

Advanced Java

Daniel Hinojosa

Generics Topics

- Generics
- Get Put Principles
- Wildcards
- Advanced I/O

Lab: Pre-Class Check

Before we begin it is assumed that all of you have the following tools installed:

- JDK 1.8.x
- Maven 3.3.x

To verify that all your tools work as expected

```
% javac -version
javac 1.8.0_65

% java -version
java version "1.8.0_65"
Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)


% mvn -v
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T09:41:47-07:00)
Maven home: /usr/lib/mvn/apache-maven-3.3.9
Java version: 1.8.0_65, vendor: Oracle Corporation
Java home: /usr/lib/jvm/jdk1.8.0_65/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.4.0-34-generic", arch: "amd64", family: "unix"
```

NOTE

The JDK 8 Version doesn't have to be exact as long as it is Java 8.




Lab: Download the three day project

From https://github.com/dhinojosa/advanced_java_spike download the project .zip file and extract it into your favorite location.



This repository Search

Pull requests Issues Gist

dhinojosa / advanced_java_spike


Unwatch 1 Star 0 Fork 0

<> Code 0 Issues 0 Pull requests Wiki Pulse Graphs Settings

Advanced Java Spike with Projects for Java 8, Advanced Java, and Google Guice — Edit

1 commit 1 branch 0 releases 1 contributor

Branch: master New pull request Create new file Upload files Find file Clone or download

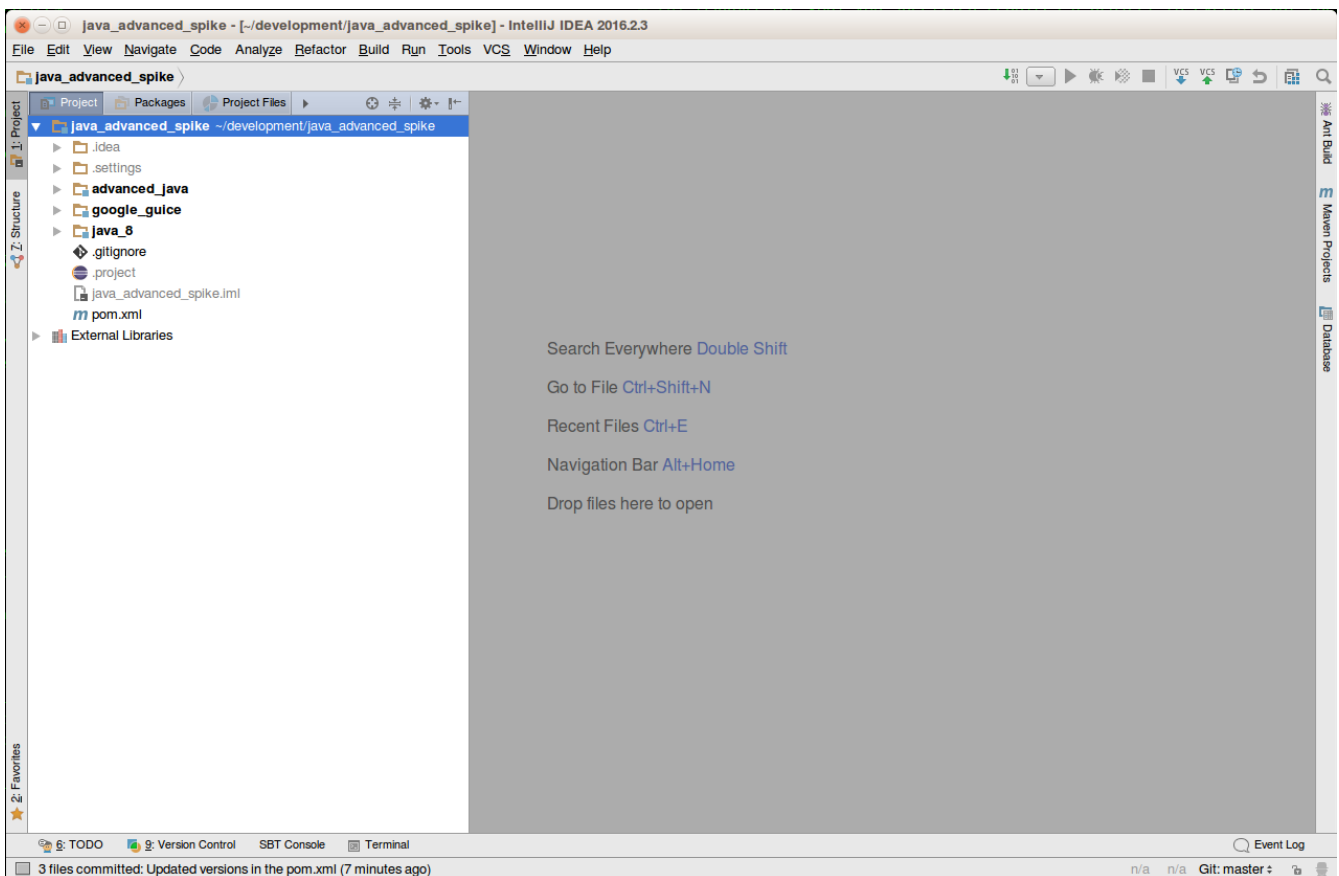
 dhinojosa initial commit with structure Latest commit f7c5689 6 minutes ago

advanced_java	initial commit with structure	6 minutes ago
google_guice	initial commit with structure	6 minutes ago
java_8	initial commit with structure	6 minutes ago
.gitignore	initial commit with structure	6 minutes ago
pom.xml	initial commit with structure	6 minutes ago

Help people interested in this repository understand your project by adding a README. Add a README

Optional Lab: Open Project in IntelliJ

Once downloaded and extracted to your favorite location, In IntelliJ Open The Project, IntelliJ will recognize it as a Maven project and you are good to go.

