Lyon 1

IUT Lyon 1

l'excellence technologique

Programmation OO

Programmation Orientée Objet en Java

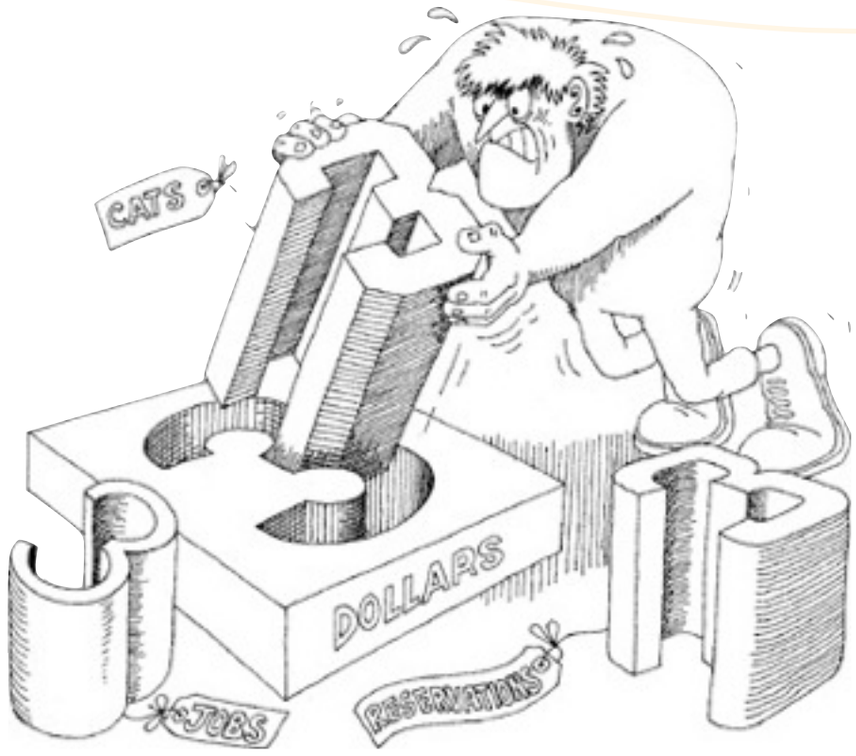
IUT Lyon 1

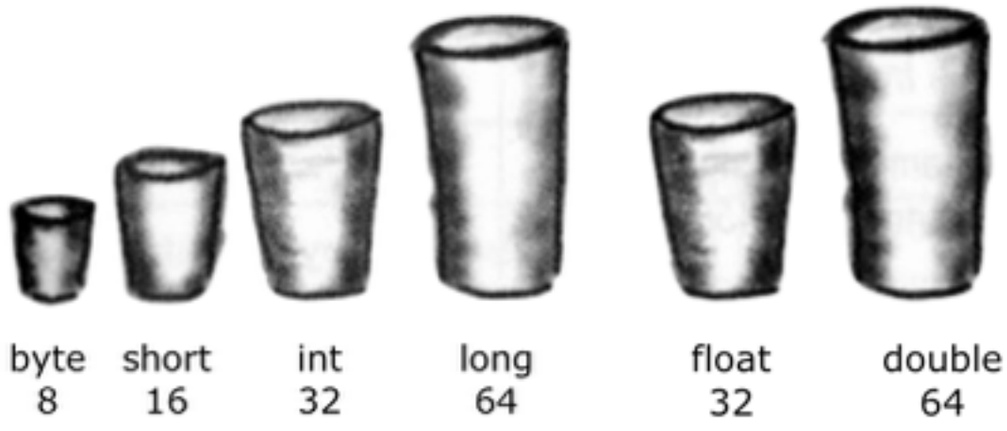
l'excellence technologique

POO Java

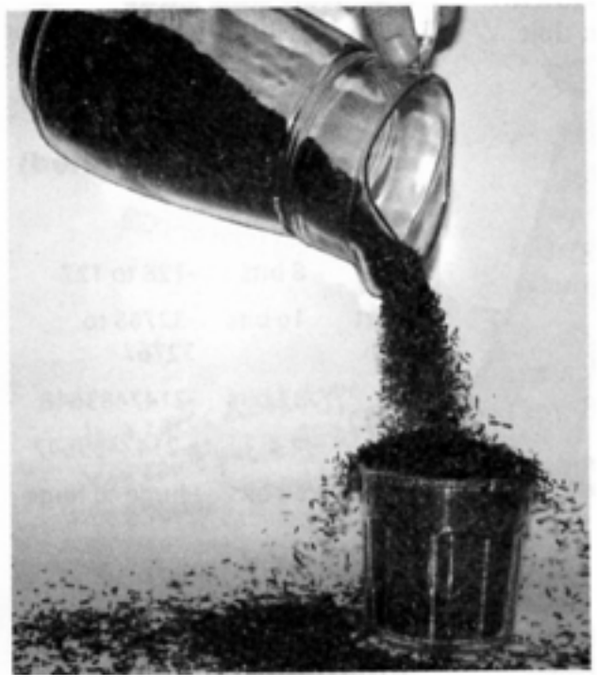
Eléments de syntaxe avancés

- Types primitifs
  - boolean ***true*** ou ***false***
  - char 16 bits (caractères)
  - Numériques (signés)
    - Entiers
      - byte 8 bits -128 à 127
      - short 16 bits -32 768 à 32 767
      - int 32 bits -2 147 483 648 à 2 147 483 647
      - long 64 bits immense
    - Décimaux
      - float 32 bits
      - double 64 bits





- Les pièges !
  - Vérifier que la valeur peut entrer dans la variable
  - Le compilateur vous empêche de faire :
    int x = 24;
    byte b = x;
  - Les valeurs littérales sont par défaut de type **int** sauf si on précise le type
    long var = 234L;



```
public class LongDivision {
    private static final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
    private static final long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
    public static void main(String[] args) {
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}
```

↳ Tester et corriger le programme

- Classes d'encapsulation (ou classes d'emballage)
  - Permettent d'utiliser un type primitif sous forme d'objet
    - Boolean
    - Number, Byte, Short, Integer, Long, Float, Double
    - Character
  - Ces classes définissent des constantes
    - MIN_VALUE et MAX_VALUE pour les types numériques
    - NaN, POSITIVE_INFINITY, NEGATIVE_INFINITY pour les types Float et Double
  - Elles fournissent également des méthodes pour manipuler les données
    - Character : isLetter(), isLowerCase(), isDigit(), isWhiteSpace(), ...
    - Elles fournissent des méthodes "parse" pour la conversion de chaîne de caractères

- Classes d'encapsulation (ou classes d'emballage)
  - auto-boxing / auto-unboxing
    - Auto-boxing
      - Integer myInt = 13;
      - remplacé par Integer myInt = new Integer(13); par le compilateur
    - Auto-unboxing
      - int x = myInt;
      - remplacé par int x = myInt.intValue(); par le compilateur

- Classes d'encapsulation (ou classes d'emballage)
  - auto-boxing / auto-unboxing
    - Les pièges

