Simplicity *itself*

Dynamic and static Groovy

Untyped variables

```
class Person {
    def name
    def dob

    def ageOn(date) {
        def diffYears = date.getAt(YEAR) - dob.getAt(YEAR)
        def dobInTargetYear = dob.updated(year: date[YEAR])
        return dobInTargetYear > date ? diffYears - 1 : diffYears
    }
}
```

No compilation error even though type of *dob* is unknown

We assume that *date* and *dob* are of the required types

Methods and properties are resolved at runtime. If they exist on the target objects, the code runs. The type does not need to be known in advance.

You can also define state and behaviour at runtime. Or interpret property access and method calls however you want. Some examples follow...

Expando

```groovy
import static java.util.Calendar.YEAR

def person = new Expando(name: "Joe Bloggs")
person.dob = Date.parse("yyyy-MM-dd", "1980-01-18")

person.ageOn = { Date date ->
    def diffYears = date.getAt(YEAR) - dob.getAt(YEAR)
    def dobInTargetYear = dob.updated(year: date[YEAR])
    return dobInTargetYear > date ? diffYears - 1 : diffYears
}

println person.ageOn(new Date() - 800)
```

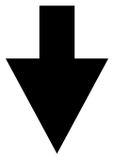
We're creating *name* and *dob* properties on the fly, as well as an *ageOn()* method. *Expando* is a fully dynamic object, just as in JavaScript.

Expando

- Properties can be added via
  - a setter
  - named arguments constructor
- Methods are properties with *closure* values
- Note that `myClosure()` is shorthand for `myClosure.call()`

MarkupBuilder

```groovy
import groovy.xml.MarkupBuilder

def xml = new MarkupBuilder()
xml.institution {
    courses {
        course title: "Computer Science"
    }
    students {
        student "Joe Bloggs"
        student "Jane Doe"
    }
}
```



```xml
<institution>
  <courses>
    <course title='Computer Science' />
  </courses>
  <students>
    <student>Joe Bloggs</student>
    <student>Jane Doe</student>
  </students>
</institution>
```

MarkupBuilder

- Method calls translate to elements
- Named arguments translate to attributes
- Nested closures become nested elements
- Normal arguments translate to element content

How do they work?

If Groovy can't find the target method or property on an object, it will attempt to call `methodMissing()` or `propertyMissing()` as a fallback option. It will only throw a `Missing*Exception` if those fallbacks don't exist.

Both `Expando` and `MarkupBuilder` implement those two methods, although their behaviour is markedly different.

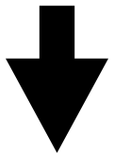
Example

```
class PrintCalls {
    void hi() { println "Hi!" }

    def methodMissing(String name, args) {
        println "Calling ${name}(${args})"
    }

    def propertyMissing(String name) {
        println "Fetching property ${name}"
    }

    void propertyMissing(String name, value) {
        println "Setting property ${name} to ${value}"
    }
}
```

The method signatures should be exactly as shown, otherwise they may not work!

Example

```
def p = new PrintCalls()
p.hi()
p.bark()
p.multiply(2, 6.0)
p.status
p.status = "Done"
```



Note how `methodMissing()` is *not* called if the target method exists

```
Hi!
Calling bark([])
Calling multiply([2, 6.0])
Fetching property status
Setting property status to Done
```

Adding behaviour at runtime

- Categories (via the use() function)
  - http://mrhaki.blogspot.co.uk/2009/09/groovy-goodness-use-categories-to-add.html
- Extension methods
  - http://mrhaki.blogspot.co.uk/2013/01/groovy-goodness-adding-extra-methods.html
- ExpandoMetaClass
  - http://mrhaki.blogspot.co.uk/2009/11/groovy-goodness-add-methods-dynamically.html

