مقدمة في لغة Java

د. محي الدين مراد

تحدثنا في المحاضرة السابقة عن بعض المفاهيم الأساسية في البرمجة غرضية التوجه مثل الـ design time الذي يتم تحديد بناء الـ class فيه وأنواع الـ data members وأنواع الـ behaviors (methods) والتي تكون مغلفة encapsulated ضمن هذا الـ class.

تتمثل الـ OOP بخمسة مفاهيم رئيسية:

1- التجريد (Abstraction):

مفهوم التجريد هو عملية تنفيذ مهمة ما دون معرفة كيفية عملها و الدخول في تفاصيلها , مثال بسيط :
نفرض لدينا هاتف محمول و نريد إرسال رسالة , ف كل ما نريد معرفته هو زر إرسال الرسالة لا أكثر ولا يهمنا طريقة إرسال هذه الرسالة أو العمليات التي تمر بها .
لذا يمكننا اعتبار الصف المجرد نموذجاً عاماً فلا نستطيع إنشاء object منه لأن ليس لديه معلومات حول كيفية تنفيذ المهام بل نقوم بتوريثه و عمل override للتوابع المجردة الموجودة داخله وبهذا فإننا نتيح لكل ابن من أبناء هذا الصف أن يضيف التفاصيل الخاصة به.

2- التغليف (Encapsulation):

أي لا يمكن أن تتواجد أي data أو behaviors خارج Scope لـ class معين.
كما ذكرنا سابقا فإن الـ data members تنقسم إلى data members (class و object) وكذلك الأمر بالنسبة لـ methods.

3- إعادة استعمال الغرض (Object Reusability): وتتمثل في التركيب (composition) والوراثة (inheritance).

4- تعددية الأشكال (Polymorphism).

5- Later Binding.

الباني (Constructor):

هو عبارة عن تابع لا يقوم برد اي قيمة سواء كانت int, char, ... يتم فيه إعطاء القيم الأولية للـ object data members أي ما نسميه الـ (initialization) على خلاف الـ class data members التي يتم الـ

initialization لها ضمن الـ class نفسه لأنه يمكن استخدامها قبل بناء object، يتم استدعاؤه مرة واحدة عند كل عملية instantiate (خلق) لـ object.

في حال عدم تعريف أي constructor صراحة ضمن الـ class فإنه يتم استدعاء الـ default constructor.
يمكن عمل overloading لهذا التابع مثل أي تابع آخر، وذلك بغرض بناء مجموعة من الـ objects بأشكال مختلفة .

مثال على استعمال الـ constructor:

```
Class Car{
    public Car( ) { }
    public Car(int x) { this( ); }
    public Car(int x, int y) { this(5); }
}
class Main{
public static void main(){
    Car c = new Car(10,20);
}
```

- **توضيح:**

قمنا بتعريف **reference** هو **Car c** يتم حجز عنوان له بالذاكرة قبل ربطه بـ **object**(يكون عنوانه محجوز سواء تم ربطه بـ **object** أم لا).

قمنا ببناء **object** (Car c = new Car(10,20);) يبقى محجوزاً في الذاكرة طوال فترة استخدامه ضمن الـ Scope الذي يتم استعماله فيه حاليا، وينتهي الـ life time الخاص بهذا الـ object عند الخروج من هذا الـ Scope ليصبح مكانه في الذاكرة متاحاً للاستعمال من جديد (وذلك في حالة تفعيل عمل الـ garbage collector).

this keyword: هي عبارة عن كلمة مفتاحية تقوم برد reference على الـ object الحالي الذي يتم التعامل معه.

