IPRO

FIABILITÉ DANS LES PROGRAMMES OBJET

Guillaume Bouyer

ENSIIE

www.ensiie.fr/~bouyer

FIABILITÉ D'UN LOGICIEL

- Robustesse
  - capacité du logiciel à fonctionner, même en présence d'événements exceptionnels
  - *ex.* saisie d'informations erronées par l'utilisateur
- Correction
  - le logiciel donne les résultats corrects lorsqu'il fonctionne normalement

LA ROBUSTESSE “À LA MAIN”

Utiliser la valeur de retour des méthodes pour signaler une erreur à la méthode appelante

1. Prévoir une valeur ou une plage de valeurs non utilisées par le fonctionnement normal de la méthode
   - parfois impossible selon les valeurs
   - difficile de connaître les causes réelles de l'erreur
   - doit gérer la propagation de l'erreur
2. Créer une structure contenant le résultat + d'autres informations pour les erreurs
   - lourd
   - structure inutile si pas d'erreur
   - doit gérer la propagation de l'erreur

LA ROBUSTESSE “À LA MAIN” : EXEMPLE

```
int factorielle(int n){
    if (n==0)
        return 1;
    else
        if (n<0) {
            System.out.println("erreur n négatif"); // lu par aucune méthode
            return 0; //valeur anormale pour factorielle
        }
        else
            return n*fact(n-1);
}
```

FIABILITÉ EN JAVA

Plusieurs mécanismes ou outils notamment

- l'encapsulation
- les exceptions
- les assertions
- les outils de tests (comme `JUnit`)
- les débuggeurs

LES ERREURS ET EXCEPTIONS

OBJECTIF DES *EXCEPTIONS*

Système dédié pour **gérer, détecter et traiter les anomalies “exceptionnelles”** qui se produisent dans le fonctionnement du programme, pour ne pas que le programme se termine brutalement.

Ex : événement rare, cas interdit, erreur de saisie…

PRINCIPE ET VOCABULAIRE

Le mécanisme des exceptions suit les étapes suivantes :

1. Une méthode peut **lever** (*générer, lancer*) **une exception** :
    - une anomalie de fonctionnement à l'intérieur de la méthode provoque la création d'un objet spécifique qui explique la cause du problème
2. L'exception est envoyée à la méthode appelante
3. Cette méthode peut **attraper** (*saisir, traiter*) l'exception (par un bloc d'instruction spécifique) :
    - soit un traitement est effectué et le programme reprend son cours
    - soit l'exception se **propage** vers la méthode appelante
4. Si aucune méthode ne traite l'exception, provoque la **terminaison** du programme

CONTENU D'UNE EXCEPTION ET EXEMPLE

Une exception est donc un objet qui possède

- un type
- un message d'erreur
- une stacktrace : séquence des méthodes menant à l'erreur, dans l'ordre inverse jusqu'au `main`

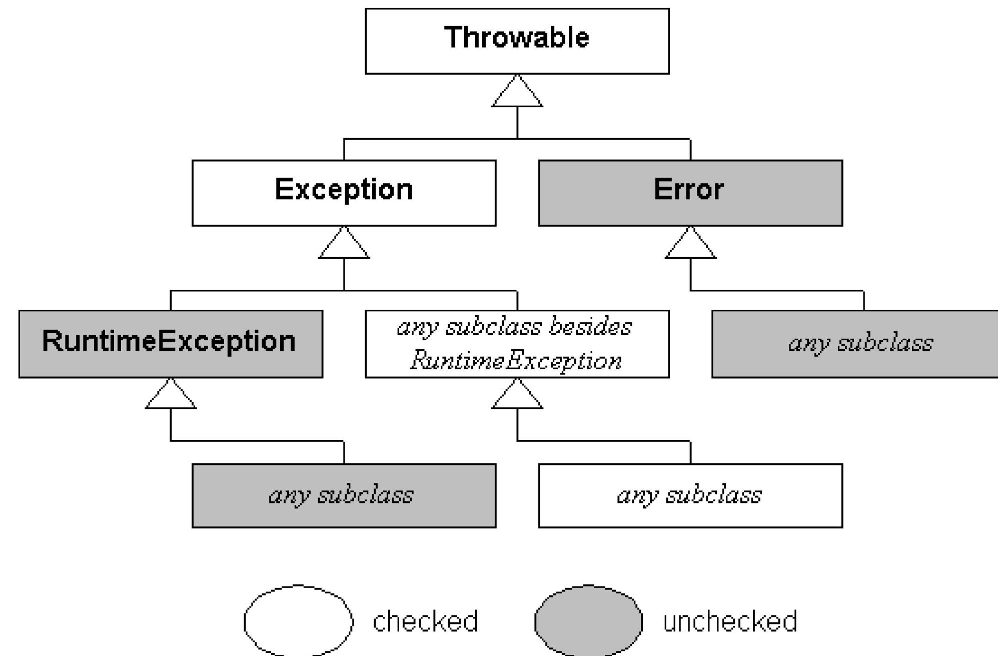
```
Exception in thread "main" java.lang.IllegalArgumentException:
  output path requires an argument
at exception.Example.parseOutputPath(Example.java:19)
at exception.Example.lambda$0(Example.java:27)
at exception.Example.parseArguments(Example.java:31)
at exception.Example.main(Example.java:39)
```

