(We started this lecture by finishing talking about garbage collection; see the previous notes for this material.)

Parametric polymorphism is a programming language feature with implications for compiler design. The word polymorphism in general means that there are value or perhaps other entities that can have more than one type. We have already talked about subtype polymorphism, in which one type can act like another type. Subtyping places constraints on how we implemented objects. In parametric polymorphism, the ability for something to have more than one shape is by introducing a parameter that is a type.

A typical motivation for parametric polymorphism is support for collection libraries such as the Java Collection Framework. Prior to the introduction of parametric polymorphism, all that code could know about the contents of a `Set` or `Map` was that it contained `Objects`. This led to code that was clumsy and not type-safe. With parametric polymorphism, we can apply a parameterized type such as `Map` to particular types: a `Map⟨String, Point⟩` maps `String`s to `Point`s. We can also parameterized procedures (methods) with respect to types. Using Xi-like syntax less clumsy than Java's, we might write a `is_member` method that can look up elements in a map:

```
contains⟨K, V⟩ (c: Map⟨K, V⟩, k: K): Value { ... }
Map⟨K, V⟩ m
...
p: Point = contains⟨String, Point⟩(m, "Hello")
```

Although we call `Map` a parameterized type, it isn't really a type; it is a type-level *function* that maps a pair of types to a new type. We might denote its signature as `type*type→type`. This function is evaluated at compile time rather than at run time.

When code is parameterized with respect to some type parameter `T`, the name `T` is a new kind of type that we haven't talked about. It represents an unknown type. With ordinary parametric polymorphism we have no information about `T`, and therefore we must generate code that is prepared to handle a value of any type that is usable as an argument. This may be challenging if there are types whose representations have different sizes. For example, in Java we cannot use primitive types as type parameters because types such as `long` do not fit into a word.[1]

Some languages support *constrained parametric polymorphism* in which constraints can be placed on type parameters. The constraint forms a contract between clients using the parameterized abstraction, and the code that implements the abstraction. The compiler can then make use of this information both when type-checking code that uses a type `T` and when generating code for it.

1 C++ templates

2 Polymorphism in ML

3 Java and C#

[1] Limitations of the Java Virtual Machine are also partly to blame