
⭐ الصفحة 2 — Topics (الموضوعات)

1. مقدمة في معالجة الاستثناءات
2. معالجة الاستثناءات في C++
3. معالجة الاستثناءات في Java
4. معالجة الاستثناءات في Python وRuby
5. مقدمة في معالجة الأحداث Event Handling
6. معالجة الأحداث في Java
7. معالجة الأحداث في #C

⭐ الصفحة 3 — Introduction to Exception Handling

مقدمة في معالجة الاستثناءات

- في اللغات التي لا تحتوي على معالجة للاستثناءات:
  عند حدوث استثناء، تنتقل السيطرة إلى نظام التشغيل، ويظهر خطأ ويتم إنهاء البرنامج.
- أما في اللغات التي تدعم معالجة الاستثناءات:
  يمكن للبرنامج **التقاط بعض الأخطاء والتعامل معها** بحيث يمكنه الاستمرار بدل الإنهاء.

⭐ الصفحة 4 — Basic Concepts

المفاهيم الأساسية

- العديد من لغات البرمجة تسمح بالتقاط أخطاء الإدخال والإخراج (مثل الوصول لنهاية الملف EOF).
- هو حدث غير عادي يمكن اكتشافه من قبل الهاردوير أو السوفتوير، الاستثناء Exception ويتطلب معالجة خاصة.
- **معالجة الاستثناء** العملية الخاصة التي تُنفّذ بعد حدوث الاستثناء تسمّى **Exception Handling**.
- **المعالج** كود المعالجة يُسمّى **Exception Handler**.

⭐ الصفحة 5 — Exception Handling Alternatives

بدائل معالجة الاستثناءات

- عندما يحدث الحدث المرتبط به (Raised) يتم رفع الاستثناء.
- اللغة التي لا تدعم معالجة الاستثناءات يمكنها مع ذلك:
  - تعريف الاستثناءات
  - اكتشافها
  - رفعها
  - معالجتها (تعريف المستخدم)

البدائل المتاحة:

1. استخدام **قيمة الإرجاع** لتحديد حالة استرجاع الدالة (نجاح/فشل).
2. تمرير **دالة لمعالجة الأخطاء** إلى جميع الدوال الأخرى.

⭐ الصفحة 6 — Advantages of Built-in Exception Handling

مزايا وجود معالجة استثناءات مدمجة في اللغة

- كتابة كود اكتشاف الأخطاء يدويًا أمر متعب ويجعل الكود مزدحمًا.
- معالجة الاستثناءات تشجع المبرمج على التفكير في العديد من الأخطاء المحتملة أثناء تنفيذ البرنامج.

⭐ الصفحة 9 — Exception Handling Control Flow

تدفق التحكم في معالجة الاستثناءات

((الصفحة تحتوي رسم فقط
يبين الشكل كيف تنتقل السيطرة من الجزء الذي يرفع الاستثناء إلى المعالج المناسب ثم العودة إلى التدفق الطبيعي للبرنامج.

⭐ الصفحة 10 — Exception Handling in C++

معالجة الاستثناءات في C++

- أُضيفت ميزة معالجة الاستثناءات إلى C++ سنة 1990.
- تصميمها مستوحى من لغات: CLU ،Ada ،ML.
- تعتمد على ثلاث كلمات رئيسية:
  1. **try** — لتحديد الكود الذي قد يُسبب استثناء.
  2. **throw** — لرفع/إطلاق الاستثناء.
  3. **catch** — لمعالجة الخطأ.

⭐ الصفحة 11 — Exception Handling in C++

معالجة الاستثناءات في C++

الصياغة العامة:

```
try {
   // الكود المتوقع أن يسبب استثناء
   throw exception;
}
catch (formal parameter) {
   // كود المعالجة
}
```

⭐ الصفحة 13 — The catch Function

دالة المعالج catch

- اسم **catch** مستخدم لجميع معالجات الأخطاء — وهو اسم محمّل *Overloaded*، لذا **يجب** **مختلفًا في كل catch أن يكون نوع المعامل formal parameter**.
- المعامل ليس بالضرورة أن يكون له متغير.
  - يمكن أن يكون مجرد **اسم نوع** فقط للتمييز بين المعالجات.
- يمكن استخدام المعامل لنقل معلومات إلى المعالج (مثل رسالة خطأ).

⭐ الصفحة 14 — The catch Function (متابعة)

- catch يمكن أن يكون المعامل داخل **ثلاث نقاط (...)** هو
  → في هذه الحالة يقوم المعالج بالتعامل مع **جميع الاستثناءات التي لم يتم التعامل معها سابقًا**.