AVANTAGES DES EXCEPTIONS

- Gestion des erreurs propre et explicite
  - décrire précisément la nature des erreurs et les lignes de code les ayant provoquées
- Facilité de programmation et de lisibilité
  - regrouper la gestion d'erreurs à un même niveau
  - éviter des redondances dans l'écriture de traitements d'erreurs
  - encombrer peu le reste du code avec ces traitements

TYPES ET HIÉRARCHIE DES EXCEPTIONS/ERREURS

Héritent de `java.lang.Throwable`



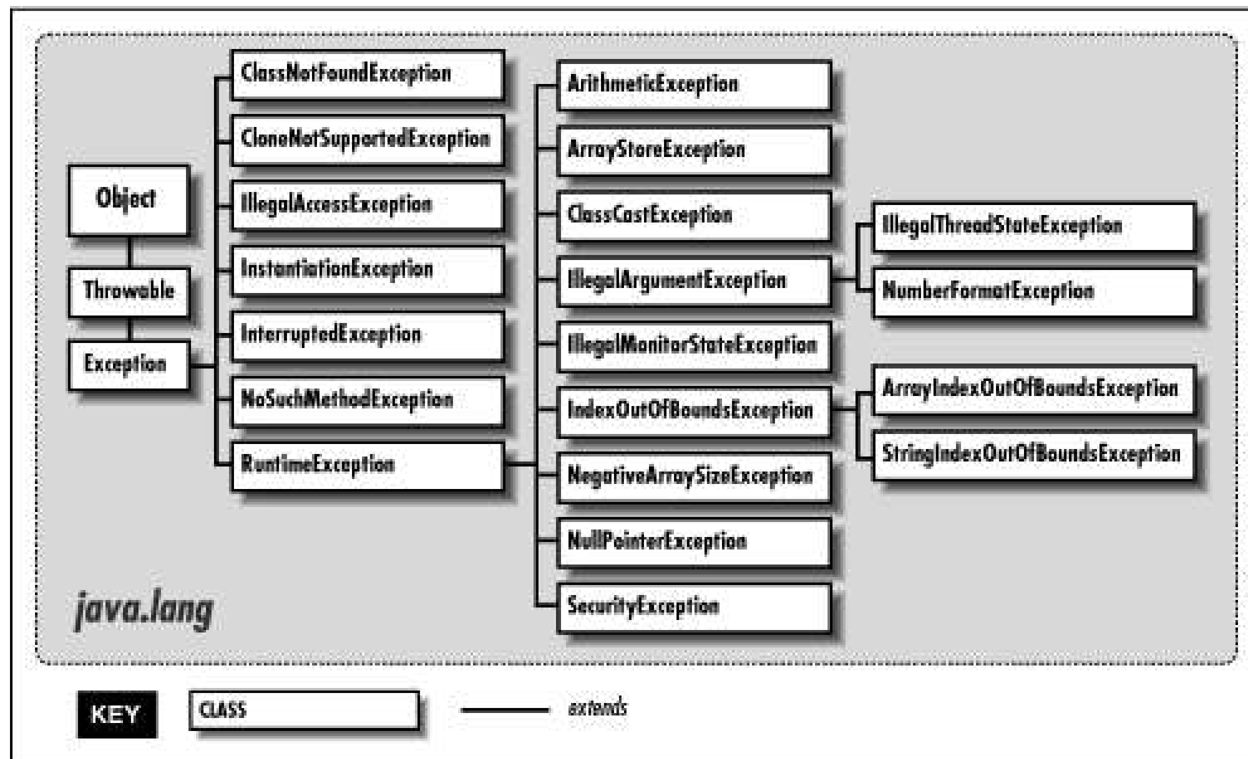
ERROR

> *“Indicates serious problems that a reasonable application should not try to catch”*

