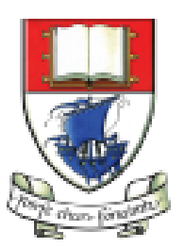
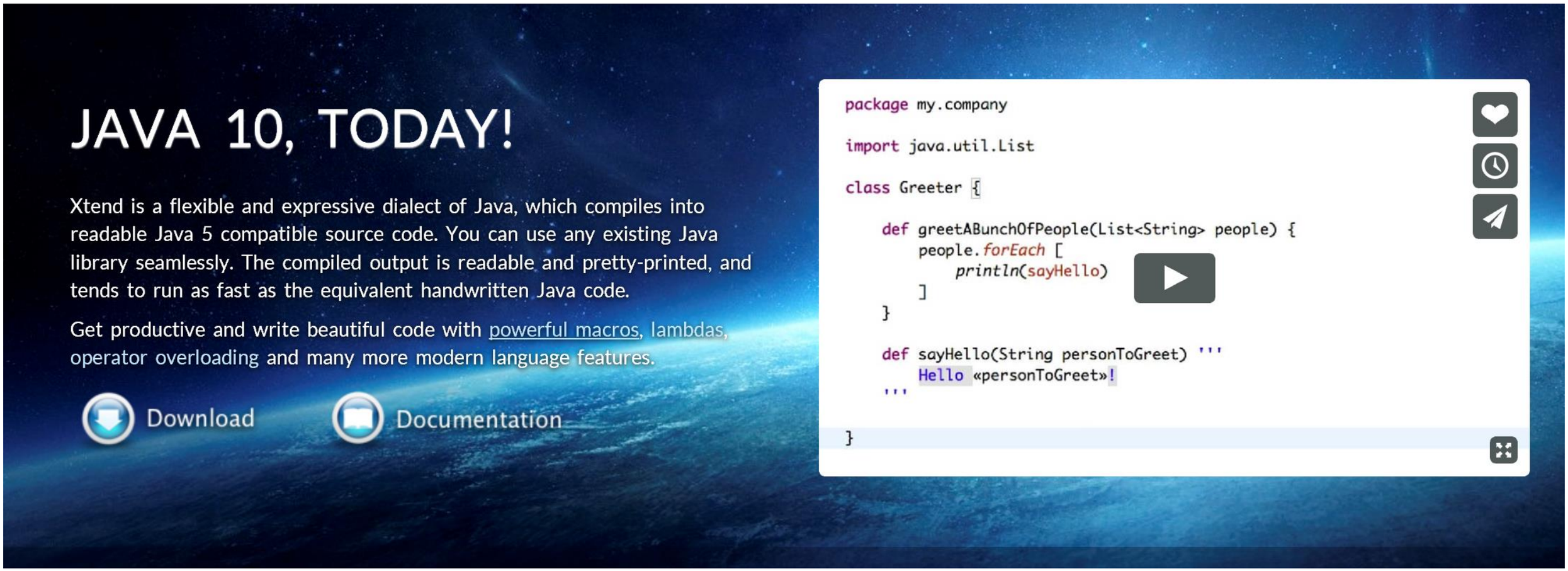
Xtend Programming Language

Produced by: Eamonn de Leastar (edeleastar@wit.ie)



Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/



[Excellent Xtend User Guide]

(Version 2.6)

[API Docs (version 2.8)]

In Xtend everything is an expression and has a return type.
Statements do not exist.

Hello World

xtend

```
class HelloWorld
{
  def static void main(String[] args)
  {
    println("Hello World")
  }
}
```

java

```
public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello World");
  }
}
```

Relevant XTend Features (for pace-console-xtend)

- Java Interoperability
  - Type Inference
  - Conversion Rules
- Classes
  - Constructors
  - Fields
  - Methods
  - Override
  - Inferred return types

- Expressions
  - Literals
  - Casts
  - Field access & method invocation
  - Constructor call
  - Lambda Expressions
  - If expression
  - switch Expression
  - return expression

- Annotations
  - @Accessors
  - @Data

Type inference

- Xtend and Java → statically typed language (type checking done at compile time).

- Supports Java's type system i.e.
  primitive types, arrays all Java classes, interfaces, enums and annotations that reside on the class path.

