# Agile Software Development

MSc in Computer Science

Produced by

Eamonn de Leastar
edeleastar@wit.ie

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

http://elearning.wit.ie

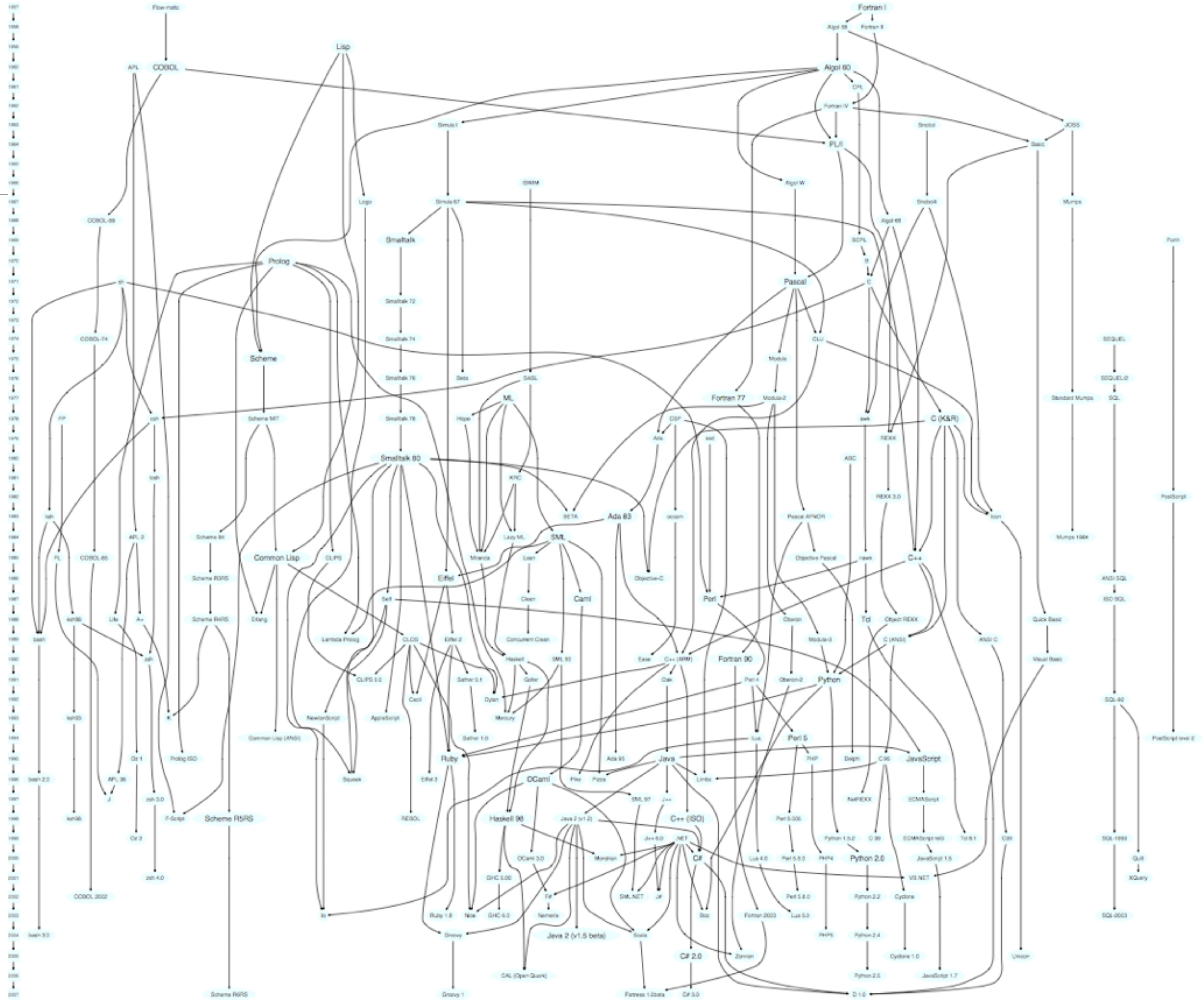
Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

