Exception Handling

- Exception handling
  - Exception
    - Indication of problem during execution
      - E.g., divide by zero
  - Chained exceptions

1

Exception-Handling Overview

- Uses of exception handling
  - Process exceptions from program components
  - Handle exceptions in a uniform manner in large projects
  - Remove error-handling code from “main line” of execution
- A method detects an error and throws an exception
  - Exception handler processes the error
  - Uncaught exceptions yield adverse effects
    - Might terminate program execution

2

Exception-Handling Overview

- Code that could generate errors put in try blocks
  - Code for error handling enclosed in a catch clause
  - The **finally** clause always executes
- Termination model of exception handling
  - The block in which the exception occurs expires
- throws clause specifies exceptions method throws
- Divide by Zero
  - Common programming mistake
  - Throws ArithmeticException

3

程式錯誤的分類

程式的錯誤可以依照性質分成三種：

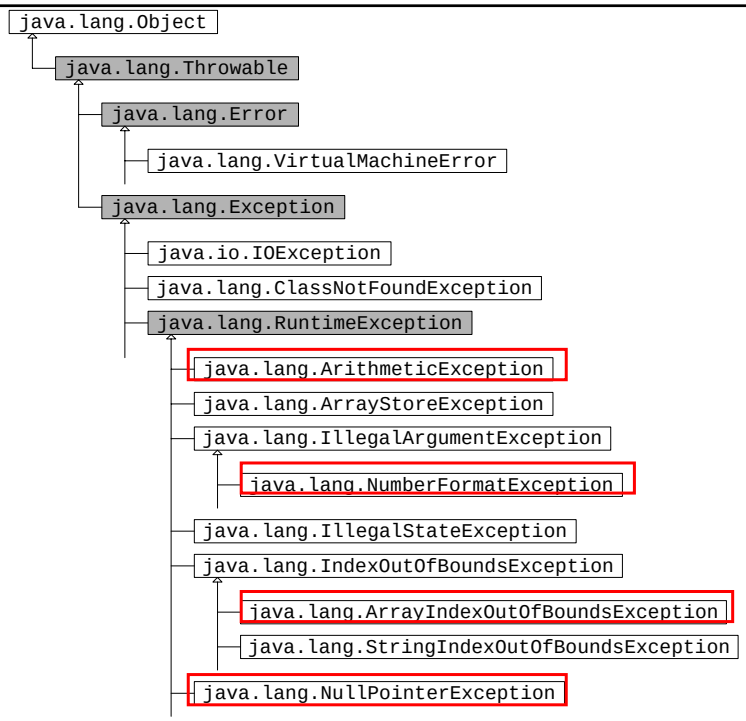
- 編譯期錯誤：編譯時所發生的錯誤，經常是語法錯。
- 執行期錯誤：在編譯程式的過程中不容易發現，常在程式執行的過程中發生的錯誤。
- 邏輯錯誤：程式設計者在設計程式時，所發生的邏輯上的問題。

4

Java例外的繼承關係

- 錯誤或例外都是java.lang.**Throwable**的延伸類別
- **Error**及其延伸類別：其為嚴重的錯誤，通常捕捉到也無法處理，因此也很少去捕捉。
- **Exception**及其延伸類別（不包含RuntimeException及其延伸類別）：這類例外在一般情況很可能發生，大多屬於環境問題。
- **RuntimeException**及其延伸類別：此類例外為程式的執行期錯誤，例如，某數除以0、陣列索引值超出範圍等。

5



6

RuntimeException

- 常見的RuntimeException（Unchecked Exception）

執行期例外	說明
ArithmeticException	數學運算時的例外。例如：某數除以0。
ArrayIndexOutOfBoundsException	陣列索引值超出範圍。
NegativeArraySizeException	陣列的大小為負數。
NullPointerException	物件參照為null，並使用物件成員時所產生的例外。
NumberFormatException	數值格式不符所產生的例外。


```
27
28       // set up label and inputField2; register listener
29       container.add( new JLabel( "Enter denominator and press Enter ",
30          SwingConstants.RIGHT ) );
31       inputField2 = new JTextField();
32       container.add( inputField2 );
33       inputField2.addActionListener( this );
34
35       // set up label and outputField
36       container.add( new JLabel( "RESULT ", SwingConstants.RIGHT ) );
37       outputField = new JTextField();
38       container.add( outputField );
39
40       setSize( 425, 100 );
41       setVisible( true );
42
43    } // end DivideByZeroTest constructor
44
45    // process GUI events
46    public void actionPerformed( ActionEvent event )
47    {
48       outputField.setText( "" );   // clear outputField
49
50       // read two numbers and calculate quotient
51       try {
52          number1 = Integer.parseInt( inputField1.getText() );
53          number2 = Integer.parseInt( inputField2.getText() );
```