في هذه الحالة عندما ترد ضمن constructor فإنها تقوم باستدعاء constructor آخر (يكون حسب الـ arguments التي يتم ارسالها للتابع)، كما في التابع الباني الثاني تم استعمال this وعدم تمرير اي قيم لها مما يؤدي إلى تطابق الاستدعاء مع الباني الأول، وكذلك بالنسبة للتابع الباني الثالث تم استعمال this مع تمرير قيمة int لها مما يؤدي إلى تطابق الاستدعاء مع الباني الثاني.

```
Class Car{
  public Car( ) { } ⟵
  public Car(int x) { this( ); } ⟵
  public Car(int x, int y) { this(5); }
}
```

مثال آخر على استعمال this:

```
Class Car{
    int gear;
    public void startEngine(int gear){
            this.gear = gear;
            shiftGear(this);
    }
    void shiftGear(Car c) { }
}
```

- **توضيح:**

this.gear: كما ذكرنا سابقا فإن this تقوم برد reference على الـ object الذي يتم التعامل معه حاليا، ففي هذه الحال this تكافئ instance من الـ class Car وبالتالي عند محاولة الوصول إلى العنصر gear باستخدام (.) فإنه سوف يتم التعامل مع gear التي تقع في Scope الـ class وليس Scope التابع الذي تم استدعاء this فيه، أما عند التعامل مع المتحول gear بشكل مباشر ضمن Scope التابع فإنه سوف يتم التعامل مع المتحول gear الخاص بهذا التابع.

shiftGear(this): إن التابع يقوم بأخذ reference على object من نوع Car وبما أن الـ object الذي نتعامل معه حاليا من هذا النوع قمنا باستخدام this.

- **تلخيص:**

this keyword: هي عبارة عن كلمة مفتاحية تقوم برد reference على الـ object الحالي الذي يتم التعامل معه، يمكن استعمالها في ثلاث حالات:

a- استدعاء بواني أخرى ضمن الـ class نفسه وهنا أن يجب أن تكون الـ this أول تعليمة ضمن جسم الباني.

b- الوصول إلى object data members بدلا من متحولات التابع الذي يتم التعامل معه(والتمييز بين object data members والـ arguments).

c- الحصول على reference الـ object الحالي لاستعماله كوسيط لتابع ما.

(لمزيد من المعلومات يرجى مراجعة كتاب *Thinking in Java page:116*)

ملاحظة:

يمكن أن يكون التابع الباني public أو private ولكن عند استعمال الـ access modifier: private لا يمكن استدعاؤه بسبب عدم القدرة على الوصول إليه، ولكن يمكن أن يستعمل ضمن توابع بواني public أخرى في بناء الـ object الخاص بها.

أجيال لغات البرمجة:

1- **الجيل الأول:** مثل الـ binary code.
2- **الجيل الثاني:** مثل Assembled Languages.
3- **الجيل الثالث:** وتتمثل باللغات غرضية التوجه OOP مثل C#,C++,Java,....
4- **الجيل الرابع:** وتتمثل باللغات الغير إجرائية مثل SQL والتي تتعامل مع قواعد البيانات حيث يكون الجهد المبذول لتنفيذ أي عملية فيها (تخزين أو قراءة أو استعلام من ملف في الـ data base) فإنها تتم بسطر أو تعليمة برمجية واحدة (one statement).
5- **الجيل الخامس:** وهي الـ normal languages والتي تقترب في تنفيذها من التعلم البشري.

تصنيف لغات البرمجة حسب عملية الترجمة:

1- Assembled Languages:

يتم ترجمتها باستخدام برنامج يدعى بـ Assembler، حيث يقوم بتحويل كل تعليمة برمجية إلى سطر وحيد من الأصفار والواحدات (binary code)، ويقوم بترجمة البرنامج بشكل كامل قبل الوصول لمرحلة التنفيذ.

2- Interpreted Languages:

