O package java.util.function

- Este package possui várias interfaces funcionais que foram adicionadas e cujo objectivo é poderem ser utilizadas para parametrizar as classes através da injecção de comportamento.
- São interfaces funcionais cuja definição contém apenas um método.
  - que é abstracto e será instanciado pelo programador.

(*) seguindo a estrutura apresentada em Functional Interfaces in Java, Ralph Lecessi, 2019

- Considerem-se os quatros tipos básicos de entidades deste package:

Model	Has Arguments	Returns a Value	Description
Predicate	yes	boolean	Tests argument and returns true or false.
Function	yes	yes	Maps one type to another.
Consumer	yes	no	Consumes input (returns nothing).
Supplier	no	yes	Generates output (using no input).

- Estas definições são utilizadas criando-se uma função de um destes tipos de dados e definindo-a utilizando uma expressão lambda.

Predicate<T>

- A interface `Predicate` faz a avaliação de uma condição associada a um valor de entrada que é de um tipo genérico.
- O método devolve true se a condição for verdade, falso caso contrário

```
@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);
}
```

- O programador associa depois um tipo de dados quando define o predicado:

```
Predicate<Integer> p1;
```

- e faz a associação da expressão que representa a condição. Exemplo:

```
p1 = x -> x > 7;
```

- testando a veracidade com a invocação de test

```
if (p1.test(9))
  System.out.println("Predicate x > 7 é verdade para x == 9.");
```

- Claro que uma mais valia desta abordagem é poder passar para um método um predicado
  - invocar o teste do predicado dentro do método

```
public static void result(Predicate<Integer> p, Integer arg) {
   if (p.test(arg))
     System.out.println("O predicado é true para " + arg);
   else
     System.out.println("O predicado é falso para " + arg);
}
```

```
public static void main(String[] args) {
    Predicate<Integer> p1 = x -> x == 5;

    result(p1,5);
    result(y -> y%2 == 0, 5);
}
```

- Num exemplo de uma das aulas práticas em que temos uma Turma de Alunos, podemos definir um método mais geral que permita identificar os alunos que satisfazem determinado predicado:

```
/**
 * Passar um predicado para um método, possibilitando assim a parametrização
 * de comportamento através de um parâmetro.
 * Este método devolve a lista dos alunos que satisfazem o predicado p
 */

public List<Aluno> alunosqueRespeitamP(Predicate<Aluno> p) {
  return this.alunos.values().
              stream().
              filter(a -> p.test(a)).
              map(Aluno::clone).
              collect(Collectors.toList());
}
```

generaliza-se o mecanismo de filtragem

- Definindo agora os predicados podemos obter diversos filtros de informação:
  - não tendo que repetir código
  - parametrizando o comportamento de filtragem pretendido

```
Predicate<Aluno> p = a -> a instanceof AlunoTE;

List<Aluno> alunosTE = t.alunosqueRespeitamP(p);
for (Aluno a: alunosTE)
  System.out.println(a.toString());
```

Function<T,R>

- Esta interface funcional aceita dois tipos de dados, sendo que recebe um parâmetro `T` e devolve um resultado do tipo `R`.

```
@FunctionalInterface
public interface Function<T, R>
{
    R apply(T t);
    ...
}
```

- o método `apply` transforma o objecto do tipo `T` numa resposta do tipo `R`.

- Na definição da Function é necessário associar os tipos de dados e depois definir o seu comportamento através de uma lambda expression

```
Function<String, Integer> f;
f = x -> Integer.parseInt(x);
```

