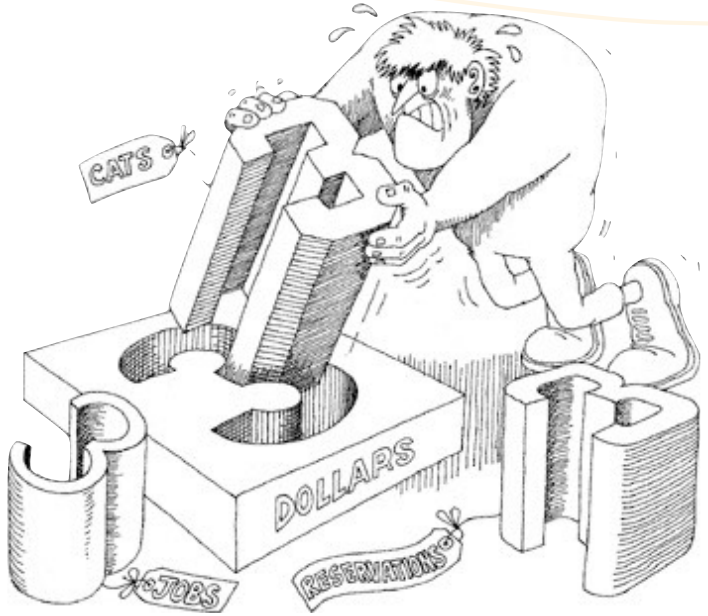
IUT Lyon 1

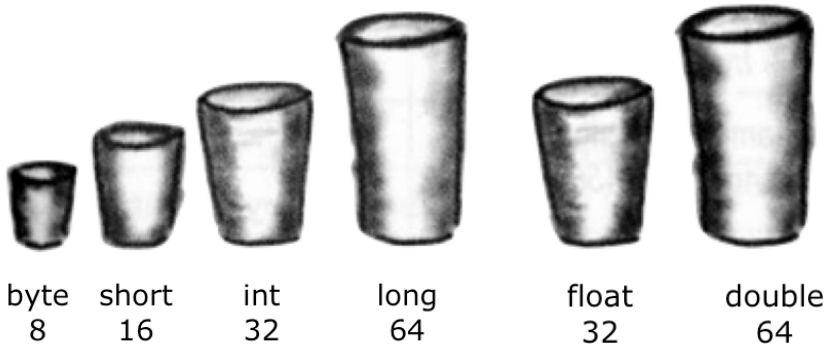
l'excellence technologique

POO Java

Eléments de syntaxe « avancés »

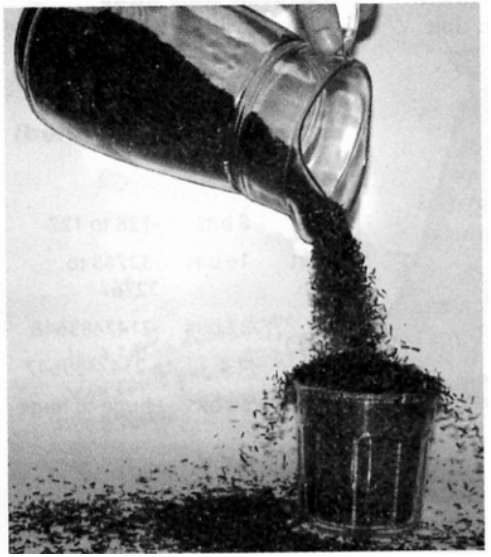
- Types primitifs
  - boolean ***true*** ou ***false***
  - char 16 bits (caractères)
  - Numériques (signés)
    - Entiers
      - byte 8 bits -128 à 127
      - short 16 bits -32 768 à 32 767
      - int 32 bits -2 147 483 648 à 2 147 483 647
      - long 64 bits immense
    - Décimaux
      - float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