Outline

DivideByZeroTest.java

Line 51

Lines 52-53

```
54
55          result = quotient( number1, number2 );
56          outputField.setText( String.valueOf( result ) );
57       }
58
59       // process improperly formatted input
60       catch ( NumberFormatException numberFormatException ) {
61          JOptionPane.showMessageDialog( this,
62             "You must enter two integers", "Invalid Number Format",
63             JOptionPane.ERROR_MESSAGE );
64       }
65
66       // process attempts to divide by zero
67       catch ( ArithmeticException arithmeticException ) {
68          JOptionPane.showMessageDialog( this,
69             arithmeticException.toString(), "Arithmetic Exception",
70             JOptionPane.ERROR_MESSAGE );
71       }
72
73    } // end method actionPerformed
74
75    // demonstrates throwing an exception when a divide-by-zero occurs
76    public int quotient( int numerator, int denominator )
77       throws ArithmeticException
78    {
79       return numerator / denominator;
80    }
```

Outline

DivideByZeroTest.java

Line 55

Line 60

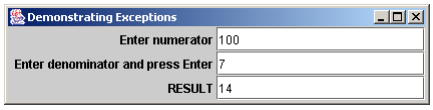
Line 67

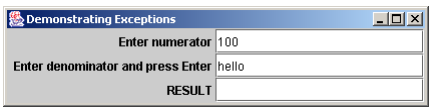
Line 77

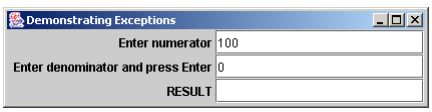
```
81
82    public static void main( String args[] )
83    {
84       DivideByZeroTest application = new DivideByZeroTest();
85       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
86    }
87
88 } // end class DivideByZeroTest
```

Outline

DivideByZeroTest.java





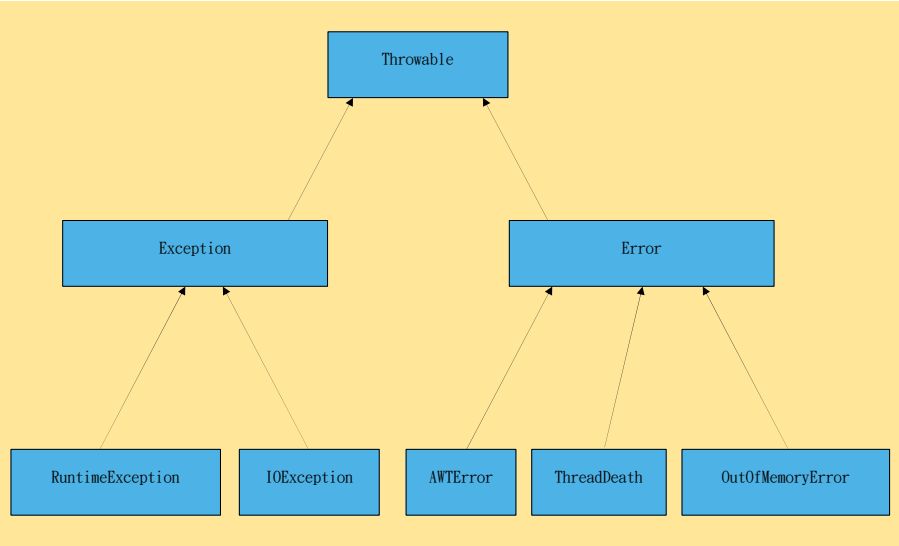


Java Exception Hierarchy

- Superclass Throwable
  - Subclass Exception
    - Exceptional situations
    - Should be caught by program
  - Subclass Error
    - Typically not caught by program
- Checked exceptions
  - Catch or declare
- Unchecked exceptions

11

Inheritance hierarchy for class Throwable



12

Rethrowing an Exception

- Rethrow exception if catch cannot handle it

13

finally Clause

- Resource leak
  - Caused when resources are not released by a program
- The finally block
  - Appears after catch blocks
  - Always executes
  - Use to release resources

