オブジェクト指向

モデリングとシミュレーション

2019年度

目的

- 基本的クラスの利用
  - 講義で使用するライブラリ
- 抽象クラスの利用
  - インターフェースの利用

Javaの豊富なライブラリ

- プログラミングで共通的に必要なライブラリが、言語とともに配布されている。
- 良く使うライブラリを紹介し、利用方法を学ぶ
- マニュアル
  - https://www.oracle.com/technetwork/jp/java/javase/documentation/api-jsp-316041-ja.html

原始型と対応するクラス

- クラスでない型
  - int、double、boolean、char、など
  - ポインタが存在しないことに注意
- 対応するクラス
  - Integer、Double、Boolean、Character、など
- マニュアルを確認

原始型と対応するクラス

- Boxing：原始型からクラスへの代入

```
int x = 1;
Integer y = x;
```

- Unboxing：クラスから原始型への代入

```
Integer y = 1:
int x = y;
```

文字列クラスString

- C/C++のように、文字の配列ではない
- 比較、部分文字列などのメソッド
- マニュアルを確認

抽象クラス

- Abstract Class
  - メソッドの一部が実装されていない
    - abstract キーワードのついたメソッド
  - インスタンスを生成できない
- 継承クラスで抽象メソッドを実装
- 一つのクラスしか継承(extend)できない

抽象クラス

- Interface
  - 少数のメソッドのみを持つ
  - 他のクラスから呼ばれる方法を定義
- 複数のinterfaceを実装(implement)できる
  - 未実装のメソッドを実装

抽象クラス
リストを例に

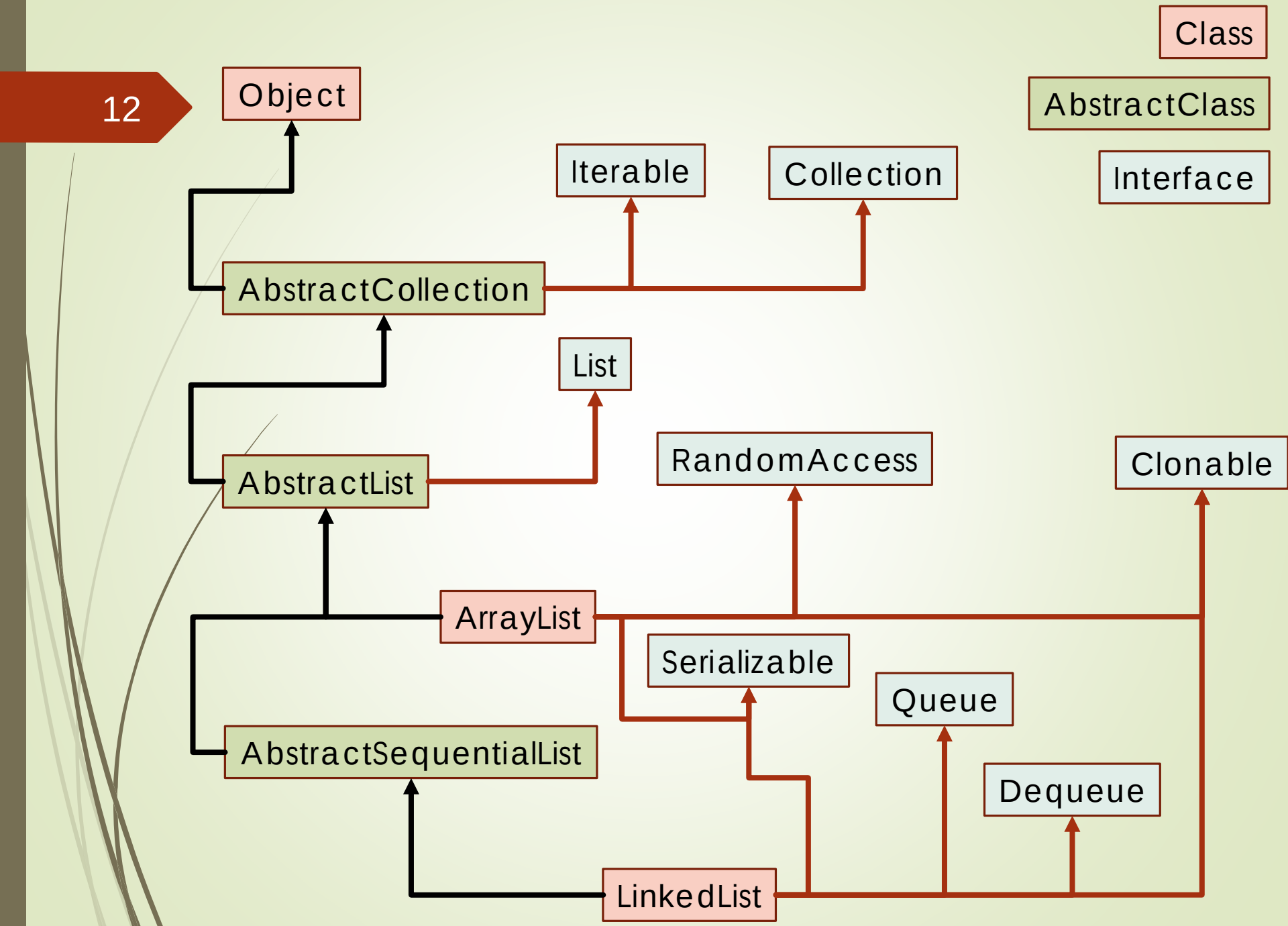
- java.util.List
  - リストを操作するメソッドが定義されたインターフェース
  - リストを操作する方法のみを定義
    - 要素の追加・削除、リストの大きさ
- マニュアルを確認

