علوم الحاسب

COMPUTER SCIENCE

كتاب الطالب

12

المسار التكنولوجي

الفصل الدراسي الثاني
2020–2021

النسخة التجريبية

binarylogic

الوحدة 1

binarylogic

علوم الحاسب المستوى الثاني عشر

المسار التكنولوجي

كتاب الطالب / الفصل الدراسي الثاني 2020 - 2021

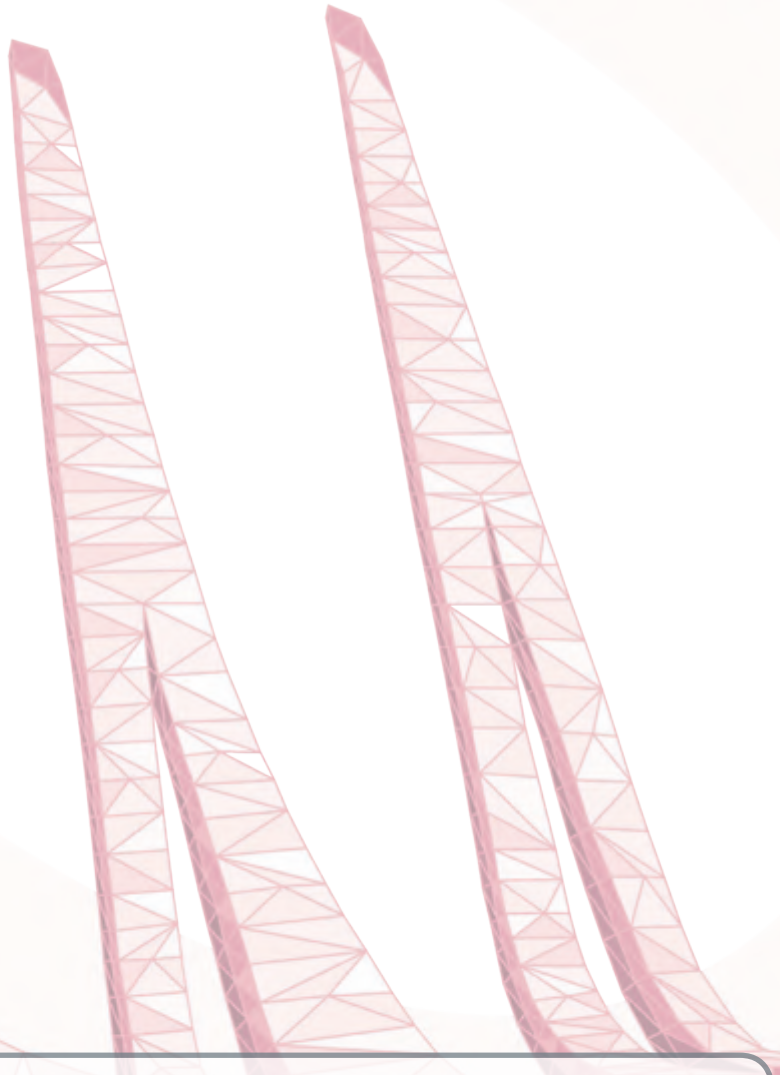
ISBN: 978-618-05-5246-1

PUBLISHED BY **MM PUBLICATIONS**

علوم الحاسب

COMPUTER SCIENCE

كتاب الطالب



الاسم .....

الشعبة .....

حضرة صاحب السمو الشيخ تميم بن حمد آل ثاني
أمير دولة قطر

النشيد الوطني

قَسَمًا بِمَنْ رَفَعَ السَّمَاءْ قَسَمًا بِمَنْ نَشَرَ الضِّيَاءْ
قَطَرٌ سَتَبْقَى حُرَّةً تَسْمُو بِرُوحِ الأَوْفِيَاءْ
سِيرُوا عَلَى نَهْجِ الأُلَى وَعَلَى ضِيَاءِ الأَنْبِيَاءْ
قَطَرٌ بِقَلْبِي سِيرَةٌ عِزٌّ وَأَمْجَادُ الإِبَاءْ
قَطَرُ الرِّجَالِ الأَوَّلِين حُمَاتُنَا يَوْمَ النِّدَاءْ
وَحَمَائِمٌ يَوْمَ السَّلاَمْ جَوَارِحٌ يَوْمَ الفِدَاءْ

مفاتيح رموز الكتاب

برامج أخرى:
قسم في نهاية الوحدة يعرض بعض الأدوات والبرامج البديلة.

المصطلحات:
قسم يوضح ما تعلمته والمفردات الجديدة التي يحتويها الدرس.

مشروع الوحدة:
نشاط في نهاية كل وحدة يدمج المهارات التي يتم تدريسها في الوحدة.

ماذا تعلمت:
قسم يركز على النقاط المهمة التي يحتاج الطلاب إلى مراجعتها.

تمرين عملي

تمرين نظري

نصيحة ذكية:
معلومات مفيدة.

كن آمنًا:
معلومات لحماية نفسك.

لمحة تاريخية:
أحداث حقيقية في الماضي.

أهلا بك!

تعال معي لنستكشف عالم تكنولوجيا المعلومات

انتقل إلى حاسوبك واتبعني!

وزارة التعليم والتعليم العالي
إدارة المناهج الدراسية ومصادر التعلم

المراجعة والتدقيق
فِرَق من:
إدارة التوجيه التربوي
الميدان التربوي

الإشراف العلمي والتربوي
إدارة المناهج الدراسية ومصادر التعلم
قسم المواد الدراسية

1. البرمجة كائنية التوجه
Object-oriented Programming

الكفايات الأساسية للمنهج التعليمي الوطني لدولة قطر

- التفكير الإبداعي والتفكير الناقد
- الكفاية اللغوية
- الكفاية العددية
- التواصل
- التعاون والمشاركة
- التقصي والبحث
- حل المشكلات

1. البرمجة كائنية التوجه
Object-oriented Programming

سنناقش في هذه الوحدة مبادئ البرمجة كائنية التوجه. سيتعلم الطلبة في هذه الوحدة كيفية إنشاء التصنيفات أو الفئات Classes والكائنات Objects وسيتعرفون على خصائصها ودوالها واستخداماتها المختلفة. سيتعلمون أيضًا المبادئ الرئيسة الخاصة بالبرمجة كائنية التوجه وكيفية توظيف معرفتهم المكتسبة في إنشاء مشروع باستخدام مزايا البرمجة كائنية التوجه.

ماذا سنتعلمْ؟

في هذه الوحِدة سنتعلم:

> المقصود بالبرمجة كائنية التوجه.

> الفرق بين البرمجية كائنية التوجه والبرمجة الإجرائية.

> ما هي الكائنات.

> ما هي الفئات.

> إنشاء الكائنات البرمجية والفئات البسيطة.

> وظيفة الإنشاء واستخدام المعامل self.

> إنشاء وتعريف الخصائص.

> كيفية الوصل إلى خصائص الكائن.

> إنشاء وتعريف الوظائف.

> استدعاء الوظائف من الكائنات.

> تعريف الوظائف الثابتة ووظائف الفئة.

> المبادئ الأربعة الرئيسة للبرمجة كائنية التوجه.

> مفهوم الوراثة في البرمجة كائنية التوجه.

> استخدم الوراثة في البرمجة بلغة Python.

> أنواع الوراثة.

> مزايا استخدام الوراثة في البرمجة.

> مفهوم التجريد في البرمجة كائنية التوجه.

> استخدام التجريد في البرمجة بلغة Python.

> مفهوم التغليف في البرمجة كائنية التوجه.

> استخدام معدلات الوصول لتعديل حالة تغليف الخصائص والوظائف في برامج Python.

> ميزات استخدام التغليف في البرمجة.

> مفهوم تعدد الأشكال في البرمجة كائنية التوجه.

> استخدم تعدد الأشكال في البرمجة بلغة Python.

> أوجه الاستفادة من مبدأ تعدد الأشكال.

> تطبيق مبادئ البرمجة كائنية التوجه لتحسين برنامج متكامل في Python.

مواضيع الوحدة

> الكائنات Objects والفئات Classes

> السمات Attributes والوظائف Methods

> مبادئ البرمجة كائنية التوجه (1)

> مبادئ البرمجة كائنية التوجه (2)

> تطبيق على مبادئ البرمجة كائنية التوجه

الأدوات

> Python

الكائنات Objects والفئات Classes

للبرمجة منهجيات ونماذج مختلفة يتم استخدامها لتنظيم المقاطع البرمجية ويطلق عليها اسم "نماذج البرمجة **Programming Paradigms**، وقد درست بتوسع نموذج "البرمجة الإجرائية" **Procedural Programming**، والذي يعتمد على استخدام الدوال **Functions** والإجراءات **Procedures**، ويوظف مفهوم التسلسل والتكرار والجمل الشرطية لكتابة البرامج، وقد نجح هذا النموذج في إنتاج العديد من البرامج، ولكنه يبقى قاصرًا أمام المشكلات البرمجية المعقدة والمتشعبة.

البرمجة كائنية التوجه Object Oriented Programming (OOP)

البرمجة كائنية التوجه هي أحد نماذج البرمجة الحديثة التي تستخدم على نطاق واسع في إنتاج البرامج الممتدة والمتشعبة.

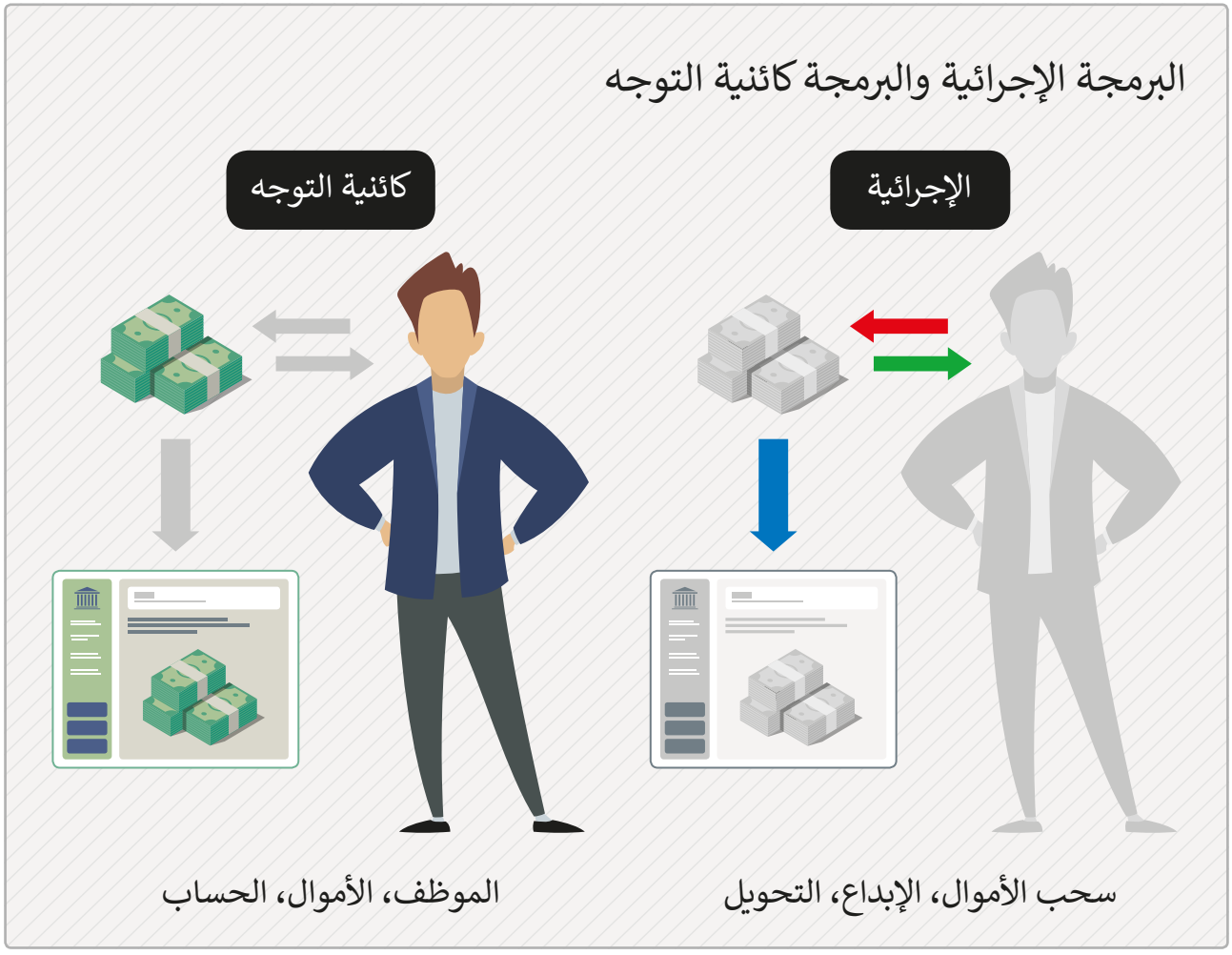
تعتبر البرمجة كائنية التوجه نموذجًا للبرمجة يعتمد على استخدام الكائنات **Objects** والفئات **Classes**. يسمح لنا هذا المفهوم بتصنيف البرامج كمجموعة من الكائنات لكل منها بيانات خاصة بها.

الفرق بين البرمجة الإجرائية والبرمجة كائنية التوجه

تنظم البرمجة الإجرائية المقاطع البرمجية في شكل إجراءات وعمليات وأوامر يتم تنفيذها بشكل متسلسل، بينما تنظم البرمجة كائنية التوجه المقاطع البرمجية إلى كائنات لكل منها متغيراته وثوابته ودواله الخاصة.

تأمل المثال في الصورة أدناه، يمكنك أن تلاحظ أنه عند كتابة المقاطع البرمجية لبرنامج مخصص للمصارف والبنوك، ستركز البرمجة الإجرائية على كتابة إجراءات خاصة للعمليات البنكية المختلفة مثل: السحب النقدي، الإيداع، تحويل الأموال، ....

بينما تعمل البرمجة كائنية التوجه إلى تنظيم المقاطع البرمجية في صورة كائنات مثل: الموظف، العميل، الأموال، الحساب البنكي، ... وهكذا.



نظريًا يمكن استخدام أي من نماذج البرمجة (إجرائية أو كائنية التوجه أو غيره) لحل مشكلة أو موقف برمجي، ولكن يتم اختيار النموذج البرمجي المناسب حسب المشكلة وخصوصياتها ومدى تعقيدها، وعليه تكون بعض النماذج أنسب من غيرها لحل بعض المشكلات والمواقف البرمجية.

تسمح بعض لغات البرمجة بتطبيق أكثر من نموذج برمجي، مثل لغة **Python** التي تتيح لك كتابة التعليمات البرمجية باستخدام نموذجي البرمجة الإجرائية وكائنية التوجه.

الكائنات Objects

الكائنات هي إحدى العناصر الرئيسة المستخدمة أثناء كتابة المقاطع البرمجية باستخدام لغة **Python**.

الكائنات **Objects** هي كيانات (**Entities**) ذات خصائص محددة **Attributes** ويمكنها تأدية وظائف معينة **Methods**.

يتشابه الكائن البرمجي (**Object**) مع الكائن الواقعي عند وصفه برمجيًا. فإذا نظرنا إلى كائن واقعي من حولنا كالسيارة مثلًا، ورغبنا بوصف هذا الكائن، فإننا سنوضح خصائص السيارة مثل نوعها (الشركة المصنعة)، وطرازها وسنة صنعها ولونها، وعدد الكيلومترات المقطوعة، وبالطبع سعر تلك السيارة. تصف جميع هذه الخصائص سيارة معينة ويطلق عليها اسم السمات **Attributes**.

بالإضافة إلى الخصائص، يوجد لكل كائن إجراءات يمكنه القيام بها تسمى وظائف Methods، والتي يمكنها تغيير حالة خصائص الكائن. فعلى سبيل المثال يمكن للسيارة أن تتسارع أو أن تلتف يمينًا أو يسارًا أو أن تتغير قيم بعض خصائصها الأخرى.

يطلق على خصائص أي كائن في مصطلحات لغة Python اسم **Attributes**، بينما يتم تسمية الإجراءات التي تتم عليه باسم الوظائف **Methods**.

السيارة = كائن

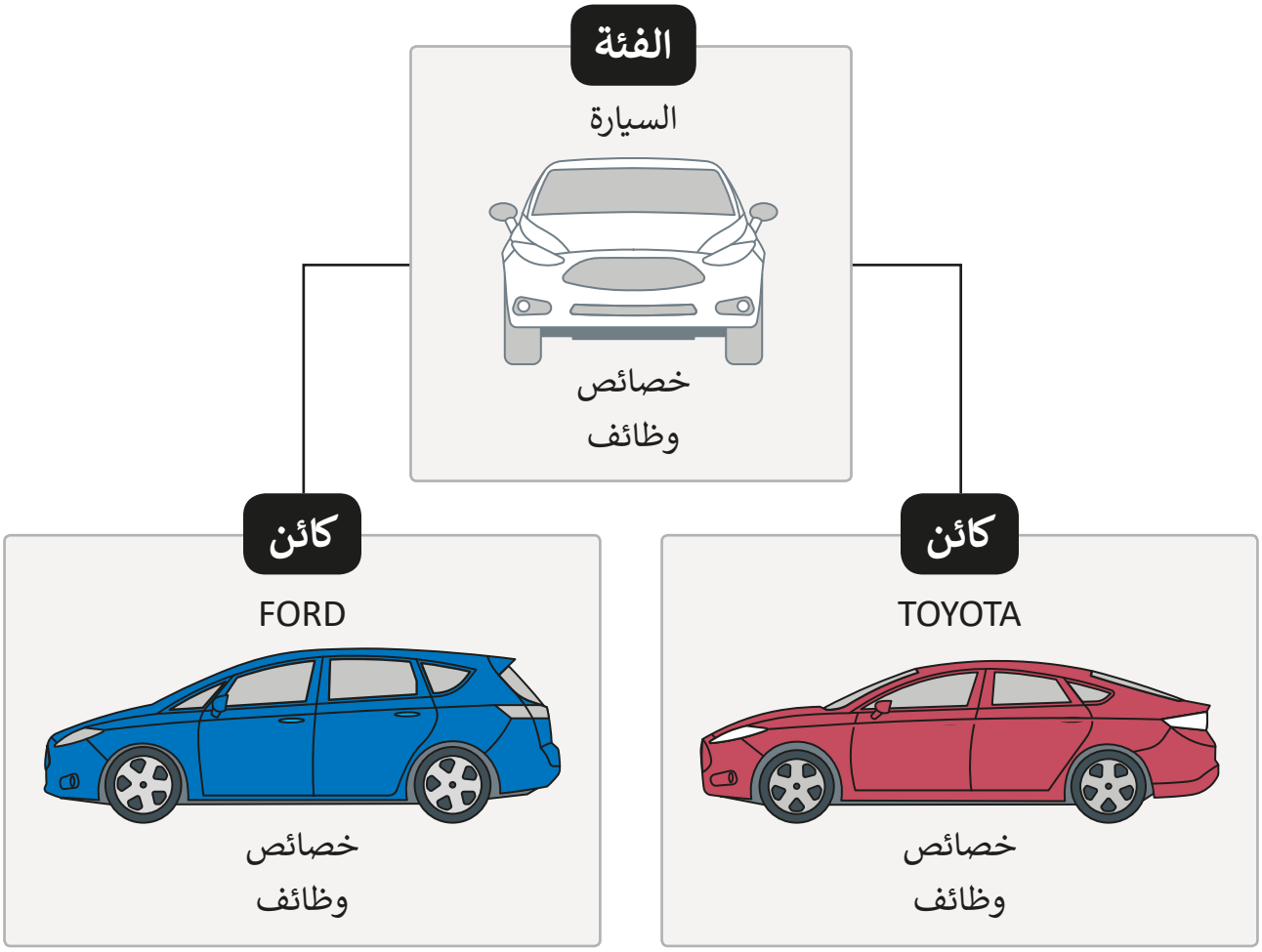
الخصائص ATTRIBUTES
نوع السيارة
الطراز
اللون
سنة الصُنع
سعر السيارة

الوظائف METHODS
تتسارع
تلتف يميناً
تلتف يساراً

الفئات Classes

الفئة (**Class**) هي تعريف مجرد يحدد خصائص ووظائف كائن ما، فهي بمثابة قالب لجميع الكائنات (**Objects**) التي تنتمي إلى تلك الفئة، حيث يكون كل كائن هو نموذج مثيل (**Instance**) لهذه الفئة.

