Switch caseはJavaSE7からStringの使用が可能
JavaSE8からdefaultメソッドとstaticメソッドを宣言可能になった

Objectクラスは11のメソッドが宣言されており、全てのクラスはこれらのメソッドを継承していることになる。
試験に出るのは以下の3つ
Public boolean equals(Object obj){　return (this == obj);　}メモリアドレスの比較、インスタンスが等しいかを判定
int hashCode()　このオブジェクトのハッシュコードを返す　オブジェクトを検索する際のパフォーマンス向上に用いられる
異なるオブジェクトは可能な限り異なるハッシュ値を返す実装が望ましい
String toString　このオブジェクトの文字列表現を表す

全てのクラス(自作クラスも含む)で上記のメソッドをオーバーライドすることが望ましい
特にhashCode()とequals()は同時にオーバーライドすることが推奨

final　クラスは継承不可　変数は再代入不可　メソッドはオーバーライド不可
仮引数にも使用可能　abstractとの併用は不可
一般的にfinalフィールドにstaticをつけることが望ましい

staticメンバはインスタンス変数からでもアクセス可能だが、本来はクラス名を指定して参照するのが望ましい

staticは初期化が必須　コンストラクタ　staticイニシャライザ等を使う
equlas()メソッド→ハッシュ値が同じか、equals()をオーバーライドしたらhashcode()も合わせてオーバーライドする

シングルトン

private staticフィールドを宣言し、自身の単一のインスタンスを生成して、そのフィールドにセットする
→インスタンスを1つだけ生成し、常にその1つのインスタンスのみを参照できるようにする
コンストラクタをprivateで宣言する
→外部からインスタンス生成できないようにする。クラス内部でのみ生成可能にする
自身のフィールドにセットされているインスタンスを返すpublic staticメソッドを提供
→外部からこのクラスのインスタンス変数を取得できるよにする

```
e.g. public class Foo{
        private static final Foo foo = new Foo();
        private Foo{}
```

```
        public static Foo getInstance(){
            return foo;
        }
    }
```

不変クラス　　スレッドセーフなオブジェクトとして利用可能

そのクラスから生成されたインスタンスおいて、一度設定された状態(フィールド値)を変更できないように設計されたクラス
String,
Integer等のプリミティブ型ラッパークラス
BigDecimal
java.awt.Color
java.time.LocalDate
などが代表的な不変クラス

設計する際は以下を実装
クラス　　継承されないようにfinal宣言
フィールド　　外部からアクセス不可、値の変更不可にするためprivate finalに指定
アクセサメソッド　ゲッターのみ提供し、セッター等の値を変更するメソッドは一切提供しない
配列やコレクションなどの可変コンテナクラスや可変クラスへの参照を外部に公開しないようにする
this参照を外部に公開しないようにする
コンストラクタや普通のメソッドは特に制約はない

abstract
具象・抽象メソッドを記述可能　アクセス修飾子に制限はない。子クラスは公開範囲を広めることは可能だか狭めることは不可
抽象クラスを実装した具象クラスは必ず抽象メソッドを実装しなければならない→コンパイルエラー
抽象メソッドを記述すると必ず明示的にabstractを指定する

interface
アクセス修飾子はpublicのみ　final修飾子も不可
変数宣言は自動的にpublic static final　メソッドはpublic abstractが暗黙的に付与
メソッドはpublic final
SE8からデフォルト・staticメソッド(static抽象メソッドは記述不可)の記述可
デフォルトメソッドはオーバーライド可能(defaultは削除しpublicをつけて再定義)
staticメソッドは具象を記述可　publicのみ(省略すると暗黙的にpublic付与)
implementしなくても他クラスから参照可
インターフェースに記述したstaticメソッドは参照変数.メソッド名での呼び出しは不可　インターフェース名.メソッド名を使用
defaultメソッド(具象メソッド)は public(固定) default 戻り値 メソッド名(引数リス

ト){} equals hashcode tostringは定義不可
複数のインターフェースを実装した場合、特定のメソッドを呼ぶには、親インターフェース名.super.メソッド名()

シグニチャが同じデフォルトメソッドを記述した複数のインタフェースをimplementsした場合コンパイルエラー
→具象クラスでオーバーライドすることにより解決可能

implementsによる多重継承した場合は、シグニチャが同じデフォルトメソッドを記述した複数のインタフェースの距離が
同じならコンパイルエラー、距離が異なる場合は近い方のデフォルトメソッドが呼ばれる　黒本59参照s

型変換
→暗黙的キャスト
byte short int long float double　char int　演算時左右オペランドのうち下位が上位のdouble float long int に変換される
←明示的キャスト
継承関係のないものをキャストするとコンパイルエラー　継承関係のあるもでキャストできない場合はClassCastException
子クラスを親クラスの型で生成した場合、非staticメソッド以外は全て親クラスのメンバ(staticメソッドも)が呼ばれる
完全一致＞暗黙の型変換＞AutoBoxing>可変長引数

ネストクラス（クラスの中にクラスを記述可)
インタフェースや抽象クラスの定義も可能
staticクラスか非static(インナークラス)の２つ　外側クラス名と重複不可　アクセス修飾子可 abstract final可能
非staticクラスの場合　out.A a = new out().new A();　new out().new A().methodA();
new A().methodA()はNG
staticの場合 out.B b = new out.B();　new out.B.methodB() new B().methodB()
out.B.methodB() B.method()も可能
outは入れ子の外側クラス
staticクラスは非static・staticメンバを記述可　外側クラスのインスタンス変数(フィールド変数(非static))にアクセス不可
非staticクラスはstaticメンバをもつことができない。外側クラスのインスタンス変数にアクセス可能
同名変数はthisを用いる、外側クラスの同名変数にアクセスするには外側クラス名.this.変数名

ローカルクラス(メソッドの中にクラスを記述、非staticのみ)
アクセス修飾子を使用できない。abstractとfinalは使用可　外側クラスのメンバにアクセス可能
外側クラスの引数・ローカル変数にアクセス可能→アクセスする変数がfinalでない

といけない(代入不可)、SE8から自動で付与　フィールド変数はfinal付与は不要

匿名クラス(アクセス修飾子・static・abstract・final使用不可)　代入可能
クラス指定をしないで、クラス定義とインスタンス化を１つの式として定義。サブクラス・インターフェースを実装したクラスになる
外側クラスのメンバにアクセス可能
外側クラスのメソッドの引数・ローカル変数にアクセス可能(ローカル変数・引数は要final)、コンストラクタの定義不可
New　スーパクラス・インターフェース名(){};　無名の実装クラスがインスタンス化される

列挙型　全ての列挙型は暗黙的にjava.lang.Enumのサブクラスとしてコンパイルされるのでextendsは不可 implementsは可能

関連する定数を一つにまとめるための型　JavaSE5で導入
列挙型は暗黙的にfinalでコンパイルされる
ネストされた列挙型は暗黙的にstaticになる
コンストラクタを記述した場合は暗黙的にprivateになる
引数を取るコンストラクタを記述した場合は、引数を渡さない列挙定数の宣言は不可
構文　　就職し enum 識別子き{列挙定数１, 列挙定数2, …}
```
e.g. public enum Stoplight{
        GREEN, YELLOW, RED; //最後にセミコロンはつけてもつけなくてもどちらでも良い
    }
```
クラスやインタフェースと同じように、トップレベル(public or パッケージなし)もしくは入れ子の型として宣言することができ、コンパイルされる
ローカルクラスとは異なり、メソッド内で宣言することはできない
実体はクラスなのでメンバーとしてフィールドやメソッドを宣言することができる
ただし、列挙定数はフィールドやメソッドよりも前に記述し、最後の列挙定数の後ろにはセミコロンを記述する必要がある
列挙定数は暗黙的にpublic static finalなメンバーとなるため、明示的に修飾子をつけることはできない
列挙型にはアクセス修飾子をつけることはできるが、列挙定数にはいかなるアクセス修飾子もつけることはできない
```
e.g. public enum Stoplight{
        GREEN, YELLO, RED;
        private int val;
        void someMethod()
    }
```
列挙定数を文字列として返すには以下の2つのメソッドがある
`String green = Spotlight.GREEN.name();`　　Enumクラスでfinal宣言されている

String green = Spotlight.GREEN.toString();　　オーバーライド可能　使用推奨

文字列から列挙型インスタンスを取得するにはvalueOfを使用 存在しない文字列を指定した場合にはIllegalArgumentException 大文字小文字識別
int original()　　列挙定数の序数を返す　0からスタート
static E[] values()　　列挙型の配列を返す

Collectionインターフェース

上３つはCollectionがルート　MapはMapがルート　java.utilをインポート
List　　データ項目に順序付けしたコレクション　順序づけて管理　重複可能 ArrayList LinkedList Vector E remove(int index)を実装
Set　　ユニークな値のコレクション　順不同で管理　重複不可　HashSet TreeSet LinkeHashSet
Queue　　FIFO形式のデータを入出力で行うコレクション ArrayDeque
　　削除はremove()しかないので引数を指定するとCollectionインターフェースのメソッドが呼ばれる boolean remove(Object o)
Map　　個々のキーに対応する値をマップしたオブジェクト　キーの重複不可、値の重複は可能 HashMap TreeMap LinkedHashMap

ArrayList　　挿入削除は線形的に実行、挿入削除の頻度が高くランダムアクセス頻度が低い場合はLinkedListが適している　<T>は参照型ならなんでも良い
　　ランダムアクセス(検索)は高速、挿入と削除は低速　同期性はサポートしていない（スレッドセーフではない)
LinkedList　　各要素(ノード)に個々のデータ項目に加えて次のノードに対するポインタが格納　挿入と削除がArrayListより高速　同期性はサポートしていない
Vector　　同期性をサポート　マルチスレッド環境を必要としない場合はArrayListを使用しないとパフォーマンスが低下する
上記のクラスはインデックス順でソートはなし

HashSet　　データ項目アクセスはTreeSetより高速、データ項目を順序づけできない　ソートも順序付けも行わない　同期性はサポートしない　重複不可　順不同
TreeSet　　SorrtedSetインターフェースの実装クラス。ソートされたデータ項目を得る、HashSetよりも低速　同期性のサポートはしていない　重複不可
　　キー元にソートを管理、自然順序付け(文字列は辞書順、数値は昇順)
LinkedHashSet　HashSetと同等の機能に加え、全てのデータ項目に対する二重リンクリストを追加、挿入順で順序づけられ　同期性のサポートはしていない

HashMap　　ハッシュテーブルのデータ構造体をベースにした実装。キーと値は順不同で格納　nullをキー・値として使用可能　同期性はサポートしていない
LinkedHashMap 全てのエントリに対する二重リンクリストを保持するという点でHashMapと異なる。キーの挿入順で順序付け　同期性サポートしていない

TreeMap　　SortesMapインターフェースを実装したクラス　キーの昇順による順序付けが維持_(ソート)　同期性はサポートしていない

Collectionインターフェースの主なメソッド紫本８５ページ参照

メソッド名	説明
Boolean add(E e)	引数の要素をコレクションに追加する。コレクションが変更された場合はtureを返す
void clear()	このコレクションから全ての要素を削除する
boolean contains(Object obj)	指定された要素objがこのコレクション中に存在する場合はtrueを返す
boolean containsAll(Collection<?> c)	指定されたコレクションの要素が全てのコレクションに含まれている場合はtrueを返す。
boolean remove(Object o)	引数に指定された要素を削除し、要素が削除された場合はtrueを返す
boolean removeAll(Collection<?> c)	メソッド引数に指定されたコレクションにある全ての要素をこのコレクションから削除し、コレクション内容に変化があった場合はtrueを返す
boolean removeif(Predicate<? super E> filter)	指定された処理を満たすこのコレクションの要素を全て削除する。ラムダ式
Object[] toArray()	このコレクションの要素が全て格納されている配列を返す
<T> T[] toArray(T[] array)	このコレクションの全ての要素を含む配列を返す。Tは配列要素のデータ型を返す。
Int size()	コレクション中の要素数を返す。

asList()はListオブジェクトを生成して返す。java.util.ArrayListクラスと型の互換性はない

ArrayList<integer> list = Arrays.asList(1,2,3)は不可

ArrayList<integer> list = new ArrayList<>(Arrays.asList(1,2,3))は可能

Mapインターフェースの主なメソッド紫本８５ページ参照

メソッド名	説明
V put(K key, V value)	指定されたキー値ペアを格納。重複する場合は新しい値で上書き
Void putAll(Map<? extends K, ? Extends V> m)	指定されたマップにある全てのキー値ペアをこのマップに格納する
V remove(Object key)	キー値ペアが存在する場合は、それを削除。戻り値は、指定キーに割り当てられた値(削除処理前) を返し、値が存在しない場合はnullを返す
Int size()	このマップ中に存在するキー値ペアの数を返す。
Collection<V> values()	このマップ中の値をコレクションとして返す。

merge()　第一引数にキー、第二引数に値、第三引数にBiFunctionを指定。ラムダ式の引数にはmergeメソッドの第一引数のキーの値、第二引数が渡される
キーが存在しない場合は第一引数のキーが生成され、値は第二引数で指定したものとなる
Map.Entry<K, V>インタフェースはMapオブジェクトに格納されている1つの要素を表現する。entrySet使用して取り出す
for(Map.Entry<Integer, String> entry : map.entrySet()){} getKey,getValueでキーと値の取り出し

ジェネリクス
<>はSE７より型推論により省略可　型パラメータ(<>)で扱えるデータ型は参照型のみ　staticメンバには使用不可　インターフェースで使用可
配列、インスタンス生成、instance of、.classの参照は不可
メソッドにのみ使う場合は アクセス修飾子 <T> 戻り値の型 メソッド名(データ型 引数)　呼び出す際は<>をつけても省略してもどちらでも良い
クラスに使用する場合は class Hoge<T> { private T foo; public T getHoge(return foo;)}
Newする際には1.new Hoge(ture) 2.new Hoge<>(10) 3.new Hoge<String>(“ABC”)どれでもコンパイル可能　ただし1は警告が出る
<T extends Object> TはObjectクラスかサブクラスに対応しなければならない　独自クラスの場合はTを？に変える
<T super タイプ> タプに指定したデータ型やそのスーパクラス(スーパインターフェースも)に対応
<? Extends hoge>の場合、実行するまで?の型がわからないので、add等の何かしらのオブジェクトを格納するコードを記述するとコンパイルエラー
<? Super hoge>の場合は、hogeと同じ型の場合は要素の追加が可能

ジェネリックメソッド

宣言されているメソッドだけをジェリクスだけで利用可能
型パラメータで宣言した型変数はそのメソッドのスコープ内において自由に参照可能
構文　〈型パラメーター〉　戻り値　メソッド名(引数) {}

```
class Foo {
	<T> voidIt(T t){}
}
class Foo{
	<T> T doit(){}
}
```

ComparableとComparator

ソートを利用するには、Comparableを実装しなければならない。java.langパッケージ　ラムダ式で実装不可

実装しないでTreeSetなどのソートするオブジェクトに格納すると実行時例外
Integer,Double,StringなどはComparableを実装しているのでTreeSetオブジェクト等に格納できる
Comparableインターフェースには、compareTo()メソッドのみ宣言されている。自然順序を提供(昇順)
Public int compareTo(T o)　自オブジェクトと引数oに渡されたオブジェクトを比較結果を整数で返す
比較ルール　　==は０　自＜比較対象→負の数　自＞比較対象→正の数

```
e.g. public int compareTo(Product o){
        return this.id - o.id;        ==は０　自＜比較対象→負の数　自＞比較対象→正の数
    }
```

```
Import java.util.*;
Class hoge implements Comparator<String>
Public int compare(String s1, String s2){
    return s1.compareTo(s2)          自然順序　s1.toLowerCase()で文字を小文字にする
    return s2.compareTo(s1)          逆になる　s1.toUpperCase()で文字を大文字にする
}
```

Comparatorインターフェースは比較ルールを独立したクラスとして定義可能
java.utilパッケージ　　ラムダ式で実装可
compare(), equals()メソッドで宣言されている。compare()メソッドをオーバーライドして並び順を決定する実装を行う。
Comparatorには抽象メソッドとしてequals()メソッドが宣言されているがObjectクラスのPublicメソッドため、関数型インターフェース仕様に沿っている。実装は任意
Staticメソッドとして自然順序付けを行うnaturalOrder()メソッドを持つ。
Public int compare(T o1, T o2)　interface Comparator<T>となっているので、比較対象のデータ型をTに指定する必要がある
比較ルールは上記と同じ o1 < o2　なら負の整数 o1 > o2 なら正の整数　o1 == o2 なら０を返す　サンプルコードでは1,-1,0が返る

配列のソートはArraysクラスを用いる。sortメソッドで自然順序、コンパレータを指定すると独自定義でソート
様々な参照型のオブジェクトを要素に持つ配列をsortメソッドでソートするとClassCastExceptionが発生
配列をリストに変換→asListメソッド。変換されたリストは要素の上書き可、固定サイズのリストのため要素の追加や削除は不可 UnsupportedOperationException発生

ComparableとComparatorの違い

Comparable　自身が他のオブジェクトと比較可能　compareTo(T o)
java.lang　多くの値クラス(String,Integerなど)が実装済
ただし、Objectクラスは実装していないので、ユーザーが作成したクラスは全て比較不可となっており、自身で実装する必要あり

Comparator　他のオブジェクトと比較する　compare(T o1, T o2)
java.util　TreeSet, TreeMapなどのに格納される要素の並べ替え順序をデフォルトとは越の順序に変更したい場合
TreeSet, TreeMapはComparatorオブジェクトを引数に取るコンストラクタを提供している
実装の際はimplements Comparator<T>とSerializableも同時にするのが望ましい

```
e.g. class Hoge implements Comaprator<String>{
     @Override
     public int compare(Sting o1, String o2){
          return o1.length() - o2.length(); //文字列の長さによる並べ替え順の制御
     }}

     Set<String> set = new TreeSet<>(new Hoge); //コンストラクタに渡すことで独自定義が適用
     set.add(“hoge”);　　set.add(“zoo”);
     println(set)　//zoo hogeが表示　デフォルトは辞書順だが文字数順に変更されている

     Set<String> set = new TreeSet<>(new Comparator<String>(){
          public int compare(String o1, String o2){
               return o1.length() - o2.length();
          }
     }); //匿名クラスで記述した場合
```