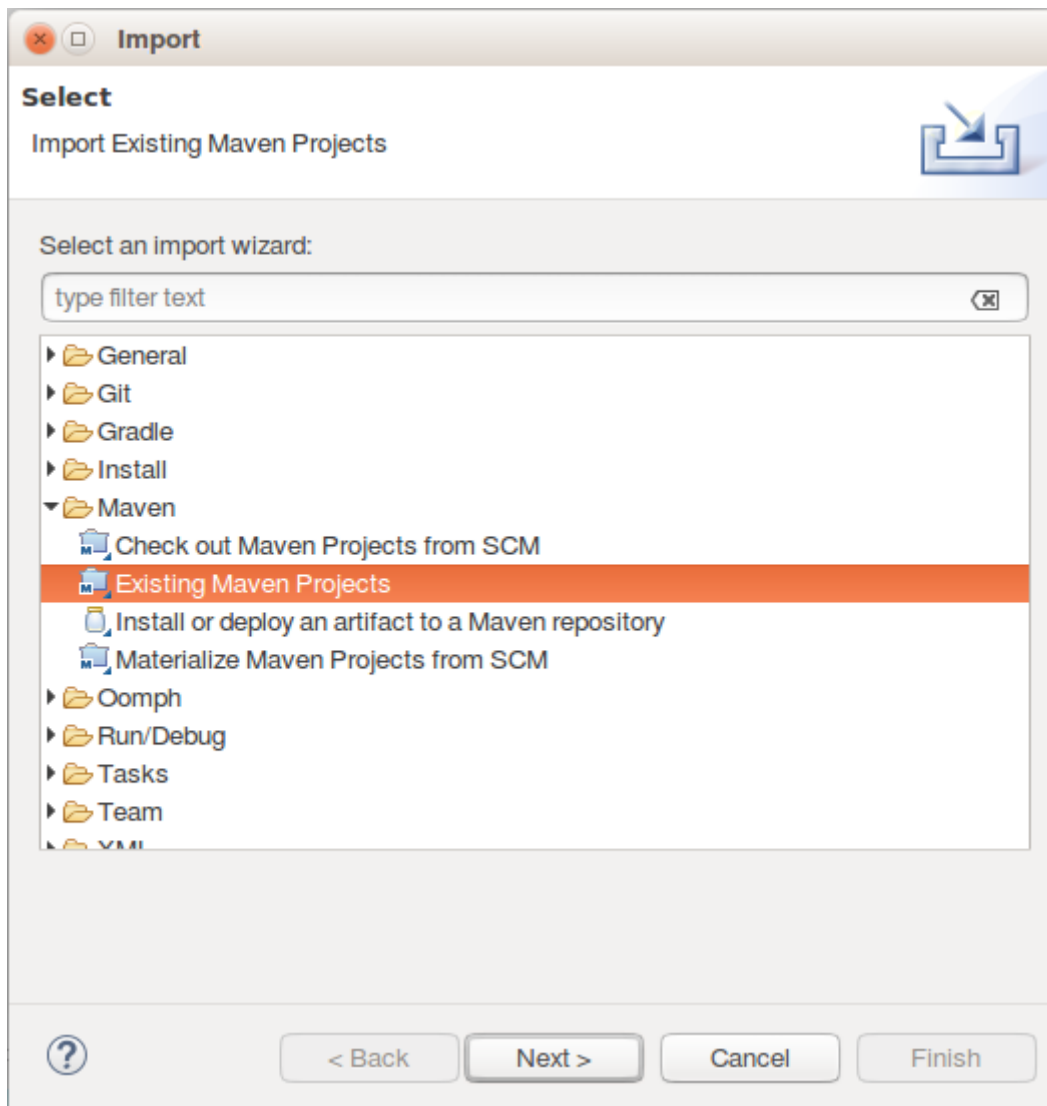


Optional Lab: Open Project in Eclipse

Once downloaded and extracted:

Step 1: Select *File > Import Project* in the menu.