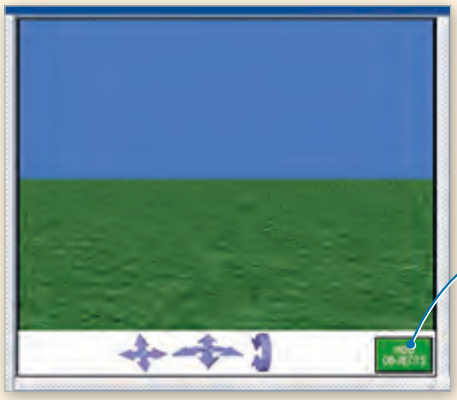
لكي نتعرف على استخدام الفئات، سنعاود النظر إلى مثال السيارة، فنحن في العادة نميز السيارة بصفتها سيارة ركاب عادية ذات أربعة أبواب، ولكن في الحقيقة فإن شركات صناعة السيارات تصنع سيارات أخرى كسيارات الدفع الرباعي والشاحنات وكذلك السيارات الرياضية. ماذا لو أردنا وصف تلك السيارات أيضًا؟ سنلاحظ أن جميع هذه الكائنات (السيارات بأنواعها المختلفة) لها خصائص مشتركة مثل أن جميعها لها عجلات، وكذلك يوجد لها وظائف مشتركة كوظيفة التسارع على سبيل المثال.



تعرفت سابقًا استخدام لغة **Alice** وهي لغة برمجة تعتمد بصورة أساسية على اللبنات البرمجية. تظهر الفئات والكائنات بشكل واضح عند البرمجة باستخدام **Alice**، لنرى ذلك قبل الانتقال إلى استخدام البرمجة كائنية التوجه في **Python**.

الكائنات والفئات في Alice

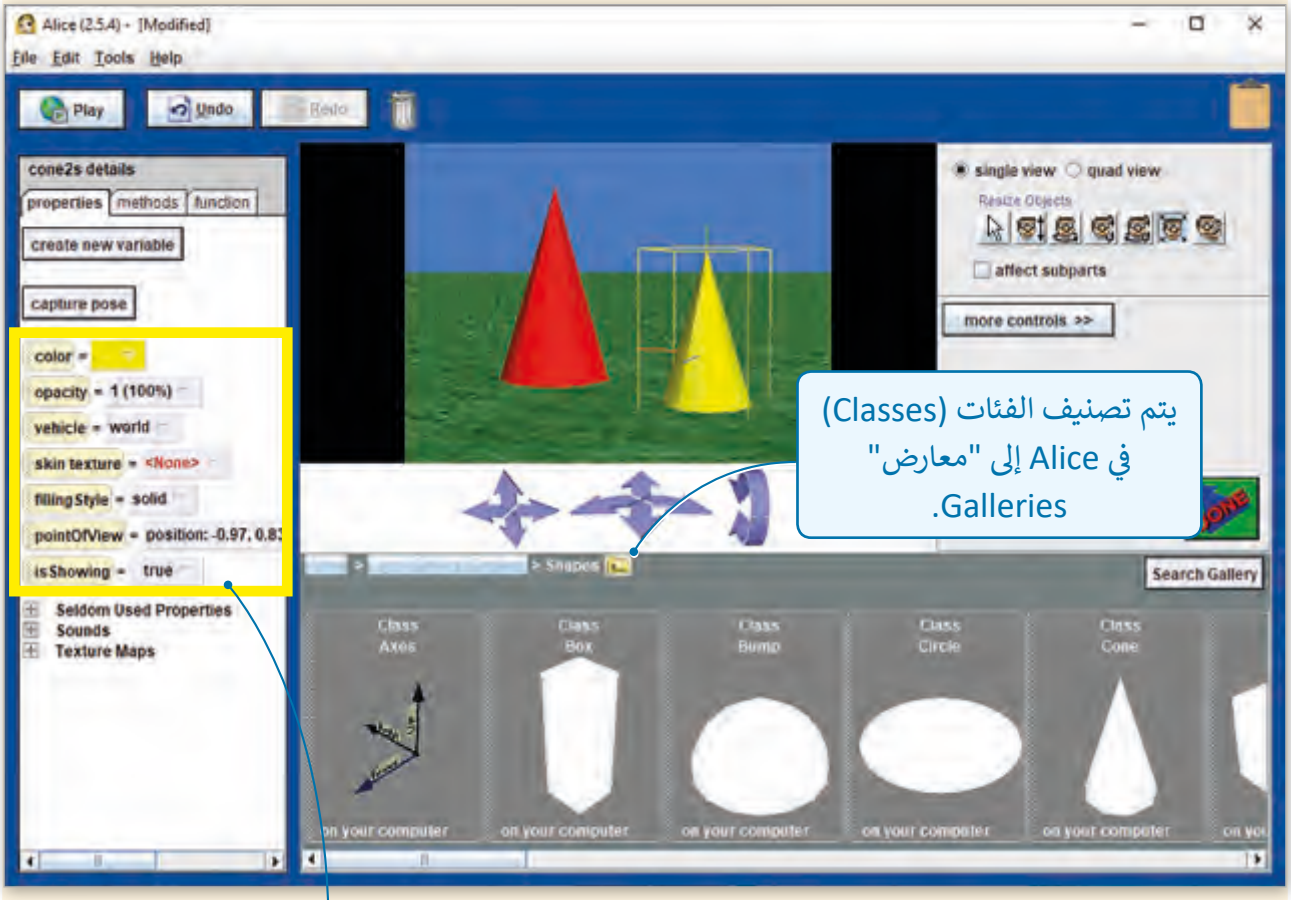
يتشابه وصف الكائن في **Alice** مع الاسم الذي يطلق عليه؛ فالكائن عبارة عن أي شكل ثلاثي الأبعاد يتم تصميمه سواء كان شخصًا أو حيوانًا أو قطعة من الأثاث أو مبنى أو أي شيء موجود في العالم الافتراضي. تتم إضافة معظم الكائنات إلى عالم البرنامج من خلال زر إضافة الكائن **Add object**، كما أن هناك العديد من الكائنات الموجودة بشكل افتراضي في البرنامج مثل: **The ground** (الأرضية)، **Camera** (الكاميرا)، **Light** (الضوء)، وغيرها.



اضغط زر **Add object** لاستعراض الكائنات الموجودة.

يحتوي معرض الكائنات (**Object Gallery**) على الفئات الخاصة بإضافة الكائنات والبحث عنها. إن كل فئة (**Class**) هي نموذج ثلاثي الأبعاد يستخدم لإنشاء كائن (**Object**) من نوع معين.

كما هو دارج في استخدام اللغات التي تعتمد على تقنيات البرمجة كائنية التوجه، يتم إنشاء الكائن **Object** في Alice من الفئة **Class** والتي هي بمثابة القالب لجميع الكائنات المماثلة التي تم إنشاؤها، فعلى سبيل المثال فإن المخروطين في عالم البرنامج في **Alice** الموضح في الصورة أدناه، قد تم إنشاؤهما من فئة اسمها **Cone** (مخروط). تحدد فئة الكائن وظائف وخصائص الكائن الموجود لدينا.



لنستعرض مثالًا يساعدنا في فهم ما تصفه الفئة كقالب أو كمخطط للكائن، فعلى سبيل المثال: جميع الكائنات من فئة "الجمل" **Camel Class** لها شكل ولون الجمل، ويمكنها أداء جميع مهمات الجمل. عندما نضيف كائنًا إلى عالمنا فإننا ننشئ مثيلًا **(Instance)** لفئة "الجمل" ويمكننا حينها تغيير بعض خصائصه ووظائفه.

إنشاء فئة Creating a class

يمكن تعريف الفئات في **Python** بسهولة، وسنستخدم نحن صيغة أسهل (تتطابق مع لغة **Python**) كفرصة للتعرف على بعض المبادئ الرئيسة الخاصة بالبرمجة كائنية التوجه.

لتعريف الفئة نستخدم الصيغة العامة:

```
class Class_Name:
    <attribute 1>
    <attribute 2>

    <attribute n>

    <Method 1>
    <Method 2>

    <Method n>
```

مثال 1:

لننظر إلى المثال التالي حيث سننشئ فئة خاصة بتعريف كائن من نوع **Cat** (قطة) وله ثلاث سمات: الاسم والعمر واللون **(name, age, color)**، ووظيفتين هما الأكل والنوم **(eat, sleep)**.

```
class Cat():
    name
    age
    color

    def eat():
        return eat

    def sleep():
        return sleep
```

يمثل المتغير **self** مثيل الكائن نفسه.

يتم استدعاء وظيفة الإنشاء _init_ في Python بشكل تلقائي عند إنشاء كائن من فئة معينة.

Python

```
class Cat():
    def __init__(self,name,age,color):
        self.name=name
        self.age=age
        self.color=color

    def eat(self):
        return self.eat

    def sleep(self):
        return self.sleep
```

تعريف الفئة

سمات الفئة

وظائف الفئة

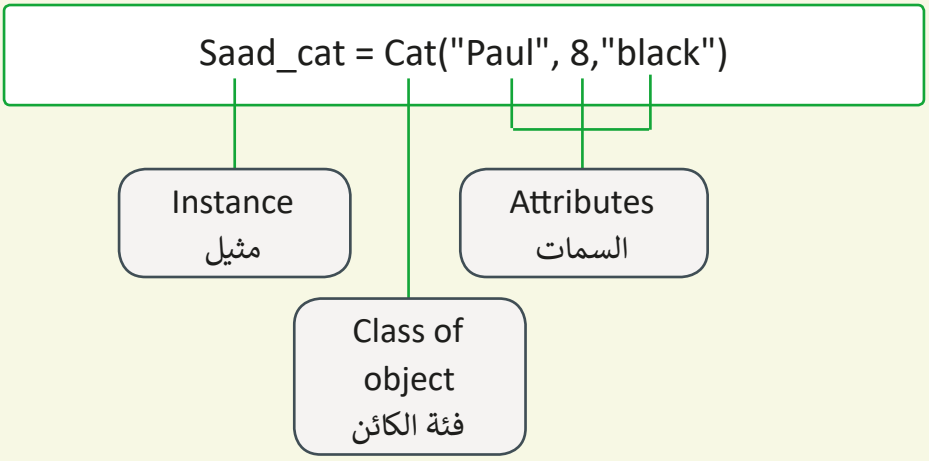
جميع الكائنات التي تنتمي إلى الفئة Cat ستمتلك فقط الخصائص الموضحة في تعريف الفئة، وستكون قادرة فقط على أداء الوظائف المحددة في التعريف.

إنشاء مثيلات الكائنات Creating Instances

سنستخدم الآن الفئة التي قمنا بتعريفها لإنشاء كائنين من نوع cat (قطة)، سنطلق على الأول: **Khaled_cat** (قطة خالد) وهي بيضاء اللون **White** وعمرها سنتين، واسمها **Kitty**، أما الثانية فسنطلق عليها اسم **Saad_cat** (قط سعد) وهو أسود اللون **Black** وعمره 8 سنوات ويدعى "**Paul**".

```
class Cat():
    name
    age
    color

    def eat():
        return eat

    def sleep():
        return sleep
Khaled_cat = Cat(name="Kitty", age=2, color="white")
Saad_cat = Cat(name="Paul", age=8, color="black")
```

يمكن إنشاء كائنات مختلفة أو مثيلات (**Instances**) من فئة **Cat** باستخدام صيغة بسيطة جدًا، وذلك بوضع اسم الفئة بين الأقواس وإعطاء قيم لخصائصها (**Attributes**).



بهذه الطريقة، ومن خلال إعطاء قيم مختلفة لخصائص كل كائن يمكننا إنشاء العديد من الكائنات من فئة **Cat**. جميع الكائنات الجديدة لها نفس الخصائص تمامًا ونفس الوظائف المحددة في تعريف الفئة، ولكن يمكن أن تختلف قيم خصائصها عن بعضها البعض.

Python

```
class Cat:
    def __init__(self,name,age,color):
        self.name=name
        self.age=age
        self.color=color

    def eat(self):
        return self.eat

    def sleep(self):
        return self.sleep
Khaled_cat = Cat("Kitty", 2,"white")
Saad_cat = Cat("Paul", 8, "black")
```

جميع الكائنات (**Objects**) التي يتم إنشاؤها من الفئة (**Class**) تسمى نماذج الفئة (**Instances of the Class**).

إنشاء الكائنات الجديدة أو مثيلات الفئة

إن إنشاء الفئة Class بحد ذاته لا يعني وجود كائنات أو نماذج لهذه الفئة، إنما هو مجرد تعريف أو قالب يحدد خصائص ووظائف الكائنات التابعة لهذه الفئة، ويتم إنشاء تلك الكائنات لاحقًا بشكل منفصل واستخدامها في البرمجة كما يتضح في المثال السابق.

مثال 2:

لنستعرض مثالًا في **Python** لإنشاء فئة تحت مسمى "مركبة" "**vehicle**"، تضم خصائص ووظائف مختلفة للمركبة.

```
class Vehicle:
    def __init__( self, color, price, consumption,speed):
        self.color=color
        self.price=price
        self.consumption=consumption
        self.speed=speed

    def accelerate(self, amount) :
        self.speed += amount
        return  self.speed

    def decelerate(self, amount) :
        self.speed -= amount
        return  self.speed
car1=Vehicle("yellow", 1500.00,12,0)
car1.accelerate(20)
print(car1.speed)
```

تزيد وظيفة accelerate سرعة السيارة بالمقدار الذي يتم تمريره كقيمة للمتغير amount.

بالمثل، تقلل الوظيفة decelerate سرعة السيارة بالمقدار الذي يتم تمريره كقيمة للمتغير amount.

نصيحة ذكية

اترك سطرين فارغين بين كل فئة والأخرى (كفاصل بين الفئات) لتوضيح أن كل فئة قائمة بذاتها، واترك سطرًا واحدًا فارغا عند تعريف الوظائف الموجودة داخل الفئات (كفاصل بين الوظائف).

خصائص الفئة Vehicle

- Color < (اللون).
- Price < (السعر).
- Fuel Consumption < (استهلاك الوقود)
- Current Speed < (السرعة الحالية)

وظائف الفئة Vehicle

- Accelerate > (زيادة السرعة)
- Decelerate > (تقليل السرعة)

نلاحظ أن خصائص الفئة تم إدراجها داخل الدالة **_init_** كمتغيرات من نوع **self**، كما يظهر هذا المتغير كأول معامل في جميع الوظائف المعرفة في الفئة.

لنتناول هذا الأمر بمزيد من التفصيل.

استخدام المعامل Self

المعامل self هو مرجع إلى الكائن أو المثيل الحالي للفئة، أي أنه يشير إلى المثيل الذي يتم إنشائه من الفئة المعرفة في المقطع البرمجي، وفي مثالنا هذا نقوم بإنشاء الكائن car1 كمثيل من الفئة Vehicel باستخدام الجملة البرمجية car1=vehicle()، وعند استدعاء الوظيفة car1.accelerate(10) يتم تطبيق الإجراءات المدرجة في هذه الوظيفة على هذا الكائن بالتحديد.

يسمح معامل self بتطبيق إجراءات الوظيفة التي يتم استدعاؤها على مثيل واحد فقط من الفئة (**class**) في وقت معين وليس على الفئة بكاملها.

لا تعتبر كلمة "**self**" كلمة محجوزة في **Python** ولكن من المتعارف استخدامها في هذا السياق. وكما نعلم فإن المترجم يقوم بفحص الكود البرمجي عند تنفيذ البرنامج، وقد نحصل على رسائل خطأ عند كتابة الكلمات المحجوزة بطريقة غير سليمة، لذا ينبغي على المبرمج أن يكون حذرًا عند كتابته للكلمات المحجوزة داخل البرنامج.

نصيحة ذكية

كن حذرًا عند تعاملك مع المسافة البادئة (Indentation) في الجمل البرمجية فهي مهمة في Python. وبالنسبة للفئات فإن مستوى المسافة البادئة لكل سطر في التعليمات البرمجية يحدد كيفية تجميع الجمل البرمجية معًا. استخدم أربعة فراغات لتعيين المسافة البادئة.

وظيفة _init_

الغرض الأساسي من هذه الوظيفة الخاصة هو إنشاء الكائنات وتهيئة (**initialize**) خصائصها المعرفة داخل الفئة التي تنتمي إليها، يتم استدعاء هذه الوظيفة بشكل تلقائي عند إنشاء أي نموذج من الفئة المعرفة في المقطع البرمجي، ولذلك تسمى الوظيفة **__init__** بوظيفة الإنشاء أو **Constructor** (المنشيء).

عند كتابة الوظيفة **__init__** في تعريف الفئة ينبغي تحديد كافة خواص الفئة داخلها لتهيئتها، وذلك تجنبًا لحدوث الأخطاء في المقطع البرمجي لاحقًا، لاحظ الخطأ في المقطع البرمجي التالي، والذي يتم فيه تعريف واستخدام الخاصية speed في إحدى وظائف الفئة دون وضعها مسبقًا داخل وظيفة الإنشاء:

```
class Vehicle:
    def __init__ ( self, color, price, consumption):
        self.color=color
        self.price=price
        self.consumption=consumption

    def set_speed(self,speed=100):
        self.set_speed=speed

    def accelerate(self, amount) :
        self.speed += amount
        return  self.speed

    def decelerate(self, amount) :
        self.speed -= amount
        return  self.speed

car1=Vehicle("yellow", 1500.00,12)
car1.accelerate(10)
```

تم تعريف خاصية (**speed**) في وظيفة **set_speed()**. والتي سيتم استدعاؤها لاحقًا في الوظيفة **accelerate**.

تحاول وظيفة accelerate تعديل خاصية (**speed**) والتي لم يتم تضمينها مسبقًا في وظيفة الإنشاء (**constructor**).

```
AttributeError: "Vehicle" object has no attribute "speed"
```

لم يتم التعرف على الخاصية speed عند استدعاء الوظيفة acceletate لأنها لم تكن ضمن الخصائص المعرفة في وظيفة الإنشاء، وإنما في الوظيفة set_speed، التي لم يتم استدعاؤها إطلاقًا بعد إنشاء الكائن.

تهيئ الوظيفة **__init__** جميع خصائص الفئة **Vehicle** بحيث تتعرف عليها جميع وظائف الفئة الأخرى عند استخدامها، ما عدا خاصية **speed** والتي يتم تهيئتها بواسطة وظيفة **set_speed**.

تحدث المشكلة في السطرين البرمجيين التاليين عند إنشاء الكائن car1 كنموذج للفئة **Vehicle** ثم استدعاء الدالة **accelerate**:

```
car1=Vehicle("yellow",1500.00,12)
car1.accelerate(10)
```

ستقوم الدالة **accelerate** بمحاولة تغيير خاصية السرعة speed والتي لم يتم تهيئتها بعد.

تهيئة خصائص الكائن:

إن تهيئة الخصائص خارج **_init_** يزيد من إمكانية ظهور أخطاء منطقية.

دعونا نتعامل مع هذه الحالة كما في المثال التالي.

```
class Vehicle:
    def __init__ ( self, color, price, consumption):
        self.color=color
        self.price=price
        self.consumption=consumption
        self.speed=0
    def set_speed(self,speed=0):
        self.speed=speed
        return self.speed
    def accelerate(self, amount) :
        self.speed += amount
        return self.speed
    def decelerate(self, amount) :
        self.speed -= amount
        return self.speed
car1=Vehicle("yellow", 1500.00,12)
print("Speed before accelerate is :",car1.speed)
car1.set_speed(50)
print("Speed after setting acceleration is :",car1.
speed)
car1.accelerate(10)
print("Speed after 1st acceleration is :",car1.speed)
car1.accelerate(5)
print("Speed after 2nd acceleration is :",car1.speed)
```

من الأفضل تعريف جميع الخصائص في المُنشئ (**constructor**).

