList);
```

Intermediate stream operations

- Mapping, filtering, checking for duplicates, and sorting are the commonly used operations
- E.g., filtering

```
List<String> stringList = Arrays.asList("Hello", "World!", "How", "Are", "You");
List<String> filteredList = stringList.stream()
        .filter(s -> s.length() > 3)
        .collect(Collectors.toList());
System.out.println(filteredList);
```

Intermediate stream operations

- Mapping, filtering, checking for duplicates, and sorting are the commonly used operations
- E.g., checking for duplicates

```
List<String> stringList = Arrays.asList("Hello", "World!", "How", "Are", "You");
List<String> dupsRemoved = stringList.stream()
        .map(s -> s.substring(0,1))
        .distinct()
        .collect(Collectors.toList());
System.out.println(dupsRemoved);
```

Intermediate stream operations

- Mapping, filtering, checking for duplicates, and sorting are the commonly used operations
- E.g., sorting

```
List<String> stringList = Arrays.asList("Hello", "World!", "How", "Are", "You");
List<String> sortedList = stringList.stream()
        .map(s -> s.substring(0,1))
        .sorted()
        .collect(Collectors.toList());
System.out.println(sortedList);
```

What about performance ? — An example

- Simple for loop

```
long tStart = System.currentTimeMillis();
double sum = 0;

for (long j = 1; j < 1e8; j++) sum += Math.log10(j);

long tEnd = System.currentTimeMillis();
long tDelta = tEnd - tStart;
double elapsedSeconds = tDelta / 1000.0;

System.out.println("sum: " + sum + "\ntime: " + elapsedSeconds + " seconds");
```

- Average taken from 3 runs

sum: 7.565705482087342e8

time: 3.31 seconds

What about performance ? — An example

- Equivalent stream computation

```
long tStart = System.currentTimeMillis();
double sum = 0;

sum = LongStream.range(1, (long) 1e8).mapToDouble(x -> Math.log10(x)).sum();

long tEnd = System.currentTimeMillis();
long tDelta = tEnd - tStart;
double elapsedSeconds = tDelta / 1000.0;

System.out.println("sum: " + sum + "\ntime: " + elapsedSeconds + " seconds");
```

- Average taken from 3 runs

sum: 7.565705482087342e8

time: 3.78 seconds

Seems slower!

Parallelizing is simple

- Equivalent parallel computation

```
long tStart = System.currentTimeMillis();
double sum = 0;

sum = LongStream.range(1, (long) 1e8).parallel().mapToDouble(x -> Math.log10(x)).sum();

long tEnd = System.currentTimeMillis();
long tDelta = tEnd - tStart;
double elapsedSeconds = tDelta / 1000.0;

System.out.println("sum: " + sum + "\ntime: " + elapsedSeconds + " seconds");
```

- Average taken from 3 runs

sum: 7.565705482087342e8

time: 1.72 seconds

Much faster!

When to use a parallel stream

- When operations are independent
- Operations are applied to many elements of efficiently splittable data structures

By the way, always measure the performance before and after adding .parallel() to your stream

Stream Interfaces

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {

   // Intermediate Operations
  Stream<T> filter(Predicate<T>);
  <R> Stream<R> map(Function<T, R>);
  IntStream mapToInt(ToIntFunction<T>);
  LongStream mapToLong(ToLongFunction<T>);
  DoubleStream mapToDouble(ToDoubleFunction<T>);
  <R> Stream<R> flatMap(Function<T, Stream<R>>); IntStream flatMapToInt(Function<T,
  IntStream>); LongStream flatMapToLong(Function<T, LongStream>); DoubleStream
  flatMapToDouble(Function<T, DoubleStream>); Stream<T> distinct();
  Stream<T> sorted();
  Stream<T> sorted(Comparator<T>);
  Stream<T> peek(Consumer<T>);
  Stream<T> limit(long);
  Stream<T> skip(long);
```