⭐ الصفحة 15 — Throwing Exceptions

إطلاق الاستثناءات (Throw)

- يتم رفع الاستثناءات باستخدام:

```
throw [expression];
```

- الأقواس تدل على أن وجود التعبير اختياري.
- يمكن استخدام throw بدون معامل داخل catch فقط لإعادة إطلاق نفس الاستثناء.
- التعبير (expression) هو ما يحدد أي **نوع** catch سيتم استدعاؤه.

⭐ الصفحة 16 — Throwing Exceptions

(تكملة الشريحة السابقة – تحتوي مثالاً فقط)

توضّح هذه الصفحة مثالًا يبيّن كيفية استخدام:

```
throw;
```

الـ catch الحالية لإعادة رفع الاستثناء مرة أخرى إلى معالج آخر خارج.

⭐ الصفحة 17 — Unhandled Exceptions

الاستثناءات غير المُعالجة

- إذا لم يتم التعامل مع الاستثناء داخل الدالة، فسيتم **تمريره إلى الدالة التي استدعت هذه الدالة**.
- يستمر تمرير الاستثناء حتى يصل إلى الدالة **main**.
- إذا لم يتم العثور على أي catch. لمعالجته، يتم استدعاء **المعالج الافتراضي** عادةً يؤدي هذا إلى إنهاء البرنامج →.

⭐ الصفحة 18 — Continuation

استمرار التنفيذ بعد المعالجة

- بعد انتهاء تنفيذ catch، يعود التحكم إلى **أول سطر بعد آخر catch**.
- خيارات التصميم في C++:
  - المعالج الافتراضي "unexpected" يقوم بإنهاء البرنامج.
  - يمكن للمستخدم إعادة تعريف unexpected.
  - يمكن للدوال throw باستخدام **تحديد الاستثناءات التي قد ترفعها**.

⭐ الصفحة 19 — Evaluation

تقييم تصميم معالجة الاستثناءات في C++

- لا توجد استثناءات جاهزة مسبقًا (predefined) في C++.
- من غير المعتاد ألا تكون الاستثناءات **مسماة**، وأن الأخطاء التي يكتشفها الجهاز أو النظام لا يمكن التعامل معها.
- ربط نوع المعامل في catch بنوع الـ throw يجعل الكود **أقل وضوحًا**.

⭐ الصفحة 20 — Exception Handling in Java

معالجة الاستثناءات في Java

- تعتمد C++ على مبادئ البرمجة الكينونية لكن أكثر توافقًا مع OOP.
- جميع الاستثناءات هي **كائنات (Objects)** من فئات ترث من الفئة **Throwable**.

⭐ الصفحة 21 — Classes of Exceptions

فئات الاستثناءات في Java

تنقسم فئة **Throwable** إلى فئتين رئيسيتين:

1) Error

- تُرمى من قبل **Java** نفسه **مفسّر**.
- أمثلة: أخطاء الذاكرة مثل **heap overflow**.
- **لا يجب** على البرامج التعامل معها لأنها أخطاء نظام.

2) Exception

- الفئة الأساسية لمعظم الاستثناءات.
- الاستثناءات التي يعرّفها المستخدم عادة ترث من هذه الفئة.
- لها فئتان جاهزتان مهمتان:
    - **IOException**
    - **RuntimeException**
        - مثل:
            - ArrayIndexOutOfBoundsException
            - NullPointerException

⭐ الصفحة 22 — Java Exception Handlers

معالجات الاستثناءات في Java

- ولكن مع اختلافات مثل C++:
  - يجب أن يحتوي اسم معامل (لا يمكن تركه بدون اسم) كل **catch**.
  - يجب أن يكون نوع المعامل **من الفئة أو أحد فروعها Throwable**.

رفع الاستثناء يتم باستخدام:

```
throw new MyException();
```

ملاحظة: غالبًا يتم إنشاء كائن الاستثناء أثناء عملية الإطلاق.

⭐ الصفحة 23 — Binding Exceptions to Handlers

ربط الاستثناءات بالمعالجات في Java

- ربط الاستثناء بالمعالج أبسط من C++:
  - يتم ربط الاستثناء مع **أول** catch يكون معاملها من نفس فئة الاستثناء أو من **أسلافه** (Superclass).
- يمكن للمعالج:
  - معالجة الاستثناء ثم **إطلاقه مجددًا** باستخدام:
  - `throw;`
  - أو إطلاق **استثناء مختلف**.