- a utilização faz-se pela invocação na Function do método `apply`

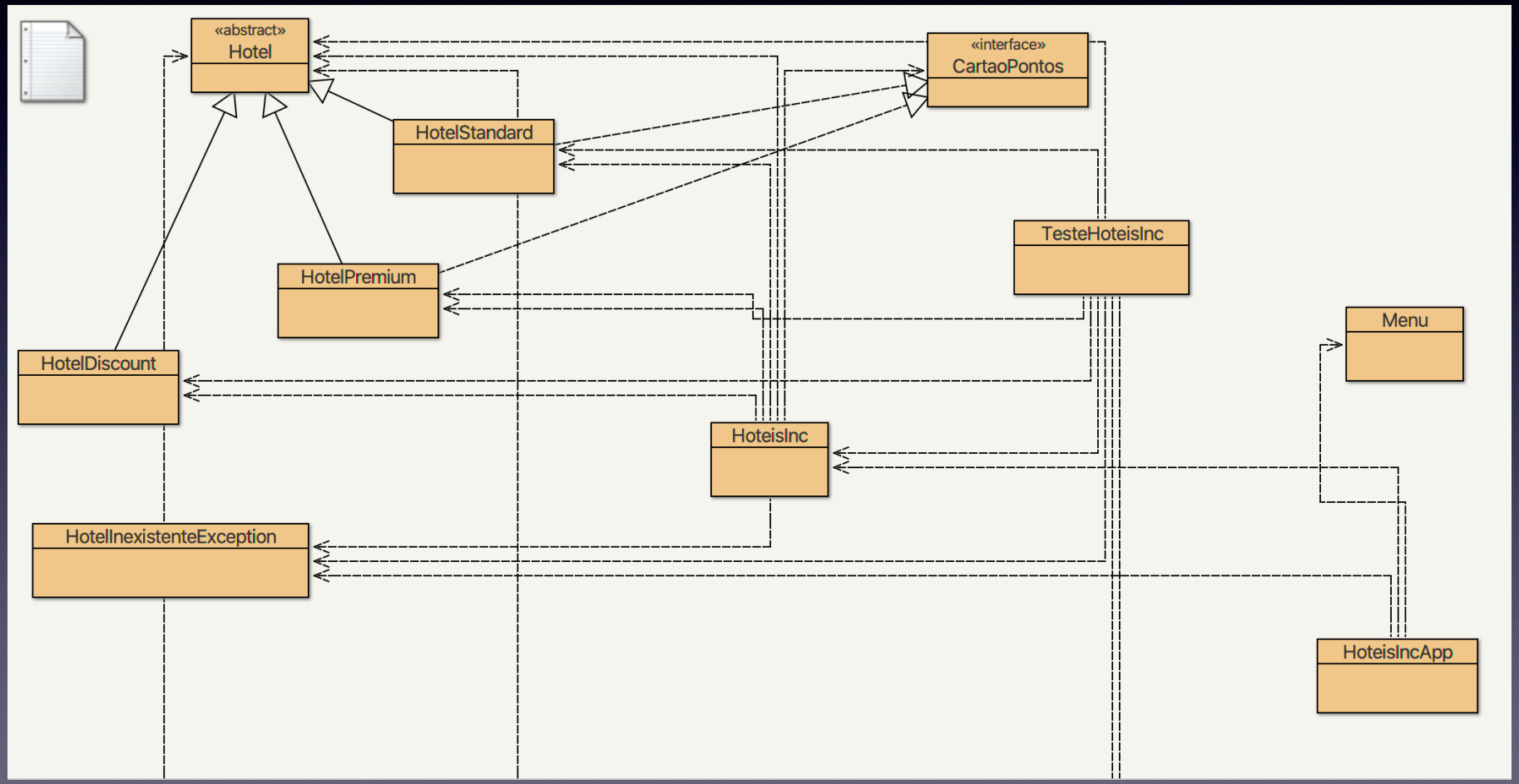
```
Integer i = f.apply("100");
System.out.println(i);

OUTPUT:

100
```

- Como vimos para os predicados é possível passar estas Function como parâmetro

```
public class Transformer {

  private static <T,R> R transform(T t, Function<T,R> f) {
    return f.apply(t);
  }


  public static void main(String[] args) {

    Function<String,Integer> fsi = x -> Integer.parseInt(x);
    Function<Integer,String> fis = x -> Integer.toString(x);

    Integer i = transform("100", fsi);
    String s = transform(200, fis);
    System.out.println(i);
    System.out.println(s);
  }

}
```

- Podemos utilizar a declaração de Function
  - para tornar mais genéricos todos os métodos que fazem travessias a colecções e aplicam uma função
  - evita-se a repetição de código
  - tal é possível derivado do facto de que é permitido passar uma expressão lambda como parâmetro

- Seja o seguinte projecto:



- Consideremos que queremos fazer diferentes métodos:
  - obter o preço de todos os hotéis da cadeia de hotéis
  - listar o nome de todos os hotéis
  - listar o número de estrelas de todos os hotéis
- Todos estes métodos vão ter um código muito semelhante.

- Pode ser definido um método que aplique a função a todos os objectos do tipo Hotel

```java
/**
 * Método que recebe uma Function<T,R> e aplica a todos os
 * hóteis existentes.
 */

public <R> List<R> devolveInfoHoteis(Function<Hotel,R> f) {
  List<R> res = new ArrayList<>();
  for(Hotel h: this.hoteis.values())
    res.add(f.apply(h));


  return res;
}
```

- E, de cada vez, que seja necessário aplicar um novo tipo de selecção de informação criamos uma `Function`

- Exemplo:

```
Function<Hotel,Double> fpreco = h -> h.precoNoite();
Function<Hotel,String> fnome = h -> h.getNome();


List<Double> precos = osHoteis.devolveInfoHoteis(fpreco);
for(Double d: precos)
  System.out.println(d.toString());

List<String> nomes = osHoteis.devolveInfoHoteis(fnome);
for(String d: nomes)
  System.out.println(d.toString());
```

- Existe também a possibilidade de definir funções binárias, do tipo Function<T,U,R>

```
@FunctionalInterface
public interface BiFunction<T,U,R> {
    R apply(T t, U u);
}
```

- apesar de terem tipos T e U, poderão representar o mesmo tipo de dados.

Consumer<T>

- Esta interface é utilizada para processamento de informação
  - não devolve resultado, é `void`, e aplica o método `accept` a todos os objectos

```
@FunctionalInterface
public interface Consumer<T>
{
    void accept(T t);
    ...
}
```

- Podemos também generalizar muitos métodos que fazem travessias e modificam os visitados

```java
/**
 * Método que recebe uma Consumer<T> e aplica a todos os
 * hóteis existentes.
 */

public void aplicaTratamento(Consumer<Hotel> c) {
  this.hoteis.values().forEach(h -> c.accept(h));
}
```

```java
Consumer<Hotel> downgradeEstrelas = h -> h.setEstrelas(h.getEstrelas()-1);
osHoteis.aplicaTratamento(downgradeEstrelas);
```

Supplier<T>

- Supplier é uma interface que é utilizada para gerar informação. Não tem parâmetros e devolve um resultado do tipo T.

```
@FunctionalInterface
public interface Supplier<T>
{
     T get();
}
```

- Pode ser utilizada por exemplo para permitir criar métodos que fazem pretty printing de informação.
  - Criam-se expressões de pretty printing e aplicam-se a todos os objectos
  - Com isto evita-se estar sempre a alterar o método toString.