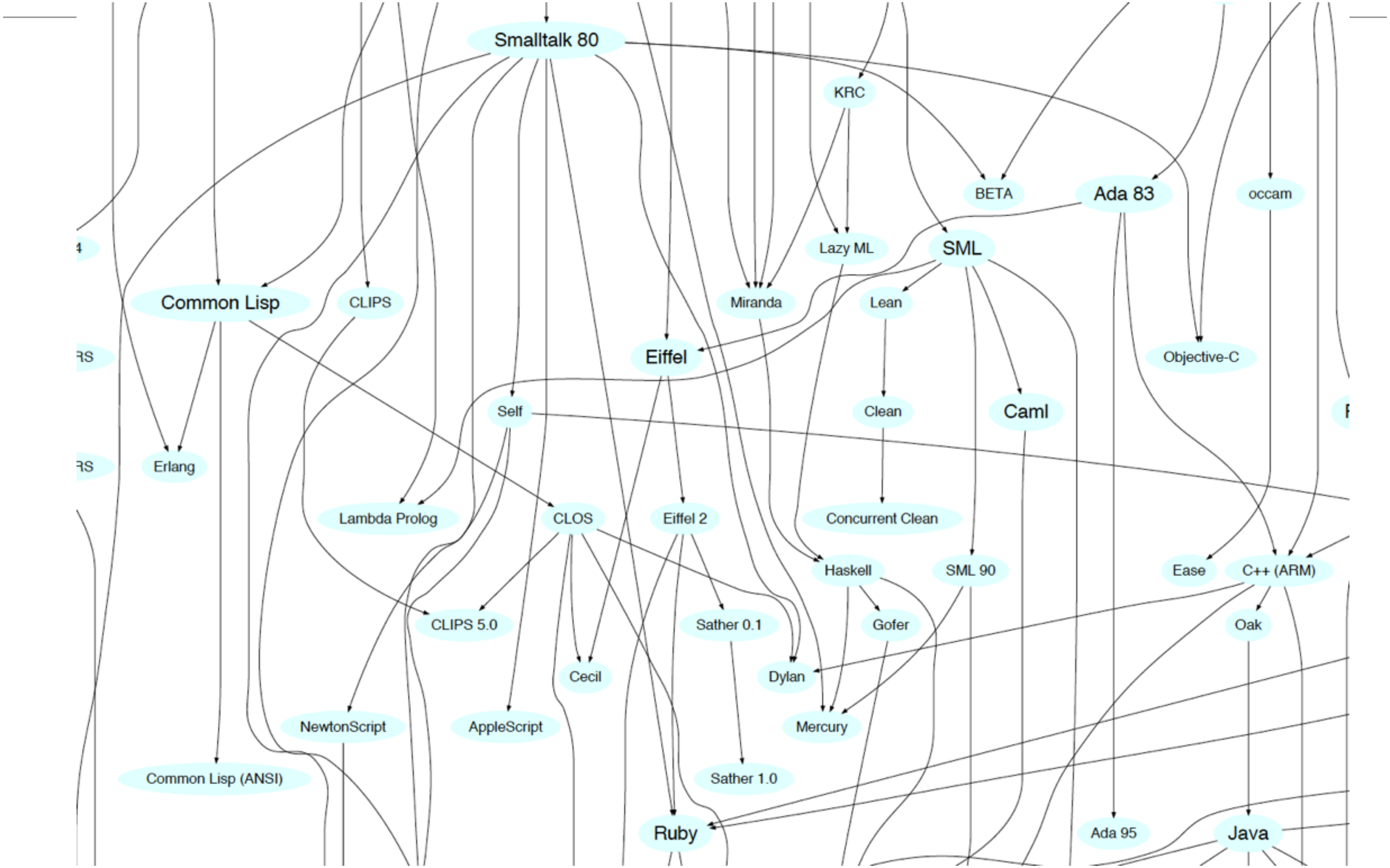
# Programming Language Wish Lists

Agile Software Development

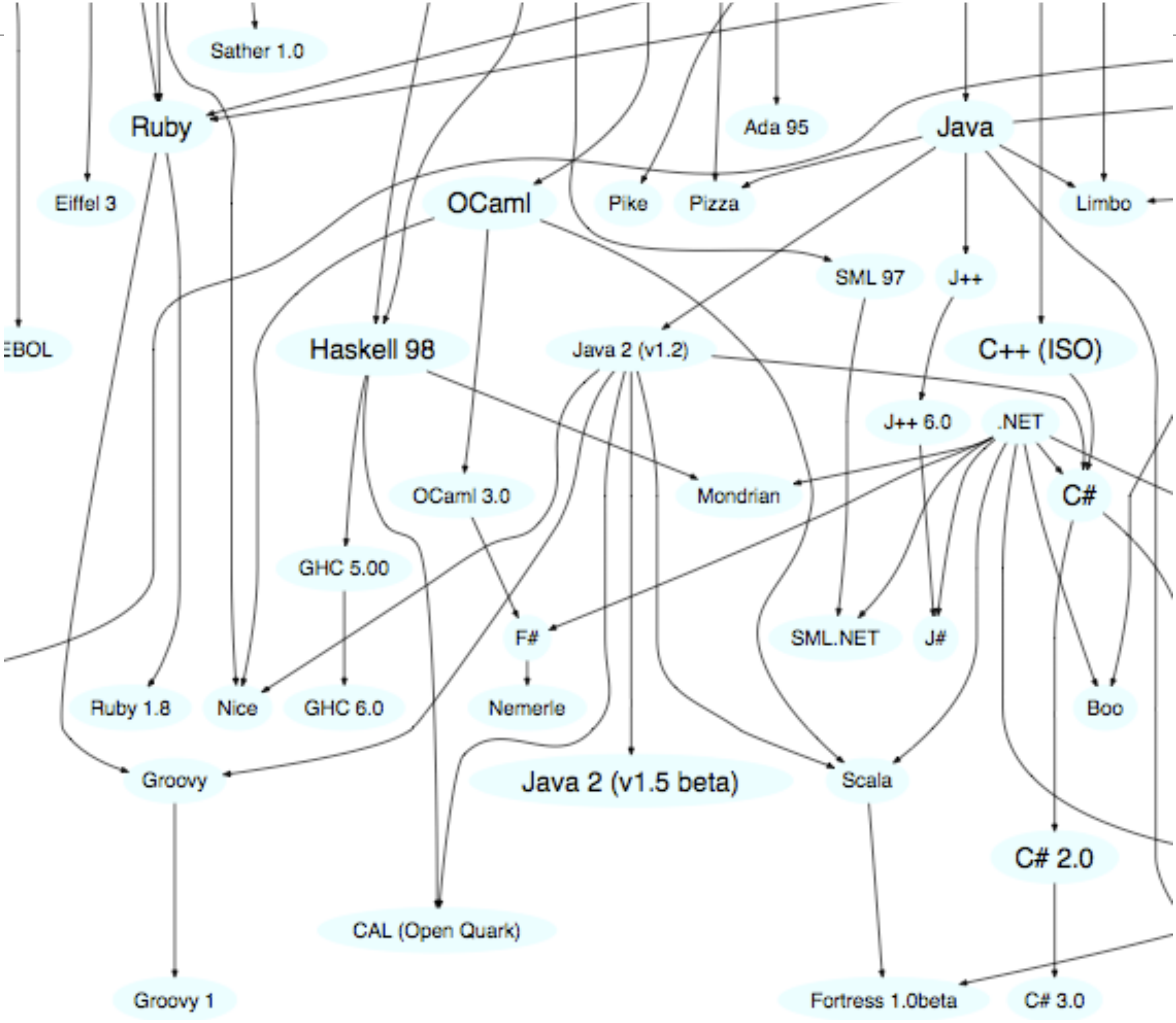
# Family Tree (3)



# Smalltalk Cluster



# Ruby, Groovy, Java, Scala Cluster



# Paul Grahams Wish List for a Programming Language

http://www.paulgraham.com/diff.html

1. Conditionals
2. A function type
3. Recursion
4. Dynamic typing
5. Garbage collection
6. Programs composed of expressions
7. A symbol type
8. A notation for code using symbols and trees
9. The whole language there all the time

Lisp programming Language has all of these features (since mid 1960's)

# Java?

1. Conditionals
2. A function type (but coming in Java 8)
3. Recursion
4. Dynamic typing
5. Garbage collection
6. Programs composed of expressions
7. A symbol type
8. A notation for code using symbols and trees
9. The whole language there all the time

# Groovy/Ruby/Python/Scala/Xtend

(from Neal Ford)