⭐ الصفحة 24 — Continuation

استمرار البحث عن المعالج

- إذا لم يتم العثور على معالج داخل كتلة try الحالية:
  - يستمر البحث في **أقرب try محيط بها** (enclosing try).
- إذا لم يوجد معالج داخل الدالة:
  - يتم **تمرير الاستثناء (propagate)** إلى الدالة التي استدعتها.
- إذا لم يوجد معالج حتى الوصول إلى main:
  - يتم إنهاء البرنامج.

⭐ الصفحة 25 — Continuation

التأكد من التقاط جميع الاستثناءات

- يمكن إضافة catch يلتقط **جميع الاستثناءات**:
  - وذلك باستخدام معامل من نوع **Exception**.
- يجب أن يكون هذا الـ catch هو **الأخير** في سلسلة الـ catch.

⭐ الصفحة 26 — Checked and Unchecked Exceptions

الاستثناءات المُراقَبة وغير المُراقَبة في Java

- جملة **throws** في Java تختلف عن C++.
- الاستثناءات في Java تُقسم إلى نوعين:

1) Unchecked Exceptions (غير المراقَبة)

- تشمل:
  - Error
  - RuntimeException
  - وكل الفئات المتفرعة منهم
- لا يُطلب من المبرمج التعامل معها أو التصريح بها في دالة.

2) Checked Exceptions (المراقَبة)

- وهي جميع الاستثناءات الأخرى.
- يجب على المبرمج إما:
  1. **معالجتها داخل الدالة**، أو
  2. **التصريح بها** في جملة throws.

⭐ الصفحة 27 — Checked and Unchecked Exceptions

(تكملة الشريحة السابقة – تحتوي جدولًا فقط)

الخلاصة المعروضة في الجدول:

- **Checked:** يجب التعامل معها أو التصريح بها.
- **Unchecked:** لا يجب التصريح بها أو التعامل معها.

⭐ الصفحة 28 — The finally Clause

كتلة finally في Java

- يمكن وضع finally في نهاية كتلة try لتحديد كود يتم تنفيذه في جميع الأحوال.

الصيغة العامة:

```
try {
    ...
}
catch (...) {
    ...
}
finally {
    ...
}
```

كتلة finally سواء حدث استثناء أو لم يحدث **تُنفَّذ دائمًا**.

⭐ الصفحة 29 — The finally Clause

متى يتم تنفيذ finally؟

- إذا لم يحدث استثناء ← يتم تنفيذ finally قبل الخروج من try.
- إذا حدث استثناء وتم التقاطه ← يتم تنفيذ finally بعد انتهاء catch.
- إذا حدث استثناء ولم يتم التقاطه ← يتم تنفيذ finally **قبل تمرير الاستثناء للخارج**.

⭐ الصفحة 30 — The finally Clause

مثال يوضح تشغيل finally في حالة وجود return داخل حلقة

في الكود التالي:

```
try {
    for (index = 0; index < 100; index++) {
        ...
        if (...) {
            return;
        }
    }
}
finally {
    ...
}
```

← سيتم تنفيذ كتلة finally **سواء خرجت الدالة باستخدام return أو انتهى التكرار طبيعيًا**.

⭐ الصفحة 31 — Assertions

التأكيدات (Assertions) في Java

- يُفترض أن يكون صحيحًا (Boolean) هي عبارات داخل البرنامج تعبر عن شرط منطقي أثناء التنفيذ.
- إذا كان الشرط **صحيحًا** ← **لا يحدث شيء**.
- إذا كان الشرط **خاطئًا** ← **يتم إطلاق استثناء من النوع** **AssertionError**.
- يمكن **تعطيل** أو **تشغيل** التأكيدات أثناء التشغيل بدون تعديل البرنامج أو إعادة ترجمته.

⭐ الصفحة 32 — Assertions

أشكال كتابة assert في Java

هناك شكلان رئيسيان:

1)

```
assert condition;
```

مثال:

```
assert name == null;
```

2)

```
assert condition : expression;
```

- إذا كان الشرط خاطئًا، يتم طباعة الرسالة الموجودة في expression.

مثال:

```
assert name == null : "no name";
```

⭐ الصفحة 33 — Evaluation

تقييم معالجة الاستثناءات في Java

- أنواع الاستثناءات في Java مقارنة بـ C++ **أوضح وأكثر تنظيمًا**.
- جملة **throws** في Java أفضل من C++ لأنها توضح للمبرمج الاستثناءات المتوقعة.
- كتلة **finally** مفيدة جدًا للتأكد من تنفيذ كود معيّن مهما حدث.
- Java مفسّر يرمي العديد من الاستثناءات التي يمكن للبرنامج التعامل معها.

