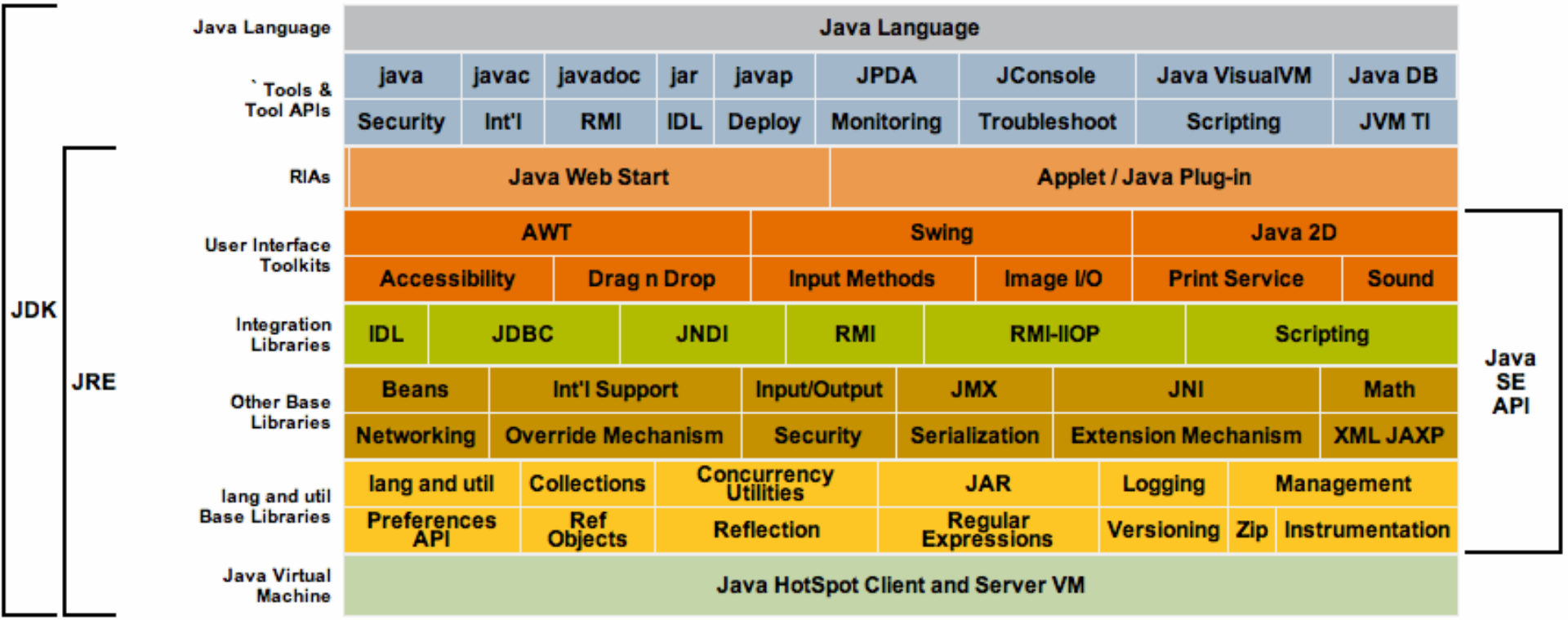
JCertif university

Ateliers SCJP
Pour Java 2 Platform, Standard Edition 7.0

Diagramme JDK 1.7



JCertif university

Agenda

- ☑ Declaration and Access Control
- ❑ Object Orientation
- ❑ Assignments
- ❑ Operators
- ❑ Flow Control, Exceptions and Assertions
- ❑ Strings, I/O, Formatting, and Parsing
- ❑ Generics and Collections
- ❑ Inner Classes
- ❑ Threads
- ❑ The exam

JCertif university

Identifiers

- Le nom d'un identifiant doit débuter par une lettre, « $ » ou « _ ».
- Les identifiants sont « case sensitive ».
- Il ne doit pas être identique à un mot-clé.

JCertif university

Keywords

abstract	class	false	import	package	super	try
assert	const	final	instanceof	private	switch	void
boolean	continue	finally	int	protected	synchronized	volatile
break	default	float	interface	public	this	while
byte	do	for	long	return	throw	enum
case	double	goto	native	short	throws	
catch	else	if	new	static	transient	
char	extends	implements	null	strictfp	true	

JCertif university

Source Files

- Un fichier source peut contenir plusieurs classes – au moins une de ces classes doit être déclarée publique et le fichier *.java* doit porter son nom.
- Un fichier source est composé de :
  - package statements
  - import statements
  - class definitions
- Le nom d'un package ne peut être composé de symboles (ex.: /, \, ' ', etc.).
- L'utilisation de l'étoile (*) dans les import n'a aucun impact sur la performance d'exécution de la classe.

JCertif university

Access Modifiers

- public
  - Le plus permissif : peut être accédé de n'importe quelle classe.
  - Il s'applique aux classes, aux variables (d'instance et de classe) ainsi qu'aux méthodes.
- private
  - Le plus restrictif.
  - Il ne s'applique qu'aux variables (d'instance et de classe) ainsi qu'aux méthodes - une classe ne peut pas être déclarée privée.
  - Une variable/méthode est visible seulement au niveau de la classe où elle est déclarée.
  - Il permet de « cacher » une variable/méthode.

Access Modifiers

- default
  - Visibilité attribuée par défaut (pas un mot-clé).
  - Il s'applique aux variables (d'instance et de classe), aux classes et aux méthodes.
  - Même visibilité que public, mais au niveau du package seulement.
  - Les classes situées dans un même répertoire constituent un package.
  - default = visibilité public au niveau du package uniquement, alors que public = visible partout peu importe le package.
  - Une classe peut hériter d'une classe située dans un autre package mais ne peut pas accéder à ses variables et méthodes « default » - juste les variables public ou protected.

Access Modifiers

- protected
  - Il se rapproche plus du private que du public.
  - protected est plus accessible que default.
  - Il s'applique seulement aux variables (d'instance et de classe) ainsi qu'aux méthodes.
  - Il rend disponible les méthodes et les variables aux classes enfant (héritage) – visibilité au niveau des enfants même si ceux-ci sont situés dans un autre package.

private → **default** → **protected** → **public**

JCertif university

L'héritage et sa visibilité

- Une méthode ne peut pas être surchargée pour être plus privée qu'elle ne l'est déjà
  - Si une méthode est déclarée public, elle ne peut pas être surchargée par une méthode de moindre visibilité tel que default, protected ou private.
  - Dans le cas contraire, on peut augmenter la visibilité d'une méthode lors d'une surcharge :

private –> default -> protected -> public

JCertif university

Final

- Final s'applique aux méthodes, aux classes et aux variables.
- Idem à « lecture seule » (read only)
- Utilisé généralement pour identifier une constante.
- Une classe final ne peut pas être héritée.
- Le contenu d'une variable final ne peut pas être modifié une fois celui-ci assigné : on peut l'assigner après sa déclaration mais une fois assigné, on ne peut plus modifier son contenu.
- Une méthode final ne peut pas être surchargée.
- Augmente la performance des applications.

Abstract

- Il ne peut être appliqué qu'aux méthodes et aux classes.
- Une classe abstraite ne peut pas être instanciée.
- Force une classe enfant à implémenter une méthode non définie dans la classe parent.
- Une méthode abstraite n'a pas de contenu, mais seulement une signature.
  - ex.: abstract methode(); // pas d'accolade ouvrante ni fermante
- Une classe enfant qui hérite d'une classe parent abstraite doit implémenter la ou les méthodes abstraites de la classe parent, sinon, elle doit être définie elle aussi comme abstraite.
- Si une classe contient une ou plusieurs méthodes abstraites, elle doit obligatoirement être déclarée abstraite.
- Si une classe implémente une interface, mais qu'elle n'implémente pas toutes les méthodes déclarées dans celle-ci, elle doit être déclarée abstraite.

JCertif university

Native

- Il ne s'applique qu'aux méthodes uniquement.
- Il indique que la méthode est implémentée ailleurs
  - Pas dans une classe enfant, mais dans une librairie externe – fonctionnalité de JNI (Java Native Interface) -> communication entre Java et C / C++
- On utilise généralement native avec un bloc *static initializers* pour charger une librairie dynamique en mémoire (.dll, .so, .sl, etc) qui contient la définition de la méthode dans un autre langage.
- Les méthodes clone() et notify() sont natives dans la classe Object.

JCertif university

Synchronized

- *synchronized* sera vu plus en détails sur le chapitre portant sur les Threads

JCertif university

Utilisation des Modifiers

Modifier	Class	Variable	Method	Constructor	Bloc de code flottant
public	oui	oui	oui	oui	non
protected	non	oui	oui	oui	non
(default)*	oui	oui	oui	oui	oui
private	non	oui	oui	oui	non
final	oui	oui	oui	non	non
abstract	oui	non	oui	non	non
static	non	oui	oui	non	oui
native	non	non	oui	non	non
synchronized	non	non	oui	non	oui

* = default n'est pas un mot-clé

JCertif university

Importing

- import = appel à une classe externe sans spécifier son nom complet
- static import = appel à un membre publique (public, default, protected) d'une classe sans spécifier le nom de la classe à laquelle il appartient :
  - resultat.setMax(Constantes.MAX_RESULT);
  - resultat.setMax(MAX_RESULT);
- Comment déclarer un *static import* :
  - import static projet.Constantes.MAX_RESULT; ou
  - import static projet.Constantes.*;
- Au lieu de
  - import projet.Constantes;

JCertif university

Importing (suite)

- Aucun impact négatif sur la performance (seulement à la compilation).
- Utiliser le moins souvent possible; code plus complexe à maintenir.

JCertif university

Class Fundamentals

- Le CLASSPATH est composé de la valeur de la variable d'environnement + arguments *–cp* et *-classpath*
- Il faut connaître la signature de la méthode *main*:
  - public static void main(String args[])
- Trois types de variables :
  - *member* : appartiennent à une classe
    - détruites au déchargement de la classe
  - *automatic* : appartiennent à une méthode
    - détruites à la fin de la méthode
  - *class* : variables *static* d'une classe
    - détruites au déchargement de la classe

JCertif university

Class Fundamentals (suite)

- Les variables *member* ont une valeur par défaut qui leur est associée alors que ce n'est pas le cas des variables *automatic*
  - une erreur de compilation va survenir si la variable *automatic* n'est pas initialisée
- Les arguments sont toujours passés par valeur, DONC on peut changer le contenu d'un objet (pas son adresse, seulement son contenu).