1. Conditionals
2. A function type
3. Recursion
4. Dynamic typing (or Type Inference)
5. Garbage collection
6. Programs composed of expressions
7. A symbol type
8. A notation for code using symbols and trees
9. The whole language there all the time

+ Metaprogramming

# Groovy/Ruby/Python/Scala

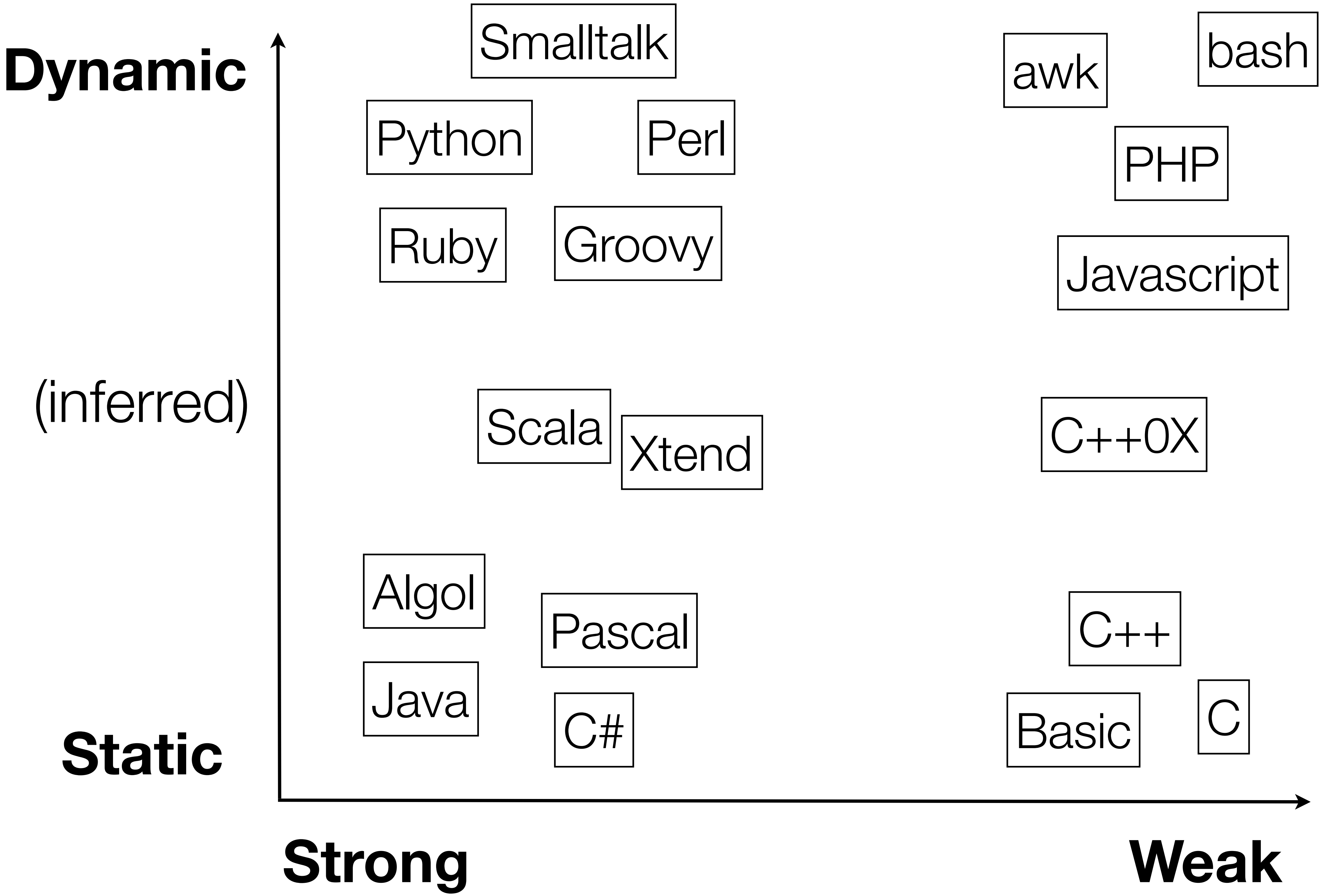
(from Neal Ford)

1. Conditionals
2. A function type
3. Recursion
4. Dynamic typing (or Type Inference)
5. Garbage collection
6. Programs composed of expressions
7. A symbol type
8. A notation for code using symbols and trees
9. The whole language there all the time

