Java Notes

Encapsulation

QUOI : assurer que les détails d'implémentation ne sont pas visibles aux utilisateurs. Les variables d'une classe seront cachées des autres classes, accessibles seulement par les méthodes de la classe courante. C'est ce qu'on appelle la dissimulation de données.

COMMENT déclarez les variables de la classe comme privées et fournissez des méthodes setter et getter publiques pour modifier et afficher les valeurs des variables.

For example:

```
class BankAccount {
  private double balance=0;
  public void deposit(double x) {
    if(x > 0) {
      balance += x;
    }
  }
}
```

Cette mise en œuvre cache la variable du solde, permettant l'accès à celle-ci uniquement par la méthode de dépôt, qui vérifie le montant à déposer avant de modifier la variable.

En résumé, l'encapsulation offre les avantages suivants:
- Contrôle de la façon dont les données sont accessibles ou modifiées
  - Code plus sauvegarde facilement modifié
- Possibilité de modifier une partie du code sans affecter d'autres parties

```
public class Person {
  private int age;
  public void setAge( int age) {
    if (age > 0) {
      this.age = age;
    }
  }
}
```

Rappelez le modificateur d'accès privé, qui rend les membres visibles uniquement pour les sous-classes.

Héritage

L'héritage est le processus qui permet à une classe d'hériter des propriétés (méthodes et variables) d'une autre. Avec l'héritage, l'information est placée dans un entre hiérarchique plus maniable.

```
class MyClass {
  public static void main(String [] args) {
    Dog d = new Dog();
    d.eat();
  }
}
class A {
  public static void main(String args[]) {
    Dog dog = new Dog();
    dog .makeSound();
  }
}
```

```
class Animal {
  protected int legs;
  public void eat() {
    System.out.println("Animal eats");
  }
}

class Dog extends Animal {
  Dog() {
    legs = 4;
  }
}
```

Comme vous pouvez le voir, la classe Dog hérite du champ "legs" de la classe Animal. Nous pouvons maintenant déclarer un objet Dog et appeler la méthode eat ou sa superclasse:

Lorsqu'une classe est héritée d'une autre classe, elle hérite de toutes les variables et méthodes non-privées de la superclasse.

```
class Dog extends Animal {
  // some code
}
```

Ici, Dog est la sous-classe, et Animal est la superclasse.

La classe héritant des propriétés d'un autre est la sous-classe (également appelée classe dérivée ou classe enfant); La classe dont les propriétés sont héritées est la superclasse (classe de base ou classe parent).

Pour hériter d'une classe, utilisez le mot-clé extends. Cet exemple montre comment faire que la classe Dog hérite de la classe Animal.

Vous pouvez accéder à la superclasse de la sous-classe à l'aide du mot-clé super.
Par exemple, super var accède au membre var de la superclasse.

```
Java
class A {
  int var = 0;
  public void p() {
    System.out.println("Var A");
  }
}

class B extends A {
  int var = 0;
  public void p() {
    System.out.println("Var B");
  }
}

class Program {
  public static void main(String[] args) {
    B b = new B();
    b.p();
  }
}

Output
Var B
```

Les constructeurs ne sont pas des méthodes membres, et ne sont donc pas hérités par des sous-classes.
Cependant, le constructeur de la superclasse est appelé lorsque la sous-classe est instanciée.

Polymorphisme

Le polymorphisme, qui se réfère à l'idée de «avoir de nombreuses formes», se produit quand il y a une hiérarchie de classes liées les unes aux autres par héritage.
Un appel à une méthode membre entraînera l'exécution d'une implémentation différente, en fonction du type de l'objet invoquant la méthode.
Voici un exemple: Dog et Cat sont des classes qui héritent de la classe Animal. Chaque classe a sa propre implémentation de la méthode makeSound ().

```
Java
class Animal {
  public void makeSound() {
    System.out.println("Son ...");
  }
}

class Cat extends Animal {
  public void makeSound() {
    System.out.println("Meow");
  }
}

class Dog extends Animal {
  public void makeSound() {
    System.out.println("Woof");
  }
}

class Program {
  public static void main(String[] args) {
    Animal A = new Dog();
    Animal B = new Cat();
    A.makeSound();
    B.makeSound();
  }
}
```

```
a.makeSound();
//Outputs "Woof"

b.makeSound();
//Outputs "Meow"
```

Comme la variable de référence a contient un objet Dog, la méthode makeSound () de la classe Dog sera appelée. Idem va de même pour la variable b.