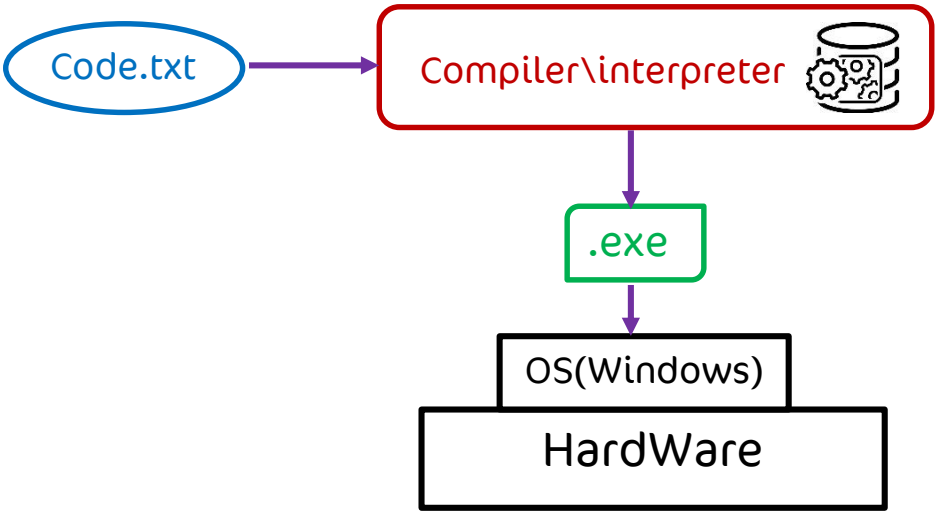
يتم ترجمتها باستخدام برنامج يدعى بـ Interpreter، حيث يقوم بجلب تعليمة برمجية وحيدة من البرنامج، ويقوم بتفحصها للتأكد من عدم وجود أخطاء (في حال وجود أخطاء تتوقف عملية الترجمة والتنفيذ)، ثم يقوم بترجمة التعليمة البرمجية إلى binary code، ويقوم مباشرةً بتنفيذ الـ binary code الخاص بهذه التعليمة، وتتكرر هذه العملية حتى الانتهاء من جميع التعليمات البرمجية أو وجود خطأ في أحد التعليمات، ونلاحظ أنّ عملية الترجمة والتنفيذ تتم معاً في حالة الـ interpreted languages.

3- Compiled Languages:

يتم ترجمتها باستخدام برنامج يدعى بـ Compiler، حيث يقوم بتفحص كامل البرنامج (الكود البرمجي) للتأكد من عدم وجود أخطاء (syntax errors)، ثم يقوم بتحويل جميع التعليمات إلى binary code، ويحتفظ بنسختين من البرنامج، نسخة الكود البرمجي، ونسخة الـ binary code، وبعدها يمكن تنفيذ البرنامج بعد الانتهاء كلياً من عملية الترجمة، ونلاحظ أنّ عملية التنفيذ تلي عملية الترجمة، حيث لا يمكن التنفيذ في حال عدم اكتمال عملية الترجمة أو وجود خطأ فيها.

من اللغات البرمجية التي تحتاج compiler لترجمتها C, C++, Java، حيث تقوم الـ compilers الخاصة بهذه اللغات بالتحسين على الكود البرمجي قبل التحويل إلى الـ binary code مما يؤدي إلى تحسين وقت التنفيذ.

وبعد الانتهاء من عملية البرمجة يصبح البرنامج قابل للتنفيذ من قبل نظام التشغيل الذي يتوافق معه الـ compiler (Windows, Linux, osX,....).