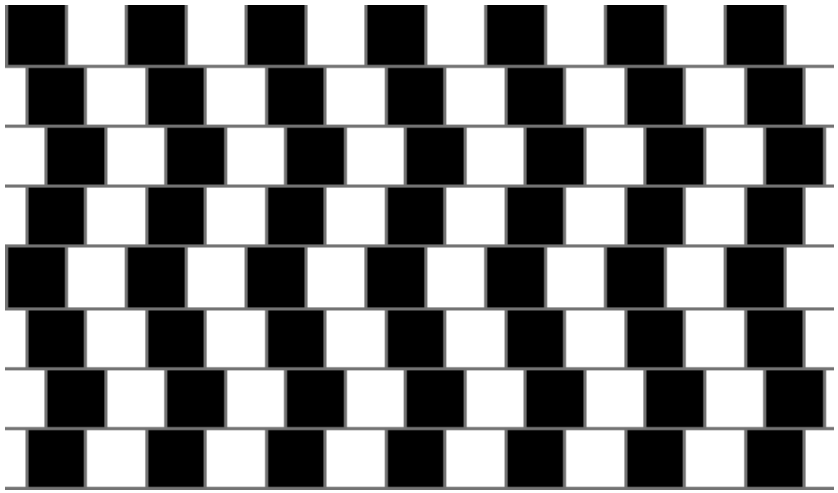
↳ Tester et corriger le programme

Les pièges !

- Attention aux faux-semblants

```
public class Elementary {
    public static void main(String[] args) {
        System.out.println(12345 + 54321);
    }
}
```

↳ Quel résultat s'affiche ?

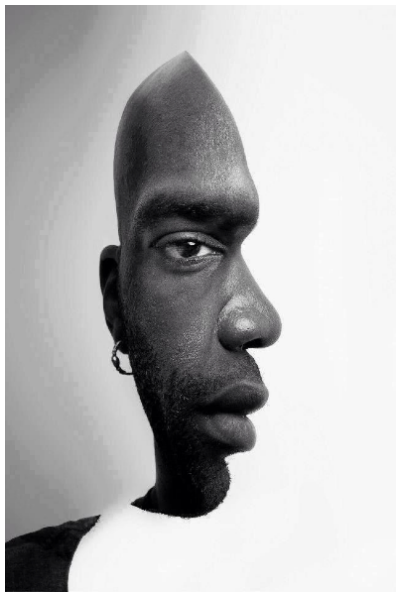


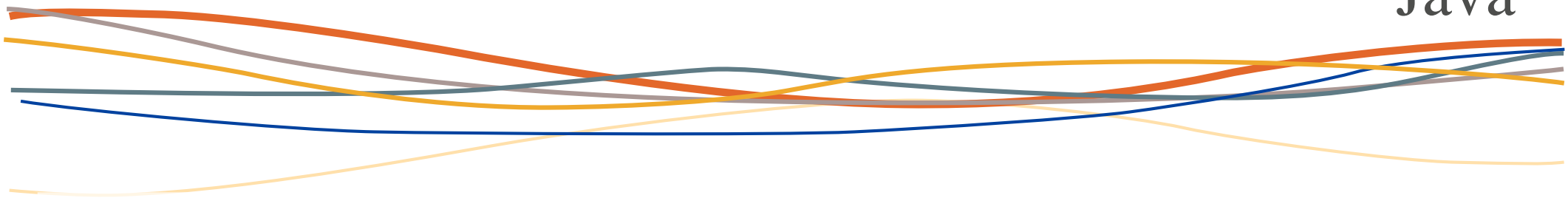
Les pièges !

- Attention aux faux-semblants

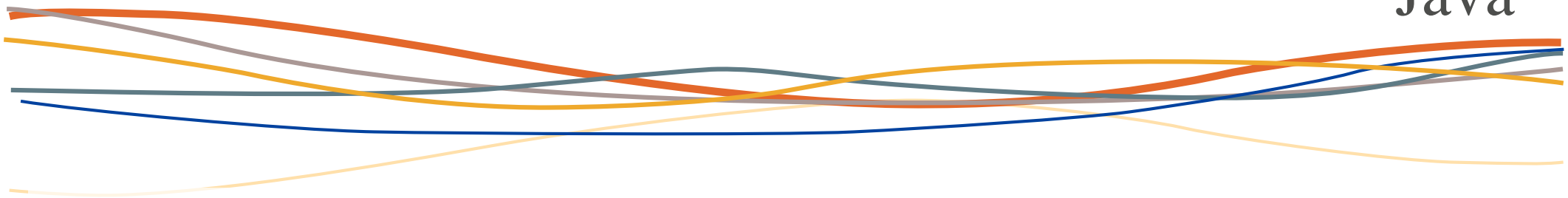
```
public class JoyOfHex {
    public static void main(String[] args) {
        System.out.println(
          Long.toHexString(0x100000000L + 0xcafebabe));
    }
}
```

↳ Quel résultat s'affiche ?