JCertif university

Agenda

- ☑ Declaration and Access Control
- ☑ Object Orientation
- ❑ Assignments
- ❑ Operators
- ❑ Flow Control, Exceptions and Assertions
- ❑ Strings, I/O, Formatting, and Parsing
- ❑ Generics and Collections
- ❑ Inner Classes
- ❑ Threads
- ❑ The exam

Héritage, Is-A, Has-A

- *Is-A* fait référence à la super classe
  - Cercle « *Is-A* » une Forme
  - class Cercle **extends** Forme
- *Has-A* fait référence à une variable d'instance
  - Cercle « Has-A » diamètre
  - class Cercle extends Forme

```
{
    Integer diametre;
    …
}
```

JCertif university

Overriding / Overloading

- Réutiliser le même nom de méthode avec les mêmes arguments et type de retour s'appelle *overriding*
  - remplacer une méthode parent
- Réutiliser le même nom de méthode, mais avec des arguments différents et peut-être même un type de retour différent s'appelle *overloading*
  - créer une nouvelle méthode ayant le même nom qu'une méthode existante, mais qui exploite des arguments différents pour effectuer le même type de travail

JCertif university

Overriding

- *overriding* permet de redéfinir une méthode provenant d'une classe parent
- Pour chevaucher une méthode (*overriding)* il faut:
  - avoir le même type de retour ou retourner une sous-classe du type de retour
  - avoir le même nom
  - avoir les mêmes arguments (même type et définis dans le même ordre)
  - ne doit pas restreindre l'accessibilité de la méthode parent
    - private -> default -> protected -> public
  - ne doit pas lancer d'exceptions qui ne sont pas compatibles avec celle(s) déclarée(s) dans la méthode chevauchée

JCertif university

Overloading

- Une méthode s'identifie par son nom et la séquence exacte de ses arguments (leur type).
- On ne peut pas surcharger une méthode simplement en changeant son type de retour.
- *overloading* signifie réutiliser un nom de méthode identique pour effectuer un travail similaire, mais à partir d'arguments de types différents.
- Les méthodes surchargées (*overloading*) peuvent avoir un type de retour différent.
- Une méthode surchargée peut appeler une autre méthode surchargée (c'est le nombre, l'ordre et le type des arguments qui détermine la méthode à exécuter).

JCertif university

Paramètres variables (…)

- « … » est utilisé pour identifier des paramètres variables
- Le type spécifié avant « … » est dominant.
- On peut n'en spécifier aucun.
- On peut en spécifier plusieurs.
- La variable identifiée après « … » sera implémentée comme un vecteur.

JCertif university

Constructors

- Un constructeur est appelé à partir du mot-clé *new.*
- Un constructeur n'a pas de type de retour.
- Si on ne code pas un constructeur de façon explicite, le compilateur en génère un pour nous (*default constructor*) qui fait simplement appeler le constructeur parent – si la classe est définie comme étant publique, alors celui-ci sera aussi définit comme *public.*
- On peut faire appel au constructeur parent à l'aide du mot-clé *super.*
- Le constructeur de la classe parent doit toujours être appelé en premier dans un constructeur (même avant de déclarer les variables), sinon on aura une erreur de compilation (par défaut le compilateur tente d'ajouter l'instruction *super()* dans tous les constructeurs).

JCertif university

Constructors

- Si on ne spécifie pas explicitement le mot-clé *this* ou *super*, le compilateur va tenter d'ajouter la ligne suivante :
  - *super()*
  - si la classe parent ne contient pas de constructeur parent sans arguments, on aura une erreur de compilation.
- Si au moins un constructeur est définit dans la classe, le compilateur ne va pas en définir un pour nous automatiquement.
- Un constructeur privé ne permet pas d'instancier la classe.

JCertif university

Static

- Il peut être appliqué aux variables, aux méthodes et à des bouts de code placés au niveau de la classe appelés *Static Initializers.*
- Il appartient à une classe et non aux instances de celle-ci : global à toutes les instances d'une même classe.
- On peut faire appel à une variable ou une méthode *static* via une instance d'une classe (objet / variable / référence) ou directement via le nom de celle-ci (ce dernier étant le moyen privilégié)
  - ex.: MaClasse.genererNombreUnique()
- Une variable *static* est unique pour toutes les instances d'une même classe.

JCertif university

Static

- Une méthode *static* ne peut pas accéder à une variable d'instance (variable non *static*) ni faire appel à une méthode non *static* directement (elle peut cependant instancier une classe et appeler ses méthodes).
- Généralement utilisé pour des méthodes dites « utilitaires » ou pour déclarer des constantes avec le mot-clé *final.*
- Une méthode *static* ne peut pas être surchargée par une méthode non *static.*

JCertif university

Static Initializers

- Il permet d'exécuter un bout de code définit en dehors d'une méthode (directement dans la classe).
- C'est comme si on définissait une méthode, mais sans lui donner de nom.
- Le code est exécuté une seule fois lors du chargement de la classe en mémoire.
- Les mêmes règles que précédemment au niveau des accès aux variables et méthodes *static* définies dans la classe.
- On utilise généralement cette technique pour charger des librairies dynamiques (.dll, .so, .sl, etc) en mémoire avec JNI (Java Native Interface) -> communication entre Java et C / C ++
- On peut avoir plusieurs blocs de code *static initializers* et ceux-ci seront exécutés dans leur ordre d'apparition.

JCertif university

Agenda

- ☑ Declaration and Access Control
- ☑ Object Orientation
- ☑ Assignments
- ❑ Operators
- ❑ Flow Control, Exceptions and Assertions
- ❑ Strings, I/O, Formatting, and Parsing
- ❑ Generics and Collections
- ❑ Inner Classes
- ❑ Threads
- ❑ The exam

JCertif university

Primitive Data Types

- Il faut connaître la taille de chaque type ainsi que la valeur minimale et maximale.
- Il faut savoir que *byte (8)*, *short (16)*, *int (32)* et *long (64)* sont signés excepté *boolean* et *char (16).*
- Il faut aussi savoir que le type *boolean* n'accepte que deux valeurs possibles soit : *true* et *false.*
- Il faut connaître les constantes des classes *Float* et *Double*:
  - *Float.NaN, Float.NEGATIVE_INFINITY, Float.POSITIVE_INFINITY*
  - *Double.NaN, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY*

JCertif university

Literals

- La valeur d'une variable de type char se définit comme suit: 'v'
- Il faut savoir comment on peut spécifier un caractère Unicode dans un type *char.*
  - ex.: *'\u4567'*
- Il n'est pas requis de connaître par cœur les « escape sequences » (\).
- Un entier octal se définit comme suit : 0123
  - toujours préfixé par un 0
- Un entier hexadécimal se définit comme suit : 0x1234 ou 0X1234

JCertif university

Literals (suite)

- Pour forcer un *long,* il faut utiliser le suffixe *L.*
  - ex.: 4L ou 4l
- Le suffixe F ou f désigne un *float.*
- Le suffixe D ou d désigne un *double.*
- Pas de suffixe pour *short, int* et *byte*
- Par défaut, un entier est de type *int*.
- Par défaut, les chiffres à virgules flottantes sont de type *double.*
- Attention aux assignations
  - ex.: short s = 10 + x; // erreur de compilation
  - le résultat sera de type *int* et non de type *short*
- Une chaîne de caractères en Java représente un objet de type *String ("")*

Literals (suite)

(nouveauté dans la version 7.0)

- Binary literals avec des underscores pour plus de lisibilité

```
int mask = 0b101010101010;

int mask = 0b1010_1010_1010;

long big = 9_223_783_036_967_937L;

int one_million = 1_000_000;//plutôt que int one_million = 1000000;
```

JCertif university

Primitive : Assignment

- Se produit lorsqu'on assigne une valeur de type différent du type attendu (type primitif).
- Un *boolean* ne peut pas être convertit dans aucun autre type (aucun *cast* possible sur le *boolean*).
- Une conversion automatique se produit d'un type plus petit vers un type plus grand (*widening conversion*).
- Dans le cas contraire, un *cast* est requis pour forcer la conversion vers le type le plus petit (*narrowing conversion*).

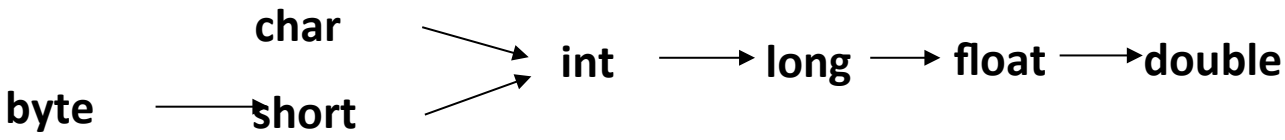
JCertif university

Conversion : Assignment

- Les littéraux à virgule flottante sont de types *double* et les entiers de type *int*
  - faire attention aux assignations lors des déclarations de variables
- Il existe cependant une exception à la règle : une assignation d'un littéral de type *int* vers un type *byte*, *short* ou *char* lors d'une déclaration ne requiers pas de *cast* lorsque la valeur se situe dans la plage de valeurs supportées par le type en question.

Conversion : Assignment

widening conversion ->



<- narrowing conversion (requiers un cast)

JCertif university

Conversion : Method call

- Il se produit lors d'un passage de paramètres de types différents à une méthode.
- Les mêmes règles que l'assignation s'appliquent dans ce cas (*widening* est permis et *narrowing* est interdit sans *cast*).

Conversion : Arithmetic Promotion

- Il se produit lors d'une opération arithmétique (calcul mathématique).
- Le compilateur fait toujours une conversion des opérandes vers le type le plus grand (*widening conversion*).
- Règle sur les opérateurs unaires (+, -, ++, --, etc.) :
  - si l'opérande est de type *byte*, *short* ou *char*, il est automatiquement convertit en *int* excepté pour les opérateurs ++ et --
- Règles sur les opérateurs binaires (*, /, +, -, etc) :
  - si un des opérandes est de type *double,* alors tous les autres seront convertis en *double*
  - si un des opérandes est de type *float*, alors tous les autres seront convertis en *float*
  - si un des opérandes est de type *long*, alors tous les autres seront convertis en *long*
  - dans tous les autres cas, il y aura une conversion vers le type *int*

JCertif university

Primitives and Casting

- Forcer Java à faire une conversion.
- On peut faire un *cast* vers un type plus grand (*widening* – permis, mais non obligatoire) ou un type plus petit (*narrowing* – obligatoire, sinon erreur de compilation)
- Un *cast* sur un primitif se fait comme suit :
  - int x = (int)1.234;
- Aucun *cast* permis avec le type *boolean.*

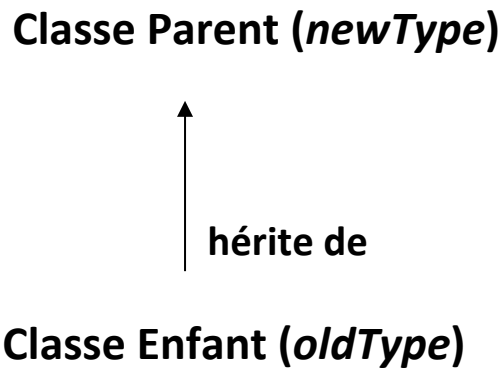
JCertif university

Reference : Assignment Conversion

- Il se produit lorsqu'on assigne une référence à un objet vers un autre objet de type différent.
- Trois types de références possible :
  - *class*
  - *interface*
  - *array*
- *newType = oldType* où :
  - *oldType* peut être *class*, *interface* ou *array*
  - *newType* : idem

Reference : Assignment Conversion

- La règle fondamentale est la suivante : la conversion automatique est permise si la référence est située au haut de la hiérarchie (héritage) c-à-d que *oldType* hérite de *newType.*



JCertif university

Reference : Assignment Conversion

	class	*interface*	*array*
class	***oldType* doit être une sous classe de *newType***	***newType* doit être de type *Object***	***newType* doit être *Object***
interface	***oldType* doit implémenter *newType***	***oldType* doit être une interface enfant de *newType***	***newType* doit être *Cloneable* ou *Serializable***
array	**Erreur de compilation**	**Erreur de compilation**	***oldType* doit être un *array* contenant des objets de type compatible**

JCertif university

Method-Call Conversion

- Il se produit lors du passage de paramètres à une méthode dont les types sont différents.
- Les règles de conversion sont les mêmes que lors d'une assignation.
- Convertir vers une superclasse est permis et l'inverse est interdit.