```
Long sum = 0L;
for (long i = 0; i < Integer.MAX_VALUE; i++) {
    sum += i;
}
```

- Cette boucle s'exécute en 7 secondes sur ma machine alors qu'en la corrigeant elle passe à 1,1 secondes
- Quelle correction faut-il apporter ?
- Tester (en mesurant le temps d'exécution)
- Autre piège : ne pas comparer des instances de classes d'encapsulation avec == mais plutôt avec equals()

- Les pièges des flottants
  - Les types float et double ne permettent pas de représenter des valeurs exactes
    - Exemple

```
public class Monnaie {
    public static void main(String[] args) {
        System.out.println(2.00 - 1.1);
    }
}
```

  - Les types double et float sont inadaptés là où un résultat exact est attendu
    - Solutions
      - Utiliser des entiers représentants des centimes (200 - 110 = 90)
      - Utiliser la classe BigDecimal qui permet de représenter des grands nombres et d'interagir avec le type SQL DECIMAL

Les pièges des flottants

Classe BigDecimal

- Utiliser systématiquement le constructeur BigDecimal(String) et non pas BigDecimal(double) car sinon la précision est perdue avant la représentation

```
import java.math.BigDecimal;

public class Monnaie {
    public static void main(String[] args) {
        System.out.println(new BigDecimal("2.0").subtract(new BigDecimal("1.1")));
    }
}
```

- Pour cet exemple, la solution n'est pas particulièrement élégante et est un peu plus lente
- Pour des applications financières dans lesquelles on veut des résultats exacts et la maîtrise des détails des méthodes utilisées pour arrondir notamment, ces classes peuvent être utilisées

- Les pièges des flottants
  - Ecrire un petit programme qui permet de résoudre le problème suivant
    - Vous avez un euro en poche et vous arrivez devant un étal avec des boites de bonbons à 10 c€, 20c€, 30 c€, et ainsi de suite jusqu'à 1 €. Vous en achetez un de chaque en commençant par ceux qui coutent 10 c€ jusqu'à ce que vous n'ayez plus assez d'argent pour acheter le prochain bonbon de l'étal.
  - Combien de bonbons avez-vous acheté ?
  - Combien vous reste-t-il en poche ?
  - Tester dans un premier temps en commettant l'erreur de choisir le type double, puis écrivez deux solutions (une avec les centimes et l'autre avec BigDecimal)