クラス	インターフェース	重複(項目)	順序付け/ソート	同期性	
ArrayList	List	可	インデックス順・ソートなし	無	ランダムアクセス(検索)は高速、挿入と削除は低速
LinkedList	List	可	インデックス順・ソートなし	無	挿入と削除がArrayListより高速
Vector	List	可	インデックス順・ソートなし	有	マルチスレッド環境で使用、シングルではパフォーマンス低下

Set<E>ではnull要素もユニークな要素として認められるが重複不可のため1つだけ格納可能

HashSet Set 不可 順序付けなし・ソートなし 無 データ項目アクセスはTreeSetより高速　順不同
LinkedHashSet Set 不可 挿入順・ソートなし 無 HashSetと同等の機能に加え、全データ項目に対する二重リンクリストを追加
TreeSet Set 不可 自然順or比較ルールでソート 無 SortesMapインターフェースを実装したクラス 要素をソートして管理

要素をComparable<T>にキャスト implementsしないとClassCastException
PriorityQueue Queue 可 自然順or比較ルールでソート 無 優先度ヒープに基づく制限なしの優先度キュー synchronaizedされない

HashMap<K,V> Map 不可 順序付けなし・ソートなし 無 キーと値は順不同で格納 nullをキー・値で使用可 <K,V>を省略するとobject型

streamを使うにはentrySetメソッドを使用し、setを取得する
LinkedHashMap Map 不可 挿入順・アクセス順・ソートなし　無 全エントリに対する二重リンクリストを保持するという点でHashMapと異なる
Hashtable Map 不可 順序付けなし・ソートなし 有 ハッシュ表(キーを値にマップ)を実装　どのオブジェクトでもキー・値に使用可
TreeMap Map 不可 自然順or比較ルールでのソート 無 SortesMapインターフェースを実装したクラス　キーをもとにソート

NavigableMapインターフェースはSortedMapインターフェースのサブインターフェース。以下のメソッドを使用可能
K higher[lower]Key(K key) 指定されたキーよりも確実に大きい[小さい]キーの中で最小[最大]のものを返す。
NavigableMap<K, V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive) fromKeyからtoKeyのキー範囲をもつビューを返す。
fromKey[toKey]：返されるマップ内のキーの下端点[上端点]
fromInclusive[toInclusive]：返されるビューに下端点[上端点]が含まれるようにする場合にはtrue
ceiling()[floor()]メソッドは指定された要素と等しいかそれより大きい[小さい]要素の中で最小[最大]のものを返す
Mapは指定されたキーまたは値が含まれているか確認する場合は、containsKey(), containsValue()を使用する

Arrays Java.util(Collectionは実装していない)

順序付けなしは出力した際に挿入した順では出力されない.出力結果は不定。
自然順序(数値は昇順、文字は辞書順)でソートした際に数値と文字列が混ざっていた場合は数値が先にソート
<>を指定しないでリストを作成すると、様々な型のオブジェクトを格納するobject型のリストができる。拡張forでobject型(警告が出る)以外の型を指定するとコンパイルエラー

Deque

Queue<E>インタフェースのサブインタフェース。両端から要素を挿入・削除できるデータ構造
ArrayDeque<E>, LinkedList<E>などが提供されている　両方ともスレッドセーフではない
インデックスによる要素のアクセスはサポートしない
末尾
addLastで追加 getFirst,removeFirstで取り出し
先頭
addFirstで追加　getLast,removeLastで取り出し

ArrayDeque<E> 内部で配列を使用したDequeインタフェースの実装
・FIFO(キュー)として用いる場合にはLinkedList<E>クラスを使用するよりも高速
・FILO(スタック)として用いる場合にはjava.util.Stack<E>よりも高速
・null要素を格納できない
マルチスレッドのアプリケーションで使用する場合は、スレッドセーフ設計されているStack<E>クラスが安全

ラムダ式(関数型インターフェース)　<>の型指定をしないとコンパイルエラー
引数に__を使うことは不可
外側で定義されている変数を参照可能だがその変数はfinal(実質的finalも)でなければいけない、よってラムダ式内で再代入するとコンパイルエラー
ラムダ式内で外側変数を参照した後にラムダ式の外で参照した変数に再代入してもコンパイルエラー
ラムダ式の外側で定義された変数名をラムダ式内の引数、ローカル変数で定義するとコンパイルエラー
ラムダ式の引数に使った変数名を外側のメソッド内でラムダ式より後で定義して使うのは問題ない
ラムダ式の外側のメソッドがstaticな場合、thisの使用は不可
要件　・単一の抽象メソッド(staticメソッド・defaultメソッドは定義可能)
java.langObjectのpublicメソッドは抽象メソッドとして定義可能
関数型インタフェースとして明示的に宣言する場合は@FunctionInterfaceを付与する abstract classは抽象クラスなので関数型インタフェースではない
参照型の関数型インターフェース　java.util.finction

インターフェース	抽象メソッド	概要
Function<T,R>	R apply(T t)	実装するメソッドは

、引数としてTを受け取り、結果としてRを返す。
BiFunction<T,U,R>　　R apply(T t, U u)　　実装するメソッドは、引数としてTとUを受け取り、結果としてRを返す。
UnaryOperator<T>　　T apply(T t)　　実装するメソッドは、引数としてTを受け取り、結果としてTを返す。Functionを拡張したもの
BinaryOperator<T>　　T apply(T t1, T t2)　　実装するメソッドは、引数としてTを２つ受け取り、結果としてTを返す。BiFunctionを拡張したもの

Consumer<T>　　void accept(T t)　　実装するメソッドは、引数としてTを受け取り、結果を返さない。
BiConsumer<T,U>　　void accept(T t,U u)　　実装するメソッドは、引数としてTとUを受け取り、結果を返さない。

Predicate<T>　　boolean test(T t)　　実装するメソッドは、引数としてTを受け取り、boolean値を結果として返す。

BiPredicate<T,U>　　boolean test(T t,U u)　　実装するメソッドは、引数としてTとUを受け取り、boolean値を結果として返す。

Supplier<T>　　T get()　　実装するメソッドは、何も引数として受け取らず、結果としてTを返す。

基本型の関数型インターフェース(int long double)　下記はint型の場合(long、doubleも同様の命名規則)
インターフェース　　抽象メソッド　　概要
IntFunction<R>　　R apply(int value)　　実装するメソッドは、引数としてintを受け取り、結果としてRを返す。
IntUnaryOperator　　int applyAsInt(int operand)　　実装するメソッドは、引数としてintを受け取り、結果としてintを返す。
IntBinaryOperator　　int applyAsInt(int left, int right)　　実装するメソッドは、引数としてintを２つ受け取り、結果としてintを返す。

IntConsumer　　void accept(int value)　　実装するメソッドは、引数としてintを受け取り、結果を返さない。
IntPredicate　　boolean test(int value)　　実装するメソッドは、引数としてintを受け取り、boolean値を結果として返す。
IntSupplier　　int getAsInt()　　実装するメソッドは、何も引数として受け取らず、結果としてintを返す。

BooleanSupplier　　boolean getAsBoolean()　　引数なし、boolean値を返す　boolean特化インターフェースはこれのみ

Int, double, long固有の関数型インターフェース(異なるデータ型から基本データ型の結果を返す、下記はint型の場合(long、doubleも同様の命名規則)

インターフェース	抽象メソッド	概要
ToIntFunction<T>	int applyAsInt(T value)	実装するメソッドは、引数としてTを受け取り、結果としてintを返す。
ToIntBiFunction<T,U>	int applyAsInt(T t, U u)	実装するメソッドは、引数としてT,Uを受け取り、結果としてintを返す。
IntToDoubleFunction	double applyAsDouble(int value)	実装するメソッドは、引数としてintを受け取り、結果としてdoubleを返す。
IntToLongFunction	long applyAsLong(int value)	実装するメソッドは、引数としてintを受け取り、結果としてlongを返す。
ObjIntConsumer<T>	void accept(T t, int value)	実装するメソッドは、引数としてTとintを受け取り、結果を返さない。

Int, double, longに特殊化した関数型インターフェースは対応したラッパークラスであっても不適合な型となりコンパイルエラー
Char, byte, short, floatは存在しない？ Int, double, longのみ?

ラムダ式の省略記法　右辺
(String str)　→　(str)　→ str
インターフェースの宣言時に引数の型は決定しているため、型推論により型を省略できる。
引数が１つの場合は()を省略可能。データ型を明示した場合や引数がない、複数の引数がある場合は省略できない。引数なしは ()-> と記述する。

ラムダ式の省略記法　左辺
{　return str.toUpperCase();　}　　→　　str.UpperCase()
処理が一文の場合は{}の省略が可能。{}を省略した場合、returnの省略が可能。returnを省略した場合、必ず{}も省略しないといけない。
ラムダ式で記述する場合、(String str1,　str2) -> hoge　は不可(型の省略は不可)
(String str1,String str2) ->　と記述する

メソッド参照
ラムダ式内で呼び出されるメソッドが一つの場合、SE8ではラムダ式を使用せず記述する方法が導入。これをメソッド参照と呼ぶ
メソッド参照では引数が複数の場合でも省略可能
構文 クラス名orインスタンス変数名 :: メソッド名　e.g. Integer::parseInt;
メソッド参照では、クラス内に同じ名前のstaticメソッドのとインスタンスメソッドが存在する場合、呼び出し時に特定できないとコンパイルエラー

Staticメソッド参照
呼び出すメソッドがstaticメソッドの場合「クラス名::メソッド名」

インスタンスメソッド参照

呼び出すメソッドがインスタンスメソッドの場合「インスタンスメソッド名:メソッド名」
どのオブジェクトに対してインスタンスメソッドを呼び出すのかを変数名で指定できない場合は、クラス名で指定　e.g. s::toUpperCase→String::toUpperCase
これによりapply()メソッドの呼び出し時に指定した引数がtoUpperCase()メソッドを実行する対象のオブジェクトとして扱われる
インスタンスメソッド参照の場合はインタフェースの第一引数がメソッドを実行する対象のオブジェクトとして渡され、第二引数がそのメソッドの引数として渡される

コンストラクタ参照
メソッド参照でコンストラクタを呼び出す　構文：「クラス名::new」　主にsupplierを使うことが多い？
配列の生成も可能

```
Function<Integer, String[]> obj1 = length -> new String[length];   ラムダ式
Function<Integer, String[]> obj1 = String[]::new;                  コンストラクタ参照
System.out.println(obj1.apply(5).length);    →    結果は5が出力
```

匿名クラス、ラムダ式、メソッド参照を使った場合の各々のコード量

```
Consumer<List<Integer>> con1 = new consumer<List<Integer>>(){       匿名クラス
        public void accept(List<Integer> list){
                Collections.sort(list);
        }
}
Consumer<List<Integer>> con1 = lamdaList -> Collections.sort(lamdaList);    ラムダ式
Consumer<List<Integer>> con1 = Collections::sort;                    メソッド参照
```

ストリームAPI　（黒本終了後に全ての問題を解き終え、問題に出てきたメソッドを中心に覚える）
コレクション、配列、I/Oリソースなどのデータをもとに、集計操作を行うAPI。
複数の処理の入出力を繋ぐための仕組みを提供しており、これをストリームのパイプライン処理と呼ぶ
パイプライン処理には処理の元となるデータソースが必要。データソースを元にストリームオブジェクトを生成し、後続する処理を行う。
パイプラインの途中で行う処理を中間操作と呼び、パイプラインの最後に行う処理を終端操作と呼ぶ。
各中間操作のメソッドはその中間操作結果をソースとする新しいストリームを返す。
中間操作では「何を行うか」のみをパイプラインで繋げ、終端操作のメソッド実行時に全ての処理が行われる。
中間操作は何度も実行される。終端操作は一回のみしか実行できない。終端操作を

行ったストリームは再度終端操作を行うことができない。(コンパイルエラー)
IllegalStateExceptionが呼ばれる
終端操作を記述しないと、コンパイルエラーにはならないが、中間操作は何も実行されない
Streamインターフェースはiterableを継承していないため、拡張for文で使うとコンパイルエラー
Collectionsクラス、Optionalは完璧にする of ofNullable orElse orElseGet
ifPresentは必須
Map, reduce, limit, distinct, sorted, flatmap, peekメソッドも把握
SQLのmax()やcount()関数のように、入力操作をもとに結合操作を繰り返し実行して、単一の結果を得る操作をリダクション操作という

ストリームの生成

メソッド名	説明
default Streme<E> stream()	Collectionインターフェースで提供。このコレクションをソートとして使用して、逐次的なStreamオブジェクトを返す。
	ListやMapをストリームで扱う
	IntStream int = Arrays.stream(array, 1, 4); 要素の範囲を選択できるが終端は含まない
static<T>Stream<T>stream(T[] array)	Arrayクラスでで提供。指定された配列をソートとして使用して、逐次的なStreamオブジェクトを返す。
static IntsStream stream(int[] array)	Arrayクラスで提供。指定されたint型の配列をソースとして使用して、逐次的なStreamオブジェクトを返す。
static<T>Stream<T>of(T t)[(T… values)]	Streamインターフェースで提供。指定された単一の要素[要素]をソースとして使用して、逐次的なStreamオブジェクトを返す。
	参照型オブジェクトの集合からStreamオブジェクトを得る
static<T>Stream<T>generate(Supplier<T> s)	Streamインターフェースで提供。Supplierにより生成される要素に対する順序付けされていない、Streamオブジェクトを返す。
	limit()を使用しないと無限に生成
static<T>Stream<T>iterate(T seed, UnaryOperator<T> f)	Streamインターフェースで提供。順序付けされた無限順次Streamオブジェクトを返す。

limit()を使用しないと無限に生成
static IntStream range(int startInclusive, int endExclusive)
IntStream。startInclusive(含む)からendExclusive(含まない)の範囲の値を含む、順序付けされた順次IntStreamを返す
static IntStream rangeClosed(int startInclusive, int endInclusive)
IntStream。startInclusive(含む)からendInclusive(含む)の範囲の値を含む、順序

付けされた順次IntStreamを返す
各メソッドの戻り値は、java.util.Streamの他、IntStreamなど

テキストがいるから読み込んだ文字列をStreamオブジェクトとして扱う場合
java.io.BufferedReaderのLinesメソッド
java.nio.file.FilesクラスのLinesメソッドを使用

基本データ型のストリームの種類 BaseStreamインタフェースを基底としている
インタフェース名　　説明
Stream<T>　　順次及び並列の集約操作をサポートする汎用的なストリーム
IntStream　　順次及び並列の集約操作をサポートするint値のストリーム。short, byte, char型に使用可能
LongStream　　順次及び並列の集約操作をサポートするlong値のストリーム
DoubleStream　　順次及び並列の集約操作をサポートするdouble値のストリーム。float型に使用可能

終端操作メソッド(問題に出てきたもののみ抜粋)　Java.util.stream.Streamインターフェースで宣言
リダクションとはデータの集合を一つに要約する終端処理　リダクションを再度実行することは不可
可変リダクションとはなんらかの可変コンテナに要素を収集する操作のこと。メソッドはcollect() List, MapなどのCollectionやStringBuilderなど

メソッド名　　説明
boolean allMatch(Predicate<? super T> predicate)　　全ての要素が指定された条件に一致していた場合trueを返す。ストリームが空の場合はtrue、それ以外はfalse
boolean anyMatch(Predicate<? super T> predicate)　　いずれかの要素が指定された条件に一致していた場合trueを返す。ストリームが空の場合はtrue、それ以外はfalse
boolean noneMatch(Predicate<? super T> predicate)　　どの要素も指定された条件に一致しなければtrueを返す。ストリームが空の場合はtrue、それ以外はfalse
long count()　　要素の個数を返す
リダクション
void forEach(Consumer<? super T> action)　　各要素に対して指定されたアクションを実行する。繰り返し処理などでも用いる。
SE8よりiterableインタフェースにもforEachが追加。iterableを実装するCollectionではストリームオブジェクトを取得なしで使用可
e.g.
Arrays.asList(“A”, “B”, “c”).forEach(System.out::println); →　ABC
Mapインタフェース

にもforEachは追加されており、引数はBiConsumerになっており引数は2ついることに注意
T reduce(T identity, BinaryOperator<T> accumulator)　元の値と結合的な累積関数を使ってこのストリームの要素に対してリダクションを実行し、リデュースされた値を返す
戻り値は
java.util.Optional型になるOptionalクラスの実体は１つの値を保持しているクラス
リダクションとはデータの集合を一つに要約する終端処理　リダクションを再度実行することは不可
e.g.
```
IntStream stream = IntStream.of(1, 2, 3);

stream.reduce(1, (x, y) -> x + y));  1 + 1 + 2 + 3 = 7が結果　第一引数は初期値として加算される
```
Object[] toArray()　ストリームから配列に変換、要素を含む配列を返す。メソッドの引数はなく、戻り値はObject型
<A> A[] toArray(IntFunction<A[]> generator)　ストリームから配列に変換、要素を含む配列を返す。独自の配列型を指定可能。

Optionalクラス
JavaSE8で追加されたクラス。実体は１つの値を保持しているクラス。Optionalクラスの各メソッドは、保持している値がnullかnot nullによって処理が異なる。
Nullチェックを Optional側に任せることができる。
Optionalオブジェクトは、保持している値がnullの場合はemptyオブジェクトになるが、Optionalオブジェクト自体がnullではない。
メソッド名　説明
static <T> Optional<T> empty()　空のOptionalインスタンスを返す。このOptionalの値は存在しない。
static <T> Optional<T> of(T value)　引数で指定された非null値を含むOptionalを返す。
T get()　値が存在する場合は値を返し、それ以外はNoSuchElementExceptionをスロー。e.g. empty()など
Optional型以外を取り出すとエラー
戻り値が
OptionalIntの場合は、getAsInt()メソッドを使用する。
boolean isPresent()　存在する値がある場合はtrueを返し、それ以外の場合はfalseを返す。
void ifPresent(Consumer<? super T> consumer)　値が存在する場合は指定されたコンシューマをその値で呼び出し、それ以外の場合(中身がnullなど)は何も行わない。
T orElse(T other)　存在する場合は値を返し、それ以外の場合はOtherを返す。
e.g.

hoge.orElse(0.0) → empty()の場合は0.0が返る。この場合hoge<T>はdoubleで宣言する必要あり
T orElseGet(Supplier<? extends T> other)　値が存在する場合はその値を返し、そうでない場合はサプライヤーを呼び出し、その呼び出しの結果を返す。

e.g.
hoge.orElseGet(()->Math.random()) → empty()の場合は乱数が返る。 T get()を実装　この場合はdoubleを返す必要あり
findFirst()　最初の要素を返す。ストリームが空の場合は空のOptionalを返す。
findAny()　いずれかの要素を返す。ストリームが空の場合は空のOptionalを返す。
Optional<T> min(Comparator<? Super T> comparator)　指定されたComparatorに従って最小要素を返す。ストリームが空の場合は空のOptionalを返す。
Optional<T> max(Comparator<? Super T> comparator)　指定されたComparatorに従って最大要素を返す。ストリームが空の場合は空のOptionalを返す。
naturalOrder()で自然順序付け可能
<X extends Throwable> T orElseThrow(Supplier<? Extends X> exceptionSupplier) throws X extends Throwable
　値が存在する場合は、その含まれている値を返し、それ以外の場合は指定されたサプライヤによって作成された例外をスローする

e.g.
hoge.orElseThrow(illegalArgumentException::new) emptyの場合はillegalArgumentExceptionがスローされる。
覚える

インターフェース名	戻り値	メソッド
Stream	Optional<T>	findAny, findFirst, max, min
IntStream	OptionalDouble	average
IntStream	OptionalInt	findAny, findFirst, max, min　値の取得はgetAsInt()で取得、get()はコンパイルエラー
IntStream	int	sum()
IntStream		iterate()　e.g. IntStream.iterate(1, n -> n + 1) ←初期値を1に指定し、１づつ加算した要素を無限に用意 要limit()等で制限。

中間操作　処理の結果Streamを返す　終端操作の処理後に中間操作の呼び出しは不可

メソッド	説明
Stream<T> filter(Predicate<? Super T> predicate)	指定された条件に一致するものから構成されるストリームを返す。フィルタリング(データの取捨選択)に用いる

e.g.