JCertif university

Arrays

- Il contiennent toujours un contenu homogène
  - de même type – attention à l'héritage et aux interfaces
- Les crochets peuvent être placés avant ou après le nom de la variable ou d'une méthode lors de la déclaration d'un *array*.
- La taille d'un *array* est toujours spécifié à l'aide du mot-clé *new.*
- Les *arrays* sont des objets contenant des méthodes.
- On peut initialiser un *array* lors de sa déclaration à l'aide des accolades ({ }).
- Un index en Java commence toujours à 0.
- *.length* est un attribut publique des *arrays* et non une méthode.

JCertif university

Wrapper Classes

- Chaque type primitif possède son *wrapper*.
- Une classe *wrapper* encapsule un type primitif immuable (lecture seule)
  - ex.: la classe *Integer* encapsule le type *int*
- On peut créer un *wrapper* à l'aide de son constructeur qui accepte le type natif qu'il représente ou une chaîne de caractères qui contient une donnée qui est convertie dans le type en question (excepté la classe *Character*).
- Dans le cas des constructeurs qui acceptent une chaîne de caractères comme paramètre d'entrée, si la valeur ne peut être convertie dans le type primitif encapsulé, l'exception *NumberFormatException* sera lancée
  - excepté pour la classe *Boolean* qui ne lance pas cette exception
- La méthode *equals()* de ces classes est chevauchée et permet de comparer le contenu de deux objets de type *wrapper* correctement

JCertif university

Wrapper Classes

Primitif	**Wrapper**
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

JCertif university

Wrapper Classes

- On peut obtenir la valeur encapsulée via l'une des méthodes suivantes :
  - *booleanValue()* pour la classe *Boolean*
  - *charValue()* pour la classe *Character*
  - *byteValue()*
  - *shortValue()*
  - *intValue()*
  - *longValue()*
  - *floatValue()*
  - *doubleValue()* pour les autres classes (qui héritent de la classe abstraite *Number*)

Wrapper Classes

- Les classes *wrapper* fournissent des méthodes statiques pratiques concernant le type primitif encapsulé.
- Toutes les classes *wrapper,* excepté la classe *Character,* contiennent la méthode statique *valueOf (String s)* qui crée un *wrapper* à partir d'une valeur spécifiée dans une chaîne de caractères.
- La méthode statique *parseXXX* des classes *Byte*, *Short*, *Integer*, *Long*, *Float* et *Double* convertit une chaîne de caractères dans le type primitif spécifié (ex.: *parseByte (), parseShort()*) – l'exception *NumberFormatException* est lancée si la conversion n'est pas possible.

JCertif university

Wrapper Classes

- Les méthodes *Boolean.getBoolean()*, *Integer.getInteger()* et *Long.getLong()* retournent la valeur d'une propriété système dans le type spécifié.
- La méthode *toString()* est chevauchée pour toutes les classes *wrapper.*
- Les classes *wrapper* sont déclarées *final* et ne peuvent donc pas être héritées.

JCertif university

Autoboxing – auto-unboxing

- Assignation automatique d'une valeur primitive à son *wrapper.*
- La valeur du type primitif doit correspondre à la valeur du *wrapper* spécifié.
- Fonctionne également sur le retour d'une méthode.
- Aucune optimisation, d'un cas comme dans l'autre, il y aura conversion d'un vers l'autre (code auto-généré par le compilateur) – utiliser avec modération.

Math Class

- Contient une collection de méthodes et deux constantes qui sont utilisées à des fins d'opérations mathématiques.
- La classe est déclarée *final*
  - on ne peut pas l'instancier – son constructeur est déclaré *private*
- Toutes les méthodes de la classe sont déclarées *static.*
- Les deux constantes sont les suivantes :
  - *Math.PI*
  - *Math.E*
- Plusieurs méthodes sont implantées en langage C (code natif).

JCertif university

Math Class – Methods

Methods	Returns
int abs(int i)	Absolute value of i
long abs(long l)	Absolute value of l
float abs(float f)	Absolute value of f
double abs(double d)	Absolute value of d
double ceil(double d)	**The smallest integer that is not less than d (returns as a double)**
double floor(double d)	**The largest integer that is not greater than d (returns as a double)**
int max(int i1, int i2)	Greater of i1 and i2
long max(long l1, long l2)	Greater of l1 and l2
float max(float f1, float f2)	Greater of f1 and f2
double max(double d1, double d2)	Greater of d1 and d2

JCertif university

Math Class – Methods

Method	Returns
int min(int i1, int i2)	Smaller of i1 and i2
long min(long l1, long l2)	Smaller of l1 and l2
float min(float f1, float f2)	Smaller of f1 and f2
double min(double d1, double d2)	Smaller of d1 and d2
double random()	Random number >= 0.0 and < 1.0
int round(float f)	Closest int to f
long round(double d)	Closest long to d
double sin(double d)	Sine of d
double cos(double d)	Cosine of d
double tan(double d)	Tangent of d

JCertif university

Agenda

- ☑ Declaration and Access Control
- ☑ Object Orientation
- ☑ Assignments
- ☑ Operators
- ❑ Flow Control, Exceptions and Assertions
- ❑ Strings, I/O, Formatting, and Parsing
- ❑ Generics and Collections
- ❑ Inner Classes
- ❑ Threads
- ❑ The exam

JCertif university

Unary Operators

- Increment and decrement : ++ -- (pre / post)
- Unary plus and minus : + - (positif/négatif)
- Bitwise inversion : ~ (0000 devient 1111)
- Boolean complement : ! (!true devient false)
- Cast : ( )

JCertif university

Arithmetic Operators

- Addition : +
- Soustraction : -
- Multiplication : *
- Division : /
- Modulo : %
- Multiplication et division s'appliquent aux types entiers incluant le type *char.*
- Une division ou un modulo par zéro génèrent l'exception *ArithmeticException.*
- Une multiplication ou une division d'entiers donnent un résultat de type *int* ou *long* (les fractions seront perdues).
- Modulo donne la partie entière du reste de la division.

JCertif university

Addition and String Objects

- Addition (+) est utilisé comme opérateur d'addition ET de concaténation.
- Pour convertir un type primitif (*int*, *boolean*, etc.), le compilateur utilise la classe *Wrapper* associée à ce type (*Integer*, *Boolean*, etc.).
- Pour convertir un objet en String, le compilateur fait appel à la méthode *toString()* de l'objet qui est une méthode qui appartient à la classe *Object*.

JCertif university

Règles pour l'Addition

- Addition entre deux valeurs numériques
  - résultat numérique
  - résultat au minimum de type *int*
  - le type le plus grand l'emporte
- Addition entre deux objets
  - au moins un de ces deux objets doit être de type *String*
  - les autres objets sont convertis en *String* à l'aide de la méthode *toString()* excepté les types primitifs où les classes *Wrapper* sont utilisées

JCertif university

Arithmetic Error Conditions

- En cas d'erreur de division par zéro, la classe *ArithmeticException* est lancée.
- Aucune autre exception n'est lancée en cas d'erreur de calcul.
- Pour les valeurs infinies sur les nombres à virgule flottante, la JVM utilise les constantes des classes *Float* et *Double*
  - Ex: *Float.NaN* // Not A Number
- Deux valeurs *NaN* sont différentes
  - *Float.NaN != Float.NaN*
- Pour tester une valeur *NaN*, il faut utiliser la méthode *isNaN()* des classes *Float* et *Double.*

JCertif university

Comparison Operators

- Less than : <
- Less than or equal : <=
- Greater than : >
- Greater than or equal : >=
- La comparaison entre deux nombres de types différents va provoquer une promotion vers le type le plus grand des deux.
- On peut utiliser ces opérateurs sur le type *char.*

JCertif university

Comparison Operators (suite)

- L'opérateur *instanceof* teste le type d'un objet (classe à laquelle celui-ci appartient)
  - gauche : variable ou élément d'un *array*
  - droite : classe, interface ou *array* (type)
  - on ne peut pas utiliser *java.lang.Class* ni le nom d'une classe écrite en chaîne de caractères
    - ex.: «nom classe»
- si la valeur comparée avec l'opérateur est nulle (null), aucune exception n'est lancée
- on peut détecter un vecteur de deux façons :
  - x instanceof Object[]
  - x.getClass().isArray()

JCertif university

== vs equals()

- == compare l'égalité de deux types primitifs.
- == compare l'égalité de l'adresse de deux pointeurs (pour les «Object reference»).
- La méthode *equals()* héritée de la classe *Object* permet de comparer le contenu de deux instances.
- La méthode *equals*, si non surchargée (overloading), retourne par défaut le résultat de l'opérateur == implémenté dans la classe *Object.*

JCertif university

Bitwise Operator : AND (& - &&)

Op1	Op2	Op1 AND Op2
0 / false	0 / false	0 / false
0 / false	1 / true	0 / false
1 / true	0 / false	0 / false
1 / true	1 / true	1 / true

JCertif university

Bitwise Operator OR (| - ||)

Op1	Op2	Op1 OR Op2
0 / false	0 / false	0 / false
0 / false	1 / true	1 / true
1 / true	0 / false	1 / true
1 / true	1 / true	1 / true

JCertif university

Bitwise Operator XOR (^)

Op1	Op2	Op1 XOR Op2
0 / false	0 / false	0 / false
0 / false	1 / true	1 / true
1 / true	0 / false	1 / true
1 / true	1 / true	0 / false

JCertif university

Short-Circuit Logical Operators

- L'opérateur && : si l'expression de gauche est fausse, celle de droite n'est pas évaluée.
- L'opérateur || : si l'expression de gauche est vraie, celle de droite n'est pas évaluée.

Conditional Operator (?)