Les pièges des flottants

Ne jamais faire des tests d'égalité sur des flottants

```
public class FloatingPointsEquality {

    public static void main(String[] args) {
        double a = 6.6 / 3.0;

        if (a == 2.2) {
            System.out.println("Normal !");
        } else {
            System.out.println("WTF !");
        }

        System.out.println(a);

    }

}
```

- Les pièges des flottants
  - Ne jamais faire des tests d'égalité sur des flottants
  - Solution

```
public class FloatingPointsEquality {

    private static final double EPSILON = 1e-6;

    public static void main(String[] args) {
        double a = 6.6 / 3.0;

        if (a >= (2.2 - EPSILON) && a <= (2.2 + EPSILON)) {
            System.out.println("Normal !");
        } else {
            System.out.println("WTF !");
        }

        System.out.println(a);

    }

}
```

- Types énumérées
  - enum
  - Possèdent des méthodes :
    - values()
    - name()

```
import javax.swing.JOptionPane;

public class EnumTest {

	public enum Jour {
		DIMANCHE, LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI}

	public static void main(String[] args) {
		Jour[] tabJours = Jour.values();
		Jour jour = tabJours[(int)(Math.random()*tabJours.length)];
		String msg;

		switch (jour) {
		case SAMEDI:
		case DIMANCHE:
			msg = "Vive le week-end !";
			break;
		case LUNDI :
			msg = "Les lundis sont nuls";
			break;
		default:
			msg="Vivement le week-end !";
			break;
		}

		JOptionPane.showMessageDialog(null, jour.name() + " : " + msg);
	}
}
```

- Types énumérées
  - Les enums sont implémentés comme des classes
  - Ils peuvent posséder
    - un constructeur
    - des méthodes
  - Ecrire une classe PoidsSurPlanetes qui demande la masse sur terre et qui affiche le poids sur toutes les planètes du système solaire (ou pour avoir un résultat plus parlant la masse équivalente sur terre)

```
public enum Planet {
    MERCURE (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    TERRE   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass;   // en kg
    private final double radius; // en mètres
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass()   { return mass; }
    private double radius() { return radius; }

    // Constante gravitationnelle universelle  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    public double surfaceGravity() {
        return G * mass / (radius * radius);
    }

    public double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }

}
```

- Classes anonymes et listeners
  - Programmation événementielle
    - Comment associer une action à un appui sur le bouton ?
    - Swing met en place (de façon *presque* transparente pour l'utilisateur) un thread qui récupère les événements de la souris et du clavier en provenance du système d'exploitation
    - Les événements sont ensuite transmis au composant concerné
    - L'utilisateur peut configurer les différents composants pour être notifié quand un événement se produit et pour déclencher les traitements associés

cf. quelques transparents plus loin pour le lancement d'un programme swing

- Classes anonymes et listeners
  - Les objets qui veulent être prévenus lorsqu'une action sur un élément d'IHM se produit doivent s'enregistrer auprès de ces composants
    - c'est le concept d'event listener (observateur d'événement)
  - Pour que les composants d'IHM soient indépendants de leurs utilisateurs, c'est aux utilisateurs de s'adapter et de présenter un type défini par les composants
    - Les observateurs doivent implémenter une interface

Classes anonymes et listeners

- Associer une action à un bouton
  - La classe JButton définit une méthode addActionListener qui prend en paramètre un objet de type ActionListener
  - ActionListener est une interface qui définit la méthode

```
void actionPerformed(ActionEvent e)
```

  - L'objet e de type ActionEvent permet de connaitre la source de l'événement ou les modificateurs (touches Shift, Ctrl, ...)
  - Il existe deux grandes alternatives pour associer une action à un bouton
    - Implémenter l'interface ActionListener dans la classe de gestion de l'IHM
    - Créer une instance d'une classe anonyme implémentant l'interface ActionListener

```java
public class PremierBouton implements ActionListener {
    JLabel monLabel;

    public void go(){
        JFrame cadre = new JFrame();
        cadre.setLayout(new FlowLayout(FlowLayout.LEFT));

        JButton monBouton = new JButton("Mon bouton");
        monBouton.addActionListener(this);
        monLabel = new JLabel();

        cadre.add(monBouton);
        cadre.add(monLabel);
        cadre.setSize(300,60);
        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cadre.setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        monLabel.setText("Hello !");
    }

     public static void main(String[] args) {
        new PremierBouton().go();
     }
}
```

