CanWay® 嘉为

JAVA BASICS

Functional programing, Generics, Maven and more…

Java workshop training, August,2017.

Dr. Kishore Biswas (Forrest/柯修)

PhD. Artificial Intelligence—Natural Language Processing.

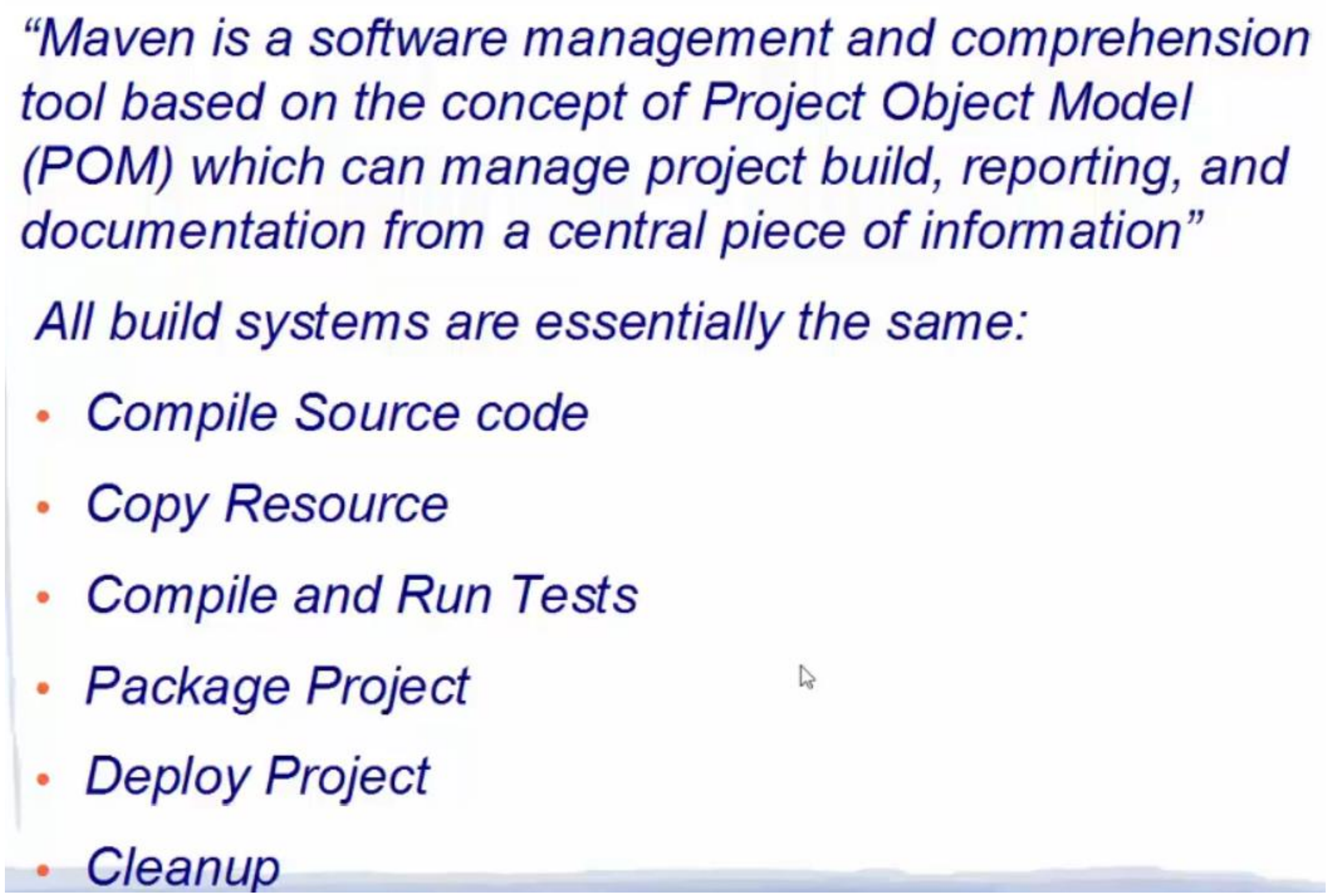
CanWay IT Training ®.

CONTENTS

1) Maven (Continued..)

What is Maven?

- A Java project management and integration build tool.
- Based on the concept of XML Project Object Model (POM).
- Originally developed for building Turbine.
- A small core with numerous plugins (in Jelly).

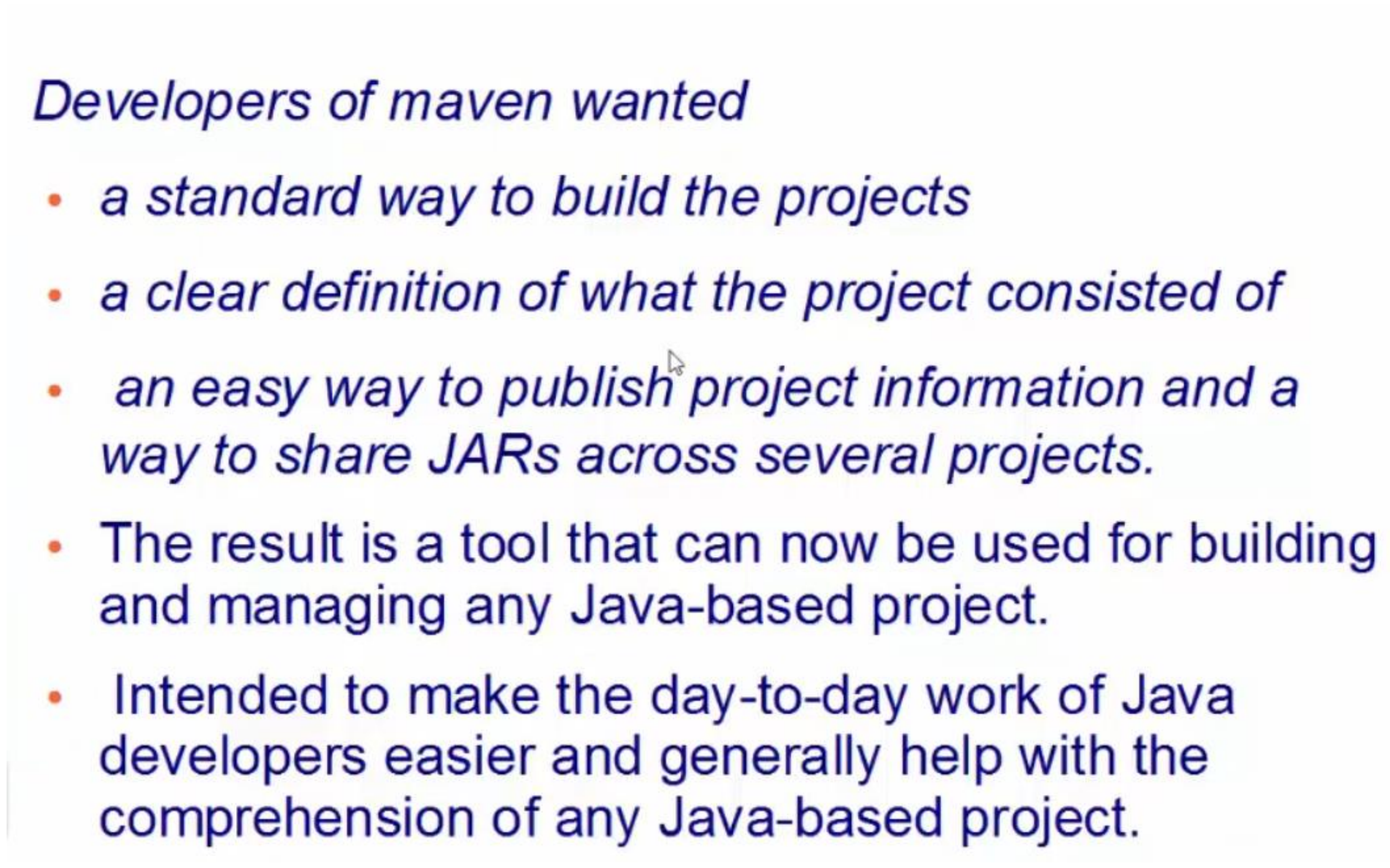


"Maven is a software management and comprehension tool based on the concept of Project Object Model (POM) which can manage project build, reporting, and documentation from a central piece of information"

All build systems are essentially the same:

- *Compile Source code*
- *Copy Resource*
- *Compile and Run Tests*
- *Package Project*
- *Deploy Project*
- *Cleanup*

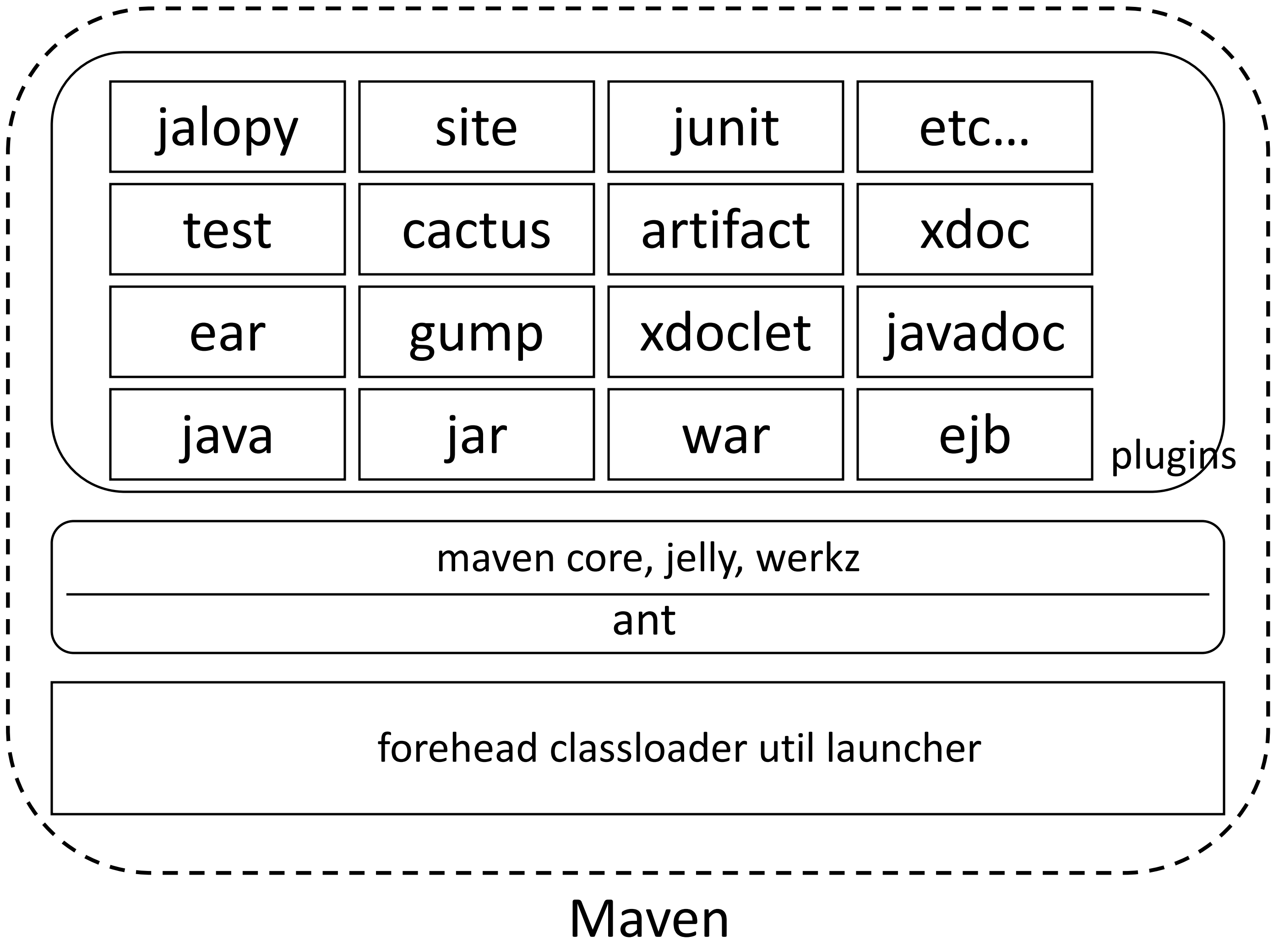
What is Maven?



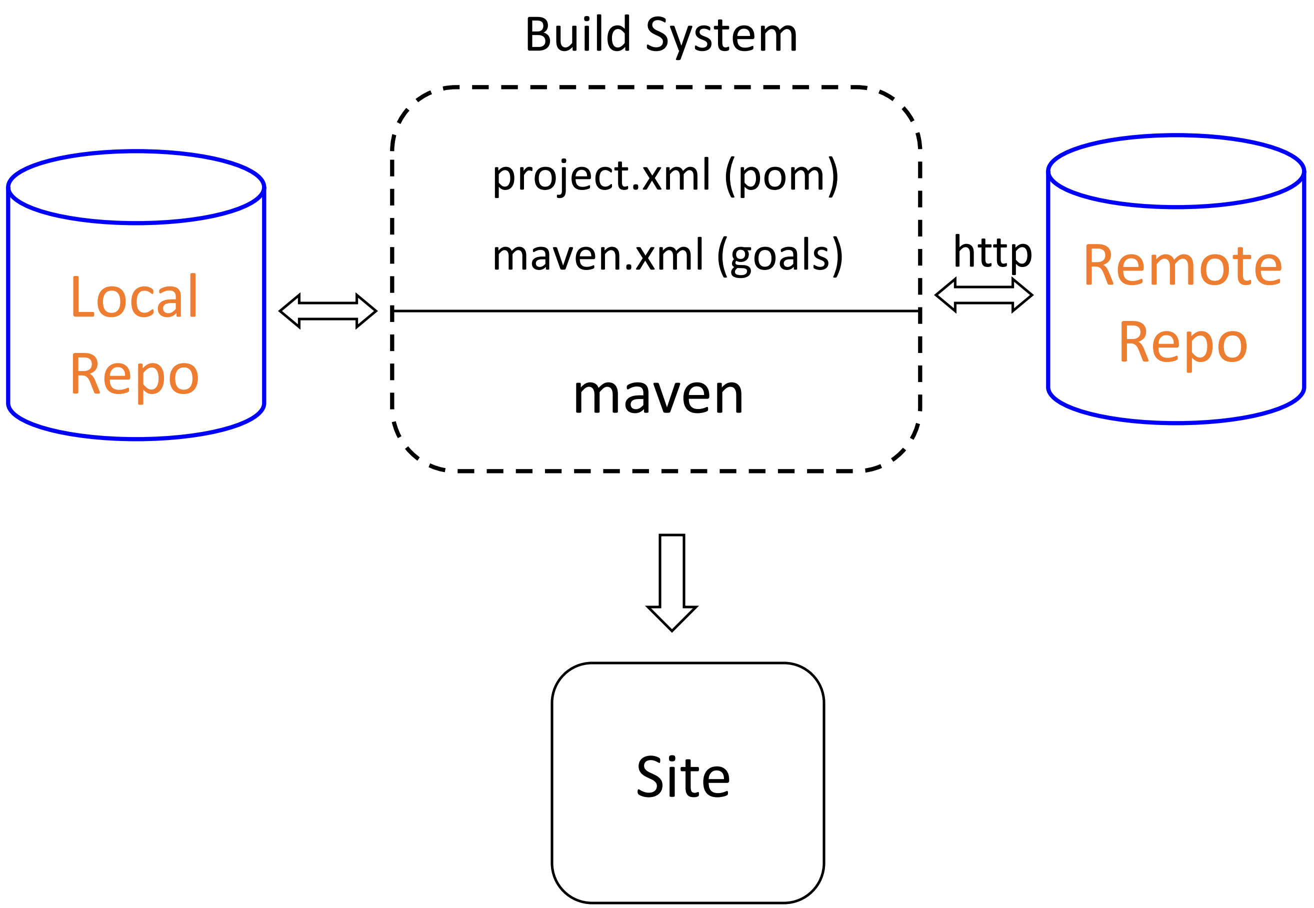
Developers of maven wanted

- *a standard way to build the projects*
- *a clear definition of what the project consisted of*
- *an easy way to publish project information and a way to share JARs across several projects.*
- The result is a tool that can now be used for building and managing any Java-based project.
- Intended to make the day-to-day work of Java developers easier and generally help with the comprehension of any Java-based project.