Cela démontre que vous pouvez utiliser la variable Animal sans savoir qu'il contient un objet de la sous-classe.
Cela est très utile lorsque vous avez plusieurs sous-classes de la superclasse.

```
public static void main(String[] args) {
  Animal a = new Dog();
  Animal b = new Cat();
}
```

Comme vous les objets Cat et Dog sont des objets Animal, nous pouvons faire ce qui suit dans main: Nous avons créé deux variables de référence de type Animal et les avons pointées vers les objets Cat et Dog.
Maintenant, nous pouvons appeler les méthodes makeSound ().

Overriding & Overloading

Méthode de substitution: overriding

Comme nous l'avons vu dans la leçon précédente, une sous-classe peut définir un comportement spécifique au type de sous-classe, ce qui signifie qu'une sous-classe peut implémenter une méthode de classe parent en fonction de son exigence.
Cette fonctionnalité est connue sous le nom de méthode de substitution.

```
Java
class Animal {
  public void makeSound() {
    System.out.println("Son ...");
  }
}

class Cat extends Animal {
  public void makeSound() {
    System.out.println("Meow");
  }
}

class Program {
  public static void main(String[] args) {
    Cat C = new Cat();
    C.makeSound();
  }
}
```

Dans le code ci-dessus, la classe Cat remplace la méthode makeSound () de sa super-classe Animal.

Règles pour la substitution de méthodes:
- Doivent avoir le même type de retour et arguments
- Le niveau d'accès ne peut pas être plus restrictif que le niveau d'accès de la méthode surchargée (Exemple: si la méthode de superclasse est déclarée publique, la méthode de substitution dans la sous-classe ne peut ni être privée ni protégée).
  - Une méthode déclarée définitive ou statique ne peut pas être remplacée
  - Si une méthode ne peut pas être héritée, elle ne peut pas être remplacée
    - Les constructeurs ne peuvent pas être ignorés

Le dépassement de méthode est également connu sous le nom de polymorphisme d'exécution.

Surcharge de méthode: overloading

Lorsque les méthodes ont le même nom, mais des paramètres différents, il est connu comme la méthode de surcharge.
Cela peut être très utile pour les classes qui ont le même fonctionnalité de méthode pour différents types de paramètres.
L'exemple suivant illustre une méthode qui renvoie le maximum de ses deux paramètres.

```
int max(int a, int b) {
  if(a > b) {
    return a;
  }
  else {
    return b;
  }
}
```

La méthode ici ne fonctionnera que pour les paramètres de type entier. Si vous souhaitez utiliser des nombres à - Flottant pour les doubles, aussi bien. Pour cela, vous devez surcharger la méthode max:

```
Java
class Program {
  static double max(double m, double n) {
    if (m > n) {
      return m;
    }
    else {
      return n;
    }
  }
  static int max(int m, int n) {
    if (m > n) {
      return m;
    }
    else {
      return n;
    }
  }

  public static void main(String[] args) {
    System.out.println("Max: " + max(5, 10));
    System.out.println("Max: " + max(7.5, 12.5));
  }
}
```

Maintenant, notre méthode max travaillera aussi avec les doubles.
Une méthode surchargée doit avoir une liste d'arguments différente; Les paramètres doivent être différents selon leur type, leur nombre ou les deux.
Un autre nom pour la surcharge de méthode est le polymorphisme en temps de compilation.

Abstraction

L'abstraction de données n'offre au monde extérieur que des informations essentielles, dans le cadre d'un processus de représentation de caractéristiques essentielles sans inclure de détails d'implémentation.
Un bon exemple du monde réel est un livre. Lorsque vous entendez le terme livre, vous ne connaissez pas les détails précis, comme le nombre de pages, la couleur ou la taille, mais vous comprenez l'idée ou l'abstraction d'un livre.
Le concept d'abstraction est que nous nous concentrons sur les qualités essentielles, plutôt que sur les caractéristiques spécifiques d'un exemple particulier.

En java, l'abstraction est réalisée en utilisant des classes abstraites et des interfaces.
- Une classe abstraite est définie à l'aide du mot-clé abstract.
- Si une classe est déclarée abstraite, elle ne peut pas être instanciée (vous ne pouvez pas créer d'objets de ce type).
  - Pour utiliser une classe abstraite, vous devez l'hériter d'une autre classe.
  - Toute classe contenant une méthode abstraite doit être définie comme abstraite.