.filter(n -> n % 2 == 0) →　２で割り切れる数値のみを篩い分け
Stream<T> distinct()　重複を除いた要素から構成されるストリームを返す。
Stream<T> limit(long maxSize)
maxSize以内の長さに切り詰めた結果から構成されるストリームを返す。　e.g. limit(10L)
Stream<T> skip(long n)　先頭からn個の要素を破棄した残りの要素で構成されるストリームを返す。 e.g. skip(5L)
Stream<T> sorted()　自然順序に従ってソートした結果から構成されるストリームを返す。
Stream<T> sorted(Comparator<? super T> comparator)　指定されたComparatorに従ってソートした結果から構成されるストリームを返す。e.g. sorted(Comparator.reverseOrder())　逆順
Stream<T> peek(Consumer<? super T> action)　このストリームの要素からなるストリームを返す。要素がパイプラインを通過する際にその内容を確認するデバッグとして使用

<R> Stream <R> map(Function<? super T, ? extends R> mapper
指定された関数を適用した結果から構成されるストリームを返す。ストリーム型と戻り値が違う場合もある

マッピング処理(写像)→データ集合における個々のデータを別のデータに変換した新しいデータ集合を生成
<R> Stream <R> flatMap(Function<? super T, ? extends Stream<? Extends R>> mapper)　指定された関数を適用した複数の結果から構成される1つのストリームを返す。

Listなどの入れ子構造のストリームを平坦なストリームにする際に用いる

flatMapで平坦化した後でmapを行う　e.g. List<List<Integer>>>

e.g. flatMap(e->e.stream()).map(e->e+1).forEach(System.out::println)

インターフェース名	Streamの生成	DoubleStreamの生成	IntStreamの生成	LongStreamの生成
Stream	map()	mapToDouble()	mapToInt()	mapToLong()
DoubleStream	mapToObj()	map()	mapToInt()	mapToLong()
IntStream	mapToObj()	mapToDouble()	map()	mapToLong()
LongStream	mapToObj()	mapToDouble()	mapToInt()	map()

ストリームインターフェースの方変換を行うメソッドの使用例
Stream<String>→　Stream<Integer>　→　IntStream　→　Stream<String>
Stream<String>→　IntStream　→　Stream<Integer>　→　Stream<String>

元の型　メソッド名　引数
変換後の型戻り値)
Stream<String>　map　Function<? superT, ? extends R>mapper
Stream<integer>
Stream<String>　mapToInt　ToIntFunction<? superT>mapper
IntStream　ToIntFunction<T> int applyAsInt(T value)
Stream<Integer>　mapToInt　ToIntFunction<? superT>mapper
IntStream　IntFunction<R> R apply(int value)
Stream<Integer>　map　Function<? superT, ? extends R>mapper
Stream<String>
IntStream　mapToObj　IntFunction<? extends U> mapper
Stream<String>
IntStream　boxed　なし
Stream<Integer>

boxed()メソッド
基本データ型のストリームから、Streamの各ラッパークラスの型へ変換するためのメソッド。
boxed()メソッドを利用した型変換
IntStream　→　Stream<Integer>
DoubleStream　→　Stream<Double>
LongStream　→　Stream<Long>

暗黙の型変換を行うためのメソッド　mapToDouble()と違い引数を指定することなく型変換が可能
LongStream asLongStream()　IntStreamインターフェースで提供　要素をlongに変換した結果から構成されるLongStreamを返す
DoubleStream asDoubleStream()　IntStreamインターフェースで提供　要素をdoubleに変換した結果から構成されるDoubleStreamを返す
DoubleStream asDoubleStream()　LongStreamインターフェースで提供　要素をdoubleに変換した結果から構成されるDoubleStreamを返す

collect()メソッド　可変リダクション
ストリームから要素をまとめて1つのオブジェクトを取得することができる。
構文
<R, A> R collect(Collector<? super T, A, R> collector)　引数はjava.util.stream.Collector型。CollectorインタフェースはSE8から追加された、通常のインターフェース。

Collectorsクラスの主なメソッド
メソッド名
説明
static <T> Collector<T, ?, List<T>> toList()
Listに蓄積するCollectorを返す
static Collector<CharSequence, ?, String>joining()
入力要素(文字列)を検出順に連結して1つのStringにするCollectorを返す。単一の文字列を返すのでprintln出力可
static Collector<CharSequence, ?, String>joining(CharSequencedelimiter)
入力要素を検出順に指定された区切り文字で区切りながら連結するCollectorを返す
static <T> Collector<T, ?, Integer> summingInt(ToIntFunction<? super T> mapper)
int値関数を適用した結果の合計を生成するCollectorを返す。要素がない場合、結果は0になる
static <T> Collector<T, ?, Double> averagingInt(ToIntFunction<? super T> mapper)　　int値関数を適用した結果の算術平均を生成するCollectorを返す。要素がない場合、結果は0になる

averagingXXXメソッドの戻り値は全てDouble型になる
static <T> Collector<T, ? , Set<T>> toSet()
Setに蓄積するCollectorを返す。

static <T, K, U> Collector<T, ?, Map<K, U>> toMap(Function<? super T, ? extends K>key, Function<? super T, ? extends U> value
Mapに蓄積するCollectorを返す。第一引数にキー、第二引数に値を指定　オーバーロードされている
マップ先のキーが重複する場合、実行時にIllegalStateExceptionがスローされる。
第三引数を使用するとマッピングしてマージした結果を返す

static <T, K,> Collector<T, ?, Map<K, List<T>>> groupingBy(Function<? super T, ? extends K> classifier
指定した関数に従って要素をグループ化し、結果をMapに格納して返すCollectorを返す。オーバーロードされている

static <T> Collector<T, ?, Map<Boolean, List<T>>> partitioningBy(Predicate<? super T> predicate
指定した関数に従って要素をtrueもしくはfalseでグループ化し、結果をMapに格納して返すCollectorを返す。

static <T, U, A, R> Collector<T, ?, R> mapping(Function<? super T, ? extends U>mapper, Collector<? super U, A, R> downstream)
マッピングを行い、マップ後に指定された他のCollectorを適用し、その結果をMapに格納して返すCollectorを返す
第一引数に要素に対して行いたい処理、第二引数はマップ後に行いたい処理を指定。

static <T> Collector<T, ?, Optional<T>> maxBy(Comparator<? super T> comparator)
指定されたComparatorに従ってOptional<T>として記述された最大要素を生成するCollectorを返す。
戻り値がOptional型なので変数宣言でOptional<T>とする。値の取得にはget()を使用

例外処理

Checked例外　　データベースなどJava実行環境以外の環境が原因、例外処理が必須
unchecked例外　　実行時例外、メモリ不足、プログラムの例外処理では復旧できない、例外処理は任意

例外クラスは３種類に分類
Error(unchecked例外)　：例外処理は任意
RuntimeException(unchecked例外)　：例外処理は任意
RuntimeException以外のExceptionのサブクラス(checked例外)　：例外処理は必須
ユーザー定義例外は一般的にはpublicで、Exceptionクラスを継承したクラスとして定義する。厳密なルールはない
Exceptionクラスを継承することでThrowableクラスが提供する以下のメソッドを使用可能
Void printStackTrace()　　エラートレースを出力。エラーを追跡し発生箇所を特定する
String getMessage()　　エラーメッセージを取得する

主な例外クラス

カテゴリ	クラス名	説明
Error	AssertionError	assert文を使用している際に、boolean式でfalseが帰ると発生
Unchecked例外	StackOverflowError	アプリケーションでの再帰の回数が多すぎる場合、無限ループ時に発生
	NoClassDefFoundError	読み込もうとしたクラスファイルが見つからない場合に発生
RuntimeException	ArrayIndexOutOfBoundsException	不正なインデックスで要素にアクセスしようとした場合に発生
Unchecked例外	ArrayStoreException	不正な型のオブジェクトを配列に格納した場合に発生
	ArithmeticException	算術計算で例外的条件が発した場合にスロー、ゼロ除算など
	ClassCastException	参照変数において間違ったキャストを行なった場合に発生
	IllegalStateException	メソッドの呼び出しが正しくない状態で行われた場合に発生

DateTimeException 日付/時間の計算時に誤った処理を行なった場合に発生
MissingResourceException リソースが見つからない場合に発生
ArithmeticException 整数をゼロで除算した場合に発生
NullPointerException nullが代入されている参照変数に対して、メソッド呼び出しを行なった場合に発生
NumberFormatException 整数を表さない文字列を整数に変換しようとした場合に発生
RuntimeException以外 IOException 入出力を行う場合に発生
checked例外 FileNotFouncdException ファイル入出力において、目的のファイルがなかった場合に発生
例外処理必須 ParseException 解析中に予想外のエラーがあった場合に発生
SQLException データベース・アクセス時にエラーがあった場合に発生

例外のキャッチ

構文として以下が記述可能 tryのみの使用はコンパイルエラー　各ブロック内にtry-catch-finallyを入れ子にすることも可能
・try - catch
・try - finally
・try - catch - finally
Finallyブロックは、必ず行いたい処理があれば利用。リソースの解放にも用いる。
e.g. close処理
Catchを複数記述する際はサブクラス側から記述する。スーパークラスから記述するとコンパイルエラー
catch(home | foo)と記述することで複数の例外をまとめてキャッチするマルチキャッチがSE7から利用可能(継承関係にある例外クラスは記述できない、キャッチした参照変数は暗黙的にfinalになる)
e.g. catch(ArithmeticException | FileNotFoundException e){ e = hoge ;}　キャッチした参照変数は暗黙的にfinalになるのでコンパイルエラー 通常のキャッチでは代入可能
throwsとthrow

Throws指定された例外クラスのオブジェクトがメソッド内で発生した場合、呼び出し元に転送される。カンマ区切りで複数指定可能
Checked例外は呼び出し元に転送するには必ずthrowsを記述し呼び出し元で例外処理をする。
UnChecked例外は例外処理をしなくても呼び出し元に転送される仕組みになっており

、main()側でキャッチしなければ、Java実行環境に例外がスローされ、例外メッセージが表示される。

Throwを記述すると、例外クラスや独自定義例外を任意の場所でスロー可能

Throwsキーワードが使用されているメソッドをオーバーライドする際には以下のルールを遵守する必要がある。
・サブクラスのメソッドでは、スーパークラスのメソッドがスローする例外クラスと同じか、その例外クラスのサブクラスとする
・RuntimeExceptionは、スーパークラスのメソッドに関係なくスローできる
・スーパークラスのメソッドにthrowsがあっても、サブクラス側でthrowsを記述しないことは可能
オーバーライドはメソッド名、引数リストが１全く同じ、戻り値の型は同じもしくはその型のサブクラスのみ、アクセス修飾子は同じかそれよりも範囲の広いもの

rethrow

Catchした例外にエラーメッセージを追記したり、異なる例外クラスに変更したりした後に、再度スローすることをrethrowと呼ぶ
再スロ～していく過程で元の原因となった例外以外の例外オブジェクトを格納している場合がある。このような例外オブジェクトを取得するには以下のメソッドを使用する
Throwable getCause()　　格納されている例外オブジェクトを個別に取り出すことができる

Try-with-resources

SE7よりtryブロック終了時に暗黙的にclose()メソッドを呼び出すことが可能 tryのみの使用も可能(catch、finally省略可能)
構文 try(リソース1; リソース2; ……..n)　　実際にクロースされる順はリソース2→リソース１の順になり宣言と逆順
記述できるものは、java.lang.AutoCloseable または java.io.Closeableインタフェースの実装クラス　e.g. java.ioパッケージ java.sqlパッケージなど

インタフェース	メソッド名	
java.lang.AutoCloseable	void close() throws Exception	このリソースを閉じ、ベースとなるリソースを全て解放する。
java.io.Closeable	void close() throws IOException	このストリームを閉じて、それに関連する全てのシステムリソースを解放する。

Throwableの拡張

close()メソッドでスローされた例外(抑制された例外)を受け取るにはSE7で追加されたThrowableクラスの機能拡張を利用する。

Throwableに追加されたメソッド
メソッド名　　説明
final void addSuppressed(Throwable exception)　　この例外を提供する目的で抑制された例外に、指定された例外を追加する。
final Throwable{} getSuppressed()　　try-with-resources文によって抑制された例外を全て含む配列を返す。
e.g. Throwable[] error = e.getSuppressed() ;　　e.getMessage();

アサーション
プログラムが前提としている条件をチェックし、プログラムの正しい動作を保証するための機能。エラーを表示させることができ、バグの検出が有効
構文 assert boolean式;　　assert boolean式:メッセージ;
e.g. assert(hoge > 0):”hoge += hoge”;　　hogeが０より大きいかをチェック。falseが返るとAssertionErrorがJava実行環境よりスロー
AssertionErrorがスローされないように修正することが目的なので、
AssertionErrorのための例外処理コードは書かない
SE5以降ではコンパイル時点ではアサーション機能は有効になっているためコンパイルは通常通り行われるが、実行時は無効になっている。有効にするには以下のオプションを追記
javac hoge.java　　コンパイル時点ではアサーション機能は有効なためここでは通常通りコンパイルを行う
java hoge　　オプションをつけないとアサーション機能は無効になる
java -ea hoge　　-eaをつけることでアサーション機能は有効になる
java -da:foo-ea bar　　-daを使用することで明示的に無効化可能　例ではfooは無効、barは有効にしている

アサーションの実行を正常動作の一部としてコーディングはしない。
プログラムにバグを残さないための1つの手法なので、リリースする段階では、必ずAssertionErrorがスローされないようにする。
アサーションは以下の条件を検証する際に用いる
事前条件
メソッドが呼び出された時にtrueであるべき条件。publicメソッド内の引数チェックに使用することは非推奨。publicメソッドは引数をチェックする必要がもともとあるため
Privateメソッドの引数などの検証に利用
事後条件
メソッドが正常に実行された後にtrueであるべき条件。事後条件のアサーションはpublicメソッド内、その他のメソッド内で使用できます。
不変条件
常にtrueであるべき条件。プログラムが正しい動作をするために、常に満たしていなけれ亜ならない条件を検証する。

問題ではコマンドが与えられて、選択肢を選ぶ問題があるので-eaが付いているかいないか注意する

日付・時刻API

日付・時刻。日付/時刻のためのクラスが個別に提供されている。各クラスは不変オブジェクト(イミュータブル)となるため、マルチスレッド環境下でも安全に使用できる。

パッケージ名	説明
java.time	日付/時刻APIのメインとなるパッケージ
java.time.temporal	日付/時刻APIの追加機能を提供するパッケージ　日時を表現するクラスが実装しているのは、TemporalAccesorとそのサブインタフェースであるTemporal TemporalAccesorは読み取り専用のメソッドを宣言しており、Temporalは読み取り、書き込み両方のメソッドを宣言している。
java.time.format	日付と時刻の書式化を行うパッケージ

java.timeパッケージには列挙型として曜日(DayOfWeek)とMonthがある

java.timeで提供されている主なクラス　ISO8601は日付と時刻の表記に関する国際標準規格、APIはこれをベースにしている。協定世界時(UTC)とグレゴリオ暦を基準として日付時刻時差を規定

クラス名	説明
LocalDate	ISO8601暦体系におけるタイムゾーンのない日付 e.g. 2007-12-03
LocalTime	ISO8601暦体系におけるタイムゾーンのない時刻 e.g.10:15:30
LocalDateTime	ISO8601暦体系におけるタイムゾーンのない日付/時刻 e.g. 2007-12-03T10:15:30
OffsetTime	ISO8601暦体系におけるUTC/GMTからのオフセット付きの時刻 e.g. 10:15:30+01:00
OffsetDateTime	ISO8601暦体系におけるUTC/GMTからのオフセット付きの日時 e.g. 2007-12-03T10:15:30+01:00
ZoneOffset	UTC/GMTからのタイムゾーン・オフセット e.g. +02:00
ZonedDateTime	ISO8601暦体系におけるタイムゾーン付きの日付/時刻 e.g. 2007-12-03T10:15:30+01:00 Europe/Paris
ZoneId	タイムゾーンを特定するID e.g. Asia/Tokyo　JSTは日本標準時Asia/Tokyo　PSTは米国西海岸標準時America/Los_Angels
ISO_DATE 2011-12-03	ISO_TIME 10:15:11　ISO_DATE_TIME 2012-12-32T12:32:12+01:33