14

```
1  // Fig. 15.3: UsingExceptions.java
2  // Demonstration of the try-catch-finally exception handling mechanism.
3  public class UsingExceptions {
4
5     public static void main( String args[] )
6     {
7        try {
8           throwException(); // call method throwException
9        }
10
11       // catch Exceptions thrown by method throwException
12       catch ( Exception exception ) {
13          System.err.println( "Exception handled in main" );
14       }
15
16       doesNotThrowException();
17    }
18
19    // demonstrate try/catch/finally
20    public static void throwException() throws Exception
21    {
22       // throw an exception and immediately catch it
23       try {
24          System.out.println( "Method throwException" );
25          throw new Exception();  // generate exception
26       }
```

Outline

UsingExceptions.java

```
27
28       // catch exception thrown in try block
29       catch ( Exception exception ) {
30          System.err.println(
31             "Exception handled in method throwException" );
32          throw exception;  // rethrow for further processing
33
34          // any code here would not be reached
35       }
36
37       // this block executes regardless of what occurs in try/catch
38       finally {
39          System.err.println( "Finally executed in throwException" );
40       }
41
42       // any code here would not be reached
43
44    } // end method throwException
45
46    // demonstrate finally when no exception occurs
47    public static void doesNotThrowException()
48    {
49       // try block does not throw an exception
50       try {
51          System.out.println( "Method doesNotThrowException" );
52       }
```

Outline

UsingExceptions.java

Line 32

Lines 38-40

```
53
54       // catch does not execute, because no exception thrown
55       catch ( Exception exception ) {
56          System.err.println( exception );
57       }
58
59       // this clause executes regardless of what occurs in try/catch
60       finally {
61          System.err.println(
62             "Finally executed in doesNotThrowException" );
63       }
64
65       System.out.println( "End of method doesNotThrowException" );
66
67    } // end method doesNotThrowException
68
69 } // end class UsingExceptions
```

```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException
```

Outline

UsingExceptions.java

Lines 60-63

try-catch敘述

- try-catch敘述用以捕捉並處理例外

```
try{
        //例外測試區
}
catch(例外型別   例外名){
        //例外處理區
}
```

- 只要是Throwable或其延伸類別之物件，都可以使用try-catch敘述捕捉。

18

多個catch區塊

try-catch敘述可以使用多個catch區塊

```
try {
      //例外測試區
}
catch(例外型別1  例外名1){
      //例外處理區1
}
catch(例外型別2  例外名2){
      //例外處理區2
}
...

catch(例外型別N  例外名N){
      //例外處理區N
}
```

