Advanced Java Programming

Will Provost
Edward Rayl

Version 17

JavaAdv. Advanced Java Programming
Version 17

A publication of Capstone Courseware, LLC.
877-227-2477
www.capstonecourseware.com

This book is printed on 100% recycled paper.

Course Overview

Prerequisites

- **This course is intended for Java programmers with some experience.**
- **Any course entitled "Advanced Java" will naturally draw some questions about what sorts of Java topics qualify as "advanced."**
- **This course is designed to follow our "intermediate" Java course, and so we expect the student to be comfortable with the following concepts and skills:**
  - Java as a **procedural language**: data types, flow control, writing algorithms
  - Java as an **object-oriented language**: classes, inheritance, polymorphism, interfaces and abstract classes, exception handling
  - The **functional style** of Java programming, new to Java 8: functional interfaces, lambda expressions, method references
  - Use of the **Collections API**, and therefore some familiarity with Java **generic types**

Labs

- **The course relies on hands-on experience in various topics and techniques.**
- **All lab code is written to build and run according to the standards of the Java Platform, Standard Edition 17.**
- **Lab exercises are deployed as standard Java projects, all under a common root folder.**
  - You may have gotten these files as a ZIP from your instructor.
  - Or you may have downloaded or cloned a Git repository.
- **Throughout the coursebook we'll refer to projects and other paths relative to that root – wherever you've located it on your local system – for example `Station/Step1`.**
  - You can import these projects into your IDE of choice.

Table of Contents

Chapter 1

Generics

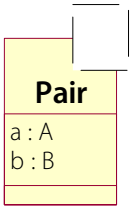
OBJECTIVES

After completing "Generics," you will be able to:

- Explain the advantages of **generics** in the Java language, and use generic types in your code:
  - Define generic classes
  - Establish **boundaries** on type parameters
  - Use **wildcards** to allow flexibility in the use of your generics
- Implement and use **generic methods**.
- Understand the limitations of Java generics, especially the concept of **type erasure**.
- Summarize the strengths and weaknesses of generics for solving various software problem patterns.
- Understand the issues around code compatibility when using generic types.
- Explore design strategies using generics, with a focus on the possible relationships between the generic and its type parameter.

Generic Types in Review

- **Generic types address a certain pattern of design challenges, in which the responsibilities of a class or method are necessarily defined in terms of one or more other types.**
  - A generic is what UML calls a **parameterized type**.
  - The **Collections API** provides a great example of generics, and is how most Java programmers first encounter generics in Java.
  - We speak not just of a "list," but of "a list of strings;" not just a "map," but "a map from integers to instances of **MyClass**."




```
List<String>
Map<Integer,MyClass>
```

- **This allows collection classes to be strongly typed in use, even though as classes they are written (yes) generically.**
  - There are now at least **two dimensions of polymorphism**, though: both the collection type and the item type can vary.
  - We'll see that there are different rules for these two dimensions as well.
- **In this chapter we assume some familiarity with the syntax of using generics – such as when using the Collections API – and we pursue development of custom generic types.**

Declaring Generic Types

- **Define a generic class by place a type parameter in angled brackets immediately following the class name:**

```
public class Things<T>
public class OtherClass<A,B,X>
```

  - There must be at least one type parameter; if more, separate them with commas.
  - The type parameters are understood to be placeholders for other **reference types** – that is, classes or arrays, but not primitives.
  - So we can't have a **Things<int>**. But thanks to **auto-boxing** we can use a **Things<Integer>** and get equivalent functionality.
  - The class code then uses the defined placeholder (such as **T**) in its code, and the compiler treats references to **T** simply as references to some object.
  - Only when the generic class is used in client code is **T** replaced with a "real" type such as **String** or **Car**.

Using Generic Types

- **When a generic is put to a specific use, the compiler can replace the type parameters throughout, and apply its usual strict type checking to everything.**

```
Things<String> stringThings = new Things<String>();
```

  - Here **String** is the **type argument** satisfying the type parameter **T**.
  - Say **Things<T>** defines a method **addThing** that takes a parameter of type **T**; the compiler can assure that it is called safely, and so would allow this:

```
stringThings.addThing ("Hello");
```

  - … but flunk this:

```
stringThings.addThing (new YourClass ());
```

- **It works the same way with return types, but now the advantage is less in avoiding silly mistakes and more in the convenience of not having to downcast a return type.**
  - If **Things<T>** also offers a method **getBestThing** that returns an instance of **T**, there is no need to downcast it to **String** when working with **Things<String>**.

```
String best = stringThings.getBestThing ();
```

  - Nonetheless, **Things<T>** could be used in different code to operate on instances of **Car**, **Ellipsoid**, or any other class.

Implicit Type Arguments

- As of Java 7, the compiler will try to deduce your type arguments when constructing new objects.
- This makes it possible to leave out the arguments in certain common constructs, where they're obvious:

```
List<String> myList = new ArrayList<> ();
Pair<Integer,String> myPair = new Pair<> (5, "Hi");
```

- Note that it is not the constructor arguments that determine the type arguments – though the compiler could probably be taught to use those, too!
- Rather it is the context in which the expression is being evaluated.
  - That is, the compiler is working **"outside-in"** in a way that it did not do in the earliest versions of Java.
  - Another example of outside-in evaluation is the auto-boxing feature we'll discuss later in this chapter.
- This feature will also work in a limited range of other code contexts, such as when passing a parameterized-type argument to a method or returning a new parameterized-type instance.
- But probably over 90% of all "diamond operator" usage is in initializing local or instance variables.

Tools

- **Our primary tools for hands-on exercises in this course are:**
  - The **Java 17** developer's kit
  - An integrated development environment (IDE) of your choice
- **The lab software supports two modes of operation:**
  - Integrated building, deployment, and testing in your IDE
  - **Command-line** builds using prepared scripts
- **We'll provide instructions for each approach separately on the next few pages.**
- **Most students will prefer to use the IDE for hands-on exercises, and where there are differences in practice we will lean in the direction of using the IDE.**

Using a Command Shell

- **To compile and run your code projects from DOS, PowerShell, bash, or similar, you'll just need to be sure that your JDK is set up correctly: a JAVA_HOME environment variable and the bin folder in your executable path.**
  - For Example, for Windows:

```
set JAVA_HOME=c:\Java17
PATH="%JAVA_HOME%\bin";...
```

  - Your JDK location will be specific to your machine: this is often found under **c:\Program Files\Java\...**
- **Note that environment changes in an open console don't affect the operating system as a whole.**
  - This can be a good thing, especially if you have another Java environment set up for work. Anything you do in a DOS or PowerShell window is isolated.
  - But remember that any new consoles will need the same setup – or you can spawn one from another and inherit the environment settings, for instance with **start** in DOS.

Using an IDE

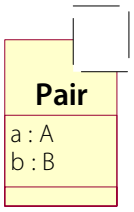
- **You can open any of the Java projects (look for a folder with an src subfolder) in your IDE of choice.**
- **Some IDEs will call this "opening" a project, and some will call it "creating" a project on the existing code.**
- **Most IDEs are set up to build open projects automatically, so the "build" step mentioned in some instructions in the course will not be an explicit action on your part.**
- **Then you can run a given Java class as an application.**
  - The user interface for this varies by IDE.
  - This boils down to invoking the **java** launcher in order to run the **main** method in that class.

A Generic Pair of Objects

EXAMPLE

- A common requirement is to manage related arrays of values, and to iterate over them in parallel.
- The strongest OO approach is to gather the related values in tuples, and then collect an array of those.
- In **Pair/Step1** there is a generic **Pair** class that facilitates this pattern for two related values.
  - See the source file **src/cc/generics/Pair.java**:

```
public class Pair<A,B>
{
  public A a;
  public B b;

  protected Pair () {}

  public Pair (A a, B b)
  {
    this.a = a;
    this.b = b;
  }
  ...
}
```



  - This is just a class that holds references to two others, **A** and **B**.
  - But the types **A** and **B** are not known when compiling this class – in fact there is no implication that actual classes **A** and **B** even exist when **Pair<A,B>** is compiled.
  - These type parameters are only replaced with specific classes (a/k/a type arguments) when other code uses the **Pair** class.