Adding behaviour at runtime

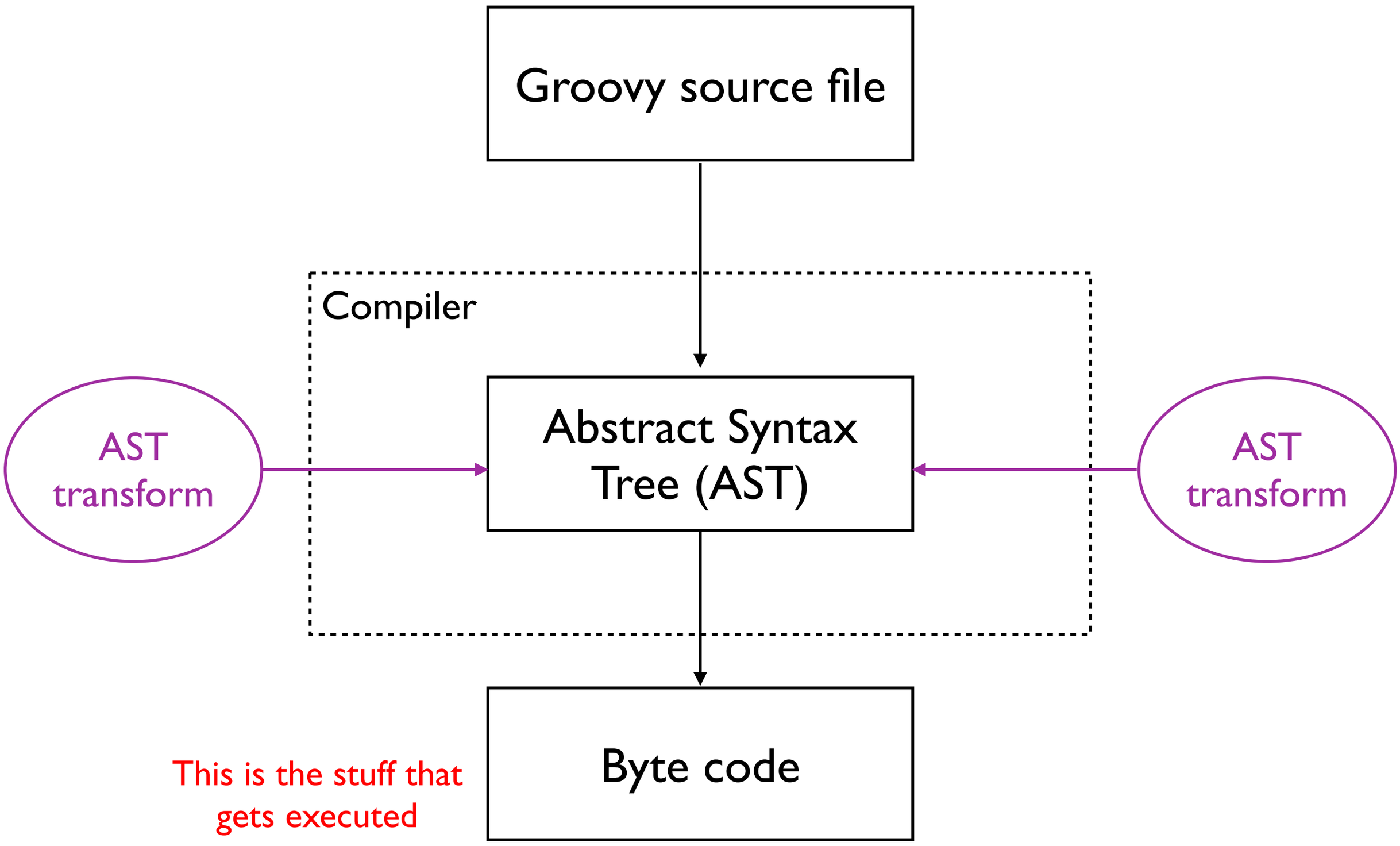
You won't be using these techniques in this course, but it's useful to know about them. Once you're more experienced, you can experiment with them.

Groovy uses the extension method mechanism for the Groovy JDK. If you're interested in seeing the implementations of the methods you've been using, go here:

https://github.com/apache/incubator-groovy/blob/master/src/main/org/codehaus/groovy/runtime/DefaultGroovyMethods.java

Note the `DGM_LIKE_CLASSES` field, which lists the other extension method classes that form the Groovy JDK.