抽象クラス
リストを例に

- java.util.AbstractList
  - Listインターフェイスを実装
  - ランダムアクセス的リストの抽象クラス
  - getメソッドが実装されていない
- マニュアルを確認

リストの実装クラス

- java.util.ArrayList
  - 検索が高速
  - AbstractListの拡張クラス
- java.util.LinkedList
  - データの挿入・追加が高速
  - AbstractSequentialListの拡張クラス


Class
AbstractClass
Interface
Object
Iterable
Collection
AbstractCollection
List
AbstractList
RandomAccess
Clonable
ArrayList
Serializable
Queue
AbstractSequentialList
Dequeue
LinkedList

リストの利用

- 生成

```
List<T> list = Collections.synchronizedList(new ArrayList<>());
```

  - スレッド間での同期を指定
- 要素追加：list.add(T t);
- 要素取り出し:T t = list.get(int j);
- 要素再設定：list.set(int j, T t);
- 要素の数：list.size()

型パラメタ

- クラスが扱う対象を特定せずにパラメタ化できる

```
public class Store<T> {
    List<T> list;

    public Store(List<T> list){this.list=list;}

    public boolean addData(T t){return list.add(t);}
}
```

Tは特定のクラスではない

型パラメタ

- List<T>のTは、型パラメタ
  - そのリストに格納されるクラスを指定
  - 左辺で型を指定した場合は、右辺では<>として省略できる。

```
//クラスインスタンスのリスト
List<Student> students=
    Collections.synchronizedList(new ArrayList<>());
//配列に登録、及び成績登録
for (int i = 0; i < names.length; i++) {
   Student s =new Student(names[i], i);
   s.setRecord(records[i]);
   students.add(s);
}
```

型パラメタの利点

- クラスの定義時
  - 対象とするクラスを一般化できる
  - 一般的処理を記述できる
- クラス利用時
  - 誤ったクラスインスタンスを渡さない

型パラメタの利点 Listの場合

- List<T> list
- listに保存できるのは、クラスTのインスタンスのみ
- listから取り出したものは、クラスTのインスタンス

インターフェースの例 Comparable

- クラスインスタンスが比較できることを定義
- 必ずcompareToメソッドを有する
  - 自分が対象よりも大きい：正
  - 自分が対象よりも小さい：負
  - 自分が対象と同じ大きさ：ゼロ
  - どのように比較するかを記述する。

- Comparableは
  - IntegerやStringなどに実装済み
- Comparableを前提に整列するメソッドを持つクラスがある

Comparableインターフェースは比較対象を指定する

```
//クラスDataは相互に比較できる
public class Data implements Comparable<Data>{
    private int x;

    public Data(int x){this.x = x;}

    //比較はx の値の大小で行う
    public int compareTo(Data o){
        return this.x – o.x;
    }
}
```

型パラメタを持つメソッド generic method

- 汎用的なメソッド（サブルーチン）
  - 対象となる具体的な型を特定せず、その特性（インターフェース）が分かれば処理できる
  - 例えば、整列は比較できるモノであれば整列できる

- 戻り値の前で型パラメタを定義
- ここでは、TはTと比較できるクラスであること
  - Tのインスタンスはcompare To()を持っている→これを使って比較できる

```
public static <T extends Comparable<T>> void sort(List<T> list) {
    //ソートの実装
}
```

listの各要素がcompare To()を持つことを利用

NetBeans 利用tips

- ソースファイルの自動整形
  - ソースファイル編集画面でマウス右ボタン→「フォーマット」
- 変数名等の変更
  - ソースファイル編集画面でマウス右ボタン→「リファクタリング」→「名前の変更」
  - プロジェクト全体に変更が行われる

- コードの挿入
  - ソースファイル編集画面でマウス右ボタン→「コードの挿入」
  - コンストラクタ、設定メソッド、取得メソッドなど
- メソッドや変数名の自動補完
  - インスタンスの後にピリオド
- 行番号に出るバルーン
  - マウスを合わせてメッセージ表示
  - マウス左ボタンで、対応策提案