لا تظهر المشكلة عند تنفيذ وظيفة **accelerate** لأن التهيئة (**initialization**) تمت على المُنشئ.

```
10
```

إنشاء المثيل وتهيئة خصائصه Instantiating the Object

كما أشرنا مسبقًا، فإن خصائص الكائنات المختلفة غالبًا ما يكون لها قيم مختلفة. لنلقِ نظرَة على مثال آخر لبرنامج مكتمل.

مثال 3:

سننشئ فئة باسم **Horse** التي سيكون لها خصائص ثلاثة وهي السلالة والحجم واللون، ولها أيضًا وظيفتين متمثلتين في التغذية والجري، سنقوم بعدها بإنشاء كائنين اثنين أو مثيلي لهذه الفئة، (**Instantiating the Object**)، وسنقوم بتعريف الخصائص واستدعاء الوظائف لكل منهما:

```
class Horse:

    def __init__(self, breed, size, color):
       self.breed=breed
       self.size= size
       self.color = color
      print ( "breed:",self.breed,"size:",self.size,"color:",self.color)

    def eat(self,food):
       self.food = food
       print  ("I am eating ", food)

    def run(self):
       print ("I am running")

Max = Horse("Arabian", "medium", "brown")
Rocky = Horse( "Andalusian", "large", "light brown")
Max.eat("carrot")
Rocky.eat("apple")
```

تقوم هذه الجملة بإنشاء الكائن Max وهو مثيل للفئة Horse.

```
breed: Arabian size: medium color: brown
breed: Andalusian size: large color: light brown
I am eating  carrot
I am eating  apple
```

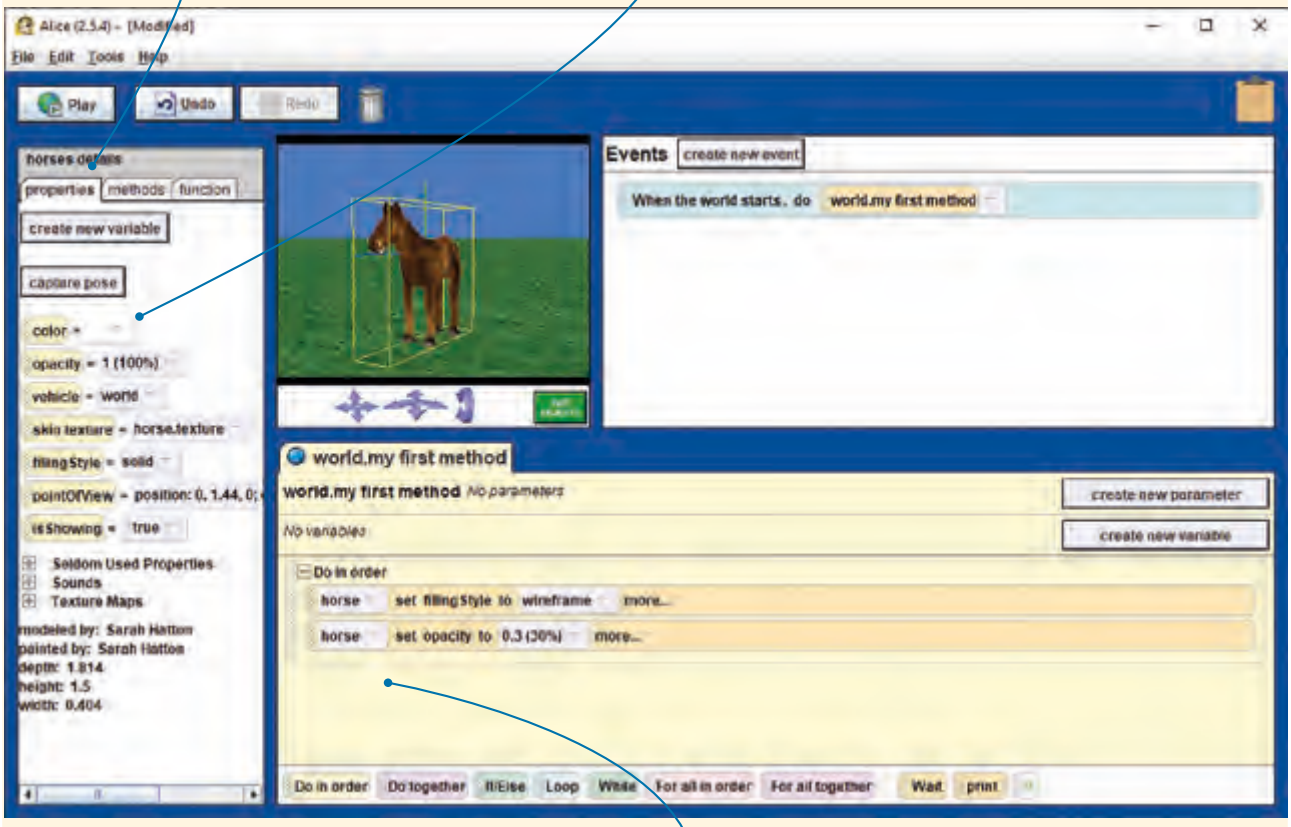
قمنا بعد تعريف الفئة بإنشاء الكائنات (المثيلات) **Max** و **Rocky**، لاحظ أننا في كل مثيل أنشأناه قمنا بتمرير قيم لجميع الخصائص المعرفة في فئة **Horse** والتي تتم تهيئها باستدعاء وظيفة **__init__** عند إنشاء المثيل، مثلًا عند تنفيذ جملة إنشاء المثيل **Max** يتم تمرير القيمة "Arabian" إلى خاصية السلالة **breed** الخاصة بهذا المثيل تحديدًا، وتقوم وظيفة **__init__** بتهيئتها عن طريق الجملة **self.breed=breed** الموجودة ضمن تعريف الفئة. وهكذا باقي الخصائص.

الخصائص في Alice

تصف خصائص الكائن في **Alice** خصائص مثيل معين، فجميع الكائنات لها خصائص معينة مثل لونها وموضعها. يتم سرد خصائص الكائن في علامة التبويب **Properties** (الخصائص) في لوحة **details** (التفاصيل).

اضغط علامة تبويب **properties** (الخصائص) للكائن **horse**.

قيمة الخاصية يمكن تغييرها مباشرةً في لوحة details من خلال استخدام القائمة المنسدلة وهذا يشكل الحالة الأولية للكائن.



يمكن تغيير قيمة الخاصية أيضًا أثناء الحركة (**animation**) عن طريق استدعاء الوظيفة.

لتغيير قيمة خاصية باستخدام وظيفة، اسحب الخاصية من لوحة **details** (التفاصيل) إلى **method editor** (محرر الوظيفة). سيؤدي هذا الأمر إلى إضافة استدعاء وظيفة "set" لتلك الخاصية.

1

أجب على الأسئلة التالية.

1. ما هو الفرق الرئيس بين البرمجة الإجرائية والبرمجة كائنية التوجه OOP؟

_____

_____

_____

_____

2. عرف المقصود بالكائن Object في البرمجة كائنية التوجه.

_____

_____

_____

_____

3. مالفرق بين الكائن والفئة في عالم البرمجة كائنية التوجه ؟

_____

_____

_____

_____

2

ضع علامة ✓ أمام العبارة الصحيحة وعلامة ✗ أمام العبارة الخطأ.

.1	تستند البرمجة الإجرائية على الكائنات والفئات.	
.2	تمكن لغة Python من كتابة البرامج كائنية التوجه وكذلك البرامج الاجرائية.	
.3	الكائنات لها خصائص محددة ويمكنها تنفيذ إجراءات معينة.	
.4	المصطلح الذي يطلع على وظائف الكائن هو Attributes.	
.5	الفئة هي قالب تعليمات برمجية لإنشاء الكائنات.	
.6	يسمى الكائن أيضاً مثيل الفئة.	
.7	يمكن أن نطلق على الوظيفة __init__ اسم المنشئ Constructor.	
.8	المعامل self هو مرجع إلى المثيل الحالي للفئة.	

3

ما هي الوظيفة __init__؟ لماذا نستخدمها؟

_____

_____

_____

_____

_____

4

لكل طالب بالمدرسة عمر، صف، وجنسية، نفذ الآتي لإنشاء فئة ومثيل منها لأحد الطلاب:

1. قم بانشاء فئة Student التي تمكن من نمذجة بيانات الطلاب.
2. قم بانشاء الكائن Khalid حسب البيانات التالية:

- العمر 16
- الصف: 12
- الجنسية: Qatari

3. اعرض الصف للطالب خالد على الشاشة.
4. اضف الوظيفة Method التي تمكن من اضافة عدد من السنين لعمر Khalid.

5 استنادًا للفئة الآتية، قم بإنشاء الكائن Ict_teacher وذلك بتعيين القيم (name) و (department) التي تتماشى مع بيانات معلمك.

```
class Employee:

    def __init__(self, name,department):
        self.name=name
        self.department=department
```

_____

6 قم بإنشاء البرنامج التالي.

1. قم بإنشاء الفئة Rectangle لتمثيل مستطيل وحدد لها خاصيتين هما الطول height والعرض width
2. قم بانشاء الوظيفة Perimeter التي تمكن من احتساب محيط المستطيل (ابحث للتوصل إلى قاعدة احتساب محيط المستطيل).
3. قم بانشاء الوظيفة Area التي تمكن من احتساب مساحة المستطيل (ابحث للتوصل إلى قاعدة احتساب مساحة المستطيل)
4. قم بإنشاء الكائن MyRect وعين طوله ليكون 20 وعرضه ليكون 15 ، كنموذج للفئة Rectangle.
5. اعرض على الشاشة محيط ومساحة المستطيل MyRect

7

أجب عن الأسئلة ونفذ التعليمات أدناه اعتمادًا على المقطع البرمجي المعطى.

```
class Car:

    def __init__(self, make):
        self.make=make
        self.speed = 60

    def speed_up(self,speed):
        self.speed = speed

        print ("I am driving at a speed", self.speed, "km/h")
    def turn(self):
        print ("I am turninng....")
```

1. حدد وظيفة الإنشاء في المقطع أعلاه.
2. سجل خصائص الفئة ووظائفها.
3. أضف الخاصيتين: النوع واللون والموديل، بحيث يتم تهيئة كل منهما داخل وظيفة الإنشاء.
4. عدل وظيفة الانعطاف turn بحيث تستقبل ضمن معاملاتها قيمة تحدد اتجاه انعطاف السيارة (يمينًا أو يسارًا).
5. قم بإنشاء الكائنات (المثيلات) الآتية لفئة السيارة:
   I. كائن يمثل سيارة رياضية "Convertible" من طراز BMW، موديل 2016، ولونها "أسود".
   II. كائن يمثل سيارة صالون "Sedan" من طراز Toyote، موديل 2018، ولونها "أحمر".
6. استدع الوظيفة التي تجعل السيارة الرياضية تنعطف يمينًا.
7. استدع الوظيفة التي تجعل السيارة الصالون تسير بسرعة 90 كيلومترًا في الساعة.

الدرس الثاني

السمات Attributes والوظائف Methods

تعلمت في الدرس الماضي أن الكائنات (**Objects**) هي كيانات تنتمي إلى فئات (**Classes**)، تتميز بسمات أو مواصفات ويمكنها تنفيذ إجراءات محددة. وكما أشرنا خلال الدرس، تسمى مواصفات الكائن أو خصائصه بـ "السمات" **Attributes**، أما الإجراءات التي يقوم بها فتسمى بـ "الوظائف" **Methods**، وهي تشبه في مفهومها الدوال **Functions** والتي درستها سابقًا.

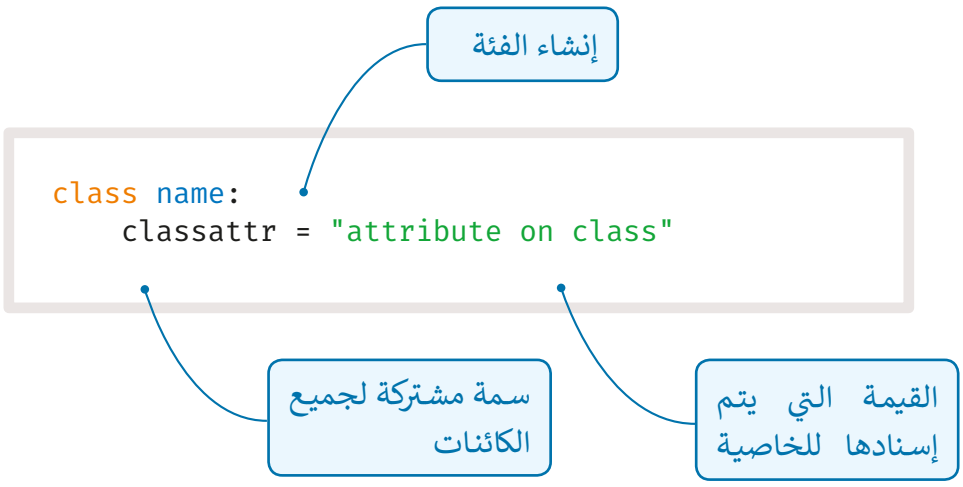
سنتعرف في درسنا الآتي مزيدًا من التفاصيل حول الخصائص والوظائف ونلقي الضوء على أنواعها.

تعريف الخصائص Defining Attributes

تعرفت في الدرس الماضي على وظيفة "المنشيء" والتي تقوم بإنشاء وتهيئة الخصائص داخل الفئة ومثيلاتها وهي الطريقة المستحسنة في إنشاء الخصائص، ولكن يمكن أيضًا تعريف الخصائص داخل الفئة دون استخدام وظيفة المنشيء أو تعريفها خارج الفئة من خلال المثيل.

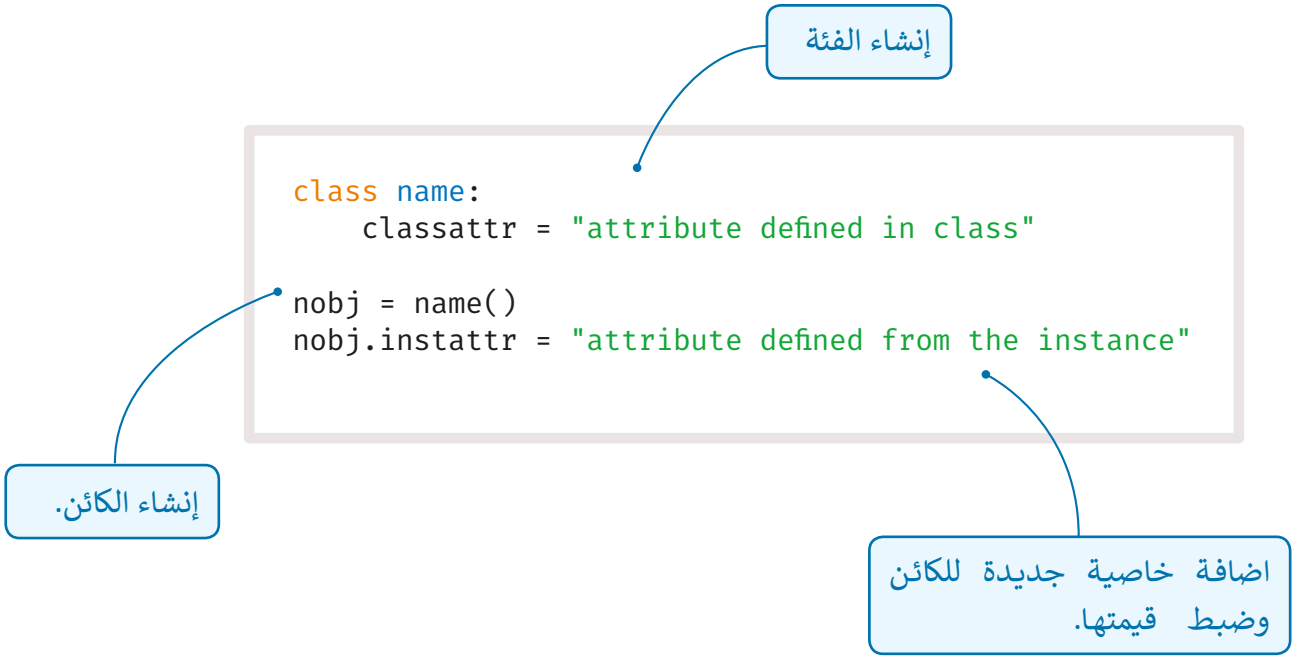
تعريف الخصائص داخل الفئة

يمكن إنشاء الخاصية في **Python** داخل الفئة (**Class**) من خلال ضبطها مباشرة إلى القيمة المطلوبة.



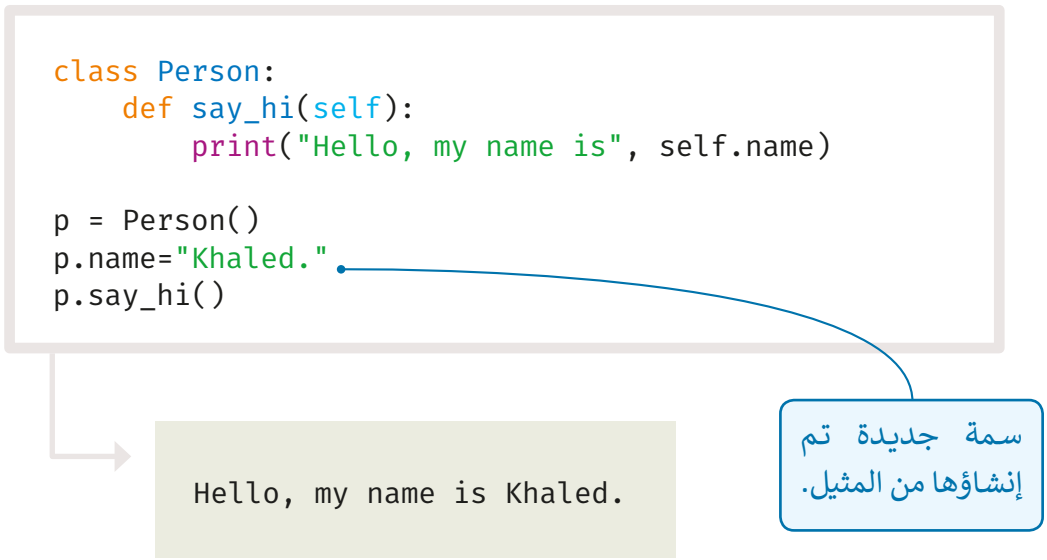
تعريف الخصائص خارج الفئة

يمكن أيضًا إضافة خاصية خارج تعريف الفئة. وذلك بإنشاء مثيل **Instance** لكائن وذلك بوضع اسم المثيل متبوعًا بنقطة ثم وضع اسم الخاصية.



مثال 1:

لنعاين المثال التالي لنتعرف أكثر على كيفية القيام بذلك.
لقد قمنا بتعريف فئة (**class**) بإسم **person** ولها وظيفة واحدة فقط وهي إنشاء تعريف لاسم الشخص.



الوصول إلى خصائص الكائن

للوصول إلى خصائص الكائن الذي أنشأناه، نقوم باستخدام ما يسمى بالتدوين النقطي (**Dot Notation**) بالصيغة التالية:

```
object_name.property
```

حيث يمثل **object_name** إسم الكائن وتمثل **property** الخاصية.