A Generic Pair of Objects

EXAMPLE

- The class **PairOfPairs** exercises this generic type:

```
public static void main (String[] args)
{
  Pair<String,Integer> score1 =
    new Pair<String,Integer> ("Will", 60);
  System.out.println ("score1: " + score1);

  // This would not compile:
  //score1.a = new Integer (60);
  //score1.b = "Will";

  Pair<String,String> sPair =
    new Pair<String,String> ("This", "That");
  sPair.b = "Will";
  System.out.println ("sPair: " + sPair);
  ...
}
```

- **score1** is a pair of string and integer.
  - The compiler will enforce this fact, replacing **A** in the generic definition of the class with **String**, and **B** with **Integer**.
  - Thus the user of the class is protected from the understandable mistake of swapping A and B when reading or writing values – an **Object**-based **Pair** would not do that.
- **sPair** is a pair of strings.
  - Where it's concerned, **b** is a **String**!
  - The two invocations of **Pair<A,B>** are as different to the compiler as, say, **String** and **Integer**, or **Thread** and **Exception**.

A Generic Pair of Objects

EXAMPLE

- **This class doesn't do all that much, but as with most of our exercises you can run it as a Java application, and see some meaningful console output.**
  - In an IDE, just run the class as a Java application.
  - From the command line, there is a **run** script:

```
run     for Windows
  or
./run   for Linux, Mac, UNIX flavors
```

  - The script will accept and pass along command-line arguments to the application, whenever that's useful; for this class it isn't.
  - In the IDE you can modify your run configuration, and set program arguments and/or JVM arguments there.

Nested Use of Generics

- A generic type can be defined in terms of other generic types; in fact the syntax makes this natural:

```
List<Pair<String,Integer>> quantities =
  new ArrayList<> ();
quantities.add (new Pair<> ("A", 5));
System.out.println ("quantities:");
for (Pair<String,Integer> quantity : quantities)
  System.out.format ("%-12s %4d%n",
    quantity.a, quantity.b);
```

- A generic type can use other generics internally, using its own type parameters as type arguments:

```
public class Box<A,B>
{
  public Box (Pair<A,B> row1, Pair<A,B> row2)
  {
    this.row1 = row2;
    this.row2 = row2;
  }

  public Pair<A,B> row1;
  public Pair<A,B> row2;
}
```

- Generics can extend generics:

```
class Box<A,B>
  extends Pair<Pair<A,B>,Pair<A,B>>
{
  public Box (Pair<A,B> row1, Pair<A,B> row2)
  {
    super (row1, row2);
  }
}
```

Type Erasure

- **An important limitation of generics is that they are fully understood only to the compiler.**
- **The compiler practices type erasure, which means it will compile a generic type to an ordinary class file.**
  - The type parameters are **erased**, and the class looks like any ordinary class at runtime.
  - The compiler has performed **strict type checking** against declared type arguments when the generic class is instantiated and used.
  - However, at runtime, the type **G<X>** is **indistinct** from **G<Y>**, and objects of these types are freely inter-convertible.
- **Type erasure is a tradeoff of runtime precision for code simplicity and backward compatibility.**
  - Pre-Java-5 classes and generics can interact in a 5+ runtime, though the compiler will produce some warnings when generic objects are used without properly stating type arguments for them.
- **However it produces some surprising effects:**
  - Type parameters cannot be used in static methods or initializer expressions – because at runtime there is only one class object, rather than several.
  - Legacy code as mixed with generic types can break the type safety that generics establish for Java-5 code.

Type Boundaries

- **It is also possible to define a generic that can only work on subtypes of a certain other class.**
  - A type parameter is declared to **extend** that other class:

```
public class PostalDelivery<T extends USAddress>
```

  - This places an **upper boundary** on type arguments – thus we have a **bounded type parameter**.
  - The compiler will enforce this boundary when resolving instances of the generic type.

```
PostalDelivery<RuralRouteAddress> x; // okay
PostalDelivery<Cucumber> y; // compile error
```

- **The boundary type can be either a class or an interface ... and, strangely, we use the extends keyword regardless.**

```
public class Manager<T extends Runnable>
```

- **Multiple bounds are legal, by a surprising syntax:**

```
public class Math<T extends Number & Mutable>
```

  - Naturally, boundary types after the first one must be interfaces, since only one superclass is possible.
- **This syntax is used on generic class declarations, as above.**
- **It is neither legal nor useful in defining method parameter types or types of local variables.**

Convertibility of Generics

- Earlier we spoke of **two dimensions of polymorphism.**
  - The generic and its type parameters both exhibit polymorphism, but there are some surprising restrictions on the variation under the type parameter.
- **Consider the following (trick) question, for some generic class G and a class D that extends B: is G<D> convertible to G<B>?**
  - Instinct immediately says, "yes, of course it is!"
  - But in fact it is not – not safely.
  - Consider what might happen:

```
List<Derived> dList = new LinkedList<Derived> ();
List<Base> bList = dList; // seems safe
bList.add (new Base ());  // hmm ...
dList.get (0).someDerivedMethod (); // OOPS!
```

  - See the problem?
  - Once the compiler allows the second line to pass, it can't prevent the fourth line from blowing up at runtime.
  - So the compiler makes the second line impossible.
  - It would do the same to an attempt to invoke a method that declared a parameter of type **List<Base>**, while passing an argument of type **List<Derived>**.

Wildcards

- **The conversion of list types would be safe, if we could guarantee that the base-type reference bList would never be used to modify the collection.**
  - And we have designs that call for methods like **Collection.addAll**, which wants to accept a collection of its own type **E** or a collection of some derived type.
  - How can these methods be precisely declared?
  - More generally, we need to be able to use generic types in situations where the type argument is not precisely known.
- **This is the motivation for the wildcard syntax.**

```
public static void printAll (Collection<?> c);
public void addAll (Collection<? extends E> c);
public void convertTo (Ref<? super T> r);
```

- **This syntax strikes a deal with the compiler:**
  - The compiler allows polymorphism over the type argument: that is, **G<D>** can be converted to **G<? extends B>**.
  - In return, the code promises not to call methods with parameters based on the type argument – or to try to assign any fields of the type argument, if they happen to be visible.
- **For collections, this second part is essentially a promise not to modify the collection.**
  - A method such as **addAll** only needs to read elements from the source collection **c**, and there's no difficulty there.

Flipping the Pair

EXAMPLE

- In **Pair/Step2**, the **Pair** class has been enhanced with various methods that "flip" the contents of one pair into the contents of another with opposite type parameters:
- Two of these methods read from the given object – one using the public fields, and one using new **get*X*** methods:

```
public void flipFrom1
  (Pair<? extends B, ? extends A> source)
{
  a = source.b;
  b = source.a;
}

public void flipFrom2
  (Pair<? extends B, ? extends A> source)
{
  a = source.getB ();
  b = source.getA ();
}
```

- For these methods, we use the wildcard syntax to bound our type arguments, because
  - We want to allow reading from pairs of derived types
  - We don't make any changes to the object we're given

Flipping the Pair

EXAMPLE

- Two methods "flip" our values into a given target:

```
public void flipTo1 (Pair<B,A> target)
{
  target.a = b;
  target.b = a;
}

public void flipTo2 (Pair<B,A> target)
{
  target.setA (b);
  target.setB (a);
}
```

- Here, wildcards would not be appropriate.
  - We do make changes to the given object – or, from the compiler's perspective, in **flipTo2**, we call methods that take arguments whose types are any of the type parameters.
  - And – correspondingly – we wouldn't want to accept **target** objects with derived-type parameters, because this could indeed be dangerous. For example ...

```
Pair<String,Base> bPair = new Pair<> ();
Pair<Derived,String> dPair = new Pair<> ();
bPair.flipTo2 (dPair);
dPair.a.derivedMethod (); // WHOOPS
```

- By using plain old type arguments, we actually make it so that the compiler will flunk the third line of code above, rather than waiting for the fourth to blow up.