- Il permet de restreindre une condition if / else en une seule ligne.
- L'expression évaluée doit retourner *true* ou *false.*
- Les expressions de droite (condition vraie et fausse) doivent retourner une valeur dont le type est compatible avec le type de la variable résultante.
- La section de gauche représente la condition.
- La section au centre représente le résultat vrai.
- La section de droite représente le résultat faux.
  - int nbResultats = (resultats != null) ? resultats.getNbRows() : 0;

Assignment Operators

- La combinaison opérateur = (ex.: +=, -=) permet de restreindre une opération et génère un *cast* implicite lorsque requis
  - *byte x = 2; x += 3*;
- Java supporte le « down-casting » dans le cas précédent ainsi que dans les déclarations et les initialisations.
  - byte x = 5;

JCertif university

Agenda

- ☑ Declaration and Access Control
- ☑ Object Orientation
- ☑ Assignments
- ☑ Operators
- ☑ Flow Control, Exceptions and Assertions
- ❑ Strings, I/O, Formatting, and Parsing
- ❑ Generics and Collections
- ❑ Inner Classes
- ❑ Threads
- ❑ The exam

JCertif university

If / Else

- La condition doit toujours retourner une valeur booléenne.
- Les accolades sont optionnelles, mais recommandée.
- Un *if / else* s'écrit comme suit :

```
if (condition)
{
}
else if(condition)
{
}
else
{
}
```

JCertif university

Switch

- Permet de faire une sélection par choix.
- La condition doit être de type *byte*, *short*, *char, enum* ou *int*
  - ne peut pas être ni *long*, ni *String*, ni *float*, ni *boolean*, ni *double*, ni *Object*, etc.
- L'argument d'un *case* doit être une constante ou une expression de constantes qui peut être résolue à la compilation (et non à l'exécution – les variables ne sont pas permises).
- Un *case* accepte qu'un seul argument, si on veut regrouper des instructions *case* ensembles, il faut les mettre un en dessous des autres sans mettre de *break* entre chacun d'eux.
- Un *break* termine l'exécution d'un *case.*
- *default* sera exécuté si aucun case ne correspond à l'expression OU si aucun *break* n'a été placé entre les instructions *case.*
- *default* peut être placé n'importe où, mais il est recommandé de le mettre à la fin d'un *switch.*

String Switch Statement

(nouveauté dans la version 7.0)

- Les Strings sont aussi des constantes (immuable)

```
int monthNameToDays(String s, int year) {
  switch(s) {
    case "April": case "June":
    case "September": case "November":
      return 30;

    case "January": case "March":
    case "May": case "July":
    case "August": case "December":
      return 31;

    case "February”:
      ...
    default:
      ...
```

JCertif university

While Loop

- La condition dans le *while* doit toujours retourner un *boolean.*
- La boucle va se répéter tant que la condition est vraie.
- Une boucle *while* peut ne pas avoir d'accolade lorsque celle-ci possède une seule instruction cependant, cette approche n'est pas recommandée.
- La condition est toujours vérifiée au début.

JCertif university

Do Loop

- La boucle s'exécute tant que la condition est vraie.
- La boucle va toujours s'exécuter au moins une fois.
- Ce type de boucle est celui qui est le moins utilisé en Java.

JCertif university

For Loop

- La condition est composée de trois parties optionnelles :
  - *statement*
    - exécuté une seule fois **avant** la première itération de la boucle
    - utilisé pour initialiser les expressions
    - peut contenir une déclaration de variable
  - *condition* (test)
    - doit être une expression booléenne
    - exécute la boucle tant que la valeur est vraie
    - évalué au moins une fois avant les itérations – ceci pourrait faire en sorte qu'aucune itération ne soit exécutée
  - *expression*
    - exécutée immédiatement **après** la première itération
    - généralement utilisée pour incrémenter le compteur de la boucle

JCertif university

For Loop

- Les accolades sont optionnelles, mais il est fortement recommandé de les spécifier.
- *statement*, *condition* et *expression* sont optionnels
  - si les trois sont absents, alors on aura une boucle infinie
- *statement* et *expression* peuvent contenir une combinaison d'éléments séparés d'une virgule
  - ces éléments doivent être de même type et doivent être déclarés ensembles dans la boucle ou séparément (en dehors de la boucle)

JCertif university

For Loop – Version améliorée

- Plus besoin de spécifier de compteurs :
  - *for (type nom_variable:array)*
    - où : *type* doit être compatible avec le type du *array*
- Ni d'itérateur :
  - *for (type nom_variable:collection)*
    - où : *type* doit être compatible avec le type de la *collection*
- La boucle s'exécute une fois pour chaque élément du *array* ou de la *collection.*
- Le *array* peut contenir des références d'objets ou des types primitifs.
- Les *collection* ne peuvent contenir que des références d'objets.

JCertif university

Break and Continue

- *continue*
  - force la réévaluation de l'expression dans une boucle
  - on peut utiliser ce mot-clé avec une étiquette pour indiquer l'endroit où on veut que la réévaluation soit effectuée (une étiquette est un mot qui se termine par « : » devant une instruction)
- *break*
  - force l'arrêt d'une boucle
  - on peut utiliser ce mot-clé avec une étiquette pour indiquer l'endroit où on veut que le saut soit effectué lors d'un arrêt forcé

JCertif university

Try / Catch

- Le code à « surveiller » doit être placé dans l'instruction *try* entre accolades.
- La ou les exceptions à attraper doivent être identifiées dans des sections *catch* entre accolades.
- Le code qui doit être exécuté, qu'il y ait exception ou pas, doit être placé dans l'instruction *finally* entre accolades.

JCertif university

Finally

- Les cas qui peuvent empêcher l'exécution du *finally* :
  - une exception est lancée dans le bloc *finally* lui-même
  - un arrêt imprévu de l'exécution du *thread*
  - l'appel à *System.exit()*
  - fermer l'ordinateur (!)
- On attrape une exception dans un *finally* de la même façon que dans un *try / catch* standard.

JCertif university

Catch Multiples

- On peut spécifier plus d'une instruction *catch* après un bloc *try.*
- Un *catch* attrape toutes les exceptions qui sont du type spécifié ainsi que les sous-classes de celle-ci.
- Un *catch* sur une action spécifique (classe enfant au niveau de l'héritage) doit toujours précéder un *catch* plus générique (classe parent).
- Seul le premier bloc *catch* qui correspond à l'exception est exécuté.

Multi catch

(nouveauté dans la version 7.0)

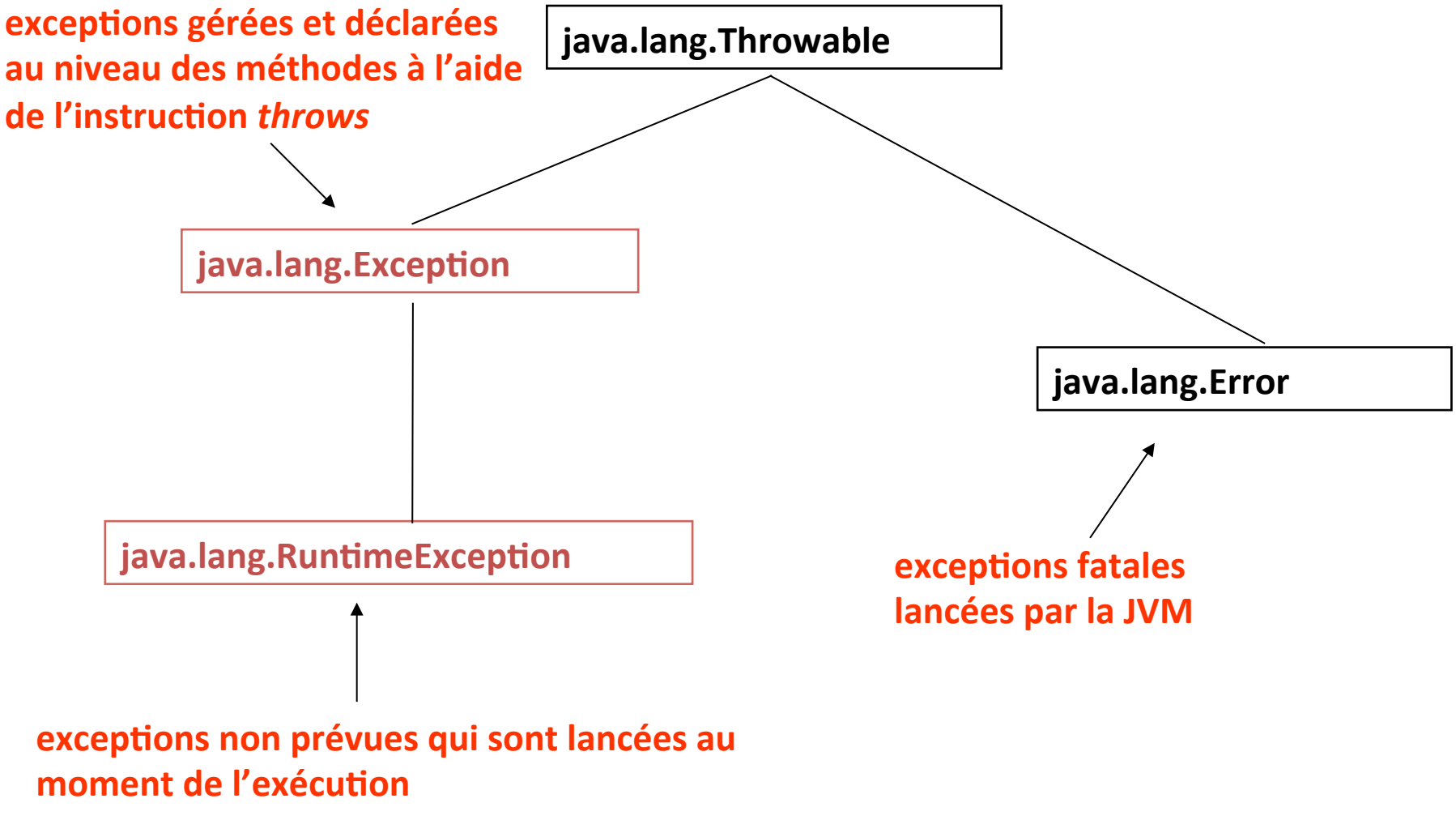
```
try {
   ...
} catch (ClassCastException e) {
   doSomethingClever(e);
   throw e;
} catch(InstantiationException |
    NoSuchMethodException |
    InvocationTargetException e) {
   log(e);
   throw e;
}
```

JCertif university

Throw / Throws

- *throw*
  - permet de lancer / relancer une exception
  - la *stack trace* débute là où l'appel du *throw* est effectué
- *throws*
  - indique au niveau de la déclaration de la méthode qu'une ou plusieurs exceptions peuvent être lancées lors de l'exécution de celle-ci
  - une méthode qui utilise ce mot-clé indique qu'elle pourrait lancer une ou plusieurs exceptions et celles-ci doivent être gérées par l'appelant

Hiérarchie des exceptions



JCertif university

Exceptions and Overriding

- Lorsque l'on *override* une méthode d'une classe parent et que celle-ci lance une exception, la méthode doit lancer :
  - la même exception
  - une ou plusieurs exception(s) enfant
  - aucune exception

JCertif university

Agenda

- ☑ Declaration and Access Control
- ☑ Object Orientation
- ☑ Assignments
- ☑ Operators
- ☑ Flow Control, Exceptions and Assertions
- ☑ Strings, I/O, Formatting, and Parsing
- ❑ Generics and Collections
- ❑ Inner Classes
- ❑ Threads
- ❑ The exam

JCertif university

Strings

- Java facilite la manipulation de chaînes de caractères à l'aide des trois classes suivantes :
  - *String*
  - *StringBuffer*
  - *StringBuilder*
- Ces classes utilisent des valeurs 16 bits Unicode.