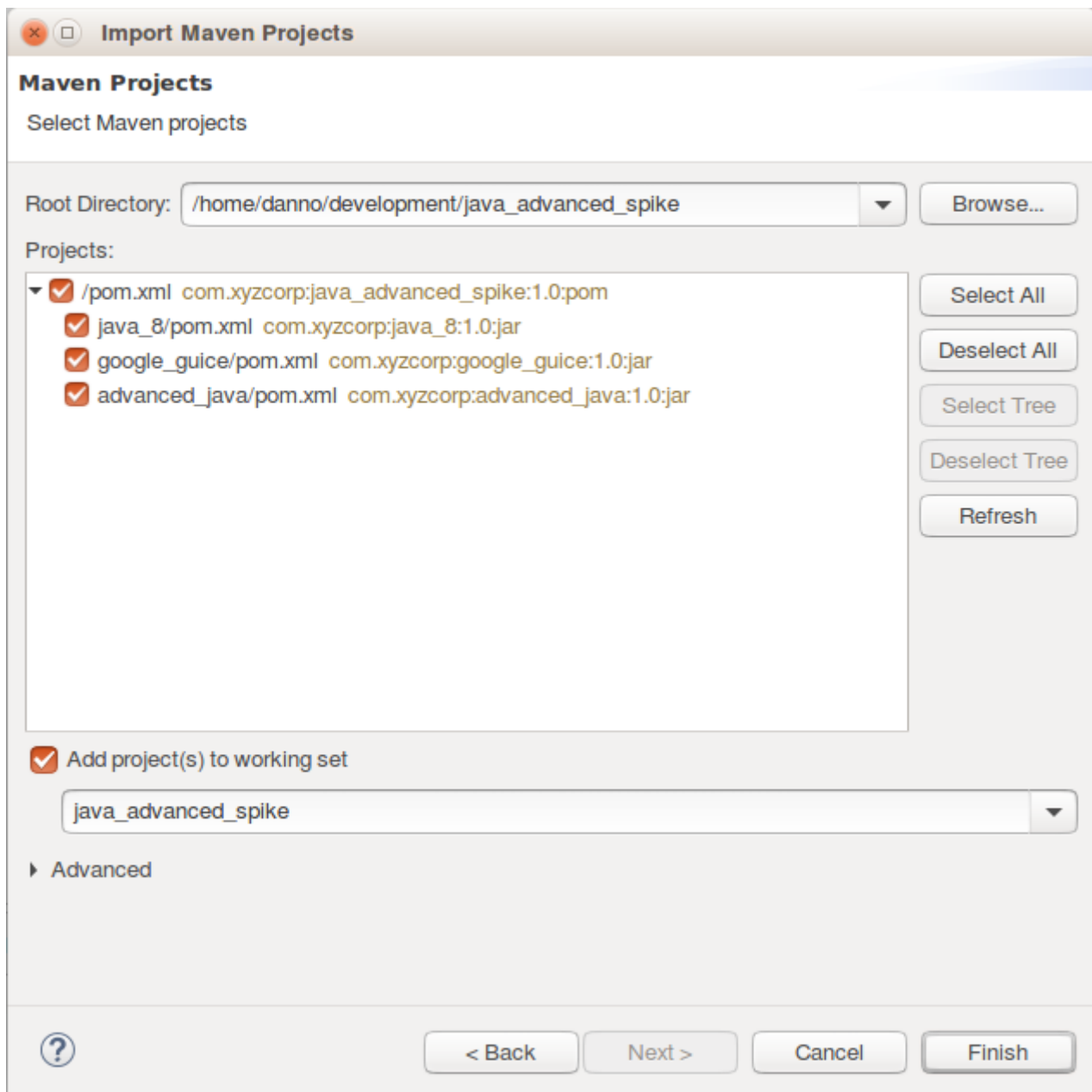
Step 2: In the following dialog box:



- Open the *Maven* category
- Select *Import Existing Maven Projects*

Optional Lab: Open Project in Eclipse (Continued)

Step 3:



- Click the *Browse:* button next to *Root Directory*
- Select the location of your *java_advanced_spike* directory.

Step 4: Click *Finish*

Generics

- Add stability to your code by making more of your bugs detectable at *compile time* * Enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.
- Provide a way for you to re-use the same code with different inputs, requiring less code
- Eliminates Casting
- One of the harder concepts in Java Programming since JDK 5.

Eliminating Casting

Before:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

After:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);    // no cast
```

Diamond Operator

In Java SE 7, you can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

```
List<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0);    // no cast
```

Generics with **for**

```
List<Integer> ints = Arrays.asList(1,2,3);
int s = 0;
for (int n : ints) { s += n; }
```

Unreadability without Generics

```
List ints = Arrays.asList( new Integer[] {
    new Integer(1), new Integer(2), new Integer(3)
} );
int s = 0;
for (Iterator it = ints.iterator(); it.hasNext(); ) {
    int n = ((Integer)it.next()).intValue();
    s += n;
}
```

Unreadability with Arrays

```
int[] ints = new int[] { 1,2,3 };
int s = 0;
for (int i = 0; i < ints.length; i++) { s += ints[i]; }
```

Notes:

- Less flexible
- Less readable

Erasure

Comparing these two sets of code:

The following uses generics...

```
List<String> words = new ArrayList<String>();
words.add("Hello ");
words.add("world!");
String s = words.get(0)+words.get(1);
```

The following doesn't and uses a *raw type*.