Flipping the Pair

EXAMPLE

- New code in **PairOfPairs** exercises one of the flip methods, while a prospective call to another is commented out.

```
Pair<String,Number> bPair = new Pair<> ("One", 1);
Pair<Double,String> dPair =
  new Pair<> (5.0, "Five");
bPair.flipFrom2 (dPair);
//bPair.flipTo2 (dPair);
System.out.println ("bPair after flip: " + bPair);
```

- Try a few things – all of which will break compilation.
- Un-comment the line shown above.
  - The compiler flunks this because it sees that the type arguments of **dPair** don't line up perfectly with those defined by the **flipTo2** method.
  - And we don't like it either! because we'd wind up trying to set the **Double** reference in **bPair** to an instance of **Number**.
- Add wildcarding to the signature of either **flipTo1** or **flipTo2**.
  - The compiler complains because the methods break the contract of a wildcard: one trying to set fields on the given object, and the other calling methods on the object that take generic parameters.
  - And we don't like this code because it leads to the same bad outcome – trying to reference a **Number** as a **Double** and eventually expecting something from it that it can't give – even though, now, the caller is being led to believe that this couldn't happen.

Generics in java.util.stream

EXAMPLE

- For another example of generic usage, see **java.util.stream.Stream<T>**: it takes some fluency with type boundaries and wildcards to understand just about any method on this class.
- Just a few examples:

```
Stream<T> filter (Predicate<? super T> predicate);
```

  - This allows the caller to pass an implementation of **Predicate** for type **T** or any base type – which makes sense, because the implementation then can handle objects of type **T** – even if written for a base type – but an implementation intended for types derived from **T** would be dangerous here.
  - Also, the predicate object will not alter the objects it's given; it will only observe them, and so we can use the wildcard for better flexibility.

```
<R> Stream<R> map
  (Function<? super T,? extends R> mapper);
```

  - The only method on **Function<T,R>** accepts a parameter of type **T** and returns an object of type **R**. So, the caller's implementation could safely be written for supertypes of T, but not subtypes; and for subtypes of R, but not supertypes.

```
Optional<T> reduce (BinaryOperator<T> accumulator);
```

  - A **BinaryOperator** both accepts and returns objects of a single type **T**, so it would not be safe to provide an implementation built to work either with supertypes or subtypes.

Generic Methods

- **Methods and constructors can themselves be generic.**
  - Define the type parameters before the method return type or constructor declaration, but after other modifiers:

```
public <T> MyClass (T sourceObject);

public <T> void convertFrom (T sourceObject);

public static <X,Y> Pair<X,Y> flipAny
  (Pair<Y,X> source)
{
  return new Pair<X,Y> (source.b, source.a);
}
```

  - See this last method in **Pair/Step2** as well.

Generic Methods

- **The parameter list of the method must include references to the type parameters, so that the compiler can deduce them when the method is called.**
  - If we were to define this method:

```
public <T> T brandNewThing ();
```

  - … then how would the compiler know what type we want with the following invocation?

```
MyClass myObject = brandNewThing ();
```

  - This is a similar problem to disambiguating overloaded methods, though with different implications.
- **Any type that relies on type T is fine, and if no such parameter would naturally appear in the list, simply use Class<T>.**

```
public <T> T brandNewThing (Class<T> cls);
```

  - Then the caller can (and must, even though it sometimes seems clumsy) supply the type as follows:

```
MyClass myObject = brandNewThing (MyClass.class);
```

Generic Methods on Generic Classes

- **Generic methods can appear on generic or normal classes.**
- **And, if on a generic type, the class' type parameters are accessible as part of the method definition – either to be used directly or as type boundaries:**

```
public <C extends A, D extends B> void flip
  (Pair<C,D> source);
```

  - This is uncommon since a wildcard parameter type will usually do the trick.
  - If type parameters could be lower-bounded (the way wildcards can), that might be useful, as in:

```
public <C super A, D super B> void flipOut
  (Pair<C,D> destination);
```

  - But, alas. Consider **ArrayList<E>.toArray** as another example of this limitation – type **T** below is unrelated to **E**:

```
public <T> T[] toArray (T[] buffer);
```

- **Of course this is moot for static methods, since type erasure puts the type parameter out of scope for statics.**
  - In fact static methods are the more likely candidates for generic methods, as in **Collections.addAll** (not **Collection.addAll**!):

```
public static <T> boolean addAll
  (Collection<? super T> c, T... elements);
```

Forging a List of Pairs

DEMO

- Let's say that we want to add a method to the **Pair** class that would convert two separate lists of disparate types to a single list of **Pair**s of those types – exactly the original motivation we stated for the **Pair** class, after all!
- How would you define that method?
  - It would be **static**, because it would have nothing to do with any specific instance of **Pair<A,B>**.
  - It should take two parameters, right? One would be a **List<A>** and one would be a **List<B>**.
  - It would return a **List<Pair<A,B>>**.
- So, it would do best as a parameterized method, whose return type **List<Pair<A,B>>** would be dictated by the types of the parameters **List<A>** and **List<B>**:

```
public static <A,B> List<Pair<A,B>>
  convert (List<A> aList, List<B> bList) { ... }
```

Forging a List of Pairs

DEMO

- Working in **Pairs/Step2**, let's define this method now ...

1. Stub the method out on **src/cc/generics/Pair.java**:

```
public static <A,B> List<Pair<A,B>>
  convert (List<? extends A> aList,
    List<? extends B> bList)
{
```

2. Let's check the sizes of the two lists, because if they're not the same, we can't really do our job:

```
  if (aList.size () != bList.size ())
    throw new IllegalArgumentException
      ("Uneven lengths!");
```

3. If all is well, create the new list:

```
  List<Pair<A,B>> result = new ArrayList<> ();
```

4. For each pair of elements, add a **Pair** of elements to the new list:

```
  for (int i = 0; i < aList.size (); ++i)
    result.add (new Pair<>
      (aList.get (i), bList.get (i)));
```

5. Return the new list:

```
  return result;
}
```

Forging a List of Pairs

DEMO

6. Now open **PairOfPairs.java** and add some code to the end of the **main** method to exercise the new method:

```
List<String> strings = new ArrayList<> ();
Collections.addAll
  (strings, "one", "two", "three");

List<Integer> integers = new ArrayList<> ();
Collections.addAll (integers, 1, 2, 3);

List<Pair<String,Integer>> translation =
  Pair.convert (strings, integers);
System.out.println
  ("translation: " + translation);
```

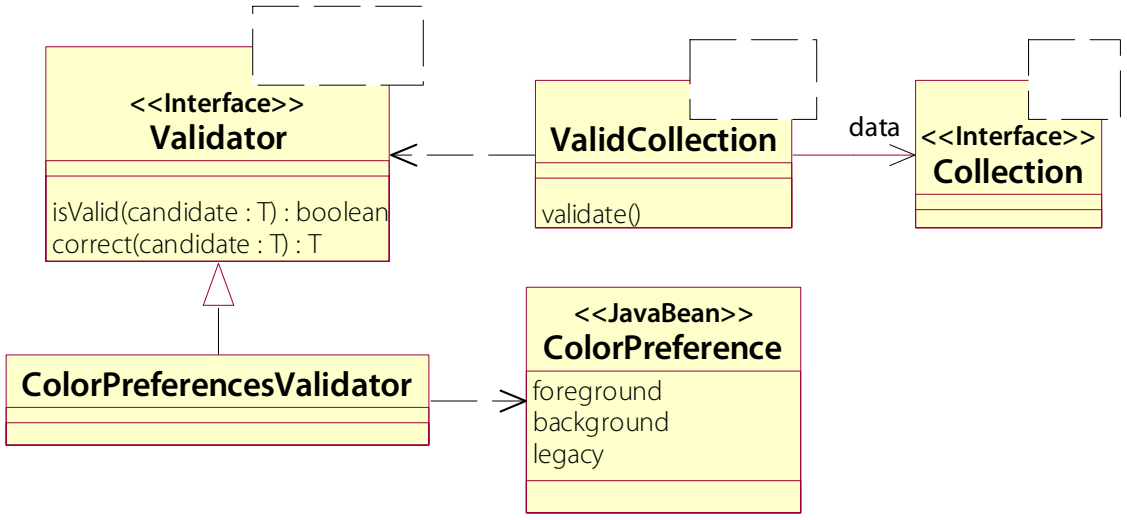
- As with most of the code in this example, we don't do a whole lot with the data that merits console input or output; it's more about seeing we can and can't do with various generic types and type parameters.
- You could print the contents of the new list, if you like, or otherwise experiment with your new generic method.
- But we end the specific instructions here.