JCertif university

String Class

- Contient une chaîne de caractères immuable (lecture seule – valeur qui ne peut pas être modifiée)
- On peut instancier un objet de type *String* de deux façons :
  - *String s1 = new String("immuable"); -> à éviter*
  - *String s2 = "immuable";*
- Lorsque l'on spécifie une chaîne de caractères dans le code, celle-ci est placée dans un « pool » de *String* de façon à ne pas répéter celle-ci plusieurs fois en mémoire si elle est déclarée à plusieurs endroits dans le code.
- La méthode *intern()* permet d'ajouter une chaîne de caractères dynamique dans le « pool » de *String.*

String Class (suite)

- Pour comparer deux chaînes de caractères, il faut utiliser la méthode *equals()* et non l'opérateur ==
- Cette méthode compare caractère par caractère
- Prendre garde: la comparaison suivante va retourner faux :

  String s1 = new String("1234")
  String s2 = "1234";
  if (s1 == s2) -> faux

- La méthode *split()* permet de découper une chaîne de caractères en morceaux en spécifiant une expression.

JCertif university

String Class (suite)

Méthode	Description
char charAt(int index)	Returns the indexed character of a string where the index of the initial character is 0
String concat(String addThis)	Returns a new string consisting of the old string followed by addThis
int compareTo(String otherString)	Performs a lexical comparison; returns an int that is less than 0 if the current string is less than otherString, equal to 0 if the strings are identical and greater than 0 if the current string is greater than otherString
boolean endsWith(String suffix)	Returns true if the current string ends with suffix; otherwise returns false
boolean equals(Object ob)	Returns true if ob instanceof String and the string encapsulated by ob matches the string encapsulated by the executing object
boolean equalsIgnoreCase(String s)	Creates a new string with the same value as the executing object, but in lower case

JCertif university

String Class (suite)

Méthode	Description
int indexOf(int ch)	Returns the index within the current string of the first occurrence of ch. Alternative forms return the index of a string and begin searching from a specified offset.
int lastIndexOf(int ch)	Returns the index within the current string of the last occurence of ch. Alternative forms return the index of a string and end searching at a specified offset from the end of the string.
int length()	Returns the number of characters in the current string.
String replace(char oldChar, char newChar)	Returns a new string, generated by replacing every occurence of oldChar with newChar.
boolean startsWith(String prefix)	Returns true if the current string begins with prefix; otherwise returns false. Alternate forms begin searching from a specified offset.

JCertif university

String Class (suite)

Méthode	Description
String substring(int startIndex)	Returns the substring beginning at startIndex of the current string and extending to the end of the current string. An alternate form specifies starting and ending offsets.
String toLowerCase()	Creates a new string with the same value as the executing object, but in lower case.
String toString()	Returns the executing object
String toUpperCase()	Converts the executing object to uppercase and returns a new string.
String trim()	Returns the string that results from removing whitespace characters from the beginning and ending of the current string.

JCertif university

StringBuffer Class

- Représente une chaîne de caractères qui peut être modifiée.
- Peut être instanciée comme suit :
  - *StringBuffer()*
  - *StringBuffer(int capacity)*
  - *StringBuffer(String initialString)*
- La classe *StringBuffer* possède une « capacité » qui s'extensionne automatiquement selon la taille de la chaîne de caractères qu'elle contient.

JCertif university

StringBuffer Class

- La classe *StringBuffer* ne chevauche pas la méthode *equals().*
- *StringBuffer* est « thread safe » – si plusieurs threads effectuent des appels concurrents sur cette classe, son contenu ne sera pas corrompu (la classe est synchronisée).
- La méthode *equals()* de cette classe est celle de la classe *Object* (elle compare des références).

JCertif university

StringBuffer Class

Méthode	Description
StringBuffer append(String str)	Appends str to the current string buffer. Alternative forms support appending primitives and character arrays; these are converted to strings before appending.
StringBuffer append(Object obj)	Calls toString() on obj and appends the result to the current string buffer.
StringBuffer insert(int offset, String str)	Inserts str into the current string buffer at position offset. There are numerous alternative forms.
StringBuffer reverse()	Reverses the characters of the current string buffer.
StringBuffer replace(int arg0, int arg1, String arg2)	Replaces the characters specified in the length of arg0 and arg1 by the string arg2.
StringBuffer setLength(int newLength)	Sets the length of the string buffer to newLength. If newLength is less than the current length, the string is truncated. If newLength is greater than the current length, the string is padded with null characters.

JCertif university

StringBuilder Class

- Presqu'identique à *StringBuffer*
- Elle n'est pas « thread safe » - il faut synchroniser les appels à celle-ci nous même.
- Elle s'exécute plus rapidement dans un contexte d'exécution à un seul thread.
- *StringBuilder* et *StringBuffer* implémentent l'interface *java.lang.Appendable.*

String Concatenation

- L'opérateur + qui est utilisé pour effectuer des additions sert aussi d'opérateur de concaténation.
- Il peut être utilisé pour concaténer plusieurs objets ensembles pour former une chaîne de caractères (chaînage)
  - a + b + c = new
  - StringBuffer().append(a).append(b).append(c).toString()
- La recette suivante est utilisée:
  - un objet de type *StringBuffer* vide est instancié
  - la méthode *append()* de cet objet est appelée pour chaque objet qu'il faut concaténer
  - la méthode *toString()* est appelée pour générer la chaîne de caractères finale
- Au moins un operand doit être de type *String.*

Console Class

- La classe *java.io.Console* est celle qui contrôle l'engin physique de saisie au clavier et d'affichage à l'écran.
- Il faut s'assurer d'invoquer la méthode *System.console ()* au préalable.
- Elle gère les entrées imprimées ou non-imprimées à l'écran
- Elle peut formater un message en sortie sur une ligne de commande.
- Ses méthodes les plus pertinentes pour l'examen sont :
  - readLine : retourne la chaîne de caractères saisit au clavier
  - readPassword : retourne un vecteur de caractères sans l'afficher à l'écran pour empêcher de connaître la longueur d'un mot de passe

JCertif university

Object Streams and Serialization

- Le procédé pour écrire un objet sur le disque est appelé sérialisation.
- Pour sérialiser un objet, il faut créer une instance de la classe *ObjectOutputStream* puis faire appel à la méthode *writeObject()* de la classe.
- Pour lire un objet, il faut créer une instance de la classe *ObjectInputStream* puis faire appel à la méthode *readObject()* de cette classe.
- Il s'agit d'inscrire / lire toutes les données (et non sa définition de classe proprement dite) d'une instance d'un objet sur le disque.

Formatting Dates

- La classe *DateFormat* est utilisée pour formater une date.
- La méthode *getDateInstance()* retourne le formateur par défaut.
- La méthode *format()* consiste à formater la date reçue en entrée.
- La classe *DateFormat* supporte quatre niveaux de détails :
  - SHORT : AA-MM-JJ
  - MEDIUM : AAAA-MM-JJ
  - LONG : AAAA-MMMM-JJ
  - FULL : AAAA-MMMM-JJJJ

JCertif university

Formatting Numbers and Currency

- La classe *java.text.NumberFormat* est utilisée pour formater des nombres.
- Pour utiliser cette classe, il faut faire appel aux méthodes suivantes :
  - getInstance()
  - getInstance(Locale loc)
  - getCurrencyInstance()
  - getCurrencyInstance(Locale loc)
- On fait ensuite appel à la méthode *format()* pour formater le nombre désiré.

JCertif university

Agenda

- ☑ Declaration and Access Control
- ☑ Object Orientation
- ☑ Assignments
- ☑ Operators
- ☑ Flow Control, Exceptions and Assertions
- ☑ Strings, I/O, Formatting, and Parsing
- ☑ Generics and Collections
- ❑ Inner Classes
- ❑ Threads
- ❑ The exam

JCertif university

Object Class

- Toutes les classes en Java héritent de la classe *java.lang.Object* même si on ne spécifie pas explicitement « *extends Object* »
  - cette instruction est ajoutée automatiquement à toutes les classes qui ne font pas partie d'un héritage
- Toutes les méthodes de cette classe sont accessibles de toutes les classes Java.
- *notify()*, *notifyAll()* et *wait()* sont utilisées dans un contexte de développement multi-tâches (thread).
- *toString()* et *equals()* sont largement utilisées et doivent (idéalement) être chevauchées afin de retourner des valeurs cohérentes en fonction du contenu de la classe dans laquelle elle sont redéfinies.

Object Class :Méthode equals()

- La signature de la méthode *equals()* est la suivante :
  - *public boolean equals(Object object)*
- Permet d'effectuer une comparaison véritable
  - comparer le contenu de deux objets et non leur adresse de pointeur tel que le fait l'opérateur ==
- La version implémentée par défaut (celle qui est dans la classe *Object*), lorsqu'elle n'est pas chevauchée (override) compare deux références de pointeurs à l'aide de l'opérateur ==.

JCertif university

Arrays vs Collections API

- Les *arrays* permettent de regrouper un ensemble d'objets.
- Les *arrays* sont limités en terme de fonctionnalités sur les objets qu'ils regroupent.
- Les *arrays* sont limités en terme de taille (celle-ci doit être définie d'avance).
- Les *arrays* peuvent conserver des éléments primitifs ainsi que des références d'objets.
- Les *collections* facilitent la manipulation de références d'objets (aucun primitif).
- Les *collections* ne peuvent conserver que des références d'objets.