Ajout de l'objet courant à la liste des "listeners" du bouton

Implémentation de l'interface ActionListener

Il faut une instance de la classe PremierBouton pour recevoir les événements

```
public class BoutonClasseAnonyme {

    public static void main(String[] args) {
        JFrame cadre = new JFrame();
        cadre.setLayout(new FlowLayout(FlowLayout.LEFT));

        JButton monBouton = new JButton("Mon bouton");
        final JLabel monLabel = new JLabel();
        monBouton.addActionListener(new ActionListener(){
            @Override
            public void actionPerformed(ActionEvent e) {
                monLabel.setText("Hello !");
            }
        });

        cadre.add(monBouton);
        cadre.add(monLabel);

        cadre.setSize(300,60);
        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        cadre.setVisible(true);
    }

}
```

La variable monLabel doit être déclarée **final** pour être accessible depuis la classe anonyme

Classe anonyme : le corps de la classe est déclaré au moment de l'instanciation

Classe anonyme

On crée une nouvelle instance d'une classe anonyme (elle n'a pas de nom) qui implémente l'interface ActionListener

```
final JLabel monLabel = new JLabel();


monBouton.addActionListener(new ActionListener(){

    @Override
    public void actionPerformed(ActionEvent e) {
        monLabel.setText("Hello !");
    }

});
```

final car monLabel doit rester lié à l'instance de la classe anonyme même quand on sort de la méthode courante (la variable perd son statut de "variable locale")

Classe anonyme - avantages

- Les méthodes d'une classe anonyme peuvent utiliser les variables et les méthodes d'intance de la classe englobante (y compris les variables privées)
- Les méthodes d'une classe anonyme peuvent utiliser les variables locales et les paramètres de la méthode dans laquelle la classe anonyme est définie s'ils sont déclarés **final**
- Le code du comportement associé à un composant est regroupé avec son instanciation
- Il n'y a pas besoin de trouver un nom pour la classe anonyme (dans une application complexe cela pourrait conduire à un grand nombre de classes avec des noms semblables)

- Exercice
  - Modifier le programme précédent en utilisant une variable d'instance pour le label
    - Utiliser une classe anonyme pour écouter les événements du bouton
    - Le bouton contiendra le texte "Quelle heure est-il ?" et le label affichera l'heure au format 23:12:46 (utiliser la méthode String.format())