- Situation : alors que le programme est en cours d'exécution, le programme s'arrête ou se gèle car la JVM est mal configurée ou corrompue
- A priori, erreur non due au programme, mais à la configuration ou l'état de l'environnement d'exécution du programme

EXCEPTIONS PRÉDÉFINIES

“Indicates conditions that a reasonable application might want to catch”



Un grand nombre de classes d'exceptions sont proposées dans l'API Java pour couvrir les cas les plus fréquents

QUELQUES CLASSES D'EXCEPTION PRÉDÉFINIES

Classe	Description
`IndexOutOfBoundsException`	Accès à un élément inexistant dans un ensemble (ex. sous-classe `ArrayIndexOutOfBoundsException`)
`IOException`	Peut se produire lors d'opérations d'entrées/sorties
`ArithmeticException`	Quand une condition arithmétique exceptionnelle se produit (ex. division par zéro)
`IllegalArgumentException`	Passage de paramètre inapproprié à une méthode (ex. `NumberFormatException` lorsqu'on tente de convertir une `String` invalide en nombre)

QUELQUES CLASSES D'EXCEPTION PRÉDÉFINIES

Classe	Description
`NullPointerException`	Appel d'une méthode ou d'une variable à partir d'un pointeur `null` ; pointeur `null` en paramètre d'une méthode n'acceptant pas cette valeur...
`ClassCastException`	Erreur lors de la conversion d'un objet en une classe incompatible avec sa vraie classe
`FileNotFoundException`	Tentative d'ouverture d'un fichier inexistant
`NoSuchElementException`	Accès à un élement inexistant dans une collection
`AWTException`	Peut se produire lors d'opérations de type graphique

CHECKED EXCEPTIONS (SOUS CONTRÔLE)

- Surviennent dans des cas exceptionnels, mais prévisibles
  - ex. `IOException`, `FileNotFoundException`
- Peuvent/doivent être traitées correctement par le développeur
- Le compilateur vérifie (check) que les méthodes utilisent correctement ces exceptions
- Une méthode qui peut lancer une checked exception doit le préciser dans sa déclaration avec la clause `throws` (cf plus loin)

UNCHECKED EXCEPTIONS (HORS CONTRÔLE)

- Concernent les `Errors` et les `RuntimeExceptions`
  - ex. `NullPointerException`, `IllegalArgumentException` ou dépassement de tableau
- Peuvent survenir dans toutes les portions du code
  - Si ces exceptions étaient contrôlées, toutes les méthodes du programme devraient prévoir le traitement et déclarer des `throws`
- Dans la mesure du possible le développeur doit corriger le programme pour que ces événements n'arrivent pas

3 ACTIONS POSSIBLES

- Dans la méthode qui peut rencontrer le problème
  - Déclaration : `throws`
  - Déclenchement : `throw`
- Dans la méthode qui l'appelle
  - Traitement : `try ... catch ... finally`

DÉCLARATION : `THROWS`

Clause `throws` en fin de signature de la méthode

Ex. méthode `parseInt` de la classe `Integer`

```
/* Convertit une chaîne de caractères, qui doit contenir uniquement des chiffres,
  en un entier. Une erreur peut se produire si cette chaîne de caractères
  ne contient pas que des chiffres. Dans ce cas une exception de la classe
  `NumberFormatException` est émise. */

public static int parseInt(String s) throws NumberFormatException { ... }
```

- toute méthode susceptible d'émettre une checked exception doit le déclarer
- vérification à la compilation
- si plusieurs classes d'exception : séparées par virgule

DÉCLENCHEMENT : THROW

Levée explicite de l'exception avec mot-clé `throw`

```
//if (conditionAnormale) {
//  throw new MyException("message");
//}
int factorielle(int n){
  if (n<0) throw new IllegalArgumentException("factorielle : argument négatif");
  else if (n==0) {return 1;}
  else return n*fact(n-1);
}
```

Création d'une instance de la classe `Exception` (ou `Throwable`)