Une méthode abstraite est une méthode qui est déclarée sans une implémentation (sans accolades, et suivie d'un point-virgule) abstract void eat();

Abstract Classes

Par exemple, nous pouvons définir notre classe Animal comme abstraite:

```
abstract class Animal {
  int legs = 0;
  abstract void makeSound();
}
```

```
class Cat extends Animal {
  public void makeSound() {
    System.out.println("Meow");
  }
}
```

La méthode makeSound est également abstraite, car elle n'a aucune implémentation dans la superclasse.
Nous pouvons hériter de la classe Animal et définir la méthode makeSound () pour la sous-classe.

```
abstract class Animal {
  public int age;
  public abstract int printAge();
}

class Dog extends Animal {
  public int printAge() {
    return age;
  }
}
```

Chaque animal fait un bruit, mais chacun a une manière différente de le faire. C'est pourquoi nous différencions une classe abstraite Animal, et laissons la mise en œuvre de la façon dont ils font des sons aux sous-classes.
Ceci est utilisé lorsque vous ne définissez significative de la méthode dans la superclasse.

Interfaces

Une interface est une classe complètement abstraite qui ne contient que des méthodes abstraites.
Quelques spécifications pour les interfaces:
- Définit à l'aide du mot-clé interface.
  - Peut contenir des méthodes abstraites et statiques finales.
- Impossible de contenir un constructeur ou des variables non finales.
  - Les interfaces peuvent étendre d'autres interfaces.
- Une classe peut implémenter n'importe quel nombre d'interfaces.

```
interface Animal {
  public void eat();
  public void makeSound();
}
```

Une interface est une classe abstraite qui ne contient que des méthodes abstraites.
Les interfaces ont les propriétés suivantes:
- Une interface est implicitement abstraite. Vous n'avez pas besoin d'utiliser le mot-clé abstract tout en déclarant une interface.
- Chaque méthode dans une interface est également implicitement abstraite, donc le mot-clé abstract n'est pas nécessaire.
- Les méthodes dans une interface sont implicitement publiques.

```
interface Animal {
}

class Cat implements Animal {
  public void eat() {
    System.out.println("Cat eats");
  }
}
```

Lorsque vous implémentez une interface, vous devez remplacer toutes les méthodes.

```
Java
interface Animal {
  public void eat();
  public void makeSound();
}

class Cat implements Animal {
  public void eat() {
    System.out.println("Meow");
  }
  public void eat() {
    System.out.println("ronronne");
  }

  public static void main(String[] args) {
    Cat c = new Cat();
    c.eat();
    c.makeSound();
  }
}
```

Utilisez le mot-clé implements pour utiliser une interface avec votre classe.

Une classe peut hériter d'une seule superclasse, mais peut implémenter plusieurs interfaces!

Exercices

Object variables store...

- References

Fill in the blanks to define a new class Falcon, based on the superclass Bird.

```
class Falcon extends Bird {
}
```

What is the output of this code?

```
class A {
  private void p() {
    System.out.println("A");
  }

  private void print(String str) {
    System.out.println("A");
  }

  private void print(int[] a) {
    System.out.println("A");
  }

  public static void main(String[] args) {
    A object = new A();
    object.print(5.0);
  }
}
```

C

What is the purpose of this code?

```
class Program {
  public static void main(String[] args) {
    int[] ints = {1, 2, 3};
    int[] ints2 = ints;
    ints[0] = 5;
    System.out.println(ints2[0]);
  }
}
```

What is the output of this code?

```
class Program {
  public static void main(String[] args) {
    int[] ints = {1, 2, 3};
    int[] ints2 = ints.clone();
    ints[0] = 5;
    System.out.println(ints2[0]);
  }
}
```

What is used for hiding the details of an object from the other parts of a program?

- Inheritance
- Polymorphism
- Encapsulation

A class Car and its subclass BMW each have a method run(), which was written by the developer as part of the class definition. If CarObj refers to an object of type BMW, what will CarObj.run(); do?

- The compiler will complain that run() has been defined twice.
- The run() method defined in BMW will be called.