- With Java, you are forced to write type signatures over and over again…it’s not a problem of static typing but simply a problem with Java.

- Although Xtend is statically typed just like Java, you rarely have to write types down because they can be computed from the context.

Type inference

```
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);
  }
}
```

- Java

```
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)
  }
}
```

- XTend

Type inference

```
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);
  }
}
```

- Java

```
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)
  }
}
```

- XTend

Conversion Rules

- In addition to Java's autoboxing to convert primitives to their corresponding wrapper types (e.g. int is automatically converted to Integer when needed), there are additional conversion rules in Xtend.

- Arrays are automatically converted to List<ComponentType> when needed and vice versa.

- Subsequent changes to the array are reflected by the list and vice versa.

- Arrays of primitive types are converted to lists of their respective wrapper types.

- Conversion works the other way around too, similar to Java's *unboxing*: all subtypes of Iterable are automatically converted to arrays on demand.

This is valid Xtend code:

```
def toList(String[] array)
{
  val List<String> asList = array
  return asList
}
```

Relevant XTend Features (for pace-console-xtend)

- Java Interoperability
  - Type Inference
  - Conversion Rules
- Classes
  - Constructors
  - Fields
  - Methods
  - Override
  - Inferred return types

- Expressions
  - Literals
  - Casts
  - Field access & method invocation
  - Constructor call
  - Lambda Expressions
  - If expression
  - switch Expression
  - return expression

- Annotations
  - @Accessors
  - @Data

Classes

- At a first glance an Xtend file pretty much looks like a Java file.
- It starts with a package declaration followed by an import section and class definitions.
- The classes in fact are directly translated to Java classes in the corresponding Java package (see the xtend-gen folder).
- An Xtend class can have constructors, fields, methods and annotations.

```
package acme.com

import java.util.List

class MyClass
{
    String name

    new(String name)
    {
        this.name = name
    }

    def String first(List<String> elements)
    {
        elements.get(0)
    }
}
```

Class Declaration

- The class declaration reuses a lot of Java's syntax but still is a bit different in some aspects:
  - All Xtend types are public by default.
  - Xtend supports multiple public top level class declarations per file. Each Xtend class is compiled to a separate top-level Java class.
  - Abstract classes are defined using the abstract modifier as in Java.

Constructors

- An Xtend class can define any number of constructors.
- Unlike Java you do not have to repeat the name of the class over and over again, but use the keyword new to declare a constructor.
- Constructors can also delegate to other constructors using this() in their first line.

```
class MyClass
{
   String name

   new(String name)
   {
      this.name = name
   }

   def String first(List<String> elements)
   {
      elements.get(0)
   }
}

class MySpecialClass extends MyClass
{
   new(String s)
   {
      super(s)
   }

   new()
   {
      //delegating to the first constructor above
      this("default")
   }
}
```

Fields

- A field can have an initializer.
- Final fields are declared using **val**, while **var** introduces a non-final field (and can be omitted).
- If an initializer expression is present, the type of a field can be inferred only if val or var was used to introduce the field.
- The keyword final is synonym to val.
- Fields marked as static will be compiled to static Java fields.
- The default visibility for fields is private. You can also declare it explicitly as being public, protected, package or private.

```
class MyDemoClass
{
    int count = 1
    static boolean debug = false
    var name = 'Foo' // type String is inferred
    val UNIVERSAL_ANSWER = 42 // final field with inferred type int
}
```

Methods

- Xtend methods are declared within a class and are translated to a corresponding Java method with exactly the same signature.

- Method declarations start with the keyword def.

- The default visibility of a method is public; you can explicitly declare it as being public, protected, package or private.

```
class MyClass
{
    String name

    new(String name)
    {
        this.name = name
    }

    def String first(List<String> elements)
    {
         elements.get(0)
    }

    def static createInstance()
    {
       new MyClass('foo')
    }
}
```

Inferred Return Types

- If the return type of a method can be inferred from its body it does not have to be declared.

```
def String first(List<String> elements)
{
    elements.get(0)
}
```

- Notice also that the keyword, return, is not used.

```
def first(List<String> elements)
{
    elements.get(0)
}
```

- Return types must be explicit in abstract and recursive methods.