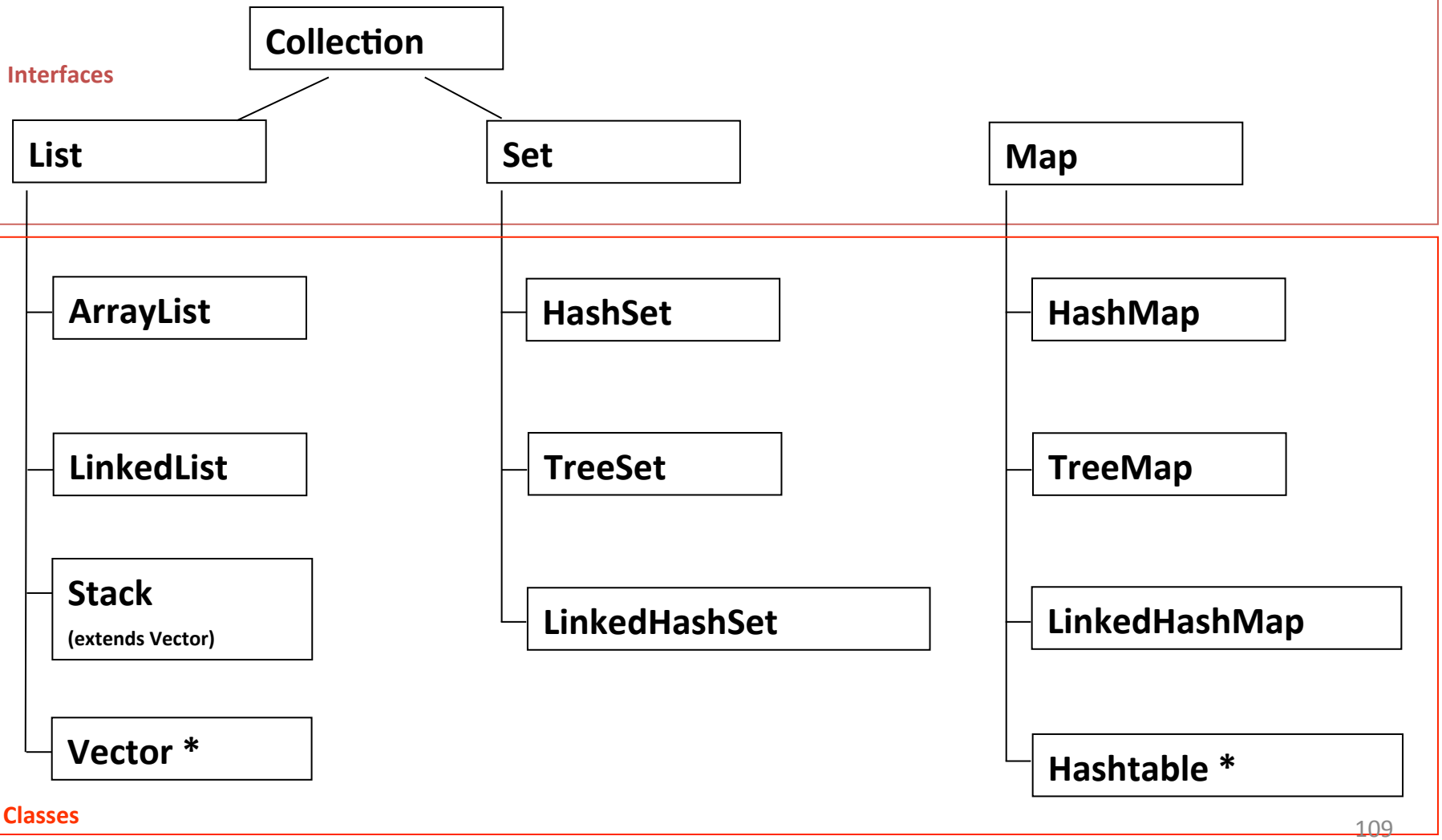
JCertif university

Collections API

- Les principales interfaces sont les suivantes :
  - *java.util.Collection*
  - *java.util.List* (*extends Collection*)
  - *java.util.Set* (*extends Collection*)
  - *java.util.Map*
- La majorité des classes qui implémentent ces interfaces ne sont pas « thread safe ».

JCertif university

Collection Types



JCertif university

Interface Collection

Nom	**Description**
add(Object x)	Adds x to this collection
addAll(Collection c)	Adds every element of c to this collection
clear()	Removes every element of this collection
contains(Object x)	Returns true if this collection contains x
containsAll (Collection C)	Returns true if this collection contains every element of c
isEmpty()	Returns true if this collection contains no elements
iterator()	Returns an Iterator over this collection

JCertif university

Interface Collection

Nom	Description
remove(Object x)	Removes x from this collection
removeAll(Collection c)	Removes every element in c from this collection
retainAll(Collection c)	Removes from this collection every element that is not in c
size()	Returns the number of elements in this collection
toArray()	Returns an array containing the elements in this collection

JCertif university

Interface Iterator

- Utilisée (en dehors de la boucle *for each*) pour naviguer à travers tous les éléments d'une collection.
- Trois méthodes :
  - *boolean hasNext()* : retourne vrai s'il reste des éléments à parcourir
  - *Object next()* : Retourne le prochain élément de la liste
  - *void remove()* : Supprime l'élément retourné par la méthode *next()*

JCertif university

Interface List

- Une liste conserve l'ordre dans lequel les éléments ont été ajoutés.
- Peut contenir des duplicatas
- Une liste contient les quatre méthodes suivantes:
  - *void add(int index, Object x)* : insert x à l'indexe spécifié
  - *Object get(int index)* : retourne l'élément à l'indexe spécifié
  - *int indexOf(Object x)* : retourne l'indexe de l'élément spécifié ou -1 si non trouvé
  - *Object remove(int index)* : supprime l'élément à l'indexe spécifié

JCertif university

Interface Set

- Aucun ordre retenu
- Ne contient aucun duplicata
- Ne contient pas de nouvelle méthode

JCertif university

Interface SortedSet

- Hérite de *java.util.Set*
- Manipule des éléments triés
- Contient les méthodes suivantes :
  - *Object first()* : retourne le premier élément de la liste
  - *Object last()* : retourne le dernier élément de la liste
  - *SortedSet headSet(Object thru)* : retourne une liste triée du premier élément jusqu'à l'élément *thru*
  - *SortedSet tailSet(Object from)* : retourne une liste triée de l'élément *from* jusqu'au dernier élément
  - *SortedSet subSet(Object from, Object to)* : retourne la liste des éléments entre *from* et *to*
- La classe *TreeSet* utilise l'interface *java.lang.Comparable* comme critère de tri.

JCertif university

Interface Map

- N'implément pas *java.util.Collection*
- Implémente deux collections : une pour la gestion des clés et l'autre pour les valeurs.
- Ne contient aucun duplicata
- Les méthodes qu'elle contient :
  - *Object put(Object key, Object value)* : ajoute une valeur ainsi que sa clé
  - *Object get(Object key)* : retourne l'objet associé à la clé spécifiée
  - *boolean containsKey(Object key)* : retourne vrai si l'élément spécifié par la clé existe
  - *void clear()* vide le contenu de la collection
  - *Set keySet()* : retourne la liste des clés (interface *Set*)
  - *Collection values()* : retourne la liste des valeurs

JCertif university

Interface SortedMap

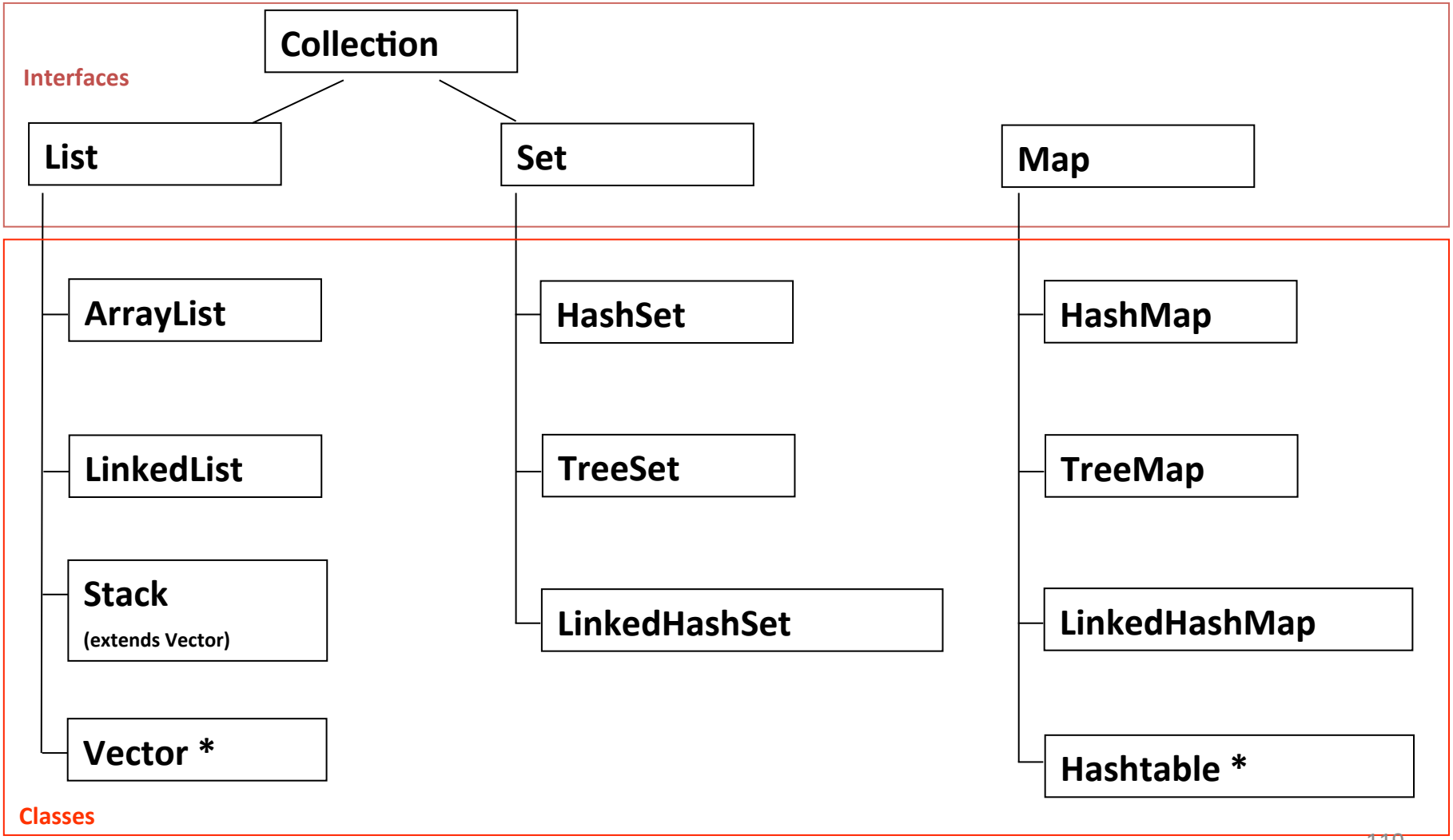
- Hérite de l'interface *java.util.Map*
- Manipule des éléments triés
- Les méthodes qu'elle contient :
  - *Object firstKey()* : retourne la première clé
  - *Object lastKey()* : retourne la dernière clé
  - *SortedMap headMap(Object toKey)* : retourne une liste triée du début de la liste jusqu'à la clé spécifiée
  - *SortedMap tailMap(Object fromKey)* : retourne une liste triée à partir de la clé spécifiée jusqu'à la fin de la liste
  - *SortedMap subMap(Object fromKey, Object toKey)* : retourne une liste triée à partir de l'élément *fromKey* jusqu'à l'élément *toKey*

JCertif university

Collection Types

- Il existe quatre méthodes de conservation des données à l'intérieur des collections :
  - array storage
    - rapide à faible volume
  - linked list
    - supporte l'ajout et la destruction d'éléments
    - efficace à grand volume
    - recherche parfois lente
  - tree
    - supporte l'ajout et la destruction
    - tri automatique
    - recherche rapide d'un élément
  - hash table
    - recherche rapide d'un élément
    - efficace à haut volume uniquement

Collection Types



JCertif university

Collection Implementations

- *HashMap* et *Hashtable* sont presque identiques; la différence est que *Hashtable* ne permet pas de conserver une valeur nulle alors que *HashMap* le permet
- *Hashtable* et *Vector* sont *synchronized* et supportent le traitement multitâches
  - les autres classes exigent que la synchronisation soit effectuée en dehors des appels aux méthodes de ces classes