+ Metaprogramming

# Typing

# Typing Spectrum

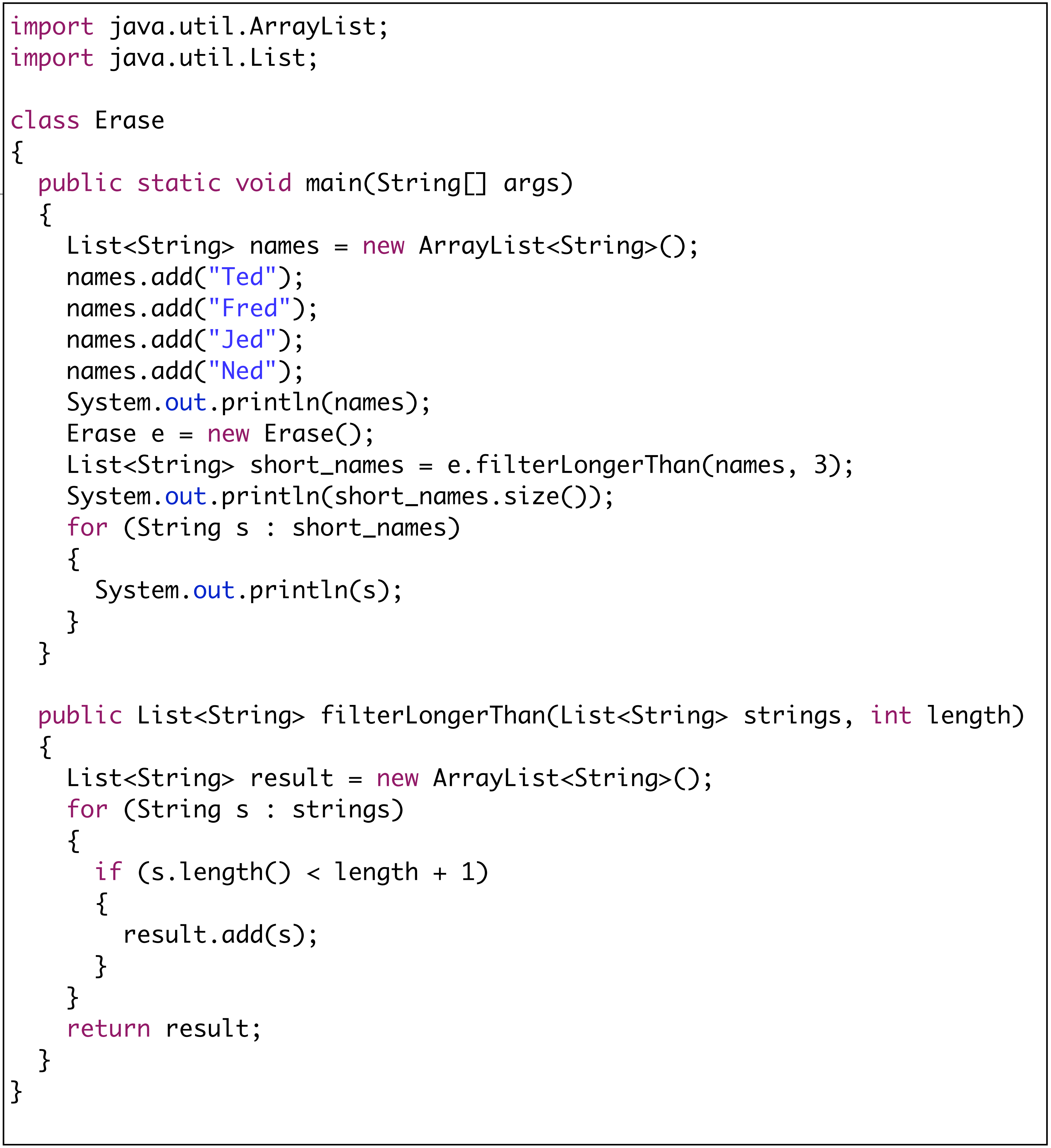


# Static to Dynamic

# Java Example

(from Jim Weirich)

- Java algorithm to filter a list of strings
- Only printing those shorter than 3 (in this test case).



```java
import java.util.ArrayList;
import java.util.List;

class Erase
{
  public static void main(String[] args)
  {
    List<String> names = new ArrayList<String>();
    names.add("Ted");
    names.add("Fred");
    names.add("Jed");
    names.add("Ned");
    System.out.println(names);
    Erase e = new Erase();
    List<String> short_names = e.filterLongerThan(names, 3);
    System.out.println(short_names.size());
    for (String s : short_names)
    {
      System.out.println(s);
    }
  }

  public List<String> filterLongerThan(List<String> strings, int length)
  {
    List<String> result = new ArrayList<String>();
    for (String s : strings)
    {
      if (s.length() < length + 1)
      {
        result.add(s);
      }
    }
    return result;
  }
}
```

# Groovy 1

- Also a valid Groovy program...

```
import java.util.ArrayList;
import java.util.List;

class Erase
{
  public static void main(String[] args)
  {
    List<String> names = new ArrayList<String>();
    names.add("Ted");
    names.add("Fred");
    names.add("Jed");
    names.add("Ned");
    System.out.println(names);
    Erase e = new Erase();
    List<String> short_names = e.filterLongerThan(names, 3);
    System.out.println(short_names.size());
    for (String s : short_names)
    {
      System.out.println(s);
    }
  }

  public List<String> filterLongerThan(List<String> strings, int length)
  {
    List<String> result = new ArrayList<String>();
    for (String s : strings)
    {
      if (s.length() < length + 1)
      {
        result.add(s);
      }
    }
    return result;
  }
}
```

# Groovy 1

- Do we need generics?
- What about semicolons...
- Should standard libraries be imported?

```java
import java.util.ArrayList;
import java.util.List;

class Erase
{
  public static void main(String[] args)
  {
    List<String> names = new ArrayList<String>();
    names.add("Ted");
    names.add("Fred");
    names.add("Jed");
    names.add("Ned");
    System.out.println(names);
    Erase e = new Erase();
    List<String> short_names = e.filterLongerThan(names, 3);
    System.out.println(short_names.size());
    for (String s : short_names)
    {
      System.out.println(s);
    }
  }

  public List<String> filterLongerThan(List<String> strings, int length)
  {
    List<String> result = new ArrayList<String>();
    for (String s : strings)
    {
      if (s.length() < length + 1)
      {
        result.add(s);
      }
    }
    return result;
  }
}
```

# Groovy 2

```
class Erase
{
  public static void main(String[] args)
  {
    List names = new ArrayList()
    names.add("Ted")
    names.add("Fred")
    names.add("Jed")
    names.add("Ned")
    System.out.println(names)
    Erase e = new Erase()
    List short_names = e.filterLongerThan(names, 3)
    System.out.println(short_names.size())
    for (String s : short_names)
    {
      System.out.println(s)
    }
  }

  public List filterLongerThan(Liststrings, length)
  {
    List result = new ArrayList();
    for (String s : strings)
    {
      if (s.length() < length + 1)
      {
        result.add(s)
      }
    }
    return result
  }
}
```

# Groovy 2

- Do we need the static types?
- Must we always have a main method and class definition?
- Consistency (size or length)?

```
class Erase
{
  public static void main(String[] args)
  {
    List names = new ArrayList()
    names.add("Ted")
    names.add("Fred")
    names.add("Jed")
    names.add("Ned")
    System.out.println(names)
    Erase e = new Erase()
    List short_names = e.filterLongerThan(names, 3)
    System.out.println(short_names.size())
    for (String s : short_names)
    {
      System.out.println(s)
    }
  }

  public List filterLongerThan(Liststrings, length)
  {
    List result = new ArrayList();
    for (String s : strings)
    {
      if (s.length() < length + 1)
      {
        result.add(s)
      }
    }
    return result
  }
}
```

# Groovy 3

```
def filterLongerThan(strings, length)
{
  List result = new ArrayList();
  for (String s : strings)
  {
    if (s.length() < length + 1)
    {
      result.add(s)
    }
  }
  return result
}

List names = new ArrayList()
names.add("Ted")
names.add("Fred")
names.add("Jed")
names.add("Ned")
System.out.println(names)
List short_names = filterLongerThan(names, 3)
System.out.println(short_names.size())
for (String s : short_names)
{
  System.out.println(s)
}
```

# Groovy 3

- Should we have a special notation for lists?