Valentine, Holiday, and Birthday inherit from the class Card. In order for the following code to be correct, what type must the reference variable card be?

```
card = new Valentine("B", 14);
card.greeting();

card = new Holiday("B");
card.greeting();

card = new Birthday("C", 12);
card.greeting();
```

- Birthday
- Valentine
- Card

Casting

L'attribution d'une valeur d'un type à une variable d'un autre type est appelée Casting de type.
Pour lancer une valeur dans un type spécifique, placez le type entre parenthèses et positionnez-le devant la valeur.

```
Java
public class Program {
  public static void main(String[] args) {
    int a = (int) 3.14;
    System.out.println(a);
  }
}
```

Le code ci-dessus coste la valeur 3.14 à un entier, avec 3 comme la valeur résultante.

```
Java
public class Program {
  public static void main(String[] args) {
    double d = 42.375;
    int b = (int) d;
    System.out.println(b);
  }
}
```

Autre exemple
Java supporte le moulage automatique de type de nombres entiers vers des points flottants, car il n'y a aucune perte de précision.
D'autre part, le moulage de type est obligatoire lors de l'attribution de valeurs en virgule flottante à des variables entières.

Downcasting

Pour les classes, il existe deux types de casting:

Upcasting

Vous pouvez lancer une instance d'une sous-classe dans sa superclasse.
Considérons l'exemple suivant, en supposant que Cat est une sous-classe d'Animal

```
Animal a = new Cat();
```

Java s'automatiquement mis à jour la variable de type Cat pour le type Animal.

Downcasting

La diffusion d'un objet d'une superclasse dans sa sous-classe s'appelle downcasting.

```
Animal a = new Animal();
((Cat)a).makeSound();
```

Cela va essayer de lancer la variable a au type Cat et appeler sa méthode makeSound ().
Pourquoi est-upcasting automatique, downcasting manuel? Eh bien, upcasting ne peut jamais échouer. Mais si vous avez un groupe de différents animaux et que vous voulez jeu faire descendre tous à un chat, alors il y a une chance que certains de ces animaux sont en fait des chiens, de sorte que le processus échoue.

Anonymous Classes

Les classes anonymes sont un moyen d'étendre les classes existantes à la volée.

```
class Machine {
  public void start() {
    System.out.println("Starting...");
  }
}
```

Par exemple considérons que nous avons une classe machine

Lors de la création de l'objet Machine, nous pouvons changer la méthode de démarrage à la volée.

```
Java
class Machine {
  public void start() {
    System.out.println("Starting...");
  }
}

class Program {
  public static void main(String[] args) {
    Machine m = new Machine();
    m.start();
    Machine m2 = new Machine() {
      @Override public void start() {
        System.out.println("Moooo");
      }
    };
    m2.start();
  }
}
```

Après l'appel constructeur, nous avons ouvert les accolades et avons surchargé la mise en œuvre de la méthode de démarrage à la volée.
L'annotation @Override est utilisée pour rendre votre code plus facile à comprendre, car il est plus évident lorsque les méthodes sont remplacées.

La modification s'applique uniquement à l'objet courant, et non à la classe «Machine». Donc, si nous créons un autre objet de cette classe, l'implémentation de la méthode start sera celle définie dans la classe.

```
Java
class Machine {
  public void start() {
    System.out.println("Starting...");
  }
}

class Program {
  public static void main(String[] args) {
    Machine m = new Machine();
    m.start();
    Machine m2 = new Machine() {
      @Override public void start() {
        System.out.println("Moooo");
      }
    };
    m2.start();
  }
}
```

```
Output
Starting...
```

Fill in the blanks to print "Hello":

```
class A {
  public void print() {
    System.out.print("A");
  }
}

class B {
  public static void main(String[] args) {
    A object = (A) new A() {
      @Override public void print() {
        System.out.print("Hello ");
      }
    };
    object.print();
  }
}
```

Inner Classes

Inner = interne, Java prend en charge les classes de nidification; Une classe peut être membre d'une autre classe.
Créer une classe interne est assez simple. Il suffit d'écrire une classe dans une classe. Contrairement à une classe, une classe interne peut être privée. Une fois que vous déclarez une classe interne privée, elle ne peut pas être consultée à partir d'un objet en dehors de la classe.

```
Java
class Robot {
  boolean[] isOn;
  int[] x, y;
  int[] dx, dy;
  int[] rx, ry;
  int n;

  boolean[] canPrintArg() {
    return isOn;
  }

  void[] move() {
    for (int i = 0; i < n; ++i) {
      int[] nx = x[i] + dx[i], ny = y[i] + dy[i];
      if (canPrintArg()[i] && (nx == rx || ny == ry
        System.out.printf("%d %d\n", rx, ry);
      }
      x[i] = nx;
      y[i] = ny;
    }
  }

  public static void main(String[] args) {
    Robot robot = new Robot();
    int[] ans = robot.move();
  }
}
```

```
Output
I is thinking
```

La classe Robot a une classe interne Cerveau. La classe interne peut accéder à toutes les variables membres et méthodes de sa classe externe, mais il ne peut pas être consulté à partir de n'importe quelle classe extérieure.