Overriding Methods

- Methods can override methods from the super class or implement interface methods using the keyword override.
- If a method overrides a method from a super type, the override keyword is mandatory and replaces the keyword def.
- The override semantics are the same as in Java, e.g. it is impossible to override final methods or invisible methods.
- Overriding methods inherit their return type from the super declaration.

```
class MyClass
{
    String name

    new(String name)
    {
        this.name = name
    }

    def String first(List<String> elements)
    {
        elements.get(0)
    }
}

class MySpecial extends MyClass
{
    new(String s)
    {
        super(s)
    }

    override first(List<String> elements)
    {
        elements.get(1)
    }
}
```

Relevant XTend Features (for pace-console-xtend)

- Java Interoperability
  - Type Inference
  - Conversion Rules
- Classes
  - Constructors
  - Fields
  - Methods
  - Override
  - Inferred return types

- Expressions
  - Literals
  - Casts
  - Field access & method invocation
  - Constructor call
  - Lambda Expressions
  - If expression
  - switch Expression
  - return expression

- Annotations
  - @Accessors
  - @Data

Collection Literals

- Convenient to create instances of the various collection types the JDK offers.

```
val myList = newArrayList('Hello', 'World')
val myMap = newLinkedHashMap('a' -> 1, 'b' -> 2)
```

- Xtend supports collection literals to create immutable collections and arrays, depending on the target type.

```
val myList = #['Hello','World']
```

- If the target type is an array, an array is created instead without any conversion:

```
val String[] myArray = #['Hello','World']
```

- An immutable set can be created using curly braces instead of the squared brackets:

```
val mySet = #{'Hello','World'}
```

- An immutable map is created like this:

```
val myMap = #{'a' -> 1 ,'b' ->2}
```

Type Casts

- A type cast behaves exactly like casts in Java, but has a slightly more readable syntax.

```
something as MyClass


42 as Integer
```

Null-Safe Feature Call

- Checking for null references can make code very unreadable.

```
if (myRef != null) myRef.doStuff()
```

- In many situations it is ok for an expression to return null if a receiver was null.

- Xtend supports the **safe navigation operator**

  ?.

  to make such code more readable.

```
myRef?.doStuff
```

Elvis Operator

```
@Data class Person
{
    String title
    String firstName
}
```

- In addition to null-safe feature calls, Xtend supports the **Elvis operator (?:)** known from Groovy.
- The right hand side of the expression is only evaluated if the left side was null.

```
val person = new Person(null, 'John')
val salutation = person.title ?: 'Sir/Madam'
println(salutation)
```

Prints Sir/Madam to the console

```
val person = new Person(‘Master', 'John')
val salutation = person.title ?: 'Sir/Madam'
println(salutation)
```

Prints Master to the console

Variable Declarations

- A variable declaration starting with the keyword **val** denotes a value, which is essentially a final, unsettable variable.

Xtend

```
val max = 100
var i = 0
while (i < max)
{
    println("Hi there!")
    i = i + 1
}
```

- The variable needs to be declared with the keyword **var**, which stands for 'variable' if it should be allowed to reassign its value.

Generated Java

```
final int max = 100;
int i = 0;
while ((i < max)) {
  {
    InputOutput.<String>println("Hi there!");
    i = (i + 1);
  }
}
```

Typing

- The type of the variable itself can either be explicitly declared or it can be inferred from the initializer expression.

- Explicit declaration: the type of the right hand expression must conform to the type of the expression on the left side.

```
var List<String> strings = new ArrayList
```

- Inferred declaration: the type can be inferred from the initializer.

```
var strings = new ArrayList<String>
```

Constructor Call

- Constructor calls have the same syntax as in Java.

- The only difference is that empty parentheses are optional e.g.:

```
new String() == new String
new ArrayList<BigDecimal>() == new ArrayList<BigDecimal>
```

- If type arguments are omitted, they will be inferred from the current context similar to Java's diamond operator on generic method and constructor calls.

```
var stringList = new ArrayList  // type will be ArrayList <String>
stringList.add("First Element")
println(stringList.get(0))
```

Lambda Expressions (1)

- A lambda expression is basically a piece of code, which is wrapped in an object to pass it around.

- As a Java developer it is best to think of a lambda expression as an anonymous class with a single method.