Stream Interfaces

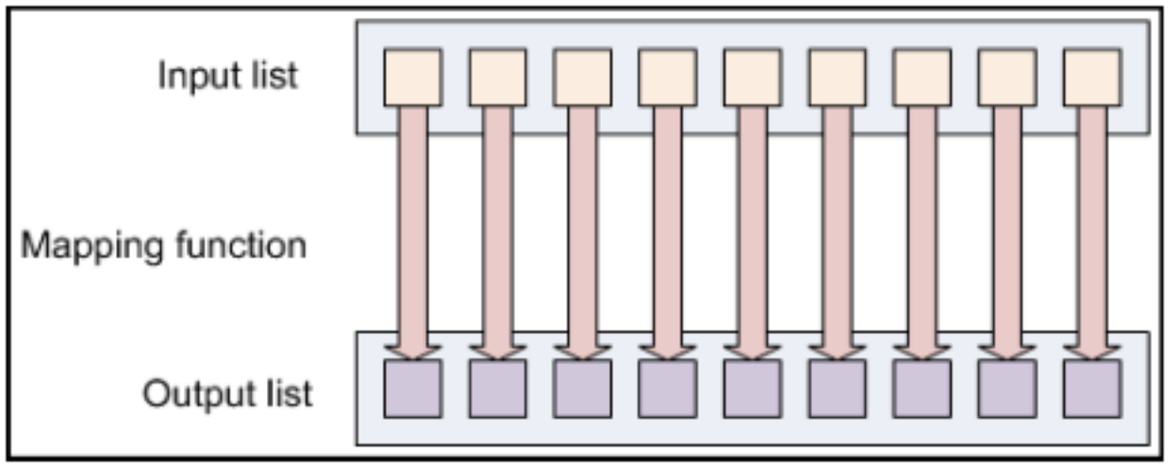
```
// Terminal Operations

void forEach(Consumer<T>); // Ordered only for sequential streams void
forEachOrdered(Consumer<T>); // Ordered if encounter order exists Object[] toArray();
<A> A[] toArray(IntFunction<A[]> arrayAllocator);

T reduce(T, BinaryOperator<T>);
Optional<T> reduce(BinaryOperator<T>);
<U> U reduce(U, BiFunction<U, T, U>, BinaryOperator<U>);
<R, A> R collect(Collector<T, A, R>); // Mutable Reduction Operation <R> R
collect(Supplier<R>, BiConsumer<R, T>, BiConsumer<R, R>); Optional<T> min(Comparator<T>);
Optional<T> max(Comparator<T>);
long count();
boolean anyMatch(Predicate<T>);
boolean allMatch(Predicate<T>);
boolean noneMatch(Predicate<T>);
Optional<T> findFirst();
Optional<T> findAny();
```

Map and reduce — A powerful tools

- Widely used in many modern technology associating with BigData, parallel computing, and distributed computing
  - e.g., Hadoop, MongoDB

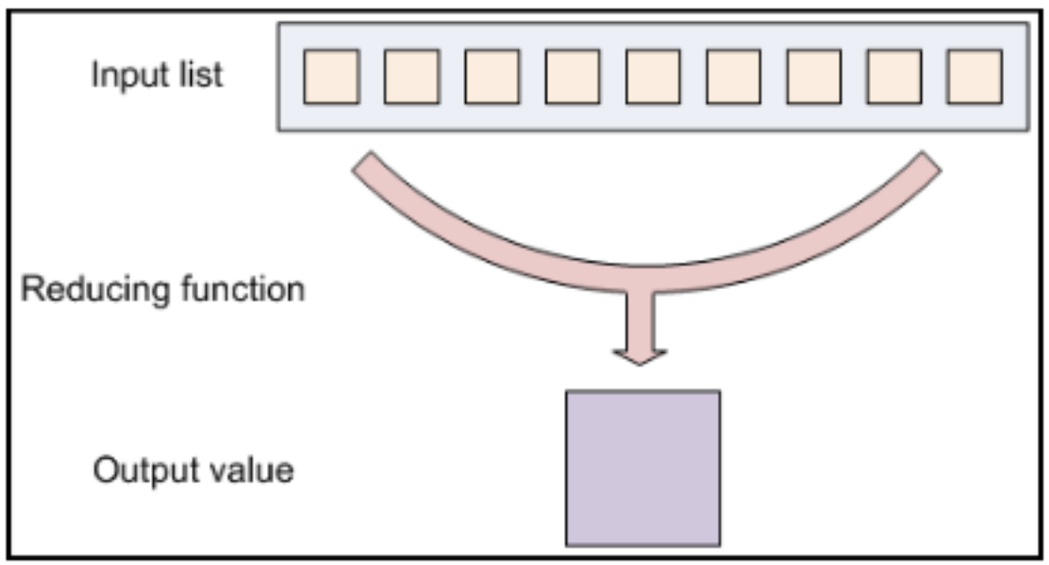
Map and Reduce

- map(f, x[0...n-1])
  - Apply the function f to each element of x[]



Map and Reduce

- reduce(f, x[0...n-1])
  - Repeatedly apply function f to partial group of items in x, then recursively replacing x until one single result remains.