NavigableSet

- La classe *java.util.NavigableSet* offre des méthodes permettant de faire une recherche spécifique d'éléments dans un *TreeSet*.
- Les méthodes importantes sont :
  - *lower()* : retourne le plus grand élément précédent à celui en paramètre
  - *floor()* : retourne le plus grand élément égal ou précédent à celui en paramètre
  - *higher()* : retourne le plus petit élément suivant à celui en paramètre
  - *ceiling()* : retourne le plus petit élément égal ou suivant à celui en paramètre
  - *pollFirst()* et *pollLast()* : retourne le premier ou dernier élément et le supprime de la liste
  - *descendingSet()* : retourne la liste des éléments dans l'ordre inverse

NavigableMap

- La classe *java.util.NavigableMap* offre des méthodes permettant de faire une recherche spécifique d'éléments dans un *TreeMap*.
- Les méthodes importantes sont :
  - *lowerKey()* : retourne la plus grande clé précédente à celle en paramètre
  - *floorKey()* : retourne la plus grande clé égale ou précédente à celle en paramètre
  - *higherKey()* : retourne la plus petite clé suivante à celle en paramètre
  - *ceilingKey()* : retourne la plus petite clé égale ou suivante à celle en paramètre
  - *pollFirstEntry()* et *pollLastEntry()* : retourne la première ou dernière clé et la supprime de la liste
  - *descendingMap()* : retourne la liste des clés dans l'ordre inverse

JCertif university

Generics

- Collection de membres d'un même type connu du compilateur.
- Avec les génériques, il n'est plus requis de transformer (cast) un objet d'une collection pour l'utiliser.
- Grâce aux génériques, le compilateur peut détecter rapidement la manipulation d'objets de types incompatibles (code plus robuste).
- L'identifiant entre « < » et « > » peut être une classe ou une interface
  - ex.: *HashMap<String, Employe> employes;*
- Lors de l'instanciation, il est recommandé de spécifier le type dans le constructeur également – si non, un message d'avertissement sera lancé par le compilateur
  - ex.: *employes = new HashMap<String, Employe>();*

JCertif university

Generics

- Plus de *ClassCastException* puisque le type des objets qui sont ajoutés dans la collection sont vérifiés à la compilation.
- La notation *<E>* indique que la méthode supporte les génériques de toute sorte. ex.: *<String>, <Integer>,* etc.
- La notation *<? extends E>* signifie que :
  - si E est une classe, alors ? doit être de même type que E ou une classe enfant de E
  - si E est une interface, alors ? doit être la même interface, une interface enfant de E ou une classe qui implémente E

JCertif university

Diamond Operator

(nouveauté dans la version 7.0)

- Pre-generics

```
List strList = new ArrayList();
```

- With Generics

```
List<String> strList = new ArrayList<String>();
List<Map<String, List<String>> strList =
  new ArrayList<Map<String, List<String>>>();
```

- With diamond (<>) compiler infers type

```
List<String> strList = new ArrayList<>();
List<Map<String, List<String>> strList =
  new ArrayList<>();
```

JCertif university

Agenda

- ☑ Declaration and Access Control
- ☑ Object Orientation
- ☑ Assignments
- ☑ Operators
- ☑ Flow Control, Exceptions and Assertions
- ☑ Strings, I/O, Formatting, and Parsing
- ☑ Generics and Collections
- ☑ Inner Classes
- ❑ Threads
- ❑ The exam

Inner Classes

- Possibilité de définir une classe à l'intérieur d'un bloc
  - entre une accolade ouvrante et une accolade fermante
- Lorsque l'on déclare une classe à l'intérieur d'une autre classe, celle-ci est nommée en fonction de sa classe parent et du package à laquelle cette dernière appartient
  - ex.: *com.InnerClasse.ClasseInterne*
- Sur le disque, le fichier *.class* qui représente la classe est nommé comme suit : *InnerClasse$ClasseInterne.class*
- La classe *inner* a accès aux attributs et méthodes de la classe parent comme s'ils avaient été définis dans celle-ci.
- On doit toujours instancier la classe parent pour pouvoir instancier une classe enfant puisque celle-ci ne peut vivre sans sa classe parent.
  - ex.: *new InnerClasse().new MonCercle()*

JCertif university

Member Classes

- On peut appliquer les modificateurs d'accès *public*, *protected*, default et *private* au classes *inner* – celles-ci respectent les mêmes règles de visibilité qui s'appliquent aux méthodes ainsi qu'aux attributs.
- On peut déclarer une classe *inner* statique – pour avoir accès à celle-ci, il faut instancier la classe parent comme suit :
  - *new InnerClasseStatique.ClasseInterne*
  - la classe parent doit toujours être instanciée avant de pouvoir utiliser une classe *inner*
- Une classe *inner* statique ne peut pas accéder aux attributs et méthodes non statiques de la classe parent.

JCertif university

Classes Defined Inside Methods

- Une classe définie à l'intérieur d'une méthode ne peut pas avoir de modificateur d'accès (*private*, *public*, *protected*, default) puisque celle-ci est locale à la méthode.
- Une classe définie à l'intérieur d'une méthode ne peut pas être déclarée *static.*
- En résumé, les règles qui s'appliquent à ce type de classe sont les mêmes que celles qui s'appliquent aux variables qui sont définies dans une méthode.

Accessing Method Variables

- Toute variable ou argument déclaré *final* dans une méthode est accessible à la classe déclarée à l'intérieur de celle-ci.
- La raison de cette restriction est la suivante : tout objet qui peut être instancié à partir d'une classe définie dans une méthode pourrait vivre en dehors de celle ci et comme les variables et les arguments sont locaux à une méthode, ceux-ci cessent de vivre après l'exécution de la méthode.
- Une classe définie à l'intérieur d'une méthode a cependant accès aux méthodes et attributs définis au niveau de la classe parent.

JCertif university

Anonymous Classes

- Classes auxquelles aucun nom n'est attribué.
- Une classe anonyme peut être déclarée pour hériter d'une classe existante ou implémenter une interface.
- On ne peut pas implémenter plus d'une interface lorsque l'on déclare une classe anonyme.
- On peut cependant hériter une classe anonyme d'une classe parent qui implémente plusieurs interfaces.
- On doit instancier une classe anonyme lors de sa déclaration à l'aide du mot-clé *new.*
- Une classe anonyme ne doit pas contenir plus de dix lignes de code.

JCertif university

Anonymous Classes

- On ne peut pas définir de constructeur dans une classe anonyme parce que celle-ci n'a pas de nom.
- On peut faire appel au constructeur d'une classe héritée en spécifiant des valeurs entre parenthèse suite au nom de la classe héritée
  - ex.: *new Button(« libellé ») { /* contenu de la classe ici */ }*
- On peut *simuler* un constructeur dans une classe anonyme en déclarant un bloc sans nom à l'intérieur de celle-ci.

JCertif university

Agenda

- ☑ Declaration and Access Control
- ☑ Object Orientation
- ☑ Assignments
- ☑ Operators
- ☑ Flow Control, Exceptions and Assertions
- ☑ Strings, I/O, Formatting, and Parsing
- ☑ Generics and Collections
- ☑ Inner Classes
- ☑ Threads
- ❑ The exam

JCertif university

Thread Fundamentals

- La gestion de traitements multitâches s'effectue selon les trois aspects suivants :
  - classe *java.lang.Thread*
  - classe *java.lang.Object*
  - Java language & JVM
- En Java on associe une instance d'un thread à la classe *java.lang.Thread.*

JCertif university

What a Thread Executes ?

- Pour exécuter un thread, il faut faire appel à la méthode *start* de celui-ci.
- L'appel à cette méthode enregistre celui-ci dans le céduleur de threads.
- Le céduleur décide quel thread exécuter sur le CPU.
- Le céduleur peut être implémenté dans la JVM ou directement dans l'OS.
- Appeler la méthode *start* ne démarre pas l'exécution de celui-ci mais rend celui-ci éligible à l'exécution dans le céduleur.

JCertif university

What a Thread Executes ?

- Lorsque le céduleur décide d'exécuter un thread, il fait appel à la méthode *run* de celui-ci.
- La méthode *run* se situe à deux endroits :
  - dans la classe *java.lang.Thread* (héritage)
  - dans toute autre classe qui implémente l'interface *java.lang.Runnable* (dont l'instance est passée au constructeur de la classe j*ava.lang.Thread*)
- On appellera donc pas la méthode *run,* mais plutôt la méthode *start* – c'est le céduleur qui va appeler la méthode *run.*

JCertif university

When Execution Ends...

- Lorsque la méthode *run* termine son exécution, le thread se termine également (état *dead*).
- Lorsqu'un thread termine son exécution (état *dead*) celui-ci ne peut pas être démarré de nouveau.
- Si on veut démarrer à nouveau un thread, il faut instancier la classe *java.lang.Thread* de nouveau.
- Un thread dans l'état *dead* vit toujours en mémoire et peut être accédé via ses attributs ou ses méthodes – il ne peut juste plus être exécuté en multitâche.

JCertif university

When Execution Ends...

- La classe *java.lang.Thread* contient une méthode *stop* qui est *deprecated* et qui par conséquent ne fait pas l'objet de la certification (pour forcer l'arrêt d'un thread, il faut utiliser une autre technique).
- Lorsque l'on implémente l'interface *java.lang.Runnable*, on peut relancer l'exécution d'un thread en créant une nouvelle instance de la classe *java.lang.Thread.*

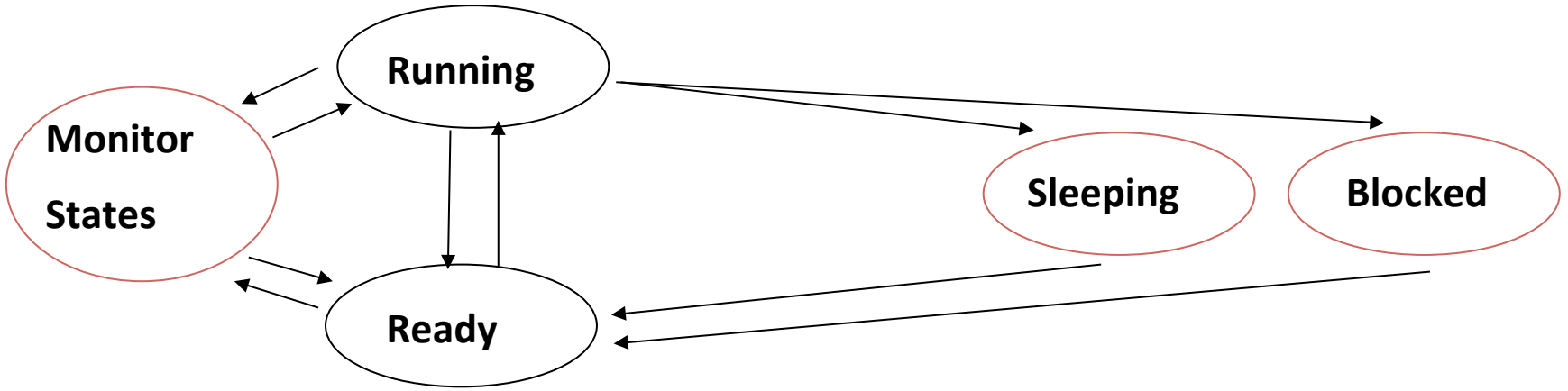
JCertif university

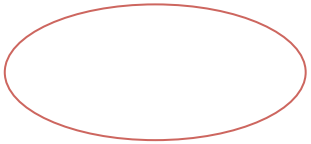
Thread States

- Les différents états possibles d'un thread :
  - Running : lorsqu'un thread s'exécute (méthode *run*)
  - Monitor States, sleeping, suspended, blocked : lorsqu'un thread ne s'exécute plus (méthode *run* en arrêt)
  - Ready : en attente d'exécution (suite à l'appel de la méthode *start*)
  - Dead : exécution terminée