- And special facilities for list processing?

```
def filterLongerThan(strings, length)
{
  List result = new ArrayList();
  for (String s : strings)
  {
    if (s.length() < length + 1)
    {
      result.add(s)
    }
  }
  return result
}

List names = new ArrayList()
names.add("Ted")
names.add("Fred")
names.add("Jed")
names.add("Ned")
System.out.println(names)
List short_names = filterLongerThan(names, 3)
System.out.println(short_names.size())
for (String s : short_names)
{
  System.out.println(s)
}
```

# Groovy 4

```groovy
def filterLongerThan(strings, length)
{
  return strings.findAll {it.size() <= length}
}

names = ["Ted", "Fred", "Jed", "Ned"]
System.out.println(names)
List short_names = filterLongerThan(names, 3)
System.out.println(short_names.size())
short_names.each {System.out.println(it)}
```

# Groovy 4

- Method needed any longer?
- Is there an easier way to use common methods (e.g. println)?
- Are brackets always needed?

```
def filterLongerThan(strings, length)
{
  return strings.findAll {it.size() <= length}
}

names = ["Ted", "Fred", "Jed", "Ned"]
System.out.println(names)
List short_names = filterLongerThan(names, 3)
System.out.println(short_names.size())
short_names.each {System.out.println(it)}
```

# Groovy 5

```
names = ["Ted", "Fred", "Jed", "Ned"]
println names
short_names = names.findAll{it.size() <= 3}
println short_names.size()
short_names.each {println it}
```

# Java vs Groovy?

```java
import java.util.ArrayList;
import java.util.List;

class Erase
{
  public static void main(String[] args)
  {
    List<String> names = new ArrayList<String>();
    names.add("Ted");
    names.add("Fred");
    names.add("Jed");
    names.add("Ned");
    System.out.println(names);
    Erase e = new Erase();
    List<String> short_names = e.filterLongerThan(names, 3);
    System.out.println(short_names.size());
    for (String s : short_names)
    {
      System.out.println(s);
    }
  }

  public List<String> filterLongerThan(List<String> strings, int length)
  {
    List<String> result = new ArrayList<String>();
    for (String s : strings)
    {
      if (s.length() < length + 1)
      {
        result.add(s);
      }
    }
    return result;
  }
}
```

```groovy
names = ["Ted", "Fred", "Jed", "Ned"]
println names
short_names = names.findAll{it.size() <= 3}
println short_names.size()
short_names.each {println it}
```

# Another Approach to Types?

- *Type Inference* : the compiler draws conclusions about the types of variables based on how programmers use those variables.
    - Yields programs that have some of the conciseness of Dynamically Typed Languages
    - But - decision made at *compile time*, not at *run time*
    - More information for static analysis - refactoring tools, complexity analysis. bug checking etc...
- Haskell, Scala, **Xtend**

```
object InferenceTest1 extends Application
{
  val x = 1 + 2 * 3         // the type of x is Int
  val y = x.toString()      // the type of y is String
  def succ(x: Int) = x + 1  // method succ returns Int values
}
```

# Java Example -> **XTend**

- Unlike Groovy - this is NOT an XTend Program

```java
import java.util.ArrayList;
import java.util.List;

class Erase
{
  public static void main(String[] args)
  {
    List<String> names = new ArrayList<String>();
    names.add("Ted");
    names.add("Fred");
    names.add("Jed");
    names.add("Ned");
    System.out.println(names);
    Erase e = new Erase();
    List<String> short_names = e.filterLongerThan(names, 3);
    System.out.println(short_names.size());
    for (String s : short_names)
    {
      System.out.println(s);
    }
  }

  public List<String> filterLongerThan(List<String> strings, int length)
  {
    List<String> result = new ArrayList<String>();
    for (String s : strings)
    {
      if (s.length() < length + 1)
      {
        result.add(s);
      }
    }
    return result;
  }
}
```

# def & var

```
import java.util.ArrayList;
import java.util.List;

class Erase
{
  def static void main(String[] args)
  {
    var List<String> names = new ArrayList<String>();
    names.add("Ted");
    names.add("Fred");
    names.add("Jed");
    names.add("Ned");
    System.out.println(names);
    var Erase e = new Erase();
    var List<String> short_names = e.filterLongerThan(names, 3);
    System.out.println(short_names.size());
    for (String s : short_names)
    {
      System.out.println(s);
    }
  }

  def List<String> filterLongerThan(List<String> strings, int length)
  {
    var List<String> result = new ArrayList<String>();
    for (String s : strings)
    {
      if (s.length() < length + 1)
      {
        result.add(s);
      }
    }
    return result;
  }
}
```