مثال 2:

بالعودة لمثال القطة **Cat**، يمكن عرض اسم كل قطة من خلال الوصول لخصائص الكائن (القطة).

```
class Cat:
    def __init__(self,name,age):
        self.name=name
        self.age=age

    def eat(self):
        return self.eat

    def sleep(self):
        print self.sleep

Khaled_cat = Cat("Kitty", 2)
Saad_cat = Cat("Paul", 8)
print(Khaled_cat.name)
print(Saad_cat.name)
```

يقصد بالتعبير **Khaled_cat.name** "اذهب إلى الكائن **Khaled_cat** واحصل على قيمة خاصية **name** (الاسم)".

```
Kitty
Paul
```

مثال 3:

بنفس الطريقة يتم الأمر مع مثال الحصان **Horse**.

```
class Horse:

    def __init__(self, breed, size, color):
        self.breed=breed
        self.size= size
        self.color = color

    def eat(self,food):
        self.food = food
        print  ("I am eating ", food)

    def run(self):
        return ("I am running")

Max = Horse("Arabian", "medium", "brown")
Rocky = Horse( "Andalusian", "large", "light brown")
print(Max.breed)
```

لتمرير القيم (**Arguments**) إلى المعاملات **breed** و **size** و **color**، ضع القيم بين الأقواس بعد اسم الفئة بنفس الطريقة التي استخدمنا بها سابقًا المتغيرات في الدوال.

```
Arabian
```

يمكننا الوصول إلى خصائص المثيل (**Instance Attributes**) من خلال استخدام التدوين النقطي (**dot notation**).

إنشاء وتعريف الوظائف

تتشابه الوظائف (**Methods**) مع الدوال (**Functions**)، ولكن مع وجود اختلافات في الصيغة بلغة **Python**، مثل:

- يتم تعريف الوظائف داخل المقطع البرمجي الخاص بالفئة، مما يجعل العلاقة بين الفئة وكل وظيفة أمرًا واضحًا.
- تختلف صيغة استدعاء الوظيفة (**calling a Method**) عن صيغة استدعاء الدالة (**calling a Function**).

يتم تعريف الوظائف داخل الفئة باستخدام نفس الصيغة المستخدمة لتعريف الدالة. فأول معامل يجب أن يكون مثيل الفئة التي يتم استدعاء وظيفتها.

مثال 4:

لنستعرض المثال التالي حيث سنُعرِّف في البداية الفئة **Student** (الطالب)، وسيقوم كل طالب مثيل للفئة **Student** بالتعريف عن نفسه.

```
class Student:
  def __init__(self, age,grade ):
    self.name=name
    self.age=age
    self.grade=12

  def show(self):
    print("I am",self.name,",I am " , self.age ,
"and I am a student of the grade" , self.grade)
```

يجب أن تأخذ كل وظيفة معامل **self** كأول معامل حيث تخبره بأن عمل هذه الوظيفة يختص بهذا الكائن.

استدعاء الوظائف Call the methods

لكي نستدعي الوظائف الخاصة بالكائنات التي أنشأناها فإننا نستخدم التدوين النقطي كما يلي:

```
object_name.method ()
```

يمكنك أن تستنتج مما تعلمته حتى الآن أن تعريف الوظائف يتم داخل الفئة، ولكن استدعاءها يتم من خلال كائن أو مثيل يتم إنشاؤه في المقطع البرمجي، ويتم تطبيق كافة إجراءاتها على هذا المثيل.

سننشئ كائنًا باسم student 1 من الفئة Student التي كنا قد عرفناها مسبقًا:

```
class Student:
  def __init__(self,name,age ):
    self.name=name
    self.age=age
    self.grade=12

  def show(self):
    print("I am",self.name,",I am " , self.age ,
"and I am a student of the grade" , self.grade)

student1=Student("Khaled",16)
student1.show()
```

سنستدعي وظيفة **show()** في الكائن **student1**.

```
I am Khaled ,I am 16 and I am a student
of the grade 12
```

لاحظ أن قيمة الخاصية grade ثابتة وتساوي 12 وذلك لأننا استخدمنا القيمة المبدئية التي تم تعيينها لهذه الخاصية عند تعريف الفئة Student.

لنستعرض مثالًا آخر:

مثال 5:

```
class Dog:


    def __init__(self, breed, color):

        self.name="Roger"
        self.breed = breed
        self.color = color

    def eat(self,food):
        print(self.name, "is eating", food)

Roger = Dog("Pug", "brown")

print("Roger details:")
print("Breed: ", Roger.breed)
print("Color: ", Roger.color)
Roger.eat("meat")
```

المُنشئ **Constructor**

كائن من فئة **Dog**

استدعاء الوظيفة وتمرير قيمة إلى معاملاتها.

```
Roger details:
Breed:  Pug
Color:  brown
Roger is eating meat
```

كيف يمكنني تغيير اسم الكلب من Roger إلى Buddy دون المساس بالوظيفة __init__؟

الوظائف الثابتة Static Methods

الوظائف الثابتة هي وظائف خاصة تضم تعليمات برمجية تنتمي إلى الفئة ولكنها لا تستخدم أي من خصائصها على الإطلاق. تعمل الوظائف الثابتة مثل الوظائف العادية، ولكن عادةً ما يتم تنظيمها في تعليمات الفئة البرمجية للقيام بعمليات ثابتة تختص بالفئة وتطبق على جميع الكائنات المنبثقة منها.

تستخدم الوظيفة الثابتة عندما تكون نتيجة استدعاء الوظيفة هي نفسها بغض النظر عن مثيل الكائن أو مكان استدعائها. لننظر إلى المثال التالي لكائن من الفئة "Dog". نحن نعلم مثلًا أنه يمكن لأي كلب أن ينبح بغض النظر عن سلالته أو لونه، ولذلك لا يرتبط هذا الإجراء بمثيل الكائن المسمى **Roger** أو **Rocky** وإنما يرتبط بالفئة نفسها.

```
class Dog:


    def __init__(self, breed, color):
        self.breed = breed
        self.color = color

    def eat(self,food):
        print(self.name, "is eating", food)

    def bark(self):
        print("I am barking!")

Roger = Dog("Pug", "brown")
Rocky=Dog("bulldog","white")

Roger.bark()
Rocky.bark()
```

هذه هي الوظيفة الثابتة، حيث لا تستخدم أي خصائص ضمن معاملاتها.

```
I am barking!
I am barking!
```

بغض النظر عن المثيل الذي يستدعي الوظيفة، ستكون النتيجة هي نفسها لكل استدعاء، وفي هذه الحالي لا داعي لاستخدام المعامل self في تعريف الوظيفة داخل الفئة، وعليه يمكن تعديل المقطع كما هو موضح تاليًا.

يمكن تغيير المقطع البرمجي السابق بواسطة استخدام جملة **@staticmethod** قبل تعريف الوظيفة.

```
class Dog:

    def __init__(self, breed, color):
        self.breed = breed
        self.color = color

    def eat(self,food):
        print(self.name, "is eating", food)

   @staticmethod
    def bark():
        print("I am barking!")

Rodger = Dog("Pug", "brown")
Rocky=Dog("bulldog","white")

Dog.bark()
Rodger.bark()
Rocky.bark()
```

تعمل الوظيفة الثابتة على مستوى الفئة مع جميع الكائنات التي تنتمي إليها وستكون النتيجة نفسها على الدوام.

تعمل الوظيفة في مستوى الفئة ومع أي كائن من نفس الفئة أيضًا.

```
I am barking!
I am barking!
I am barking!
```

قواعد عامة عن الوظائف الثابتة:

الوظائف الثابتة عبارة عن:

1. دوال بسيطة بدون المعامل **self**.
2. يجب وضعها داخل تعريف الفئة.
3. يمكن استدعاؤها بواسطة الفئة والكائن.
4. تكون نتيجة هذه الوظائف مستقلة عن حالة الكائن.
5. يتم إنشاؤها بواسطة جملة @staticmethod.

وظائف الفئة @classmethod

تعمل وظائف الفئة (**Class Methods**) على الخصائص البرمجية للفئة، وتأخذ المعامل cls دائمًا كأول معامل لها.

يصف المثال التالي وظيفة فئة تُرجع عدد المثيلات التي تم إنشاؤها من هذه الفئة، وذلك بواسطة الخاصية **no_instances**.

الصيغة العامة لتعريف class method

```
@classmethod
def func_name(cls, args...)
```

```
class Dog:
    no_instances=0

    def __init__(self, breed, color):
        self.name="Rodger"
        self.breed = breed
        self.color = color
        Dog.no_instances+=1

    def eat(self,food):
        print(self.name, "is eating", food)

    @classmethod
    def get_no_instances(cls):
        return cls.no_instances

Rodger = Dog("Pug", "brown")
Rocky=Dog("bulldog","white")

print(Dog.get_no_instances())
```

```
2
```

عند استدعاء وظيفة الفئة هنا تقوم بإظهار عدد المثيلات التي تم إنشاؤها من الفئة Dog.

الوظائف في Alice

يمتلك كل كائن في **Alice** بعض الوظائف الجاهزة، والتي تصف السلوك المحتمل للكائن. على سبيل المثال فإن معظم كائنات Alice لديها وظيفة تسمى **turn**، والتي تنفذ عملية استدارة الكائن عند استدعائها، وبشكل مشابه فإن وظيفة **move** تحرك الكائن في اتجاه محدد عند استدعائها.

يمكنك كتابة وظائفك الخاصة بالكائنات لتنفيذ المزيد من العمليات. تعتبر عملية كتابة الوظائف من أهم أساسيات علم البرمجة، وبشكل خاص إنشاء وظائف أصغر كلبنات برمجية لبناء مجموعات أكثر تعقيدًا من الأوامر.

تسمح لنا كتابة الوظيفة بالتفكير في المهمة بشكل عام بدلًا من التركيز على جميع الإجراءات الصغيرة المطلوبة لإكمالها، وهذا ما يُسمى بالتجريد (**Abstraction**).

أحد الأمثلة المميزة التي يمكن القيام بها برمجيًا هو برمجة كائن (Object) لشخص ما بحيث يمكنه السير بشكل طبيعي في العالم الافتراضي.

بالإضافة إلى التقدم إلى الأمام، يمكن أن تتضمن عملية المشي وظائف فرعية كالآتي:

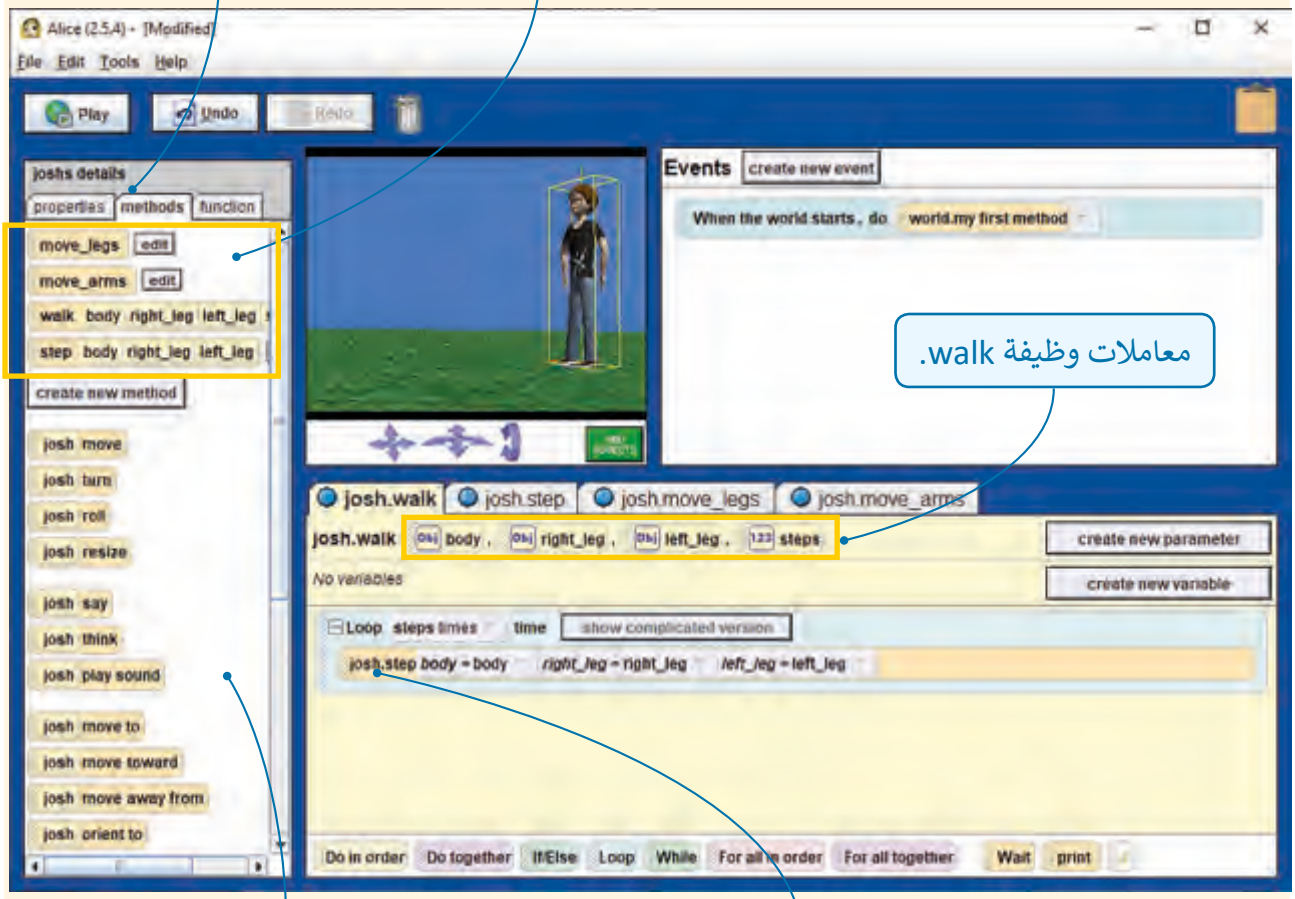
- وظائف تمثل تحريك وانثناء القدمين والذراعين بشكل مستقل.
- وظائف تمثل حركة القدمين والذراعين للقيام بخطوة واحدة مكتملة إلى الأمام.

يتم دمج هذه الوظائف معًا لإنشاء وظيفة كبيرة متكاملة تختص بجعل الكائن يتحرك بشكل يحاكي حركة الإنسان.

لنستعرض التعليمات البرمجية الخاصة بوظيفة المشي:

اضغط علامة تبويب **methods** للوظائف الخاصة بمشروع **Josh**.

الوظائف الجديدة التي تم إنشاؤها.



معاملات وظيفة walk.

الوظائف المتاحة للكائن مدرجة في لوحة details عند تحديد هذا الكائن في قائمة المشروع.

نستدعي وظيفة step في وظيفة walk ونمرر المزيد من المعاملات إلى وظيفة step.

يمكن أن تكون الوظائف في **Alice** على مستوى الشخصية **character** (أي مُعرفة كإجراءات لكائن مستقل) أو على مستوى العالم **world** (بما فيها الإجراءات لأكثر من كائن). يطلق على الوظائف في مستوى الشخصية في لغات البرمجة كائنية التوجه تسمية وظائف مستوى الفئة **class-level methods**.

1

ما الخصائص وما الوظائف الخاصة بالكائنات؟ أعط مثالًا يوضح استخدام كل منهما.

_____

_____

_____

_____

2

في المقطع البرمجي التالي قمنا بإنشاء الفئة Fruit، أكمل كتابة المقطع وذلك لـ:

> إنشاء كائن مثيل باسم Fruit1 ، بلون برتقالي وحجم 5.

> عرض خصائص الكائن Fruit1 على الشاشة.

```
class Fruit:
  def __init__(self, color, size):
    self.name="citrus"
    self.color=color
    self.size=size

_____

_____
```

```
orange
5
```

3

أكمل كتابة المقطع البرمجي التالي وذلك بـ:

> استكمال تعريف الفئة Person.

> استدعاء الوظيفة greet() لتقديم التحية من قبل الكائن p1.

```
_____________________________________________________________


  def __init__(self, name, age):
    self.name = name
    self.age = age

  def greet(self):
    print("Hello my name is " + self.name)

p1 = Person("Saad", 17)


_____________________________________________________________

_____________________________________________________________
```

```
17
Hello my name is Saad
```

4

نفذ الخطوات الآتية:

> استكمل تعريف الفئة Vehicle التي تتضمن الخصائص max_speed و milage، والوظيفة car_details التي تطبع معلومات السيارة.

> أنشيء الكائن المثيل myCar ، علمًا بأن سرعته القصوى max_speed هي 200 والمسافة المقطوعة milage تساوي 180كم.

> استدع الوظيفة الخاصة بالكائن واكتب مخرجاتها على الشاشة.

```
class Vehicle:
_____________________________________________________
_____________________________________________________
_____________________________________________________
_____________________________________________________
_____________________________________________________
_____________________________________________________
_____________________________________________________
_____________________________________________________


print(myCar.max_speed, myCar.mileage)
```

5

لماذا نستخدم جملة staticmethod@ ؟

_____

_____

_____

6

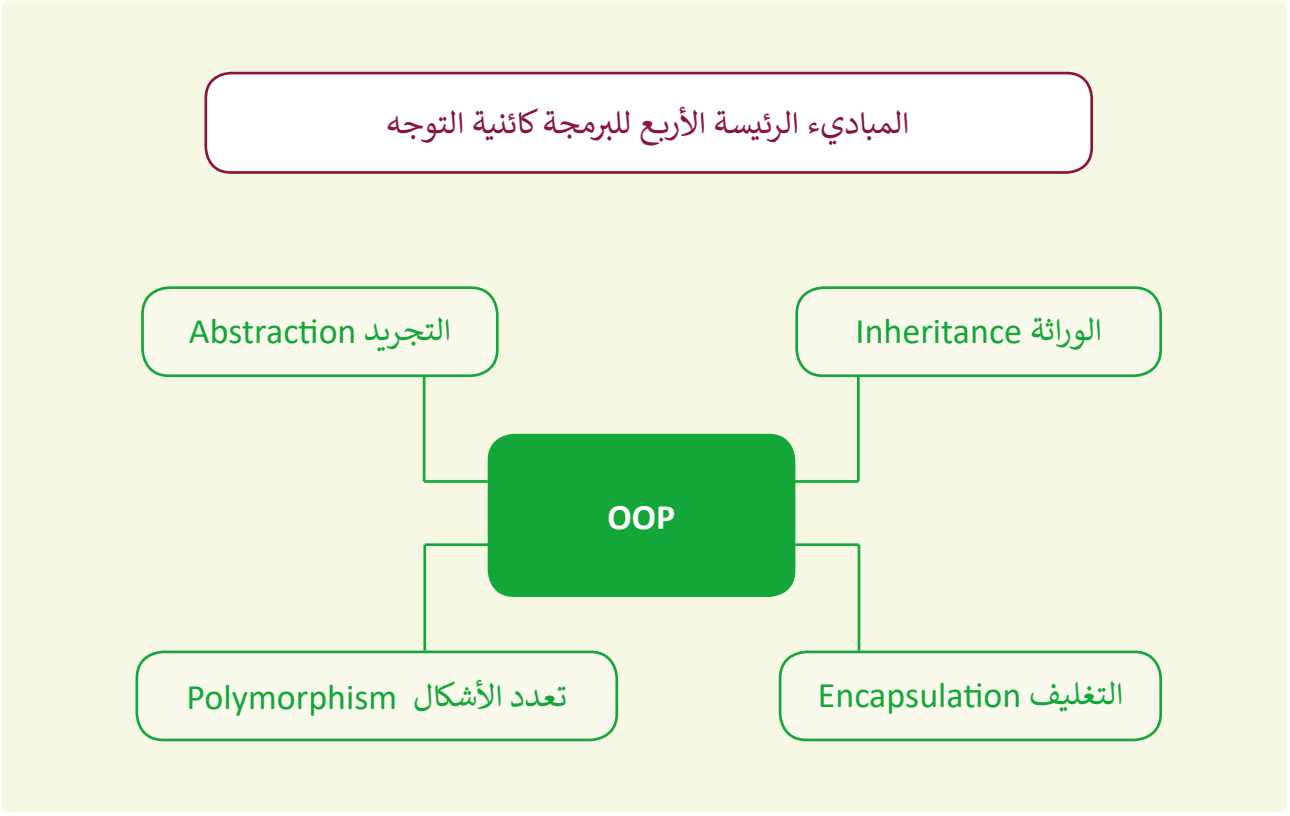
ابحث عن الأخطاء الواردة في المقطع البرمجي التالي وقم بتصحيحها، ثم أكتب نتيجة التنفيذ.

```
class Student:

    def __init__(self, name,grade):
        self.name=name
        self.grade=grade

    def work(self):
        print("My name is",self.name,"and i go to school 5
days a week in grade", self.grade)

Student("Saad","12")
student1.work
```

الدرس الثالث

مبادئ البرمجة كائنية التوجه (1)

كما تعرفنا سابقًا فإن البرمجة كائنية التوجه (**Object Oriented Programming - OOP**) تعتمد على مفهوم استخدام الكائنات بديلًا عن الأوامر، وتتعامل مع الفئات (**Classes**) والكائنات (**Objects**) بالإضافة إلى تنفيذ واستخدام المبادئ والمفاهيم الأساسية الموجهة للكائنات والتي تتمثل في الوراثة والتجريد والتغليف وتعدد الأشكال.