```
List words = new ArrayList();
words.add("Hello ");
words.add("world!");
String s = ((String)words.get(0))+((String)words.get(1))
```

But at runtime, *they are the same* due to *erasure*.

`List<Integer>`, `List<String>`, and `List<List<String>>` are all represented at run-time by the same type, `List`

Generics vs. Templates

- Generics in Java resemble templates in C++.
- Keep in mind: Syntax and semantics.
 - Syntax is deliberately similar
 - Semantics are deliberately different.

Template Declarations in C++ vs. Generic Declarations in Java

In C++, nested parameters require extra spaces, so you see things like this:

```
List< List<String> >
```

In Java, no spaces are required, and it's fine to write this:

```
List<List<String>>
```

C++ Expansion vs. Java Erasure

- C++ Templates Expansion:
 - Each instance of a template at a new type is compiled separately.
 - e.g If you use a list of integers, a list of strings, and a list of lists of string, there will be three versions of the code. (*code bloat*)
 - Efficient, possible to be optimized
- Java Generics Erasure
 - Erasure doesn't track the generic type at runtime
 - This offers flexibility and less *code bloat* than expansion
 - Maintains safety and ease of use to understand, to a point

Reification

Reification is making something real, bringing something into being, or making something concrete.

Java's Generics are *not reified* at runtime.

Reification Rationale

Generics are implemented using erasure as a response to the design requirement that they support migration compatibility: it should be possible to add generic type parameters to existing classes without breaking source or binary compatibility with existing clients.

Reification Rationale Continued

Without migration compatibility, the collection APIs could not be retrofitted to use generics; we would probably have added a separate, new set of collection APIs that use generics. That was the approach used by C# when generics were introduced, but Java did not take this approach because of the huge amount of pre-existing Java code using collections.

Automatic Boxing of Primitives

```
List<Integer> ints = new ArrayList<>();
ints.add(1); //adding a primitive 1
int n = ints.get(0);
```

This is equivalent to:

```
List<Integer> ints = new ArrayList<>();
ints.add(Integer.valueOf(1));
int n = ints.get(0).intValue();
```

Lab: Discussing Parameterized Classes:

Step 1: In the `src/main/java` folder, and inside the `com.xyzcorp` package.

Step 2: Open `Box.java`

Step 3: Discuss creating `Box` parameterized types.

Lab: Basic Generics Test

Step 1: Create the `src/test/java` directory if necessary.

Step 2: In the `src/test/java` directory, create a package called `com.xyzcorp` if it is not there.

Step 3: Inside the package `com.xyzcorp`, create a java file called `GenericBasicsTest.java` with the following contents:

```
package com.xyzcorp;

public class GenericBasicsTest {

}
```

Lab: Test Using Box

Step 1: In the `GenericsBasicsTest.java` file create a test called `testUsingBox` with the following contents.

```
@Test
public void testUsingBox() {
    Box<Integer> box = new Box<>(4);
    assertEquals(new Integer(4), box.getContents());
}
```

Step 2: Run the test.

Lab: Substituting a Type Parameter with a Parameterized Type

You can also substitute a type parameter (i.e., K, V, E, T) with a parameterized type

Step 1: In the `GenericsBasicsTest.java` file create a test called `testUsingBoxOfBoxInteger` with the following contents.

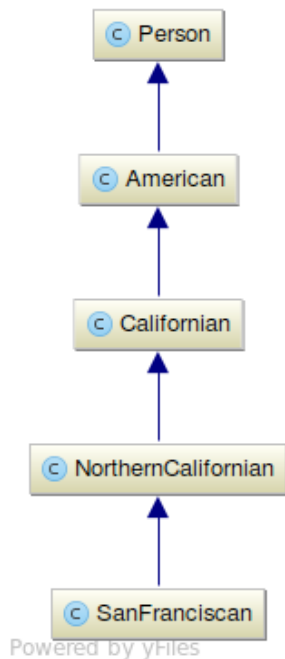
```
@Test
public void testUsingBox() {
    Box<Box<Integer>> box = new Box<>(new Box<>(10));
    assertEquals(new Integer(10), box.getContents().getContents());
}
```

Step 2: Run the test.

Wildcards

Class Diagram of People

For the wildcard generics, we will use the following classes located in `com.xyzcorp.people` package in `src/main/java`



Lab #2: Invariance

Step 1: Create a test called `testInvariance()` test located in the `GenericBasicsTest` with the following:

```
@Test
public void testInvariance() {
    //Call by site
    Box<Californian> boxOfCalifornians = new Box<>();

    //Setters OK
    boxOfCalifornians.setContents(new Californian());

    //Getters OK
    Californian californian = boxOfCalifornians.getContents();

    System.out.println("boxOfCalifornians = " + boxOfCalifornians);
}
```

Step 2: Run the test to ensure that it all works.

By default generics are invariant. Meaning that the given `Box` cannot vary in the types used. The `Box` is *always* going to be a box of `Californians` both on the assignment and instantiation.

Covariance

`S` is a subtype of `T` iff `List<S>` is a subtype of `List<T>`

`Box<? extends Californian>`

`Box<Californian>`

`Box<? extends Californian>`

`Box<SanFranciscan>`

Lab: An Attempt at Covariance

Step 1: In the `GenericBasicsTest` add the following test `testCovarianceAssignments`

Step 2: Add the following content:

```
@Test
public void testCovarianceAssignments() {
    Box<NorthernCalifornian> boxOfNorthernCalifornians = new Box<>();

    //This is an attempt at covariance, this will not work
    Box<Californian> boxOfCalifornians = boxOfNorthernCalifornians;
}
```

Step 3: Describe why this did not work.

Lab: Try Other Variance Assignments

Step 1: In the `testCovarianceAssignments` that we have been using up to this point try the following

assignments, one by one.

