Type Inference

- Type inference occurs when a compiler or runtime environment can determine the type of an identifier without it being explicitly given by the programmer.
- Important: note the word ‘or’ in the first line!
- Type inference can occur either statically or dynamically.
- In principle, a language can be entirely static and type-safe, and still allow inference to occur.
- Different languages allow different amounts of inference.
  - Scala allows more than Java.
  - Java allows more than it used to.

Type Inference in Java: An Abridged History

- Over time, as type inference has grown more popular, and programmers have become more used to it, Java has introduced an increasing amount of inference.
- Java 5 (2004), introduced generics to Java, based on work by Gilad Bracha, Martin Odersky, and others.
- This allowed (among other things) methods to be specified with a generic type parameter that could be filled in as desired.
- For example, the java.util.Collections class has a method that returns an empty list, of whatever type $T$ we like:

```java
public static <T> List<T> emptyList() {...}
```

Type Inference in Java

- With the introduction of generics, type inference entered the Java language in a real way.
- One can call generic methods with explicit types given:

```java
List<Integer> integers = Collections.<Integer>emptyList();
```

- Or, when the compiler can infer the type of the method call from context, one can omit the explicit type:

```java
List<Integer> integers = Collections.emptyList();
```
Type Inference in Java

- After generics were first introduced (2004), they stayed essentially unchanged until Java 7 (2011)
- Up to that point, creation of instances of generic classes required explicit types on both sides of an instantiation:

```java
HashMap<Integer, ArrayList<Integer>> map = new HashMap<Integer, ArrayList<Integer>>();
```
- After that, the types could be inferred as needed:

```java
HashMap<Integer, ArrayList<Integer>> map = new HashMap<>();
```

Java 8 (2014) added lambda expressions, which also allow type inference, so that a sorting function:

```java
Comparator<Integer> comp = (Integer i1, Integer i2) -> i2 - i1;
```

- Can also be written with types inferred:

```java
Comparator<Integer> comp = (i1, i2) -> i2 - i1;
```

More recently, Java 10 (2018) introduced type inference for local variables, with support for a general `var` type

```java
var i = 3;
var s = "Hello";
```

Local Variable Inference in Java 10+

- The use of the type-inferred `var` designation has the potential to make Java code easier to maintain. For example, if we write the following:

```java
HashMap<Integer, ArrayList<String>> map = new HashMap<>();
// code to fill map here
for (Integer key : map.keySet()) {
    for (String inList : map.get(key)) {
        System.out.println(inList);
    }
}
```

- Re-factoring the `map` means we have to refactor other code that uses it:

```java
HashMap<String, ArrayList<Integer>> map = new HashMap<>();
// code to fill map here
for (Integer key : map.keySet()) {
    for (String inList : map.get(key)) {
        System.out.println(inList);
    }
}
```

Writing the loops using `var` instead of explicit types eliminates some of the refactoring that would otherwise be necessary, as both of the following are now valid:

```java
HashMap<Integer, ArrayList<String>> map = new HashMap<>();
// code to fill map here
for (var key : map.keySet()) {
    for (var inList : map.get(key)) {
        System.out.println(inList);
    }
}
```
Java Type Inference and Conformance

- When doing type inference, the compiler still does static type checking, assuming the most strict type possible.
- Assuming Car and Truck both extend Vehicle, the following is legal:

```java
var vehicle = new Vehicle();
System.out.println(vehicle);
vehicle = new Car();
System.out.println(vehicle);
vehicle = new Truck();
System.out.println(vehicle);
```

- The following, however, is not:

```java
var vehicle = new Car();
System.out.println(vehicle);
vehicle = new Truck();
System.out.println(vehicle);
```

Next Week

- **Topic:** Types, Garbage Collection, Functions
- **Reading:** Chapter 8.4–8.5, 9.1–9.3 of text
- **Schedule change:**
  - No class on Friday, 05 April (next week): MICS conference
- **Office Hours:** Wing 210
  - Monday, 9:00 AM – 10:30 AM
  - Tuesday: 3:00 PM – 4:00 PM
  - Wednesday: 9:00 AM – 10:30 AM
  - Thursday: 2:00 PM – 3:00 PM