A Generic Validator

LAB 1

Suggested time: 45 minutes

In this lab you will build a generic class that performs a simple validation algorithm on a delegate collection. This will allow a validator to be plugged in, independent of the algorithm itself; hence you will need to do some work with wildcards and type boundaries. The final version of the validator will also offer generic convenience methods.



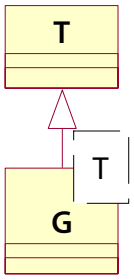
Detailed instructions are found at the end of the chapter.

What You Can't Do

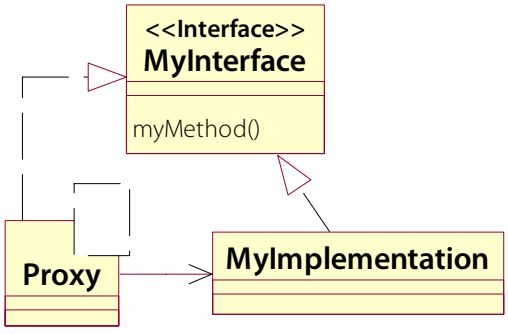
- **Especially for those with experience in other parameterized types in other languages, it is important to understand the basic limitations of Java generics.**
- **As we've discussed, type erasure means you can't distinguish types by type parameter at runtime.**
  - At runtime there is only one class **G**, not **G<X>** vs. **G<Y>**.
- **Another effect is that the Java compiler doesn't generate separate classes prior to compilation.**
  - A C++ compiler, by contrast, does exactly this when it encounters a **class template**.
  - But in Java we can't apply semantics to type **T** willy-nilly, and let the compiler discover that they do in fact apply to the types we instantiate.
  - Specifically, we can't call methods on **T** unless they are available on the upper bound of **T** – the **Object** class or class **B** where **<T extends B>**.
  - By the same logic, one can't instantiate **T** directly using **new**; methods at least are inherited, but constructors aren't.
  - The compiler would need to resolve **new T** to a constructor call, and without knowing the type argument in advance this is impossible.

Weak Suits

- **Especially for C++ experts, generics are tempting, but ultimately ineffective, as solutions to the following problem patterns.**
- **Where the generic type G is a subclass of its type parameter T ("G is a T"), generics fall short.**



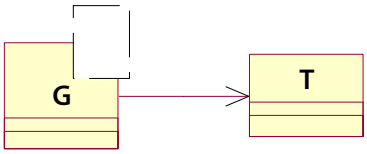
  - This pattern is not possible since the generic would require knowledge of its superclass constructors, and the compiler wants to perform type-checking on **G** before it is instantiated on a specific **T**.
  - This limits the ability, for instance, to create a generic type that adds or "mixes in" a new feature to any existing class.
- **Where G is a generic proxy to T, generics are usable but severely limited in what they can accomplish.**
  - It might be tempting to create a generic proxy (a.k.a. "stub") for all classes (or all classes under a boundary).
  - But the usual purpose of a proxy is to implement an application interface, delegating to the "real" implementation but doing some other generic job along the way.



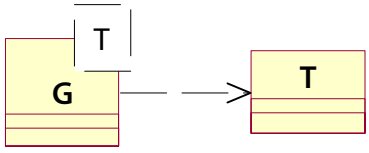
  - Under Java generics this is impossible, because the application interface would have to be known in advance of compiling the generic type – again, that's a no-no.
- **Later we'll discuss dynamic proxies, and see that these, while powerful in their own right, don't use generics.**

Strong Suits

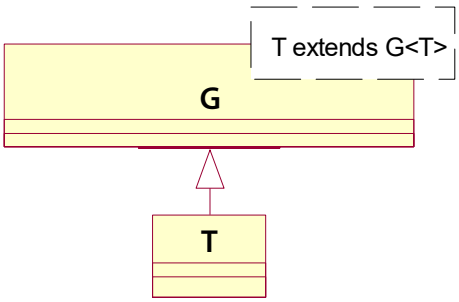
- **So, for what sorts of problems are generics useful?**
  - Where **G** is a **holder of T** ("G has a T") – this is the most common pattern, exemplified by the **Collections API**



  - Where **G delegates to T**, even if it doesn't hold a reference to an object of type **T** ("G uses T") – where **G** offers methods that take parameters based on **T** and can act on **T** instances



  - Where **G** is designed to be a **superclass of T** ("T is a G") – this is less common, but generics can make for very flexible abstract classes, and in fact enumerations are an example of this, via the type **Enum<E extends Enum<E>>**



  - Where **G** is a **factory for T** – this may seem impossible given the constraint on using the construction **new T**, but it is feasible using **reflection**, which we'll cover in a later chapter

Superclass of Type T

- A normal class can extend a generic class or implement a generic interface; we've seen this.
- But what if you want to design a generic that **must** be extended by its type parameter?
- A neat trick that is available takes advantage of the fact that scope of the type parameter includes the class declaration itself, and so we can declare:

```
public class Base<T extends Base<T>>
```

- This allows the generic type to offer useful behavior for inheritance, some of which is "pre-parameterized" in terms of its own subclass.
- The enumerated type applies just this strategy, via **Enum<E extends Enum<E>>**.
  - A partial listing of public methods:

```
public int compareTo (E other);
public Class<E> getDeclaringClass ();
public static <T extends Enum<T>> T valueOf
  (Class<T> enumType, String name);
```

  - **valueOf** is wrapped by a simpler overload in the generated enumerated type **E**, which delegates to this base implementation, providing **E.class** as the first argument.

```
public static E valueOf (String name);
```

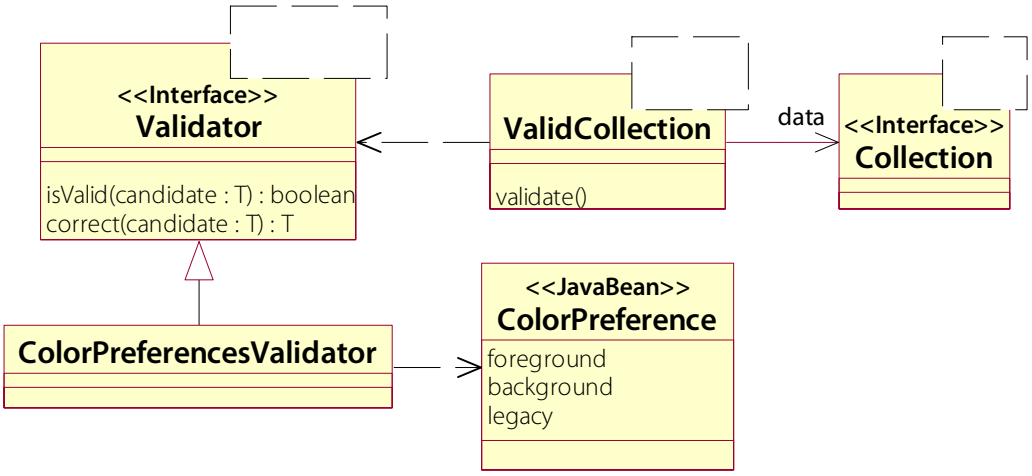
SUMMARY

- **Even in their simplest usage, Java generics offer a few key advantages over plain polymorphism using the Object class:**
  - They expand type safety through a general-purpose type, such as a collection, to cover types that it is intended to use.
  - They make code naturally more self-documenting.
- **Type erasure places significant limitations on how generics can be used, but helps preserve backward compatibility and eases migration to Java 5.**
- **Some advanced techniques, such as defining the generic as an upper boundary on the type parameter, can bring some additional benefits.**
  - These situations are relatively uncommon, and most application programmers will not encounter them.
  - Still, it's good to be opportunistic; in those particular situations, recognizing that generic types offer a solution can save a great deal of unnecessary coding, improve type safety, and improve robustness and maintainability.

A Generic Validator

LAB 1

In this lab you will build a generic class that performs a simple validation algorithm on a delegate collection. This will allow a validator to be plugged in, independent of the algorithm itself; hence you will need to do some work with wildcards and type boundaries. The final version of the validator will also offer generic convenience methods.