```
@Test
public void testCovarianceAssignments() {
    Box<NorthernCalifornian> boxOfNorthernCalifornians = new Box<>();

    //This is an attempt at covariance, this will not work
    Box<? extends Californian> boxOfCalifornians = boxOfNorthernCalifornians;
    Box<? extends Object> boxOfObjects = boxOfNorthernCalifornians;
    Box<? extends Person> boxOfPeople = boxOfNorthernCalifornians;
    Box<? extends American> boxOfAmericans = boxOfNorthernCalifornians;
    Box<? extends NorthernCalifornian> boxOfNorthernCalifornians2 =
boxOfNorthernCalifornians;
    Box<? extends SanFranciscan> boxOfSanFranciscans = boxOfNorthernCalifornians;
}
```

Step 2: Explain why the last one fails, after reviewing, please comment the last line out.

Lab: <? extends Object>

- <? extends Object> is nearly equivalent to <?>

Step 1: In the test that we are working on change `Box<? extends Object> boxOfObjects = boxOfNorthernCalifornians` to `<?>`

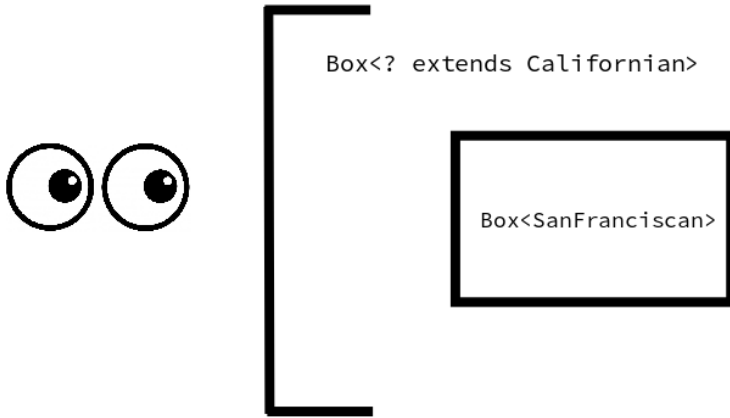
```
@Test
public void testCovarianceAssignments() {
    Box<NorthernCalifornian> boxOfNorthernCalifornians = new Box<>();

    //This is an attempt at covariance, this will not work
    Box<? extends Californian> boxOfCalifornians = boxOfNorthernCalifornians;
    Box<?> boxOfObjects = boxOfNorthernCalifornians;
    Box<? extends Person> boxOfPeople = boxOfNorthernCalifornians;
    Box<? extends American> boxOfAmericans = boxOfNorthernCalifornians;
    Box<? extends NorthernCalifornian> boxOfNorthernCalifornians2 =
boxOfNorthernCalifornians;
    Box<? extends SanFranciscan> boxOfSanFranciscans = boxOfNorthernCalifornians;
}
```

NOTE For an interesting discussion of <?> edge cases see [this StackOverflow article](#).

Get Principle

Use an `extends` wildcard when you only get values out of a structure



Lab: Covariance Get Principle

Step 1: In the `GenericsBasicsTest` create a test called `testCovarianceGetPrinciple`

Step 2: In the test itself add the following lines.

```
@Test
public void testCovarianceGetPrinciple() {
    Box<SanFranciscan> boxOfSanFranciscans = new Box<>();
    Box<? extends Californian> californians = boxOfSanFranciscans;

    Object object = californians.getContents();
}
```

Step 3: Object has been provided for you, add lines to see if you can assign to a `Person`, `American`, `Californian`, `Northern Californian`, and `San Franciscan`

Step 4: Comment out what didn't compile.

Step 5: Explain why certain retrievals didn't work

Lab: The Covariance Put Principle

Step 1: In the `GenericsBasicsTest` create a new test called `testCovariancePutPrinciple`

Step 2: Start with the following content and work your way down to `San Franciscan` from `Object`

```
@Test
public void testCovariancePutPrinciple() {
    Box<SanFranciscan> boxOfSanFranciscans = new Box<>();
    Box<? extends Californian> californians = boxOfSanFranciscans;

    californians.setContents(new Object());
}
```

Step 3: Discuss why it is *not* safe to "set" information from `californians`

Step 4: Try one more line, `californians.setContents(null)` and explain why that one makes sense.

Step 5: Comment out the lines that did not work.

Contravariance

`S` is a supertype of `T` iff `List<S>` is a supertype of `List<T>`

`Box<? super Californian>`

`Box<Californian>`

`Box<? super Californian>`

`Box<Object>`

Lab: An Attempt at Contravariance

Step 1: In the `GenericBasicsTest` add the following test `testContravarianceAssignments`

Step 2: Add the following content:

```

@Test
public void testContravarianceAssignments() {
    Box<Californian> boxOfCalifornians = new Box<>();

    //This is an attempt at contravariance, this will not work
    Box<? super Object> boxOfObjects = boxOfCalifornians;
}

```

Step 3: Describe why this did not work

Lab: Try Other Variance Assignments

Step 1: In the `testContravarianceAssignments` that we have been using up to this point try the following assignments, one by one.