- These kind of anonymous classes can be found everywhere in Java code and have always been the poor-man's replacement for lambda expressions in Java.

```
// Java Code
final JTextField textField = new JTextField();
textField.addActionListener(new ActionListener()
{
    @Override
    public void actionPerformed(ActionEvent e)
    {
        textField.setText("Something happened!");
    }
});
```

Lambda Expressions (2)

- Xtend not only supports lambda expressions, but offers an extremely dense syntax for it.

```
// Java Code
final JTextField textField = new JTextField();


textField.addActionListener(new ActionListener()
{
    @Override
    public void actionPerformed(ActionEvent e)
    {
        textField.setText("Something happened!");
    }
});
```

```
// Xtend Code
val textField = new JTextField


textField.addActionListener([ ActionEvent e |
    textField.text = "Something happened!"
])
```

Lambda Expressions (3)

- Lambda expression is surrounded by square brackets (inspired from Smalltalk).

- Also a lambda expression like a method declares parameters.

```
textField.addActionListener([ e |
   textField.text = "Something happened!"
])
```

- The lambda here has one parameter :
  `e` which is of type `ActionEvent`.

- You do not have to specify the type explicitly because it can be inferred from the context.

Lambda Expressions (4)

- Also as lambdas with one parameter are a common case, there is a special short hand notation for them, which is to leave the declaration including the vertical bar out.

```
textField.addActionListener([
  textField.text = "Something happened!"
])
```

- The name of the single variable will be **it** in that case.

- Since you can leave out empty parentheses for methods which get a lambda as their only argument, you can reduce the code above further down.

```
textField.addActionListener [textField.text = "Something happened!"]
```

```
textField.addActionListener(new ActionListener()
{
  @Override
  public void actionPerformed(ActionEvent e)
  {
    textField.setText("Something happened!");
  }
});
```

Java Code

```
textField.addActionListener([ ActionEvent e |
  textField.text = "Something happened!"
])
```

Xtend Code

```
textField.addActionListener([ e |
  textField.text = "Something happened!"
])
```

Inferred Type

```
textField.addActionListener([
  textField.text = "Something happened!"
])
```

Shorthand for single parameter lambdas

```
textField.addActionListener [textField.text = "Something happened!"]
```

No parenthesis

Lambdas & Collections (1)

- The collections have been equipped with Extension Methods that take lambda as parameters e.g. classes ListExtensions, IterableExtensions, etc.

```
def printAll(ArrayList<String>  strings) {
    strings.forEach [ s | println(s) ]
 }
```

- *Can dramatically reduce number of loops in a program!*

```
list.forEach[ element, index |
  .. // if you need access to the current index
]
list.reverseView.forEach[
  .. // if you just need the element it in reverse order
]
```

Lambdas & Collections (2)

```
val strings = newArrayList("red", "blue", "green")
val charCount = strings.map[s|s.length].reduce[sum, size | sum + size]
println(charCount)
```

Output is 12

strings.map[s|s.length]

The map method is in the ListExtensions class. It returns a list built using the Lambda expression i.e. Iterate through the ArrayList called strings, and for each object s found in the ArrayList, the length of the String s is added as an element to the returned List i.e. [3, 4, 5] is returned.

reduce[sum, size | sum + size]

The reduce method is in the IterableExtensions class. It applies the function to all elements of the List [3, 4, 5] in turn
i.e. given our list [3, 4, 5] and the function add (+), the result returned will be: add(add(3, 4), 5)

Switch Expression

- The switch expression is very different from Java's switch statement:
  - there is no fall through which means only one case is evaluated at most.
  - The use of switch is not limited to certain values but can be used for any object reference.
  - Object.equals(Object) is used to compare the value in the case with the one you are switching over.

```
switch myString
{
  case myString.length > 5 :  print("a long string.")
  case 'some' :               print("It\'s some string.")
  default :                   print("It\'s another short string.")
}
```

Switch Expression- Type guards

- Instead of or in addition to the case guard you can specify a type guard.

- The case only matches if the switch value conforms to this type.