*Are semicolons necessary?*

# No semicolons

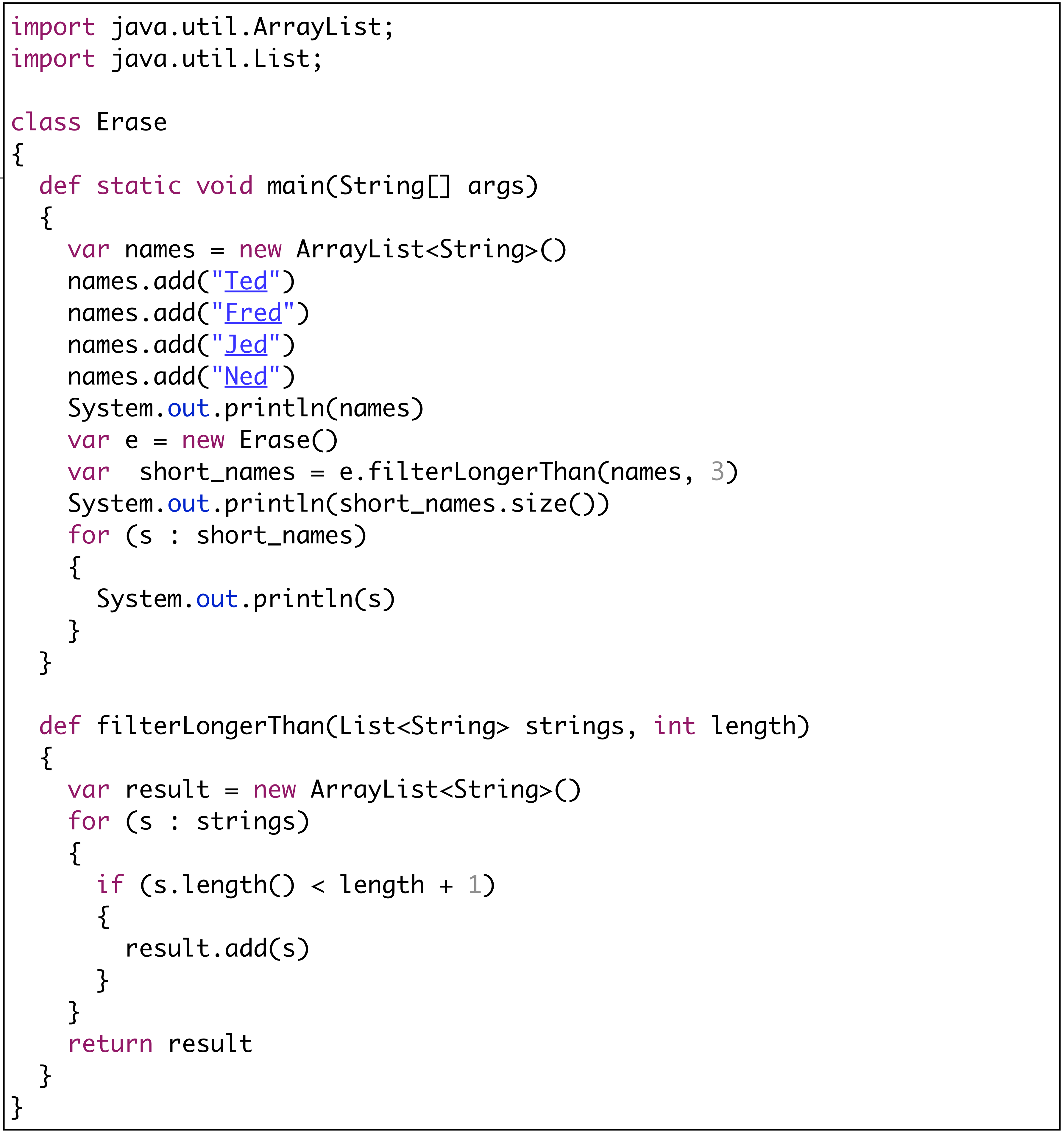
```
import java.util.ArrayList;
import java.util.List;

class Erase
{
  def static void main(String[] args)
  {
    var List<String> names = new ArrayList<String>()
    names.add("Ted")
    names.add("Fred")
    names.add("Jed")
    names.add("Ned")
    System.out.println(names)
    var Erase e = new Erase()
    var List<String> short_names = e.filterLongerThan(names, 3)
    System.out.println(short_names.size())
    for (String s : short_names)
    {
      System.out.println(s)
    }
  }

  def List<String> filterLongerThan(List<String> strings, int length)
  {
    var List<String> result = new ArrayList<String>()
    for (String s : strings)
    {
      if (s.length() < length + 1)
      {
        result.add(s)
      }
    }
    return result
  }
}
```

*Can some types be inferred?*

# Type inference



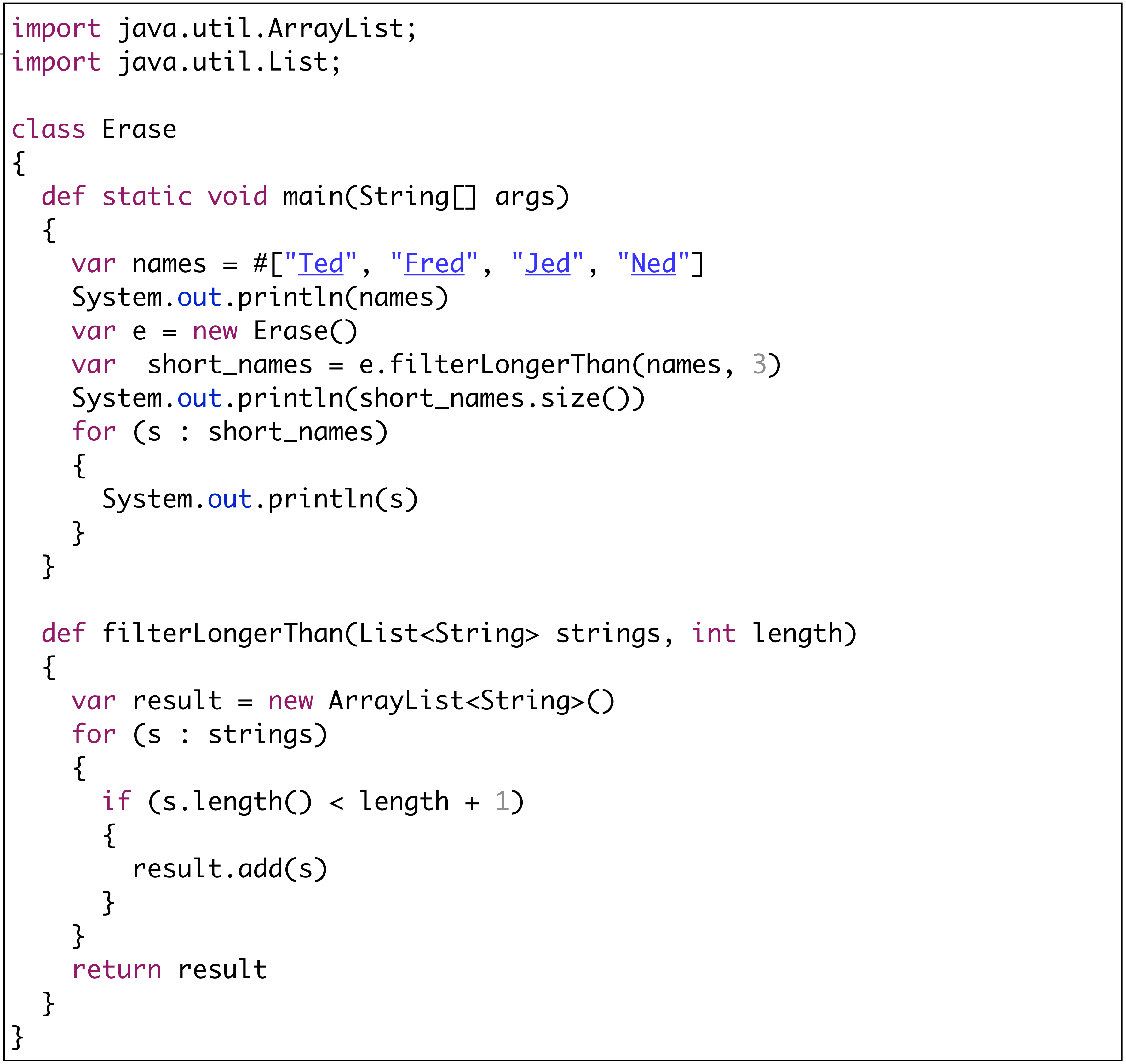
```
import java.util.ArrayList;
import java.util.List;

class Erase
{
  def static void main(String[] args)
  {
    var names = new ArrayList<String>()
    names.add("Ted")
    names.add("Fred")
    names.add("Jed")
    names.add("Ned")
    System.out.println(names)
    var e = new Erase()
    var  short_names = e.filterLongerThan(names, 3)
    System.out.println(short_names.size())
    for (s : short_names)
    {
      System.out.println(s)
    }
  }

  def filterLongerThan(List<String> strings, int length)
  {
    var result = new ArrayList<String>()
    for (s : strings)
    {
      if (s.length() < length + 1)
      {
        result.add(s)
      }
    }
    return result
  }
}
```