```

@Test
public void testCovarianceAssignments() {
    Box<Californian> boxOfCalifornians = new Box<>();

    //This is an attempt at contravariance, this will not work
    Box<? super Object> boxOfObjects = boxOfCalifornians;
    Box<? super Person> boxOfPeople = boxOfCalifornians;
    Box<? super American> boxOfAmericans = boxOfCalifornians;
    Box<? super NorthernCalifornian> boxOfNorthernCalifornians =
        boxOfCalifornians;
    Box<? super SanFranciscan> boxOfSanFranciscans =
        boxOfNorthernCalifornians;
}

```

Step 2: Explain why the first three failed.

Lab: Contravariance Get Principle

Step 1: In the `GenericsBasicsTest` create a test called `testContravarianceGetPrinciple`

Step 2: In the test itself add the following lines.

```

@Test
public void testContravarianceGetPrinciple() {
    Box<Object> boxOfObjects = new Box<>();
    Box<? super SanFranciscan> boxOfSanFranciscansAndSuperclasses = boxOfObjects;

    Object object = boxOfSanFranciscansAndSuperclasses.getContents();
}

```

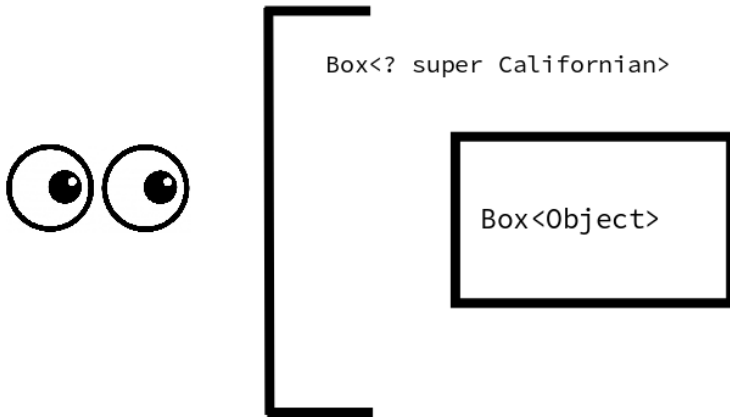

Step 3: Object has been provided for you, add lines to see if you can assign to a `Person`, `American`, `Californian`, `Northern Californian`, and `San Franciscan`

Step 4: Comment out what didn't compile.

Step 5: Explain why most of retrievals minus `Object` didn't work.

Put Principle

Use a `super` wildcard when you only get values out of a structure



Lab: The Contravariance Put Principle

Step 1: In the `GenericsBasicsTest` create a new test called `testContravariancePutPrinciple`

Step 2: Start with the following content and work your way down to `San Franciscan` from `Object`

```
@Test
public void testContravariancePutPrinciple() {
    Box<Object> boxOfCalifornians = new Box<>();
    Box<? super Californian> boxOfSanFranciscansAndSuperclasses = boxOfCalifornians;

    boxOfSanFranciscansAndSuperclasses.setContents(new Object());
}
```

Step 3: Discuss why it is **not** safe to

Step 4: Try one more line, `californians.setContents(null)` and explain why that one makes sense.

Step 5: Comment out the lines that did not work.

Using Wildcards in Methods

Step 1: In `GenericsBasicsTest` create a test method called `testVariancesInMethod()` with the following content:

```
@Test
public void testVariancesInMethod() {
    List<Integer> items = Arrays.asList(5,10, 12, 10, 19, 44);
    assertEquals(Optional.of(5), findFirst(items));
}
```

Step 2: In the same test file, create a non-test method called `findFirst` that would pass the `testVarianceInMethod`

Step 3: Discuss solution

Generic Method in a Generic Class returning a different type

Often times when a class is parameterized, a method can use another parameterized type either to use in conjunction with the types with the class:

```
class A<T> {
    public <U> U foo(T t) {
        //return a type U
    }
}
```

`U` may or not be different than `T` at runtime, but the potential should be present.

This is incorrect, and is referred to as *type hiding*.

```
class A<T> {
    public <T> T foo(T t) {
        //return a type U
    }
}
```

Generic Static Method in a Generic Class returning a different type

Also when a class is parameterized, a `static` method can use another parameterized type either to use in conjunction with the types with the class.

The type system is different from the object graph. There all types established are applicable whether is is `static` or non-`static`

```
class A<T> {
    public static <U> U foo(T t) {
        //return a type U
    }
}
```

U may or not be different than **T** at runtime, but the potential should be present.

This is incorrect, but is not *type hiding*, but is bad and unreadable form.