19

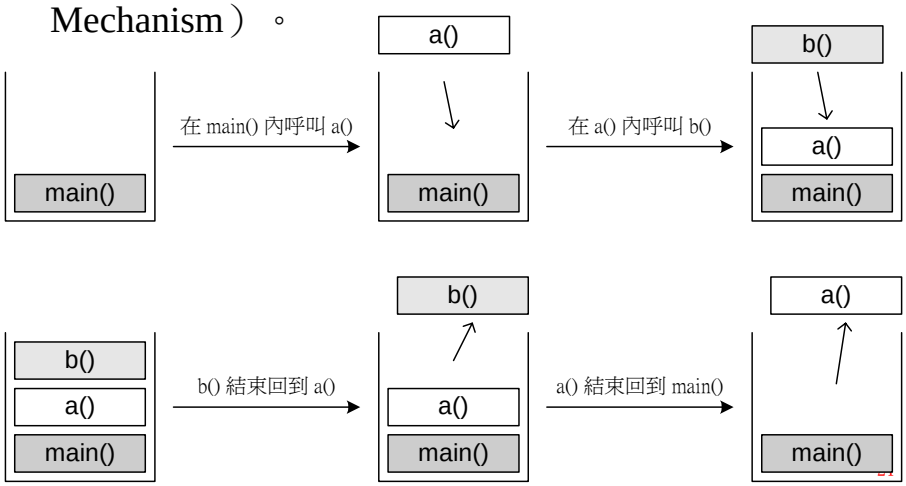
多個catch區塊

- 頂多只會有一個catch區塊內的敘述被執行。
- 特化的例外型別需放在較前的catch敘述內。

20

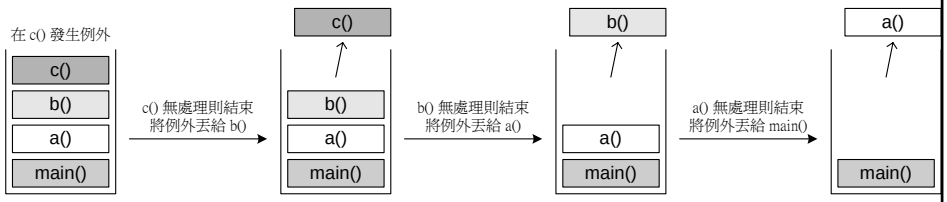
Call Stack機制

- 方法之間的呼叫是依循呼叫堆疊機制（Call Stack Mechanism）。



Call Stack機制

- 例外與呼叫堆疊機制



finally區塊

- 不論有沒有例外發生，finally區塊內的敘述皆會執行。

```
try
{
        //例外測試區
}
catch(例外型別  例外名)
{
        //例外處理區
}
finally
{
        //鐵定執行區
}
```

23

finally區塊

- finally區塊不會執行或不會完全執行：
  - 有其它例外在finally區塊中發生。
  - 在try或catch區塊使用System.exit()離開程式。
  - 未執行至finally區塊，執行緒即進入結束狀態（dead state）。

24

Stack Unwinding

- Exception not caught in scope
  - Method terminates
  - Stack unwinding occurs
  - Another attempt to catch exception

25

```
1  // Fig. 15.4: UsingExceptions.java
2  // Demonstration of stack unwinding.
3  public class UsingExceptions {
4
5     public static void main( String args[] )
6     {
7        // call throwException to demonstrate stack unwinding
8        try {
9           throwException();
10       }
11
12       // catch exception thrown in throwException
13       catch ( Exception exception ) {
14          System.err.println( "Exception handled in main" );
15       }
16    }
17
18    // throwException throws exception that is not caught in this method
19    public static void throwException() throws Exception
20    {
21       // throw an exception and catch it in main
22       try {
23          System.out.println( "Method throwException" );
24          throw new Exception();      // generate exception
25       }
26
```

Outline

UsingExceptions.java

Line 9

Line 13

Line 19

Line 24

```
27       // catch is incorrect type, so Exception is not caught
28       catch ( RuntimeException runtimeException ) {
29          System.err.println(
30             "Exception handled in method throwException" );
31       }
32
33       // finally clause always executes
34       finally {
35          System.err.println( "Finally is always executed" );
36       }
37
38    } // end method throwException
39
40 } // end class UsingExceptions
```

```
Method throwException
Finally is always executed
Exception handled in main
```

Outline

UsingExceptions.java

printStackTrace, getStackTrace **and** getMessage

- Throwable class
  - Method printStackTrace
    - Prints method call stack
  - Method getStackTrace
    - Obtains stack-trace information
  - Method getMessage
    - Returns descriptive string

28

以 throw丟出例外物件

使用throw敘述在程式中丟出例外物件

```
throw 例外物件;
```

- 丟出一包含訊息的自訂例外物件：

```
throw new RuntimeException("除數為零");
```

```
throw new Exception("找不到檔案");
```

- throw所丟出的例外物件同樣可以使用try-catch敘述處理。

29

throws關鍵字

- **throws**關鍵字使用於方法或建構子定義的標頭，用來指出例外發生時，由方法（或建構子）丟出的例外型別。
- throws之後可以接多個例外型別名，表示方法執行過程中可能丟出屬於這些例外型別的物件。
- 方法不可以使用throw丟出不包含在throws宣告的例外型別之物件

30

throws關鍵字

- 發生Checked Exception時須遵守**處理或宣告丟出法則**（The Handle or Declare Rule）：不是以try-catch-finally敘述處理就是以throws宣告丟出。

31

```
1  // Fig. 15.5: UsingExceptions.java
2  // Demonstrating getMessage and printStackTrace from class Exception.
3  public class UsingExceptions {
4
5     public static void main( String args[] )
6     {
7        try {
8           method1(); // call method1
9        }
10
11       // catch Exceptions thrown from method1
12       catch ( Exception exception ) {
13          System.err.println( exception.getMessage() + "\n" );
14          exception.printStackTrace();
15
16          // obtain the stack-trace information
17          StackTraceElement[] traceElements = exception.getStackTrace();
18
19          System.out.println(  "\nStack trace from getStackTrace:" );
20          System.out.println( "Class\t\tFile\t\t\tLine\tMethod" );
21
22          // loop through traceElements to get exception description
23          for ( int i = 0; i < traceElements.length; i++ ) {
24             StackTraceElement currentElement = traceElements[ i ];
25             System.out.print( currentElement.getClassName() + "\t" );
26             System.out.print( currentElement.getFileName() + "\t" );
```