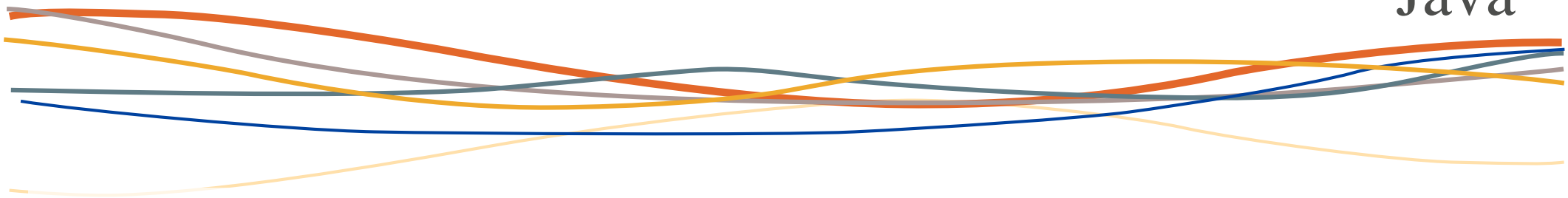




- Classes d'encapsulation (ou classes d'emballage)
  - Permettent d'utiliser un type primitif sous forme d'objet
    - Boolean
    - Number, Byte, Short, Integer, Long, Float, Double
    - Character
  - Ces classes définissent des constantes
    - MIN_VALUE et MAX_VALUE pour les types numériques
    - 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



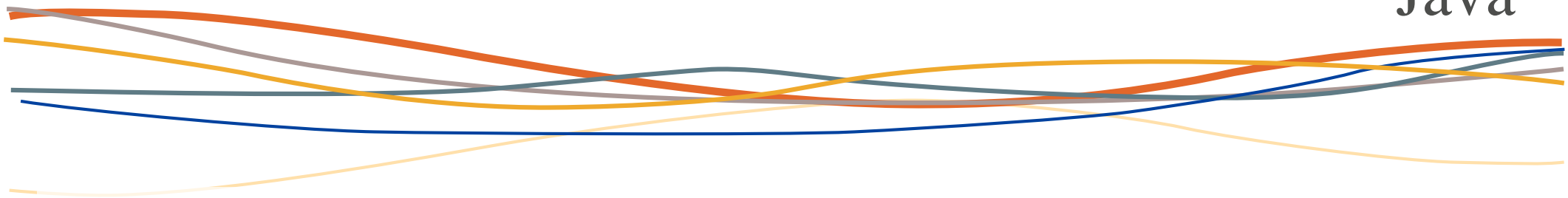
- Classes d'encapsulation (ou classes d'emballage)
  - auto-boxing / auto-unboxing
    - Auto-boxing
      - Integer myInt = 13;
      - remplacé par Integer myInt = Integer.valueOf(13); par le compilateur
    - Auto-unboxing
      - int x = myInt;
      - remplacé par int x = myInt.intValue(); par le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

    - Une correction de cette boucle divise le temps d'exécution par un facteur proche de 6 !
    - 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()



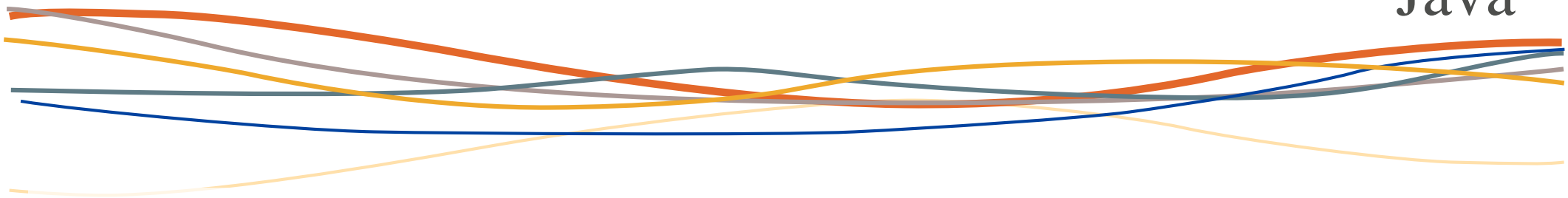
Les pièges des flottants

- Codage IEEE 754 (exemple sur 32 bits - type float)

