This slide intentionally left blank.

Tricks with Types

How to get fired
with the Java type system.

Shevek
shevek@anarres.org

Compilers

- The job of the compiler is turn your source into binary.
- That's all, right?
- No, it also helps you write correct code.
- The type system is the most significant tool in the arsenal.

Java

- Java is simple.
- Java does not allow language extensions.
- It has primitive types and classes.
- It has single inheritance and interfaces.

- So what can we do?

- It doesn't get interesting yet.

Java 1.5

- Parameterized types.
  - List<X>

```
public interface Foo<X> {
        public void add(X value);
        public X get(int index);
}
```

- Now the compiler can check our code.

```
Foo<String> x = ...;

x.add("bar");  // OK

x.add(5);       // Not OK

String value = x.get(4);       // Note, no cast.
```

Where Can We Use Parameters?

- More places than you think!

```
public class Foo<X> { // Here, we all know.

    @Override
    public <T> T add(Foo<T> remote, T value) { // Also, here!
        ...
    }
}
```

- Now we can say “These two things are of the same type.” without knowing the type!

Java 1.5 Bytecode

- What happens underneath?

```
public interface Foo<X> {
        public void add(X value);    // It's an Object.
}

public class MyFoo implements Foo<String> {
        @Override
        public void add(String value) {  // This can't override (Object)
                ...
        }
}

public class MyFoo implements Foo<String> {

        public void add(String value) {
                ...
        }

        @Override
        public synthetic void add(Object value) {// So this does.
                add((String)value);
        }
}
```

Bounded Parameters

- We can give required properties of the parameter X.

```
public interface Foo<X extends Bar> {
        public void add(X value) {
                // Now we can use the properties of Bar, but not X.
        }
}

public class MyBar extends Bar { }
public class YourBar extends Bar { }

Foo<MyBar>        // Valid
Foo<YourBar>      // Valid
Foo<String>       // Invalid
```

More Power to Type Bounds

- Help us write correct code.

```
public interface MyContainer<X> {
      public List<X> void getValues();
}

MyContainer<String> x = …;
List<String> l = x.getValues();
x.add(“foo”);
```

- Did we just modify an internal data structure?
- Can the compiler help us find out?

```
public interface MyContainer<X> {
      public List<? extends X> void getValues();
}

MyContainer<String> x = …;
List<? extends String> l = x.getValues();
x.add(“foo”);  // Illegal – can't create a value of type unknown.
```

Even More Power to Type Bounds

- We did read-only. Can we do write-only?

```
public interface MyContainer<X> {
      public List<? super X> void getTarget();
}

MyContainer<String> x = …;
List<String> l = x.getTarget();
x.add(“foo”);  // We're allowed to add Strings, or anything below.
x.get(...);       // Illegal, since we don't know the return type.
```

What Does a Bound Tell Us?

- It doesn't tell us the type, just the properties.
- We can have multiple bounds!

```
public interface MyContainer {
        public <T extends JComponent & MyPanel> void add(T panel) {
                // Now we can use the properties of JComponent
                // and MyPanel.
        }
}
```

- Now, we specified multiple behaviours in a language with only single inheritance!
- I forget what bytecode it compiles here.

Types Are Powerful

- Types are the primary tool for the compiler to prove correctness of code.
- If you used a cast, you did something wrong.
- Say what you mean, and the rest will follow.

Thank you

Guh.....? What just happened?