```
class A<T> {
    public static <T> T foo(T t) {
        //return a type U
    }
}
```

Lab: Creating a generic method in a generic class.

Step 1: In `GenericsBasicsTest` create `testMap` with the following content:

```
@Test
public void testMap() throws Exception {
    Box<Integer> box = new Box<>(4);
    Box<String> newBox = box.map(integer ->
        Stream.generate(() -> "Wow")
            .limit(integer)
            .collect(Collectors.joining()));
    System.out.println("newBox = " + newBox);
}
```

Step 2: In `Box.java` add a method called `map` that takes a `java.util.function.Function` (see Java API)

Step 3: The implementation of `map` should take the function and return a *new* `Box` with the previous state transformed by the function.

WARNING | Nerd Alert

Multiple Bounds

A type parameter can have multiple bounds

```
<T extends B1 & B2 & B3>
```

If one of the bounds is a class, it must be specified first. For example..

```
class A { /* ... */ }  
interface B { /* ... */ }  
interface C { /* ... */ }  
  
class D <T extends A & B & C> { /* ... */ }
```

If not you will receive a compile time exception.

```
class D <T extends B & A & C> { /* ... */ } // compile-time error
```

From [The Java Documentation Online](#)

Lab: Multiple Bounds

Step 1: In `src/test/java` and in the package `com.xyzcorp` create another class, `MultipleBoundsTest.java`

Step 2: Create a test in the `MultipleBoundsTest` called `testMultipleInheritance` and a non-test method `foo` with the following content.

```

package com.xyzcorp;

import org.junit.Test;

import java.io.CharArrayWriter;

public class MultipleBoundsTest {

    public <FILL_HERE> void foo(T t) throws IOException {
        t.append('c');
        t.append('d');
        t.flush();
        t.close();
    }

    @Test
    public void testMultipleInheritance() throws IOException {
        CharArrayWriter writer = new CharArrayWriter(40);
        foo(writer);
        System.out.println(writer.toCharArray());
    }
}

```

Step 3: For method `foo` replace what is in `FILL_HERE` with what you suspect the parameterized type `T` should look like if `T` is also `Appendable`, `Closeable`, and `Flushable` (all are interfaces)

<T extends Comparable<T>>

Why does `<T extends Comparable<T>>` look the way it does?

This should make sense.

```

public class Foo implements Comparable<Foo>{
    private int i = 0;
    @Override
    public int compareTo(Foo o) {
        return Integer.valueOf(i).compareTo(o.i);
    }
}

```

But if it was just `Comparable` it would have to look like this:

```
public class Foo implements Comparable {
    private int i = 0;
    @Override
    public int compareTo(Object o) {
        return Integer.valueOf(i).compareTo(o.i);
    }
}
```

Therefore, <T extends Comparable<T>>

Therefore this code should make sense.

```
public class MyCollection<T extends Comparable<T>> {
    private final T[] items;

    public MyCollection(T... items) { //varargs
        this.items = items;
    }

    public Optional<T> max() {
        if (items.length == 0) return Optional.empty();
        T result = items[0];
        for (T item : items) {
            if (item.compareTo(result) > 0) result = item;
        }
        return Optional.of(result);
    }
}
```

The Problem with <T extends Comparable>

```

public class MyCollection<T extends Comparable> {
    private final T[] items;

    public MyCollection(T... items) { //varargs
        this.items = items;
    }

    public Optional<T> max() {
        if (items.length == 0) return Optional.empty();
        T result = items[0];
        for (T item : items) {
            if (item.compareTo(result) > 0) result = item;
        }
        return Optional.of(result);
    }
}

```

WARNING `item.compareTo(result)` is unchecked because `compareTo` is only expecting `Object` not `T`

Without `<Class<T>>`

We see this everywhere, but what is it? And why does it exist?

Consider:

```

public void listMethodsFromRawClass(Class clazz) {
    clazz.getMethods();
}

```

We can call it with anything.

```
listMethodsFromRawClass(Person.class);
```

With `<Class<T>>`

With `<Class<T>` we can constrain the type of classes that are called.

Consider:

```

public void listMethodsFromRawClass(Class<Person> clazz) {
    clazz.getMethods();
}

```

We can call it with anything.

```
listMethodsFromRawClass(Person.class); //This will not work
```

Review JDK Collections library for Generics

Reminder with Object JDK 7

[java.lang.Objects](#)

Try-With-Resources

- The *try-with-resources* statement is a try statement that declares one or more resources.
- A *resource* is an object that must be closed after the program is finished with it.
- The *try-with-resources* statement ensures that each resource is closed at the end of the statement.
- Any object that implements `java.lang.AutoCloseable`, which includes all objects which implement `java.io.Closeable` are as a resource.