	exposant								mantisse																					
s	e_7	e_6	e_5	e_4	e_3	e_2	e_1	e_0	m_{22}	m_{21}	m_{20}	m_{19}	m_{18}	m_{17}	m_{16}	m_{15}	m_{14}	m_{13}	m_{12}	m_{11}	m_{10}	m_9	m_8	m_7	m_6	m_5	m_4	m_3	m_2	m_1	m_0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

$$valeur = (-1)^{(1-s)} \times mantisse \times 2^{(exposant\ -\ 127)}$$

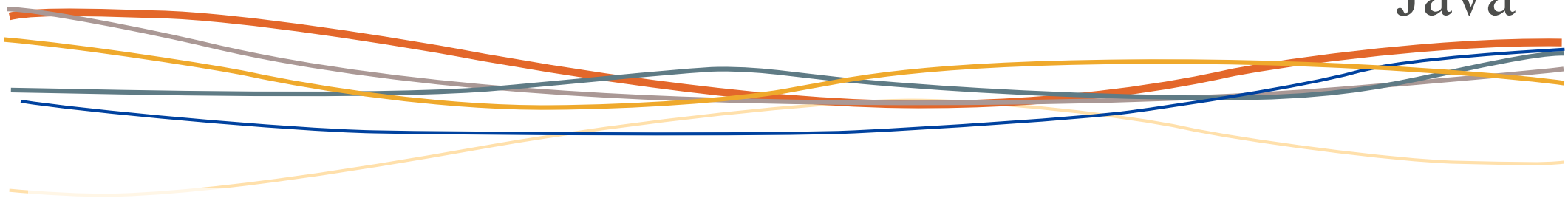
- La mantisse est normalisée pour obtenir un nombre entre 1 et 2 dont le 1 n'est pas explicitement codé dans la représentation binaire.
- Cette règle a des exceptions pour les valeurs extrêmes de l'exposant (0 et 255) et permet de représenter :
  - des très petits nombres
  - l'infini
  - NaN (Not a Number) : résultat d'une opération invalide (0/0, ∞-∞, ∞/∞, 0*∞)



- 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



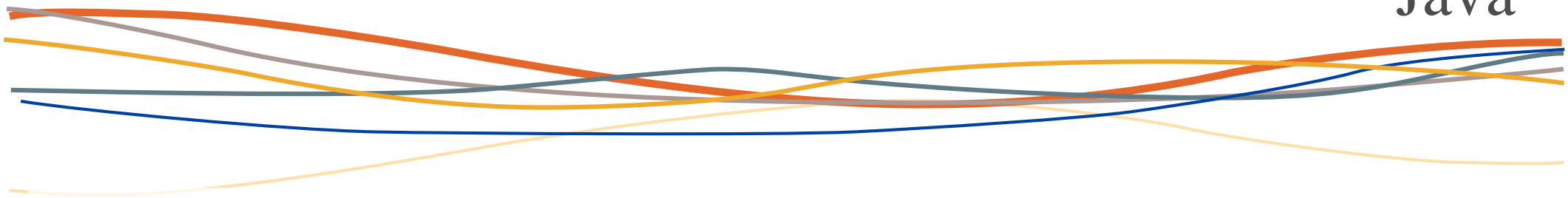
Les pièges des flottants

Classe BigDecimal

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

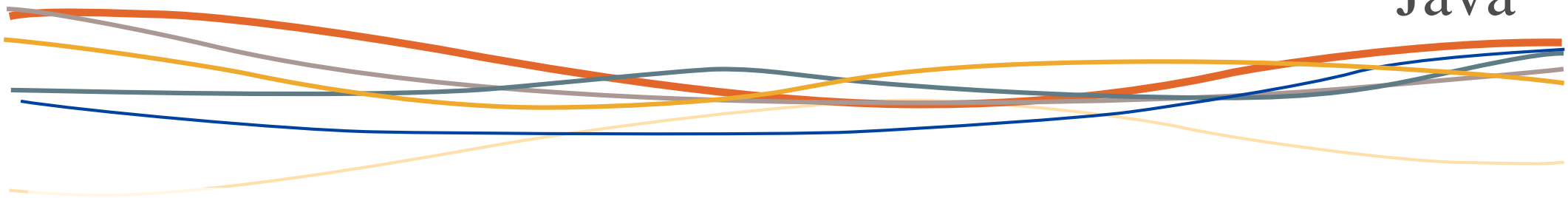
```
import java.math.BigDecimal;

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)

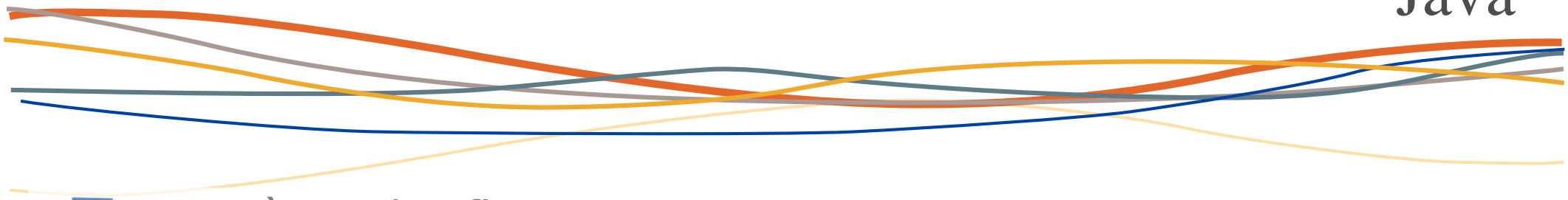


Les pièges des flottants

- 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("WAT !");
        }

        System.out.println(a);

    }

}
```