سنتعرف الآن المقصود بهذه المفاهيم الرئيسة في البرمجة كائنية التوجه.

الوراثة Inheritance

يعتبر مفهوم الوراثة من أهم الميزات الرئيسة في البرمجة كائنية التوجه.

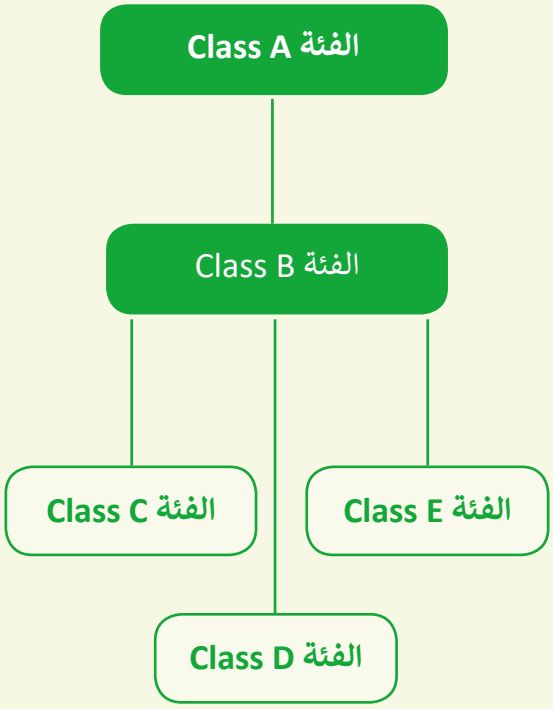
الوراثة هي عبارة عن آلية تسمح للفئة بتوريث جميع سماتها (**Attributes**) ووظائفها (**Methods**) إلى فئة أخرى.

تسمى الفئة المعرفة أولاً بالفئة الأساسية **Base Class**، وتسمى الفئة التي ترث من الفئة الأساسية بالفئة المشتقة **Derived Class**. ويطلق أحيانًا على الفئات الأساسية اسم الفئة العليا **Superclass**، ويمكن أن يطلق على الفئات المشتقة اسم الفئة الفرعية **Subclass** .

يمكن للفئة أن ترث من فئة واحدة أو أكثر من الفئات العليا، كما يمكن أن تحتوي كل فئة على عدد غير محدود من الفئات الفرعية.

ترث الفئات الفرعية خصائص ووظائف الفئة العليا، وذلك يعني أن المبرمج لا يحتاج إلى إعادة تعريف أو نسخ الخصائص والوظائف الموجودة في الفئة العليا داخل الفئة الفرعية، وإنما تحصل عليها الفئة الفرعية بشكل تلقائي من الفئة العليا، وذلك بالإضافة إلى الخصائص والوظائف التي يتم تخصيصها في الفئة الفرعية.

تتشابه الوراثة في البرمجة مع الطريقة التي ورثنا بها سماتنا عن آبائنا، مثل لون العيون والشعر وغيرها، وكما نعلم فإن آباءنا أيضًا قد ورثوا بعض هذه السمات عن آبائهم التي ورثوها عن آبائهم وهكذا... يوضح الرسم التخطيطي التالي كيفية تنظيم التسلسل الهرمي الخاص بالفئة.



توزيع الفئات هرميًا

- Class A هي فئة عليا لـ Class B.
- Class B هي فئة فرعية من Class A.
- Class B هي فئة عليا لكل من C وD وE.
- Class C وD وE فئات فرعية من Class B.

كل فئة في مستوى أسفل في المخطط تصبح أكثر تخصيصًا لغرض معين.

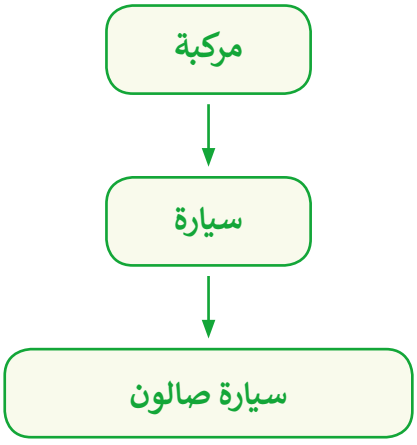
لنلقِ نظرةً على كيفية تطبيق مبدأ الوراثة في مثال المركبة.

مثال

لنفترض أن لدينا فئة عليا باسم **Vehicle** (مركبة)، وفئتين فرعيتين متخصصتين هما **Car** (السيارة) و **Bicycle** (الدراجة الهوائية)، واللتان ترثان الخصائص والوظائف من فئة المركبة.

بالمثل، يمكن أن يكون للفئة الفرعية السيارة Car فئتان فرعيتان، وهما السيارة الصالون، والسيارة الرياضية المكشوفة، واللتان سترثان بدورهما خصائص ووظائف فئة السيارة، وكذلك فئة المركبة.

تستخدم الوراثة عمومًا عندما يكون لدينا ترتيب هرمي للكائنات من العام إلى الأكثر تحديدًا.



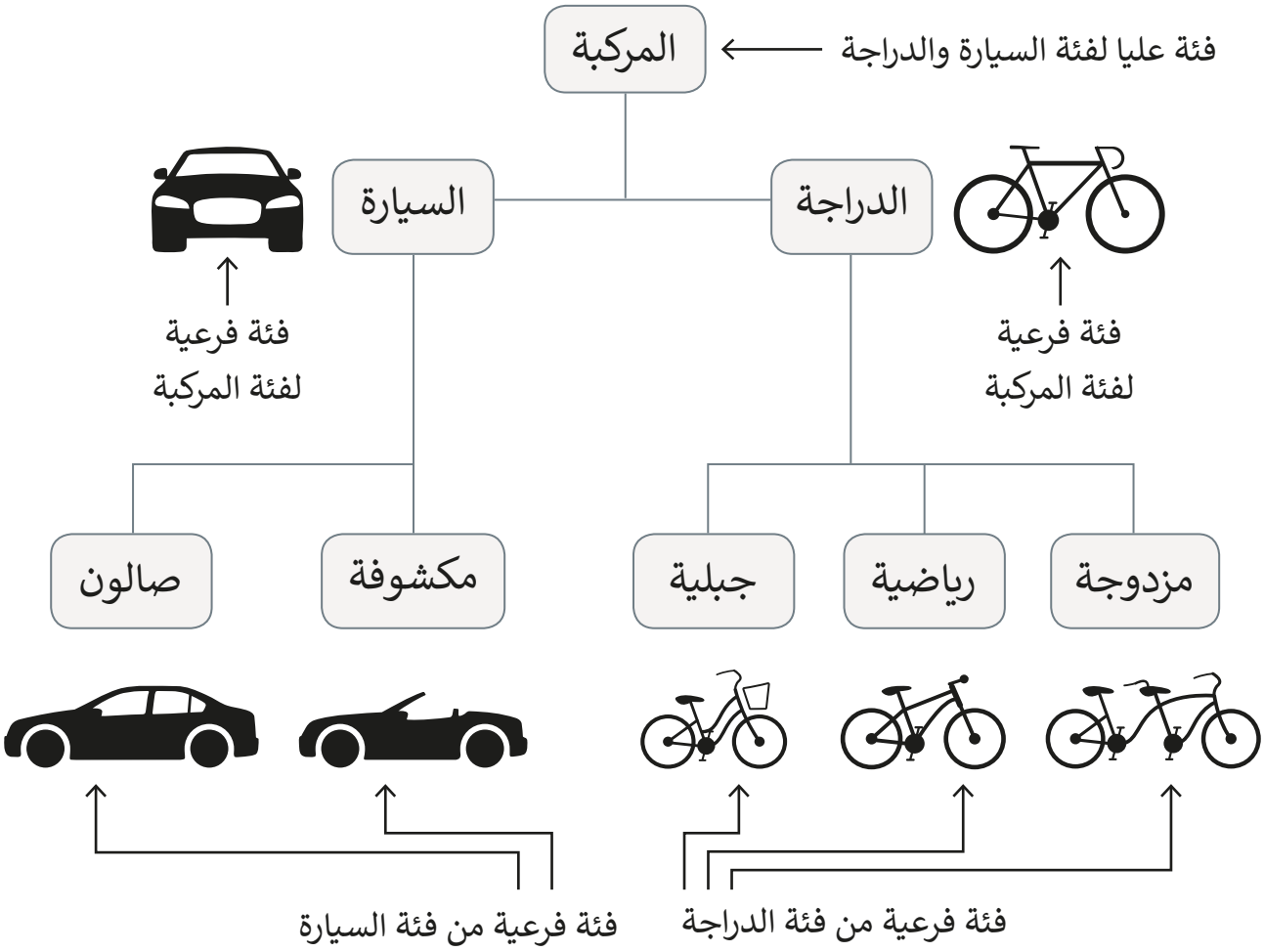
إذا درسنا العلاقة بين الفئات المتتالية (**Successive Classes**) سنرى أن كل فئة فرعية (**Subclass**) هي فئة عليا (**Superclass**) للتالية كالتالي:

- السيارة الصالون هي سيارة (**car**).
- السيارة (**car**) عبارة عن مركبة (**vehicle**).

لا تطبق هذه العلاقة على الفئة العليا المباشرة فقط بل على كل الفئات العليا.

- السيارة الصالون هي سيارة (**car**).
- السيارة الصالون هي أيضًا مركبة (**vehicle**).

بهذه الطريقة يتم إنشاء تسلسل هرمي يمكن التعبير عنه في مثال السيارة السابق بالمخطط التالي.



تتضح مزايا الوراثة في البرمجة إذا نظرنا إلى الفئات الفرعية للفئة في المثال أعلاه، فالفئات الفرعية للسيارة والدراجات تشتركان في بعض الميزات فكلاهما لديه عجلات ومقاعد ويمكنهما التحرك للأمام وللخلف بسرعة معينة.

في البداية نحدد الخصائص المشتركة لهذه الفئات في فئة المركبة العليا ثم نورثها ونخصصها إلى الفئات الفرعية. لذلك يمكن أن تحتوي السيارة أيضًا على مساحة للأمتعة، ومحرك، وأربع عجلات ثابتة وقد تحمل أكثر من راكب واحد، كما يمكننا أن نقوم بتشغيل وإيقاف محركها.

إن استخدامنا لمفهوم الوراثة في البرمجة يوفر علينا الكثير من الوقت والجهد من خلال إعادة استخدام المقطع البرمجي بسهولة مما يزيد من دقة وموثوقية البرنامج.

استخدام الوراثة في البرمجة بلغة Python

لكي نعرِّف فئة فرعية في **Python** نكتب اسم الفئة الفرعية متبوعًا باسم الفئة العليا بين قوسين وذلك حسب الصيغة التالية:

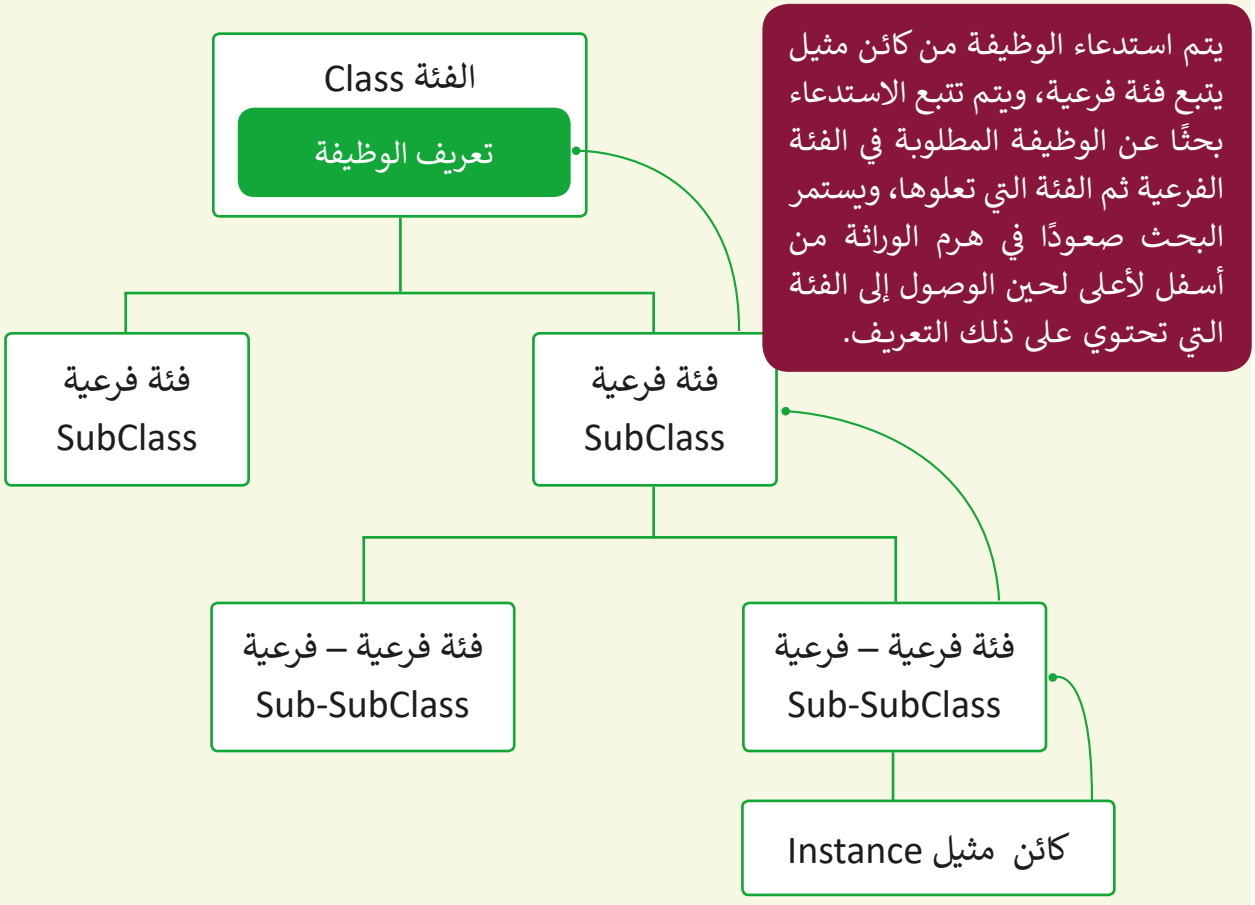
```
class SubclassName(SuperclassName)
```

حيث يمثل **SubclassName** اسم الفئة الفرعية التي نريد تعريفها، و **SuperclassName** اسم الفئة العليا التي سترث منها .

صيغة الوراثة في **Python**.

```
class SuperClass:
  Attributes and methods of the superclass

class SubClass(SuperClass):
  Body of the subclass
```

طريقة استدعاء الوظائف من الكائنات التابعة لفئات فرعية:



مثال 1:

في المثال التالي قمنا بتعريف الفئة الأساسية **BaseClass** ثم قمنا بعد ذلك بتعريف الفئة الفرعية **InheritingClass** التي ترث الخصائص والوظائف من الفئة **BaseClass**.

سننشئ مثيل (**Instance**) لفئة **InheritingClass** وسنستدعي الوظيفة **my_print()**.

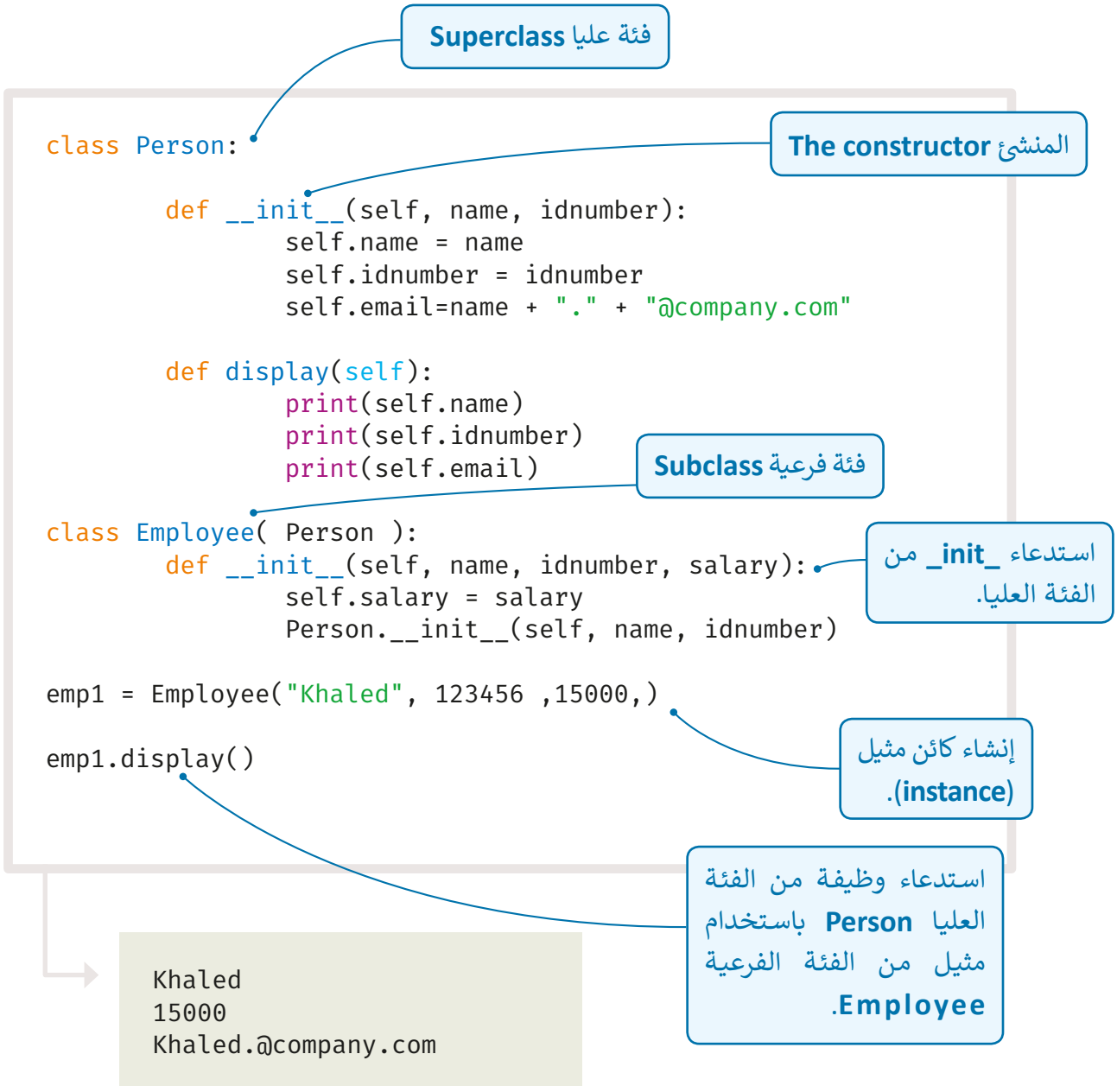
```
class BaseClass:
    def my_print1(self):
        print("Hello from Base class")

class InheritingClass(BaseClass):
    def my_print2(self):
        print("Hello from Sub class")

x = InheritingClass ()
x.my_print1()
x.my_print2()
```

```
Hello from Base class
Hello from Sub class
```

كما نرى فإن الفئة الفرعية يمكنها استخدام كافة خصائص ووظائف الفئة العليا، حيث استطاع الكائن المثيل x والذي يتبع الفئة الفرعية **InhertingClass** استخدام الوظيفة my_print1 والمعرفة أصلًا في في الفئة العليا **BaseClass**.

