Type Inference

…

Definition – Type Inference

Type inference = Java compiler's ability

- To look at each method invocation + corresponding declaration
- In order to determine the type argument (or arguments) that make the invocation applicable

To do this, we use a Type Inference Algorithm…

Definition – Inference Algorithm

- determines the types of the arguments and, if available, the type of the result being assigned, or returned
- tries to find the most specific type that works with all of the arguments

Why “most specific type”

If we didn’t try for most specific, everything would be Object ☺

Type Inference on Generic Methods Invocations

Generally, Java compiler can infer type parameters of generic method calls so you do not have to specify them

Specify type parameter w/ type witness / Explicit Type Parameter as follows;

```
BoxDemo.<Integer>addBox(Integer.valueOf(10), myIntBoxes);
```

…or let it be inferred from method's arguments

```
BoxDemo.addBox(Integer.valueOf(20), myIntBoxes);
```

Similarly…

Type Inference on Constructors Invocations

Consider the following variable declaration:

```
Map<String, List<String>> myMap
        = new HashMap<String, List<String>>();
```

You may substitute the parameterized type of the constructor with an empty set of type parameters (<>):

```
Map<String, List<String>> myMap = new HashMap<>();
```

However, remember, you may **not** omit the <>

```
Map<String, List<String>> myMap = new HashMap();
// unchecked conversion warning
```

n.b. even though it worked w/ gen methods

Let's look at type inference on a constructor

```
class MyClass<X> {
   <T> MyClass(T t) {
      // ...
   }
}
```

The compiler infers formal type parameter T = String

constructor contains a formal type parameter T

Consider the following instantiation of the class MyClass:

```
MyClass obj =
      new MyClass<Integer>("");
```

creates an instance of the parameterized type MyClass<Integer>

specifies the explicit parameter type Integer for the formal type parameter X of generic class MyClass<X>

Target Types - Definition

Java compiler takes advantage of target typing to infer type parameters of a generic method invocation

target type of an expression

- = data type that Java compiler expects

Expected type depends on…

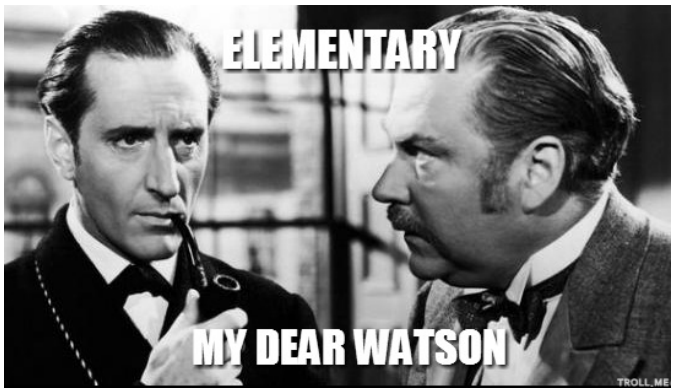
- where the expression appears
- i.e. its context

Example – emptyList(…)

```
static <T> List<T> emptyList();

// Approach #1 - using a type witness to specify T
List<String> listOne = Collections.<String>emptyList();

// Approach #2 - Relying on type inference
List<String> listOne = Collections.emptyList();
```



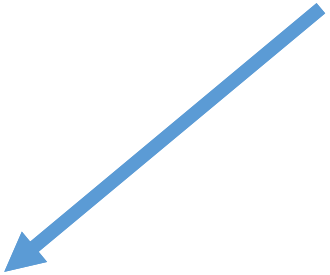
- statement expects instance of List<String>
- This is our target type
- Because emptyList(…) returns List<T>, then T must be String

Another example…

```
void processStringList(List<String> stringList) {
    // process stringList
}
```

Suppose we want to invoke it w/ an empty list

```
processStringList(Collections.emptyList());
```



```
static <T> List<T> emptyList();
```

Let's try this with Java SE 7

We get compiler error message

```
List<Object> cannot be converted to List<String>
```

Why?

- The compiler requires a value for the type argument T
- It starts assuming "Object"
- Consequently, Collections.emptyList returns value of type List<Object>
- This is incompatible with the method processStringList which expects List<String>

Solution = specify type argument

```
processStringList(Collections.<String>emptyList());
```

What about Java SE 8?

no longer necessary to do so

```
processStringList(Collections.emptyList());
```

Why?

- The notion of target type has been expanded to include method arguments

Here is what happens…

- processStringList requires an argument of type List<String>
- method Collections.emptyList returns a value of List<T>
- Using target type List<String>, compiler infers type argument T = String