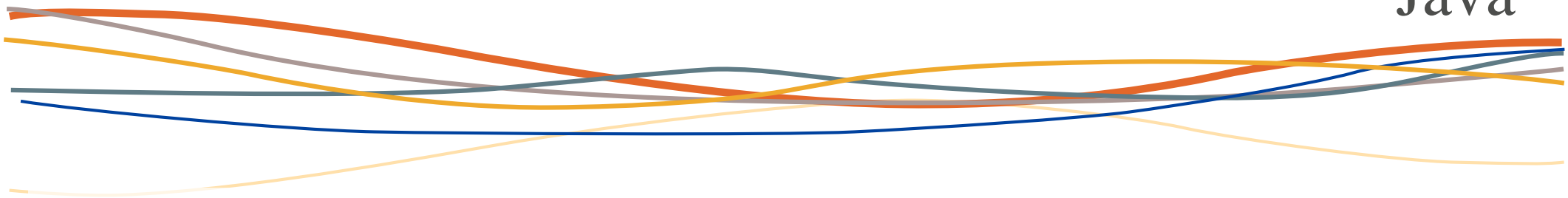
↳ Quel résultat s'affiche ?



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("WAT !");
        }

        System.out.println(a);

    }

}
```



Les pièges des flottants

- Ne jamais utiliser un float en compteur de boucle

```
public class DownForTheCount {
    public static void main(String[] args) {
        final int START = 2000000000;
        int count = 0;

        for(float f = START; (f < START + 50); f++) {
            count++;
        }
        System.out.println(count);
    }
}
```

↳ Quel résultat s'affiche ?

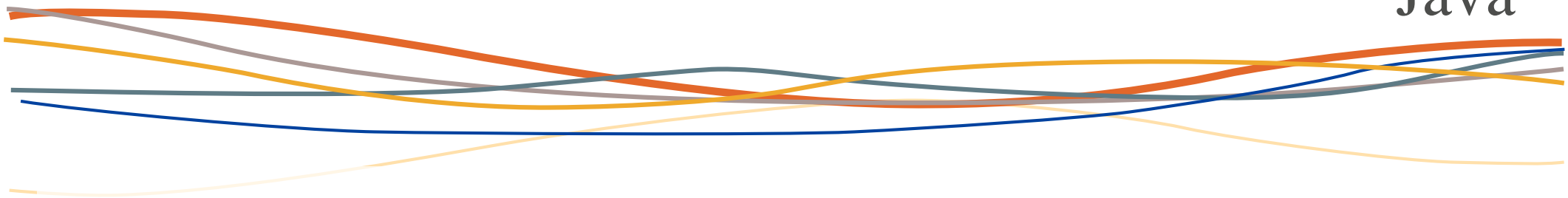
Attention aux arrondis sur les grands nombres !

- 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);
	}
}
```