- soit une classe déjà fournie par le JDK, si applicable
- soit une nouvelle sous-classe d'exception, si mieux adaptée
- souvent avec un message explicite

DÉCLENCHEMENT : EXEMPLE

```
/*
* Renvoie le nom du mois correspondant au chiffre donné en paramètre
*/
public static String month(int mois) throws IndexOutOfBoundsException {
    if ((mois < 1) || (mois > 12)) {
        throw new IndexOutOfBoundsException("le numero du mois " + mois
        + " doit être compris entre 1 et 12");
    }
    if (mois == 1)
        return "Janvier" ;
    else if (mois == 2)
        return "Février" ;
    ...
    else if (mois == 11)
        return "Novembre" ;
    else
        return "Décembre" ;
}
```

CLASSES D'EXCEPTION PERSONNELLES

Avec les classes d'exception existantes, l'information transportée peut être insuffisante pour déterminer une solution satisfaisante au problème

Le programmeur peut avoir à créer ses propres classes d'exception pour s'adapter mieux aux autres classes de l'application

Par convention, les noms de classes d'exception se terminent par `Exception`

CRÉATION D'UNE NOUVELLE CLASSE D'EXCEPTION

Le mécanisme est le même que pour tout objet. On peut :

- définir des sous-classes
- des constructeurs
  - par défaut
  - avec argument(s) : message, cause...
- redéfinir des méthodes
- ajouter des méthodes

CRÉATION D'UNE NOUVELLE CLASSE D'EXCEPTION : EXEMPLE

```
public class ComptePleinException extends Exception {
  private Compte compte;
  public ComptePleinException(Compte compte) {
    super("Compte plein : limite atteinte");
    this.compte = compte;
  }
  public ComptePleinException() {}
  public ComptePleinException(String message) { super(message); }
  public ComptePleinException(Throwable cause) { super(cause); }
  public ComptePleinException(String message, Throwable cause) { super(message, cause); }
  public Compte getCompte() {return compte;}
}
```

```
public class Compte {
  ...
  public void crediter (double n) throws ComptePleinException {
    if (solde + n > limite)
      throw new ComptePleinException(this);
    solde += n;

  }
}
```

INTERCEPTION ET TRAITEMENT : TRY...CATCH

On attrape les exceptions dans des blocs dits “traite-exception” (ou “handler”)

```
try {
  dupont.crediter(1000);
}
catch (ComptePleinException e) {
  System.err.println(e.getMessage());
}
```

```
try{
    // instructions à surveiller pouvant lever exceptions
}
catch (ClasseException1 e1){
    // instructions traitement 1
}
catch (ClasseException2 e2){
    // instructions traitement 2
}
```

INTERCEPTION ET TRAITEMENT : `TRY...CATCH`

1. le corps du `try` est exécuté
2. si aucune exception ne se produit dans le bloc correspondant, le programme se déroule normalement
3. si une exception est lancée les clauses `catch` sont examinées dans l'ordre
   - la 1ère dont le type peut correspondre à la classe de l'exception est choisie et son code exécuté
   - si aucun catch ne correspond, l'exception est propagée à la méthode précédente dans la pile d'appel
4. l'exécution du programme suit son cours

CATCH

Le nom de variable correspondant à l'objet exception permet de l'utiliser dans le traitement

L'arbre d'héritage de `Throwable` permet de regrouper les traitements des erreurs liées à toutes les sous-classes d'une classe d'exception

- Ex. `catch (IOException e) {...}` permet de regrouper le traitement de toutes les erreurs dues aux entrées/sorties (`EOFException`, `FileNotFoundException`...)

Le traitement de l'exception peut lui-même lever une exception, de même type ou différent (chaînage)

Rq : A partir de Java 7 on peut attraper plusieurs exceptions dans une même clause

```
catch (IOException|SQLException e) {...})
```

FINALLY

On peut ajouer une clause `finally` : contient un traitement qui sera exécuté dans tous les cas

- que la clause `try` ait levé ou non une exception
- que cette exception ait été saisie ou non
- même si le bloc `try` ou le bloc `catch` se termine par un `return` (sauf après `System.exit()`)

Rq : Il est possible d'avoir un `try-finally` sans `catch` et `try-catch` sans `finally`

EXCEPTION ET HÉRITAGE

Si une classe dérivée redéfinit (ou implémente) une méthode, la clause `throws` de la méthode redéfinie doit être compatible avec celle d'origine