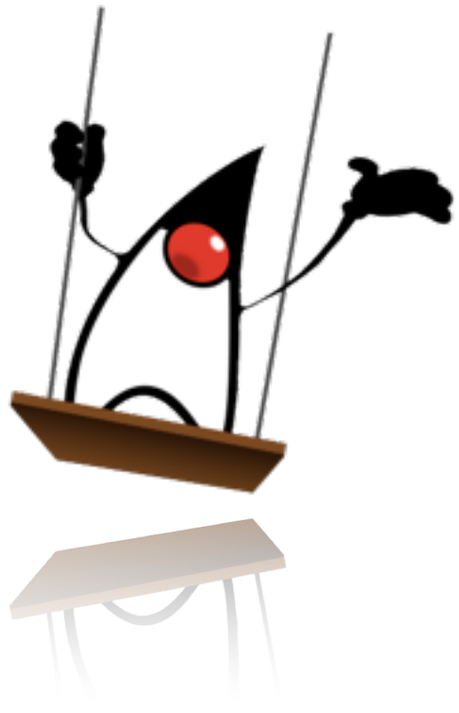
Lancement d'un programme Swing

```
public class PremierFrame {

    private static void createAndShowGUI() {
        JFrame cadre = new JFrame("Titre du cadre");
        cadre.add(new JButton("Mon bouton"));
        cadre.add(new JLabel("Mon Label"));
        cadre.setSize(300,60);
        cadre.setVisible(true);
    }

    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

Classe anonyme qui implémente l'interface Runnable dont la méthode run() sera appelée depuis le thread qui distribue les événements aux composants



IUT Lyon 1

l'excellence technologique

Gestion des erreurs

LP IEM

Les exceptions : gestion des erreurs

- Lorsqu'une erreur se produit à l'intérieur d'une méthode, la méthode crée un objet qu'elle passe au système. Cet objet est une exception. Elle contient des informations sur l'erreur et possède un type qui permet d'identifier la nature de l'erreur
  ➪ la méthode "lance une exception"

- Le système remonte la pile des appels de fonctions à la recherche d'un bloc de code traitant cette exception

- Si aucun bloc n'est trouvé le programme se termine et la machine virtuelle Java affiche un message sur la console

- Les exceptions : gestion des erreurs
  - Il existe 3 types d'exceptions
    - Les **exceptions vérifiées** (*checked exceptions*) : conditions exceptionnelles qu'une application bien conçue doit anticiper et dont elle doit récupérer (ex : saisie d'un nom de fichier inexistant par l'utilisateur,...).
    - Les **exceptions non vérifiées** (sous classes de **RunTimeException**) : conditions exceptionnelles internes à l'application et que l'application ne peut généralement pas résoudre. Ce sont généralement des bugs, erreurs de logique, mauvaise utilisation d'une API.
    - Les **erreurs** (sous classes d'**Error**) : conditions exceptionnelles externes à l'application et que l'application ne peut généralement pas résoudre (ex : erreur de lecture de fichier due à un secteur défectueux). L'application peut traiter ces exceptions mais ce n'est pas une obligation.

Les exceptions : gestion des erreurs

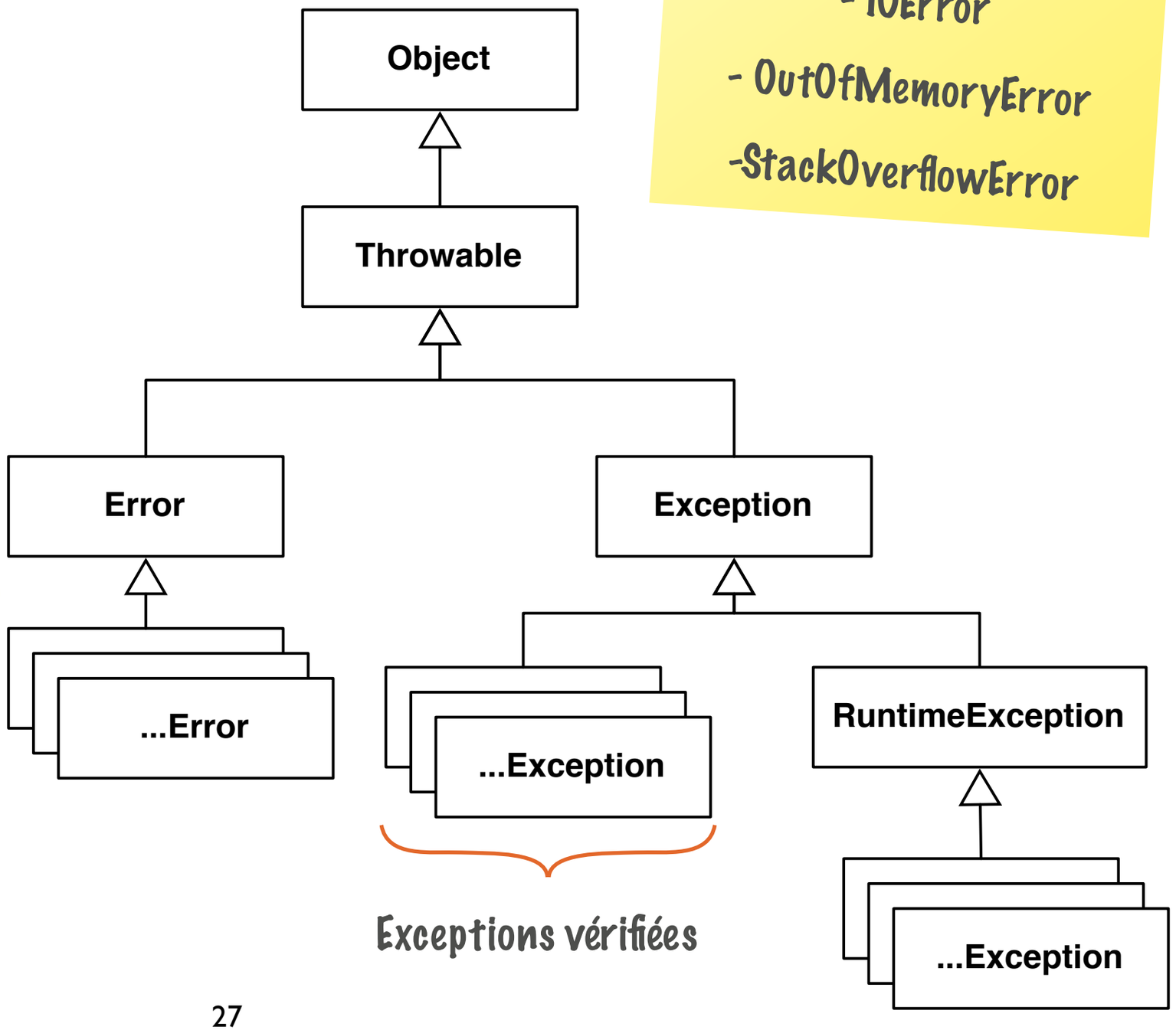
Les 3 types d'exceptions

Exemples d'exceptions non vérifiées

- ArithmeticException (division par zéro)
- NullPointerException
- IndexOutOfBoundsException
- IllegalArgumentException

Exemples d'exceptions vérifiées

- IOException
- FileNotFoundException



Exemples d'erreurs

- IOError
- OutOfMemoryError
- StackOverflowError

- Les exceptions : gestion des erreurs
  - Syntaxe
    - Lors de l'appel d'une méthode pouvant lancer une exception vérifiée, il faut soit :
      - la gérer (**try**/**catch**)
      - la re-déclarer (clause **throws** dans la déclaration de la méthode)
    - Gestion de l'exception (**try**/**catch**)

Chemin "normal"