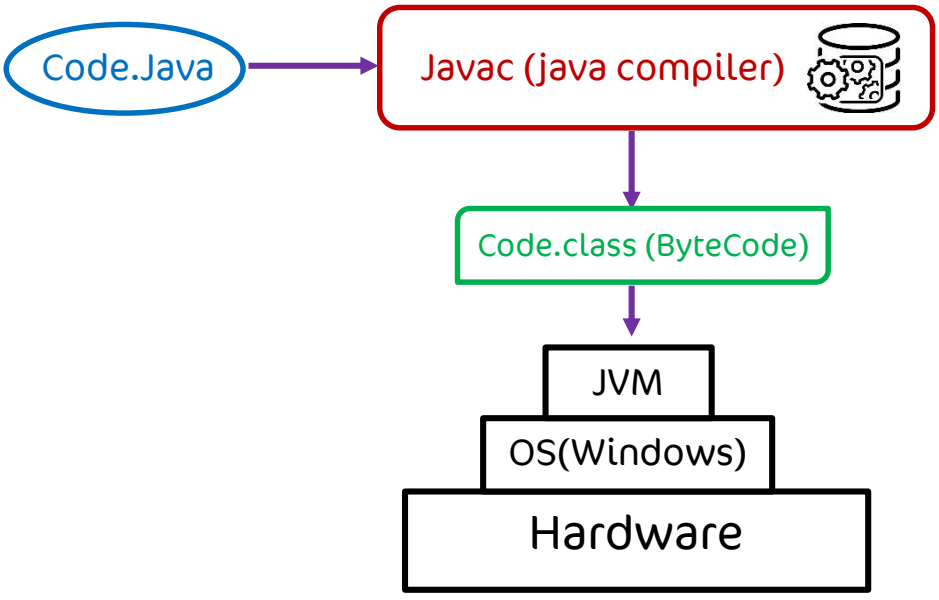


تعتبر لغة Java هي machine independent أي أنها لا تهتم بنوع الآلة التي تقوم بالتشغيل عليها وذلك لأن تنفيذ الـ code لا يقع على عاتق نظام التشغيل بل على Java virtual machine (JVM) وبالتالي عند تغيير نظام التشغيل نقوم بتغير الـ JVM التي تقوم بتنفيذ البرنامج على نظام التشغيل فقط.

فتكون مراحل التنفيذ بلغة Java كالتالي:

- يتم تخزين JAVA Code كملف نصي في ملف (.java)
- تتم ترجمة ملف (.java) إلى ملف (.class) يحمل نفس الاسم.
- ملف (.class) يحوي JAVA Bytecode (الـ Instruction Set الخاصة بالـ JVM).
- يتم تفسير الـ Bytecodes عند runtime

لذا يمكن اعتبار أن لغة Java هي interpreted and compiled language.



ملاحظات:

-يتم تجميع كل الـ classes في البرنامج ضمن ملف واحد مضغوط تكون لاحقته (.jar).

-Javadoc: هي عبارة عن أداة تقوم بتحويل كل التعليقات التي ترد ضمن الـ code اي ملف html واحد يمكن أن يتم استعماله كـ documentation للبرنامج أو المكتبة التي تم إنشاءها بلغة Java.

-تعتبر لغة java بطيئة وذلك بسبب التفسير المتكرر للتعليمات ضمن الحلقات أو للتوابع التي يمكن أن يتم استدعاءها أكثر من مرة(الـ code المتكرر تنفيذه) ، فبكل مرة سوف يتم أخذ هذا الـ code وإعادة تفسيره، لذا تم تقديم ما يدعى بـ Just-in-time compiler والذي يقوم بترجمة الـ code مرة واحدة لـ binary code ويحتفظ به للاستعماله في حالة تكرار تنفيذ هذا الـ code.

-لا تحتوي لغة Java على:

- ✓ Structures or Unions
- ✓ Typedefs, Defines or preprocessors
- ✓ goto statements.
- ✓ Pointers.
- ✓ Functions: (methods instead)
- ✓ Operator Overloading.
- ✓ Multiple Inheritance:(يمكن تجاوز ذلك من خلال الـ Interface)

Garbage Collector

إن من خصائص لغة Java هي الـ multithreading كما ذكرنا سابقا وعند عمل أكثر من thread بنفس الوقت يكون لكل واحدة منها أهمية ما (أولوية)، فيعمل الـ garbage collector كـ thread إلى جانب الـ thread الذي يتم تنفيذه حاليا (ويعتبر الأقل أولوية) وذلك بغرض تحرير اي مساحة لـ object تم الانتهاء من استعماله بعد انتهاء Scope هذا الاستعمال، فيقوم بجعل هذه المساحة قابلة للاستعمال من جديد.

قد لا يتم تفعيل الـ garbage collector خلال عمل البرنامج أبدا بسبب عدم وجود حالة تقوم بتفعيله.