UNDER THE HOOD



Architecture Overview



Artifact Repository

The Most Important Feature

- Remote Repository

${mave.repo.remote}/<**groupId**>/<**type**>s/<**artifactId**>-<**version**>.<**type**>

```
…
<dependencies>
  <dependency>
     <groupId>xalan</groupId>
     <artifactId>xalan</artifactId>
     <version>2.5.1</version>
     <type>jar</type>
  </dependency>
  …
</dependencies>
…
```

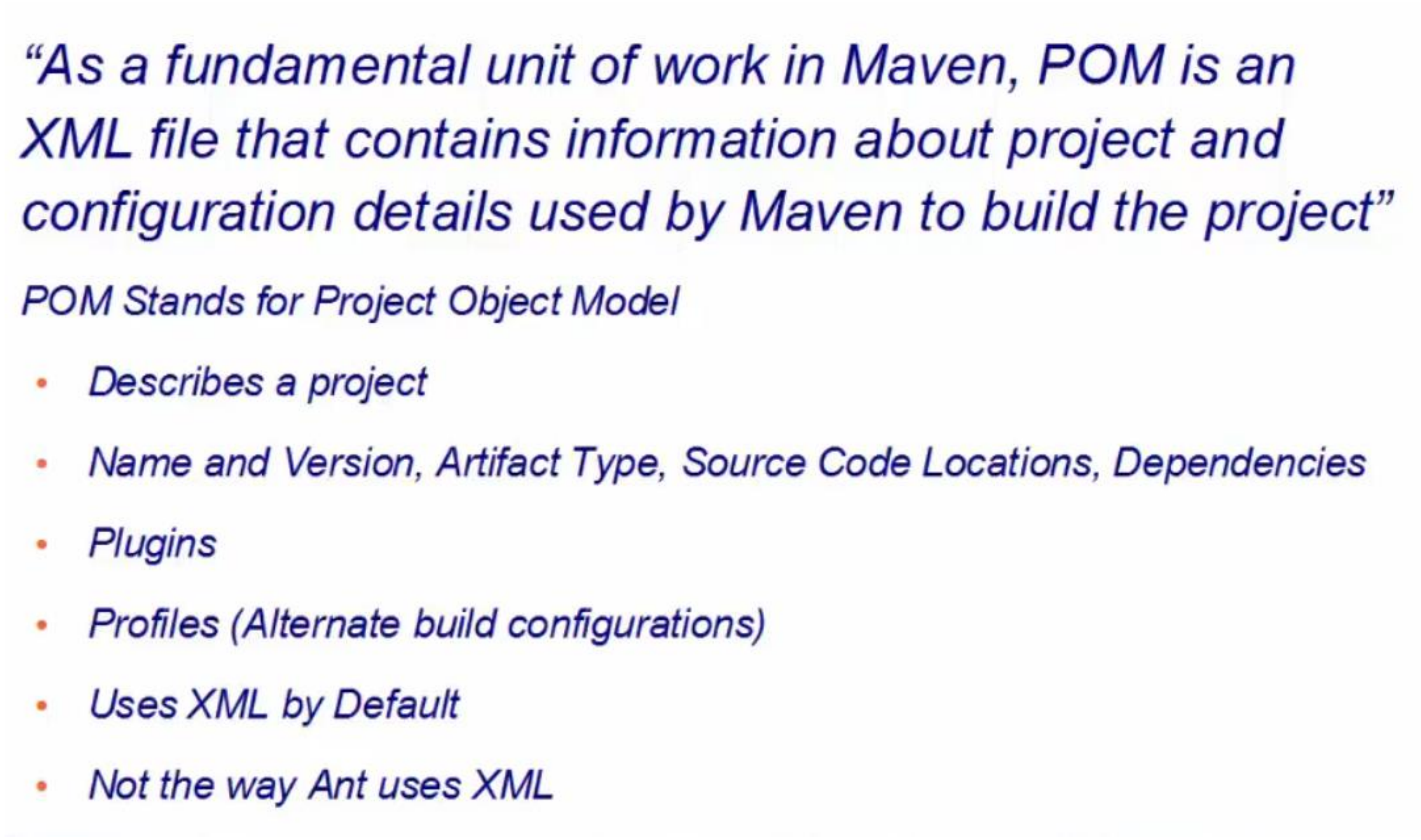
```
- repository
      - […]
      - xalan
            - jars
                  - xalan-2.5.0.jar
                  - xalan-2.5.1.jar
                  - […]
      - […]
```

Artifact Repository

- Local Repository
  - A local mirror/cache of downloaded artifacts from remote repositories.
  - Located at ${user.home}/.maven/repository

${mave.repo.local}/<**groupId**>/<**type**>s/<**artifactId**>-<**version**>.<**type**>

THE POM (PROJECT OBJECT MODEL)

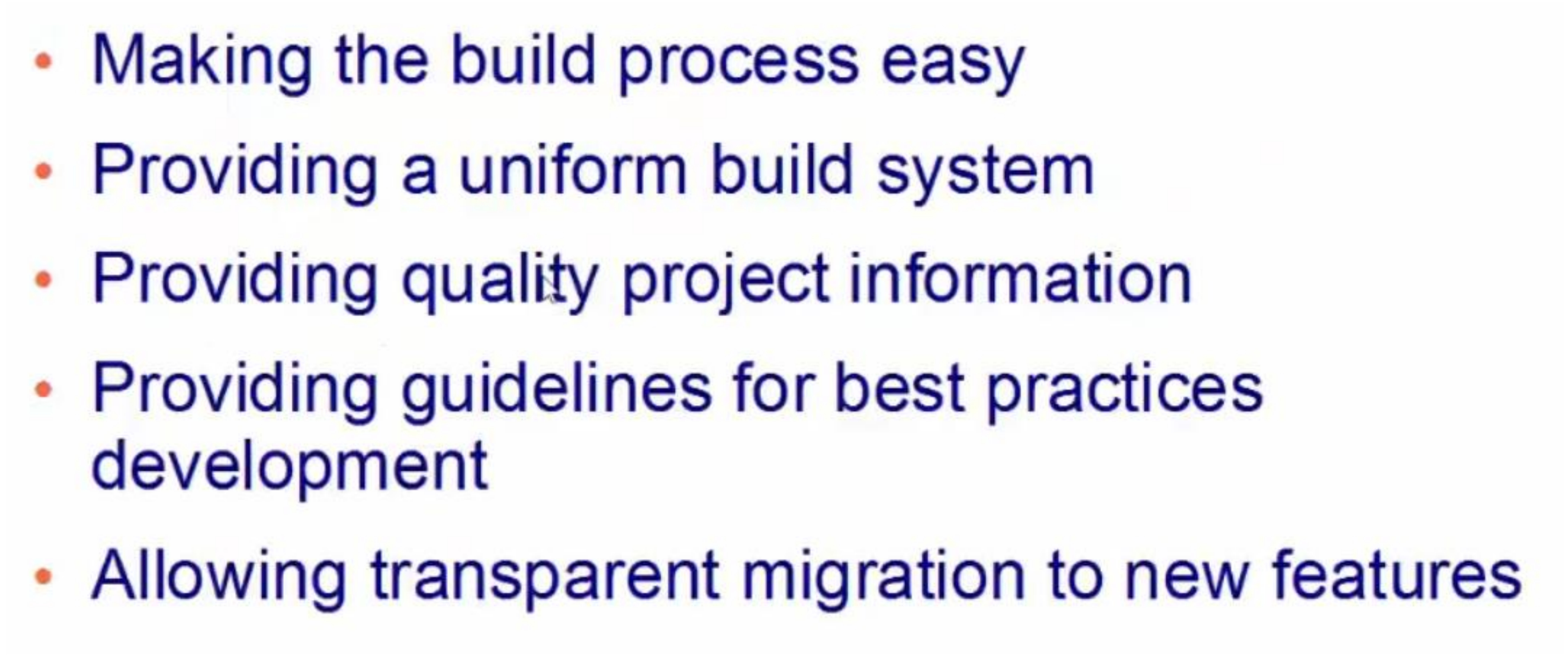


"As a fundamental unit of work in Maven, POM is an XML file that contains information about project and configuration details used by Maven to build the project"

POM Stands for Project Object Model

- *Describes a project*
- *Name and Version, Artifact Type, Source Code Locations, Dependencies*
- *Plugins*
- *Profiles (Alternate build configurations)*
- *Uses XML by Default*
- *Not the way Ant uses XML*

OBJECTIVES OF MAVEN



- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices development
- Allowing transparent migration to new features

IMPLEMENTING AN EXAMPLE

- Get Started
    - Download from
        - http://maven.apache.org/start/download.html
    - Current version: 3.0rc1
    - Environment setup
        - export MAVEN_HOME=c:/maven-1.0rc1
        - export PATH="$MAVEN_HOME/bin;$PATH"

          (or set MAVEN_HOME = c:\maven-1.0rc1

          set PATH = %MAVEN_HOME%\bin;%PATH%)
    - run *install_repo.sh* to populate the local repository

CREATE A NEW PROJECT

Type:

maven genapp

It will prompt for

- project id
- project name
- project package name

A Sample Service J2EE Project

- EJB (stateless session beans exposed as web services)
- Data components
- Web application

Directory Layout

<u>Project Directory Layout</u>

sampleservice

- project.xml - Master POM of the project
- maven.xml- Reactor definition
- project.properties - Properties related to the project
- application/ - Application component
- service-data/ - Common data component
- service-ejb/ - EJB/WS component
- service-web/ - Web Application component
- target/ - Generated artifact directory
- xdocs/ - Various documents in xml format
- …

Directory Layout

<u>A Component Directory Layout</u>

…

service-data	- Data component subproject
• project.xml	- POM of the data project
• maven.xml- Goals definition	
• project.properties	- Properties related to the project
• src/	- Source directory
– conf/	- Configuration and resource files
– java/	- Java source files
– test/	- Test source files
• target/	- Generated artifact directory
• xdocs/	- Various documents in xml format

…

Project Object Model (POM)

Projects are described as Project Object Model.

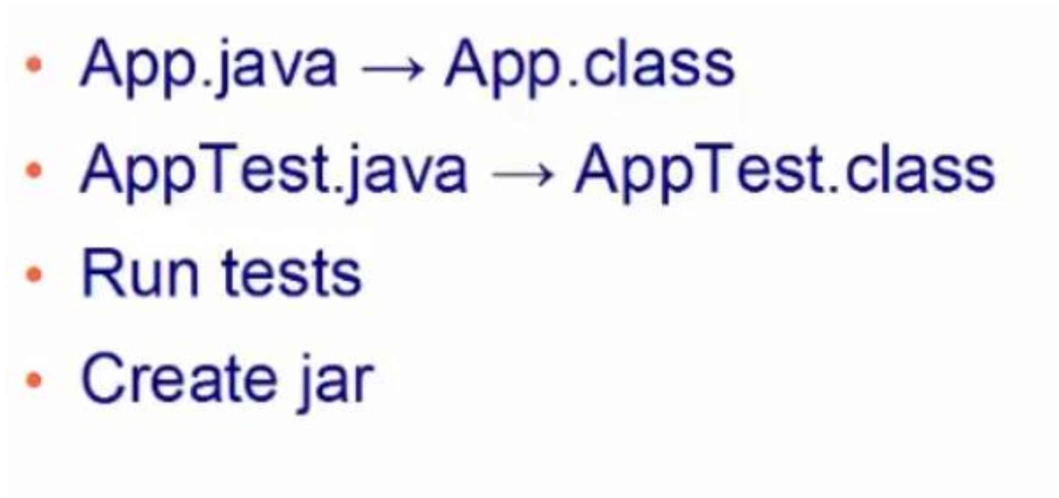
- Project Management
  - Detailed description of the project.
  - Company information.
  - Developer roles and information.
  - Mailing list and source control modules configuration.
- Project Build
  - Source code and test code location.
  - Resources location

Project Object Model (POM)

- Project Dependency
  - Libraries needed for build and runtime.

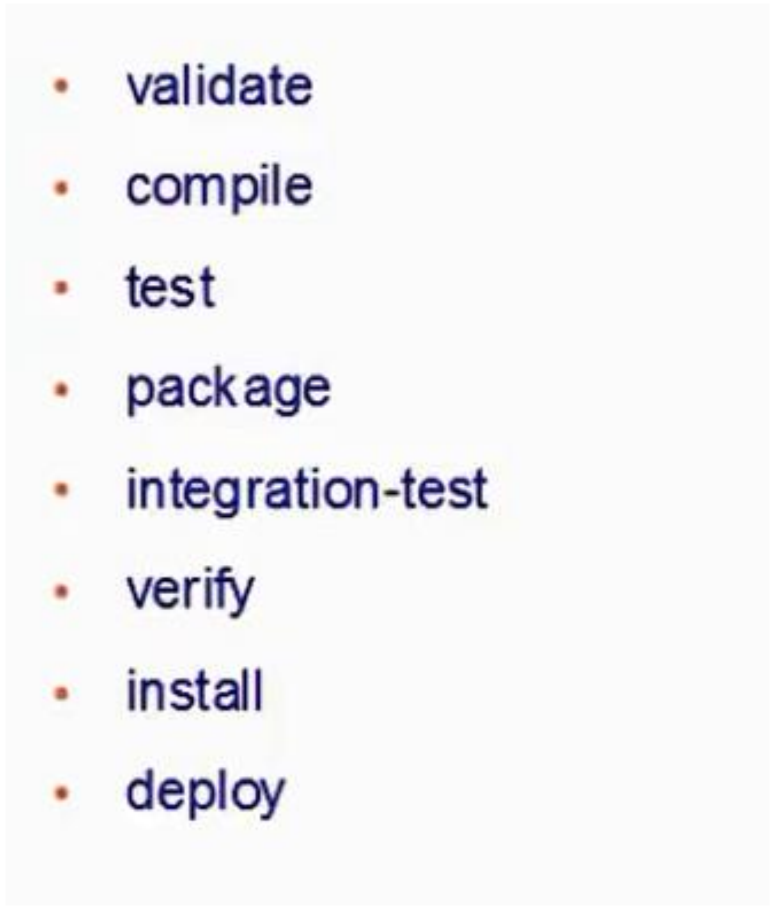
- Project Reports
  - Junit reports
  - Javadoc reports
  - Checkstyle reports, ….etc