```
try {
    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
} catch (Exception e) {
    e.printStackTrace();
}
```

Si une exception se produit on affiche le contenu de la pile des appels

- Les exceptions : gestion des erreurs
  - Syntaxe
    - Redéclaration de l'exception (**throws**)
      - La méthode englobante doit redéclarer toutes les exceptions que peuvent lancer les méthodes qu'elle appelle
      - La gestion est donc reportée au niveau supérieur (ici c'est la JVM qui devra gérer)

```
public static void main(String[] args) throws ClassNotFoundException,
InstantiationException, IllegalAccessException, UnsupportedLookAndFeelException {

  //...
    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
    //...
```

- Les exceptions : gestion des erreurs
  - Syntaxe
    - Gestion de l'exception (**try**/**catch**)
      - dans l'exemple précédent on avait regroupé toutes les exceptions dans un seul bloc catch (en exploitant le polymorphisme)
      - On peut affiner le traitement en fonction de la nature de l'exception

```
try {
    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
} catch (ClassNotFoundException e) {
    //Faire qqchose en rapport avec cette exception
} catch (InstantiationException e) {
    //Faire qqchose en rapport avec cette exception
} catch (IllegalAccessException e) {
    //Faire qqchose en rapport avec cette exception
} catch (UnsupportedLookAndFeelException e) {
    //Faire qqchose en rapport avec cette exception
}
```

- Les exceptions : gestion des erreurs
  - Exemples
    - Importer le fichier fr.lpiem.erreurs.Calcul
      - Observer l'utilisation des exceptions pour vérifier le domaine de variation des arguments
      - Est-ce une exception vérifiée ?
    - Importer le fichier fr.lpiem.erreurs.CalculAvecTryCatch
      - Tester le programme et provoquer une exception
      - Quelle méthode provoque l'exception "attrapée" ?
      - Est-ce une exception vérifiée ?

- Les exceptions : gestion des erreurs
  - Exemples
    - Importer le fichier fr.lpiem.erreurs.CalculAvecTryCatchToutesExceptions
    - Analyser la gestion des exceptions (blocs catch) et tester le programme
    - Que fait la méthode `getMessage()` ?
