Java 8:

THE RISE OF LAMBDA

CSci 142
RIT

These slides show how the shortened "lambda" versions of a piece of code relate to the full-blown object-based code.

First, A Small Example

- A nonsense example: shows a very complicated way to say, “Hello”.
- Remember that the `println` method prints objects by invoking their `toString` methods.
- We shall see how to create instances of “instant classes”.

Non-Instant (Named) Class

```
class Something extends Object {
    @Override
    public String toString() {
        return “Hello”;
    }
}

// ====

System.out.println( new Something() );
```

Instant (Anonymous) Class

```
System.out.println( new Object() {
    @Override
    public String toString() {
        return “Hello”;
    }
} );
```

A More Practical Example: Sample Class, Condensed

```
class Person {

    private String name;

    private int age;

    public Person( String n, int a ) { name = n; age = a; }

    @Override public int hashCode() { /* … */ }

    @Override public boolean equals( Object o ) { /* … */ }

    @Override public String toString() {
                return name + '[' + age + ']'; }

    public boolean mayDrink() { return age >= 21; }

}
```

Setup Code

```
Collection< Person > people =
                new HashSet< Person >();

people.add( new Person( "Alice", 20 ) );

people.add( new Person( "Bob", 22 ) );

people.add( new Person( "Carol", 21 ) );

people.add( new Person( "Dan", 18 ) );
```

Action Code

```
for ( Person p: people ) {

    System.out.println( p );

}
```

```
for ( Iterator<Person>  ip=people.iterator();  ip.hasNext();  ) {

    Person  p = ip.next();

    System.out.println(  p );

}
```

```
Dan[18]
Bob[22]
Alice[20]
Carol[21]
```

Functional Interfaces

- How to represent a function in Java, a language that wraps all code in classes?
- Build *functional interfaces* (F.I.).
  - An F.I. only has one abstract method.
  - To define a function, wrap it inside a class that implements an F.I.

Example F.I.: `Consumer`

```
interface Consumer< T > {

    public abstract void accept( T t );

    // Default methods not shown.

}
```

Iterable.forEach needs Consumer

```
interface Consumer< T > {

    public abstract void accept( T t );

}
```

```
interface Iterable< T > {

    public default void forEach( Consumer< T > action ){
        for ( T t: this ) {
            action.accept( t );
        }
    }

}
```

A Printing Consumer

```
class PersonPrinter
          implements Consumer< Person > {

    @Override
    public void accept( Person t ) {

        System.out.println( t );

    }

}
```

Let's rename the parameter!

```
class PersonPrinter
          implements Consumer< Person > {

    @Override
    public void accept( Person p ) {

        System.out.println( p );

    }

}
```

Rewriting Action

```
people.forEach( new PersonPrinter() );

// See forEach in Iterable interface.
```

```
Dan[18]
Bob[22]
Alice[20]
Carol[21]
```

Nested Anonymous Class

```
people.forEach(  new Consumer<Person>()  {
        public  void  accept(  Person  p ) {
            System.out.println(  p );
        }
});
```

```
Dan[18]
Bob[22]
Alice[20]
Carol[21]
```

Lambda #1

```
people.forEach(
                            ( Person  p ) -> {
            System.out.println(  p );
        }
 );
```

```
Dan[18]
Bob[22]
Alice[20]
Carol[21]
```

Lambda #2

```
people.forEach(
                                        p   ->
            System.out.println(  p )


 );
```

```
Dan[18]
Bob[22]
Alice[20]
Carol[21]
```

Lambda #3

```
people.forEach(


            System.out::println


 );
```

(This is called a *method reference*.)

```
Dan[18]
Bob[22]
Alice[20]
Carol[21]
```

Condensing. . .

```
people.forEach( new Consumer<Person>() {
        public void accept( Person p ) {
            System.out.println( p );
}});

people.forEach(
  (Person p) -> { System.out.println(p); } );

people.forEach( p -> System.out.println(p) );

people.forEach( System.out::println );
```

`java.util.stream.Stream`

- If you use a `Stream` you have access to many methods besides `forEach`.
- `Stream`s are sequences of elements.
- They often come out of collections.
- Their elements can be produced, transformed, and consumed.

Using Streams

```
people.stream()
    .forEach(p -> System.out.println(p));
```

Another Functional Interface

```
interface Predicate< T > {
    public abstract boolean test( T t );
    // Default methods not shown.
}
```

Modifying a Stream with a Filter

```
people.stream()
    .filter( p -> p.mayDrink() )
    .forEach( p -> System.out.println(p) );
```

```
Bob[22]
Carol[21]
```

Simplifying, Again

```
people.stream()
    .filter( Person::mayDrink )
    .forEach( System.out::println );
```

```
Bob[22]
Carol[21]
```

Advanced Example (*optional*)

- From the Perp Project
- Read assembly text and generate machine instructions
- Goal: avoid if / else if / …, based on a read-in keyword, to decide what to do with the rest of the line.
- Solution: a Map
  - from keywords already read in
  - to functions that each take the input Scanner and produce the correct instruction.

(Also uses anonymous class with nested initializer.)

```
private static Map< String, Function< Scanner, Machine.Instruction>  > gen
        = new HashMap< String, Function< Scanner, Machine.Instruction  > >()
{{
    put( "PUSH", in -> { int i = in.nextInt();
                         return new Machine.PushConst( i ); } );
    put( "LOAD", in -> { String v = in.next();
                         return new Machine.Load( v ); } );
    put( "STORE", in -> { String v = in.next();
                          return new Machine.Store( v ); } );
    put( "ADD", in -> new Machine.Add() );
    put( "SUB", in -> new Machine.Subtract() );
    put( "MUL", in -> new Machine.Multiply() );
    put( "DIV", in -> new Machine.Divide() );
    put( "NEG", in -> new Machine.Negate() );
    put( "SQRT", in -> new Machine.SquareRoot() );
    put( "PRINT", in -> new Machine.Print() );
}};
```

Advantages of the Approach used in Perp

- Additions are easy.
  - Code to handle each keyword is co-located with the keyword.
  - All such code is in one location.
- No need for a “set of legal keywords”; the key set of this map provides that for free.

Advanced Example (*optional*)

- From the LDOM Project
- It is often the case that to implement a method on a collection of components, you call the same method on each of your components.
- This can be done with loops, but lambda expressions are more concise and less error-prone.
- Some examples follow.

From Class TextSequence

```
public long characterCount() {
    return texts.stream()
            .mapToLong( text -> text.characterCount() )
            .sum();
}

public void replace( String oldS, String newS ) {
    texts.stream()
            .forEach( part -> part.replace( oldS, newS ) );
}
```

Recommendations

- Think carefully about whether or not this abbreviated λ notation clarifies (due to conciseness) or obscures the code's purpose.
- Short λ's are generally good.
- Long λ's should either
  - Call another method to do the work, or
  - Be restated as a full class