Outline

UsingExceptions.java

Line 8

Lines 13-14

Lines 25-26

```
27             System.out.print( currentElement.getLineNumber() + "\t" );
28             System.out.print( currentElement.getMethodName() + "\n" );
29
30          } // end for statement
31
32       } // end catch
33
34    } // end method main
35
36    // call method2; throw exceptions back to main
37    public static void method1() throws Exception
38    {
39       method2();
40    }
41
42    // call method3; throw exceptions back to method1
43    public static void method2() throws Exception
44    {
45       method3();
46    }
47
48    // throw Exception back to method2
49    public static void method3() throws Exception
50    {
51       throw new Exception( "Exception thrown in method3" );
52    }
```

Outline

UsingExceptions.java

Lines 27-28

Line 37

Line 39

Line 43

Line 45

Line 49

Line 51

```
53
54 } // end class Using Exceptions
```

```
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:51)
        at UsingExceptions.method2(UsingExceptions.java:45)
        at UsingExceptions.method1(UsingExceptions.java:39)
        at UsingExceptions.main(UsingExceptions.java:8)

Stack trace from getStackTrace:
Class           File                 Line    Method
UsingExceptions UsingExceptions.java 51      method3
UsingExceptions UsingExceptions.java 45      method2
UsingExceptions UsingExceptions.java 39      method1
UsingExceptions UsingExceptions.java 8       main
```

Outline

UsingExceptions.java

Chained Exceptions

- Wraps existing exception in a new exception
  - enables exception to maintain complete stack-trace

35

```
1  // Fig. 15.6: UsingChainedExceptions.java
2  // Demonstrating chained exceptions.
3  public class UsingChainedExceptions {
4
5     public static void main( String args[] )
6     {
7        try {
8           method1(); // call method1
9        }
10
11       // catch Exceptions thrown from method1
12       catch ( Exception exception ) {
13          exception.printStackTrace();
14       }
15    }
16
17    // call method2; throw exceptions back to main
18    public static void method1() throws Exception
19    {
20       try {
21          method2(); // call method2
22       }
23
24       // catch Exception thrown from method2
25       catch ( Exception exception ) {
26          throw new Exception( "Exception thrown in method1", exception );
```

Outline

UsingChainedExceptions.java

Line 8

Lines 12-14

Line 18

Line 21

Line 26

```
27       }
28    }
29
30    // call method3; throw exceptions back to method1
31    public static void method2() throws Exception
32    {
33       try {
34          method3(); // call method3
35       }
36
37       // catch Exception thrown from method3
38       catch ( Exception exception ) {
39          throw new Exception( "Exception thrown in method2", exception );
40       }
41    }
42
43    // throw Exception back to method2
44    public static void method3() throws Exception
45    {
46       throw new Exception( "Exception thrown in method3" );
47    }
48
49 } // end class Using Exceptions
```

Outline

UsingChainedExceptions.java

Line 31

Line 34

Line 39

Line 46

```
java.lang.Exception: Exception thrown in method1
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:26)
        at UsingChainedExceptions.main(UsingChainedExceptions.java:8)
Caused by: java.lang.Exception: Exception thrown in method2
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:39)
        at UsingChainedExceptions.method1(UsingChainedExceptions.java:21)
        ... 1 more
Caused by: java.lang.Exception: Exception thrown in method3
        at UsingChainedExceptions.method3(UsingChainedExceptions.java:46)
        at UsingChainedExceptions.method2(UsingChainedExceptions.java:34)
        ... 2 more
```

Outline

UsingChainedExceptions.java

Declaring New Exception Types

- Extend existing exception class

Constructors and Exception Handling

- Throw exception if constructor causes error

39

Assertion

- 斷言（Assertion）是J2SE 1.4新增的功能，其目的是為了方便程式的除錯。
- 斷言的用處是在於**去除邏輯上的錯誤**。

40

斷言的宣告

- 第一種斷言的宣告語法：

```
assert 布林運算式;
```

  - **布林運算式**的運算結果必須為**布林值**，否則在程式編譯時會發生錯誤。
  - 布林運算式的結果若為false，則斷言將丟出一個沒有任何錯誤訊息的AssertionError物件。
  - AssertionError類別屬於java.lang套件，為Error類別的子類別。