各クラスはstaticメソッドであるnow()を使用して時刻等を取得している。表記例は以下のとおり

項目	ISO8601表記	例

日付　　YYYY-MM-DD　　2016-02-24
月や日が１桁の場合は０で埋める
時刻　　hh:mm:ss
11:39:02.441　一桁の場合は０で埋める。ナノ秒までサポートしているため
hh:mm:ss.s
日付・時刻　　YYYY-MM-DDThh:mm:ss
2016-02-24T11:39:02.44　　日付と時刻の間にTを記述
時刻＋時差　　hh::mm:ss±hh:mm
11:39:02.441+09:00　　日本はUTCより９時間進んでいるため+09:00となる
日付・時刻＋時差　　YYYY-MM-DDThh:mm:ss±hh:mm
2016-02-24T11:39:02.441+09:00

各クラスのコンストラクタはprivate修飾子が付与されているため、newによるインスタンス化はできないので、staticメソッドを使用してオブジェクトを作成。
now() of() parse()各クラスでオーバーロードされている
メソッド名　　説明
static LocalDate now()　　現在の日付からLocalDateオブジェクトを取得する

Static LocalDate of(int year, int month, int dayOfMonth)　　年月日からLocalDateオブジェクトを取得する、月日は一桁の場合は０を付与していなくても良い
日付の指定で値(存在しない月など)が不正な場合はコンパイルは成功するが、DatetimeExceptionが発生
Static LocalDate of(int year, Month month, int dayOfMonth)　　年月日からLocalDateオブジェクトを取得する、月日は一桁の場合は０を付与していなくても良い
第二引数java.time.month は12ヶ月を表す列挙型(1～12月存在)　　日付の指定で値(存在しない月など)が不正な場合はコンパイルは成功するが、DatetimeExceptionが発生

e.g. LocalDate.of(2016, Month.FEBRUARY, 24)
Month列挙型はintと比較できずコンパイルエラーとなる。
列挙定数のintはordinal()を使って取り出すがMonthれ巨型はgetValue()を使う
ordinal()だと0～11が帰るがgetValue()は1～12が返り月に対応している
Static LocalDate parse(CharSequence text)　　2007-12-03などのテキスト文字列からLocalDateオブジェクトを取得　ハイフンやコロンを省略できない
月日が一桁の場合は０を付与ないとコンパイルは通るが実行時エラー DateTimeParseExceptionが発生

LocalDateオブジェクトが保持する年月日はgetXXX()メソッドで取り出している。
getYear, getMonth, getMonthValue, getDayOfMonth()
getMonth()は戻り値がMonth型

日付・時刻のフォーマット
Java.time.format.Date.TimeFormatterクラスが提供　不変クラス、スレッドセーフ
定義済みフォーマッタを参照するには、ofLocalizeDate, ofLocalizeDateTimeがある。
BASIC_ISO_DATE　基本的な日付
ISO_DATE　オフセットあり、無しのISO日付
ISO_LOCAL_DATE　ISOローカル日付
ISO_OFSET_TIME　オフセット付きの時間

e.g. DateTImeFormatter formatter = DateTimeFormatter.BASIC_ISO_DATE;
println(formatter.format(LocalDateTime.now());
DateTImeFormatter formatter = DateTimeFormatter.ofLocalizeDateTime(Format.Style.MEDIUM);

ofPattern(String pattern, Local locale)　第一引数のみの場合はデフォルトである日本のロケールに従う、第二引数をofPattern(“MMMM”,Local.US)とすると米国ロケールを使用
パターンでMは月を表すが、MMMMとすると地域に従った月を表す　この後にformat()メソッドで整形する
LocalDate plusDays(long daysToAdd)　指定された日数を加算した、このLocalDateのコピーを返す。plusXXXXはyears,weeks,monthsがある
minusXXXXメソッドも提供されている。時刻を扱うクラスでは　plusHourやplusSecondなどもある
日付/時刻APIは不変オブジェクトのため、各メソッドを実行すると処理後の日付時刻を保持した新しいオブジェクトが返り、処理元のオブジェクトは変化しない
メソッドチェイン可能　e.g. dateTime = dateTime.minusDays(1).minusHours(7).minusSeconds(15);
時刻の加減算を行うメソッドは、時刻を扱うクラスのみ使用可能。LocalDateクラスでは時刻を扱わないためコンパイルエラーになる

ZonedDateTimeクラス
タイムゾーンは共通の標準時を使う地域や区分。標準時はUTCとの差で示す。日本はUTC+9。タイムゾーンとして韓国やインドネシアなどもある
タイムゾーンを含んだ日付・時刻クラスはZonedDateTimeとして提供されている
タイムゾーンである地域を表すクラスといて、java.time.ZoneIdを使用。ZonedIdオブジェクトの生成は、文字列で表現されてあタイムゾーンIDを元に行う。
JSTは日本標準時Asia/Tokyo　PSTは米国西海岸標準時America/Los_Angels

Static ZoneId systemDefault()　システム・デフォルト・タイムゾーンを

表すZoneidオブジェクトを取得する
static ZoneId of(String zoneId)　　指定されたIDからZoneIdオブジェクトを取得する。e.g. ZonedId.of(“America/Los_Angeles”)
Static ZonedDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond, ZoneId zone)
年月日時分秒ナノ秒、タイムゾーンからZonedDateTimeオブジェクトを取得する
static ZonedDateTime of(LocalDateTime localDateTime, ZoneId zone)
LocalDateTimeとタイムゾーンを元にZonedDateTimeオブジェクトを取得する

FormatStyleクラス　列挙値として提供されているスタイルを指定してフォーマットを行う。覚える
FormatStyle.FULL　　2016年2月20日10時30分45秒JST　　ゾーン情報を含む(この例ではJST)
FormatStyle.LONG　　2016/02/20 10:30:45 JST　　ゾーン情報を含む(この例ではJST)
FormatStyle.MEDIUM 2016/02/20 10:30:45
FormatStyle.SHORT　16/02/20 10:30
LocalDateTImeオブジェクトはゾーン及び時差は保持していないため、FULLやLONGを使用すると実行時にDateTimeExceptionが発生。MEDIUMとSHORTは問題ない。
また、日付、時刻に合わせてofLocalizedXXX()メソッドが提供されている。

OffsetDateTimeクラス
時差を含んだ日付・時刻クラスはOffsetDateTimeとして提供されている。of()メソッドでOffsetDateTimeオブジェクトの生成が可能。
ZoneOffsetは時差を表す。
ZoneOffsetオブジェクトの取得にはof()メソッドを使用し、引数はString型で、+hh:mmや-hh:mmといったフォーマットで時差を指定します。e.g. ZoneOffset.of(“+09:00”)
Static OffsetDateTime of(int year, int month, int dayOfMonth, int hour, int minute, int second, int nanoOfSecond, ZoneOffset offset)
年月日時分秒ナノ秒、タイムゾーンからOffsetDateTimeオブジェクトを取得する
static ZonedDateTime of(LocalDateTime localDateTime, ZoneId zone)
LocalDateTimeとタイムゾーンを元にZonedDateTimeオブジェクトを取得する
SummerTIme
Gold試験では米国を例にした夏時間の問題が出題される割合が高い。以下は全て米国を例にしている。
米国は3月に１時間進めて11月に戻す。2016年度の米国の夏時間の開始、終了は以下のとおり。
2016年度の夏時間開始日時：2016年3月13日(日)2時0分 EST(East Standard Time: 東部標準時)

2016年度の夏時間終了日時：2016年11月6日(日)2時0分 EDT(East Daylight Time: 東部夏時間)

ZonedDateTimeはタイムゾーンを含んだ日付・時刻クラスなので夏時間が考慮されている。なので上記の夏時間切り替えの日にちの加減算処理は注意が必要
LocalTime Time = LocalTime.of(2,00) ←のように時間を2時に設定しても、夏時間開始日時と被っていた場合は、出力した際に暗黙で１時間進めた結果が返される
2016年3月13日(日)1時30分に設定し、１時間加算した場合、夏時間の＋1時間と合わせて合計で2時間プラスされる。時差を表示した場合はマイナス1時間される。
夏時間の終了間際で処理は上記の真逆のことが発生する。

日や時間の間隔
間隔を扱うクラス
Period　　年月日単位で間隔を扱う。
Duration　　時間単位で間隔を扱う。

Periodクラスの主なメソッド
メソッド名　　説明
static Period between(LocalDate start, LocalDate end)　　2つの日付間の年数、月数、及び日数で構成されるPeriodを取得する。
e.g. s=LocalDate.of(2016,Month.JANUARY,1)
e=LocalDate.of(2017,Month.MARCH,5)
　　Period period = Period.between(s, e); println(period);　　→ピリオドクラスのtoString()メソッドでP1Y2M4Dが返される　Pはピリオド、Yは年数、Mは月数、Dは日数

int getXXX()　　XXXはYears, Months, Daysがあり、それぞれ年数、月数、日数を取得する
Static Period ofXXX(int XXX)　　XXXはYears, Months, Weeks, Daysがあり,それぞれ年数、月数、週数、日数を表すPeriodを取得する。メソッドチェイン可能
e.g.
Period.ofYears(2)　→ P2Yが返る　Period.ofWeeks(3)　→ P21Dが返る　週の表示がないため日数で返す
e.g.
Period.ofYears(1).ofMonths(1) → P1Mとなる。これは年数を取得した後に月数取得で上書きしている
static Period of(int years, int months, int days)　　年数、月数、日数を表すPeriodを取得する　e.g.Period.of(0, 10, 50)　→ P10M50Dが返る

Periodクラスの加減算用のメソッド　minusXXX()も同様
Period plusXXX(long XXX)　　XXXはYears, Months, Daysを指定。指定された年数・月数・日数を加算して、この期間のコピーを返す
Period plus(TemporalAmount amountToAdd)　　年数・月数・日数をに対して

、別々に加算を行い、この期間のコピーを返す。

LocalDateTime、DateTimeも加減算のメソッド(plus(), minus())を持つ。戻り値は各クラス名となる
引数はTemporalAmountインタフェース。PeriodクラスはTemporalAmount型を持つので指定可能　e.g. Period period = Period.ofMonths(1);
println(dateTime.plus(period));
Timeクラスは日付情報を持っていないため、time.plus(period)はコンパイルは成功するが、UnsupportedTemporalTypeExceptionがスロー

Durationクラス
時間単位で間隔を扱う、秒を表すlong値とナノ秒を表すint値(0と999,999,999の間になる)を保持。

Durationオブジェクトの取得はofXXX()を使用　PTはPeriod of Time
Duration.ofXXX(int XXX)　XXXにはDays, Hours, Minutes, Seconds, Millis, Nanosを指定可能 PT24H, PT1H, PT1M, PT1S, PT0.001S, PT0.000000001Sがそれぞれ返る

秒数及びナノ秒数で構成されるDurationオブジェクトを作成することができる。

static Duration ofSeconds(long seconds)
static Duration ofSeconds(long seconds, long nanoAdjustment)　第一引数に秒数、第二引数にナノ秒を指定する。
e.g. Duration.ofSeconds(12, 300_000_000);→12.3秒を表す　実際はPT12.3Sと出力される
static Duration of(long amount, TemporalUnit unit)　第二引数で指定された単位での間隔表すDurationを作成。TemporalUnitは以下を参照

e.g. Duration.of(12, ChronoUnit.SECONDS)　→　PT12S　of(60,SECOUNDS)はPT1Mになる
TemporalUnitインタフェースの実装として、java.time.temporalパッケージにChronoUnit列挙型が提供されている

ChronoUnit列挙型

定数名	説明
DAYS	1日
HOURS	1時間
MINUTES	1分
SECONDS	1秒
MILLIS	1ms
NANOS	1ナノ秒

minus(), plus()で加減算可能 truncatedTo()で時間を切り捨てたLocalDateTimeを返す
以下のメソッドを使用することでも日数の計算が可能
LocalDate start = LocalDate.of(2016, Month.APRIL, 1);
LocalDate end = LocalDate.of(2016, Month.APRIL, 10);
Temporal tmp = ChronoUnit.Days.addTo(start, 5);　　五日後を取得
long t = ChronoUnit.Days.betweens(start, end);　　startからendまでの日数を取得

Durationクラスの加減算はPeriodクラスとメソッド名・使い方は同じなのでそちらを参照、例外の出方も同じ

まとめ
Period　　年月日単位で間隔を扱う。LocalDate、LocalDateTime、ZonedDateTimeで利用可能。LocalTImeは使用不可→UnsupportedTemporalTypeExceptionがスロー
Duration　　時間単位で間隔を扱う。　LocalTime、LocalDateTime、ZonedDateTimeで利用可能。LocalDateは使用不可→UnsupportedTemporalTypeExceptionがスロー

Instantクラス
単一の時点を扱うクラス。UTCでの1970年１月１日0時0分0秒(1970-01-01T00:00:00Z→以降エポックと呼ぶ)から推定されるエポック秒
エポック秒を表すlong値とナノ秒を表すint値(0と999,999,999の間になる)を保持。
エポック後のInstantは正の値を持ち、エポック前のInstantは負の値を持つ。
Instantオブジェクトの作成メソッド
Static Instant now()　　システムクロックから現在のInstantを取得する。
static Instant ofEpochSecond(long epochSecond)　エポックから秒数を使用してInstantを取得する。
static Instant ofEpochMilli(long epochMilli)　　エポックからのミリ秒数を使用してInstantを取得する。
Static Instant ofEpochMilli(long epochSecond, long nanoAdjustment)　　エポックからの秒数と秒のナノ秒を使用してInstantを取得する。
java.util.Dateクラスやjava.util.CalendarクラスにはSE8よりInstantオブジェクトに変換するtoInstant()メソッドが提供されている
default Instant toInstant(ZoneOffset offset)　LocalDateTimeクラスのスーパーインターフェースであるChronoLocalDateTimedで定義されているデフォルトメソッド
　　LocalDateTimeクラスは、時差・タイムゾーンを持たないためInstantオブジェクトの変換には時差の情報が必要
　　そのため、toInstant()メソッドの引数に時差であるZoneOffsetオブジェクトを指定する。
　　e.g.LocalDateTime ldt =

LocalDateTime.of(2016,12,31,15,30);　　ldt=toInstant();←コンパイルエラー

ldt=toInstant(ZoneOffset.of(“”+09:00”);はコンパイル成功
e.g.Instant instant1 = Instant.ofEpochSecond(0);→1970-01-01T00:00:00Zが出力

Instantクラスの主なメソッド

メソッド名	説明
Instant plusSeconds(long secondsToAdd)	指定された秒を加算したものを返す
Instant plus(long amountToAdd, TemporalUnit unit)	第一引数に指定された量(第二引数に単位を指定)を加算したものを返す。

TemporalUnitはChronoUnit列挙型が使用可能
Instantクラスのplus()やminus()メソッドで指定可能なChronoUnit列挙型

フィールド名	説明
NANOS	plusNanos(long)と同等
MICROS	1000倍されたplusNanos(long)と同等
MILLIS	1000000倍されたplusNanos(long)と同等
SECONDS	plusSeconds(long)と同等　e.g.plus(10, ChronoUnit.SECONDS) 10秒加算
MINUTES	60倍されたplusSecods(long)と同等 e.g.plus(10, ChronoUnit.MINUTES) 10分加算
HOURS	3600倍されたplusSecods(long)と同等
HALF_DAYS	43200(3600×12時間)倍されたplusSeconds(long)と同等
DAYS	86400(3600×24時間)倍されたplusSeconds(long)と同等

上記以外を指定した場合は、実行時にUnsupportedTemporalTypeExceptionがスロー
e.g.e.g.plus(10, ChronoUnit.YEARS)←コンパイルは成功するが、
UnsupportedTemporalTypeExceptionがスロー

日付・時刻クラスはprivateなためnewできない

Fileクラス

java.ioパッケージでファイルとの読み書きなどの入出力処理を行う。
Java.io.Fileクラスはディスクに保存されているファイルやディレクトリをオブジェクトとして表現するクラス。ファイル名、パス名、ファイルの有無、特定ファイルに関する情報を獲得するために利用
ファイルやディレクトリそのものを取り扱うが、ファイルの読み書きはできない
ファイルの所有者やセキュリティ属性も取得できない
シンボリックリンクなどのUNIX系のファイルシステム固有の機能も利用できない

Fileクラスの主なコンストラクタとメソッド

コンストラクタ名	説明
File(String pathname)	指定されたパス名文字列を抽象パス名に変換して、Fileオブジェクトを作成

File(String parent, String child)　　親パス名文字列及び子パス名文字列からFileオブジェクトを作成
File(File parent, String child)　　親抽象パス名及び子パス名文字列からFileオブジェクトを作成
Fileオブジェクトはファイルもしくはディレクトリのパス名を表すだけなので、実際に存在するかどうかは関係ない。それらはexists()で確認する

メソッド名	説明
boolean createNewFile() throws IOException	この抽象パス名が示すからの新しいファイルを作成する。
File[] listFiles()	この抽象パス名が示すディレクトリ内のファイル、ディレクトリをFile配列として返す
boolean IsFile()	この抽象パス名が示すファイルが普通のファイルかどうかを判定する
boolean IsDirectory()	この抽象パス名が示すファイルが普通のディレクトリかどうかを判定する
boolean delete()	この抽象パス名が示すファイルまたはディレクトリを削除する
boolean mkdir()	この抽象パス名が示すディレクトリを作成
boolean mkdirs()	この抽象パス名が示すディレクトリを作成。必要な存在していない親ディレクトリがあれば一緒に生成する
boolean renameTo(File dest)	この抽象パス名が示すファイル名を変更する
String getAbsolutePath()	この抽象パス名の絶対パス名文字列を返す
String getName()	この抽象パス名が示すファイルまたはディレクトリの名前を返す