Creating JAR files with Maven



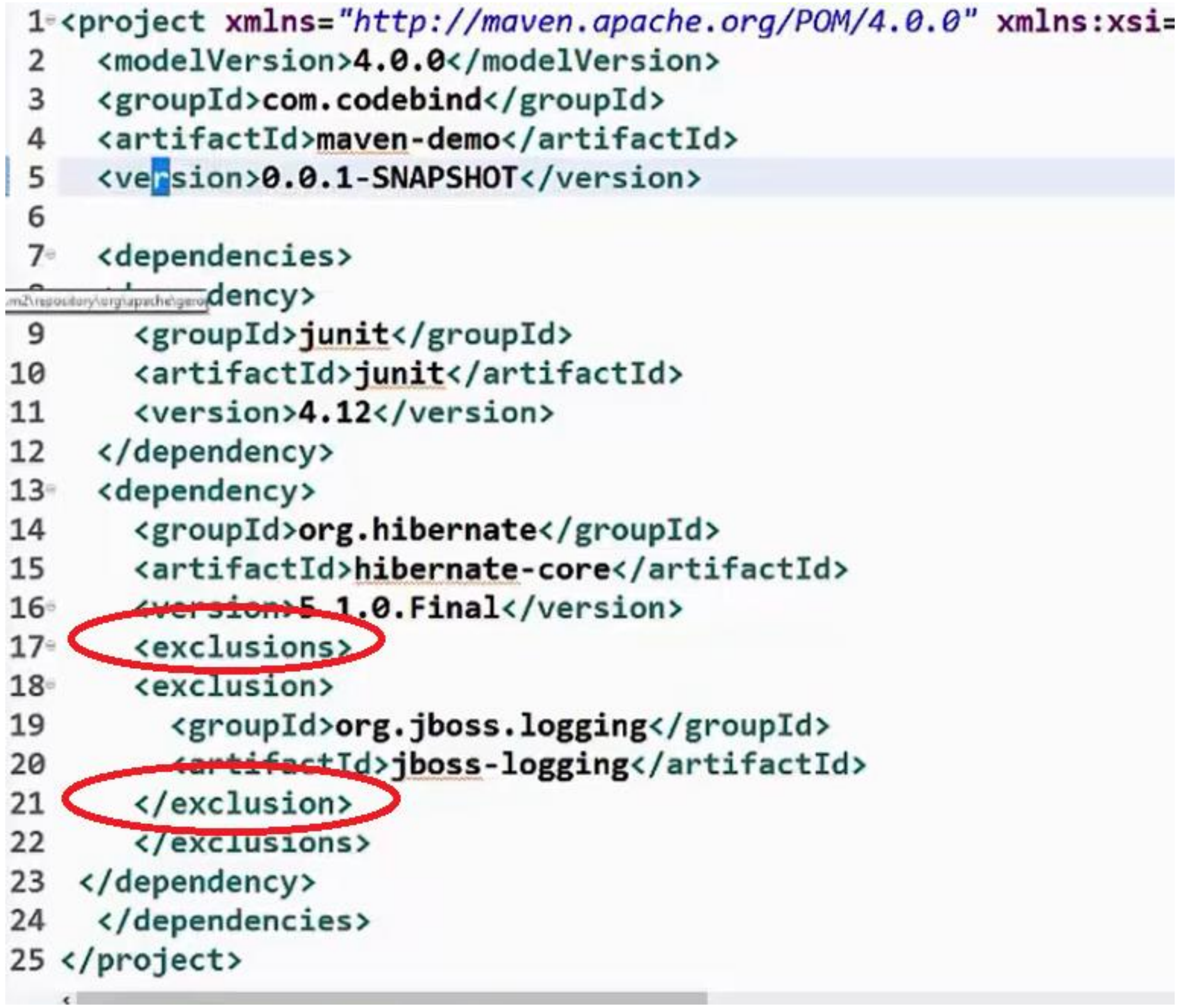
- App.java → App.class
- AppTest.java → AppTest.class
- Run tests
- Create jar

MAVEN BUILD LIFECYCLE



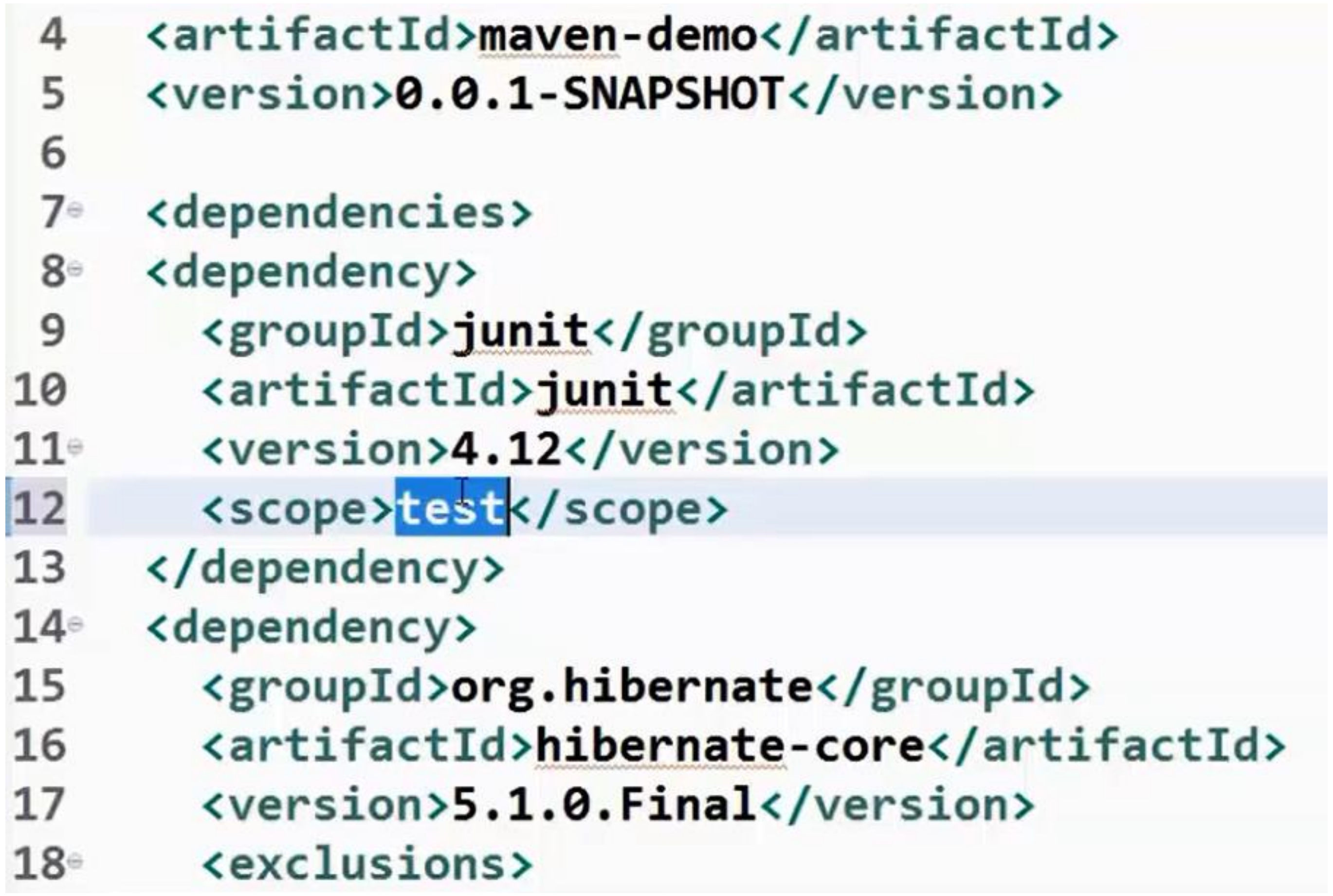
- validate
- compile
- test
- package
- integration-test
- verify
- install
- deploy

EXCLUSION OF DEPENDENCY



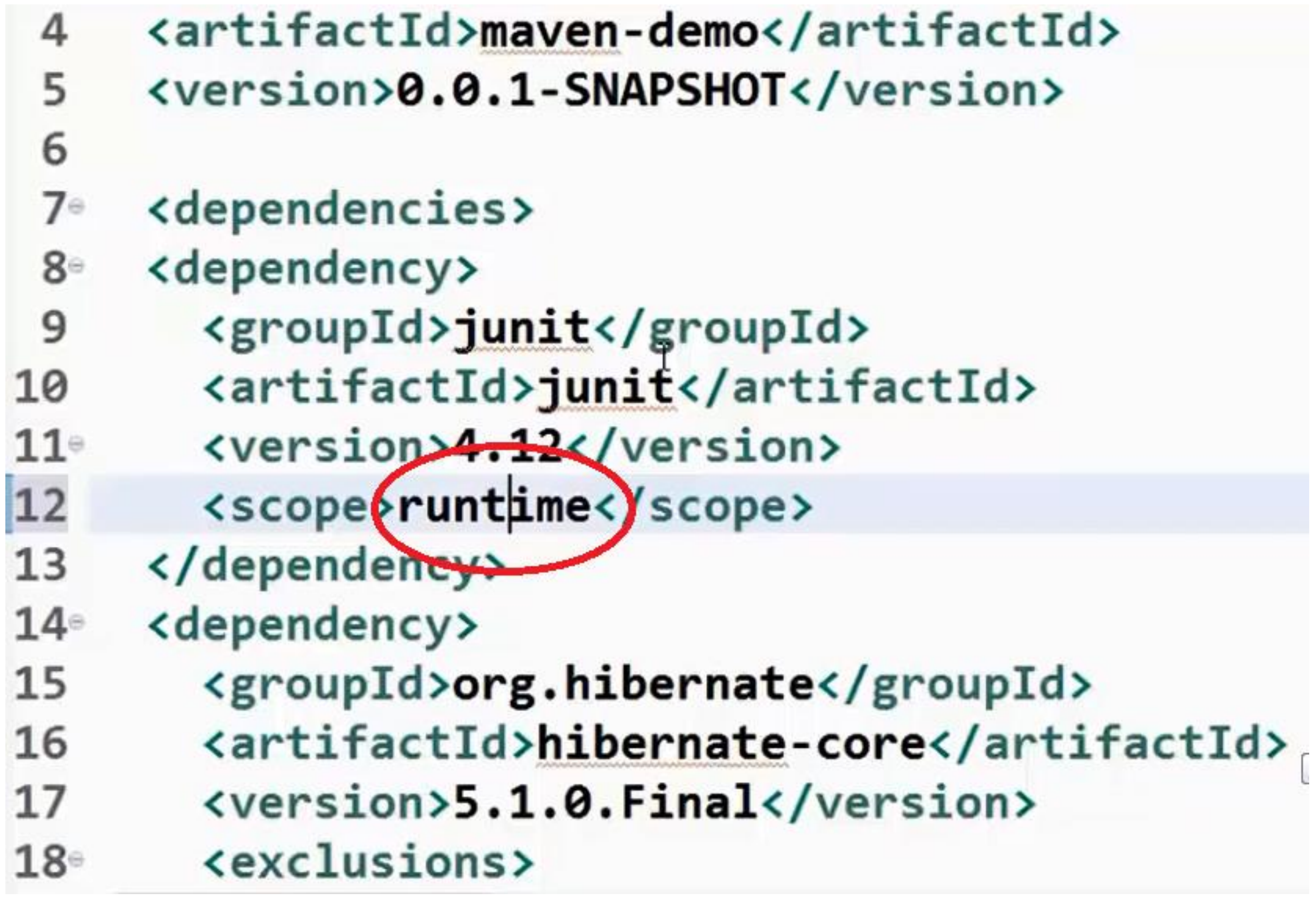
```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.codebind</groupId>
  <artifactId>maven-demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.1.0.Final</version>
    <exclusions>
    <exclusion>
      <groupId>org.jboss.logging</groupId>
      <artifactId>jboss-logging</artifactId>
    </exclusion>
    </exclusions>
 </dependency>
  </dependencies>
</project>
```

MAVEN SCOPE



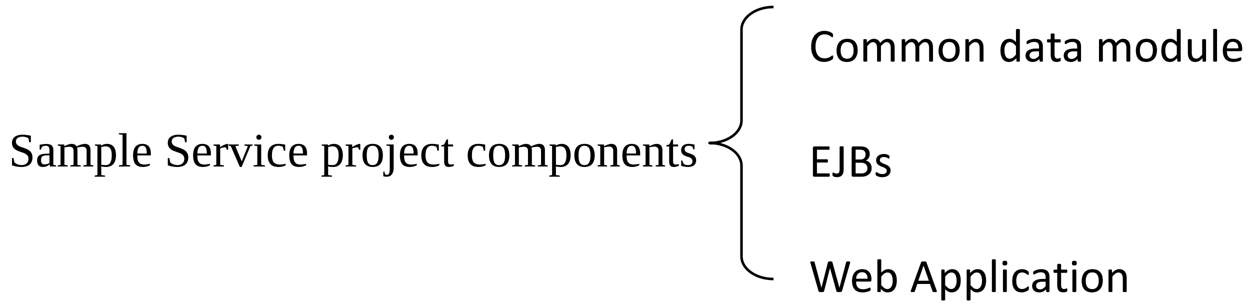
```xml
<artifactId>maven-demo</artifactId>
<version>0.0.1-SNAPSHOT</version>

<dependencies>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.1.0.Final</version>
  <exclusions>
```

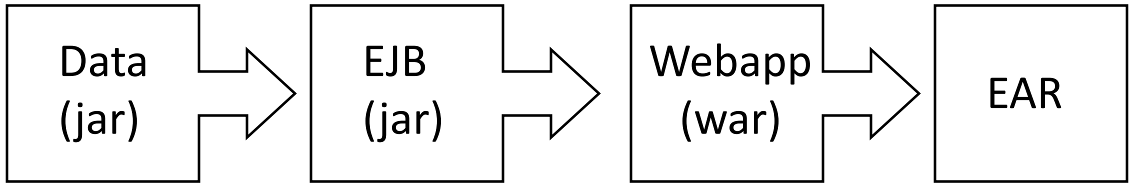
MAVEN SCOPE



```
<artifactId>maven-demo</artifactId>
<version>0.0.1-SNAPSHOT</version>

<dependencies>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.1.0.Final</version>
  <exclusions>
```

Build Process Design





Customizing Maven

- Override plugin properties in
  - project.properties
  - build.properties
- Use maven.xml
  - Override plugin goals
  - Intercept plugin goals with <preGoal/> and <postGoal/>
- Write you own plugin
  - In Java, Jelly, or other scripting language.

Real Life Maven

- Single artifact per project
  - Can be tweaked to deliver multiple artifacts as long as no type conflicts
  - Fine-grained design
- Project Migration/Mavenizing
  - Can co-exist with ant
  - May require different directory structure
- Too Slow?
  - Use console plugin
- Are you ready for maven?
  - Culture change

SUMMARY-MAVEN

- Pros
  - Work out-of-box for standard projects
  - Build assets highly reusable, thanks to core/plugin architecture
  - Build rules are more dynamic
  - Best suited for project integration
  - IDE friendly
- Cons
  - Incomplete documentation
  - Missing convenience details
  - Not yet mature. Still waiting for the R1.

2) FUNCTIONAL PROGRAMMING IN JAVA

THE LAMBDA CALCULUS

- The lambda calculus was introduced in the 1930s by Alonzo Church as a mathematical system for defining computable functions.
- The lambda calculus is equivalent in definitional power to that of Turing machines.
- The lambda calculus serves as the computational model underlying functional programming languages such as Lisp, Haskell, and Ocaml.
- Features from the lambda calculus such as lambda expressions have been incorporated into many widely used programming languages like C++ and now very recently Java 8.

What is the Lambda Calculus?

- The central concept in the lambda calculus is an expression generated by the following grammar which can denote a function definition, function application, variable, or parenthesized expression:

  expr → λ var . expr | expr expr | var | (expr)

- We can think of a lambda-calculus expression as a program which when evaluated by beta-reductions returns a result consisting of another lambda-calculus expression.

Example of a Lambda Expression

- The lambda expression

  $$\lambda x . (+ x 1) 2$$

  represents the application of a function λ x . (+ x 1) with a formal parameter x and a body + x 1 to the argument 2. Notice that the function definition λ x . (+ x 1) has no name; it is an *anonymous function*.
- In Java 8, we would represent this function definition by the Java 8 lambda expression `x -> x + 1`.

More Examples of Java 8 Lambdas

- A Java 8 lambda is basically a method in Java without a declaration usually written as (parameters) -> { body }. Examples,

```
1. (int x, int y) -> { return x + y; }
2. x -> x * x
3. ( ) -> x
```

- A lambda can have zero or more parameters separated by commas and their type can be explicitly declared or inferred from the context.
- Parenthesis are not needed around a single parameter.
- ( ) is used to denote zero parameters.
- The body can contain zero or more statements.
- Braces are not needed around a single-statement body.

WHAT IS FUNCTIONAL PROGRAMMING?

- A style of programming that treats computation as the evaluation of mathematical functions
- Eliminates side effects
- Treats data as being immutable
- Expressions have referential transparency
- Functions can take functions as arguments and return functions as results
- Prefers recursion over explicit for-loops

WHY DO FUNCTIONAL PROGRAMMING?