Remarque: le code ci-dessus ne sera pas compilé, car le fichier a plusieurs classes publiques. Vous devez le diviser en plusieurs fichiers pour qu'il fonctionne.

```
public class Robot {
  int[] x, y;
  int[] dx, dy;
  int[] rx, ry;
  int n;

  boolean[] canPrintArg() {
    return isOn;
  }

  void[] move() {
    int[] nx = x[i] + dx[i], ny = y[i] + dy[i];
    if (canPrintArg()[i] && (nx == rx || ny == ry
      System.out.printf("%d %d\n", rx, ry);
    }
    x[i] = nx;
    y[i] = ny;
  }

  public static void main(Robot[] a) {
    int[] ans = a[0].move();
  }
}
```

The Equals() Method

Comparaison des objets

N'oublier pas que lorsque vous créez des objets, les variables stockent des références aux objets.
Ainsi, lorsque vous comparez des objets à l'aide de l'opérateur de test d'égalité (==), il compare les références et non les valeurs d'objet

```
Java
class Animal {
  String name;
  boolean[] isAlive;

  public static void main(String[] args) {
    Animal a0 = new Animal("Kitty");
    Animal a1 = new Animal("Kitty");
    System.out.print(a0.name == a1.name);
  }
}
```

```
Output
false
```

En objet de deux objets portant le même nom, le test d'égalité renvoie faux, car nous avons deux objets différents (deux références au emplacements de mémoire différents).

```
Java
class Program {
  int[] ints = {1, 2, 3};
  int[] ints2 = ints;
  int[] ints3 = ints.clone();

  public static void main(String[] args) {
    System.out.print(ints == ints2);
    System.out.print(ints == ints3);
  }
}
```

```
Output
true false
```

équivaut à()

Chaque objet a une méthode prédéfinie equals() qui est utilisée pour les tests d'égalité sémantique.
Mais, pour que cela fonctionne pour nos classes, nous devons les remplacer et vérifier les conditions dont nous avons besoin.
Il existe un moyen simple et rapide de générer la méthode equals(), autre que de l'écrire manuellement.
Juste un clic droit dans votre classe, allez à Source> Generate hashCode () and equals () ...

Open Files	Actions
Foo.class	Generate Implement Methods...
Foo.class	Generate Delegate Methods...
Foo.class	Generate Unique Members...
Foo.class	Generate hashCode() and equals()
Foo.class	Generate toString()
Foo.class	Generate Constructors using Fields...
Foo.class	Generate Constructors from Superclass...
Foo.class	Generate Getters and Setters
Foo.class	Generate Wrapper Constructors
Foo.class	Generate Delegate Methods...

```
Fichier Edition Source Refactoring Navigation Search Compilation Exécution Fenêtre Aide
Foo.class
class A {
  private int[] ints = {1, 2, 3};
  int[] ints2 = ints;

  public static void main(String[] args) {
    A a = new A();
    a.ints[0] = 5;
    System.out.print(a.ints2[0]);
  }
}
```

Fichier Edition Source Refactoring Navigation Search Compilation Exécution Fenêtre Aide
Foo.class
class A {
  private int[] ints = {1, 2, 3};
  int[] ints2 = ints.clone();

  public static void main(String[] args) {
    A a = new A();
    a.ints[0] = 5;
    System.out.print(a.ints2[0]);
  }
}

```
Java
class Animal {
  boolean[] isAlive;
  String name;

  @Override
  public int hashCode() {
    int result = 0;
    int prime = 31;
    result = prime * result + (name == null ? 0 : name.hashCode());
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Animal other = (Animal) obj;
    if (name == null)
      if (other.name != null)
        return false;
    else if (!name.equals(other.name))
      return false;
    return true;
  }
}
```