41

斷言的宣告

- 第二種斷言的宣告語法：

```
assert 布林運算式 : 訊息運算式;
```

  - 布林運算式為false時，會丟出AssertionError物件。
  - AssertionError物件所包含的錯誤訊息是由訊息運算式所構成。
  - **訊息運算式**結果通常為String物件，其運算結果不可以是void（回傳型別為void的方法）。

42

啟動斷言功能

- 斷言是J2SE 1.4新增的功能，所以程式裡包含斷言敘述時，必須使用1.4以上的編譯器版本編譯。
- 使用「-source」參數指明編譯出來的bytecode為1.4版：

```
C:\>javac  -source  1.4  FileName.java
```

43

啟動斷言功能

- 執行時，使用「-enableassertions」或「-ea」參數，指明啟動斷言功能：

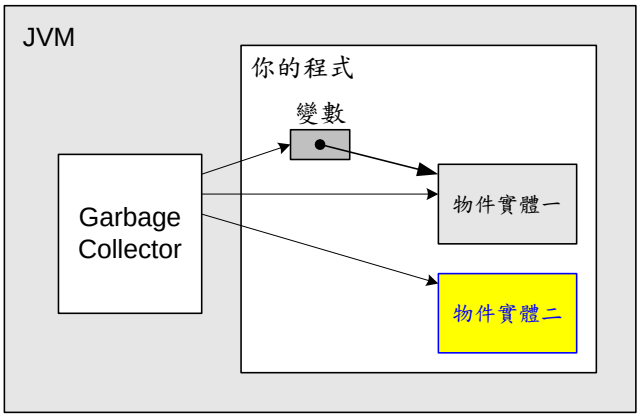
```
C:\>java  -ea  FileName
```

- 沒有使用「-enableassertions」或「-ea」啟動斷言功能，則程式中的斷言敘述會被解譯器忽略。
- 一般只有在程式測試時（軟體開發過程），才會開啟斷言功能。

44

資源回收者

- Garbage Collector在程式一開始執行時就會監視（trace）每一個變數和物件實體，當物件實體不被參考時，此物件實體即成為Unreachable Object。



45

資源回收者

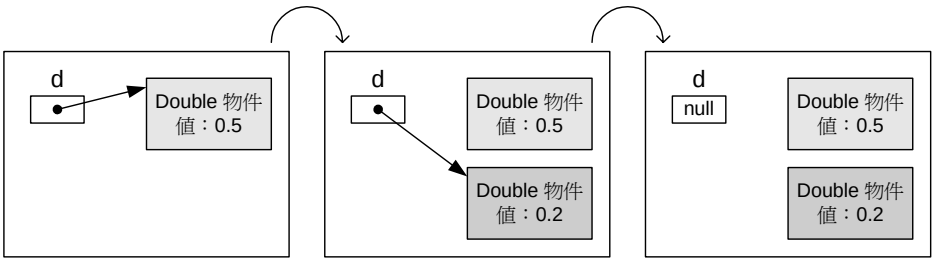
- 當物件不再被**參照變數**（reference variable）參考（refer）時，它就會被認為是不再使用的garbage。

```
Double d = new Double(0.5);
d = new Double(0.2);
d = null;
```

46

資源回收者

```
Double d = new Double(0.5);
d = new Double(0.2);
d = null;
```



47

資源回收者

- Unreachable Object佔用的資源並不一定馬上就會被釋放。
- 資源釋放和執行環境（可用記憶體多寡，CPU繁忙程度等）有關，程式設計者無法預測或決定資源何時被釋放。
- 執行資源回收的動作只能被建議，程式設計者無法強迫（force）資源回收的執行。

48

finalize()方法

- Garbage Collection機制在終結物件並釋放記憶體之前，會先呼叫物件中的finalize()方法。
- finalize()方法定義於Object類別，不過為空方法，主要是讓延伸類別覆蓋之用。

```
protected void finalize()throws Throwable
```

- finalize()方法不提供多載的功能。而且finalize()不接受傳入參數及回傳值，所以一律以void定義，並且無參數。

49