ملاحظة:

بأخذ مثال المصفوفة والتي تكون عناصرها من 1 لـ مليون التي نقوم بتعبئتها باستخدام الـ multithreading التي وردت في المحاضرة السابقة، يكون لكل thread يقوم بتعبئة قسم ما أولوية، عند تساوي هذه الأولوية يتم الانتهاء من تعبئة كل الاقسام بنفس الوقت، أما عند إعطاء أولوية لـ thread ما أكثر من باقي الـ threads سوف يتم الانتهاء من تنفيذ هذا الـ thread قبل باقي الـ threads.

فعند تفعيل thread الـ Garbage Collector (والذي يعتبر الأقل أولوية) يكون عندها قد دخل الـ thread الرئيسي (thread تابع الـ main مثلا) في حالة pause لسبب ما (يقوم بتحميل ملف من server أو اي task منفصلة عن البرنامج ولكن يتوقف عمله عليها) عندها تصبح أولوية الـ Garbage Collector مناسبة ليتم تفعيله.

يمكن عمل إجبار للـ Garbage Collector أن يعمل وذلك عن طريق استعمال التابع: System.gc( );، والذي يقوم برفع أولوية الـ GC لتصبح الأولى.

التابع finalize( ):

يترافق استدعاء وتنفيذ هذا التابع مع تفعيل الـ GC، ولا يمكن ضمان استدعاؤه خلال التنفيذ وذلك بسبب عدم ضمان عمل الـ GC.

✓ **مثال:**

```
class Book {
    boolean checkedOut = false;
    Book (boolean checkOut) {
            checkedOut = checkOut;   }
    void checkIn () {
            checkedOut = false;    }
    public void finalize() {
            if (checkedOut)
                    System.out.println("Error: checked out");  }
}
public class BM {
    public static void main(String[ ] args) {
        Book novel = new Book(true);
        novel.checkIn();
        new Book(true);
        // Force garbage collection & finalization:
        System.gc( ); }
 }
```

- **توضيح:**

ضمن الـ class BM قمنا بإنشاء object من الـ class Book بدون أي reference لتشير إليه، عندها سوف يتم إنشاء هذا الـ object ثم سوف يتم تدمير هذا الـ object واستدعاء التابع finalize( ) حالما يتم استدعاء System.gc( ) وذلك لأن الـ object غير مستعمل وبالتالي سوف يتم التخلص منه.

ملاحظة:

يمكن أن يتم اسناد القيمة null لـ reference تشير على object تم الانتهاء من استعماله، وذلك بغرض التخلص منه في حال استدعاء الـ GC:

```
public class BM {
    public static void main(String[ ] args) {
        Book novel = new Book(true);
        //Some Code and things to do with object that novel is referring to
        novel = null; //getting rid of the object that novel is referring to } }
```

إن وجود Garbage Collector في لغة Java أدى إلى عدم الحاجة إلى Destructor كما في لغة C++.

Java primitive types and wrapper types:

Primitive types: وهي بنى المعطيات الأولية التي نتعامل معها عادة .... int, char, float,

wrapper types: لكل primitive type يوجد class مساعد من هذا النوع، أي عند إنشاء متحول من نوع wrapper type يكون نوع هذا المتحول عبارة عن reference على object يكون بحسب نوع الـ wrapper.

Primitive type	Size	Wrapper type
boolean	-	Boolean
char	16-bit	Character
byte	8-bit	Byte
short	16-bit	Short
int	32-bit	Integer
long	64-bit	Long
float	32-bit	Float
double	64-bit	Double
void	-	Void

✓ مثال:

```
public class Main {
    public static void main(String[ ] args) {
        int a;
        Integer k;//equals Integer k = new Integer( );
    }
}
```

ملاحظة:

الفائدة من وجود الـ wrapper type هي عند الحاجة لعمل collection لعدة أنماط معطيات ضمن نفس البنية، فمثلا يمكن إنشاء مصفوفة من نوع Object وهو يعتبر class أب لكل الـ classes عندها يمكن إسناد أي نوع من الـ classes لعنصر في هذه المصفوفة، فيتم جمع عدة أنماط بنفس البنية.