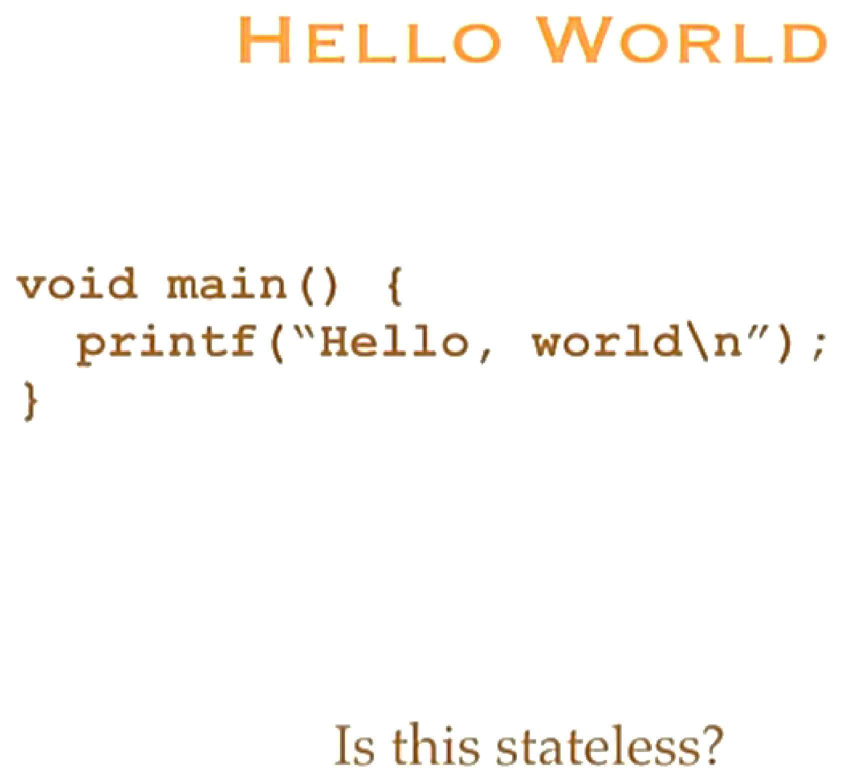
- Allows us to write easier-to-understand, more declarative, more concise programs than imperative programming
- Allows us to focus on the problem rather than the code
- Facilitates parallelism

STATE: WHAT IS IT?

- 1; // a value
- int x=1; // x is an id that has a value.
- x=x+1; // now x has changed state.

- State has failed?
- How can this fail?

HOW ABOUT THIS?

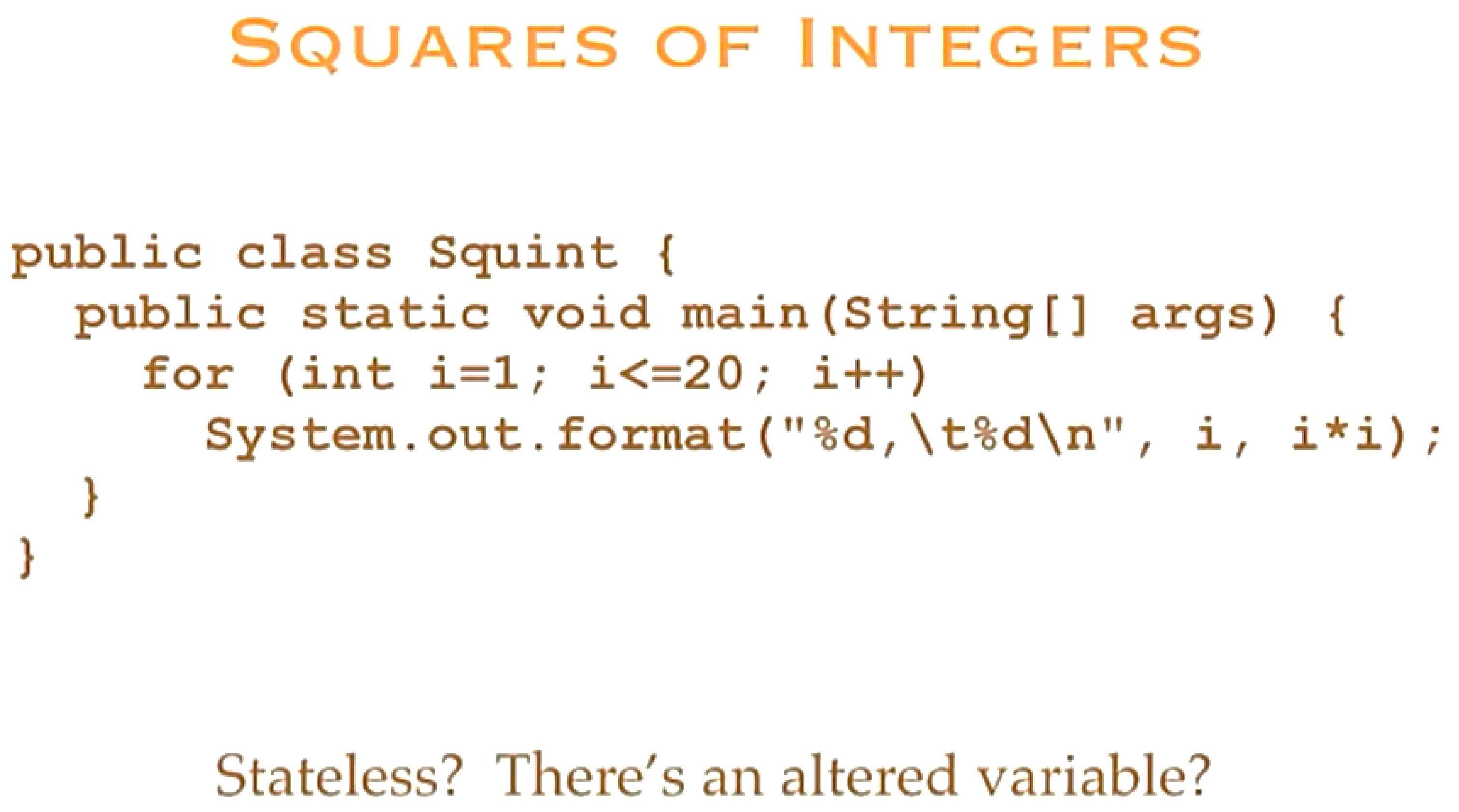


HELLO WORLD

```
void main() {
  printf("Hello, world\n");
}
```

Is this stateless?

A Common Example

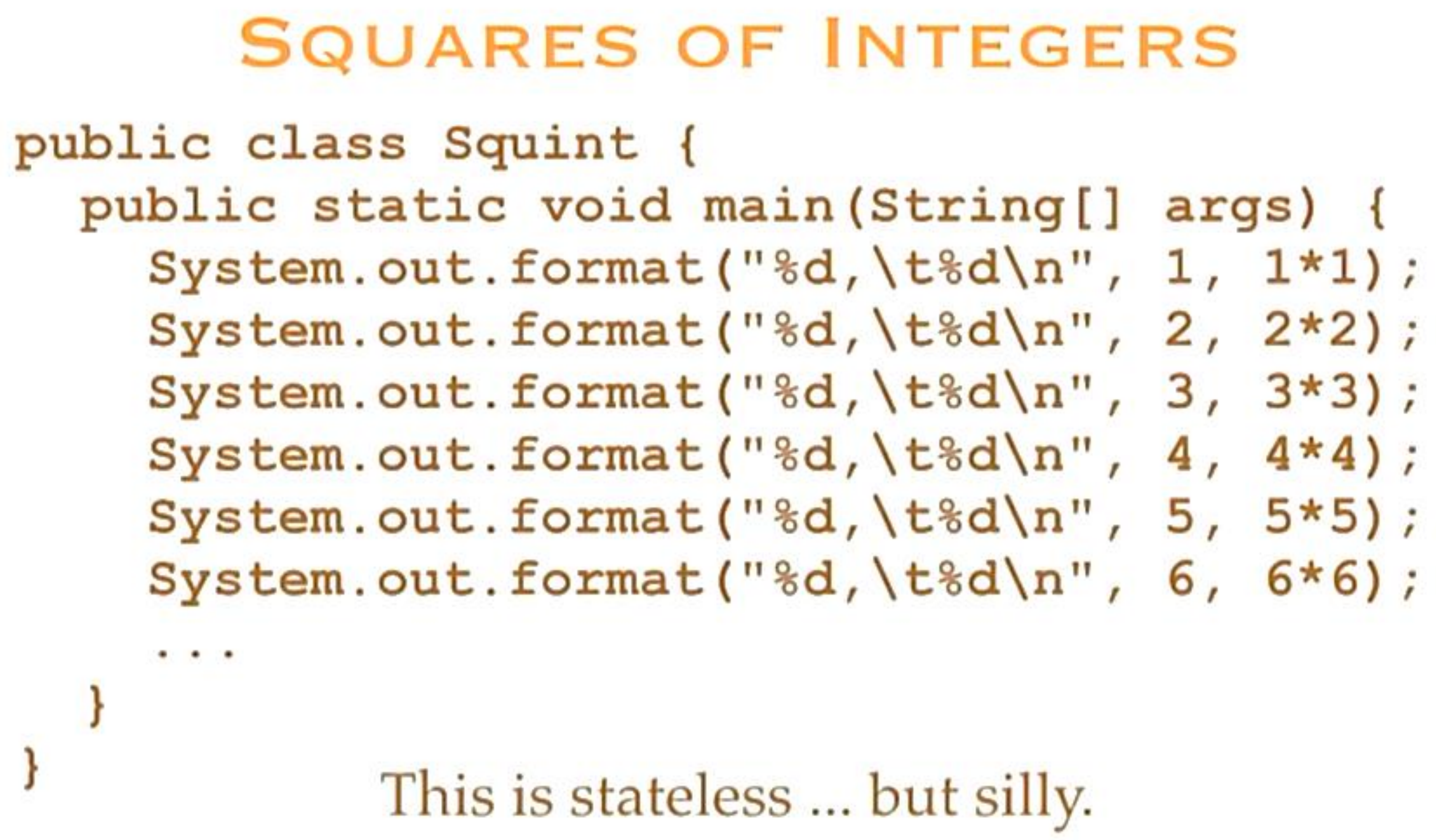


Squares of Integers

```
public class Squint {
  public static void main(String[] args) {
    for (int i=1; i<=20; i++)
      System.out.format("%d,\t%d\n", i, i*i);
  }
}
```

Stateless? There's an altered variable?

STATELESS WAY OF PROGRAMMING 1

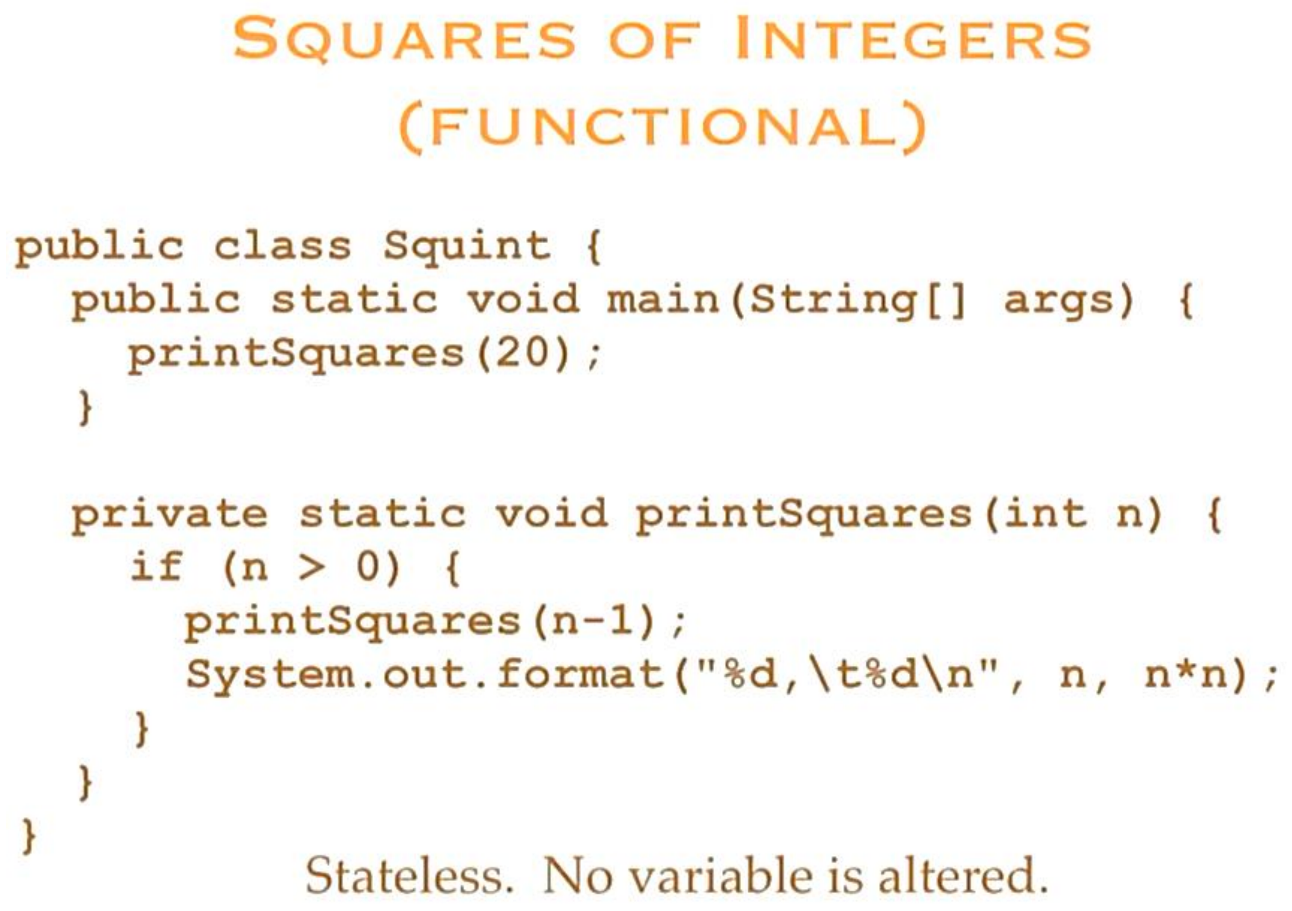


SQUARES OF INTEGERS

```
public class Squint {
  public static void main(String[] args) {
    System.out.format("%d,\t%d\n", 1, 1*1);
    System.out.format("%d,\t%d\n", 2, 2*2);
    System.out.format("%d,\t%d\n", 3, 3*3);
    System.out.format("%d,\t%d\n", 4, 4*4);
    System.out.format("%d,\t%d\n", 5, 5*5);
    System.out.format("%d,\t%d\n", 6, 6*6);
    ...
  }
}
```

This is stateless ... but silly.

STATELESS WAY OF PROGRAMMING 2

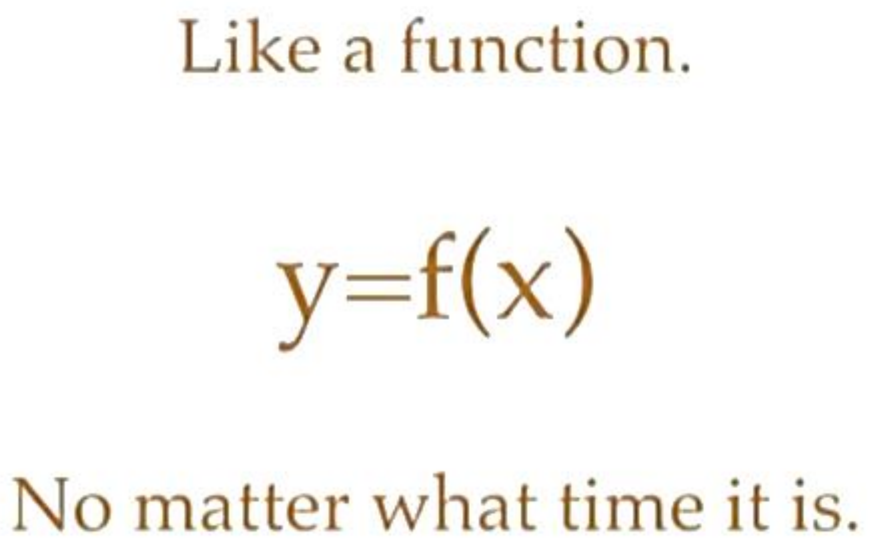


SQUARES OF INTEGERS
(FUNCTIONAL)

```
public class Squint {
  public static void main(String[] args) {
    printSquares(20);
  }

  private static void printSquares(int n) {
    if (n > 0) {
      printSquares(n-1);
      System.out.format("%d,\t%d\n", n, n*n);
    }
  }
}
```

Stateless. No variable is altered.

FUNCTIONAL CONCEPT



Like a function.

$$y=f(x)$$

No matter what time it is.