Windowsベースのファイルセパレータを明示的に使用する場合は、
C:¥¥hoge¥¥foo¥¥bar.txtのようにエスケープシーケンスを用いる
システム依存のファイルセパレータを取得するには、システムプロパティから取得可能　System.getProperty(“システムプロパティ名”);
システムプロパティ名　　SE7からSystemクラスにline.Separetor()メソッドが追加されている

	プロパティ名	Windows	Mac
改行コード	line.separetor	¥r¥n	¥n
ファイルセパレータ	file.separetor	¥¥	/
パスセパレータ	path.separetor;		:

ストリーム　データの送受信を連続的に行うもの
java.ioパッケージの入出力ストリームで扱う様々なクラスを組み合わせて利用する

ために採用されているデザイン・パターンとしてDecoratorがある。
バイトストリームはbyte単位で読み書きをするストリーム、キャラクタストリームはchar単位でデータを読み書きするストリーム。

	バイトストリーム	キャラクタストリーム
出力ストリーム	OutPutStream	Writter
入力ストリーム	InPutStream	Reader

主なバイトストリーム
FileInputStream　ファイルからbyte単位の読み込みを行うストリーム
FileOutputStream　ファイルからbyte単位の書き出しを行うストリーム
DateInputStream　基本データ型のデータを読み込めるストリーム
DateOutputStream　基本データ型のデータを書き出せるストリーム
主なキャラクタストリーム
FileReader　ファイルからchar単位の読み込みを行うストリーム
FileWritter　ファイルからchar単位の書き出しを行うストリーム
BufferedReader　char単位で文字、配列、行をバッファリングしながら読み込むストリーム
BufferedWritter　char単位で文字、配列、行をバッファリングしながら書き出すストリーム

FileInputStreamとFileOutputStreamクラス
Byte単位でファイルの入出力を行うクラス。Fileオブジェクトやファイルパス文字列を元にファイル内に記述されたデータをbyte単位で入出力する

FileInputStreamとFileOutputStreamのコンストラクトとメソッド

コンストラクタ名	説明
FileInputStream(File file) throws FileNotFoundException	引数で指定されたFileオブジェクトからデータを読み込むための入力ストリームを作成 e.g.new FileInputStream(new File(“hoge/foo.txt”)) 指定したファイルが存在しない場合はFileNotFoundExceptionがスロー
FileInputStream(String name) throws FileNotFoundException	引数で指定されたファイルからからデータを読み込むための入力ストリームを作成
FileOutputStream(File file) throws FileNotFoundException	引数で指定されたFileオブジェクトへデータを書き出すための出力ストリームを作成 存在しなければ新規作成
FileOutputStream(String name) throws FileNotFoundException	引数で指定されたファイルへデータを書き出すための出力ストリームを作成

FileOutPutStreamコンストラクタは実行するたびに指定されたファイルの先頭から書き込むので、すでにファイルが存在していたり、追記したい場合は以下のコンストラクタを使用
FileOutputStream(File file, boolean append) throws FileNotFoundException
boolean値を第二引数で指定。trueなら追記。falseすると先頭書き込みになる
FileOutputStream(String name, boolean append) throws FileNotFoundException

メソッド名	説明

Int read() throws IOException　　入力ストリームからバイトデータを読み込む。ファイルの終わりに達すると－1を返す。FileInputStreamクラスのメソッド
Void close() throws IOException　　このストリームを閉じる　FileInputStreamクラスのメソッド
Void write(int b) throws IOException　　引数で指定されたバイトデータをファイル出力ストリームに書き出す　FileOutputStreamクラスのメソッド
　　文字列を指定すると文字コード(数値)の並びで出力
Void write(byte[] b) throws IOException　引数で指定されたバイト配列をファイル出力ストリームに書き出す　FileOutputStreamクラスのメソッド
Void close() throws IOException　　このストリームを閉じる　FileOutputStreamクラスのメソッド

DateInputStreamとDateOutputStreamクラス
基本データ型及びString型のデータを読み書きできるストリーム。単体では使用できないので、他のストリームと連結して使用する必要がある。
具体的には他のストリームをコンストラクタの引数に指定して生成する必要がある。

DateInputStreamとDateOutputStreamクラスの主なコンストラクタとメソッド
コンストラクタ
DataInPutStream(InputStream in)　　引数で指定されたInputStreamオブジェクトを使用するDateInputStreamを作成
DataOutPutStream(OutputStream out)　　引数で指定されたOutputStreamオブジェクトを使用するDateOutputStreamを作成

メソッド名
Final int readInt() throws IOException　　4バイトの入力データを読み込む。入力の途中で予想外のファイルの終了または予想外のストリームの終了があった場合はEOFExceptionがスロー
final String readUTF() throws IOException　　UTF-8形式でエンコードされた文字列を読み込む。予想外のファイルの終了または予想外のストリーム終了があった場合はEOFExceptionがスロー
final void writeByte(int v) throws IOException　　byte値を1バイト値として出力ストリームに書き出す。例外がスローされない場合、バイト数は1増加する。
final void writeInt(int v) throws IOException　　int値を4バイト値として出力ストリームに書き出す。例外がスローされない場合、バイト数は4増加する
final void writeUTF(String str) throws IOException　　引数で指定されたデータをUTF-8エンコーディングを使った形式にして出力ストリームに書き出す。
e.g. DataOutPutStream dos = new DataOutPutStream((new FileOutPutStream(“hoge/foo.txt”);

```
    DataInPutStream dis = new DataInPutStream((new FileInPutStream(“
hoge/foo.txt”)){
          dos.writeInt(100); dos.writeUTF(“田中”);
println(dis.readInt()+dis.readUTF()
    );
```

UTF8でエンコーディング　されているため、UTF8以外のエンコーディングをデフォルトとしているテキストエディタでそのまま開いた場合は文字化けする

FileReaderとFileWritter

Java.io

キャラクターストリームに属するクラス。char単位で読み書きを行い、入出力データの文字コードは自動的に変換。

Java言語は一文字をUnicode(16ビットデータ)として扱っている。キャラクターストリームを使用するとJavaプログラムからファイルが保存されているOSの文字コードを意識することなく入出力を行える

FileReaderとFileWiterのメソッドとコンストラクタ

コンストラクタ名	説明
FileReader(File file) throws FileNotFounndException	引数で指定されたFileオブジェクトからデータを読み込むための、FileReaderオブジェクトを作成
FileReader(String fileName) throws FileNotFounndException	引数で指定されたファイルからデータを読み込むための、FileReaderオブジェクトを作成
FileWriter(File file) throws FileNotFounndException	引数で指定されたFileオブジェクトへデータを書き出すための、FileWriterオブジェクトを作成
FileWriter(String fileName) throws FileNotFounndException	引数で指定されたファイルへデータを書き出すための、FileWriterオブジェクトを作成

FileWriterクラスはFileOutputStreamと同様に指定したディレクトリ以下にファイルが存在しない場合は新規作成する。

ileWriterクラスはFileOutputStreamクラスのコンストラクタと同様に実行するたびに指定されたファイルの先頭から書き込む。追記したい場合は以下のコンストラクタを使用

FileWriter(File file, boolean append) throws IOException

boolean値を第二引数で指定。trueなら追記。falseすると先頭書き込みになる

FileWriter(String name, boolean append) throws IOException

メソッド名	説明
Int read() throws IOException	ストリームから単一文字を読み込む。ファイルの終わりに達すると－1を返す(親クラスのInoutStreamReaderクラスのメソッド)　FileReaderクラス
void write(String str) throws IOException	引数で指定された文字列を書き出す(親クラスのWriterクラスのメソッド FileWriterクラス
Int flush() throws IOException	目的の送信地に、直ちに文字

を書き出す(親クラスのOutPutStreamクラスのメソッド)
FileWriterクラス

```
e.g. FileWriter fw = new FileWriter((new File( “hoge/foo.txt” );
     FileReader fr = new FileReader(new File( “hoge/foo.txt” )){
          fw.write( “田中” ); fw.flush(); inti=0;
          while((i = fr.read()) != -1){
               System.out.println((char)i);
          }
     );        ShiftJISで読み込んでも文字化けしない
```

BufferdReaderとBufferedWriterクラス　java.io
文字列をブロック単位で読み書きするためのストリームを提供。バッファはデータを一時的にためておく意味。バッファに文字列をためていき、たまった文字列をまとめて読み込んだり書き出したりする
BufferdReaderとBufferedWriterクラスのコンストラクタとメソッド

コンストラクタ名	説明
BuffredReader(Reader in)	引数で指定された入力ストリームからデータを読み込むための、BufferedReaderオブジェクトを作成。デフォルトサイズのバッファでバッファリングする
BuffredReader(Reader in, int sz)	引数で指定された入力ストリームからデータを読み込むための、BufferedReaderオブジェクトを作成。引数で指定されたのバッファサイズでバッファリングする
BuffredWriter(Writer out)	引数で指定された出力ストリームへデータを書き出すための、BufferedWriterオブジェクトを作成。デフォルトサイズのバッファでバッファリングする
BuffredWriter(Writer out, int sz)	引数で指定された出力ストリームへデータを書き出すための、BufferedWriterオブジェクトを作成。引数で指定されたバッファサイズでバッファリングする

メソッド名	説明
Int read() throws IOException	ストリームから単一文字を読み込む。ファイルの終わりに達すると－1を返す
String readLine() throws IOException	1行のテキストを読み込む。1行の終わりは、改行¥nか、復帰¥r、または復行とそれに続く改行のどれかで認識される。終わりに達するとnullを返す。
void mark(int readAheadLimit) throws IOException	ストリームの現在位置にマークを設定する。引数にはマークを保持しながら読み込むことができる文字数の上限を指定 e.g. mark(256) → マークを保持しながら読み込むことができる文字数の上限を２５６バイトに指定
void reset() throws IOException	ストリームをmark()によりマークされた位置にリセットする
long skip(long n) throws IOException	引数で指定された文字数をスキップする。

void write(String str) throws IOException　　引数で指定された文字列を書き出す(親クラスのWriterクラスのメソッド
void newLine() throws IOException　　改行文字を書き出す。改行文字はシステムのline.separatorプロパティにより定義
void flush() throws IOException　　目的の送信地に、直ちに文字を書き出す
read, readLine, mark, reset, skipはBufferedReaderクラスそれ以外はBufferedWriterクラス
mark(), reset()のサポートの有無は入力ストリームの種類ごとに異なる。
FileInputStreamはサポートしていない。代わりにInputStreamはmarkSupported()が提供されている
サポートされている場合はtrue、それ以外の場合はfalseを返す
e.g. BufferedWriter fw = new BufferedWriter(new FileWriter(“hoge/foo.txt”);
BufferdReader fr = new BufferdReader(new FileReader(“hoge/foo.txt”)){
fw.write(“田中”);　fw.newLine();　fw.write(“山田”);
fw.flush();　String data=null;　newLine()で田中と山田の間に改行コードを書き出し
while((data = fr.readLine()) != null){　readLine()メソッドを使用し、１行単位で読み込み　ファイルの最後まで到達するとnullを返す
System.out.println((char)i);　→　田中
}　山田　と出力される
);　ShiftJISで読み込んでも文字化けしない

e.g. BufferdReader br = new BufferdReader(new FileReader(“hoge/foo.txt”)){
txtはapple, orange, banana
System.out.println(br.readLine());　→　apple
A地点　br.mark(256)　→　マークを保持しながら読み込むことができる文字数の上限を２５６バイトに指定　この時点でreset()された場合、前の行には戻れなくなる
System.out.println(br.readLine());　→　orange
System.out.println(br.readLine());　→　banana
br.reset() →　マークされた位置にリセットされる　ここではA地点に戻る
System.out.println(br.readLine());　→　orange
br.skip(2);　二文字スキップ　ここではbananaのbaをスキップする
System.out.println(br.readLine());　→　nana
}　山田　と出力される
);　ShiftJISで読み込んでも文字化けしない

e.g. hoge.txtの中身はabcdefgh
try(BufferedReader in = new BufferedReader(new FileReader(“hoge.txt”))){
for(int i = 0; i<3;i++){
in.skip(i);

```
                println((char)in.read());          //skip(0)なのでaを出力
skip(1)なのでcを出力 skip(2)なのでfを出力
            }
                                                   //read()で読み込んだ場合、カーソ
ルは次の文字になっている。ここではgにカーソルが合わせている
            in.mark(3);                            //ストリームの現在位置をマーク
ここではg
            println(in.readLine());                //改行までを出力ここではghが出力
            in.reset();                                //マーク位置まで戻る　ここ
ではgに戻る
            println((char)in.read());                  //現在のカードルはgに戻って
いるのでgを出力
        }catch(IOException e){
            e.printStackTrace();
        }
```

Systemクラスの定数

Systemクラスの定数

定数	説明
public static final InputStream in	標準入力ストリーム
public static final PrintStream out	標準出力ストリーム
public static final PrintStream err	標準エラー出力ストリーム

PrintStremとPrintWriterはデータの出力専用であり、通常のWriter, FileWriterに比べて以下の拡張機能がある。両方は提供する機能はほぼ同じ。PrintWriter使用推奨

・プリミティブ型をそのまま出力
・書式変換付き出力
・自動フラッシュ

in定数はキーボード入力、out, err定数はディスプレイ出力と一致する。in定数を用いるとコンソールからデータで読み込むことが可能

```
e.g. BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
     String s = br.readLine();          System.out.println(“input :  “ + s);
```

出力ストリームとしてjava.io.Writerクラスがある。出力時に書式化可能。

```
e.g.  FileOutputStream out = new FileOutputStream(“my.txt”);
      PrintWriter writer = new PrintWriter(out);
      writer.write(“GOLD”); writer.write.append(“Silver”);
writer.println(100.0);     writer.flush();     writer.close();
```

write()メソッドはオーバーロードされているが、基本的には文字列もしくは単一文字の書き込み用とし提供されている

append()メソッドはwrite()メソッドと同様の処理を行う。基本データ型を引数にとるprint(), println()メソッドが提供されている。
write(), append(), print(), println()の各メソッドは全て書き込み処理を行う。違いは暗黙でflush()が呼ばれるか否か。
PrintWriterのインスタンス化を行う際に、コンストラクタで自動フラッシュを行うようboolean値で指定(true)すると、
println(), printf(), format();の各メソッドは処理後、暗黙でflush()メソッドが実行される

シリアライズ　java.io

オブジェクトを出力ストリームに書き出すことをシリアライズまたは直列化という。シリアライズされたオブジェクトを読み込んでメモリ上に復元することをデシリアライズ、直列化復元。
シリアライズしたいオブジェクトはクラス定義の際に注意が必要。シリアライズ可能なオブジェクトとなるようにクラス定義する必要がある。
シリアライズ化するにはjava.io.Serializableインタフェースを実装。
Serializableインターフェースはメソッドや定数を持たないインターフェースなのでオーバーライドするメソッドはない
Implements Serializableするだけでオブジェクトは出力ストリームへの書き込み、入力ストリームからの読み込みができるようになる。
オブジェクトのシリアライズは、オブジェクトが保持する固有データ、つまりインスタンス変数がシリアライズ対象データとなる。
シリアライズ可能なデータは、基本データ型、配列、他のオブジェクトへの参照です。static変数はシリアライズ対象外。明示的に対象外にしたいインスタンス変数がある場合は変数にtransient修飾子を指定

ObjectInputStreamクラスとObjectOutputStreamクラス
オブジェクトを入出力するためのストリームを生成する。シリアライズ可能なオブジェクトの入出力には、ObjectInputStream, ObjectOutputStreamクラスを使用
ObjectInputStream, ObjectOutputStreamクラスの主なメソッドとメソッド
コンストラクタ名　説明
ObjectInputStream(InputStream in) throws IOException　引数で指定されたInputStreamから読み込むObjectInputStreamを作成
ObjectOutputStream(OutputStream out) throws IOException　引数で指定されたOutputStreamに書き出すObjectOutputStreamを作成

final Object readObject() throws IOException, ClassNotFoundException
ObjectInputStreamからオブジェクトを読み込む。戻り値はObject型なので適宜キャストする必要あり

ファイルから読み込んでオブジェクトの復元を行なっているため、コンストラクタは呼ばれない(親が実装してた場合のみ)

static変数やtransient修飾子指定した値はデフォルトの初期値0で初期化される
final void writeObject(Object obj) throws IOException
引数で指定されたオブジェクトをObjectOutputStreamに書き出す。

引数のオブジェクトはシリアライズ可能なオブジェクトでする必要がある
e.g. Hoge hoge = new Hoge(100, “hoge”);
ObjectOutputStream oos = new ObjectOutputStream(new FIleOutputStream(“hoge/foo.txt”);
ObjectIntputStream ois = new ObjectInputStream(new FIleInputStream(“hoge/foo.txt”);
oos.write(hoge) → 書き出し
Hoge hoge2 = (Hoge)ois.readObject(); → 読み込み 戻り値がObject型なのでキャストする必要がある 例外処理は2つ記述 コンストラクタは呼ばれない
hoge2.getId(); hoge2.getName();でメンバ変数の参照可能

シリアライズの継承

注意点
・配列・コレクションをシリアライズする場合は、その要素のそれぞれがシリアライズ可能でなければならない
・シリアライズされたオブジェクトがオブジェクト参照変数によって参照するオブジェクト(参照先のオブジェクト)はシリアライズ可能でなければならない
・static変数及びtransient指定された変数はシリアライズ対象外となる。
・あるクラスがシリアライズ可能であれば、そのクラスを親クラスとする全ての子クラスは、明示的にSerializableを実装していなくても暗黙的にシリアライズ可能
→デシリアライズした際にコンストラクタは呼ばれない
・サブクラスがSerializable実装している場合は親クラスはでシリアライズの際にインスタンス化される。(引数を持たないコンストラクタが実行される)
・コンストラクタと初期化ブロックはでシリアライズ時には実行されない。
transient指定した変数はnullになる