تمرين:

ما خرج البرنامج التالي:

```
class Tag {
    Tag(int marker) {
            System.out.println  ("Tag(" + marker + ")");   }
}
class Card{
    Tag t1 = new Tag(1);
     Card() {
        System.out.println ("Card ()");
        t3 = new Tag(33);   }
    Tag t2 = new Tag(2)  ;
    void f() { System.out.println( "f()" ) ;  }
    Tag t3 = new Tag(3);
}
public class BM {
      public static void main(String[ ] args) {
      Card t = new Card();
      t.f( );
      }
}
```

Output:

Tag("1")

Tag("2")

Tag("3")

Card( )

Tag("33")

f( )

- **توضيح:**

عند إنشاء object من الـ class Card فإنه في البداية يتم عمل initialize للـ class data members التي تكون بداخله ثم يتم بعدها استدعاء الـ constructor، ففي هذه الحالة يتم إنشاء كل من t1,t2,t3 وطباعة (القيمة ضمن الـ Tag(constructor ثم يتم بعدها استدعاء الـ Card( ) ليتم تغيير الـ reference t3 ليقوم بإنشاء object جديد فيتم طباعة tag(33)، بعد الانتهاء من تنفيذ Card( ) يتم استدعاء التابع f( ) ليتم طباعة f( ).

مثال آخر:

بالتعديل على المثال السابق:

```
class Tag {
    Tag(int marker) {
            System.out.println  ("Tag(" + marker + ")");   }
}

class Card{
    static Tag t1 = new Tag(1);
     Card() {
        System.out.println ("Card ()");
        t3 = new Tag(33);   }
    Tag t2 = new Tag(2)  ;
    void f() { System.out.println( "f()" ) ;  }
    Tag t3 = new Tag(3);
}
public class BM {
      public static void main(String[ ] args) {
      Card t = new Card();
      Card m = new Card(); } }
```

Output:

Tag("1")
Tag("2")
Tag("3")
Card( )
Tag("33")
Tag("2")
Tag("3")
Card( )
Tag("33")

- **توضيح:**

لا يتكرر استدعاء الباني لـ t1 وذلك لأنه static class data member يتم إنشاؤه وإسناد القيمة null عند تشغيل البرنامج أي يكون هذا الـ data member على مستوى الـ class وليس على مستوى كل object لذا يتم استدعاء الباني الخاص فيه مرة واحدة فقط.

تلخيص:

يكون تسلسل الـ initialization:

1. Class data member.
2. Object data member.
3. استدعاء الـ constructor.

Function Overloading

هي عبارة عن عملية إعادة تعريف للتابع وذلك بتغيير توقيع (signature) هذا التابع، يتمثل signature التابع بالـ arguments type و اسم هذا التابع.

لا يمكن عمل overloading بالتعديل على الـ return type.

أمثلة على الـ function overloading باستخدام أنواع المعطيات الأولية:

void f1(char x) void f1(byte x) void f1(int x) void f1(long x) void f1(float x) void f1(double x)	void f2(byte x) void f2(short x) void f2(int x) void f2(long x) void f2(float x) void f2(double x)	void f3(short x) void f3(int x) void f3(long x) void f3(float x) void f3(double x)
void f4(long x) void f4(float x) void f4(double x)	void f5(float x) void f5(double x)	void f6(double x)

ما النوع المطابق للاستدعاء $f_?(x)$ حيث $x = 5$:

f1(int)	f2(int)	f3(int)	F4(long)	f6(float)	f7(double)

- **توضيح:**

- في حال وجود method له argument من النوع نفسه يتم عندها استدعاء هذا التابع.
- في حال عدم وجود method له argument من النوع نفسه:
  1. في حال توافر method له نمط أكبر من النمط المطلوب عندها يتم استدعاء هذا التابع.
  2. في حال عدم توافر method له نمط أكبر من النمط المطلوب عندها يحدث compile-error ونحتاج عندها لعمل ما يدعى بالـ casting لأصغر أقرب نمط من النمط المطلوب ليتم عندها الاستدعاء.