*What about Collection Literals?*

# Collection Literals



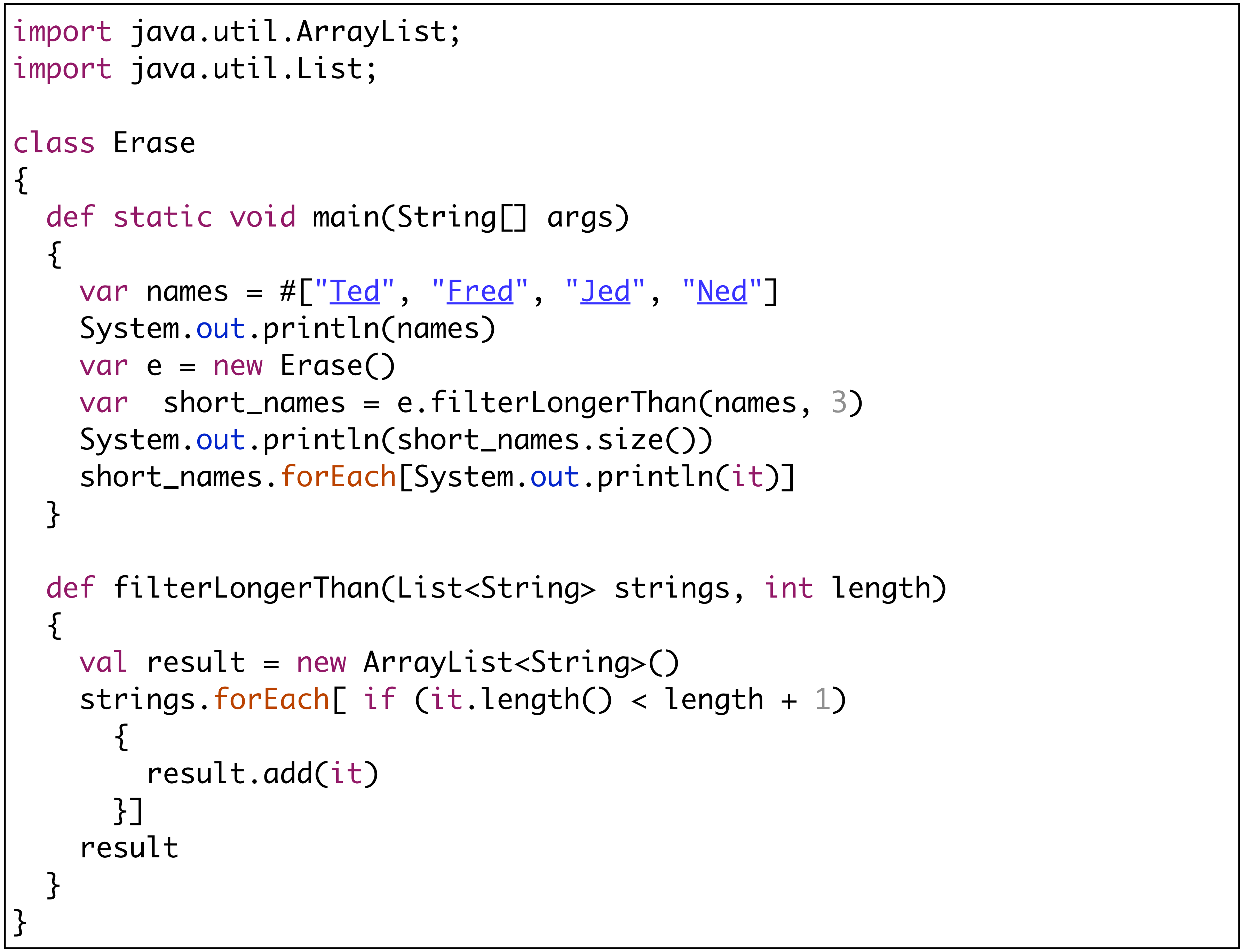
```
import java.util.ArrayList;
import java.util.List;

class Erase
{
  def static void main(String[] args)
  {
    var names = #["Ted", "Fred", "Jed", "Ned"]
    System.out.println(names)
    var e = new Erase()
    var  short_names = e.filterLongerThan(names, 3)
    System.out.println(short_names.size())
    for (s : short_names)
    {
      System.out.println(s)
    }
  }

  def filterLongerThan(List<String> strings, int length)
  {
    var result = new ArrayList<String>()
    for (s : strings)
    {
      if (s.length() < length + 1)
      {
        result.add(s)
      }
    }
    return result
  }
}
```

*Can Lambas simplify code?*

# Lambdas



```
import java.util.ArrayList;
import java.util.List;

class Erase
{
  def static void main(String[] args)
  {
    var names = #["Ted", "Fred", "Jed", "Ned"]
    System.out.println(names)
    var e = new Erase()
    var  short_names = e.filterLongerThan(names, 3)
    System.out.println(short_names.size())
    short_names.forEach[System.out.println(it)]
  }

  def filterLongerThan(List<String> strings, int length)
  {
    val result = new ArrayList<String>()
    strings.forEach[ if (it.length() < length + 1)
      {
        result.add(it)
      }]
    result
  }
}
```