Thread States

Thread Scheduler = gestionnaire des états des threads





= not ready states

JCertif university

Thread States

- Énumération concernant les états d'un thread
- *Thread.State*
  - *NEW* (thread non démarré)
  - *RUNNABLE* (thread en cours d'exécution)
  - *BLOCKED* (thread en attente - verrou)
  - *WAITING* (thread en attente d'exécution)
  - *TIMED_WAITING* (thread en attente avec délai)
  - *TERMINATED* (thread terminé)
- Méthode *getState()* pour obtenir l'état d'un thread

JCertif university

Thread Priorities

- Tous les threads ont une priorité qui se présente sous forme d'entier de 1 à 10.
- Plus la priorité est élevée, plus le thread risque de s'exécuter en premier.
- Si plusieurs threads ont la même priorité, le céduleur ne garantit pas que celui qui sera choisit sera celui qui attends (état ready) depuis le plus longtemps.
- La priorité par défaut est 5.
- On peut changer la priorité d'un thread à l'aide de la méthode *setPriority()* et on peut obtenir la priorité d'un thread à l'aide de la méthode *getPriority().*

JCertif university

Thread Priorities

- Il existe trois constantes qui représentent les priorités des threads :
  - *Thread.MAX_PRIORITY* (10)
  - *Thread.MIN_PRIORITY* (1)
  - *Thread.NORMAL_PRIORITY* (5)
- La spécification indique que les threads doivent avoir des priorités, mais ne dicte pas précisément l'implémentation des céduleurs – ceci a pour effet de ne pas garantir l'ordre de priorité sur toutes les plateformes.

JCertif university

Daemon Threads

- Threads de type infrastructure
  - garbage collector
  - GUI event processing
  - thread principal (méthode *main*)
- Méthode *setDaemon()* permet de définir un *thread* daemon.
- Un thread de type daemon a la même durée de vie que la JVM.

JCertif university

Thread Stack

- Deux méthodes pour obtenir la pile d'un thread ou de tous les threads :
  - *getStackTrace()* affiche la pile d'un thread
  - *getAllStackTrace()* affiche la pile de tous les threads (incluant les threads de type daemon)

Controlling Threads : Yielding

- Un appel à la méthode *yield* place le thread qui est en cours d'exécution (état *running*) dans l'état *ready.*
- S'il y a d'autres threads dans l'état *ready,* alors le thread sera mis en attente (état *ready*).
- Si aucun thread n'est en attente, alors le thread continuera son exécution.
- La plupart des céduleurs ne vont pas placer un thread à l'état *ready* si les autres threads en attente (état *ready*) on une priorité moindre.
- La méthode *yield* est une méthode statique de la classe *java.lang.Thread* et met en attente d'exécution (état *ready*) le thread en cours d'exécution qui a fait appel à celle-ci.

JCertif university

Controlling Threads : Sleeping

- Un thread à l'état *sleeping* passe un certain temps sans s'exécuter et utiliser du temps CPU.
- Un appel à la méthode *sleep* demande au céduleur de mettre le thread en cours d'exécution (état *running*) à l'état *sleeping* pendant une période de temps définie.
- On peut spécifier le temps d'attente en millisecondes ou en nanosecondes.
- La méthode *sleep* est une méthode statique de la classe *java.lang.Thread* et met en arrêt temporaire (état *sleeping)* le thread en cours d'exécution qui a fait appel à celle-ci.

JCertif university

Controlling Threads : Sleeping

- Lorsque le délai d'attente est expiré le thread passe à l'état *ready* et non à l'état *running* où il était lors de l'appel à la méthode *sleep().*
- Il ne s'exécutera pas forcément immédiatement après son délai; c'est le céduleur qui décidera à nouveau du moment de son exécution.
- Un appel à la méthode *sleep* pourrait arrêter l'exécution d'un thread pendant un moment plus long que celui spécifié.

JCertif university

Controlling Threads : Sleeping

- Un thread à l'état *sleeping* pourrait être interrompu à l'aide de la méthode *interrupt* de la classe *java.lang.Thread.*
- Si cette situation arrive, le céduleur va placer le thread dans l'état *ready* et lorsque le thread sera prêt à être exécuté (état *running*), l'exception *InterruptedException* sera lancée suite à la sortie de la méthode *sleep.*
- Il faut donc gérer l'exception *InterruptedException* lorsque l'on fait appel à la méthode *sleep.*

Controlling Threads : Blocking

- Un thread qui se trouve dans l'état *Blocking* est en attente de terminaison d'une opération d'entrée / sortie.
- Cette situation se produit généralement lorsqu'on tente de lire ou d'écrire des octets sur le réseau et qu'un délai d'attente se fait sentir – pour ne pas bloquer l'exécution des autres threads, le céduleur va placer le thread en attente (état *blocking*) afin de permettre aux autres threads de s'exécuter.
- Cet état s'active lorsque l'on fait appel aux méthodes *wait* et *notify* de la classe *java.lang.Object.*

JCertif university

Controlling Threads : Monitor States

- La méthode *wait* de la classe *java.lang.Object* indique au céduleur de changer l'état d'un thread de *running* à *waiting.*
- Les méthodes *notify* et *notifyAll* de cette même classe indiquent au céduleur de changer l'état d'un thread de *waiting* à *ready.*
- Ces méthodes peuvent être appelées dans du code synchronisé (*synchronized*).

JCertif university

Monitors, Wait and Notify

- Un monitor est un objet qui peut bloquer et faire revivre un thread.
- Un monitor c'est l'attente d'un changement d'état d'un objet par un thread avant d'effectuer son travail (exécution).
- Un monitor fournit les fonctionnalités suivantes :
  - un lock pour chacun des objets
  - l'utilisation du mot-clé *synchronized* pour accéder au lock d'un objet
  - les méthodes *wait*, *notify* et *notifyAll* qui permettent à l'objet de « contrôler » l'exécution du thread

JCertif university

Object Lock and Synchronization

- Tous les objets ont un verrou.
- Ce verrou est contrôlé par du code synchronisé.
- Un thread qui désire exécuter du code synchronisé doit obtenir un verrou sur l'objet en question.
- Si le verrou est disponible – il n'est pas en cours d'utilisation par un autre thread – alors le thread peut poursuivre son exécution – il devient alors le maître de ce verrou jusqu'à la fin de l'exécution du code synchronisé.
- Si le verrou est sous le contrôle d'un autre thread, le thread en attente d'exécution est alors placé dans l'état *seeking lock.*
- Lorsque le verrou sera libéré, le thread sera à nouveau placé dans l'état *ready* par le céduleur.

JCertif university

Object Lock and Synchronization

- Le processus de gestion des verrous est transparent – tout ce que le programmeur doit faire c'est déclarer un bout de code *synchronized* comme suit :
  - synchroniser une méthode entière en utilisant le mot-clé *synchronized* dans sa déclaration
    - pour exécuter la méthode, un thread devra acquérir un verrou sur l'objet qui contient la méthode
  - synchroniser un sous-ensemble d'une méthode en entourant celle-ci entre accolades ({ }) et en insérant l'expression suivante : *synchronized(objetContenantLeVerrou)*
    - cette technique permet de synchroniser un bout de code à partir de n'importe quel objet (les types primitifs ne sont pas autorisés / les vecteurs (array) peuvent être synchronisés mais pas leur contenu ie : les références d'objets qu'ils contiennent)

JCertif university

Object Lock and Synchronization

- Lorsque l'on chevauche (override) une méthode et que celle-ci est déclarée *synchronized* la méthode qui chevauche n'est pas *synchronized* par défaut.
- On ne peut pas appliquer le mot-clé *synchronized* à un constructeur
  - on peut cependant synchroniser un bout de code à l'intérieur d'un constructeur
- On ne peut pas appliquer le mot-clé *synchronized* aux méthodes d'une interface.
- Java supporte les verrous de type re-entrant ce qui permet à une méthode synchronisée d'une classe d'appeler une méthode synchronisée de cette même classe.

JCertif university

Wait and Notify

- Ces méthodes permettent à un thread de faire une pause pour ensuite poursuivre son exécution là où il était rendu par la suite.
- Ces méthodes doivent être appelées dans un bloc de code synchronisé.
- La méthode *wait* libère un verrou et met le thread en attente (état *waiting*).
- La méthode *notify* sélectionne de façon arbitraire un thread qui est dans l'état *waiting* et le place dans l'état *seeking lock*
  - on ne peut donc pas sélectionner le thread en question – on peut cependant forcer le changement d'état de tous les threads à l'aide de la méthode *notifyAll*

JCertif university

Wait and Notify

- On peut spécifier un délai d'attente dans la méthode *wait* – lorsque le délai arrive à expiration, le thread est alors placé dans l'état *seeking lock.*
- Si le thread est dans l'état *waiting* et qu'un appel à la méthode *interrupt* est effectué alors le thread est alors déplacé dans l'état *seeking lock.*

JCertif university

Class Lock

- Tous les objets ont un verrou.
- Les classes ont elles aussi un verrou.
- Le verrou d'un objet permet de synchroniser le code déclaré dans une méthode d'instance (dans ce cas, le verrou s'applique à l'objet *this*).
- Le verrou d'une classe permet de synchroniser le code déclaré dans une méthode statique (dans ce cas, le verrou s'applique à la classe elle-même).
- Lorsque la JVM charge une classe en mémoire, une instance de la class *java.lang.Class* est créée pour la classe chargée et le verrou est alors possible grâce à celle-ci
  - toutes les classes et instances de classes peuvent avoir accès à cet objet via la méthode *getClass()* de la classe *java.lang.Object*

JCertif university

Deadlock

- Si un thread (a) est en attente d'un verrou et que ce verrou est déjà occupé par un autre thread (b) et que celui-ci est en attente d'un verrou que le premier thread (a) occupe, alors on est en situation de *deadlock.*
- Si deux threads sont en attente d'un verrou que l'autre thread possède, on est en situtation de *deadlock.*
- Le mot-clé *synchronized* représente un verrou, pour détecter une situation de *deadlock* il faut donc se fier à celui-ci.
- Des synchronisations imbriquées peuvent être une source d'une situation de *deadlock.*