⭐ الصفحة 34 — Exception Handling in Python

معالجة الاستثناءات في Python

- الاستثناءات في Python **كائنات** هي **(Objects)**.
- الفئة الأساسية لها هي **BaseException**.
- جميع الاستثناءات المعرّفة مسبقًا أو المعرّفة من المستخدم محصورة تحت الفئة **Exception**.

أهم الفئات الفرعية الجاهزة:

- **ArithmeticError**
  - وتشمل:
    - OverflowError
    - ZeroDivisionError
    - FloatingPointError
- **LookupError**
  - وتشمل:
    - IndexError
    - KeyError

⭐ الصفحة 35 — Exception Handling in Python

الصياغة العامة لمعالجة الاستثناءات

```
try:
    # الكود الذي قد يسبب استثناء
except Exception1:
    # معالجة الاستثناء الأول
except Exception2:
    # معالجة الاستثناء الثاني
...
else:
    # يتم تنفيذها إذا لم يحدث أي استثناء
finally:
    # يتم تنفيذها دائمًا سواء حدث استثناء أو لم يحدث
```

هل تريدين أن أكمل الصفحات **36 إلى 40** الآن؟
اكمل: اكتبي فقط.

سأكمل الآن ترجمة **الصفحات من 36 إلى 40** مع كتابة رقم الصفحة بوضوح كما طلبتِ.

⭐ الصفحة 36 — Exception Handling in Python

شرح إضافي لمعالجة الاستثناءات في Python

- بالتعامل مع الاستثناء المحدد له **وأيضًا جميع الفئات الفرعية** من except يقوم كل معالج ذلك الاستثناء.

- إذا كان نوع الاستثناء في except هو **Exception**:
  - فإنه يستطيع التعامل مع **جميع الاستثناءات الجاهزة والمخصصة**.
- إذا لم يتم التعامل مع الاستثناء داخل الكتلة الحالية:
  - يتم تمريره إلى أقرب try خارجي (propagation).
- إذا لم يوجد أي معالج:
  - يتم استدعاء المعالج الافتراضي ويُنهى البرنامج.
- جملة **raise** مشابهة لـ **throw** في C++ وJava.

⭐ الصفحة 37 — Exception Handling in Python

(الصفحة تحتوي مثالًا يوضح التنفيذ)

المثال يوضح:

- كيفية استخدام try / except / else / finally
- كيف يتم التقاط استثناء محدد
- كيف يتم تنفيذ finally دائمًا

⭐ الصفحة 38 — Exception Handling in Python

(صفحة مكملة – توضيح بالرسم)

تحتوي على رسم يوضح:

- تدفق تنفيذ try
- ماذا يحدث عند وقوع استثناء
- انتقال التنفيذ بين except وelse وfinally

⭐ الصفحة 39 — Exception Handling in Ruby

معالجة الاستثناءات في Ruby

- الاستثناءات في Ruby هي **كائنات** مثل باقي اللغات.
- Ruby تحتوي على العديد من الاستثناءات الجاهزة.
- الاستثناءات التي يمكن للمبرمج التعامل معها عادة تكون:
  - من نوع **StandardError**
  - أو من فئة مشتقة منها.
- الفئة StandardError مشتقة من **Exception**، والتي تحتوي على:
  - دالة **message**: رسالة الخطأ
  - دالة **backtrace**: تتبع مسار الاستثناء في الكود

- رفع الاستثناءات يتم باستخدام:

```
raise "bad parameter" if count == 0
```

⭐ الصفحة 40 — Exception Handling in Ruby

كيفية كتابة المعالجات في Ruby

تستخدم Ruby بناء الجمل التالي:

```
begin
   # الكود
rescue
   # المعالج
end
```

ويمكن إضافة:

- **else** → ينفذ عند عدم حدوث استثناء
- **ensure** → يشبه finally في Java (ينفذ دائمًا)

⭐ الصفحة 41 — Exception Handling in Ruby

متابعة حول معالجة الاستثناءات في Ruby

- تختلف Ruby عن بقية اللغات في نقطة مهمة:
  - يمكن **إعادة تشغيل الكود الذي تسبب في الاستثناء** مرة أخرى داخل نفس المعالج.
- يتم ذلك باستخدام الكلمة:
- `retry`
- عندما تكتب retry داخل rescue:
  - يعود التنفيذ إلى بداية كتلة begin
  - ويتم تنفيذ الكود مرة أخرى
  - وهذه ميزة غير موجودة في Java أو C++ أو Python.