Lab project: **Validator/Step1**

Answer project(s):
Validator/Step2 (intermediate)
Validator/Step3 (intermediate)
Validator/Step4 (intermediate)
Validator/Step5 (final)
Validator/Step6 (alternate)
Validator/Step7 (alternate)

Files: * to be created
src/cc/generics/ValidCollection.java
src/cc/generics/Validator.java
src/cc/preference/TestValidation.java
src/cc/inherit/TestValidation.java

A Generic Validator LAB 1

Instructions:

1. Review the starter code, which includes a parameterized **Validator** interface and two packages of code against which this generic system you're building can be applied: **cc.preference** and **cc.inherit**. **Validator** looks like this:

```
public interface Validator<T>
{
    public boolean isValid (T candidate);
    public T correct (T candidate);
}
```

2. Create the source file for the generic class **cc.generics.ValidCollection**. Declare the class, with a single type parameter **T**.
3. Define a private field **data**, whose type is **Collection<T>**.
4. Define a constructor that accepts a **Collection<T>**, and use that parameter to initialize your delegate collection **data**.
5. Define a method **validate** that takes a parameter of type **Validator<T>**, and returns a **boolean**. Iterate over **data**, and pass each element in the collection in a call to **isValid** on the given validator. If any call to **isValid** returns **false**, return **false** from your method.

   You will almost undoubtedly get into some arguments with the compiler at this point! It's a tricky syntax, and often the compiler won't intuit things that seem obvious to the programmer. Consult with your instructor on errors that don't seem to make sense.

A Generic Validator LAB 1

6. The **cc.preference** package includes a single element type **ColorPreferences**, with three color values **foreground**, **background**, and **legend**. The **ColorPreferencesValidator** implements **Validator** specifically for **ColorPreferences** objects:

```
public class ColorPreferencesValidator
    implements Validator<ColorPreferences>
{
    public boolean isValid (ColorPreferences candidate) {...}
    public ColorPreferences correct (ColorPreferences candidate) {...}
}
```

7. The **TestValidation** application in this package builds a list of color preferences, and then wraps it in a **ValidCollection** and **validate**s it. It then does the same to a second list.

```
ValidCollection<ColorPreferences> validList1 =
    new ValidCollection<ColorPreferences> (list1);
System.out.println (validList1.validate (colorValidator)
    ? "List 1 is valid using color preferences validator."
    : "List 1 is invalid using color preferences validator.");
```

8. You will have to un-comment the bulk of this code in order to test your new class. Then run the class, passing "preference" as a command-line argument, and see output as below. (Again: in the IDE, edit your run configuration to set this program argument, and from the command line just **run preferences**.)

```
List 1 is invalid using color preferences validator.
List 2 is valid using color preferences validator.
```

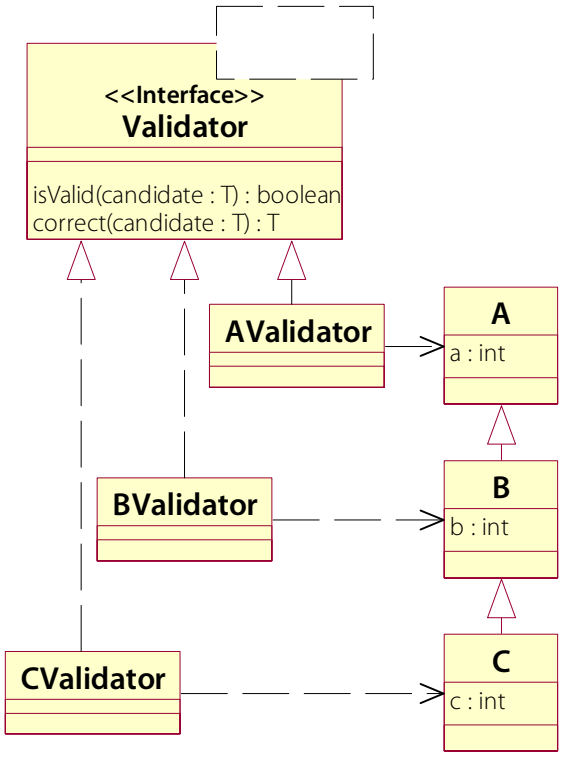
This is the intermediate answer in **Step2**.

A Generic Validator LAB 1

9. Now consider the **cc.inherit** package. Here we have a simple arrangement of classes **A** extended by **B** extended by **C**, a special validator for each, and another **TestValidation** application. This will challenge our existing **ValidCollection**'s usefulness, in two ways:

   - We'd like to accept collections of any type under **T** as delegates – that is, set an upper bound on what object types we'll accept.
   - We'd like to be able to apply validation rules dedicated to types that are supertypes of **T** – thus setting a lower bound based on **T**.

   TestValidation attempts both of these things. First it builds a list of **B** objects and attempts to validate them using the **AValidator**. There's no natural reason that **AValidator** can't work on **B** instances, but our **validate** method doesn't allow it. Then the test application tries to pass a **List<C>** to a **ValidCollection<B>**, and our constructor doesn't allow that either.



10. You can see the second of these two issues immediately, by un-commenting the code in the **main** method: errors appear on all attempts to instantiate **ValidCollection**s of types **B** or **C**.
11. Okay, let's fix this. Change the type of the field **data** to use an upper-bounded wildcard:

```
private Collection<? extends T> data;
```

12. … and the constructor to do likewise:

```
public ValidCollection (Collection<? extends T> data)
```

13. … and **validate** to use a lower-bounded one:

```
public boolean validate (Validator<? super T> validator)
```

    Now you'll see the two errors have cleaned up.

A Generic Validator LAB 1

14. Test and see the following results: set the argument "inherit" in your run configuration, or **run inherit** from the command line.

```
listBC is valid using AValidator.
listBC is invalid using BValidator.

listC is valid using BValidator.

listC is invalid using CValidator.
```

    This is the intermediate answer in **Step3**, which also includes a default validator for any **Object**, and a new overload of **validate** that uses this default.

15. Finally, let's define a generic method **validate** that will simplify the process of using the class. All this method does is wrap the creation and use of a **ValidCollection**, based on a source collection and a validator. So this step is mostly about getting the method declaration syntax correct – use examples earlier in the chapter as guidelines. Make the method static and return a **boolean**.

16. Change **cc.preferences.TestValidator** to skip creating its own **ValidCollection** wrapper object, and instead to use the new method, as in:

```
ValidCollection.validate (list2, colorValidator)
```

17. You should be able to build, run, and get the same results as before.

    This is the final answer in **Step4**.

18. Try this same experiment with **cc.inherit.TestValidation**. This works, too, but consider the implications of an expression like this:

```
ValidCollection.validate (listC, bValidator)
```

19. What is type T in this case? All the compiler knows is **C extends T** and **B super T**. This turns out to be enough! Whether the compiler sees this as **T==B** or **T==C** is not clearly specified, so long as the constraints are met, and within those constraints all the code that is invoked by the method is likewise safe according to declared type boundaries.

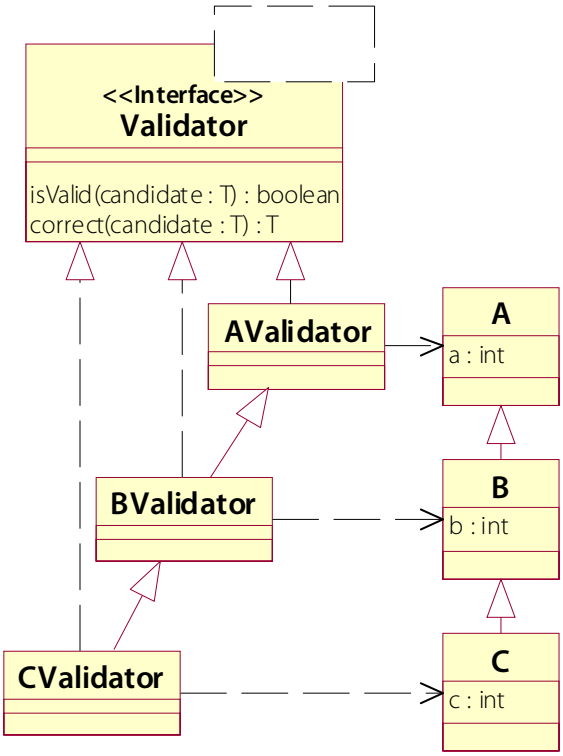
Optional Steps

20. If you like, play around with the **ValidCollection** a bit more by making it **extend** a collection type instead of delegating. This gives some convenience in making **validate** a member of the collection, but it means that **ValidCollection** must commit to a base collection implementation (say, **ArrayList**) – the classic limitation of single inheritance, since we'd have to create a **ValidList**, a **ValidSet**, and so on. Nonetheless this step would give you some additional exercise in working with generics and inheritance, which we haven't addressed elsewhere in the lab.