*What are List Comprehensions?*

# Filters/List Comprehensions

```
import java.util.List;

class Erase
{
  def static void main(String[] args)
  {
    var names = #["Ted", "Fred", "Jed", "Ned"]
    System.out.println(names)
    var e = new Erase()
    var  short_names = e.filterLongerThan(names, 3)

    System.out.println(short_names.size())
    short_names.forEach[System.out.println(it)]
  }

  def filterLongerThan(List<String> strings, int length)
  {
    val list = strings.filter[it.length() <= 3]
    list
  }
}
```

*Do we need the class Erase at all?*

# Final Version

```
class Erase
{
  def static void main(String[] args)
  {
    var names = #["Ted", "Fred", "Jed", "Ned"]
    println(names)
    var  short_names =  names.filter[it.length() <= 3]
    println(short_names.size())
    short_names.forEach[println(it)]
  }
}
```

```java
import java.util.ArrayList;
import java.util.List;

class Erase
{
  public static void main(String[] args)
  {
    List<String> names = new ArrayList<String>();
    names.add("Ted");
    names.add("Fred");
    names.add("Jed");
    names.add("Ned");
    System.out.println(names);
    Erase e = new Erase();
    List<String> short_names = e.filterLongerThan(names, 3);
    System.out.println(short_names.size());
    for (String s : short_names)
    {
      System.out.println(s);
    }
  }

  public List<String> filterLongerThan(List<String> strings, int length)
  {
    List<String> result = new ArrayList<String>();
    for (String s : strings)
    {
      if (s.length() < length + 1)
      {
        result.add(s);
      }
    }
    return result;
  }
}
```

java

```
class Erase
{
  def static void main(String[] args)
  {
    var names = #["Ted", "Fred", "Jed", "Ned"]
    println(names)
    var  short_names =  names.filter[it.length() <= 3]
    println(short_names.size())
    short_names.forEach[println(it)]
  }
}
```

xtend

```groovy
names = ["Ted", "Fred", "Jed", "Ned"]
println names
short_names = names.findAll{it.size() <= 3}
println short_names.size()
short_names.each {println it}
```

groovy

# Another ‘Shopping List’

| |
|---|
| Object-literal syntax for arrays and hashes |
| Array slicing and other intelligent collection operators |
| Perl 5 compatible regular expression literals |
| Destructuring bind (e.g. x, y = returnTwoValues()) |
| Function literals and first-class, non-broken closures |
| Standard OOP with classes, instances, interfaces, polymorphism, etc. |
| Visibility quantifiers (public/private/protected) |
| Iterators and generators |
| List comprehensions |
| Namespaces and packages |
| Cross-platform GUI |
| Operator overloading |
| Keyword and rest parameters |
| First-class parser and AST support |
| Type expressions and statically checkable semantics |
| Solid string and collection libraries |
| Strings and streams act like collections |

http://steve-yegge.blogspot.com.au/2007/02/next-big-language.html

# Java

| | |
|---|---|
| Object-literal syntax for arrays and hashes | |
| Array slicing and other intelligent collection operators | |
| Perl 5 compatible regular expression literals | |
| Destructuring bind (e.g. x, y = returnTwoValues()) | |
| Function literals and first-class, non-broken closures | |
| Standard OOP with classes, instances, interfaces, polymorphism, | y |
| Visibility quantifiers (public/private/protected) | y |
| Iterators and generators | y |
| List comprehensions | |
| Namespaces and packages | y |
| Cross-platform GUI | y |
| Operator overloading | |
| Keyword and rest parameters | |
| First-class parser and AST support | |
| Type expressions and statically checkable semantics | y |
| Solid string and collection libraries | y |
| Strings and streams act like collections | y |

# Google GO

| | |
|---|---|
| Object-literal syntax for arrays and hashes | y |
| Array slicing and other intelligent collection operators | y |
| Perl 5 compatible regular expression literals | |
| Destructuring bind (e.g. x, y = returnTwoValues()) | y |
| Function literals and first-class, non-broken closures | y |
| Standard OOP with classes, instances, interfaces, polymorphism, | |
| Visibility quantifiers (public/private/protected) | y |
| Iterators and generators | |
| List comprehensions | |
| Namespaces and packages | y |
| Cross-platform GUI | |
| Operator overloading | |
| Keyword and rest parameters | y |
| First-class parser and AST support | y |
| Type expressions and statically checkable semantics | y |
| Solid string and collection libraries | y |
| Strings and streams act like collections | |

# Python

| | |
|---|---|
| Object-literal syntax for arrays and hashes | y |
| Array slicing and other intelligent collection operators | y |
| Perl 5 compatible regular expression literals | y |
| Destructuring bind (e.g. x, y = returnTwoValues()) | y |
| Function literals and first-class, non-broken closures | y |
| Standard OOP with classes, instances, interfaces, polymorphism, | y |
| Visibility quantifiers (public/private/protected) | |
| Iterators and generators | y |
| List comprehensions | y |
| Namespaces and packages | y |
| Cross-platform GUI | |
| Operator overloading | |
| Keyword and rest parameters | y |
| First-class parser and AST support | y |
| Type expressions and statically checkable semantics | |
| Solid string and collection libraries | y |
| Strings and streams act like collections | y |

# Xtend

| | |
|---|---|
| Object-literal syntax for arrays and hashes | y |
| Array slicing and other intelligent collection operators | y |
| Perl 5 compatible regular expression literals | |
| Destructuring bind (e.g. x, y = returnTwoValues()) | |
| Function literals and first-class, non-broken closures | y |
| Standard OOP with classes, instances, interfaces, polymorphism, | y |
| Visibility quantifiers (public/private/protected) | y |
| Iterators and generators | y |
| List comprehensions | y |
| Namespaces and packages | y |
| Cross-platform GUI | y |
| Operator overloading | y |
| Keyword and rest parameters | |
| First-class parser and AST support | |
| Type expressions and statically checkable semantics | y |
| Solid string and collection libraries | y |
| Strings and streams act like collections | y |

Active Annotations ?