    - Dans quelle classe est-elle définie ?
    - Où le message est-il initialisé ?
    - Tester les différentes exceptions

- Les exceptions : gestion des erreurs
  - Syntaxe
    - Le mot clé `finally` permet de fournir un bloc de traitement exécuté systématiquement qu'il y ait eu une exception ou non
    - Ce bloc permet de faire le ménage quand le traitement a été interrompu par une exception (fermeture de fichiers, connexions réseaux,...)
  - Exemple
    - Importer le fichier fr.lpiem.erreurs.CalculAvecTryCatchFinally
      - Vérifier que le bloc finally est exécuté dans tous les cas de figure

Exceptions ou assertions

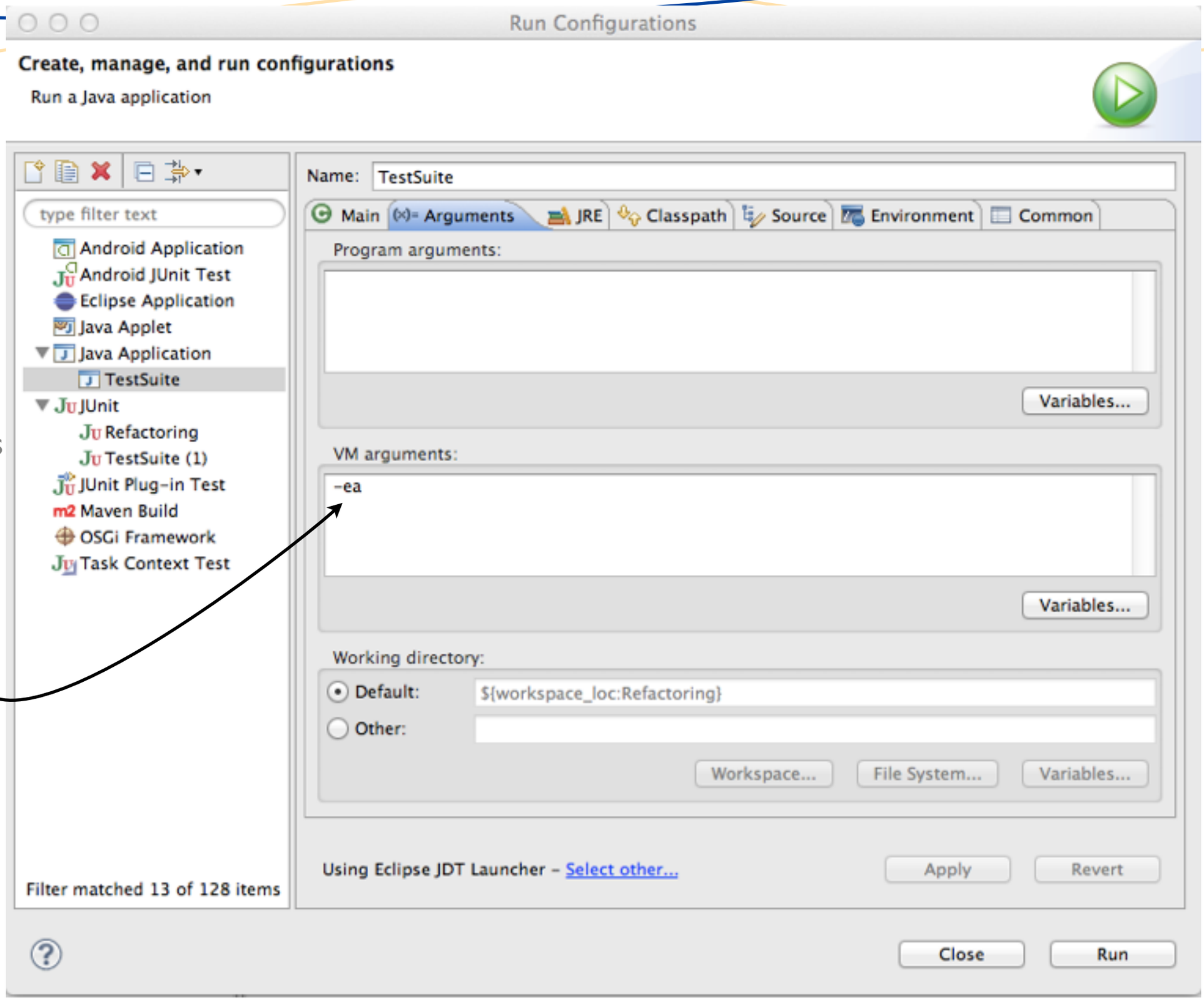
- Les méthodes publiques doivent être protégées par des exceptions (documentées) contre des arguments invalides

```
/**
     * Returns a BigInteger whose value is (this mod m). This method
* differs from the remainder method in that it always returns a
* non-negative BigInteger.
*
* @param m the modulus, which must be positive
* @return this mod m
* @throws ArithmeticException if m is less than or equal to 0 */
public BigInteger mod(BigInteger m) { if (m.signum() <= 0)
          throw new ArithmeticException("Modulus <= 0: " + m);
       //... Do the computation
}
```

- Exceptions ou assertions
  - Pour les méthodes non exportées, l'auteur du package maitrise les circonstances dans lesquelles les méthodes sont appelées
    - Il est quand même nécessaire, à des fins de debug et de documentation de tester le bon respect des domaines de validité des arguments d'une méthode : plages de valeurs, non nullité d'un argument, toute autre hypothèse
  - Solution élégante : utiliser les assertions
    - Les assertions sont activables
      - A la différence des exceptions, elles peuvent être retirées du code de production pour améliorer les performances
      - Il faut passer l'option -ea à la machine virtuelle java pour que les assertions soient prises en compte
      - Lorsqu'elle sont activées, le non respect d'une condition déclenche une AssertionError

Assertions

- Activation des assertions sous Eclipse
- Sous-menu Run configurations du menu Run



Assertions

- Utiliser l'instruction assert suivi d'une condition booléenne