    The alternate implementation and adjusted test applications are in **Step5**.

A Generic Validator LAB 1

21. Consider other designs for the **cc.inherit** package, especially for the validators. Intuition says that if **B** extends **A**, **BValidator** should probably extend **AValidator** – right? And then call **super.isValid** as part of its own **isValid** implementation? You might give this a try.

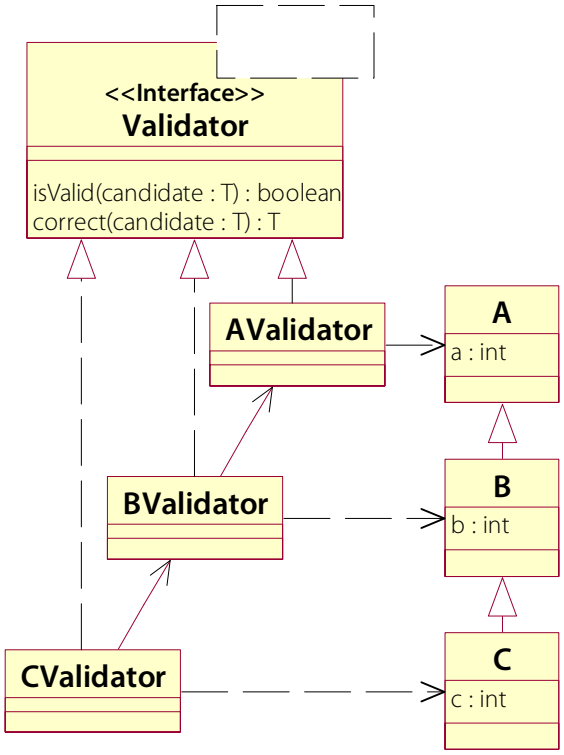


You'll find that you bump up against one limitation of generics: you can either extend **AValidator** or implement **Validator<B>**, but not both. To the compiler, **Validator<A>** and **Validator<B>** are naturally two distinct interfaces, with distinct method signatures; but at runtime of course they are the same type, and a class can't implement the same interface two different ways.

And, if you go this route, you'll find it's not just some type safety that you lose. For example in **TestValidation**, the code that tries to treat **Bvalidator** as a **Validator<B>** is now out of luck – because, again, **Validator<A>** and **Validator<B>** are distinct types, with no convertible relationship, and there is no type-boundary or wildcard usage that will make it possible to cast from one to the other. So this is a bit of a dead end.

A Generic Validator LAB 1

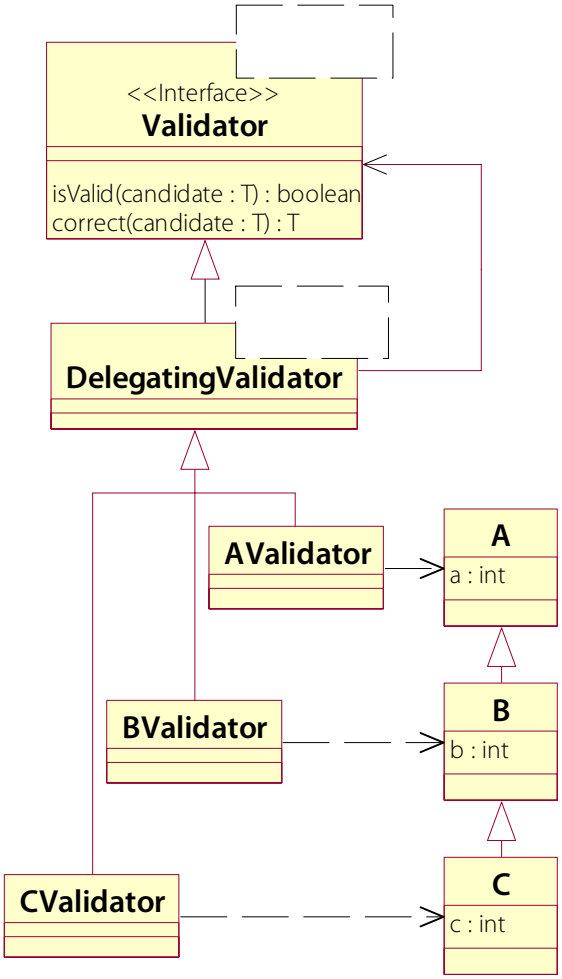
22. What about a delegation-based approach? Can you have **BValidator** delegate to **AValidator**? No problem at all, and similarly for **CValidator**:



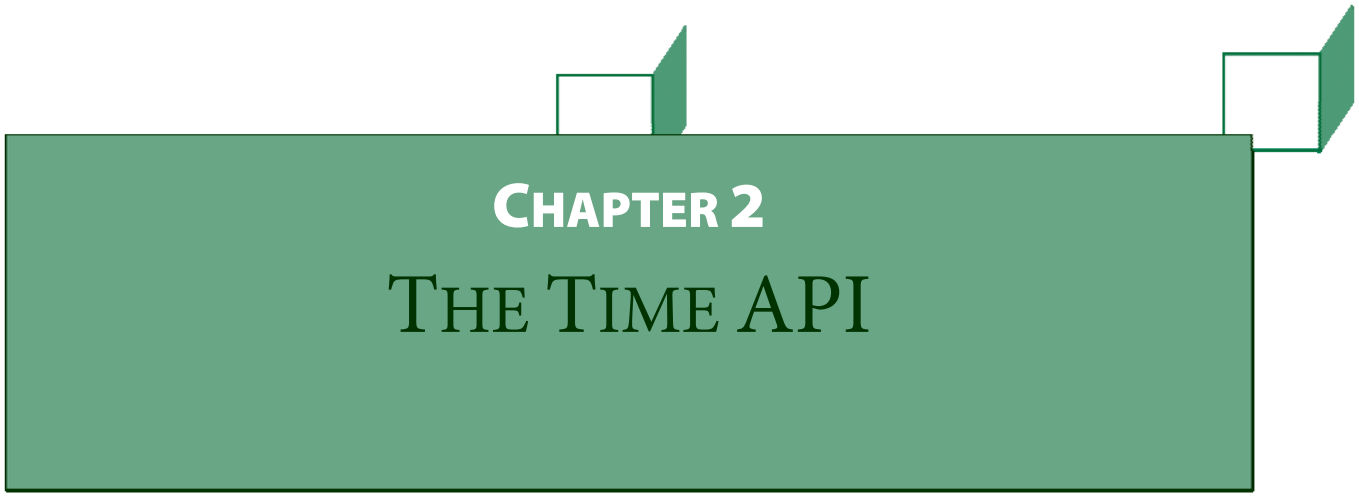
See this alternate implementation in **Step6**.

A Generic Validator LAB 1

23. Could you industrialize this approach, by creating a parameterized, delegating validator type? Yes! and you can even constrain one type to use the other as an upper bound, as in **DelegatingValidator<T extends D, D>**.



You can see this alternate implementation in **Step7**.



Chapter 2

The Time API

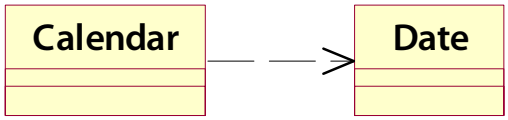
OBJECTIVES

After completing "The Time API," you will be able to:

- Explain key concepts of date-and-time programming, including precision, multiple representations, offsets, and time zones.
- Create and manipulate date and time objects.
- Perform date and time arithmetic.
- Parse and format date and time information.
- Move between local, offset, and zoned times.
- Translate times between time zones.