Soit une méthode `m()` de `B` qui redéfinit une méthode d'une classe mère `A`, elle ne peut pas déclarer renvoyer plus d'exceptions contrôlées que `m()` de `A`, mais seulement :

- les mêmes exceptions
- des sous-classes de ces exceptions
- moins d'exceptions
- aucune exception

EXCEPTION ET PERFORMANCES

Si aucune exception n'est levée, l'impact sur les performances d'un `throw` puis d'un bloc `try-catch` est négligeable

Par contre la levée d'une exception peut coûter cher car il faut remplir le stacktrace

- parcourir la pile à l'envers
- allouer les StackTraceElement qui conviennent

Donc on n'utilise pas des exceptions dans le flow normal d'exécution

EXCEPTIONS EN PRATIQUE

Il faut réserver les exceptions aux traitements des erreurs ou des cas exceptionnels et éviter de les utiliser pour traiter un cas “normal”

Ex. : Si un fichier est lu du début à la fin

- Utiliser la valeur spéciale renvoyée par la méthode de lecture quand elle rencontre la fin du fichier
- Ne pas utiliser `EOFException` pour repérer la fin du fichier
- Sauf si on rencontre la fin du fichier avant d'avoir lu les valeurs dont on a besoin

EXCEPTIONS EN PRATIQUE : MESSAGES EXPLICITES

- Ne jamais avoir un bloc `catch` avec un affichage de message trop vague (ou pire aucun message !). Ce sont des informations précieuses pour la résolution du problème

=> Pendant le développement, afficher au moins `e.printStackTrace` (cf classe `Throwable`)

EXCEPTIONS EN PRATIQUE :
ERROR

Les `Error` sont réservées aux erreurs qui surviennent dans le fonctionnement de la JVM

- ne devraient jamais arriver
- ne devraient jamais être lancées par le code écrit par le développeur

Ex.

- problème de mémoire `OutOfMemoryError`
- une méthode qui n'est pas trouvée `NoSuchMethodError`

EXCEPTIONS EN PRATIQUE : `RUNTIMEEXCEPTION`

Comme une Error, une exception non contrôlée ne devrait jamais arriver

- due à une erreur indépendante du code d'où elle a été lancée (réseau défaillant, passage d'un mauvais paramètre…)
- due à un bug du code (ex. `ArrayIndexOutOfBoundsException`)

EXCEPTIONS EN PRATIQUE : `RUNTIMEEXCEPTION`

A utiliser quand

- On sait que le problème ne pourra pas être résolu par une des méthodes appelantes
- Mauvaise utilisation de la méthode
  - *Ex.* passage d'un paramètre invalide
    `IllegalArgumentException`

EXCEPTIONS EN PRATIQUE : `RUNTIMEEXCEPTION`

Une bonne solution est de laisser remonter l‘exception jusqu’à une classe proche du début de l'action qui a provoqué le problème

- cette classe attrape l'exception
- l'enregistre comme événement inattendu (cf système de logging)
- remet éventuellement les choses en place
- stoppe proprement l'action qui a provoqué le problème
- prévient l'utilisateur s'il le faut

EXCEPTIONS EN PRATIQUE : EXCEPTIONS CONTRÔLÉES

Pour les cas peu fréquents mais, au contraire des exceptions non contrôlées, pas inattendus

EXCEPTIONS EN PRATIQUE : EXCEPTIONS CONTRÔLÉES

Si une exception du JDK convient, il vaut mieux l'utiliser car ces exceptions sont bien connues des développeurs

S'il n'existe pas d'exception du JDK au bon niveau d'abstraction pour l'application, il vaut mieux créer un nouveau type d'exception

LES ASSERTIONS

PRINCIPE

- Une assertion est une déclaration qui permet de tester des hypothèses sur des points critiques du programme
  - post-conditions, invariants de classe, après les traitements complexes...
- Contient une expression booléenne qui doit être vraie lorsque l'assertion s'exécutera. Sinon, le système lancera une erreur.
- Moyen rapide et efficace pour détecter et corriger les bugs pendant la programmation
- En supplément, sert de commentaire

ASSERT

```
assert assertion [: expression];
```

- `assertion` est une expression booléenne
- la valeur de `expression` est affichée si `assert` provoque une erreur
- si l'assertion n'est pas vérifiée une `AssertionError` est lancée (et l'expression passée à son constructeur)