コンソール

SE6からjava.io.Consoleクラスが提供されている。Consoleオブジェクトを用いることでコンソール上での入力(標準入力)、出力(標準出力)を扱う。
Consoleクラスのコンストラクタはprivate指定のためnewできない。ConsoleオブジェクトはSystemクラスのConsole()で取得 Console console = System.console();
Consoleクラスはシングルトンパターンが適用されており、Console()メソッドが呼ばれると、実行中は一意のConsoleオブジェクトが返る。
入出力が可能な端末かどうかは、使用する端末に依存します。コンソールデバイスが利用できない場合、Console()メソッドはnullが返る。
Consoleオブジェクトに対し以下のメソッドを呼ぶことで、コンソールのストリー

ム読み書きできる
Consoleクラスの主なメソッド

メソッド名	説明
PrintWriter writer()	PrintWriterオブジェクトを取得する。
Console format(String fmt, … args)	指定された書式文字列及び引数を使用して、書式付き文字列をこのコンソールの出力ストリームに書き出す
Console printf(String fmt, Object… args)	指定された書式文字列及び引数を使用して、書式付き文字列をこのコンソールの出力ストリームに書き出す
String readLine()	コンソールから単一行のテキストを読み込む
String readLine(String fmt, Object… args)	書式設定されたプロンプトを提供し、次にコンソールから単一行のテキストを読み込む
char[] readPassword()	エコーを無効にしたコンソールからパスワードまたはパスフレーズを読み込む
char[] readPassword(String fmt, Object… args)	設定されたプロンプトを提供し、次にエコーを無効にしたコンソールからパスワードまたはパスフレーズを読み込む

e. g. Console console = System.console();
PrintWriter pw = console.writer(); Consoleオブジェクトを取得
while(true){
String str = console.readLine(); コンソール上で入力した文字列が読み込まれる
if(str.equals(“”)){ break;}
pw.append(str + ‘¥n’); write()メソッドと同様に動作する。文字列の書き出し→pw.write(str + ‘¥n’);でも可能 System.out.println(str)でも可能
pw.flush();
} コンソールでは何も入力せずEnterキーを押すと終了する。

e. g. Console console = System.console();
String name = console.readLine(“%s”, “name:”); 名前の入力を促し、読み込んでいる。
System.out.println(“You Are “+ name);
char[] pw = console.readPassword(“%s”, “pw: “); パスワードの入力をそくし、読み込んでいるが、入力したパスワードは画面に表示していない。
System.out.print(“Your Password : ”);
for(char c: pw)
System.out.println(c);

一番の特徴は、入力文字のエコーバックを抑制できる点にあり、パスワードなどを入力させるプログラムで効果を発揮する。

ストリームの書式化及び解析

入出力ストリームのフォーマットを行うためのAPIが提供されている。
format()メソッドを使って入力データを書式化した後は、書式化された出力を
Formatterクラスや PrintWriterクラスを使って、ファイルなどの入出力デバイスに流すことができる。
Formatterクラスは、数値、文字列、一般的なデータ型、日付や時刻のデータなどを書式化するフォーマッタを生成。

Formatterクラスのコンストラクタとメソッド

コンストラクタ名	説明
Formatter()	新しいフォーマッタを作成する
Formatter(Locale l)	指定されたロケールを持つ新しいフォーマッタを作成する
Formatter(File file) throws FileNotFoundException	指定されたファイルを持つ新しいフォーマッタを作成する
Formatter(PrintStream ps)	指定された出力ストリームを持つ新しいフォーマッタを作成する
Formatter(OutputStream os)	指定された出力ストリームを持つ新しいフォーマッタを作成する

メソッド名	説明
Void flush()	フォーマッタをフラッシュする
void close()	フォーマッタを閉じる
Formatter format(String format, Object… args)	第一引数に書式化パターンをを指定し、書式化したい値を第二引数で指定し、フォーマッタに書き出す
String toString()	フォーマッタの中身をStringオブジェクトで返す

PrintWriterクラスやPrintStreamクラスに用意されているformat(),printf()メソッドはFormatterクラスのformat()メソッドと同じ振る舞いをする

PrintWriter format(String format, Object… args)	第一引数に書式化パターンを指定し、書式化したい値を第二引数で指定し、このライターに書き出す。
PrintWriter printf(String format, Object… args)	第一引数に書式化パターンを指定し、書式化したい値を第二引数で指定し、このライターに書き出す。

各メソッドの第一引数には、書式情報を含んだ文字列を指定します。第二引数以下で変換される値を指定する。
format(<フォーマット指示子>, <引数>)
<フォーマット指示子>は書式化を行うための指示を与え、<引数>は書式化されるデータを指定

フォーマット指示しの構文([]は省略可能)
%[インデックス$][フラグ][幅][.精度]変換の種類　e.g.　“%,4.2f”
インデックス　%の後に、数字$を記述すると置換引数を明示的に指定可能　数字の示す順番に変数を記述する必要がある
フラグ　　+　符号を出力　　0　秋を0で埋める　　,　数値を桁ごとに「,」で区切る。ロケールに依存
幅　　出力時の最小文字数
.精度　　精度。出力に書き込まれる最大文字数
変換の種類　　b(boolean), c(文字), d(整数), f(浮動小数点数), s(文字列), n(行区切り文字)

e.g. String str = “hoge”
Formatter fm = new Formatter();
fm.format(“hoge %s ¥n”, hoge);
System.out.format(“hoge %s ¥n”, hoge);
System.out.printf(“hoge %s ¥n”, hoge);
上記は全て結果は同じ

NIO.2
これ以降は紫ぼんの２５５ページから参照

ファイルの属性の取得・設定、任意ディレクトリの変更・監視などができる新しいファイルシステムAPIが追加
従来のjava.io.Fileの問題点を打ち消すためにjava.nio.fileパッケージでPathインタフェースが追加された
Java.nio.file　　ファイル、ファイル属性及びファイルシステムにアクセスするためのインターフェースとクラスを提供
java.nio.attribute　　ファイル・ファイルシステム属性へのアクセスを提供するインタフェースクラスを提供

Pathオブジェクトを取得するには以下のメソッドを使用
FileSystems.getDefault().getPath(“path”);
Paths.get(“Path”);
java.io.FIleオブジェクトからPathオブジェクトを得るには
Path path = file.toPath();

相対パスの場合 getRoot()はnullを返す
e.g.　　C:¥x¥y¥z の場合
getRoot()　　C:¥を返す
getName(int index)　　getName(0)の場合はxが返る　ルートに最も近い階層が0　getFileName()　　zを返す　　最下層のパスを返す　ファイルだけでなくディレクトリも対象　ルートだけの場合はnullを返す
getFileCount()　　3を返す　　パス階層全体のパスの個数をカウント　ルートは含まない　ルートだけの場合は0を返す

Subpath()で要素外や開始と終了インデックスを同じにした場合は
IlligalArgumentExceptionがスロー
e.g. Path path = Paths.get(“c:¥x¥y¥z”);
println(path.subpath(1, 2)); yが返る 終端は含まない

URIは先頭にfile:///がつく
get()で指定されたパスの文字列を取得、引数が2つ以上だと連結したものを返す
isAbsolute()は絶対パスの時のみtrueを返す

Normalize,resolve, relativeは試験によく出る

normalize()により冗長部が消されると、./hoge → hoge になる 冗長部について調べる
基本的にはカレントディレクトリを表すピリオドや親ディレクトリを表すダブルピリオドが対象
e.g. C:¥¥x¥¥y¥¥..¥¥..¥¥xx¥¥.¥¥a.txt → C:¥xx¥a.txt
¥¥..¥¥..¥¥xxは二階層上を指しC:¥を表すので¥¥x¥¥yは除去

toAbsolutePath()は現在のPathオブジェクトを元に絶対パスを返す(正確ではない)
toRealPath()では現在のPathを元に実際の絶対パスを返す。IOExceptionをスローするので例外処理必須

resolve()で引数の相対パス同士を連結 絶対パスを指定した場合はそのままそれを返す 物理的にディレクトリを作成 空の場合は自分自身を表すパスを返す
e.g. Path path1 = Paths.get(“C:¥¥abc”);
Path path2 = Paths.get(“C:¥¥xyz”);
Path path3 = Paths.get(“abc”);
println(path1.resolve(path2).resolve(path3)); C:¥xyz¥abc セパレーターも付与
println(path1.resolveSibling(path2)); C:¥abc 引数が相対パスだと呼んだ側の親ディレクトリ配下のパスとして解決 絶対パスだとそのまま返す
呼んだ側が相対パスだと引数のパスをそのまま返す
println(path1.relativise(path2)); ..¥xyzを返す 引数のパスを呼んだ側の相対パスとして考える

isSameFile()は相対パスでも絶対パスでも指定された2つのファイルまたはディレクトリであればtrueを返す IOExceptionにより例外処理必須 NoSuchFileException

delete()は指定されたパスのファイル、ディレクトリを削除。物理的に存在しない

場合 java.nio.file.NoSuchFileException
deleteIfExsists()は物理的に存在しているかの確認を行うので上記の例外が発生しない。削除対象がディレクトリの場合は空の時のみ削除→ファイル等が格納されているとDirectoryNotEmptyException

Filesクラス

ファイルやディレクトリの単一ファイル属性を取得するにはFilesクラスのgetAttribute()を使用
static Object getAttribute(Path path, String attribute, LinkeOption… options)
第二引数には取得したい属性名を記述　省略するとbasic

戻り値はObjectなのでキャストする必要あり

copy(), move()
copy()
第三引数にREPLACE_EXISTINGを指定しないと、移動先、コピー先に同盟ファイルが存在する場合はFileAlreadyExistsExceptionが発生
デフォルトではファイル属性はコピーされない
ディレクトリのコピーも可能、ディレクトリ内にファイルがある場合はファイルはコピーされない

move(hoge, foo, StandardCopyOption.ATOMIC_MOVE)
普通のリネームで上書きされる。ディレクトリの場合は存在している場合はエラー、
シンボリックリンクはシンボリックリンクファイルとして移動するので、NOFOLLOW_LINKSの有無は関係ない

Linuxではファイルシステムの機能してシンボリックリンクがある。これはファイルもしくはディレクトリを指し示すショートカットのようなもので、
ファイルの実態は1つだが、シンボリックリンクからその実態を参照する ls -laで表示
copy()メソッドでシンボリックリンクをコピーしても参照先の実体をコピーしてしまう。あくまでリンクファイルとしてコピーする場合はオプションとして、NOFOLLOW_LINKSを指定
シンボリックリンクはファイル及びディレクトリに対して作成可能

Streamオブジェクトの取得
readAllLines()はpathオブジェクトを引数に、１行ごとの要素を持つListオブジェクトを取得することができる。拡張forなどで出力可能　デフォルトではutf8　第二引数で文字コード指定可能
lines()は上記と違い、ファイル内のすべての行を読み取る。読み取った行をStream<String>型で返す

java.nio.file.attribute
ファイル及びファイルシステム属性のアクセスを提供するインタフェースとクラス群は java.nio.file.attributeで提供。
BasicFileAttributes　　基本的なファイル属性を表す
DosFileAttributes　　DOS(NTFS含む)ファイルシステム(Windows全般)におけるファイル属性全般を扱う
PosixFileAttributes　　UNIX, LINUXなどのPOSIX互換のファイルシステムにおけるファイル属性全般を扱う

ファイル属性のセットも表す。属性セットを属性ビューという。AttributeViewインタフェースで提供

readAttributes()を使用すると基本的なメタデータを参照できる
BasicFileAttributeオブジェクトを取得できる。
最終更新時間やファイルサイズなど参照可能だが変更するメソッド、setXXX等は提供されていない

getRootDirectories()メソッドはルートディレクトリを取得。複数ドライブがある場合はイテレーターで全て取得できる
newDirectoryStream()で拡張forを用いてディレクトリ内のエントリを取得可能

Stream<Path> walk()　　ディレクトリ探索において、サブディレクトリ以下も再帰的に探索する.最大深度はInteger.MAX_VALUE　シンボリックリンクは辿らない
walkFileTree()　　再利用性の高いディレクトリ階層の再起処理(トラバース)を簡単に行える　引数はFileVisitorインタフェースを実装
list()　　メソッド対象となるディレクトリのみ探索を行う。再帰的に辿らないのでサブディレクトリ以降は探索しない
　　ディレクトリを表すPathオブジェクトのみ引数指定可能。ファイルを表すPathオブジェクトを指定したらNotDirectoryExceptionがスロー
find()　　パスに対して条件を指定して合致するパスのみで構成するストリームを返す。第一引数は探索対象のpath、第二引数は階層の深さ、第三引数はBiPredicate　boolean test(T t, U u)

FileSystemは抽象クラス、Pathはインタフェースなのでnewできない
FileSystemはプラットフォーム固有のファイルシステムを表す
ファイルそのものを表すのがFileSystemクラス、ファイルシステムを取得するためのファクトリを提供するのがFileSystemsクラス
static FileSystem getDefault()で取得する。デフォルト以外のファイルシステムを利用するには以下を使用
getFileSystem(), newFileSystem()を使用して、そのファイルシステムを参照することのできるURIオブジェクトやClassLoaderオブジェクトなどの

適切な情報を引数に渡す必要がある

java.nio.file.Filesクラスはstaticメソッドのみを提供するユーティリティクラス
ファイルやディレクトリも作成・削除・コピー・移動、ファイル属性の取得・設定など
FileクラスのlistFiles()はディレクトリ内の抽象パスをFile型配列で返す。Filesクラスのlist()はディレクトリ内のエントリを要素にもつStreamを返す。戻り値は型パラメータを持つStream型
黒本255参照

NIOで提供されている機能
・シンボリックリンクをサポート
・ファイル更新時間の取得
・単一のメソッドでディレクトリーツリー内を横断
・ファイルやディレクトリの削除
・システムに依存した属性をサポート
・ディレクトリ内にあるすべてのファイルを表示
赤字はjava.io.Fileクラスでも提供されている。更新時間を取得するには
lastModified()を使用

NewIOは絶対パスと相対パスでの挙動の違いをしっかり理解する必要がある
Windowsベースのファイルセパレータを使用する場合はエスケープシーケンス￥を使用する

Thread

並列処理ユーティリティが提供する機能として以下の4つがある
java.util.concurrent　　java.util.concurrent.atomic
　　java.util.concurrent.locksで提供
カウンティング・セマフォ　　有限のリソースに対して並行的にアクセスするプロセスやスレッド間における同期や割り込み制御に用いられる仕組みのこと
　　バイナリ・セマフォ　　リソースに対するアクセスが可能か不可能かのいずれかの値をとる
　　カウンティング・セマフォ　　アクセス可能なリソース数を任意に設定することが可能
並列コレクション　　状態へのアクセスを直列化することでスレッドセーフを実現、パフォーマンス劣化の要因にもなる
アトミック変数　　その処理が不可分であることを表す。処理は完全に終了しているか、未着手のどちらかでないといけない
スレッド・プール　　必要となるスレッドを予め生成し、貯蔵。スレッド生成におけるオーバーヘッドを軽減、APで必要なスレッドの管理性を向上
　　Executorフレームワークが提供

並行処理ユーティリティのCyclicBarrier
各スレッドの足並みを合わせるための機能を提供、各スレッドは設定されたバリアに到達すると他のスレッドがバリアに到達するまで待機　スレッド間の協調に利用可能
CyclicBarrierのコンストラクタには、スレッドパーティの数に加えて、トリップが発生した際に実行する処理をRunnableオブジェクトで実装して渡すことが可能

Threadクラスを使用したタスクの実行すると、以下の問題点がある
スレッド数の上限を設定することができない。上限を超えると
OutOfMemoryException
スレッド生成と破棄におけるオーバーヘッドが生じたり、無駄にリソースを消費する場合がある

Runnableインタフェースの抽象メソッドであるrun()メソッドをオーバーライドしてスレッド処理を記述
Runnableインタフェースを実装したクラスのオブジェクトをThreadクラスのコンストラクタの引数に渡す。その後start()メソッドを呼び出すとrun()メソッドが実行される

```
TestThread test = new TestThread();
Thread thread = new Thread(test);
thread.start()
```

```
class TestThread
implements Runnable{

public void run(){}
}
```

Runnableインタフェースの抽象メソッドであるrun()メソッドしかないのでラムダ式で記述可能
従来

```
new Thread(new Runnable(){
     public void run(){
          System.out.println( “hello” );
     }
}).start()
```

ラムダ式

```
New Thread( () -> { System.out.println( “hello” );}).start();
```

スレッドの状態
java.langにはThread.Stateがあり、スレッドの状態を表す列挙型。6つの状態がある
スレッドはstart()が呼び出された時点で始まる。タスクが完了すると終了
start()を呼び出した時点では即座に実行されるわけではない。start()はスレッドを実行可能状態にする。スケジューラにより実行状態にされるまでそのまま
スレッドが実行状態に移った時に、run()が呼ばれる。終了したスレッドを再度実行すると(1つのオブジェクトに対し、２回start()を呼び出すなど)
IlligalThreadStateExceptionが発生

スレッドには優先順位があるが、OSからの実行の割り当てに依存するため、優先度の高いものが必ずしも先に実行されることが保証されているわけではない
setPriority(int newPriority)で優先度変更　1から10まで、デフォルトは５

スレッド制御

static void sleep()　　millisミリ秒休止　InterruptedException
sleep時間を経過しても即座に実行状態に移行しない。スケジューラにより実行状態になる
final void join()　　実行中のスレッドが終了するまで待機
InterruptedException
static void yield()　　現在実行中のスレッドを一時休止、他のスレッドに機会を与える
void interrupt()　　休止中のスレッドに割り込みを入れる。割り込みを入れられたスレッドは、java.lang.InterruptedException例外をjava実行環境から受け取り処理を再開
　　割り込まれる側はtry catchで受け取り処理を再開

排他制御

Synchronizedを指定すると同時に1つのスレッドからしか実行されないことを保証
staticにも使用可能
e.g. public synchronized void add(int a) { …}
　　syncronized(ロック対象のオブジェクト) 自オブジェクトを指定する場合はthisを指定