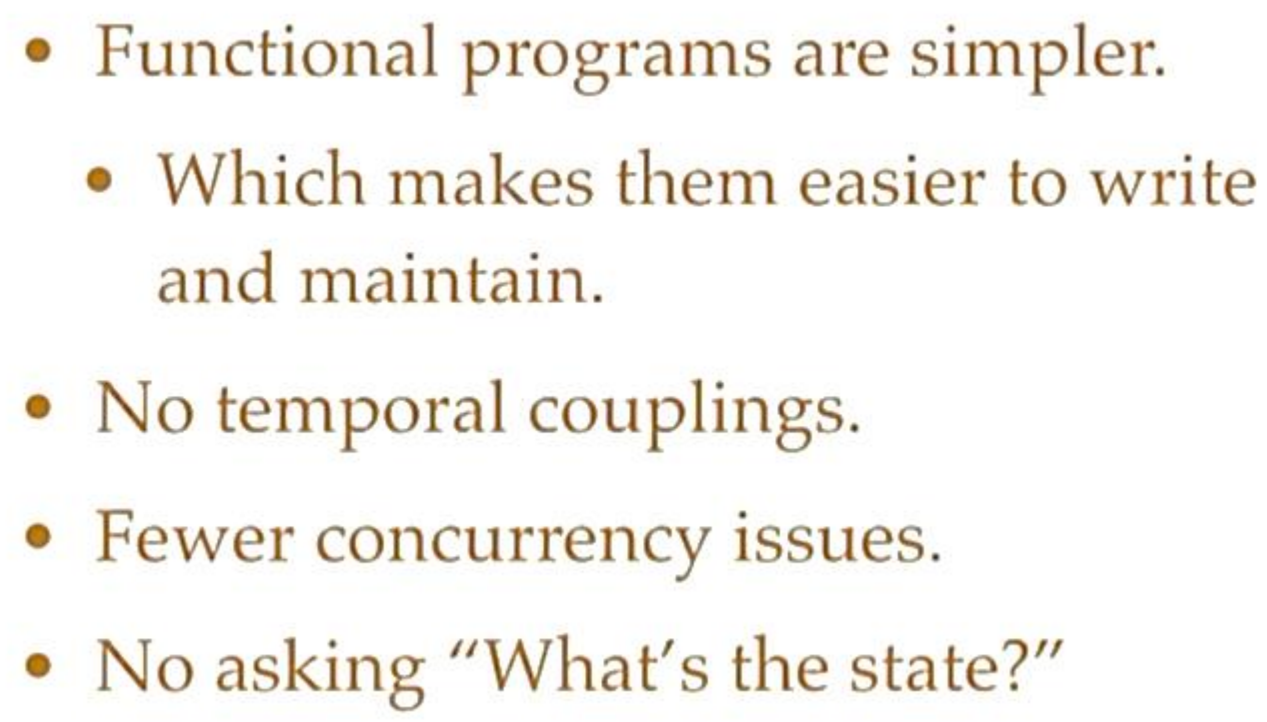
Failure of Substitution

```
AssertEquals(f(x), f(x));
```

Comparison

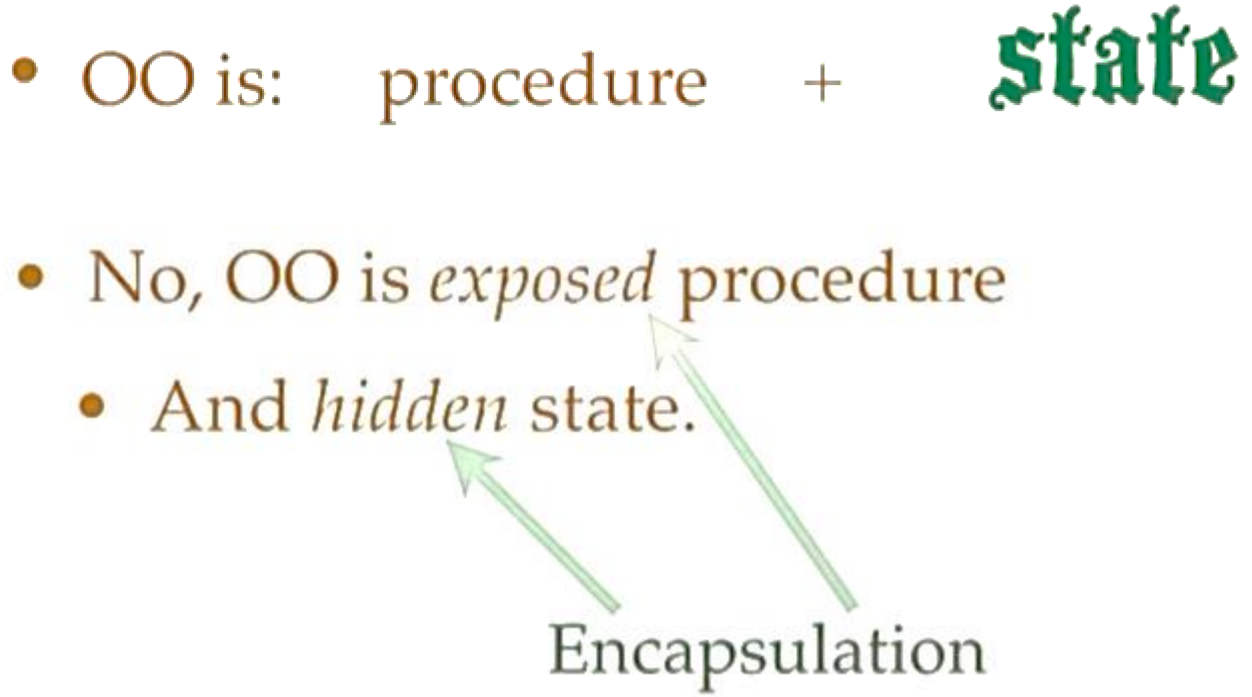
- In which does the bulk of the code obey: `assertEquals(f(x), f(x))`?
- Which is simpler?
- Which is faster?
- Which is more thread safe?
- Which one uses more memory?

We should change the way of coding



- Functional programs are simpler.
  - Which makes them easier to write and maintain.
- No temporal couplings.
- Fewer concurrency issues.
- No asking "What's the state?"

Can OOP be functional programming?



- OO is: procedure + state
- No, OO is *exposed* procedure
  - And *hidden* state.

Encapsulation

IMPOSE DISCIPLINE ON CHANGE OF STATE

- Some state must change.
- Separate functional and non-functional modules
- Use transactions (STM) to manage state change.
- No locking.

Functional programming is like describing your problem to a mathematician. Imperative programming is like giving instructions to an idiot.

—arcus

JAVA 8

- Java 8 is the biggest change to Java since the inception of the language
- Lambdas are the most important new addition
- Java is playing catch-up: most major programming languages already have support for lambda expressions
- A big challenge was to introduce lambdas without requiring recompilation of existing binaries

Benefits of Lambdas in Java 8

- Enabling functional programming
- Writing leaner more compact code
- Facilitating parallel programming
- Developing more generic, flexible and reusable APIs
- Being able to pass behaviors as well as data to functions

Java 8 Lambdas

- Syntax of Java 8 lambda expressions
- Functional interfaces
- Variable capture
- Method references
- Default methods

Example 1:
Print a list of integers with a lambda

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> System.out.println(x));
```

- x -> System.out.println(x) is a lambda expression that defines an anonymous function with one parameter named x of type Integer

Example 2: A Multiline Lambda

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> {
   x += 2;
   System.out.println(x);
});
```

- Braces are needed to enclose a multiline body in a lambda expression.

Example 3: A Lambda with a Defined Local Variable

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach(x -> {
   int y = x * 2;
   System.out.println(y);
});
```

- Just as with ordinary functions, you can define local variables inside the body of a lambda expression

Example 4: A lambda with a declared parameter type

```
List<Integer> intSeq = Arrays.asList(1,2,3);

intSeq.forEach((Integer x -> {
   x += 2;
   System.out.println(x);
});
```

- You can, if you wish, specify the parameter type.

Implementation of Java 8 Lambdas

- The Java 8 compiler first converts a lambda expression into a function
- It then calls the generated function
- For example, `x -> System.out.println(x)` could be converted into a generated static function

```
public static void genName(Integer x) {
    System.out.println(x);
}
```

- But what type should be generated for this function? How should it be called? What class should it go in?

Functional Interfaces

- Design decision: Java 8 lambdas are assigned to functional interfaces.
- A functional interface is a Java interface with exactly one non-default method. E.g.,

```
public interface Consumer<T> {
   void accept(T t);
}
```

- The package `java.util.function` defines many new useful functional interfaces.

ASSIGNING A LAMBDA TO A LOCAL VARIABLE

```
public interface Consumer<T> {
  void accept(T t);
}
void forEach(Consumer<Integer> action {
  for (Integer i:items) {
    action.accept(t);
  }
}
List<Integer> intSeq = Arrrays.asList(1,2,3);

Consumer<Integer> cnsmr = x -> System.out.println(x);
intSeq.forEach(cnsmr);
```

PROPERTIES OF THE GENERATED METHOD

- The method generated from a Java 8 lambda expression has the same signature as the method in the functional interface
- The type is the same as that of the functional interface to which the lambda expression is assigned
- The lambda expression becomes the body of the method in the interface

Variable capture

- Lambdas can interact with variables defined outside the body of the lambda

- Using these variables is called variable capture

Local Variable Capture Example

```
public class LVCExample {
  public static void main(String[] args) {
    List<Integer> intSeq = Arrays.asList(1,2,3);

    int var = 10;
    intSeq.forEach(x -> System.out.println(x + var));
  }
}
```

- Note: local variables used inside the body of a lambda must be final or effectively final

STATIC VARIABLE CAPTURE EXAMPLE

```
public class SVCExample {
  private static int var = 10;
  public static void main(String[] args) {
    List<Integer> intSeq = Arrays.asList(1,2,3);
    intSeq.forEach(x -> System.out.println(x + var));
  }
}
```

METHOD REFERENCES

- Method references can be used to pass an existing function in places where a lambda is expected
- The signature of the referenced method needs to match the signature of the functional interface method

Summary of Method References

Method Reference Type	Syntax	Example
static	ClassName::StaticMethodName	String::valueOf
constructor	ClassName::new	ArrayList::new
specific object instance	objectReference::MethodName	x::toString
arbitrary object of a given type	ClassName::InstanceMethodName	Object::toString

Conciseness with Method References

We can rewrite the statement

```
intSeq.forEach(x -> System.out.println(x));
```

more concisely using a method reference

```
intSeq.forEach(System.out::println);
```

DEFAULT METHODS

Java 8 uses lambda expressions and default methods in conjunction with the Java collections framework to achieve backward compatibility with existing published interfaces

STREAM API

- The new java.util.stream package provides utilities to support functional-style operations on streams of values.
- A common way to obtain a stream is from a collection:

```
Stream<T> stream = collection.stream();
```

- Streams can be sequential or parallel.
- Streams are useful for selecting values and performing actions on the results.

STREAM OPERATIONS

- An intermediate operation keeps a stream open for further operations. Intermediate operations are lazy.
- A terminal operation must be the final operation on a stream. Once a terminal operation is invoked, the stream is consumed and is no longer usable.

Example Intermediate Operations

- `filter` excludes all elements that don’t match a Predicate.

- `map` performs a one-to-one transformation of elements using a Function.

A STREAM PIPELINE

A stream pipeline has three components:

1. A source such as a Collection, an array, a generator function, or an IO channel;
2. Zero or more intermediate operations; and
3. A terminal operation

STREAM EXAMPLE

```
int sum = widgets.stream()
                 .filter(w -> w.getColor() == RED)
                 .mapToInt(w -> w.getWeight())
                 .sum();
```

Here, `widgets` is a `Collection<Widget>`. We create a stream of `Widget` objects via `Collection.stream()`, filter it to produce a stream containing only the red widgets, and then transform it into a stream of `int` values representing the weight of each red widget. Then this stream is summed to produce a total weight.

PARTING EXAMPLE: USING LAMBDAS AND STREAM TO SUM THE SQUARES OF THE ELEMENTS ON A LIST

```
List<Integer> list = Arrays.asList(1,2,3);

int sum = list.stream().map(x -> x*x).reduce((x,y) -> x + y).get();

System.out.println(sum);
```

- Here map(x -> x*x) squares each element and then reduce((x,y) -> x + y) reduces all elements into a single number

What's Wrong With the Following Code?

```
// "strs" is a string array
// We want to get an array
// containing the lengths
// of the strings in "strs"

int[] result =
    new int[strs.length];

for(int i = 0;
    i < strs.length;
    i++) {

  String str = strs[i];
  result[i] = str.length();
}
```

- The code works.
- We are telling the computer ***how*** to compute the result array.
- What if we could tell the computer ***what*** we wanted to compute, and let the computer figure out the best way to do it?

IMPERATIVE VS. FUNCTIONAL

Imperative Programming

- You tell the computer how to perform a task.

Functional Programming

- You tell the computer what you want, and you let the computer (i.e. the compiler or runtime) figure out for itself the best way to do it.

Java 8: Lambdas and Streams

- Lambdas
  - Like “mini functions”
- Streams
  - Similar to lists, but provide functional programming constructs and perform lazy evaluation

```
// Convert List to Stream
// (constant time)
Stream<String> myStream =
  myList.stream();

// Convert Stream to List
// (linear time)
List<String> myList =
  myStream.collect(
    Collectors.toList());
```

FUNCTIONAL ARRAY OPERATIONS

Filter, Map, Reduce, and others

Filter

Keep elements that pass a boolean test

```
// Java 7
List<String> result =
  new ArrayList<String>();

for (String str : myList) {
  if (str.length() > 5) {
    result.add(str);
  }
}
```

```
// Java 8
Stream<String> filtered =
  myStream.filter
    (s -> s.length() > 5);
```

```
# Python (full syntax)
filtered = filter(
  lambda s: len(s) > 5,
  source)

# Python (shortcut syntax)
filtered = \
  (s for s in source
    if len(s) > 5)
```

Find

Return one element satisfying a boolean test

```
// Java 7
String result = null;

for (String str : myList) {
  if (str.length() == 5) {
    result = str;
    break;
  }
}
```

```
// Java 8
Optional<String> result =
  myStream
  .filter
    (s -> s.length() == 5)
  .findAny();
```

```
# Python
filtered = next(
   (s for s in source
     if len(s) == 5),
  "not found")
```

Note: Under the hood, Java 8 and Python both use *lazy evaluation*, so neither language traverses the entire stream if it finds a matching element early in the stream.

Map

Transform elements by a common operation

```
// Java 7
List<Integer> lens =
  new ArrayList<Integer>();

for (String str : myList) {
  lens.add(str.length());
}
```

```
// Java 8
Stream<Integer> lens =
  myStream
  .map(s -> s.length());
```

```
# Python
lens = \
  (len(s) for s in source)
```

Reduce

Apply a binary operation to all elements

```
// Java 7
int totalLen = 0;
for (String str : myList) {
  totalLen += str.length();
}
```

```
// Java 8
int totalLen =
  myStream
  .reduce(0,
   (t,s) -> t + s.length(),
   Integer::sum);
```

```
# Python
from functools import reduce
totalLen = reduce(
  lambda t,s: t + len(s),
  source, 0)
```

Convenience Reducers

Min, Max, Sum, Average, Count, etc.