Ex :

```
assert i!=0 : "i = " + i + " devrait être non nul";
```

ASSERT

Ces vérifications ne sont pas actives par défaut : directive de compilation

```
java -enableassertions [:nompackage]
```

Elle peuvent être activées en période de test, et désactivées en production

Les assertions désactivées ne coûtent qu'un test de drapeau mais elles prennent de la place dans le code compilé

ASSERTIONS EN PRATIQUE

- Pour vérifier la logique interne
- Pas pour les vérifications nécessaires au bon fonctionnement du programme : activation non garantie en production
- Pas pour vérifier les arguments d'une méthode publique
  - utiliser les `RuntimeException` (`IllegalArgumentException`, `IndexOutOfBoundsException`, `NullPointerException`…)
- Pas pour vérifier une condition qui dépend de l'extérieur de l'application
  - ex. saisie d'une valeur par l'utilisateur
- Expressions testées ne doivent pas avoir d'effets secondaires
- Pas assez puissant pour vérifications du type `∀i t[i] > 0`

MUTABILITÉ

OBJETS MUTABLES VS IMMUABLES

Objet immuable (ou non-modifiable)

Se dit d‘une instance dont l’état ne peut plus changer après sa création, même en ayant sa référence

Ex.: pour les `String`, création d'une nouvelle instance si nouvelle affectation de valeur

Objet mutable (ou modifiable)

Peut être altéré si l'on a sa référence

cf la plupart des ex. vus précédemment, ex. `StringBuilder`

OBJETS IMMUABLES

Avantages

- simples à construire, à tester et à utiliser
- besoin ni de constructeur par copie, ni d'une mise en oeuvre de clone (simple copie de référence suffit)
- invariants de classe testés à la création seulement
- bonnes clés pour certaines collections comme les Map et les Set
- automatiquement thread-safe et n'ont pas de problèmes de synchronisation
- ont une “failure atomicity” : s'ils lancent une exception, ne se laissent jamais dans un état indésirable ou indéterminé

OBJETS IMMUABLES : EXEMPLE ?

```java
import java.util.Date;
public final class Person {
    private String firstName;
    private String lastName;
    private Date dob;

    public Person( String firstName, String lastName, Date dob){
        this.firstName = firstName;
        this.lastName = lastName;
        this.dob = dob;
    }

    public String getFirstName(){
        return this.firstName;
    }

    public String getLastName(){
        return this.lastName;
    }

    public Date getDOB(){
        return this.dob;
    }
}
```

OBJETS IMMUABLES : EXEMPLE ?

```
Date myDate = new Date();
Person myPerson = new Person( "David", "O'Meara", myDate );
myDate.setMonth( myDate.getMonth() + 1 );
```

`Date` est mutable, donc `Person` n'est pas immuable !

CRÉER DES OBJETS IMMUABLES

Règles à suivre :

- Tous les champs doivent être `private final`
- La classe doit être `final` ("strong immutability")
  - sinon possible de modifier une instance par héritage : nouveaux attributs mutables et redéfinitions de méthodes
  - ou seulement toutes les méthodes `final` ("weak immutability", mais tout comportement ajouté pourrait être mutable)
- La référence à `this` ne doit jamais être exportée

CRÉER DES OBJETS IMMUABLES

Règles à suivre (suite) :

- Protéger les attributs faisant référence à un objet non immuable ou non primitif
  - `private`
  - référence jamais exportée
  - représenter l'unique référence à cet objet
  - Faire des copies défensives profondes en entrée et en sortie (“deep copy”)
- Ne pas fournir de méthodes “mutatrices” qui modifient l'état de l'objet

COPIE DÉFENSIVE : EXEMPLES

```
public Person(String firstName, String lastName, Date dob){
        this.firstName = firstName;
        this.lastName = lastName;
        this.dob = new Date( dob.getTime() ); // copie défensive
    }
```

```
Date myDate = myPerson.getDOB();
myDate.setMonth( myDate.getMonth() + 1 );
```

```
public Date getDOB(){
    return new Date(this.dob.getTime());
}
```

SOURCES

C. Dubois, ENSIIE

R. Grin, Univ. Nice-Sophia Antipolis

H. Fauconnier, IRIF

R. Forax, Univ. Marne la Vallée

G. Picard, ENSMSE