同期制御

排他制御を実現してもメソッドの順番を制御できない。同じメソッドが連続で呼ばれたりする
Objectクラスで提供されている以下のメソッドを用いて同期制御を実現

final void wait()　　他のスレッドがこのオブジェクトのnotify(), notifyAll()を呼び出すまで現在のスレッドを待機させる
final coid wait(long timeout)　　他のスレッドがこのオブジェクトのnotify(), notifyAll()を呼び出すまで現在のスレッドを指定の時間が経過するまで待機させる
final void notify()　　このオブジェクトの待機中のスレッドを1つ再開する。特定の再開するスレッドを指定することはできない
final void notifyAll()　　このオブジェクトの待機中のすべてのスレッドを再開する
synchronized指定されていないメソッドやブロックで使用した場合は、
IllegalMonitorStateExceptionが発生

スレッドはwint()メソッドを実行すると待機状態になり、以下のいずれかが発生すると実行可能状態になる
待ち時間を指定したwait()の場合はタイムアウトした時
他のスレッドがnotify()を呼んだ時
他のスレッドがnotifyAll()をよんだ時
他のスレッドがinterrupt()をよんだ時

資源の競合

複数のスレッドが共有資源の獲得と解放を行なっているが、獲得が必要な時には他のスレッドにロックされ、進まない処理を繰り返し状況が発生することがある。
上記のように実質的に処理が進まない状態をライブロックという
コンパイルのタイミングでは検出できない。例外の通知もない→プログラムで制御。ロック順番を予め決めたり、wait()を使用しタイムアウト時間を設定し強制的に待機状態を解除など
共有オブジェクトをなかなか解放しないスレッドがあると、他のスレッドが実行を長時間待たされる→スレッドスタベーションという、優先順位を見直す、ロック粒度小さくロック回数減らすなどを行う

並列コレクション

_BlockingQueue, ConcurrentHashMap, CopyOnWriteArrayListインタフェースが含まれる

ConcurrentHashMap　　アトミック操作をサポート　ロックストライピング方式を採用　staticやisEmptyメソッドが返す値が近似値となる場合あり
ConcurrentLinkedDeque
CopyOnWriteArrayList　　要素を変更した際は配列のコピーを作成し、コピー時に操作をする　他のスレッドはコピー前のリストを参照
　　リストサイズが大きくなるとパフォーマンスが低下。Collections.synchronizedListの使用を検討
上記３クラスの利用方法は理解しておく
イテレーターを用いなくても、拡張for文で繰り返し処理を行うと、実行環境上ではイテレータが使用されているが処理内容によっては
ConcurrentModificationExceptionが発生
イテレーターはファイルファスト・イテレーターと呼び、このイテレータの反復処理中にコレクションに変更の可能性があるならば即座に例外を発生して処理を中断する仕様になっている
内部イテレーターによるコレクションの変更を行う場合や複数のスレッドで1つのコレクションオブジェクトを使用する場合はスレッドセーフなコレクションを使用

並列処理機能を提供するパッケージ

java.util.concurrent　　並列プログラミングでよく使用されるユーティリティを提供。マルチスレッドアプリケーションで活用されるコレクションフレームワーク群やスレッドプールなど
java,util.concurrent.atomic　　パッケージjavaでアトミック操作を行うクラス群を提供
これらのパッケージが提供する機能を総称して、Concurrency Utilitiesという。非同期処理、スレッドプーる、Concurrencyコレクション、アトミック変数などの機能を提供

コレクションクラスの多くは同期性をサポートしていないので、synchronizedメソッドやCollectionsクラスのsynchronizedList()メソッド、synchronizedMap()などを使用する
ただし、synchronizedXXX()メソッドを使用して、スレッドセーフなオブジェクトを取得しても、イテレーターを使う時には明示的に同期化を行う実装をしなければconcurrentModificationExceptionが発生

```
e.g. List list = Collections.synchronizedList(new ArrayList());
     synchrnoized(list){
          Iterator iter = list.iterator();
          while(iter.hasNext())   foo(iter.next());}
```

BlockingQueueはQueueインタフェースを拡張したもの。要素の取得時にキューがからであれば要素が追加されるまで待機したり、要素の格納時にキュー内が一杯であれば空が生じるまで待機する
ConcurrentLinkedQueueクラスはスレッドセーフなQueueインタフェースの実装クラス
ConcurrentLinkedQueueクラスはLinkedBlockingQueueクラスと同様にリンクノードに基づいたキューを提供している。違いとしてブロッキングを使用していないため高速に処理が行われる
offerXXX()で第一引数の要素を格納
詳細は紫本295を参照

Mapインタフェースの拡張

java.util.concurrentでは以下のクラスとパッケージを提供
ConcurrentMapインタフェース　　アトミックなputIfAbsent(), remove(), replace()メソッドを提供するマップ
ConcurrentHashMapクラス　　ConcurrentMapインタフェースを実装した同期をサポートしたマップクラス

ConcurrentMapインタフェース のメソッド

V putIfAbsent(K key, V value)　　指定されたキーがまだ値と関連づけられていない場合は指定された値に関連づける。別スレッドからの割り

込みが入らないこと
boolean remove(Object key, Object value)　　キー及びそれに対応する値をこのマップから削除する。キーがマップにない場合は何もしない
V replace(K key, V value)　　キーが値に現在マッピングされている場合にのみそのキーのエントリを置換する
boolean replace(K key, V oldValue, V newValue)　　キーに指定された値で現在マッピングされている場合にのみ、そのキーのエントリを置換する

ArrayList及びSetクラス・インタフェースの拡張

CopyOnWriteArrayList　　もととなる配列の新しいコピーを作成することにより、スレッドセーフを実現するArrayListを拡張
CopyOnWriteArraySet　　内部でCopyOnWriteArrayListオブジェクトを使用して、スレッドセーフを実現するSetインタフェースの拡張

通常のArrayListでスレッドを利用して、リストに変更(追加や削除)を行うとconcurrentModificationExceptionが発生 CopyOnWriteArrayListを使用する
CopyOnWriteArrayListではイテレータを作成した時点での状態を参照するため、イテレーター取得後の元のリストへ追加削除変更があっても反映されない。
SE6よりNavigatableMapインタフェースが提供されている。SortedMapインタフェースのサブインタフェースで指定されたキーに対して、最も近い要素を返すナビゲーションメソッドを持つ
これによりキーをもとに検索する場合、指定したキーに一致するものがない場合、指定したキーに近い要素を返す。これを実装したクラスとしてTreeMapがある。
また、同じようにナビゲーションメソッドを持つセットとしてNavigableSetインタフェースがあり、同期性無しの実装クラスTreeSet、同期性ありの実装クラスとしてConcurrentSkipListSetがある。

Executorフレームワーク(試験に出る可能性が高い)　紫本300あたりを参照

タスクの実行におけるライフサイクルの管理
スレッドプールの実装
タスクの遅延開始と周期的実行
必ずプログラムの終了時にはshutdown()を記述する。記述しないとプログラムが終了しない

Executorフレームワークを使うとスレッドの再利用や、スケジューリングを行うスレッドコードを簡単に実装できる
Executorクラスはスレッドプールを利用するメソッドが提供されており、タスク数に応じて自動的にスレッドを増やすものもあれば、固定のスレッド数で処理を行うものもある。
いずれも利用可能なスレッドがタスクの実行中の場合、処理しきれないタスクはスレッドが使用可能になるまで待機する

Executorインタフェース　　送信されたRunnableタスク(1つの処理)を実行するオブジェクト
ExecutorServiceインタフェース　　終了を管理するメソッド、及び非同期性タスクの進行状況を追跡するFutureを生成するメソッドを提供する
Futureインタフェース　　非同期性計算の結果を表す。計算が完了したかどうかのチェック、完了までの待機、計算結果の取得などを行うためのメソッドを提供
　　Future<?>とすると例外オブジェクトやnullなどなんでも格納できる
　　submit()はタスクを表すFutureを返す。Futureオブジェクトはタスクが完了したかどうかのチェック、完了までの待機、タスク結果の取得などを行うメソッドがある
　　get()メソッドでタスクの完了結果を取得できる。正常にタスクが完了した場合、Futureのget()はnullを返す。
　　get()は処理が完了するまで待機する
　　submit()で第二引数を指定した場合、正常にタスクが完了すると第二引数の値を返す
Callableインタフェース　　タスクを行うクラス。結果を返し、例外をスローすることがある。
Executorsクラス　　Executor、ExecutorService, ScheduledExcecutor, Service, ThreadFactory, Callableオブジェクト用のファクトリ及びユーティリティメソッドを提供

通常、Executorオブジェクトは明示的にスレッドを作成する代わりに使用される。new Thread(new(RunnableTask())).start()を呼ぶ代わりにExecutorオブジェクトのexexute()メソッドを呼ぶ
e.g. Executor executor = //Executor オブジェクト　executor.execute(new RunnableTask());
execute()はRunnable型なのでラムダ式で記述可能
execute()はfor文で回す際、１回目のexecute()が完了しなければ２回目のexecute()は実行されない。
shutdown()はExecutorServiceの終了となり、新しいタスクの受け入れはしない。すでに実行中のタスクや待機しているタスクがあれば実行され、終えた後にExecutorServiceの終了と
shutdown()を記述しないと待機状態となりプログラムは終了せず待機状態となる
shutdownNow()は即時的なシャットダウン(タスクキューに残っていてまだ実行されていないタスクはキャンセル)を試みる
Executor終了後に新たなタスクの実行を依頼しようとすると
,java.util.concurrent.RejectedExecutionExceptionがスロー

検証用にisShutdown()がある。ExecutorServiceがシャットダウンしていた場合、trueを返す。
ExecutorServiceから実行された全てのタスクが完了しているか検証するメソッドと

してisTerminated()メソッドも提供されている

e.g. ExecutorService service = Executors.newSingleThreadExcecutor();
service.execute();　これにより単一スレッドで実行され計算結果の処理は常に同じになる
ExecutorServiceとタスクの生成
↓
タスク実行中　タスク受け入れOK　isShutdown()→false　isTerminated()→false
↓ shutdown()メソッドの実行
タスク実行中　タスク受け入れOK　isShutdown()→true isTerminated()→false
↓全てのタスクが終了
タスク終了　タスク受け入れOK　isShutdown()→true isTerminated()→true

実行中のタスク処理をシャットダウンする場合は、shutdownNow()メソッドを使用する。
shutdownNow()は戻り値として実行を待機していたタスクのリスクを返す。
submit()はタスクを表すFutureを返す。
Futureオブジェクトはタスクが完了したかどうかのチェック、完了までの待機、タスク結果の取得などを行うメソッドがある。タスクのキャンセル試行も行う
get()メソッドでタスクの完了結果を取得できる。正常にタスクが完了した場合、Futureのget()はnullを返す。submit()で第二引数を指定した場合、正常にタスクが完了すると第二引数の値を返す
get()はInterruptedExceptionとExecutionExceptionをスローする宣言をしているので例外処理は必須
ScheduledExecutorServiceインタフェースはExecutorServiceを継承している。
ExecutorService service = Executors.newSingleThreadSchduledExecutor()と記述可能
ScheduledExcecutorServiceインタフェースが提供する主なメソッドは以下のとおり
scheduleAtFixedRate()　指定された初期遅延の経過後に初めて有効　第三引数で指定された時間ごとにタスクを実行
scheduleWithFixedDelay()　指定された初期遅延の経過後に初めて有効　第三引数で指定された時間に従って遅延した後、タスクを実行
上記2つのメソッドともに第一引数はRunnable型。Callable型を指定するとコンパイルエラー
ExecutorService は上記2つのメソッドを実装していないので使用するとコンパイルエラー

Callableインタフェース

Runnableインタフェースのrun()はvoidなので処理結果をスレッドに戻せない。
java.util.concurrent.Callableでは処理結果をオブジェクトで返すことが可能

V call() throws Exception　　タスクを実行し結果を返す。タスクが実行できない場合は例外をスローする。
Callableは抽象メソッドcall()しか持たないのでラムダ式で記述可能。

scheduleAtFixedRate()　　指定された初期遅延の経過後に初めて有効　　第三引数で指定された時間ごとにタスクを実行
scheduleWithFixedDelay()　　指定された初期遅延の経過後に初めて有効　　第三引数で指定された時間に従って遅延した後、タスクを実行
上記2つのメソッドともに第一引数はRunnable型。Callable型を指定するとコンパイルエラー

Runnnableインタフェースのrun()とCallableインタフェースのcall()メソッドの違いと共通点

・各メソッド共に引数を取らない
・call()メソッドは戻り値を返すことが可能、throws Exceptionによりchecked例外のスロー可能
・run()はthrows指定していないのでunchecked例外は任意でスロー可能
・各メソッド共にラムダ式で記述可能
V call() throws Exception
void run()

アトミック　　java.util.concurrent.atomic
スレッドセーフなステートを宣言できる
syncrhonuzdeは処理中は1つのスレッドで実行される、syncrhonuzdeメソッドが行なっている操作は「分割不可能な操作」　これをアトミック操作という
アトミックを保証するメソッドとして、ConcurrentMapインタフェースのputifAbsent()がある。
アトミック操作を簡単に実装するため、アトミックに操作できる値(boolean型、int型、long型、参照型)を表すクラスを提供している。
これらのクラスはロックを制御するコーディングを行うことなく、整数の取得格納加算減算を行うことができる。
AtomicBoolean AtomicInteger AtomicLong AtomicReference

AtomicIntegerのメソッド　　staticなので複数スレッドを立て値を更新すると最後に更新されたものが反映
addAndGet(int delta)　　アドミックに指定された値を現在の値に追加する。戻り値は増分後の値
Boolean compareAndSet(int expect, int update)　　現在の値が第一引数と等しい場合、アトミックに第二引数で指定された値に更新する。
Int incrementAndGet()　　アトミックにインクリメントし、更新値を返す
Int get()　　現在の値を取得する。
Int getAndIncrement()　　アトミックにインクリメ

ントし、更新前の値を返す

パラレルストリーム

SE8より並行処理を行うストリームが導入。
Collectionインタフェースのparallel()メソッドは、ストリームを元にパラレルストリームを返す。

default Stream<E> parallelStream　　Collectionインタフェースで提供　コレクションをソースとしてパラレルストリームを返す
S parallel()　　BaseStreamインタフェースで提供　ストリームをソースとしてパラレルストリームを返す
boolean isParallel()　　BaseStreamインタフェースで提供　このストリームがパラレルストリームであればtrueを返す
S sequential()　　BaseStreamンタフェースで提供　ストリームをソースとして、シーケンシャルストリームを返す

インタフェースなのでnewはできない
変数宣言の際は、今までと同じStreamインタフェースを使用
普通のStreamはシーケンシャルなストリーム処理であるため、実行ごとに同じ結果になる。
パラレルストリームによる処理は、要素を並列に処理を行うため、どの要素から処理されるかは実行時によって異なる。ただしパラレル処理を行うことにより実行時間が短縮される傾向にある
e.g.　Arrays.asList(“a”, “b”, “c”).parallelStream().forEach(s -> System.out.prin(s + “”);
実行するたび結果が変わる
　　Arrays.asList(“a”, “b”, “c”).parallelStream().parallel()でも冗長だが可能
parallel().parallelStream()はコンパイルエラー
　　reduce()を使う際T reduce(T identity,　BinaryOperator<T> accumulator)なので引数と戻り値の型が一致していないといけない
　　事前処理としてmapで要素の文字列を文字数に変換するなどをする

forEachOrdered()はシーケンス、パラレル処理の両方で各要素が検出順に処理されることを保証、パラレル処理の場合はパフォーマンスが低下する可能性がある

findAny(), findFirst()の挙動はパラレルでも同じ、limit(), skip()は先頭から順序に従って処理を行うため、パラレルストリームでは時間を要する

groupingByConcurrnt()メソッドとConcurrentMap()メソッドの利用。

groupingByConcurrnt()　　指定した関数に従って要素をグループ化し、結果をConcurrentMapに格納して返す並行Collecterを返す groupingBy()と同等の処理(要

素のグループ化)を行う。
マルチスレッド環境においてスレッドセーフにグルーピングが行われる。
toCurrentMap()　Mapに蓄積する並行Conllectorを返す。toMap()メソッドと同等の処理(要素を元にマップに変換)を行う。
groupingBy()メソッドとtoMap()メソッドは、戻り値がMapなのに対して、
groupingByConcurrnt()メソッドとConcurrentMap()メソッドの戻り値はConcurrentMapになる
従ってマルチスレッド環境においてスレッドセーフにMap処理を行う場合はこれらのメソッドを使用する

Fork/Joinフレームワーク　試験に出る　java.util.concurrentパッケージ

SE7で追加されたFork/JoinフレームワークではExecutorServiceインタフェースの実装。重い計算を小さなタスクに分散し、複数のスレッドによって並列実行することで、高速に処理することを目的とする

スレッドプール内のスレッドにタスクを分散。その後work-stealingアルゴリズムにより、処理が終わったスレッドはビジー状態の他のスレッドからタスクをスティールすることができる。

Fork/Joinフレームワークの主なインタフェースとクラス

ForkJoinPoolクラス　Fork/Joinタスクを実行するためのExcecutorServiceインタフェースの実装クラス
ForkJoinTaskクラス　ForkJoin内で実行する抽象基底クラス
RecursiveActionクラス　結果を返さない再帰的なForkJoinTaskのサブクラス
RecursiveTaskクラス　結果を生成する再帰的なForkJoinTaskのサブクラス

通常はForkJoinTaskクラスを直接継承したサブクラスを定義するのではなく、RecursiveActionクラスもしくはRecursiveTaskをサブクラス化する。
2つのクラスの使い分けは、結果を返さない処理の場合はRecursiveActionクラスを使用し、結果を返す処理の場合はRecursiveTaskを使用
親クラスであるForkJoinTaskクラスには、分岐/結合処理を行うためのメソッドが用意されている。

ForkJoinTaskクラスの主なメソッド

final ForkJoinTask<V> fork()　このタスクを非同期で実行するための調整を行う
static void invokeAll(ForkJoinTask<?>t1, ForkJoinTask<?> t2)　指定さ