```
// Java 7
int totalLen = 0;
for (String str : myList) {
  totalLen += str.length();
}

double avgLen =
  ((double)totalLen)
    / myList.size();
```

```
// Java 8
OptionalDouble avgLen =
  myStream
  .mapToInt
    (s -> s.length())
  .average();
```

```
# Python 3.4
from statistics import mean
totalLen = mean(
  len(s) for s in source)
```

Collect

Aggregate elements (for example, grouping)

```java
// Java 7
Map<Integer,List<String>>
  byLength =
    new HashMap<Integer,
       List<String>>();

for (String str : myList) {
  int key = str.length();
  if(!byLength.containsKey(key)){
    byLength.put(key,
      new ArrayList<String>());
  }
  byLength.get(key).add(str);
}
```

```java
// Java 8
Map<Integer,List<String>>
 byLength =
  myStream.collect(
   Collectors.groupingBy(
    String::length));
```

```python
# Python
# caveat: requires a sorted
# list before grouping
totalLen = groupby(
  source,
 key = lambda s: len(s))
```

Flat Map

Do a map and flatten the result by one level

```
// Java 7
List<Character> chars =
  new ArrayList<Character>();

for (String str : myList) {
  for(char ch : str.toCharArray()){
    chars.add(ch);
  }
}
```

```
// Java 8
Stream<Character> chars =
  myStream
  .flatMapToInt
    (s -> s.chars())
  .mapToObj(i -> (char) i);
```

```
# Python
chars = \
  (p for q in (
    list(s) for s in source
  ) for p in q)
```

Sorting

Example: Sort strings by length (longest to shortest) and then alphabetically

Java 8

```java
Stream<String> result =
  myStream
  .sorted(
    Comparator
    .comparing(String::length)
    .reversed()
    .thenComparing(
      Comparator
      .naturalOrder()
    )
  );
```

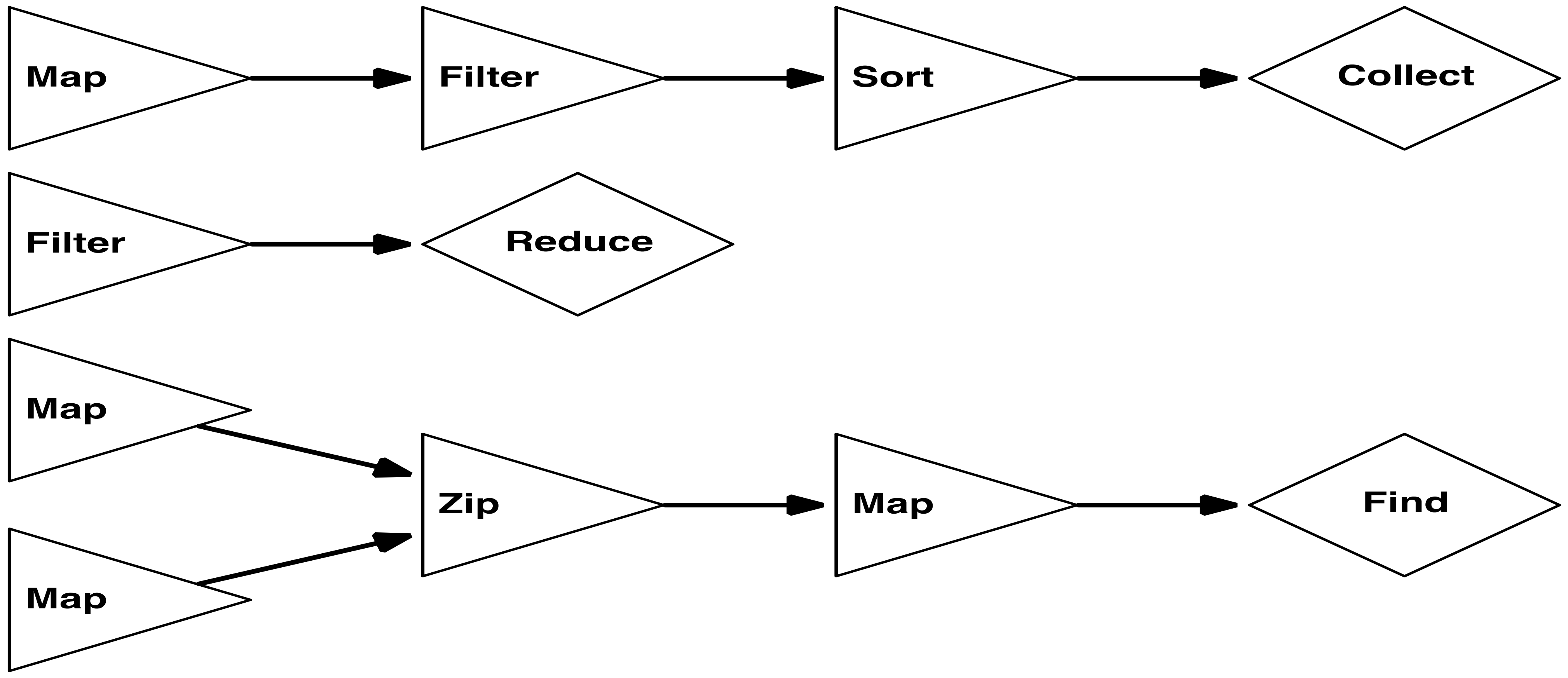
Python

```python
# Python
result = sorted(
  source,
  key = lambda s: (
    -len(source),
    source
  )
)
```

OTHER STREAM METHODS

- Concat
  - Takes two streams and lazily appends them together
- Distinct
  - Removes duplicate elements
- For Each
  - Perform an operation on each element
- Limit
  - Truncate the stream to a given length
- Generate
  - Create an infinite stream by evaluating a lambda function whenever a new element is requested
- Reverse
  - Reverse the stream
- Skip
  - Remove the first *n* elements
- Zip
  - Iterate over two streams at the same time

CHAINING STREAM METHODS



Side-by-Side Comparison

Imperative

- Maintains state.
- Manually parallelizable.
- More code.
- Less intuitive.
- Runtime exceptions, like NullPointerException.

Functional

- Stateless.
- Automatically parallelizable.
- Less code (usually).
- More intuitive.
- More errors caught at compile time.

HOW TO START THINKING IN THE FUNCTIONAL STYLE

- Avoid “for” loops
  - If you find yourself using one, there’s probably a better way to do what you want.
- Use fewer variables
  - See if you can connect multiple steps together into an expressive “chain” of method calls.
- Avoid “side effects”
  - When you run a function, it shouldn’t inadvertently change the value of its input or some other variable.

Language Support

Languages designed for functional programming

- Haskell (1990)
- Erlang (1986)
- Mathematica (1988)
- Elm (2012)
- Lisp (1958), including dialects like Clojure (2007) and Scheme (1975)

Languages with good functional support

- JavaScript (1997)
- Java 8 (2014)
- C++11 (2011)
- Python (1991)
- Ruby (1995)
- Scala (2003)
- Julia :D (2012)

Why has functional programming not been more popular in industry?

- Lack of Knowledge Base
  - Most people learn imperative programming first and get comfortable in that programming style
  - People who’ve never seen the functional programming style before think it’s unreadable
  - This is starting to change, with more and more schools teaching it in their intro classes (MIT, Yale, UC Berkeley, and others)
- Lack of Mainstream Compilers
  - Functional programming has been around since the ’50s, but has mostly been confined to niche languages like Haskell and Lisp
  - The JDK added support last year (2014)

Advanced: More on Java 8 Lambda Functions

- Not Just One Type
  - Depending on the relationship between the argument and return types, lambda functions can be one of many types.
  - To name a few:
    - Predicate<T> takes a T and returns a boolean
    - BinaryOperator<T> takes two T's and returns a T
    - BiFunction<T,U,V> takes a T and a U and returns a V
- Lexical Scoping
  - Lambda functions make a *closure* over the scope *in which there are declared in code*, which is common in other languages but is new in Java 8.
- Optionals
  - In order to avoid returning "null" values, Java 8 has a new Optional<T> type, having methods like:
    - ifPresent(*lambda*)
    - orElse(*alternate value*)

3) Generics

CONCEPTS

- Generalizing Collection Classes
- Using Generics with other Java 1.5 Features
- Integration of Generics with Previous Releases
- User built generic classes

- String [] s = new String [10]
- ArrayList l = new ArrayList ()

What are Generics?

- Generics abstract over Types
- Classes, Interfaces and Methods can be Parameterized by Types
- Generics provide increased readability and type safety

Java Generic Programming

- Java has class Object
  - Supertype of all object types
  - This allows “subtype polymorphism”
    - Can apply operation on class T to any subclass S <: T
- Java 1.0 – 1.4 do not have templates
  - No parametric polymorphism
  - Many consider this the biggest deficiency of Java
- Java type system does not let you cheat
  - Can cast from supertype to subtype
  - Cast is checked at run time

Example Generic Construct: Stack

- Stacks possible for any type of object
  - For any type t, can have type stack_of_t
  - Operations push, pop work for any type
- In C++, would write generic stack class

```
        template <type t> class Stack {
                private: t data;  Stack<t> * next;
                public: void    push (t* x) { … }
                            t*  pop  (    ) { … }
        };
```

- What can we do in Java?

Simple Example using Interfaces

```
interface List<E> {
   void add(E x);
  Iterator<E> iterator();
}
interface Iterator<E> {
   E next();
  boolean hasNext();
}
```

What Generics are not

- Generics are not templates
- Unlike C++, generic declarations are
  - Typechecked at compile time
  - Generics are compiled once and for all
- Generic source code not exposed to user
- No bloat

The type checking with Java 1.5 Generics changes the way one programs (as the next few slides show)

JAVA 1.0 VS GENERICS

```
class Stack {
  void push(Object o)  { ... }
  Object pop() { ... }
  ...}

String s = "Hello";
Stack st = new Stack();
...
st.push(s);
...
s = (String) st.pop();
```

```
class Stack<A> {
  void push(A a) { ... }
  A pop() { ... }
  ...}

String s = "Hello";
Stack<String> st =
        new  Stack<String>();
st.push(s);
...
s = st.pop();
```

Java Generics are type checked

- A generic class may use operations on objects of a parameter type
  - Example: PriorityQueue<T> …   if x.less(y) then …
- Two possible solutions
  - C++: Link and see if all operations can be resolved
  - Java: Type check and compile generics w/o linking
    - This requires programmer to give information about type parameter
    - Example: PriorityQueue<T extends ...>

HOW TO USE GENERICS

```
List<Integer> xs = new LinkedList<Integer>();
xs.add(new Integer(0));
Integer x = xs.iterator.next();
```

Compare with

```
List xs = new LinkedList();
xs.add(new Integer(0));
Integer x = (Integer)xs.iterator.next();
```

Collection Class Example

```
HashMap<Intger, Double> hm =
                    new HashMap<Intger, Double> ();

 // Note Auto-boxing from 15.
hm.put (1,2.0);
double coeff = hm.get(1);
```

Hashmap$_{1.4}$ hm => Hashmap$_{1.5}$<Object, Object>

List Usage: Without Generics

```
List ys = new LinkedList();
ys.add("zero");
List yss;
yss = new LinkedList();
yss.add(ys);
String y = (String)
((List)yss.iterator().next()).iterator().next();

// Evil run-time error
Integer z = (Integer)ys.iterator().next();
```

List Usage: With Generics

```
List<String> ys = new LinkedList<String>();
ys.add("zero");
List<List<String>> yss;
yss = new LinkedList<List<String>>();
yss.add(ys);
String y = yss.iterator().next().iterator().next();

// Compile-time error – much better!
Integer z = ys.iterator().next();
```

List Implementation w/o Generics

```
class LinkedList implements List {
   protected class Node {
                        Object elt;
                        Node next;
                        Node(Object elt){elt = e; next = null;}
   }
   protected Node h, t;
   public LinkedList() {h = new Node(null); t = h;}
   public void add(Object elt){
            t.next = new Node(elt);
            t = t.next;
   }
}
```

Inner Class
Remember these?

LIST IMPLEMENTATION WITH GENERICS

```
class LinkedList<E >implements List<E>
   protected class Node {
   E elt;
   Node next;
            Node(E elt){elt = e; next = null;} }
   protected Node h, t;
   public LinkedList() {h = new Node(null); t = h;}
   public void add(E elt){
            t.next = new Node(elt);
            t = t.next;
   }
   // …
}
```

RECALL THE INTERATOR INTERFACE

```
class LinkedList<E >implements List<E>
    • // …
    public Iterator<E> iterator(){
       return new Iterator<E>(){
                              protected Node p = h.next;
                              public boolean hasNext(){return p != null;}
                              public E next(){
                                          E e = p.elt;
                                          p = p.next;
                                          return e;
                              }
                  }
       }
    }
```

Methods can be generic also

```
interface Function<A,B>{
   B value(A arg);}
interface Ntuple<T> {
   <S> Ntuple<S> map(Function<T,S> f);
}
Ntuple<Integer> nti = ....;
nti.map (new Function<Integer, Integer> {
                        Integer value(Integer i) {
                                    return new Integer(i.intValue()*2);
                        }
                }
            );
```

Example: Generics and Inheritence

1st consider this code w/o Generics

```
class ListUtilities {
   public static Comparable max(List xs) {
            Iterator xi = xs.iterator();
            Comparable w = (Comparable) xi.next();
            while (xi.hasNext()) {
                        Comparable x = (Comparable) xi.next();
                        if (w.compareTo(x) < 0) w = x;
            }
            return w;
   }
}
```

What dangers lurk here?

Consider what happens (and when)

```
List xs = new LinkedList();
xs.add(new Byte(0));
Byte x = (Byte) ListUtilities.max(xs);
List ys = new LinkedList();
ys.add(new Boolean(false));

// Evil run-time error
Boolean y = (Boolean) ListUtilities.max(ys);
```

With Generics we get compile check

```
List<Byte> xs = new LinkedList<Byte>();
xs.add(new Byte(0));
Byte x = ListUtilities.max(xs);
List<Boolean> ys = new LinkedList<Boolean>();
ys.add(new Boolean(false));

// Compile-time error
Boolean y = ListUtilities.max(ys);
```

Generics and Inheritence

- Suppose you want to restrict the type parameter to express some restriction on the type parameter
- This can be done with a notion of subtypes
- Subtypes (weakly construed) can be expressed in Java using inheritance
- So it’s a natural combination to combine inheritance with generics
- A few examples follow

Priority Queue Example

```
interface Comparable<I> {  boolean lessThan(I); }

class PriorityQueue<T extends Comparable<T>> {
        T queue[ ] ;   …
        void insert(T t) {
              ... if ( t.lessThan(queue[i]) ) ...
         }
         T remove() { ... }
         ...
  }
```

Said to be bounded

Bounded Parameterized Types

- The <E extends Number> syntax means that the type parameter of MathBox must be a subclass of the Number class
  - We say that the type parameter is bounded

```
new MathBox<Integer>(5); //Legal
new MathBox<Double>(32.1); //Legal
new MathBox<String>(“No good!”);//Illegal
```

Bounded Parameterized Types

- Inside a parameterized class, the type parameter serves as a valid type. So the following is valid.

```
public class OuterClass<T> {

  private class InnerClass<E extends T> {
    …
  }
  …
}
```

Syntax note: The <A extends B> syntax is valid even if B is an interface.

Bounded Parameterized Types

- Java allows multiple inheritance in the form of implementing multiple interfaces, so multiple bounds may be necessary to specify a type parameter. The following syntax is used then:

  <T extends A & B & C & …>

- **Example**

```
interface A {…}
interface B {…}

class MultiBounds<T extends A & B> {
…
}
```

Another Example …

```
interface LessAndEqual<I> {
          boolean lessThan(I);
          boolean equal(I);
}
class Relations<C extends LessAndEqual<C>> extends C {
          boolean greaterThan(Relations<C> a) {
                     return a.lessThan(this);
           }
           boolean greaterEqual(Relations<C> a) {
                     return greaterThan(a) || equal(a);
           }
          boolean notEqual(Relations<C> a) { ... }
          boolean lessEqual(Relations<C> a) { ... }
          ...
}
```

Generics and Subtyping

- Is the following code snippet legal?

```
List<String> ls = new ArrayList<String>(); //1
List<Object> lo = ls; //2
```

- Line 1 is certainly legal. What about line 2? Is a List of Strings a List of Object. Intuitive answer for most is “sure!”. But wait!

- The following code (if line 2 were allowed) would *attempt to assign an Object to a String!*

```
lo.add(new Object()); // 3
String s = ls.get(0); // 4:
```

For all types P and C *(i.e C is a subtype of P)*
Subtype(P,C) !=> Subtype(Generic<P>,Generic<C>)

Subclassing a Generic Class

```
import java.awt.Color;

public class Subclass extends MyClass<Color> {


    // You almost always need to supply a constructor
    public Subclass(Color color) {
        super(color);
    }


    public static void main(String[ ] args) {
        Subclass sc = new Subclass(Color.GREEN);
        sc.print(Color.WHITE);
    }

}
```

Wildcards

- Consider the problem of writing code that prints out all the elements in a collection before 1.5.

```
void printCollection(Collection c) {
   Iterator i = c.iterator();
   for (k = 0; k < c.size(); k++) {         System.out.println(i.next());
   }
}
```

1ST NAÏVE TRY W/ GENERICS

```
void printCollection(Collection<Object> c) {
   for (Object e : c) {
            System.out.println(e);
   }
}
```

- The problem is that this new version is much less useful than the old one.
- The old code could be called with any kind of collection as a parameter,
- The new code only takes Collection<Object>, which, as is *not* a supertypeof all kinds of collections!

Correct Way – Use Wildcards

- So what *is* the supertype of all kinds of collections?
- Pronounced "collection of unknown" and denoted Collection<?>,
- A collection whose element type matches anything.
- It's called a *wildcard type* for obvious reasons.