Map and Reduce

- Conventionally, the reduce function are usually replaced by a terminal operation which indicates grouping

```
String text = “... ... ...”

Pattern pattern = Pattern.compile(" ");

Map<String, Integer> wordCount = pattern.splitAsStream(text)
        .map(word -> word.replaceAll("[^a-zA-Z]", "").toLowerCase().trim())
        .filter(word -> word.length() > 0)
        .map(word -> new AbstractMap.SimpleEntry<>(word, 1))
        .collect(toMap(e -> e.getKey(), e -> e.getValue(), (v1, v2) -> v1 + v2));

wordCount.forEach((k, v) -> System.out.println(String.format("%s ==>> %d", k, v)));
```

Usage Example

- From the program we are familiar with

```
public static List<Integer> primeNumbers(int n) {
    List<Integer> primeNumbers = new LinkedList<>();
    if (n >= 2) {
        primeNumbers.add(2);
    }
    for (int i = 3; i <= n; i += 2) {
        if (isPrime(i)) {
            primeNumbers.add(i);
        }
    }
    return primeNumbers;
}

private static boolean isPrime(int number) {
    for (int i = 2; i*i < number; i++) {
        if (number % i == 0) {
            return false;
        }
    }
    return true;
}
```

How to Make a Stream Version

```
public static List<Integer> primeNumbers(int n) {
    return IntStream.rangeClosed(2, n)
            .filter(x -> isPrime(x)).boxed()
            .collect(Collectors.toList());
}

private static boolean isPrime(int number) {
    return IntStream.rangeClosed(2, (int) (Math.sqrt(number)))
            .noneMatch(x -> number % x == 0);
}
```

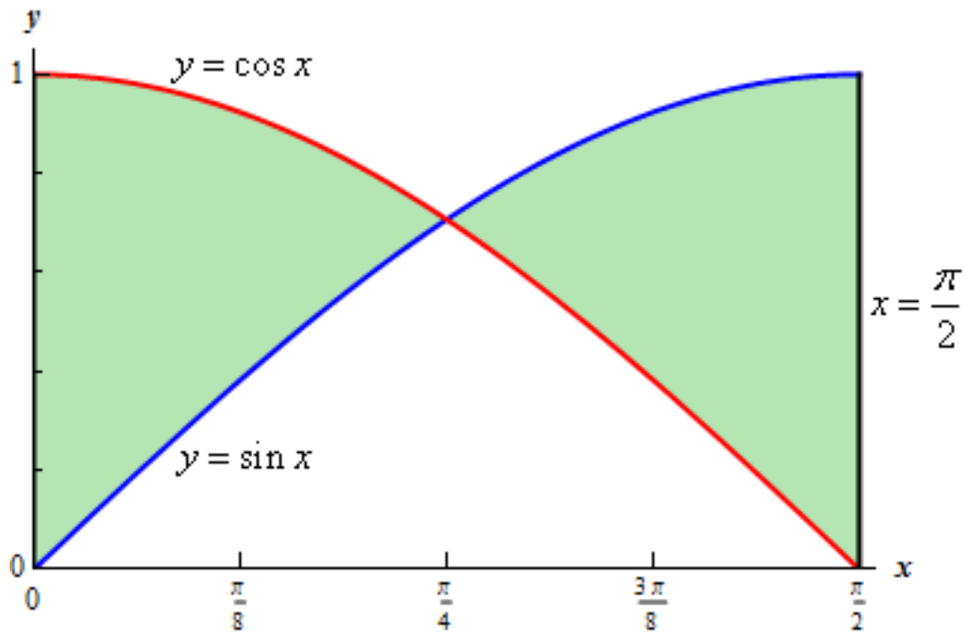
- Let’s discuss about code readability

What about the performance

- Generate prime numbers between 2 -> 1e7
  - Brute force: 22.623 seconds
  - Stream: 33.369 seconds
  - Parallel stream (only the main loop): 14.013 seconds

Practice

- Find the area under the curve (highlighted in green)



How to Approach it

- For x in [0, Pi/4] the green area is the area under cos(x) - sin(x)
- For x in [Pi/4, Pi/2] the green area is the area under sin(x) - cos(x)
- One of the most comprehensible approaches is to divide the range into chunks, then, compute the area for all the chunks, and finally, merge sum them all

A Workable Solution

```
public interface DoubleFunction {
    public double f(double x);
}

public static double AreaUnderCurve( DoubleFunction upper,
                                      DoubleFunction lower,
                                      double start,
                                      double end,
                                      int chunk) {
    double piece = (end - start)/chunk;

    return LongStream.range(0, chunk)
            .parallel()
            .mapToDouble(x -> upper.f(start+x*piece)*piece - lower.f(start+x*piece)*piece)
            .sum();
}
```

```
double area1 = AreaUnderCurve(x->Math.cos(x), x->Math.sin(x),0,Math.PI/4,(int)1e6);
double area2 = AreaUnderCurve(x->Math.sin(x), x->Math.cos(x),Math.PI/4,Math.PI/2,(int)1e6);

System.out.println(area1+area2);
```

Question