- A case with both a type guard and a predicate only matches if both conditions match.

```
def length(Object x)
{
  switch x
  {
    String case x.length > 0 : x.length
                // length is defined for String
    List<?> : x.size
                // size is defined for List
    default : -1
  }
}
```

Relevant XTend Features (for pace-console-xtend)

- Java Interoperability
  - Type Inference
  - Conversion Rules
- Classes
  - Constructors
  - Fields
  - Methods
  - Override
  - Inferred return types

- Expressions
  - Literals
  - Casts
  - Field access & method invocation
  - Constructor call
  - Lambda Expressions
  - If expression
  - switch Expression
  - return expression

- Annotations
  - @Accessors
  - @Data

Active Annotations

- Xtend comes with ready-to-use active annotations for common code patterns.

- They reside in the org.eclipse.xtend.lib.annotations plug-in/jar which must be on the class path of the project containing the Xtend files.

@Accessors

- For fields that are annotated as @Accessors, the Xtend compiler will generate a Java field, a getter and, if the field is non-final, a setter method.

- The generated methods are, by default, public.

- @Accessors can be used at class level too e.g.:

  @Accessors **class** Person {

```
@Accessors String name
```

generates

```
private String name;

public String getName()
{
  return this.name;
}

public void setName(final String name)
{
  this.name = name;
}
```

@Accessors

- You can use the AccessorType to change the defaults.

```
@Accessors(PUBLIC_GETTER, PROTECTED_SETTER) int age
@Accessors(NONE) String internalField
```

generates

```
@Accessors(PUBLIC_GETTER, PROTECTED_SETTER) private int age
@Accessors(NONE) private String internalField

public int getAge() {
    return this.age;
}
protected void setAge(final int age) {
    this.age = age;
}
```

@Data

- The annotation @Data will turn an annotated class into a value object class. A class annotated with @Data is processed according to the following rules:
  - all fields are final,
  - getter methods will be generated (if they do not yet exist),
  - a constructor with parameters for all non-initialized fields will be generated (if it does not exist),
  - equals(Object) / hashCode() methods will be generated (if they do not exist),
  - a toString() method will be generated (if it does not exist).

```
@Data class Person
{
  String firstName
  String lastName
}
```

```
@Data class Person
{
  String firstName
  String lastName
}
```

```
@Data
@SuppressWarnings("all")
public class Person {
  private final String _firstName;
  public String getFirstName() {
    return this._firstName;
  }
  private final String _lastName;
  public String getLastName() {
    return this._lastName;
  }
  public Person(final String firstName, final String lastName) {
    super();
    this._firstName = firstName;
    this._lastName = lastName;
  }
  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((_firstName== null) ? 0 : _firstName.hashCode());
    result = prime * result + ((_lastName== null) ? 0 : _lastName.hashCode());
    return result;
  }
```

```
  @Override
  public boolean equals(final Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Person other = (Person) obj;
    if (_firstName == null) {
      if (other._firstName != null)
        return false;
    } else if (!_firstName.equals(other._firstName))
      return false;
    if (_lastName == null) {
      if (other._lastName != null)
        return false;
    } else if (!_lastName.equals(other._lastName))
      return false;
    return true;
  }
  @Override
  public String toString() {
    String result = new ToStringHelper().toString(this);
    return result;
  }
}
```

Additional Reading: Features (1)

- Extension methods - enhance closed types with new functionality
- Lambda Expressions - concise syntax for anonymous function literals
- ActiveAnnotations - annotation processing on steroids
- Operator overloading - make your libraries even more expressive
- Powerful switch expressions - type based switching with implicit casts
- Multiple dispatch - a.k.a. polymorphic method invocation

Additional Reading: Features (2)

- Template expressions - with intelligent white space handling
- No statements - everything is an expression and has a return type
- Properties - shorthands for accessing and defining getters and setter
- Type inference - you rarely need to write down type signatures anymore
- Full support for Java generics - including all conformance and conversion rules
- Translates to Java not bytecode - understand what is going on and use your code for platforms such as Android or GWT

Features Relevant to pacemaker-console-x

- Extension methods - enhance closed types with new functionality
- Lambda Expressions - concise syntax for anonymous function literals
- ActiveAnnotations (@Accessors) - shorthands for accessing and defining getters and setter
- Powerful switch expressions - type based switching with implicit casts
- Type inference - you rarely need to write down type signatures anymore