れたタスクをフォークしてパラレルに処理を実行する
final V join()　　計算が完了した後、計算の結果を返す。

ForkJoinPoolクラスの主なメソッド

void execute(ForkJoinTask<?> task)　　このタスクを非同期で実行。処理結果は受け取らない
<T> T invoke(ForkJoinTask<?> task)　　タスクの実行が終了するまで待機し、処理結果をタスクから受け取る。同期呼び出し
<T> ForkJoinTask<T> submit(ForkJoinTask<T> task)　　非同期でタスクを実行し、処理結果をタスクから受け取る

RecursiveActionクラスもしくはRecursiveTaskクラスのcompute()メソッド
処理結果としての戻り値が必要ない場合はRecursiveAction
処理結果としての戻り値が必要な場合はRecursiveTask<V>

RecursiveActionクラスの主なメソッド
protected abstract void compute()　　このタスクによって実行される計算処理を実装する。戻り値はない

RecursiveTaskクラスの主なメソッド
protected abstract V compute()　　このタスクによって実行される計算処理を実装する。戻り値は任意のオブジェクト

各compute()メソッドは抽象メソッドなのでオーバーライドしてタスク処理を実装
Fork/Joinフレームワークではタスクの分割手法として分割統治法を用いる。

RecursiveActionクラス　継承する際は extends RecursiveAction

e.g.　ForkJoinTask<?> task = new hoge()
　　ForkJoinPool pool = new ForkJoinPol;　　タスクの実行はForkJoinPool クラスを使用
　　pool.invoke(task);

RecursiveTaskクラス　継承する際は extends RecursiveTask<T> 型パラメータを使用する

e.g.　ForkJoinTask<T> task = new hoge()　継承した際に指定した型パラメータに合わせる
　　ForkJoinPool pool = new ForkJoinPol;　　タスクの実行はForkJoinPool クラスを使用
　　Double sum = pool.invoke(task);　　戻り値がある

Fork/Joinフレームワークを使用することで、タスクの分割や結果の取得などを簡単に行うことができる。
パラレル処理は分割に伴うオーバーヘッドが発生するので扱う要素数が大きくないとパフォーマンスの改善は見込めない。分割の割合を適切に行わないと同じように改善されない
ストリームAPIでの処理は順次処理(Sequential)もしくは並列処理(Parallel)のいずれかで実行可能
java.util.Collectionのstreamメソッドかparallelを使用したのかで決定する。
ストリーム処理の途中で実行モードを変更することが可能
順次処理から並行処理の変更にはparallelメソッドを使用、並行処理から順次処理の変更にはsequentialメソッドを使用

JDBC

JDBC　APIは以下の2つ
java.sql　　データベースに格納されたデータにアクセスして処理する基本APIを提供。
javax.sql　　データベースにアクセスする際、サーバ側での処理(接続プールや分散トランザクションなど)を行う場合に使用するAPIを提供

JDBCドライバ
Javaプログラムとデータベースを結びつけるのに必要なJDBCインタフェースを実装したクラス

JDBCドライバの種類
タイプ1　JDBC-ODBC ブリッジドライバ　　ODBCドライバを利用できるデータベースにJDBCからのアクセス可能にするためのドライバ
タイプ2　ネイティブブリッジドライバ　　JDBC API　に対する呼び出しを、データベースに付属するネイティブAPIの呼び出しへと変換するタイプのドライバ
タイプ3　ネットドライバ　　JDBC APIの呼び出し、ネットワークを介して中間層のサーバーに転送する。３階層のアプローチで構成されるドライバ
タイプ4　ダイレクトドライバ　　全てJava言語で作られているドライバ　　MySQLドライバはこれに属する

基本的なJDBCアプリケーションの作成

1. java.sqlのインポート　　2. DBの指定　　3.DBとの接続　　4. ステートメントの取得　　5.SQL文の実行　　6.結果の取得と処理　　7.接続のクローズ

java.sqlパッケージの主なクラス・インタフェース
DriverManager　　DBのドライバ管理をし、DBとの接続を支援する
Connection　　特定のDBとの接続(セッション)を表現　　newによるインスタンス化はできないDriverManager.getConnection()を使用

Statement　　静的SQL文を実行し、作成された結果を返すために使用されるオブジェクト
ResultSet　　DBの結果セットを表すDBオブジェクト

DB接続する際の通信先のDBを特定するためのJDBC　URLは以下のとおり
jdbc:<subprotocol>:<subname>e.g.　jsbc:mysql://localhost/hogedb　　ポート番号はデフォルトの3306番を使用するので省略可能
jdbc　　プロトコル、JDBC URLのプロトコルは常にjdbcとなる
<subprotocol>　　DBに接続するためのドライバ独自のプロトコル名。ドライバごとに異なる。データベース製品名をさす
<subname>　　DBを特定するための情報。ホスト名やポート番号、DB名など。subprotocolに準じて異なる

getConnection(String url, String user, String password) throws SQLException
ユーザとパスワードは省略可能
従来のJDBCではClass.forName()メソッドでドライバクラスのロードが必要だったが、JDBCドライバ4.0より自動ロード機能で自動的に検出するようになったため呼び出しは不要になった
Class.forName(“com.mysql.jdbc.Driver”);は記述しなくてもよくなった　ついでに例外処理のClassNotFoundExceptionの処理記述もしなくてよくなった
getConnection()が呼び出されると、JDBCドライバクラス自身がDriverManagerクラスへの登録まで行うようになった

ResultSetで結果を取得した際は必ず１回はnext()を実行しないと行を取り込めない。next()を呼び出さずデータの取り出しを試みるとSQLExceptionが発生
next()は次に行が存在する場合はtrueを返し、ない場合はfalseを返す

NUMBER型を取り出すにはgetBigDecimal()
ResultSetインタフェースではObject型で取り出す、getObject()も提供されている。
instance of でチェックした後にキャストして利用

SQLException

以下の条件下等で発生
ネットワークケーブルの物理的な問題などにより通信が切断した
SQLコマンドのフォーマットが不適切である。
サポートされていない機能を使用した。
存在しない列を参照した

SQLExceptionクラスの主なメソッド
int getErrorCode()　　ベンダ固有の例外コードを取得する　　e.g. 1146
String getSQLState()　　SQLStateを取得する　　e.g. 42S02

SQLException getNextException()　　例外に関連したSQLExceptionオブジェクトにチェーンされた例外を取得する

Iterator<Throwable> iterator()　　チェーンされたSQLExceptionのイテレーターを返す
void setNextException(SQLException ex)　　チェーンの最後にSQLExceptionオブジェクトを追加する

SQLステートメントの実行

Statement	標準的なSQL文を実行
PreparedStatement	プリコンパイルされたSQL文を実行
CallableStatement	ストアドプロシージャを実行

日付はjava.sql.Date
時刻はjava.sql.Time
日付と時刻はjava.sql.Timestamp

Statement

ResultSet executeQuery(String sql)　　SQL文を実行。該当レコードがない場合でもnullにならない
int executeUpdate(String sql)　　データ操作言語(DML)、DDLを指定して実行。戻り値は行数を返し、何も返さないSQL文の場合は０を返す
boolean execute(String sql)　　SQL文の実行結果がResultSetオブジェクトの場合はTrueを、更新行数または結果がない場合はfalseを返す
　　select文以外を引数に指定した場合もfalse（delete, update, insertなど）
　　引数には検索/更新のいずれの処理文も指定が可能
　　getResultSet()でResultSetオブジェクトを取得、getUpdateCount()で更新行数を取得する

execute	select-true	insert-false	update-false	delete-false
executeQuery	select-ResultSet	insert-×	update-×	delete-×
executeUpdate	select-×	insert-int値	update-int値	delete-int値

ResultSetの高度な機能

ResultSetオブジェクトでは、問い合わせ結果のスクロール/相対位置指定、ResultSetオブジェクト上でのデータの挿入・更新も可能
上記を実現するには、各ステートメントの取得時にResultSetインタフェースで提供されている定数を指定

CONCUR_READ_ONLY　　更新できないResultSetオブジェクトの平行処理モードを示す
CONCUR_UPDATABLE　　更新できるResultSetオブジェクトの平行処理モードを示す。レコードの挿入や更新、削除を行うことができる
TYPE_FORWARD_ONLY　　カーソルが順方向にしか移動しないResultSetオブジェクトのタイプを示す
TYPE_SCROLL_INSENSITIVE　　スクロール(順方向、逆方向に移動)可能だが、DBのデータに対して行われた変更を反映しないResultSetオブジェクトのタイプを示す
TYPE_SCROLL_SENSITIVE　　スクロール(順方向、逆方向に移動)可能で、DBの最新の内容を反映するResultSetオブジェクトのタイプを示す

e.g. Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);

ResultSetインタフェースのカーソル移動用のメソッド
boolean absolute(int row)　カーソルをこのResultSetオブジェクト内の指定された行番号に移動　負の数を指定した場合は最終行に移動後指定したぶんだけ後退する
　　正と負何れにしてもレコード数以上を指定するとSQLException　0を指定すると先頭行
boolean relative(int rows)　カーソルを現在位置から正または負の相対行数だけ移動する
boolean next()　　カーソルを現在の位置から1行順方向に移動する
boolean previous()　　カーソルをResultSetオブジェクト内の前の行に移動する
boolean first()　　カーソルをResultSetオブジェクト内の先頭行に移動する
boolean last()　　カーソルをResultSetオブジェクト内の最終行に移動する
void afterLast()　　カーソルをResultSetオブジェクトの最終行の直後に移動　　行の直前に移動するのでgetXXXメソッドで行を取得しようとするとSQLException
void beforeFirst()　　カーソルをResultSetオブジェクトの先頭行の直前に移動　　行の直前に移動するのでgetXXXメソッドで行を取得しようとするとSQLException
　　戻り値がvoidなのでprintln()などで出力しようとするコンパイルエラー
int getRow()　　現在の行の番号を取得する。最初の行は１となる。

ResultSetオブジェクト上でのデータの挿入・更新

ResultSetインタフェースのカーソル更新用のメソッド
void updateString(int columnIndex, String x) throws SQLException　　第1引数で指定された列を、第2引数で指定したString値で更新する

void updateInt(int columnIndex, int x) throws SQLException　第1引数で指定された列を、第2引数で指定したint値で更新する
void updateRow() throws SQLException　更新内容をDBに反映する
void moveToInsertRow() throws SQLException　カーソルを挿入行に移動する
void insertRow() throws SQLException　挿入行の内容をデータベースに挿入する
void deleteRow() throws SQLException　DBから現在の行を削除する
void cancelRowUpdates()
ResultObjectの現在の行に対して行なった更新を全てキャンセル

同じステートメントを用いてSQL文を複数実行した際は、1つ目が終わった段階で接続が一旦閉じられてから2つ目のSQL文が実行されるので、1つ目のSQL文の実行結果を取得しようとするとSQLException
e.g. Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(sql1);
int col = stmt.executeUpdate(sql2);　実行時に一旦接続が一旦閉じられるので上のrsの結果は取得できない
rs.getString(1);　SQLExceptionがスロー

ローカライズとフォーマット

ロケール
国や言語で分けた地域を表す情報プログラムを実行する地域によって表示を変えたい場合などに使用
ロケールはjava.util.Localeクラスのオブジェクト(ロケールオブジェクト)で表す。
ロケールオブジェクトはnewを使用してインスタンス化する以外にLocaleクラスの定数やgetDefault()メソッドからも取得可能

Localeクラスの主なコンストラクタ・メソッド・定数　引数のないコンストラクタは提供されていない
Locale(String language)　引数で指定された言語コードからロケールオブジェクトを生成する
Locale(String language, String country)　引数で指定された言語コード、国コードからロケールオブジェクトを生成する
Locale(String language, String country, String variant)　引数で指定された言語コード、国コードからロケールオブジェクトを生成する
e.g. Locale us = Locale.US //米国
Locale us = Locale.JAPAN
Locale jp = new Locale(“ja”);

Local.Builderクラスのbuild()メソッドを使用することも可能。コンストラクタを使用するLocaleクラスとは異なり、setterメソッドによってLocaleオブジェクトを構成する
e.g. Local locale = new Locale.Builde().setLanguage(“ja”).setScript(“Jpan”).setRegion(“JP”).build();
setLanguage()は言語、setScript()はISO15924で定義されている四文字の文字体系、setRegion()は地域を設定している。build()により、設定された値から作成されたLocale()オブジェクトを取得

getXXXはXXXのコードを返す。getDisplayXXXはXXXの名前を返す
e.g. getCountry()は国コードを返す　getDisplayCountry()は国名を返す
言語コード　日本はja　国コードはJP

リソースハンドル

アプリケーションのUIにおいてロケールがUSの場合は英語でJPの場合は日本語でと自動的に切り替えたい場合リソースハンドルを使用
リソースハンドルはロケール固有のリソース(メニューに表示する文字列など)の集合
APはユーザのロケールに合致するリソースをリソースハンドルから取得
クラスパスが設定されているディレクトリ上にプロパティファイルやクラスファイルを配置する必要がある

java.util.Propertiesクラスでは、プロパティファイルとしてXMLとテキストファイルを扱う

リソースハンドルはjava.util.ResourceBundleクラスのサブクラスである
ListResourceBundle, PropertyResourceBundleクラスを取り扱う
ListResourceBundle　　リソースハンドルをリソースのリストとして管理するクラス　ソース内に記述したリソースからリソースバンドルを作る
　　サブクラスでgetContentsをオーバーライドする必要がある。
PropertyResourceBundle　　リソースハンドルをプロパティファイルで管理するクラス

ListResourceBundleクラスの利用

定義ルール
ListResourceBundleクラスを継承したpublicなクラスを作成する
getContents()メソッドをオーバーライドし、配列でリソースのリストを作る
リソースはキーと値を要素とする配列として作成

protected abstract Object[][] getContents()　　各リソース(キーと値のペア)

をObject型の配列にしてかえす

リソースハンドルの命名規約

デフォルト用と米国用を作った際を考える　パッケージ化は任意
MyResources
MyResources_en_US１
ここではMyResourcesが共通名になっている。これを基底名という
これに加えて、言語コードと国コードを加えてリソースハンドルのファイルを命名する　書き順は　基底名　_　言語コード　_　国コード

作成したリソースバンドルの読み込みにはResourceBundleクラスのstaticメソッドであるgetBundle()を使用

static final ResourceBundle getBundle(String baseName)
引数で指定された名前、デフォルトのロケール、呼び出し側のクラスローダを使用して、リソースハンドルを取得
static final ResourceBundle getBundle(String baseName, Local locale
引数で指定された名前、引数で指定されたロケール、呼び出し側のクラスローダを使用して、リソースハンドルを取得
パッケージ化している場合は　パッケージ名. 基底名とする
上記で取得したリソースハンドルを検索用メソッドを使用する
Set <string> keySet()　　このリソースハンドルに含まれる全てのキーをset型で返す

PropertyResourceBundleクラスの利用

作成ルール
プロパティファイル名は、ListResourceBundleと同様に、基底名、言語コード、国コードの組み合わせとする
拡張子は.propertiesとする
リソースであるキーと値のペアは、プロパティファイル内に「キー＝値」の形式で列記する

PropertyResourceBundleによるリソースハンドルの使用ではサブクラス化は必要ない。プロパティファイルを作成するだけ

デフォルト用と米国用を作った際を考える　キーに対する各値に　P_　を付与　＝もしくは:で区切る　　#, !はコメント
MyResources.properties
send = P_¥u9001¥u4fe1
cancel = P_¥u53d6¥u6d88

MyResources_en_US.properties
send = P_send
cancel = P_cancel
getObject(“send”);で値を取得　引数に数値を用いるとコンパイルエラー

Propertiesクラス

キーの内容を取得する　　プロパティファイルそのものの読み込みには
,FileInputStream or FileReaderを使用し、loadメソッドに渡す
プロパティの値の取得にはgetProperty()を使用 キーが存在しない場合はnullを返す　第二引数を指定するとnullの場合は第二引数が返る
void list(PrintStream out)　指定された出力ストリームにプロパティリストを出力する
void list(PrintWriter out)　指定された出力ストリームにプロパティリストを出力する

InputStream in = new FileInputStream(“MyResources.properties”)
Properties props = new Properties();
props.load(in);
println(props.getProperty(“send”));　sendの値を表示
props.list(System.out);　　プロパティファイルのキーと値を全て表示
props.forEach(k, v) -> System.out.println(k + “” + v);　　プロパティファイルのキーと値を全て表示　forEachはBiConsumer

MyResources.propertiesではボタン名がUNICODEで記述されている。プロパティファイルをISO-8859-1コードで読み込むので表現されない文字(日本語など)は、Unicode変換しておく必要あり
java開発環境で提供されているnative2asciiコマンドを使用すると,Shift-JISなどで記述したプロファイルをUnicodeに変換することができる。
e.g. >native2ascii MyResources.properties_sjis MyResources.properties
sjisをUnicodeに変換

getBundle()で取得するのはListResourceBundleクラスと同じだが指定の仕方が異なる。取得後、getString(“key”)で値の取得

ListResourceBundle　　getBundle(“foo.bar.MyResources”) パッケージ名も記述
PropertyResourceBundle　　getBundle(“MyResources”)　プロパティファイル名のみ　ファイル名は基底名のみで、言語コードと国コード、拡張子は含まない

検索順はクラスファイル→プロパティファイル　ファイル名が長いのが優先

ロケールに対応した適切なリソースハンドルが読み込めない場合は、
MissingResourceException例外が発生

ListResourceBundle, PropertyResourceBundleは同一アプリケーションに混在していてもエラーも警告も出ない

フォーマット

扱うデータによって格納している形式と表示形式を変えたい場合に書式化する

数値、通過、日付、時刻、テキストメッセージをフォーマットするために使用される主なクラス　java.textパッケージで提供
抽象クラスのためnewできない
数値及び通貨　NumberFormat　DecimalFormat　DecimalFormatSymbols
日付及び時刻　DateFormat　SimpleDateFormat　DateFormatSymbols
テキストメッセージ　MessageFormat　ChoiceFormat

NumberFormat オブジェクトを使って数値や通貨をフォーマットするにはformat()をしよう

通貨フォーマットで異なるロケールを指定した場合はParseExceptionが発生