✓ مثال:

بفرض قمنا بالاستدعاء f(x) حيث double x = 0:

void f(byte x)	void f(short x)	void f(int x)
void f(long x)	void f(float x)	void f(char x)

عندها يكون أقرب أصغر نمط هو float فيصبح الاستدعاء كالتالي : f((float(x)).

يتم الإنتقال إلى الأنماط ذات الحجم الأعلى كالتالي :

byte to: short, int, long, float, or double
short to: int, long, float, or double
char to: int, long, float, or double
int to: long, float, or double
long to: float or double
float to: double

✓ مثال:

ما خرج البرنامج التالي:

```
public class Leaf {
        int i = 0;
        Leaf increment () {
                i++;
                return   this; }
        void   print () {
                 System.out.println ("i = " + i);      }
        public static void main (String[ ] args) {
                Leaf x = new Leaf();
                x.increment().increment().increment().print( );
        }
}
```

Output:

i = 3

Finals:

A. Final Variables:

الـ Final Variable، هو Constant Variable، ولا يمكن تعديله، ويجب أن تتم تهيئته.
يعرف غالباً على أنه public static، وذلك لأن استعماله الخارجي شائع أكثر.
خصائص الـ Final Data:

الـ field، الذي يعرّف على أنه static وfinal، يعتبر class variable، ذو قيمة ثابتة لا يمكن تغييرها.

- استخدام الـ final مع الـ primitives، يجعل قيمتها ثابتة.
- استخدام الـ final مع الـ object references، يجعل الـ reference ثابتاً، أي فور تهيئة الـ reference ليشير إلى object معين، لا يمكن تعديل الـ reference ليشير إلى object آخر، ولكن الـ object يمكن تعديله.
  لا تؤمّن الـ Java طريقة لجعل الـ object بحد ذاته ثابتاً.

Field:
data member, member

B. Blank Finals:

تسمح الـ Java بتعريف blank finals، وهي fields معرّفة كـ final، وغير مهيّئة بقيمة ابتدائية، لكن يجب تهيئتها ضمن الـ constructor.
يجب دوماً تهيئة الـ blank finals قبل استخدامها، ويقوم الـ compiler بالحرص على ذلك.
تؤمّن الـ blank finals مرونة باستخدام الـ final keyword، فمن الممكن الآن أن يكون الـ final field مختلفاً باختلاف الـ object، مع الحفاظ على خاصية الثبوتية ضمن الـ object.

C. Final Arguments:

تسمح الـ Java بجعل arguments معيّنة final، وذلك من خلال تعريفها كـ final ضمن الـ argument list.
وهذا يعني أنه من غير الممكن ضمن الـ method تغيير ما يشير إليه الـ argument reference.

D. Final Methods:

الـ Final Method لا يمكن أن تتم عمليّة الـ Override عليها ضمن subclass.

إن الـ Final Method لها سمتان:

- وضع "قفل" على الـ method، لمنع الـ class الوريث من تعديل الـ method. فعند الحاجة لتصميم method ذات سلوك ثابت خلال عملية الوراثة، فإنّ هذه الـ method تعرّف كـ final method، حيث لا يمكن القيام بعملية الـ override عليها.
- الفعالية.

E. Private and Final:

الـ Private Methods تكون Final بشكل غير صريح، حيث لا يمكن الوصول إلى الـ Private Methods، ولا يمكن القيام بعملية الـ Override عليها. من الممكن إضافة الـ final specifier إلى private method، ولكن هذا لا يضيف معنى آخر.

F. Final Classes

الـ Final Class، لا يمكن أن يكون له subclass، أي لا يمكن الوراثة منه.
عند تعريف الـ Class كـ Final Class، فإنّ الـ Fields ضمنه لا تصبح Final تلقائياً.
ولكن، وبما أنّ الـ Final Class يمنع الوراثة، فإنّ جميع الـ Methods ضمنه تصبح Final بشكل غير صريح.
من الممكن إضافة الـ final specifier، إلى method ضمن final class، ولكنّ ذلك لا يضيف معنى آخر.

انتهت المحاضرة

لا تنسونا من صالح الدعاء