A History of Time ... in Java

- Ever since Java 1.0 we have had the **java.util.Date** class to represent temporal information.
  - A **Date** encapsulates an instant in time, with precision to the millisecond, and can be converted to and from a **long** representing the number of milliseconds between the instant and a fixed point in the past (the turn of the year 1970).
  - Originally, **Date** also had methods to manipulate and to format its instant as year/month/day and hour:minute:second.msec. But the implementation didn't support internationalization.



- As of Java 1.1, most of the **Date** API was deprecated in favor of the new **Calendar** class, which does a better job of supporting internationalized management and formatting.
  - **Calendar** can work as a factory for **Date** objects.
  - Or, less commonly, its instances can be used directly.
- More sophisticated formatting is now provided by the **DateFormat** and **SimpleDateFormat** classes.
- Still the API has its limitations, as discussed in a moment.
- Open-source alternatives have flourished – most notably the **Joda Time** API.

Limitations of Date and Calendar

- **One of the harder things about working with Date is that it always "thinks" that it represents a millisecond-precision instant.**
- **Often the actual problem at hand concerns only certain parts of that whole.**
  - You may care about year, month, and day, but not time of day.
  - You may want to express time of day, for any date or for something that happens daily.
- **In these cases Date can be hard to handle: especially, it won't give sensible results for things like equals and compareTo operations.**
- **As to Calendar, it's API is rigorous, but a bit clumsy by modern standards.**
- **It is also, arguably, overly concerned with internationalization.**
  - **Time zones** are baked into **Calendar** instances, but not into **Date**s, which always are understood to represent UTC – or, often misunderstood by the developer to represent local time!
  - It's common to get a **Date** from certain **Calendar** operations that's in **UTC** when you were expecting local time.
  - Then you need to **format** representations of **Date** for the desired time zone.

Dates and Times – The

EXAMPLE

- In **DateTime/Step1** we show some typical applications of **Date** and **Calendar** – see **src/cc/datetime/DateTime.java**.
  - The **main** method calls a static method **useJavaUtilDate**, which starts by getting a **Date** that captures this current system time:

```
public static void useJavaUtilDate()
{
  Date now = new Date ();
  System.out.println ("  Now: " + now);
```

  - Then, to get a specific calendar date, it creates a **Calendar**, and sets it to a specific year, month, and day:

```
Calendar calendar = Calendar.getInstance ();
calendar.set (2015, 2, 7);
System.out.println
  ("  Specific date: " + calendar.getTime ());
```

  - It does this a second time, in the same way ...

```
Calendar calendar2 = Calendar.getInstance ();
calendar.set (2015, 2, 7);
System.out.println
  ("  Specific date: " + calendar.getTime ());
```

  - ... and compares the two:

```
System.out.println ("  Same date? " +
  calendar.equals (calendar2));
```

Dates and Times – The EXAMPLE

- **If you run the application, you'll see why we go through this apparently needless exercise. A couple of surprises:**

```
Using java.util.Date and java.util.Calendar:
  Now: Fri Oct 24 23:07:05 EDT 2014
  Specific date: Sat Mar 07 23:07:05 EST 2015
  Specific date: Sat Mar 07 23:07:05 EST 2015
  Same date? false
  ...
```

  - For one thing, the specific date is March 7, 2015 – because **Calendar**'s month values are zero-based.
  - And, the two apparently equivalent dates are not in fact **equal**. Why is this?
  - **Date** always represents a millisecond-precision instant.
  - So we really compared two instants, each of which had been set to have a certain year, month, and day, but which were fundamentally unequal when they were created – off by a few milliseconds.
  - This sort of **precision** issue bites Java developers all the time.

Dates and Times – The EXAMPLE

- The next passage of code does the same job, but more robustly:

```
calendar.setTimeInMillis (0);
calendar2.setTimeInMillis (0);
calendar.set (2015, 2, 7);
calendar2.set (2015, 2, 7);
System.out.println ("  Same date? " +
  calendar.equals (calendar2));
```

- Now we try a little basic date arithmetic – advance one week, and then days as necessary to find a Monday:

```
calendar.add (Calendar.WEEK_OF_YEAR, 1);
while (calendar.get (Calendar.DAY_OF_WEEK)
    != Calendar.MONDAY)
  calendar.add (Calendar.DAY_OF_YEAR, 1);
System.out.println
  ("  Future date: " + calendar.getTime ());
```

  - We test to see if the same day of the following month is also a Monday:

```
calendar.add (Calendar.MONTH, 1);
System.out.println ("  Next month a Monday? " +
  (calendar.get (Calendar.DAY_OF_WEEK) ==
    Calendar.MONDAY));
```

Dates and Times – The EXAMPLE

– Finally we set the current server time on that day:

```
  Calendar nowAsCalendar =
    Calendar.getInstance ();
  calendar.set (Calendar.HOUR_OF_DAY,
    nowAsCalendar.get (Calendar.HOUR_OF_DAY));
  calendar.set (Calendar.MINUTE,
    nowAsCalendar.get(Calendar.MINUTE));
  calendar.set (Calendar.SECOND, 0);
  calendar.set (Calendar.MILLISECOND, 0);
  System.out.println ("  This time on that day: "
    + calendar.getTime ());
}
```

- Here is the rest of the application output:

```
...
Same date? true
Future date: Mon Mar 16 19:00:00 EDT 2015
Next month a Monday? false
This time on that day:
  Thu Apr 16 23:07:00 EDT 2015
```

- Note that the output format is just the default **toString** representation of **Date**; there is a more sophisticated formatting system, but we're not using it in this example.

The Time API

- **Java 8 introduces a new API for managing temporal data, in packages java.time and java.time.temporal.**
- **The Time API decomposes concepts of temporal data more finely, and develops more natural encapsulations for common use.**
  - Especially, the notions of **time zones** and **offsets** are isolated in a few types, and developers don't need to deal with that added complexity unless it really is part of the problem at hand.
  - Most work focuses instead on types that capture **local time**.
  - **Date** and **time** are neatly separated, so that you don't have to manage the one if you're only interested in the other.
  - In fact there is a more general encapsulation of **precision**, such that you can express and manage whatever part of time information matters: **month-and-day**? **year**? **instant**?
- **The Time API is also quite easy to use, thanks to**
  - A **fluent-API** approach
  - An abundance of **helper methods**
  - A focus on **naming conventions** and **patterns**

Time Representations

- **The Time API allows for multiple, overlapping representations of temporal information, defining a temporal field as representing data at some precision and within some enclosing scope.**
- **Consider a specific instant on the Gregorian calendar – which in U.S. English we'd describe as "November 25, 1959, at 3:03 PM and 40 seconds," probably not mentioning the milliseconds:**

```
1959-11-25  PM 03:03:40.234
```

- **We might care about parts of this expression:**
  - Only the date – so at a **precision** of days
  - Only the time – so at second or millisecond precision, but within the **scope** of days.
  - Only a count of milliseconds – so not bounded by the nearest second, minute, hour, day, etc., but in total from some **reference point**.
- **We might also represent some of the information differently:**
  - Using a **24-hour clock** instead of a 12-hour one:

```
1959-11-25     15:03:40.234
```

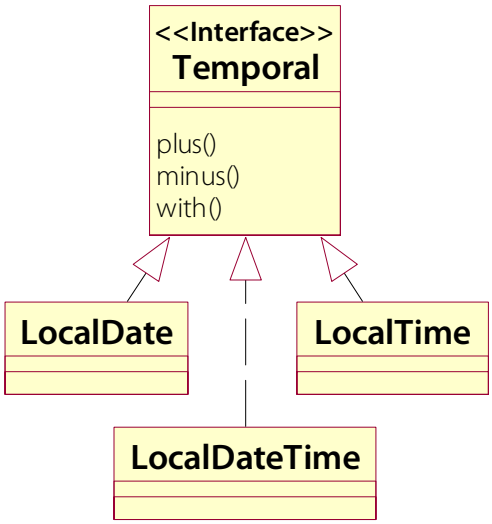
  - Expressing interest in the **day of the week** or **day of the year**:

```
1959-11-25 (a Wednesday) (329th day of 1959)
```