⭐ الصفحة 42 — Introduction to Event Handling

مقدمة في معالجة الأحداث (Event Handling)

- الحدث Event، مثل **إشعار** بأن شيئًا ما حدث، هو:
  - الضغط على زر بالفأرة
  - الكتابة داخل مربع نص
  - تغيير اختيار زر راديو
- معالج الحدث Event Handler:

    - هو قطعة كود يتم تنفيذها **ردًا على حدوث الحدث**.
- نظام الأحداث يسمح بالبرمجة التفاعلية (GUI).

⭐ الصفحة 43 — Java Swing GUI Components

مكوّنات واجهات Java Swing

- **JTextField** مربع النص ← كائن من الفئة
- **JRadioButton** زر الراديو ← كائن من الفئة
- **JFrame** إطار التطبيق ← كائن من الفئة
- يحتوي JFrame على عدة طبقات تسمى **panes**.
- يتم وضع المكوّنات داخل **JPanel** لترتيبها وتنظيمها.
- يتم إضافة الـ panel إلى **content pane** داخل الإطار.

⭐ الصفحة 44 — Java Swing GUI Components

(محتوى تكميلي للصفحة السابقة – يوضح بالرسم كيفية ترتيب المكونات)

تتضمن الصفحة رسمًا يوضح:

- JFrame
- JPanel
- Content Pane
- كيفية دمج العناصر الرسومية معًا داخل نافذة واحدة.

⭐ الصفحة 45 — The Java Event Model

نموذج الأحداث في Java

- تفاعل المستخدم مع مكونات الواجهة (GUI) **يُنتج أحداثًا**.
- يمكن التقاط هذه الأحداث بواسطة **مستمعي الأحداث Event Listeners**.
- مكوّن الحدث Event Generator يقوم بـ:
    - عند وقوع الحدث Listener إرسال رسالة إلى الـ.
- **الواجهات Interfaces** تُستخدم لفرض شكل موحد لطريقة معالجة الحدث.
- أي فئة تريد التعامل مع حدث معيّن يجب أن:
    - **تنفّذ (implements)** الواجهة الخاصة بذلك الحدث.

⭐ الصفحة 46 — The Java Event Model (continued)

متابعة: نموذج الأحداث في Java

- فئة من الأحداث تسمى **ItemEvent**
  - تُستخدم عند:
    - الضغط على Checkbox
    - الضغط على Radio Button
    - تغيير عنصر في List
- الواجهة **ItemListener** تحتوي على الطريقة:
- `itemStateChanged()`

  وهي المسؤولة عن التعامل مع أحداث ItemEvent.

- تتم إضافة مستمع الحدث باستخدام:
- `addItemListener(...)`

⭐ الصفحة 47 — Event Handling in C#

معالجة الأحداث في لغة C#

- معالجة الأحداث في C# (وباقي لغات .NET) لطريقتها **مشابهة تمامًا** لـ Java.
- هناك طريقتان لإنشاء واجهات رسومية في .NET:
  1. **Windows Forms**
  2. **WPF – Windows Presentation Foundation**

في Windows Forms:

- يتم إنشاء واجهة C# GUI عبر **توريث الفئة Form**.
- لا نحتاج لإنشاء frame أو panel كما في Java.
- عناصر الواجهة مثل:
  - **Label** لعرض النص
  - **RadioButton** لخيارات التحديد

⭐ الصفحة 48 — Event Handling in C# (continued)

متابعة معالجة الأحداث في C#

- يتم تحديد مكان المكونات باستخدام خاصية **Location** من نوع **Point** وإعطائها كائن:

```
private RadioButton plain = new RadioButton();
plain.Location = new Point(100, 300);
plain.Text = "Plain";
controls.Add(plain);
```

- جميع معالجات الأحداث في C# (البروتوكول لها نفس الشكل):
- `void HandlerName(object sender, EventArgs e)`

⭐ الصفحة 49 — Event Handling in C# (continued)

كيف نختبر حالة زر الراديو؟

- نستخدم الخاصية:
- `plain.Checked`

  لمعرفة ما إذا كان زر الراديو مُفعّلًا.

- عندما يتغير الزر من غير مُحدد ← مُحدد
  يتم رفع حدث:
  CheckedChanged

⭐ الصفحة 50 — Event Handling in C# (continued)

تسجيل الحدث Event Registration

لتسجيل معالج حدث لزر الراديو:

```
plain.CheckedChanged += new EventHandler(rb_CheckedChanged);
```

- حيث rb_CheckedChanged هو اسم المعالج.
- يتم ربط هذا المعالج بحدث CheckedChanged للزر plain.