```
void printCollection(Collection<?> c) {
  for (Object e : c) {
         System.out.println(e);
  }
}
```

Using Wildcards Again

```
public class Census {
public static void addRegistry(Map<String, ? extends Person>
  registry) { ...}
}...
// Assuming Drivers are a subtype of Person
Map<String, Driver> allDrivers = ...;
Census.addRegistry(allDrivers);
```

Implementing Generics

- Type erasure
  - Compile-time type checking uses generics
  - Compiler eliminates generics by erasing them
    - Compile List<T> to List, T to Object, insert casts

- “Generics are not templates”
  - Generic declarations are typechecked
  - Generics are compiled once and for all
    - No instantiation
    - No “code bloat”

…

How do generics affect my code?

- They don’t – except for the way you’ll code!
- Non-generic code can use generic libraries;
- For example, existing code will run unchanged with generic Collection library

ERASURE

- Erasure erases all generics type argument information at compilation phase.
  - E.g. List<String> is converted to List
  - E.g. String t = stringlist.iterator().next() is converted to String t = (String) stringlist.iterator().next()
- As part of its translation, a compiler will map every parameterized type to its type erasure.

ERASURE

- List <String> l1 = new ArrayList<String>();
- List<Integer> l2 = new ArrayList<Integer>();
- System.out.println(l1.getClass() == l2.getClass());

4) Annotations

Motivation

- Computer scientists and engineers are always trying to add new features to programming languages
- Sometimes they are genuine revisions; sometimes they are orthogonal to, or outside of, the purposes of the language
- The preference is to not rewrite the compiler
- One of the new ease-of-development features in Java 5 are annotations

Previous Examples

- The Java platform has always had various ad hoc annotation mechanisms
  - Javadoc annotations

```
/**
* Locate a value in a
* collection.
* @param value the sought-after value
* @return the index location of the value
* @throws NotFoundException
*/
int search( Object value ) { …
```

  - **`@transient -`** an ad hoc annotation indicating that a field should be ignored by the serialization subsystem
  - **`@deprecated -`** an ad hoc annotation indicating that the method should no longer be used

INTRODUCTION

- Annotations provide data about a program that is not part of the program itself. An **annotation** is an attribute of a program element.
- As of release 5.0, the platform has a general purpose annotation (metadata) facility that permits to define and use **your own** annotation types.
- The facility consists of:
  - a syntax for declaring annotation types
  - a syntax for annotating declarations
  - APIs for reading annotations
  - a class file representation for annotations
  - an annotation processing tool

USAGE

- Annotations have a number of uses, among them:
  - **Information for the compiler -** Annotations can be used by the compiler to detect errors or suppress warnings
  - **Compiler-time and deployment-time processing -** Software tools can process annotation information to generate code, XML files, and so forth
  - **Runtime processing -** Some annotations are available to be examined at runtime (reflection)

Annotation Type Declaration [1/2]

- Similar to normal interface declarations:

```
public @interface RequestForEnhancement {
    int    id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date();    default "[unimplemented]";
}
```

- An at-sign @ precedes the interface keyword
- Each method declaration defines an element of the annotation type
- Methods can have default values
- Once an annotation type is defined, you can use it to annotate declarations

Annotation Type Declaration [2/2]

```
public @interface RequestForEnhancement {
    int    id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date();    default "[unimplemented]";
}
```

- Method declarations should not have any parameters
- Method declarations should not have any throws clauses
- Return types of the method should be one of the following:
  - primitives, String, Class, enum, array of the above types

Annotating Declarations [1/2]

- Syntactically, the annotation is placed in front of the program element's declaration, similar to **`static`** or **`final`** or **`protected`**

```
@RequestForEnhancement(
    id       = 2868724,
    synopsis = "Enable time-travel",
    engineer = "Mr. Peabody",
    date     = "4/1/3007"
)
public static void travelThroughTime(Date destination) { ... }
```

- An annotation instance consists of
  - the "`@`" sign
  - the annotation name
  - a parenthesized list of name-value pairs

Annotating Declarations [2/2]

- In annotations with a single element, the element should be named **`value`**:

```
public @interface Copyright {
    String value();
}
```

- It is permissible to omit the element name and equals sign (=) in a single-element annotation:

```
@Copyright("2002 Yoyodyne Propulsion Systems")
public class OscillationOverthruster { ... }
```

- If no values, then no parentheses needed:

```
public @interface Preliminary { }

@Preliminary public class TimeTravel { ... }
```

What can be annotated?

Annotatable program elements:

- package
- class, including
  - interface
  - enum
- method
- field
- only at compile time
  - local variable
  - formal parameter

Annotations Used by the Compiler

- There are three annotation types that are predefined by the language specification itself:
  - `@Deprecated` – indicates that the marked element is deprecated and should no longer be used
  - `@Override` – informs the compiler that the element is meant to override an element declared in a superclass
  - `@SuppressWarnings` – tells the compiler to suppress specific warnings that it would otherwise generate

Meta-Annotations

- **Meta-annotations** - types designed for annotating annotation-type declarations (annotations-of-annotations)

- Meta-annotations:
  - `@Target` - indicates the targeted elements of a class in which the annotation type will be applicable
    - TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, etc
  - `@Retention` - how long the element holds onto its annotation
    - SOURCE, CLASS, RUNTIME
  - `@Documented` - indicates that an annotation with this type should be documented by the javadoc tool
  - `@Inherited` - indicates that the annotated class with this type is automatically inherited

Annotation Processing

- It's possible to read a Java program and take actions based on its annotations
- To make annotation information available at runtime, the annotation type itself must be annotated with **`@Retention(RetentionPolicy.RUNTIME)`**:

```
@Retention(RetentionPolicy.RUNTIME)
@interface AnnotationForRuntime
{
   // Elements that give information for runtime processing
}
```

- Annotation data can be examined using reflection mechanism, see e.g. **`java.lang.reflect.AccessibleObject`**:
  - `<T extends Annotation> T getAnnotation(Class<T>)`
  - `Annotation[] getAnnotations()`
  - `boolean isAnnotationsPresent(<Class<? extends Annotation>)`

Bigger Example

- The following example shows a program that pokes at classes to see "if they illustrate anything"
- Things to note in example:
  - An annotation may be annotated with itself
  - How annotations meta-annotated with **`Retention(RUNTIME)`** can be accessed via reflection mechanisms

Class Annotation Example

```
@Retention(value=RetentionPolicy.RUNTIME)
@Illustrate( {
   Illustrate.Feature.annotation,
   Illustrate.Feature.enumeration } )
public @interface Illustrate {
    enum Feature {
        annotation, enumeration, forLoop,
        generics, autoboxing, varargs;

        @Override public String toString() {
            return "the " + name() + " feature";
        }
    };
    Feature[] value() default {Feature.annotation};
}
```

```
import java.lang.annotation.Annotation;

@Author(@Name(first="James",last="Heliotis"))
@Illustrate(
    {Illustrate.Feature.enumeration,Illustrate.Feature.forLoop})
public class Suggester {
    @SuppressWarnings({"unchecked"}) // not yet supported
    public static void main( String[] args ) {
        try {
            java.util.Scanner userInput =
                                    new java.util.Scanner( System.in );
            System.out.print( "In what class are you interested? " );
            Class theClass = Class.forName( userInput.next() );
            Illustrate ill =
              (Illustrate)theClass.getAnnotation( Illustrate.class );
```

… continued …

```
            if ( ill != null ) {
                System.out.println( "Look at this class if you'd " +
                                    " like to see examples of" );
                for ( Illustrate.Feature f : ill.value() ) {
                    System.out.println( "\t" + f );
                }
            }
            else {
                System.out.println(
                          "That class will teach you nothing." );
            }
        }
        catch( ClassNotFoundException cnfe ) {
            System.err.println( "I could not find a class named \"" +
                                cnfe.getMessage() + "\"." );
            System.err.println( "Are you sure about that name?" );
        }
    }
}
```

Compilation and Execution

```
$ javac *.java
Note: Suggester.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

$ java Suggester
In what class are you interested? Suggester
Look at this class if you'd like to see examples of
        the enumeration feature
        the forLoop feature

$ java Suggester
In what class are you interested? Illustrate
Look at this class if you'd like to see examples of
        the annotation feature
        the enumeration feature
```

More Execution

```
$ java Suggester
In what class are you interested? Coin
That class will teach you nothing.

$ java Suggester
In what class are you interested? Foo
I could not find a class named "Foo".
Are you sure about that name?
```

Example – JPA Annotations

- When using JPA, you can configure the JPA behavior of your entities using annotations:
  - `@Entity` - designate a plain old Java object (POJO) class as an entity so that you can use it with JPA services
  - `@Table, @Column, @JoinColumn, @PrimaryKeyJoinColumn` – database schema attributes
  - `@OneToOne, @ManyToMany` – relationship mappings
  - `@Inheritance, @DiscriminatorColumn` – inheritance controlling

Example – JUnit Annotations

- Annotations and support for Java 5 are key new features of JUnit 4:
  - `@Test` – annotates test method
  - `@Before, @After`– annotates setUp() and tearDown() methods for each test
  - `@BeforeClass, @AfterClass` – class-scoped setUp() and tearDown()
  - `@Ignore` – do not run test

CONCLUSIONS

- Java annotations are a welcome unification and standardization of approaches to annotating code that language designers and implementers have been doing for decades

- Annotations do not directly affect the semantics of a program

- It is not hard to learn!

5) JUNIT TEST

INTRODUCTION

- Programmers have a nasty habit of not testing their code properly as they develop their software.

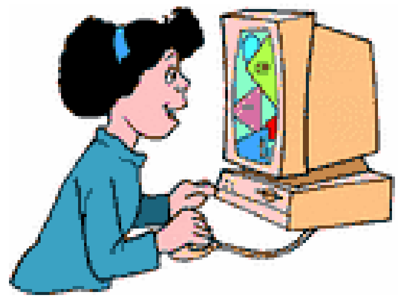
- This prevents you from measuring the progress of development- you can't tell when something starts working or when something stops working.

- Using *JUnit* you can cheaply and incrementally build a test suite that will help you measure your progress, spot unintended side effects, and focus your development efforts.

The Problem,
(The Vicious Cycle)

- Every programmer knows they should write tests for their code. Few do. The universal response to "Why not?" is "I'm in too much of a hurry." This quickly becomes a vicious cycle- the more pressure you feel, the fewer tests you write. The fewer tests you write, the less productive you are and the less stable your code becomes. The less productive and accurate you are, the more pressure you feel.

Example



- In this example pay attention to the interplay of the code and the tests.
- The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, and then write the code that will make it run.

Example,
Currency Handler Program

- The program we write will solve the problem of representing arithmetic with multiple currencies.

- Things get more interesting once multiple currencies are involved. You cannot just convert one currency into another for doing arithmetic since there is no single conversion rate- you may need to compare the value of a portfolio at yesterday's rate and today's rate.

Money.class

- Let's start simple and define a class “Money.class” to represent a value in a single currency.
- We represent the amount by a *simple int*.
- We represent a currency as a string holding the ISO three letter abbreviation (USD, CHF, etc.).

Money.class

- There is an example of the completed Money.class downloadable at :

http://www.site.uottawa.ca/~bob/jUnit/money/

Download all the java files:

- Money.java
- MoneyBag.java
- IMoney.java
- MoneyTest.java

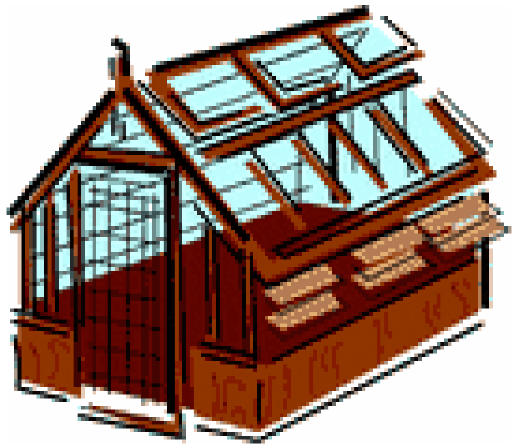
Money.class – Let's code

```
class Money {
    private int fAmount;
    private String fCurrency;
    public Money(int amount, String currency) {
        fAmount= amount;
        fCurrency= currency;
    }

    public int amount() {
        return fAmount;
    }

    public String currency() {
        return fCurrency;
    }

    public Money add(Money m) {
        return new Money(amount()+m.amount(), currency());
    }
}
```

When you add two Moneys of the same currency, the resulting Money has as its amount the sum of the other two amounts.

Let’s Start Testing

- Now, instead of just coding on, we want to get immediate feedback and practice "code a little, test a little, code a little, test a little".
- To implement our tests we use the *JUnit framework.*



Starting JUnit

JUnit comes with a graphical interface to run tests. To start up the JUnit interface, go in to DOS and set all the classpaths.

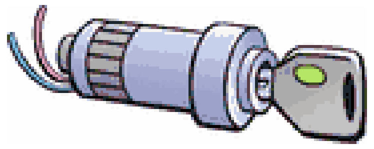
You must set the classpath of the junit.jar file in DOS for JUnit to run. Type:

```
set classpath=%classpath%;INSTALL_DIR\junit3.8.1\junit.jar
```

Also, you may need to set the classpath for your class directory. Type:

```
set classpath=%classpath%;CLASS_DIR
```

Starting JUnit



To run JUnit Type:

for the batch TestRunner type:
java junit.textui.TestRunner NameOfTest

for the graphical TestRunner type:
java junit.awtui.TestRunner NameOfTest

for the Swing based graphical TestRunner type:
java junit.swingui.TestRunner NameOfTest

Testing Money.class

- JUnit defines how to structure your test cases and provides the tools to run them. You implement a test in a subclass of TestCase. To test our Money implementation we therefore define MoneyTest.class as a subclass of TestCase. We add a test method testSimpleAdd, that will exercise the simple version of Money.add() above. A JUnit test method is an ordinary method without any parameters.

Testing Money.class (MoneyTest.class)

```
import junit.framework.*;

public class MoneyTest extends TestCase {
    //…
    public void testSimpleAdd() {
        Money m12CHF= new Money(12, "CHF");  // (1)
        Money m14CHF= new Money(14, "CHF");
        Money expected= new Money(26, "CHF");
        Money result= m12CHF.add(m14CHF);    // (2)
        Assert.assertTrue(expected.equals(result));  // (3)
        Assert.assertEquals(m12CHF, new Money(12, "CHF")); // (4)
    }
}
```

(1) Code which creates the objects we will interact with during the test. This testing context is commonly referred to as a test's *fixture*. All we need for the testSimpleAdd test are some Money objects.

(2) Code which exercises the objects in the fixture.

(3) Code which verifies the results. If not true, JUnit will return an error.

(4) Since assertions for equality are very common, there is also an Assert.assertEquals convenience method. If not equal JUnit will return an error.

Assert Function

- In the new release of JUnit the Assert function is simplified. Instead of writing Assert.assertTrue(…); you can simply write assertTrue(…);.

Assert Functions

- assertTrue() – Returns error if not true.
- assertFalse() – Returns error if not false.
- assertEquals() – Returns error if not equal.
- assertNotSame(Object, Object) – Returns error if they are the same.

Write the “equals” method in Money.class

- The equals method in Object returns true when both objects are the same. However, Money is a *value object*. Two Monies are considered equal if they have the same currency and value.

Write the "Equals" method in Money.class

```
public boolean equals(Object anObject) {
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
            && amount() == aMoney.amount();
    }
    return false;
}
```

With an equals method in hand we can verify the outcome of testSimpleAdd. In JUnit you do so by a calling **Assert.assertTrue**, which triggers a failure that is recorded by JUnit when the argument isn't true.

SETUP METHOD FOR TESTING

- Now that we have implemented two test cases we notice some code duplication for setting-up the tests. It would be nice to reuse some of this test set-up code. In other words, we would like to have a common fixture for running the tests. With JUnit you can do so by storing the fixture's objects in instance variables of your **TestCase** subclass and initialize them by overridding the setUp method. The symmetric operation to setUp is tearDown which you can override to clean up the test fixture at the end of a test. Each test runs in its own fixture and JUnit calls setUp and tearDown for each test so that there can be no side effects among test runs.

MoneyTest.class

```
public class MoneyTest extends TestCase {
    private Money f12CHF;
    private Money f14CHF;

    protected void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
    }

    public void testEquals() {
        Assert.assertTrue(!f12CHF.equals(null));
        Assert.assertEquals(f12CHF, f12CHF);
        Assert.assertEquals(f12CHF, new Money(12, "CHF"));
        Assert.assertTrue(!f12CHF.equals(f14CHF));
    }

    public void testSimpleAdd() {
        Money expected= new Money(26, "CHF");
        Money result= f12CHF.add(f14CHF);
        Assert.assertTrue(expected.equals(result));
    }
}
```

setUp method for initializing inputs

RUNNING TESTS STATICALLY OR DYNAMICALLY

Two additional steps are needed to run the two test cases:

- define how to run an individual test case,
- define how to run a *test suite*.

JUnit supports two ways of running single tests:

- static
- dynamic

Calling Tests Statically

- In the static way you override the runTest method inherited from TestCase and call the desired test case. A convenient way to do this is with an anonymous inner class. Note that each test must be given a name, so you can identify it if it fails.