Groovy compilation



AST transforms are applied to the AST of the source code in order to modify the generated byte code.

They can add properties and methods, inject code into methods, change which interfaces and classes are implemented and extended, and more.

You won't be writing AST transforms, but you will be using some existing ones.

Example AST Transforms

```groovy
import groovy.transform.Canonical

@Canonical
class Book {
    String title
    String author
}

def book = new Book("Misery", "Stephen King")

println book

assert book == new Book("Misery", "Stephen King")
```

Positional parameter constructors created

Custom `toString()` implementation added

Custom `equals()` and `hashCode()` implementations (that compare all declared properties)

`@Canonical` is an aggregation of `@ToString`, `@TupleConstructor` & `@EqualsAndHashCode`

Example AST Transforms

```
@Singleton
class BookService {
    List listBookTitles() {
        return ["Misery"]
    }
}

try { new BookService() }
catch (RuntimeException ex) {}

BookService.instance.listBookTitles()
```

Constructor is made private, so you can't instantiate the class

Static `instance` property added that is an instance of `BookService`

The singleton pattern is generally a poor solution as it makes testing hard and is just a special case of a global variable. Use with care.

Example AST Transforms

```
class HomePage {
    final String title = "Google Search"
    @Lazy String content = {
        new URL("https://www.google.com").text
    }()
}

def page = new HomePage()
println page.title

println page.content
```

Parentheses are required here!

Does not require an internet connection

Now an internet connection is required as `content` is initialised

`@Lazy` defers initialisation of properties until they are first used. It is not thread-safe by default but does become so if you add the `volatile` keyword to the declaration

Other interesting annotations

- `@Grab` - use in scripts to add 3rd-party libraries to the classpath
- `@Delegate` - implements the delegate/ composition pattern
- `@Immutable` - like `@Canonical`, but makes the objects immutable
- `@PackageScope` - for when you want Java's default visibility scope

Static type checking

Groovy is a dynamic language by default, but you can configure it to perform compile time type checking just as Java does. IDEs support this so that typos and other type errors appear as compilation errors as you type.

@TypeChecked

groovy.transform.TypeChecked

```
@TypeChecked
class BrokenCode {
    /**...*/
    int sumNumbers(numbers) {
        def result = 0
        for (int i in number) {
            result += i
        }
        return result
    }

    /**...*/
    List reverseStrings(List strings) {
        def result = []
        for (str in strings) {
            result << str.reverse()
        }
        return result
    }
```

Error! No known variable ‘number’

Error! Compiler doesn’t know what the ‘strings’ list contains

Requirements

- Properties and methods must be typed!
- You can use `def` for local variables if they are initialised at declaration time
  - unlike Java, Groovy can *infer* types
- You can't use classes that rely on dynamic behaviour, e.g. `Expando` and `MarkupBuilder`
  - mark such classes/methods with `@CompileDynamic`

Example

```
int textLength(text) {
    return text.size()
}
```

Type of `text` is unknown

```
int textLength(String text) {
    return text.size()
}
```

Adding a type fixes the error

Example

```
int sumNumbers(numbers) {
    def result
    for (i in numbers) {
        result += i
    }
    return result.toInteger()
}
```

Type of `result` is unknown

```
int sumNumbers(numbers) {
    def result = 0
    for (int i in numbers) {
        result += i
    }
    return result.toInteger()
}
```

Initialising to an integer means `result` is an integer

Generics

- Sometimes known as *parameterised types*
- Think `List<String>` (a list of strings)
- Extra information for the compiler
  - it knows what type a collection contains
- Use for certain local variables
  - `List<String> names = []`

Generics don't just apply to the Java collections, but they are probably the most common use case. The parameter (the type inside the angle brackets) is part of the type.

Here are more examples:

- `List<Integer>`
- `Map<String, Long>` (multiple parameters)
- `Comparator<String>`
- `Set<Comparator<String>>` (nested types)

Two annotations

- `@TypeChecked`
  - compiler checks types
  - same byte code as normal Groovy
- `@CompileStatic`
  - compiler checks types
  - different byte code (smaller and faster)
- Ideally only use `@CompileStatic` for performance hotspots