مثال 2:

في المثال التالي سنقوم بإنشاء فئة عليا بمسمى **Person** وفئة فرعية ترث منها الخصائص والوظائف باسم **Employee**، وسنقوم باستخدام بعض الخصائص والوظائف الخاصة بالفئة العليا من خلال الفئة الفرعية.




```
class Person:

        def __init__(self, name, idnumber):
                self.name = name
                self.idnumber = idnumber
                self.email=name + "." + "@company.com"

        def display(self):
                print(self.name)
                print(self.idnumber)
                print(self.email)

class Employee( Person ):
        def __init__(self, name, idnumber, salary):
                self.salary = salary
                Person.__init__(self, name, idnumber)

emp1 = Employee("Khaled", 123456 ,15000,)

emp1.display()
```

```
Khaled
15000
Khaled.@company.com
```

كما رأينا في المثال فإن جميع خصائص ووظائف فئة **Person** متاحة في فئة **Employee**.

أنواع الوراثة Types of Inheritance

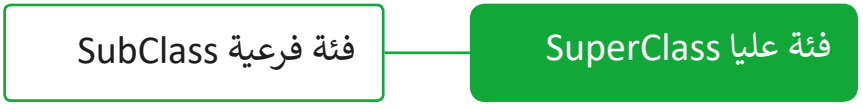
يوجد أربعة أنواع من الوراثة:

- الوراثة الأحادية Single Inheritance.
- الوراثة المتعددة Multiple Inheritance.
- الوراثة متعددة المستويات Multi-level Inheritance.
- الوراثة الهرمية Hierarchical Inheritance.

الوراثة الأحادية
Single Inheritance

في هذا النوع من الوراثة يتم اشتقاق فئة فرعية واحدة من فئة عليا واحدة.

ألقِ نظرة على المقطع البرمجي التالي.



مثال 3:

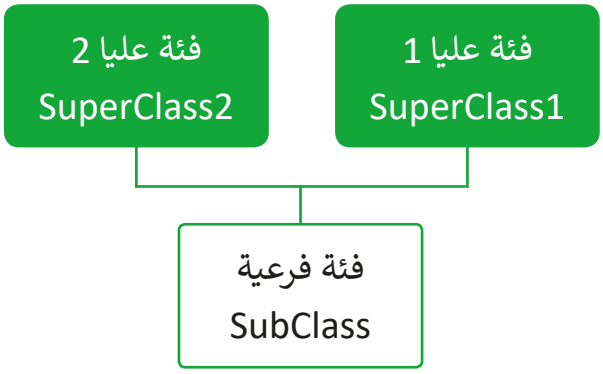
```
class SuperClass:

    def feature_1(self):
        print("feature_1 from SupertClass is running")
    def feature_2(self):
        print("feature_2 from SuperClass is running")

class SubClass(SuperClass):
    def feature_3(self):
        print("feature_3 from SubClass is running...")

obj = SubClass()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```

```
feature_1 from SuperClass is running
feature_2 from SuperClass is running
feature_3 from SubClass is running
```

الوراثة المتعددة
Multiple Inheritance

في هذا النوع من "الوراثة" يتم اشتقاق فئة فرعية من فئتين رئيستين أو أكثر.

ألقِ نظرة على المقطع البرمجي التالي.



مثال 4:

```
class SuperClass_1:

    def feature_1(self):
        print("feature_1 from SuperClass_1 is running")

class SuperClass_2:

    def feature_2(self):
        print("feature_2 from SuperClass_2 is running")

class SubClass(SuperClass_1, SuperClass_2):

    def feature_3(self):
        print("feature_3 from ChildClass is running")

obj = SubClass()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```

```
feature_1 from SuperClass_1 is running
feature_2 from SuperClass_2 is running
feature_3 from SubClass is running
```

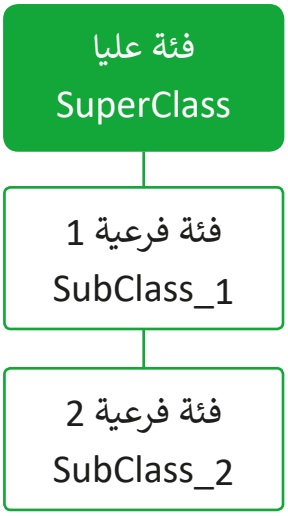
نصيحة ذكية

تعتبر الجافا بمثابة اللغة القياسية للبرمجة كائنية التوجه، ولكنها مع ذلك لا تدعم الوراثة المتعددة، لذا لا يفضل استخدام الوراثة المتعددة بشكل عام.

الوراثة متعددة المستويات
Multi-Level Inheritance

في هذا النوع من الوراثة يتم اشتقاق الفئة الفرعية من فئة فرعية أخرى تم اشتقاقها من فئة عليا.

ألق نظرة على المثال التالي:



مثال 5:

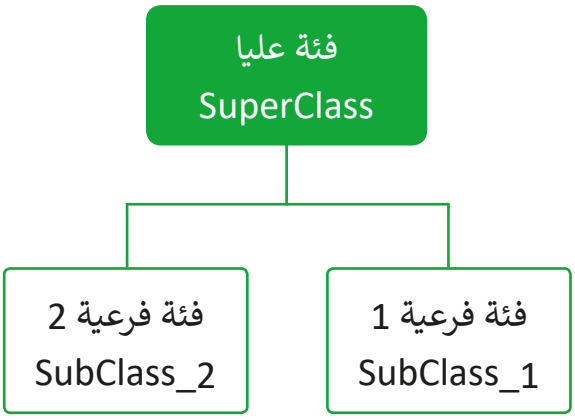
```
class SuperClass:

    def feature_1(self):
        print("feature_1 from SuperClass is running")

class SubClass_1(SuperClass):

    def feature_2(self):
        print("feature_2 from SubClass_1 is running")


class SubClass_2(SubClass_1):

    def feature_3(self):
        print("feature_3 from SubClass_2 is running")

obj = SubClass_2()
obj.feature_1()
obj.feature_2()
obj.feature_3()
```

```
feature_1 from SuperClass is running
feature_2 from SubClass_1 is running
feature_3 from SubClass_2 is running
```

الوراثة الهرمية Hierarchical Inheritance

في هذا النوع من الوراثة يتم اشتقاق فئتين فرعيتين أو أكثر من فئة عليا واحدة.

ألقِ نظرة على المثال التالي.



مثال 6:

```
class SuperClass:

    def feature(self):
        print("feature from SuperClass is running")

class SubClass_1(SuperClass):
    def feature_1(self):
        print("feature_1 from SubClass_1 is running")

class SubClass_2(SuperClass):
    def feature_2(self):
        print("feature_2 from SubClass_2 is running")

obj1 = SubClass_1()
obj2 = SubClass_2()
obj1.feature_1()
obj1.feature()
obj2.feature_2()
obj2.feature()
```

```
feature_1 from SubClass_1 is running
feature from SuperClass is running
feature_2 from SubClass_2 is running
feature from SuperClass is running
```

مزايا استخدام الوراثة في البرمجة

يساهم استخدام مبدأ الوراثة في البرمجة كائنية التوجه في زيادة جودة وموثوقية البرنامج المنتج، حيث يمكن من إعادة استخدام التعليمات البرمجية من خلال المزايا التي يوفرها والمتمثلة في:

1. يمكن تعريف الوظيفة التي نريد توريثها لفئات متعددة في الفئة العليا، مما يسمح لنا بإعادة استخدامها في جميع الفئات الفرعية.
2. تنعكس جميع التغييرات التي نقوم بها على الفئة العليا بشكل تلقائي على جميع فئاتها الفرعية، لذلك فأنت لا تحتاج إلى القيام بالتغييرات في المستويات الدنيا، حيث يتم ذلك تلقائيًا من خلال استقبال التغييرات الجديدة وفق مفهوم الوراثة.

1

ضع علامة ✓ أمام العبارة الصحيحة وعلامة ✗ أمام العبارة الخطأ.

.1	يسمح مبدأ الوراثة بتعريف فئة ترث كافة الوظائف والخصائص من فئة أخرى.	
.2	يمكن أن يكون لكل فئة عليا فئتين فرعيتين.	
.3	ترث الفئات العليا خصائص ووظائف الفئات الفرعية.	
.4	لتعريف فئة فرعية في Python نكتب: .class SuperclassName (SubclassName)	
.5	يساعد مبدأ الوراثة في عملية إعادة استخدام التعليمات البرمجية ويساهم في جودة وموثوقية البرنامج المنتج.	
.6	تنعكس التغييرات التي تجريها على فئة فرعية تلقائياً في الفئة العليا الخاصة بها.	

2

اذكر المزايا التي يقدمها استخدام مبدأ الوراثة في البرمجة.

_____

_____

_____

_____

_____

_____

3

أي نوع من الوراثة يتم تطبيقه في كل برنامج أدناه.

```
class Job:
    def __init__(self, person_name):
        self.name = person_name

    def task(self):
        print("working")

class Teacher(Job):
    def task(self):
        print("teach students")

teacher1 = Teacher("Khaled")
teacher1.task()
```

_____

```
class Dad:
    def __init__(self):
        self.eye_color = "blue"
        self.hair_color = "black"
        self.city = "Amsterdam"

    def swim(self):
        print("I can swim")

class Mum:
    def __init__(self):
        self.eye_color = "brown"
        self.hair_color = "brown"
        self.city = "Amsterdam"

    def dance(self):
        print("I can dance")

class Kid(Dad, Mum):
    def __init__(self):
         # Calling constructors of Mum class
        Mum.__init__(self)

kid = Kid()
print(kid.eye_color)
```

_____

```
class Parent:
    def __init__(self):
        self.eye_color = "blue"
        self.hair_color = "black"
        self.city = "Amsterdam"

    def swim(self):
        print("I can sing")

class Child1(Parent):
    pass
class Child2(Parent):
    pass

kid = Child2()
print(kid.eye_color)
```

_____

```
class Parent:
    def __init__(self):
        self.eye_color = "blue"
        self.hair_color = "black"
        self.city = "Amsterdam"

    def swim(self):
        print("I can sing")

class Child(Parent):
    def __init__(self):
        self.eye_color = "brown"
        self.hair_color = "brown"
        self.city = "Amsterdam"

    def dance(self):
        print("I can dance")

class GrandChild(Child):
    def __init__(self):
         # Calling constructors of Parent
class
        Parent.__init__(self)

kid = GrandChild()
print(kid.eye_color)
kid.dance()
```

_____

4

نفذ الخطوات الآتية:

لدينا فئة Shape مع السمات x و y والوظائف perimeter و area.

> قم بتحديد فئة Square جديدة ترث من فئة Shape.

> قم بإنشاء كائن مثيل للفئة Square، ثم قم باستدعاء وظائف الفئة العليا من هذا الكائن Shape.

في هذه الحالة، أعد تعريف وظيفة __init__ للشكل بحيث تكون قيمتي x و y متماثلتين دائمًا.

```
class Shape:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def area(self):
        print( self.x * self.y)

    def perimeter(self):
         print ( 2 * self.x + 2 * self.y)
```

_____

_____

_____

_____

_____

الدرس الرابع

مبادئ البرمجة كائنية التوجه (2)

تعرفنا في الدرس السابق على المبادئ الأساسية للبرمجة كائنية التوجه والمقصود بالوراثة وكيفية استخدامها. سننتقل الآن لنتعرف المبادئ الأخرى للبرمجة كائنية التوجه.

التجريد Abstraction

يُستخدم التجريد لتبسيط التصميمات المعقدة داخل المقاطع البرمجية، ويمكن للمبرمج بواسطة تقنيات التجريد إظهار التفاصيل التي تهم المستخدم وإخفاء التفاصيل الأخرى غير الضرورية للتعامل مع المقطع البرمجي، وذلك لتقليل التعقيد وتحسين كفاءة البرنامج.

لكي ندرك مفهوم التجريد بسهولة سنتناول أحد الأمثلة من حياتنا الواقعية.

عند قيادتنا للسيارة فلا يوجد داعٍ لمعرفة ما يدور داخل محرك السيارة وأجزائها الداخلية، فما يهمنا فقط هو كيفية التفاعل مع السيارة عبر وحداتها المختلفة الظاهرة لنا مثل عجلة القيادة ودواسة الفرامل ودواسة الوقود. تكون هذه الوحدات ظاهرة لنا وتعتبر بمثابة "الواجهات" التي نتعامل معها كمستخدمين لتشغيل وظائف السيارة دون معرفة الآلية الداخلية للتشغيل مثل تفاصيل عمل المحرك وبقية أنظمة السيارة. وبالتالي يمكننا القول بأن المعرفة التي لدينا عن السيارة كمستخدمين هي "معرفة مجردة"، تسمح لنا بقيادتها من خلال واجهاتها فقط.

في لغة Python يمكننا إنشاء الفئات المجردة، وهي فئات استثنائية لا يمكن إنشاء أي كائن مثيل لها، تحتوي الفئة الاستثنائية كذلك على وظائف مجردة لا تتضمن أي إجراءات، وإنما يتم تعريف إجراءات هذه الوظائف المجردة في فئات فرعية ترث خصائص ووظائف الفئة المجردة.

يطلق على الفئة التي تحتوى وظيفة مجردة اسم الفئة المجردة **Abstract Class**.

سيتضح التطبيق البرمجي للتجريد من خلال الأمثلة المقبلة، ولإنشاء فئة مجردة في Python ينبغي أولًا استيراد الفئة الجاهزة ABC من الوحدة abc، ترث أي فئة مجردة يتم إنشاؤها خصائص ووظائف الفئة ABC كما يلي:

```
from abc import ABC
Class ClassName(ABC):
```

مثال 1:

```
from abc import ABC

class Employee(ABC):
    def calculate_salary(self,monthlySalary):
        pass

    def daysofwork(self, monthsworked):
        workingdays = 21 * monthsworked
        return workingdays

class Designer(Employee):
    def calculate_salary(self,monthlySalary):
        finalmonthlySalary = monthlySalary
        return finalmonthlySalary

class Developer(Employee):
    def calculate_salary(self,monthlySalary):
        finalmonthlySalary = monthlySalary * 1.10
        return finalmonthlySalary

# a designer who has worked for 2 months
emp_1=Designer()
print("The Designer worked for", emp_1.daysofwork(2), "days.")
print("His salary is", emp_1.calculate_salary(7000*2))

# a developer who has worked for 2 months
emp_2=Developer()
print("The Developer worked for", emp_2.daysofwork(2),"days.")
print("His salary is", emp_2.calculate_salary(8000*2))
```

"**ABC**" هي وحدة قياسية (**Module**) يتم استيرادها عند تعريف فئة مجردة في برامج **Python**.

هذه هي الوظيفة المجردة داخل الفئة وهي لا تحتوي أي إجراءات، عوضًا عن ذلك يتم استخدام الأمر **pass** للانتقال إلى الأمر البرمجي التالي. يتم تعريف إجراءات الوظيفة المجردة في الفئة الفرعية التي ترث من الفئة المجردة.

```
The Designer worked for 42 days.
His salary is 14000
The Developer worked for 42 days.
His salary is 17600.0
```

تحتوي الفئة المجردة في مثالنا أعلاه على وظيفة مجردة وهي calculate_salary، وطبقًا لمبدأ التجريد لا تحتوي هذه الوظيفة على أي تعليمات برمجية، وإنما يتم تعريف تعليماتها في فئات فرعية ترث هذه الوظيفة وتحدد إجراءاتها، وهي الفئات **Designer** و **Developer**.

كما ترى في ناتج تنفيذ البرنامج السابق فإننا رفعنا الدخل الشهري للمطور **Developer** بنسبة 10% ليصبح 17600. باستخدام الوراثة يمكننا توريث جميع الخصائص وتأدية المهام المطلوبة.

ومن الجدير بالذكر أن الوراثة تعمل جنبًا إلى جنب مع بقية مباديء البرمجة كائنية التوجه، وهذا ما سيتضح في شرح بقية المباديء.

التغليف Encapsulation

يقصد بالتغليف القدرة على إخفاء الفئة **(Class)** لبياناتها الخاصة عن باقي البرنامج مع التأكد أنه يمكن الوصول إليها من خلال الوظائف العامة **(Public Methods)**.

تكمن أهمية التغليف في حماية خصائص الفئة وتحديد إمكانية الوصول إليها لتكون حصرية لبعض الوظائف المضمنة إما في الفئة نفسها فقط أو الفئات التي ترث خصائصها وسلوكها. يحافظ مبدأ التغليف على أمن البيانات المخزنة في الخصائص المحمية للفئة ويمنع تعديلها بشكل مباشر إلا من قبل الوظائف المخولة بذلك.

لفهم مبدأ التغليف سنتناول مثالًا من حياتنا الواقعية.

لفهم فكرة التغليف **Encapsulation** من الأفضل رؤية مثال حقيقي من الحياة اليومية. توجد في المدرسة أقسام مختلفة مثل الأقسام الأكاديمية والإدارية والمالية. يتعامل قسم المالية مع جميع المعاملات المالية المتعلقة بالمدفوعات للمدرسين والموردين. وبالمثل يتعامل القسم الأكاديمي مع جميع الأنشطة الأكاديمية. إذا احتاج شخص من القسم الأكاديمي إلى بيانات مالية فلا يُسمح له بالوصول المباشر إلى هذه البيانات، وإنما يجب أن يحصل على طلبه من خلال المرور عبر وسيط مخول له الوصول إلى البيانات المالية.

سنقوم في المثال الآتي بتعريف فئة تمثل موظفًا وتضم في خصائصها بيانته اسمه والراتب الذي يتقاضاه، وسندرس حالات تغليف هذه البيانات.

معدِّلات الوصول Access Modifiers

يتم القيام بعملية بالتغليف في البرمجة كائنية التوجه باستخدام ما يسمى معدِّلات الوصول **Access Modifiers**، والتي تمكن من تعديل حالة "تغليف" خصائص الفئة ووظائفها (والتي تكون عامة **Public** في الأساس) إما بجعلها محمية **Protected** أو خاصة **Private**، لنلق نظرة على هذه الحالات وكيفية تمثيلها في لغة **Python** في المثال التالي.

مثال 2:

في هذا المثال يتم تعريف الفئة Employee واستخدام معدلات الوصول لتغيير حالة خصائصها وصلاحيات الوصول إليها كالآتي:

- الحالة العامة **Public**: وتعني أن الوصول إلى الفئة والبيانات والوظائف متاح لجميع الكائنات.

```
class Employee:
    def __init__(self, name, salary):
        self.name=name
        self.salary=salary

emp1=Employee("Khaled",15000)
print(emp1.name)
```

جميع الخصائص والوظائف في الفئة المعرفة بلغة **Python** تكون عامة (**Public**) بشكل افتراضي، مما يتيح الوصول إلى جميع خصائص فئة **Employee**.

```
Khaled
```

- الحالة المحمية **Protected**: تعني أن الوصول متاح فقط للوظائف المعرفة في الفئة نفسها أو في الفئات التي ترثها.