```
// Private helper function for a recursive sort
    private static void sort(long a[], int offset, int length) {
    assert a != null;
    assert offset >= 0 && offset <= a.length;
    assert length >= 0 && length <= a.length - offset;
     //... Do the computation
    }
```

IUT Lyon 1

l'excellence technologique

Tests unitaires

LP IEM

Tests unitaires

- Qu’est-ce qu’un test unitaire ?
  - Les tests fonctionnels, d’intégration ou de validation testent les fonctionnalités d’un logiciel d’un point de vue externe ➪ tests “clients”
  - Tests unitaires s’appliquent au niveau d’une classe
  - Tests unitaires ➪ “tests programmeur”
  - Test des fonctionnalités élémentaires
    - Logique d’un algorithme
    - Rejet des entrées hors domaine
    - …

Tests unitaires

- Pourquoi des frameworks de tests unitaires ?
  - Généralement la première version d'un code est testée en détail (pas à pas, sortie de chaîne de caractère sur la console, ...)
  - Les problèmes arrivent lors des modifications (ajout de fonctionnalités, corrections de bugs,...)
  - Nécessité d'écrire des tests automatiques
    - ce n'est pas le programmeur qui vérifie les sorties du programme
    - les tests sont construits en utilisant les entrées, un contexte et le résultat attendu
    - le test vérifie que le résultat est conforme au résultat attendu

Tests unitaires

- Pourquoi des frameworks de tests unitaires ?
  - Au cours du développement on construit des suites de test qui sont exécutées plusieurs dizaines de fois par jour
    ➪ vérifier que chaque modification significative ne casse pas le programme
  - Automatisation des tests
    - un clic pour lancer une suite de tests

Tests unitaires

- JUnit
  - Framework de tests unitaire spécifique à Java
  - Intégré dans tous les IDE (Netbeans, Eclipse, ...)
  - Deux versions coexistent
    - Version 3 : utilise l'héritage pour définir les tests
    - Version 4 : s'appuie sur les annotations (introduites avec Java 5)

JUnit 4 - exemple

```
import static org.junit.Assert.*;
import org.junit.*;

public class SlidingPuzzleTest {

    Connection c;

    @Before
    public void setUp() {
        c = Connection.newInstance();
    }

    @Test
    public void testConnectioOpening() {
        assertTrue(c.isClosed());
        c.open();
        assertFalse(c.isClosed());
    }
```

Tests unitaires

- JUnit 4
  - Chaque méthode de test qui doit être invoquée automatiquement est marquée avec l'annotation @Test
  - Si on veut tester qu'une méthode lance bien une exception dans certaines conditions on annote la méthode avec @Test(expected = Exception.class)
    - On peut préciser la classe de l'exception attendue

Tests unitaires

- JUnit 4
  - L'annotation @Before permet de dire à JUnit d'exécuter une méthode avant chaque test
    - Utile pour effectuer une initialisation commune à plusieurs méthodes de test
  - Les tests s'écrivent en utilisant les méthodes statiques de la classe Assert :
    - assertEquals(int arg1 , int arg2)
    - assertEquals(Object o1, Object o2)
      - définie pour tous les types primitifs et les objets
    - assertTrue(boolean condition)
    - assertFalse(boolean condition)

Tests unitaires

- JUnit 4
  - Méthodes statiques de la classe Assert (suite)
    - assertNull(Object object)
    - assertNotNull(Object object)
    - assertSame(Object expected, Object actual)
    - assertNotSame(Object unexpected, Object actual)
  - Toutes les méthodes statiques de la classe Assert existent aussi avec un argument supplémentaire de type String affiché en cas d'échec du test
    - assertTrue("Valeur négative", x < 0)
    - assertEquals("Nb de mesures different", m.length, l.size())

- JUnit 4
  - Attention aux problèmes d'arrondis dans les tests d'égalité
    - Utiliser la méthode appropriée qui permet de spécifier une tolérance (delta) sur le test d'égalité

```
public static void assertEquals(String message,
                                double expected,
                                double actual,
                                double delta)
```

  - Exemple :

```
assertEquals("Diagonale du carre", Math.sqrt(2), Carre.diagonale(2), 1e-6);
```

JUnit 4 - Exercice

- Créer une classe immuable Point avec un constructeur prenant les coordonnées x et y en paramètre
- Ajouter une méthode double distance(Point p)
- Implémenter les méthodes hashcode() et equals(Object o)
- Ecrire des tests unitaires pour valider ces méthodes

JUnit 4 - Exercice

- Créer des classes Circle, Rectangle implémentant une interface Shape définissant :
  - une méthode boolean contains(Point p) qui renvoie vrai si les coordonnées du point (p.x,p.y) sont à l'intérieur de la forme
  - une méthode double area() retournant l'aire
- Constructeurs :
  - Circle (Point center, int radius)
  - Rectangle(Point topLeftCorner, int width, int height)
- Implémenter les méthodes hashcode() et equals(Object o)
- Ecrire des tests unitaires pour valider ces méthodes