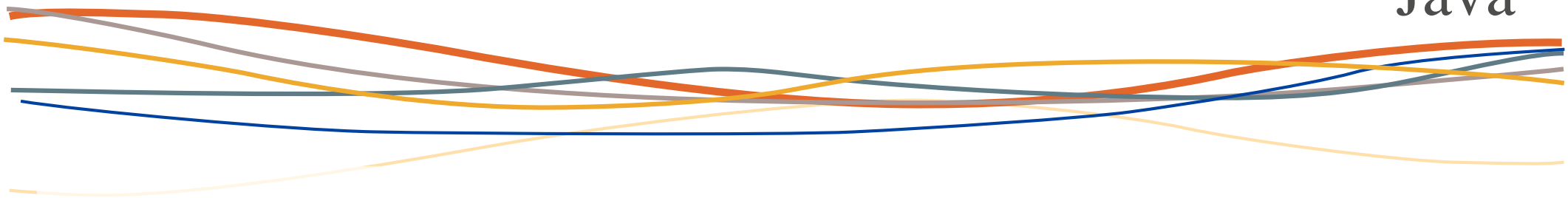
- Types énumérées
  - Les enums sont implémentés comme des classes
  - Ils peuvent posséder
    - un constructeur
    - des méthodes
  - Ecrire une classe PoidsSurPlanetes qui demande la masse sur terre et qui affiche le poids sur toutes les planètes du système solaire (ou pour avoir un résultat plus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

```
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;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    // universal gravitational constant  (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

- Exemple en programmation événementielle
  - Comment associer une action à un appui sur le bouton ?
- 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



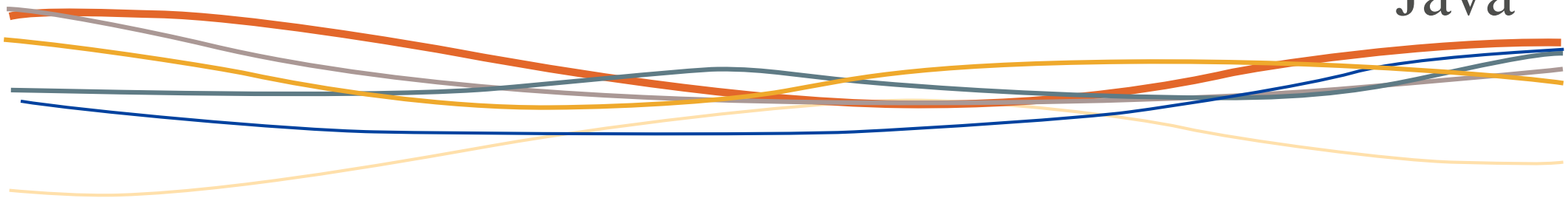
Classes anonymes et listeners

- Associer une action à un bouton (Android)
  - La classe Button définit une méthode setOnClickListener qui prend en paramètre un objet de type View.OnClickListener
  - OnClickListener est une interface (imbriquée dans la classe View) qui définit la méthode
    ```
    void onClick(View view)
    ```
  - Il existe plusieurs alternatives pour associer une action à un bouton (par le code)
    - Implémenter l'interface View.OnClickListener dans la classe de gestion de l'IHM (Activity)
    - Créer une instance d'une classe anonyme implémentant l'interface View.OnClickListener

```java
public class MainActivity extends AppCompatActivity implements
View.OnClickListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button = (Button)findViewById(R.id.button);
        button.setOnClickListener(this);
    }



    @Override
    public void onClick(View view) {
        TextView textView = (TextView) findViewById(R.id.textView);
        textView.setText("Hello world!");
    }
}
```

Ajout de l'objet courant comme "listener" du bouton

Implémentation de l'interface View.OnClickListener



```java
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button button = (Button)findViewById(R.id.button);

        final TextView textView = (TextView) findViewById(R.id.textView);

        button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                textView.setText("Hello world!");
            }
        });
    }

}
```

La variable textView 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 View.OnClickListener

```
final TextView textView = (TextView) findViewById(R.id.textView);

button.setOnClickListener(new View.OnClickListener() {

    @Override
    public void onClick(View view) {
      textView.setText("Hello world!");
    }

});
```

final car textView doit rester liée 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'instance 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)

Autre alternative -> Expression Lambda

- A partir de Java 8, les expressions lambda remplacent avantageusement les classes anonymes

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button button = (Button)findViewById(R.id.button);

        TextView textView = (TextView) findViewById(R.id.textView);

        button.setOnClickListener((View v) -> {
                textView.setText("Hello world!");
        });
    }

}
```

Expression lambda :
bloc de code qui prend en entrée
un paramètre de type View

La variable textView (capturée par l'expression lambda) doit être final ou "effectively final" (on ne ré-écrit pas dedans).