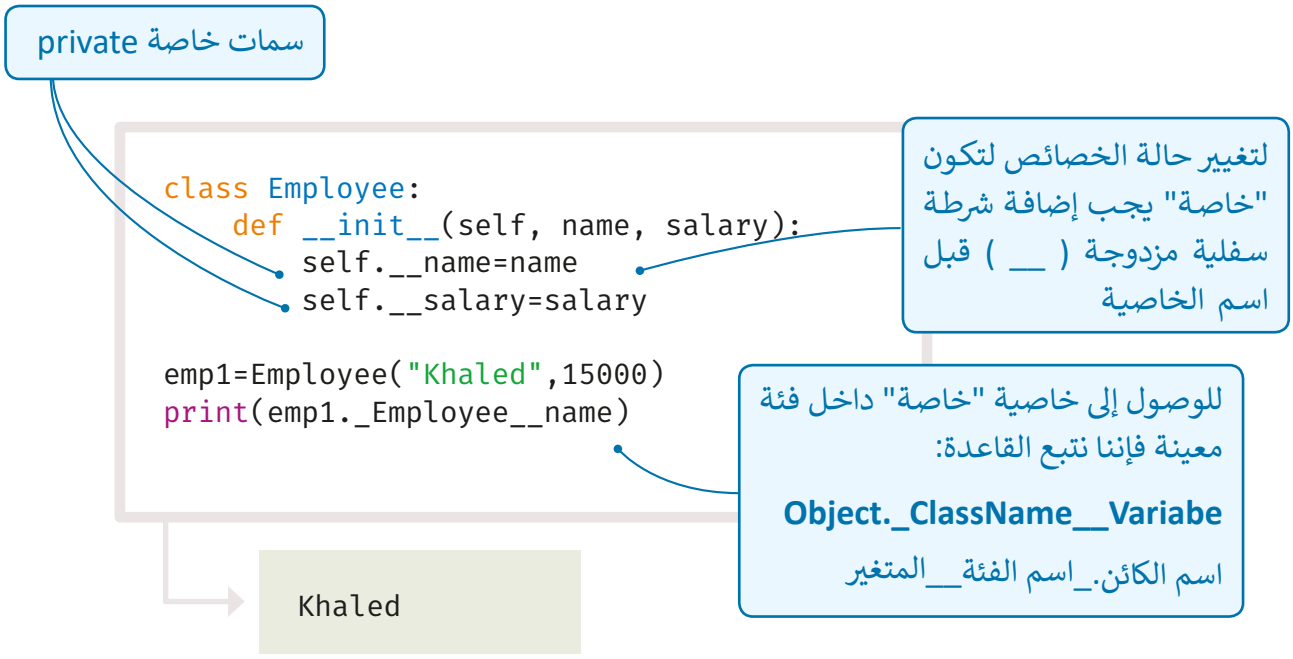
خصائص محمية

```
class Employee:
    def __init__(self, name, salary):
        self._name=name
        self._salary=salary

emp1=Employee("Khaled",15000)
print(emp1._name)
```

لجعل الخصائص "محمية" يجب إضافة شرطة سفلية مفردة (_) قبل اسم الخاصية.

```
Khaled
```

- الحالة الخاصة **Private:** تعني أن الوصول للبيانات والوظائف مقيد بفئة محددة فقط.



تنبع الحاجة الحقيقية لاستخدام التغليف في البرمجة من ضرورة حماية خصائص ووظائف كل فئة من التعليمات البرمجية الخارجية. على المطور الذي يكتب تعليمات الفئة البرمجية ويختبرها أن يتأكد من أن أي مطور آخر يمكنه فقط أن ينشئ مثيلات ويستدعي وظائف ويمرر القيم إلى معاملاتها، وذلك دون التأثير مباشرة على البيانات الخاصة أو المحمية للفئة.

توجد ميزتان رئيسيتان لاستخدام التغليف:

1. صلاحيات الوصول (التحكم) .

- يُستخدم الكائن فقط من خلال وظائفه.
- لا يسمح بتغيير بيانات الكائن مباشرة من كائن آخر أو من مقطع برمجي خارج مقطع الفئة أو الفئات الفرعية التي ترث منها.

2. يسمح بالعمل الجماعي على نفس البرنامج وعمل عدة مبرمجين بشكل متزامن.

- يمكن لعدة مبرمجين كتابة المقطع البرمجي في فئات متعددة دون تأثير عملهم على أعمال الآخرين.
- يمكن للمبرمجين استخدام وظائف الفئات دون الحاجة للوصول إلى تعليماتها البرمجية، مما يزيد من فاعلية فريق المبرمجين ويحسن جودة وتصميم البرنامج.

لنلقي نظرة على مثال أكثر تعقيدًا لنفهم أنه لا يمكنك الوصول إلى السمة الخاصة وتغييرها.

```
class Employee:
    def __init__(self, name, salary):
        self.__name=name
        self.__salary=salary

    def calculate_bonus(self):
        self.bonus = self.__salary * 0.10

emp1=Employee("Khaled",15000)

# salary is a private attribute
# and cannot be accessed by mistake
emp1.salary = 30000
print(emp1.salary)
emp1._salary = 40000
print(emp1._salary)

emp1.__salary = 50000
print(emp1.__salary)

emp1.calculate_bonus()
print(emp1.bonus)
```

Salary سمة خاصة ولا يمكن الوصول إليها بالخطأ

يتم احتساب Bonus على أساس القيمة الأصلية للراتب. يأخذ القيمة التي قمنا بتعيينها عند تهيئة المثيل.

```
30000
40000
50000
1500.0
```

تعدد الأشكال Polymorphism

يتم تطبيق مبدأ تعدد الأشكال عندما يتم تعريف وظيفة في فئة فرعية تحمل نفس اسم وظيفة أخرى موجودة في فئة أخرى تعلوها في هيكل الوراثة.

يتضح مفهوم تعدد الأشكال في البرمجة كائنية التوجه في قدرة مترجمات أو مفسرات هذه اللغة على تحديد الوظائف المناسبة التي يتم استدعاؤها بشكل تلقائي حسب نوع الكائن الذي يستدعيها أو وفق نوع وعدد المعاملات.

مثال 3:

لنلق نظرة على المثال التالي، والذي يوضح كيفية عمل تعدد الأشكال والوراثة.

يحدد تعدد الأشكال (**polymorphism**) في **Python** الوظائف الخاصة بالفئة الفرعية والتي توجد بذات الاسم لوظائف في الفئة العليا.

إن وظيفة "**flight**" "الطيران" المُعرَّفة في المثال التالي موجودة في فئة عليا وفئات فرعية أيضًا. نقوم بتعرف الوظيفة "**flight**" في الفئتين الفرعيتين "**Bird**" " (طائر) و "**Mammal**" (ثدييات) لأن الوظيفة الموروثة من فئة الأصل "**Animal**" (حيوان) لا تتناسب تمامًا مع الفئة الفرعية، لذلك تعين علينا إعادة تعريف الوظيفة في الفئة الفرعية.

إن تعريف وظيفة في الفئة بنفس اسم الوظيفة الموروثة من الفئة العليا (superclass) يسمى overriding (تراكب).

```
class Animal:
     def intro(self):
       print("There are different types of
animals")

     def flight(self):
       print("Many animals can fly but some cannot")

class Bird(Animal):
     def flight(self):
       print("Birds can fly")

class Mammal(Animal):
     def flight(self):
       print("The only mammal that can fly is the
bat")

obj1 = Animal()
obj2 = Bird()
obj3 = Mammal()

obj1.intro()
obj1.flight()

obj2.flight()

obj3.flight()
```

المقطع البرمجي لوظيفة intro هو ذاته للفئات Animal وBird وMammal.

المقطع البرمجي لوظيفة flight لفئة Bird (طائر) يختلف عن مقطع هذه الوظيفة في الفئة العليا Animal (حيوان).

```
There are different types of animals
Many animals can fly but some cannot
Birds can fly
The only mammal that can fly is the bat
```

يعتبر مبدأ تعدد الأشكال ذو فائدة كبيرة في البرمجة نظرًا للآتي:

- يتم كتابة المقطع البرمجي والفئات مرة واحدة ويمكن استخدامها وتنفيذها لاحقًا عدة مرات.
- تساعد في تقليل الاقتران بين الوظائف المختلفة وسلوك الكائنات.

تذكَّر أنه إذا لم يكن للفئة الفرعية وظيفة مُعرّفة بمقطع برمجي جديد، فسيتم تنفيذ الوظيفة التي تحمل نفس اسم فئتها العليا إذا كانت موجودة.

1

عرف كل من التجريد والتغليف وتعدد الأشكال؟ لماذا نستخدم هذه المباديء في OOP؟

2

ضع علامة ✓ أمام العبارة الصحيحة وعلامة ✗ أمام العبارة الخطأ.

.1	يمكن للمبرمج إظهار التفاصيل التي تهم المستخدم وإخفاء التفاصيل الأخرى للتقليل من مستوى تعقيد البرنامج وتحسين كفائته وذلك باستخدام تقنيات التغليف.	
.2	تسمى الفئة التي تحتوي على وظيفة مجردة أو أكثر بالفئة المجردة.	
.3	يتحكم التغليف بالوصول إلى خصائص الفئة.	
.4	جميع الخصائص والوظائف في الفئات المعرفة بلغة Python تكون عامة بشكل افتراضي.	
.5	يمكن الوصول إلى الخصائص المحمية للفئة من داخل الفئة وكذلك من الفئات الفرعية الخاصة بها.	
.6	يمكن إنشاء كائن مثيل من الفئة المجردة واستدعاء وظائفها من خلاله.	
.7	يتضمن مبدأ تعدد الأشكال تعريف وظيفة في فئة فرعية تحمل نفس اسم وظيفة أخرى موجودة في فئة عليا.	

3

اذكر حالة تغليف الخصائص والوظائف المدرجة في المقاطع البرمجية أدناه والتي تم تعديلها باستخدام معدلات الوصول.

```
class Teacher:
    def __init__(self, name, subject):
        self._name=name
        self._subject=subject
    def task(self):
        print("The teacher teaches",self._subject)

teacher1 = Teacher("Khaled","Math")
teacher1.task()
```

_____

```
class Teacher:
    def __init__(self, name, subject):
        self._name=name
        self._subject=subject
    def task(self):
        print("The teacher teaches",self._subject)

teacher1 = Teacher("Khaled","Math")
print(teacher1._name)
teacher1.task()
```

_____

```
class Teacher:
    def __init__(self, name, subject):
        self.__name=name
        self.__subject=subject
    def task(self):
        print("The teacher teaches",self.__subject)

teacher1 = Teacher("Khaled","Math")
print(teacher1._Teacher__name)
teacher1.task()
```

_____

4

أكمل البرنامج التالي لتغيير السعر من 1200 إلى 1000. هل هذا التغيير ممكن؟ قم بتعديل المقطع البرمجي بشكل مناسب بحيث يمكنك تغيير قيمة السعر.

```
class Computer:

    def __init__(self):
        self.__price = 1200

    def sell(self):
        print("Selling Price:", self.__price)
```

5

قم بتشغيل البرنامج التالي. هل يعمل بطريقة صحيحة؟ ابحث عن الخطأ وقم بإصلاحه.

```
class Employee(ABC):
    def calculate_salary(self,monthlySalary):
        pass

    def daysofwork(self, monthsworked):
        workingdays = 21 * monthsworked
        return workingdays

class Teacher(Employee):
    def calculate_salary(self,monthlySalary):
        finalmonthlySalary = monthlySalary
        return finalmonthlySalary


# a designer who has worked for 2 months
emp_1=Teacher()
print("The Teacher worked for", emp_1.daysofwork(3), "days.")
print("His salary is", emp_1.calculate_salary(10000*3))
```

6

أنشأنا في البرنامج التالي فئة Shark والتي تحدد وظائف swim() و skeleton(). قم بإنشاء فئة أخرى تسمى Fish لها وظيفتان بنفس الاسم مثل فئة Shark ولكن كل ومهمة من ومهام هذه الوظيفة تختلف عن فئة Shark. قم بإنشاء مثيل لهذه الفئات في كائنين واستدعاء الوظائف.

```
class Shark():
    def swim(self):
        print("The shark is swimming.")

    def skeleton(self):
        print("The shark's skeleton is made of cartilage.")
```

_____

_____

_____

_____

_____

_____

_____

_____

```
fish1 = Shark()
fish1.skeleton()
fish1. swim()

fish2 = Fish()
fish2.skeleton()
fish2.swim()
```

```
The shark's skeleton is made of cartilage.
The shark is swimming.
The fish's skeleton is made of bone.
The fish is swimming.
```

الدرس الخامس

تطبيق على مبادئ البرمجة كائنية التوجه

ميزات البرمجة كائنية التوجه

الآن وبعد أن تعرفنا على أهم مبادئ البرمجة كائنية التوجه الرئيسة، أصبح لدينا فكرة عامة لما يمكن أن تقدمه البرمجة كائنية التوجه للمبرمج، ويمكن تلخيص ذلك فيما يلي:

- النمطية/النمذجة (**Modularity**): يمكن تصميم مقطع برمجي لفئة واحدة وتطويره واختباره بشكل مستقل عن التعليمات البرمجية الخاصة بالفئات الأخرى، وبذلك فإن تنفيذ فئة معينة لن يؤثر على التعليمات البرمجية لفئات أخرى.
- إخفاء المعلومات (**Hiding Information**): في البرمجة كائنية التوجه ينصب اهتمامنا على الوظائف العامة (**Public Methods**) أكثر من كيفية التنفيذ الداخلي لكل فئة.
- إعادة استخدام المقطع البرمجي (**Code Reuse**): يتم الأخذ بالاعتبار عند تصميم الفئة أنه يمكن استخدامها بواسطة فئات أخرى، وذلك من خلال توظيف مفهوم الوراثة (**Inheritance**) في البرمجة.
- قابلية التوسع (**Extensibility**): يمكن توسيع أي فئة من خلال الوراثة إلى فئات أكثر تخصيصًا.
- تشخيص الأخطاء بسهولة أكثر (**Easier Debugging**): يمكن القيام بعملية تشخيص الأخطاء أولًا على مستوى الفئة وذلك من خلال ما يسمى باختبار الوحدة (**Unit Testing**). كما أن التغييرات على فئة واحدة لا يعني بالضرورة تغيير أجزاء البرنامج الأخرى التي تستخدم تلك الفئة.
- التصميم (**Design**): تعتبر مرحلة التصميم مرحلة مهمة جدًا في البرمجة كائنية التوجه، حيث أن التصميم الجيد يعني سهولة الصيانة وسهولة تطوير البرنامج، وعلى النقيض فإن التصميم السيء يؤدي لمشاكل تظهر في مرحلة التطوير.

لفهم معنى وميزات البرمجة كائنية التوجه سنقوم بتحويل التعليمات البرمجية الخاصة بالمتجر الإلكتروني الذي كتبناه سابقًا في **Python** إلى البرمجة كائنية التوجه.

إنشاء متجر إلكتروني e-shop بواسطة البرمجة كائنية التوجه

في الوحدات السابقة أنشأنا متجرًا الكترونيًا يختص بتذكارات وهدايا كأس العالم باستخدام **Python**. سنستخدم الآن الأفكار والمبادئ والتقنيات التي تعرفنا عليها في البرمجة كائنية التوجه لنجري التغييرات على المتجر الإلكتروني. سيسهل هذا الأمر من صيانة البرنامج وسيمنحنا قابلية التوسع في البرمجة.

لنلقِ نظرةً على التعليمات البرمجية السابقة التي أنشأناها في **Python** لنتذكر المتغيرات والدوال، والمنطق الذي تم اعتماده في برنامج المتجر الإلكتروني.

```
# --- IMPORTS --- #

import datetime
from datetime import date

# --- GLOBAL VARIABLES --- #

Discount = False
PriceSum = 0

# --- CART --- #

Cart = []

# --- PRODUCTS --- #

MetroGoldCard = {
    "name": "Metro Gold Card",
    "description": "Plastic rechargeable metro card. Move easily and
fast from and to all football stadiums.",
    "price": 10.0,
    "weight": 0.1
}
CeramicTeaCup = {
    "name": "Ceramic Tea Cup ",
    "description": "Dishwasher and microwave safe.",
    "price": 12.0,
    "weight": 0.8
}

SoccerBall = {
    "name": "Soccer Ball",
    "description": "This durable soccer ball is ideal for perfecting
your skills with a design that stands out.",
    "price": 28.0,
    "weight": 0.7
}
```

```
Pin = {
    "name": "Pin",
    "description": "Handmade pins with a glossy vinyl finish. Pins
are typically pinned on fabric such as backpacks, "
                   "purses and clothing.",
    "price": 3.0,
    "weight": 0.1
}

BaseballCap = {
    "name": "Baseball Cap",
    "description": "This classic baseball cap completes your look.
Ideal for sunny days, it has a soft fabric that "
                   "protects you from the sun.",
    "price": 20.0,
    "weight": 0.3
}

TShirt = {
    "name": "T-Shirt",
    "description": "This comfortable T-shirt is made of thick cotton
and comes to complete your style.",
    "price": 35.0,
    "weight": 0.2
}

Products = [MetroGoldCard, CeramicTeaCup, SoccerBall, Pin,
BaseballCap, TShirt]


# --- FUNCTIONS --- #

def GetDate():
    global Discount
    Today = date.today()
    Now = datetime.datetime.now()
    Day = Now.strftime("%A")
    Date = Today.strftime("%B %d, %Y")

    if Day == "Thursday":
        Discount = True

    print("Today is " + Day + " " + Date)

    if Discount:
        print("\nYou are eligible for a %20 discount on all
items\n")
```

```
def DisplayMenu():
    print("Your options are : \n")
    print("1 --> Display products")
    print("2 --> Display cart")
    print("3 --> Clear cart")
    print("4 --> Checkout")
    print("5 --> Exit\n")

    CheckInput()

def CheckInput():
    while True:
        try:
            Selection = int(input("What would you like to do? "))
            if Selection < 1 or Selection > 5:
                raise ValueError
            break
        except ValueError:
            print("Please enter a number between 1 and 5.\n")
        except:
            print("Something went wrong.\n")
    HandleInput(Selection)

def HandleInput(Selection):
    if Selection == 1:
        DisplayProducts()
    elif Selection == 2:
        DisplayCart()
    elif Selection == 3:
        ClearCart()
    elif Selection == 4:
        Checkout()
    elif Selection == 5:
        Exit()
```

```
def DisplayProducts():
    Index = 1
    print("\n||---- All Items ----||\n")

    for Item in Products:
        print(str(Index) + ". " + str(Item.get("name")))
        Index += 1

    while True:
        try:
            Selection = int(input("\nFor which product do you want
more details : "))
            if Selection < 1 or Selection > 6:
                raise ValueError
            break
        except ValueError:
            print("Please enter a number between 1 and 6.\n")
        except:
            print("Something went wrong.\n")

    if Selection == 1:
        DetailedDescription(Products[0])
    elif Selection == 2:
        DetailedDescription(Products[1])
    elif Selection == 3:
        DetailedDescription(Products[2])
    elif Selection == 4:
        DetailedDescription(Products[3])
    elif Selection == 5:
        DetailedDescription(Products[4])
    elif Selection == 6:
        DetailedDescription(Products[5])
    else:
        DisplayMenu()
```

```
def DisplayCart():
    global PriceSum

    print("\n||---- Your cart contains the following items----||\n")

    Index = 1

    for Item in Cart:
        PriceSum += Item.get("price")
        print(str(Index) + ". " + str(Item.get("name")) + " : " +
str(Item.get("price")))
        Index += 1

    if Discount:
        PriceSum *= 0.8

    print("\nThe total price for all the items in the cart is : " +
str(PriceSum) + "$\n")
    while True:
        try:
            Answer = input("Would you like to proceed to checkout?
(y/n) ")
            if Answer != "y" and Answer != "n":
                raise ValueError
            break
        except ValueError:
            print("Please enter y for yes or n for no.\n")
        except:
            print("Something went wrong.\n")

    if Answer == "y":
        Checkout()
    elif Answer == "n":
        DisplayMenu()
```

```
def DisplayProducts():
    Index = 1
    print("\n||---- All Items ----||\n")

    for Item in Products:
        print(str(Index) + ". " + str(Item.get("name")))
        Index += 1

    while True:
        try:
            Selection = int(input("\nFor which product do you want
more details : "))
            if Selection < 1 or Selection > 6:
                raise ValueError
            break
        except ValueError:
            print("Please enter a number between 1 and 6.\n")
        except:
            print("Something went wrong.\n")

    if Selection == 1:
        DetailedDescription(Products[0])
    elif Selection == 2:
        DetailedDescription(Products[1])
    elif Selection == 3:
        DetailedDescription(Products[2])
    elif Selection == 4:
        DetailedDescription(Products[3])
    elif Selection == 5:
        DetailedDescription(Products[4])
    elif Selection == 6:
        DetailedDescription(Products[5])
    else:
        DisplayMenu()
```

```
def ClearCart():
    global Cart
    global PriceSum

    Cart = []
    PriceSum = 0
    print("\n||---- Your cart has been cleared ----||\n")
    DisplayMenu()


def Checkout():
    if PriceSum > 0.0:
        Details = EnterDetails()
        ShippingMethod = EnterShipping()
        print("\n||---- " + Details + " has paid " + str(PriceSum) +
"$ ----||\n")
        Exit()
    else:
        print("\n||---- Your cart is empty! ----||")
        DisplayMenu()

def EnterDetails():
    Details = input("\nYour first and last name : ")
    return Details

def EnterShipping():
    ShippingMethod = input("\nYour shipping company : ")
    return ShippingMethod
```

```
def DetailedDescription(Product):
    print("\n*** " + str(Product.get("name")) + " ***\n")
    print("Description : " + str(Product.get("description")))
    print("Price : " + str(Product.get("price")))
    print("Weight : " + str(Product.get("weight")))

    while True:
        try:
            Answer = input("Would you like to add this item to the
cart? (y/n) ")
            if Answer != "y" and Answer != "n":
                raise ValueError
            break
        except ValueError:
            print("Please enter y for yes or n for no.\n")
        except:
            print("Something went wrong.\n")

    if Answer == "y":
        AddToCart(Product)
    elif Answer == "n":
        DisplayMenu()

def AddToCart(Product):
    Cart.append(Product)
    print("\n*** " + str(Product.get("name")) + " added to the cart
***\n")
    DisplayProducts()

def Exit():
    print("\n||---- Thank you for shopping at the World Cup Shop!
----||\n")

# --- MAIN PROGRAM --- #
print("\n||---- Welcome to the World Cup Shop! ----||\n")
GetDate()
DisplayMenu()
```

لقد أنشأنا ثلاث فئات جديدة: **Product** (المنتج) و **Shop** (المتجر) و **Cart** (سلة التسوق).