This will automatically create the necessary methods.

```
class Animal {
  boolean[] isAlive;
  String name;

  @Override
  public int hashCode() {
    int result = 0;
    int prime = 31;
    result = prime * result + (name == null ? 0 : name.hashCode());
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Animal other = (Animal) obj;
    if (name == null)
      if (other.name != null)
        return false;
    else if (!name.equals(other.name))
      return false;
    return true;
  }
}
```

La méthode hashCode () générée automatiquement est utilisée pour déterminer où stocker l'objet en interne. Chaque fois que vous implémentez equals, vous DEVEZ également remplacer hashCode. Nous pouvons exécuter le test à nouveau, en utilisant la méthode égale:

```
Java
class Animal {
  boolean[] isAlive;
  String name;

  @Override
  public int hashCode() {
    int result = 0;
    int prime = 31;
    result = prime * result + (name == null ? 0 : name.hashCode());
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Animal other = (Animal) obj;
    if (name == null)
      if (other.name != null)
        return false;
    else if (!name.equals(other.name))
      return false;
    return true;
  }

  public static void main(Animal[] a) {
    Animal a = new Animal("Kitty");
    Animal a2 = new Animal("Kitty");
    System.out.print(a.equals(a2));
  }
}
```

```
Output
true
```

Vous pouvez utiliser le même menu pour générer d'autres méthodes utiles, telles que getters et setters pour vos attributs de classe.

Enums

Enums
Un Enum est un type spécial utilisé pour définir des collections de constantes.
Voici un exemple simple d'Enum:

```
enum Rank {
  SOLDIER,
  SERGEANT,
  CAPTAIN
}
```

Notez que les valeurs sont séparées par des virgules.
Vous pouvez vous référer aux constantes dans l'énumération Rank avec la syntaxe point.

```
Rank r = Rank.SOLDIER;
```

Fondamentalement, les Enums définissent des variables qui représentent des membres d'un ensemble fixe.

Après avoir déclaré un Enum, nous pouvons vérifier les valeurs correspondantes avec, par exemple, une instruction switch.

```
Java
class Program {
  enum Rank {
    SOLDIER,
    SERGEANT,
    CAPTAIN
  }

  public static void main(Rank[] ranks) {
    for (Rank r : ranks) {
      switch (r) {
        case SOLDIER:
        System.out.print("Sergent sous ordre");
        break;
        case SERGEANT:
        System.out.print("Sergent sous ordre");
        break;
        case CAPTAIN:
        System.out.print("Capitaine sous ordre");
        break;
      }
    }
  }
}
```

```
Output
Soldier says hi!
```

Vous devez toujours utiliser Enums quand une variable (en particulier un paramètre de méthode) ne peut en prendre qu'une sur un petit ensemble de valeurs fixes au moment de la compilation.
Si vous utilisez Enums à la place, vous contrôlez (ou du codes de chaînes), vous augmentez la vérification en temps de compilation et évitez que des erreurs ne soient répercutées dans des constantes non valides, et vous documentez quelles valeurs sont légales à utiliser.

Certains exemples d'utilisation d'Enum incluent les noms de mois, les jours de la semaine, le jeu de cartes, etc.

```
What is the output of this code?
enum Color {
  RED, BLUE, GREEN;
}

class PrintColor {
  public static void main(String[] args) {
    Color[] c = Color.values();
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i] + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].ordinal() + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].name() + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].toString() + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[0]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].compareTo(c[0]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].compareTo(c[1]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[i]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+1)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+2)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+3)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+4)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+5)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+6)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+7)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+8)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+9)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+10)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+11)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+12)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+13)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+14)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+15)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+16)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+17)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+18)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+19)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+20)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+21)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+22)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+23)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+24)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+25)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+26)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+27)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+28)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+29)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+30)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+31)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+32)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+33)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+34)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+35)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+36)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+37)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+38)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+39)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+40)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+41)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+42)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+43)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+44)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+45)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+46)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+47)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+48)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+49)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+50)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+51)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+52)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+53)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+54)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+55)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+56)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+57)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+58)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+59)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+60)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+61)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+62)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+63)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+64)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+65)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+66)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+67)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+68)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+69)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+70)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+71)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+72)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+73)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+74)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+75)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+76)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+77)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+78)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+79)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+80)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+81)%3]) + " ");
    }
    System.out.print("\n");
    for (int i = 0; i < c.length; i++) {
      System.out.print(c[i].equals(c[(i+82)%3]) + " ");
    }
    System.out.print("\n");
    for (int i
```