```
static String readFirstLineFromFile(String path) throws IOException {  
    try (BufferedReader br =  
        new BufferedReader(new FileReader(path))) {  
        return br.readLine();  
    }  
}
```

[Oracle Documentation on Try-With-Resources](#)

Java IO (JDK 1.0-1.3)

- Initial Java Implementation consisting of `java.io.File`
- Issues were:
 - Non-performant
 - Blocking
 - File attributes were weak
 - No symbolic links available.

Java NIO (JDK 1.4)

- `Buffer`'s provided a data buffer with varying ways to create a mapping.
- `Channel`'s A representation of a resource, made use of memory mapped files for better

performance.

- **Charset** Character Encodings that map bytes with Unicode

Java NIO.2 (JDK 1.7)

- Greater support libraries for:
 - Files using **Files** static utility for files, directories
 - Paths using **Paths** static utility for paths.
 - Stronger Asynchronous I/O.

Implementing java.nio solutions



Buffer

- Container for a fixed amount of data.
- Acts as a train car where data can be stored and later retrieved.
- Buffers are filled and drained
- Contains the following methods
 - **capacity** - The maximum number of data elements the buffer can hold.
 - **limit** - The first element of the buffer that should not be read or written
 - **position** - The index of the next element to be read or written
 - **mark** - Remembered Position
 - **clear** - Clear out the buffer

The following relationship always holds:

```
0 <= mark <= position <= limit <= capacity
```

Channel

- Conduits or Tunnels to I/O Service
- Accessed with a minimum of overhead
- Buffers are the internal endpoints used by channels to send and receive data.

Lab: Classic NIO style of sending information to a file.

Step 1: In `src/main/java`, create a class in package `com.xyzcorp` called ***FastCopyFile.java***

Step 2: In ***FastCopyFile*** create a main method starting with the following content:

```
if (args.length<2) {  
    System.err.println( "Usage: java FastCopyFile infile outfile" );  
    System.exit( 1 );  
}  
  
String infile = args[0];  
String outfile = args[1];
```

This should be obvious for a setup. We will to copy from the input to the output.

Lab: Continuing Classic NIO style of sending information to file.

Step 1: Continue within the `main` method by adding some of the following information:

```
FileInputStream fin = new FileInputStream( infile ); ①  
FileOutputStream fout = new FileOutputStream( outfile ); ②  
  
FileChannel fcin = fin.getChannel();③  
FileChannel fcout = fout.getChannel();④  
  
ByteBuffer buffer = ByteBuffer.allocateDirect( 1024 ); ⑤
```

① Setup the `FileInputStream`

② Setup the `FileOutputStream`

③ From the `InputStream` we will open the `Channel` to that stream.

④ From the `OutputStream` we will open the `Channel` to that stream.

⑤ Allocate the buffer size of 1024 bytes

Lab: Finishing the copy

Step 1: Continuing with our copy, finish with the following content:

```

while (true) {
    buffer.clear(); ❶

    int r = fcin.read(buffer);

    if (r==-1) { ❷
        break;
    }

    buffer.flip(); ❸

    fcout.write(buffer); ❹
}

```

❶ Clear out the buffer

❷ Get out when we have no more data

❸ `flip` is important. It is the equivalent of `buffer.limit(buffer.position()).position(0)`

❹ Transferring the buffer over into the outward channel

Step 2: Refactor the code to use *try with resources*.

Step 3: Create a file with a list of things you want to eat right now, and call it `food.txt`, save it in whatever directory you would like.

Lab: Running FastFileCopy

Step 1: Go to the command line and invoke the application by either running

```
mvn clean compile && java -cp target/classes com.xyzcorp.FastFileCopy <src> <dest>
```

...or you can use a Maven plugin (no loading necessary).

```
mvn clean compile exec:java -Dexec.mainClass=com.xyzcorp.FastFileCopy
-Dexec.args="<src> <dest>"
```

File I/O using NIO.2

With the advent of NIO.2 with JDK 7. There have been some new utilities that provide NIO solutions to a lot of the mundane tasks previously developed.

Lab: Use the new NIO2 Utilities to perform a FileCopy

Step 1: In `src/main/java`, create a class in package `com.xyzcorp` called *FastCopyFile2.java*

Step 2: In `FastCopyFile` create a `main` method starting with the following content:

```
if (args.length < 2) {
    System.err.println("Usage: java FastCopyFile infile outfile");
    System.exit(1);
}

Path inPath = Paths.get(args[0]);
Path outPath = Paths.get(args[1]);
```

This should be obvious for a setup. We will to copy from the input to the output, but this time with the NIO.2 JDK 7 `Path`.

Lab: Continuing to use the new NIO2 Utilities to perform a FileCopy

Step 1: Continue to add to the `main` method with the following content.

```
CopyOption[] options = new CopyOption[]{
    StandardCopyOption.REPLACE_EXISTING,
    StandardCopyOption.COPY_ATTRIBUTES
};

Files.copy(inPath, outPath, options);
```

Lab: Run FastFileCopy2

Step 1: Go to the command line and invoke the application by either running

```
mvn clean compile && java -cp target/classes com.xyzcorp.FastFileCopy <src> <dest>
```

...or you can use a Maven plugin (no loading necessary).

```
mvn clean compile exec:java -Dexec.mainClass=com.xyzcorp.FastFileCopy
-Dexec.args="<src> <dest>"
```

Lab: Using java.nio with network programming

Step 1: Get the contents from the URL:
`"http://ichart.finance.yahoo.com/table.csv?s=ORCL&a=00&b=01&c=2015"`

Step 2: Using NIO2, NIO, and Streams, filter down all the items, and find the average closing price for the month of July

Thank you