المتجر Shop

لتنظيم التعليمات البرمجية للمتجر الإلكتروني، سننشئ فئة خاصة بالمتجر باسم **Shop** تحتوي على الوظائف التي تنشئ الرسائل التي نحتاجها عند تفاعل المستخدم مع متجرنا الإلكتروني.

هذه الفئة لا تمتلك أي خصائص **(Properties)**، وعليه فهي لا تحتوي على مُنشئ **(Constructor)**، وهي مستقلة عن أي معلومات في البرنامج. وهذا يعني أن كل وظيفة في هذه الفئة تُعرَّف كوظيفة ثابتة **(static method)** وذلك بوضع وسم **@staticmethod** أعلى التعريف.

المنتج Product

ستعتبر المنتجات الموجودة في المتجر الإلكتروني مثيلات **(Instances)** للكائن **Product**. يحتوي **Product** على المعلومات المطلوبة التي يحتاجها مستخدم المتجر الإلكتروني مثل: **Name** (الاسم)، **Description** (الوصف)، **Price** (السعر)، **Weight** (الوزن).

سلة التسوق Cart

تُعرِّف فئة **Cart** الوظائف التي تشكل المنطق الأساسي لتطبيقنا، حيث ستؤدي العمليات والحسابات للقوائم المختلفة مثل قائمة المنتجات وسلة التسوق.

أصبح لدينا الآن فكرة عامة عما تحتويه التعليمات البرمجية، فلنبدأ بالبرمجة خطوةً بخطوة.

لنبدأ أولًا باستيراد الوحدات، وتعريف المتغيرات العامة التي يحتاجها البرنامج، وضبط قيمها الأولية.

```
# --- IMPORTS --- #

import datetime
from datetime import date

# --- GLOBAL VARIABLES --- #

discount = False
priceSum = 0
shoppingCart = []
```

استيراد وحدة **datetime** القياسية لعرض التاريخ الحالي.

سنستخدم هذه المتغيرات العامة لاحقًا، ولكن الآن سنقوم بتعيين قيمها الأولية.

تعريف قائمة بدون قيم تمثل سلة التسوق.

Shop

تعريف فئة Shop

ترجع وظيفة **getDate()** اليوم والتاريخ.

ترجع وظيفة **strftime()** مقطعًا يمثل التاريخ والوقت باستخدام **date** (التاريخ) و **time** (الوقت) أو كائن **datetime** (اليوم والوقت).

اسم اليوم كاملًا

اسم الشهر كاملًا

```
class Shop():
    # Class Static Methods

    @staticmethod
    def getDate():
        global discount
        today = datetime.date.today()
        now = datetime.datetime.now()
        day = now.strftime("%A")
        date = today.strftime("%B %d, %Y")

        if day == "Thursday":
            discount = True

        print("Today is " + day + " " + date)

        if discount:
            print("\nYou are eligible for a %20 discount
on all items\n")
```

```
Today is Thursday October 01, 2020

You are eligible for a 20% discount on all items
```

```
@staticmethod
def displayMenu():
    print("Your options are : \n")
    print("1 --> Display products")
    print("2 --> Display cart")
    print("3 --> Clear cart")
    print("4 --> Checkout")
    print("5 --> Exit\n")

    Shop.checkInput()
```

يمكن أن تكون وظيفة **displayMenu** ثابتة (**static**) نظرًا لعدم استدعائها من أي مثيل (**instance**) لكائن.

هذه هي القائمة الرئيسة للمتجر الإلكتروني.

```
Your options are :

1 --> Display products
2 --> Display cart
3 --> Clear cart
4 --> Checkout
5 --> Exit
```

لدينا خمسة خيارات للاختيار من القائمة، عندما نقوم بتشغيل البرنامج في بيئة برمجية مثل VSC فكل ما علينا هو كتابة رقم كل خيار. على سبيل المثال إذا أردنا عرض المنتج فعلينا كتابة "1" والضغط على Enter.

وظيفة checkinput هي وظيفة ثابتة لفئة Shop مثل displayMenu.

تساعد جملة تشخيص الأخطاء "try" في التأكد من صحة إدخال البيانات.

```
    @staticmethod
    def checkInput():
        while True:
            try:
                selection = int(input("What would you like to do?
"))
                if selection < 1 or selection > 5:
                    raise ValueError
                break
            except ValueError:
                print("Please enter a number between 1 and 5.\n")
            except:
                print("Something went wrong.\n")

        Shop.handleInput(selection)
```

```
Your options are :

1 --> Display products
2 --> Display cart
3 --> Clear cart
4 --> Checkout
5 --> Exit
```

```
What would you like to do? 1
```

```
    @staticmethod
    def displayProducts():
        index = 1
        print("\n||---- All Items ----||\n")

        for item in products:
            print(str(index) + ". " + str(item.
getName()))
            index += 1

        while True:
            try:
                selection = int(input("\nFor which
product do you want more details : "))
                if selection < 1 or selection > 6:
                    raise ValueError
                break
            except ValueError:
                print("Please enter a number between 1
and 6.\n")
            except:
                print("Something went wrong.\n")
```

تعرض وظيفة displayProducts قائمة بأسماء جميع المنتجات وتتحقق من إدخال المستخدم لاختيار خطأ.

```
||---- All Items ----||

1. Metro Gold Card
2. Ceramic Tea Cup
3. Soccer Ball
4. Pin
5. Baseball Cap
6. T-Shirt

For which product do you want more details :
```

```
if selection == 1:
    products[0].detailedDescription()
elif selection == 2:
    products[1].detailedDescription()
elif selection == 3:
    products[2].detailedDescription()
elif selection == 4:
    products[3].detailedDescription()
elif selection == 5:
    products[4].detailedDescription()
elif selection == 6:
    products[5].detailedDescription()
else:
    Shop.displayMenu()
```

استدعاء وظيفة **detailedDescription()** من فئة **Product** للحصول على المزيد من المعلومات عن المنتج الذي قام المستخدم باختياره.

```
||---- All Items ----||

1. Metro Gold Card
2. Ceramic Tea Cup
3. Soccer Ball
4. Pin
5. Baseball Cap
6. T-Shirt

For which product do you want more details :
For which product do you want more details : 6

*** T-Shirt ***

Description : This comfortable T-shirt is made of thick
cotton and comes to complete your style.
Price : 35.0
Weight : 0.2
Would you like to add this item to the cart? (y/n)
```

```
    @staticmethod
    def exit():
        print("\n||---- Thank you for shopping at the
World Cup Shop! ----||\n")

    @staticmethod
    def handleInput(selection):
        if selection == 1:
            Shop.displayProducts()
        elif selection == 2:
            Cart.displayCart()
        elif selection == 3:
            Cart.clearCart()
        elif selection == 4:
            Cart.checkout()
        elif selection == 5:
            Shop.exit()
```

هذه دالة عامة تتعامل مع خيار المستخدم من القائمة الرئيسة للتطبيق.

يتم استدعاء الوظائف من فئتي **Cart** أو **Shop** وفق اختيار المستخدم.

Product

تعريف فئة Product.

```
class Product():
    def __init__(self, name, description, price, weight):
        self.name = name
        self.description = description
        self.price = price
        self.weight = weight

    def getName(self):
        return self.name
    def getDescription(self):
        return self.description
    def getPrice(self):
        return self.price
    def getWeight(self):
        return self.weight
```

إنشاء مثيل للكائن وتعريف خصائصه.

وظائف الوصول لخصائص الكائن

تتعامل هذه الوظيفة مع وظائف الوصول السابقة لإعطاء المستخدم جميع المعلومات المتعلقة بالمنتج المحدد.

```
    # Class Methods
    def detailedDescription(self):
        print("\n*** " + self.getName() + " ***\n")
        print("Description : " + self.getDescription())
        print("Price : " + str(self.getPrice()))
        print("Weight : " + str(self.getWeight()))

        while True:
            try:
                answer = input("Would you like to add this item to
the cart? (y/n) ")
                if answer != "y" and answer != "n":
                    raise ValueError
                break
            except ValueError:
                print("Please enter y for yes or n for no.\n")
            except:
                print("Something went wrong.\n")

        if answer == "y":
            Cart.addProduct(self)
        elif answer == "n":
            Shop.displayMenu()
```

نسأل المستخدم عن رغبته بإضافة هذا المنتج إلى سلة التسوق، نتحقق من الإجابة ثم نضيف المنتج إلى سلة التسوق أو نرجع إلى القائمة الرئيسة.

```
Would you like to add this item to the cart?
(y/n) "y"
Please enter y for yes or n for no.

Would you like to add this item to the cart?
(y/n) y

*** T-Shirt added to the cart ***
```

تعريف فئة **Cart**.

نستخدم هنا المتغيرات العامة **priceSum** و **shoppingCart** لطباعة أسماء وأسعار جميع المنتجات الموجودة داخل سلة التسوق.

```
class Cart():
    # Class Methods
    def displayCart():
        global priceSum
        global shoppingCart

        print("\n||-- Your cart contains the following items--||\n")

        index = 1

        for item in shoppingCart:
            priceSum += item.getPrice()
            print(str(index) + ". " + item.getName() + " : " +
str(item.getPrice()))
            index += 1

        if discount:
            priceSum *= 0.8

        print("\nThe total price for all the items in the cart is :
" + str(priceSum) + "$\n")
```

نتحقق أيضًا من إتاحة الخصم **discount**، حيث يتم تغيير السعر النهائي وفقًا لذلك.

```
||-- Your cart contains the following items--||

1. T-Shirt : 35.0

The total price for all the items in the cart is :
28.0$
```

```
        while True:
            try:
                answer = input("Would you like to proceed to
checkout? (y/n) ")
                if answer != "y" and answer != "n":
                    raise ValueError
                break
            except ValueError:
                print("Please enter y for yes or n for no.\n")
            except:
                print("Something went wrong.\n")

        if answer == "y":
            Cart.checkout()
        elif answer == "n":
            Shop.displayMenu()
```

نتحقق من إدخال المستخدم ثم نذهب إلى الدفع.

```
Would you like to proceed to checkout? (y/n) y

Your first and last name : Khaled Mohammed

Your shipping company : Comp

||---- Khaled Mohammed has paid 28.0$ ----||


||---- Thank you for shopping at the World Cup Shop! ----||
```

```
    def addProduct(Product):
        global shoppingCart

        shoppingCart.append(Product)
        print("\n*** " + Product.getName() + " added to the cart
***\n")
        Shop.displayProducts()
```

نضيف مثيلاً لكائن **Product** إلى المتغير العام (**global variable**) المسمى **shoppingCart** ونضيفه إلى نهاية القائمة. بعد ذلك نعرض قائمة المنتجات المتاحة في سلة التسوق.

```
    def clearCart():
        global priceSum
        global shoppingCart

        shoppingCart = []
        priceSum = 0
        print("\n||---- Your cart has been cleared ----||\n")
        Shop.displayMenu()
```

هنا نفرغ قائمة سلة التسوق ونعطي المتغير **priceSum** القيمة **0**.

```
    def checkout():
        if priceSum > 0.0:
            details = Cart.enterDetails()
            shippingMethod = Cart.enterShipping()
            print("\n||---- " + details + " has paid " +
str(priceSum) + "$ ----||\n")
            Shop.exit()
        else:
            print("\n||---- Your cart is empty! ----||")
            Shop.displayMenu()
```

نحصل على معلومات من المستخدم عن اسمه وشركة الشحن التي ستتولى الشحن، ثم يتم عرض رسالة الدفع وأخيرًا يتم الخروج من البرنامج.

```
    def enterDetails():
        details = input("\nYour first and last name : ")
        return details
```

هذه هي الوظائف التي تحصل على المعلومات من المستخدم لعرضها لاحقًا في رسالة الدفع.

```
    def enterShipping():
        shippingMethod = input("\nYour shipping company : ")
        return shippingMethod
```

ننشئ كل منتج في المتجر الإلكتروني من خلال إنشاء كائن مثيل للفئة Product مع البيانات المناسبة لهذا المنتج، ويتم تكرار هذه العملية مع جميع المنتجات في متجرنا الإلكتروني.

```
metroGoldCard = Product("Metro Gold Card", "Plastic rechargeable
metro card. Move easily and fast from and to all football
stadiums.", 0.1 ,10.0)

ceramicTeaCup = Product("Ceramic Tea Cup", "Dishwasher and microwave
safe.", 0.8 ,12.0)

soccerBall = Product("Soccer Ball", "This durable soccer ball is
ideal for perfecting your skills with a design that stands out.",
0.7 ,28.0)

pin = Product("Pin", "Handmade pins with a glossy vinyl finish.
Pins are typically pinned on fabric such as backpacks, purses and
clothing.", 0.1 ,3.0)

baseballCap = Product("Baseball Cap", "This classic baseball cap
completes your look. Ideal for sunny days, it has a soft fabric that
protects you from the sun.", 0.3 ,20.0)

tShirt = Product("T-Shirt", "This comfortable T-shirt is made of
thick cotton and comes to complete your style.", 0.2 ,35.0)

products = [metroGoldCard, ceramicTeaCup, soccerBall, pin,
baseballCap, tShirt]
```

نقوم بتحديث قائمة المنتجات products [] بجميع المثيلات من فئة Product.

```
print("\n||---- Welcome to the World Cup Shop! ----||\n")

Shop.getDate()
Shop.displayMenu()
```

تم إنشاء المقطع البرمجي النهائي وفق الأفكار العامة لمبادئ البرمجة كائنية التوجه، وهكذا يكون من السهل القيام بالتغييرات اللازمة وتوسيع المتجر الإلكتروني وتحسين وظائفه.

1

للحفاظ على أموالك في أحد البنوك، تحتاج إلى فتح حساب مصرفي عن طريق إيداع مبلغ أولي من المال. الوظائف الأكثر أهمية للتعامل مع الحساب المصرفي هي إيداع وسحب الأموال. فكر في الحساب البنكي ككائن ثم قم بتصميم فئة BankAccount. ما هي الخصائص والوظائف الضرورية؟

2

عرف فئة BankAccount من التمرين السابق. عرف الخصائص والوظائف الخاصة بها.

```
class BankAccount():
    def
```

3

أنت مسؤول عن سلامة أرصدة الحسابات المصرفية. هل تحتاج إلى كتابة تعليمات برمجية للتحقق من العمليات غير الصالحة؟ فكر في الأخطاء المحتملة للعملاء أو المطورين.

4

نحن بحاجة إلى إنشاء فئة لإدارة معاملات الحساب المصرفي. تحتفظ بكل حركة بالتاريـخ والمبلغ والملاحظات. أكمل التعليمات البرمجية لتعريف الفئة.

```
class Transaction():
    # create a transaction
    # each transaction has an amount, a date and notes
    def __init__(self, amount, date, notes):
        self.
        self.
        self.
```

5

اكتب التعليمات البرمجية لوظيفة السحب من فئة BankAccount للتحقق من سحب المبالغ التي هي أكبر من رصيد الحساب المتوفر. يمكنك استخدام حركات الحساب لحساب الرصيد المتوفر.

```
# withdraw an amount of money
    def makeWithdrawal(self, amount, date, note):
        # if the amount is not positive do not allow the
transaction
        if amount <= 0:
```

مشروع الوحدة

العنوان: Bank Account

الوصف: اكتب التعليمات البرمجية للتعامل مع حساب بنكي استناداً إلى التدريبات والمقاطع البرمجية التي قمت بإنشائها في هذه الوحدة. نفذ الآتي:

الأدوات: لغة برمجة بايثون Python.

خطوات التنفيذ:

توسيع التعليمات البرمجية لفتح حساب بنكي برمجياً عندما يدخل المستخدم اسمه ومبلغاً من المال الأولي.

إنشاء سلسلة من الحركات: الإيداعات والسحوبات ومتابعتها من خلال تقرير لكافة حركات الحساب البنكي.

استخدام الأدوات المناسبة للتحقق من عدم وجود أي مدخلات غير صالحة من المستخدم أو المطور.

كتابة المقاطع البرمجية المناسبة للتحقق من وظائف فتح الحساب، وإيداع، وسحب المبالغ المالية المدخلة بشكل غير صحيح.

ماذا تعلمت

تعلمت في هذه الوحدة:

> التعرف على الميزات الرئيسة للبرمجة كائنية التوجه OOP.

> إنشاء كائنات برمجية وفئات.

> تحديد الاختلاف ما بين السمات Attributes الوظائف Methods.

> التعرف على أهمية الوراثة (Inheritance) في البرمجة كائنية التوجه OOP.

> استخدام المبادئ والمميزات الرئيسة للبرمجة كائنية التوجه OOP لإنشاء مشروع.

المصطلحات

الدرس 1	برمجة كائنية التوجه Object-oriented programming	كائن Object	فئة Class
	المنشيء Constructor	مثيل Instance	
الدرس 2	السمة Attribute	الوظيفة Method	
الدرس 3	الوراثة Inheritance	الوراثة الأحادية Single Inheritance	الوراثة المتعددة Multiple Inheritance
	الوراثة متعددة المستويات Multi-Level Inheritance	الوراثة الهرمية Hierarchical Inheritance	
الدرس 4	التجريد Abstraction	التغليف Encapsulation	تعدد الأشكال Polymorphism
الدرس 5	متجر إلكتروني E-shop	عربة التسوق Cart	

الصف	الصفحة	الملاحظة

الصف	الصفحة	الملاحظة

علوم الحاسب المستوى الثاني عشر المسار التكنولوجي
الفصل الدراسي الثاني 2021 - 2020
كتاب الطالب

تم النشر بواسطة: **دار النشر MM Publications**
www.mmpublications.com
info@mmpublications.com
المكاتب
المملكة المتحدة، الصين، قبرص، اليونان، كوريا، بولندا، تركيا، الولايات المتحدة الأمريكية، الشركات المنتسبة والممثلين في جميع أنحاء العالم.

تم الإنتاج في الاتحاد الأوروبي

الرقم الدولي المعياري للكتاب: 978-618-05-5246-1 N2009002021-18113