```
TestCase test= new MoneyTest("simple add") {
    public void runTest() {
        testSimpleAdd();
    }
}
```

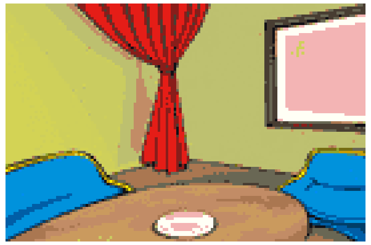
Calling Tests Dynamically

The dynamic way to create a test case to be run uses reflection to implement runTest. It assumes the name of the test is the name of the test case method to invoke. It dynamically finds and invokes the test method. To invoke the testSimpleAdd test we therefore construct a MoneyTest as shown below:

```
TestCase test= new MoneyTest("testSimpleAdd");
```

The dynamic way is more compact to write but it is less static type safe. An error in the name of the test case goes unnoticed until you run it and get a NoSuchMethodException. Since both approaches have advantages, we decided to leave the choice of which to use up to you.

Test Suites in JUnit



- As the last step to getting both test cases to run together, we have to define a test suite. In JUnit this requires the definition of a static method called suite. The suite method is like a main method that is specialized to run tests. Inside suite you add the tests to be run to a **TestSuite** object and return it. A TestSuite can run a collection of tests. TestSuite and TestCase both implement an interface called Test which defines the methods to run a test. This enables the creation of test suites by composing arbitrary TestCases and TestSuites. In short TestSuite is a Composite [1]. The next code illustrates the creation of a test suite with the dynamic way to run a test.

TEST SUITES IN JUNIT

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new MoneyTest("testEquals"));
    suite.addTest(new MoneyTest("testSimpleAdd"));
    return suite;
}
```

Since JUnit 2.0 there is an even simpler dynamic way. You only pass the class with the tests to a TestSuite and it extracts the test methods automatically.

```
public static Test suite() {
    return new TestSuite(MoneyTest.class);
}
```

Static Test Suite

Here is the corresponding code using the static way.

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(
        new MoneyTest("money equals") {
            protected void runTest() { testEquals(); }
        }
    );

    suite.addTest(
        new MoneyTest("simple add") {
            protected void runTest() { testSimpleAdd(); }
        }
    );
    return suite;
}
```

Running Your Tests

To run JUnit Type:

for the batch TestRunner type:

java junit.textui.TestRunner NameOfTest

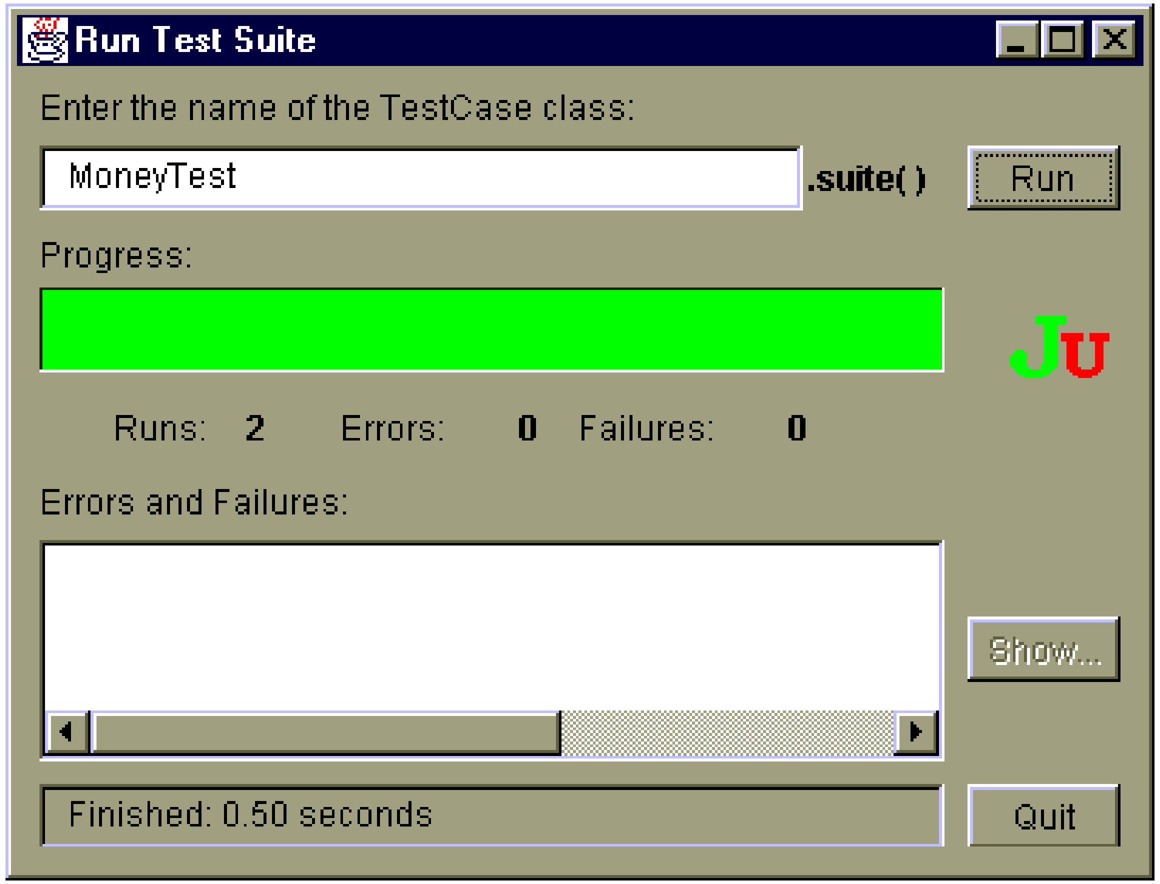
for the graphical TestRunner type:

java junit.awtui.TestRunner NameOfTest

for the Swing based graphical TestRunner type:

java junit.swingui.TestRunner NameOfTest

SUCCESSFUL TEST



Runs: 2 Errors: 0 Failures: 0

Finished: 0.50 seconds

When the test results are valid and no errors are found, the progress bar will be completely green. If there are 1 or more errors, the progress bar will turn red. The errors and failures box will notify you to where to bug has occurred.

EXAMPLE CONTINUED (MONEY BAGS)

- After having verified that the simple currency case works we move on to multiple currencies. As mentioned above the problem of mixed currency arithmetic is that there isn't a single exchange rate. To avoid this problem we introduce a *MoneyBag* which defers exchange rate conversions.

Example:

- Adding 12 Swiss Francs to 14 US Dollars is represented as a bag containing the two Monies 12 CHF and 14 USD.
- Adding another 10 Swiss francs gives a bag with 22 CHF and 14 USD.

MoneyBag.class

```
class MoneyBag {
    private Vector fMonies= new Vector();

    MoneyBag(Money m1, Money m2) {
        appendMoney(m1);
        appendMoney(m2);
    }

    MoneyBag(Money bag[]) {
        for (int i= 0; i < bag.length; i++)
            appendMoney(bag[i]);
    }
}
```

appendMoney is an internal helper method that adds a Money to the list of Moneys and takes care of consolidating Monies with the same currency

MoneyBag Test

- We skip the implementation of equals and only show the testBagEquals method. In a first step we extend the fixture to include two MoneyBags.

```
protected void setUp() {
    f12CHF= new Money(12, "CHF");
    f14CHF= new Money(14, "CHF");
    f7USD=  new Money( 7, "USD");
    f21USD= new Money(21, "USD");
    fMB1= new MoneyBag(f12CHF, f7USD);
    fMB2= new MoneyBag(f14CHF, f21USD);
}
//With this fixture the testBagEquals test case becomes:
public void testBagEquals() {
    Assert.assertTrue(!fMB1.equals(null));
    Assert.assertEquals(fMB1, fMB1);
    Assert.assertTrue(!fMB1.equals(f12CHF));
    Assert.assertTrue(!f12CHF.equals(fMB1));
    Assert.assertTrue(!fMB1.equals(fMB2));
}
```

Fixing the add Method

- Following "code a little, test a little" we run our extended test with JUnit and verify that we are still doing fine. With MoneyBag in hand, we can now fix the add method in Money.

```
public Money add(Money m) {
    if (m.currency().equals(currency()) )
        return new Money(amount()+m.amount(),
currency());
    return new MoneyBag(this, m);
}
```

- As defined above this method will not compile since it expects a Money and not a MoneyBag as its return value.

IMoney Interface

- With the introduction of MoneyBag there are now two representations for Moneys which we would like to hide from the client code. To do so we introduce an interface IMoney that both representations implement. Here is the IMoney interface:

```
interface IMoney {
    public abstract IMoney add(IMoney aMoney);
    //…
}
```

More Testing

- To fully hide the different representations from the client we have to support arithmetic between all combinations of Moneys with MoneyBags. Before we code on, we therefore define a couple more test cases. The expected MoneyBag results use the convenience constructor shown above, initializing a MoneyBag from an array.

```
public void testMixedSimpleAdd() {
    // [12 CHF] + [7 USD] == {[12 CHF][7 USD]}
    Money bag[]= { f12CHF, f7USD };
    MoneyBag expected= new MoneyBag(bag);
    Assert.assertEquals(expected, f12CHF.add(f7USD));
}
```

Update the Test Suite

The other tests follow the same pattern:

- testBagSimpleAdd - to add a MoneyBag to a simple Money
- testSimpleBagAdd - to add a simple Money to a MoneyBag
- testBagBagAdd - to add two MoneyBags

Next, we extend our test suite accordingly:

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new MoneyTest("testMoneyEquals"));
    suite.addTest(new MoneyTest("testBagEquals"));
    suite.addTest(new MoneyTest("testSimpleAdd"));
    suite.addTest(new MoneyTest("testMixedSimpleAdd"));
    suite.addTest(new MoneyTest("testBagSimpleAdd"));
    suite.addTest(new MoneyTest("testSimpleBagAdd"));
    suite.addTest(new MoneyTest("testBagBagAdd"));
    return suite;
}
```

IMPLEMENTATION

- Having defined the test cases we can start to implement them. The implementation challenge here is dealing with all the different combinations of Money with MoneyBag. Double dispatch is an elegant way to solve this problem. The idea behind double dispatch is to use an additional call to discover the kind of argument we are dealing with. We call a method on the argument with the name of the original method followed by the class name of the receiver.

The add method in Money and MoneyBag becomes:

Implementation

```
class Money implements IMoney {
    public IMoney add(IMoney m) {
        return m.addMoney(this);
    }
    //…
}
class MoneyBag implements IMoney {
    public IMoney add(IMoney m) {
        return m.addMoneyBag(this);
    }
    //…
}
```

In order to get this to compile we need to extend the interface of IMoney with the two helper methods:

```
interface IMoney {
//…
    IMoney addMoney(Money aMoney);
    IMoney addMoneyBag(MoneyBag aMoneyBag);
}
```

Implementation

- To complete the implementation of double dispatch, we have to implement these methods in Money and MoneyBag. This is the implementation in Money.

```
public IMoney addMoney(Money m) {
    if (m.currency().equals(currency()) )
        return new Money(amount()+m.amount(), currency());
    return new MoneyBag(this, m);
}

public IMoney addMoneyBag(MoneyBag s) {
    return s.addMoney(this);
}
```

Implementation

- Here is the implemenation in MoneyBag which assumes additional constructors to create a MoneyBag from a Money and a MoneyBag and from two MoneyBags.

```
public IMoney addMoney(Money m) {
    return new MoneyBag(m, this);
}

public IMoney addMoneyBag(MoneyBag s) {
    return new MoneyBag(s, this);
}
```

Are there more errors that can occur?

- We run the tests, and they pass. However, while reflecting on the implementation we discover another interesting case. What happens when as the result of an addition a MoneyBag turns into a bag with only one Money? For example, adding -12 CHF to a Moneybag holding 7 USD and 12 CHF results in a bag with just 7 USD. Obviously, such a bag should be equal with a single Money of 7 USD. To verify the problem you can make a test and run it.

REVIEW

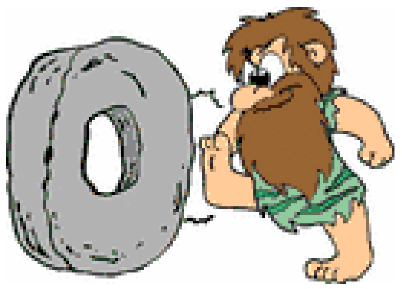
- We wrote the first test, testSimpleAdd, immediately after we had written add(). In general, your development will go much smoother if you write tests a little at a time as you develop. It is at the moment that you are coding that you are imagining how that code will work. That's the perfect time to capture your thoughts in a test.

Review

- We refactored the existing tests, testSimpleAdd and testEqual, as soon as we introduced the common setUp code. Test code is just like model code in working best if it is factored well. When you see you have the same test code in two places, try to find a way to refactor it so it only appears once.

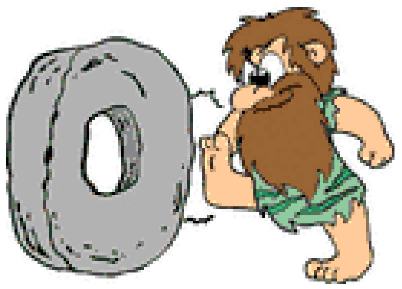
- We created a suite method, and then extended it when we applied Double Dispatch. Keeping old tests running is just as important as making new ones run. The ideal is to always run all of your tests. Sometimes that will be too slow to do 10 times an hour. Make sure you run all of your tests at least daily.

Testing Practices



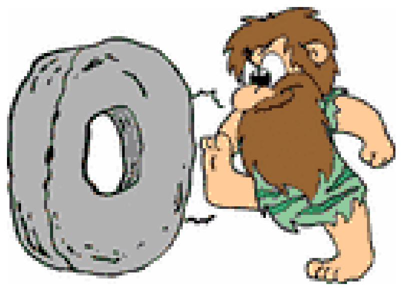
- **Martin Fowler makes this easy for you. He says, "Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead." At first you will find that you have to create new fixtures all the time, and testing will seem to slow you down a little. Soon, however, you will begin reusing your library of fixtures and new tests will usually be as simple as adding a method to an existing TestCase subclass**

Testing Practices



- You can always write more tests. However, you will quickly find that only a fraction of the tests you can imagine are actually useful. What you want is to write tests that fail even though you think they should work, or tests that succeed even though you think they should fail. Another way to think of it is in cost/benefit terms. You want to write tests that will pay you back with information.

Testing Practices



Here are a couple of the times that you will receive a reasonable return on your testing investment:

- During Development- When you need to add new functionality to the system, write the tests first. Then, you will be done developing when the test runs.
- During Debugging- When someone discovers a defect in your code, first write a test that will succeed if the code is working. Then debug until the test succeeds.

Maven Errors Occurred:

- 1) During running test: Nothing to test.

---Solution: write at least some code/method in the class.

2) Test failure: //fail("Not yet implemented"); commenting out this line

3) Runtime error. Can not find jre.

---Solution: the jre file might be corrupted. Try deleting that and download new.