- **The Time API allows you to work with whatever precision, and at whatever scope, you like, and makes it easy to extract one field from another, or to absorb one into another.**

Temporal Types

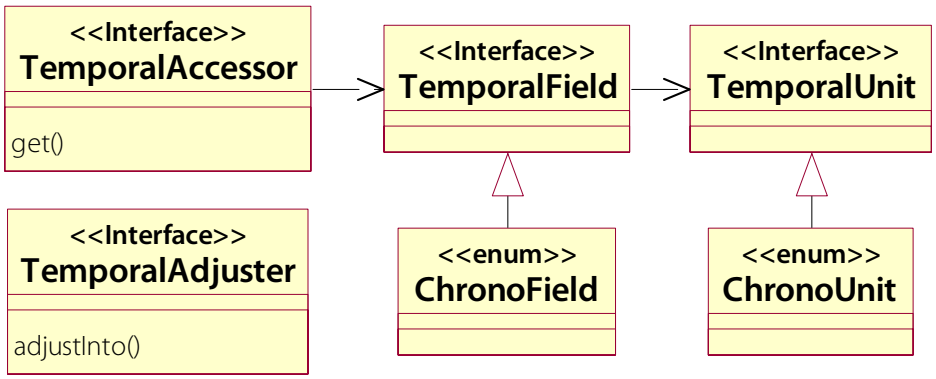
- **A set of classes that capture local time are some of the most commonly used:**
  - **LocalDate** holds information down to day precision.
  - **LocalTime** holds information up to, but excluding, days.
  - **LocalDateTime** expresses everything, and can easily be composed from or decomposed into the other two.



- **All of the types in the UML diagram in this chapter are rich with more methods than it's worth trying to show, so here we focus on type relationships, and sometimes a few methods to indicate the key responsibilities of a given type.**
- **We will break down methods in groups on some later pages.**
- **All are implementations of the type Temporal – which is to say:**
  - They measure time at some precision – from fairly coarse, such as years, to very fine, such as milliseconds.
  - They are "**complete**" in their representations to support arithmetic operations **plus** and **minus**; that is, no extra information is needed to inform an accurate operation.
- **We'll see a handful of other Temporal implementations later in the chapter.**

Temporal Accessors and Adjusters

- All of the **LocalDate/Time** classes also implement interfaces **TemporalAccessor** and **TemporalAdjuster**:



- **TemporalAccessor** encapsulates the semantics of reading temporal information in **fields** or **units**.
  - A **TemporalField** encapsulates some **scope and precision** of temporal information.
  - Fields offer methods to operate on **Temporal**s, for instance by **add**ing some number of units at that precision.
  - The units themselves are expressed as **TemporalUnit**s.
  - **ChronoField** and **ChronoUnit** enumerate legal values.
  - So you can **get** the **DAY_OF_WEEK** from a **LocalDate**.
- **TemporalAdjuster** allows you to **adjust** one temporal based on the value of another:
  - You could adjust an existing **LocalDateTime** to have a new **LocalTime** while preserving its date.
  - You could adjust a **LocalDate** to the **TemporalAdjusters.firstDayOfMonth**.

Instantiation

- Create new instances by a variety of factory methods.
- **now** returns a new object set to the system date

```
LocalDate now = LocalDate.now ();
LocalTime alsoNow = LocalTime.now ();
```

- **of** returns an object set with the given year, month, and day:

```
LocalDate then = LocalDate.of (1959, 11, 25);
LocalTime thisMorning = LocalTime.of (9, 0);
LocalTime precise = LocalTime.of (9, 10, 11, 12);
```

  - Note that **month** values are **1-based**, or you can use the **Month** type described later in the chapter.
  - **LocalTime** supports **nanoseconds**, and its important to be aware of this in some method signatures, if as most of us are you are used to working with milliseconds. That's 12 nanoseconds, above.
  - Create a **LocalDateTime** by combining date and time objects:

```
LocalDateTime complete =
  LocalDateTime.of (theDate, theTime);
```

- **from** converts from another **TemporalAccessor**.

```
LocalDate justTheDate = LocalDate.from (dateTime);
```

  - The parameter must have **sufficient information** to support an accurate conversion – which means there are very few options, and most types, even if they compile, will fail at runtime.
- You can also create new objects by **parsing**, as we'll see on the next page.

Formatting

- Convert temporal objects to and from string representations using the **java.time.format.DateTimeFormatter**.
  - The simplest way to get a formatter instance is by providing a pattern to the factory method **ofPattern**:

```
DateTimeFormatter dateFormat =
  DateTimeFormatter.ofPattern ("M/d/yy");
DateTimeFormatter timeFormat =
  DateTimeFormatter.ofPattern ("hh:mm");
```

- Then, format instances of **TemporalAccessor**.
  - Call **format** and pass the source object:

```
dateFormat.format (LocalDate.now ());
```

  - Or, call **format** on the object and pass the formatter:

```
LocalDate.now ().format (dateFormat);
```

  - Call **formatTo** to append to existing information:

```
timeFormat.formatTo (LocalTime.now (), openWriter);
```

- **parse** string representations into temporal accessors:
  - Again, use the formatter and pass the representation, or call **parse** as a factory method on the desired temporal:

```
LocalDate then = dateFormat.parse ("11/25/59");
LocalTime thisAfternoon =
  LocalTime.parse ("15:00", timeFormat);
```

- See the API docs for **DateTimeFormatter** for details on the formatting patterns and their usage.

Decomposition

- **get** the value of a particular temporal field from a temporal accessor:

```
LocalDate now = LocalDate.now ();
LocalTime alsoNow = LocalTime.now ();
int year = now.get (ChronoField.YEAR);

int month = now.get (ChronoField.MONTH_OF_YEAR);
int hour = alsoNow.get (ChronoField.HOUR_OF_DAY);
```

- All **TemporalAccessor**s have the **get** method.
- Then, specific types add strongly-typed methods for the fields they support:

```
int year = now.getYear ();
int month = now.getMonthOfYear ();
DayOfWeek dayOfWeek = now.getDayOfWeek ();

int minute = alsoNow.getMinute ();
int nano = alsoNow.getNano ();
```

- Note that this is not the right approach for getting a **LocalDate** or **LocalTime** from a **LocalDateTime**.
  - None of these types is a **TemporalField**.
  - Use the special methods **toLocalDate** and **toLocalTime**.

```
LocalDate dateOnly = complete.toLocalDate ();
LocalTime timeOnly = complete.toLocalTime ();
```

Creating and Printing Dates

DEMO

- We'll implement similar functionality to what we've seen with **Date** and **Calendar** classes, but using **LocalDate** and **LocalTime**.
- This demo proceeds in segments: we'll start now with what we've learned about creating date and time objects and formatting them; then learn a few more tricks, and resume the demonstration.
  - Do your work in **DateTime/Step1**.
  - The completed demo is in **DateTime/Step2**.

1. Open **src/cc/datetime/DateTime.java** and create a new method **useLocalDateTime** that prints the full system time:

```
public static void useLocalDateTime ()
{
  System.out.println
    ("  Now: " + LocalDateTime.now ());
}
```

2. Now get an object that represents a specific date:

```
  LocalDate date = LocalDate.of (2015, 2, 7);
```

3. At the top of the method, create a formatter:

```
public static void useLocalDateTime ()
{
  final DateTimeFormatter shortDate =
    DateTimeFormatter.ofLocalizedDate
      (FormatStyle.SHORT);
  System.out.println
    ("  Now: " + LocalDateTime.now ());
  ...
```

Creating and Printing Dates

DEMO

4. Back at the bottom, use it to format your date:

```
    LocalDate date = LocalDate.of (2015, 2, 7);
    System.out.println ("  Specific date: " +
      date.format (shortDate));
}
```

5. Let's do the same test we did with **java.util.Date**: create a second representation of that specific date, print it, and then compare the two by calling **equals**.

```
    LocalDate date2 = LocalDate.of (2015, 2, 7);
    System.out.println ("  Specific date: " +
      date2.format (shortDate));

    System.out.println
      ("  Same date? " + (date.equals (date2)));
```

6. At this point let's invoke the method to see what we get. At the bottom of **main**, add code similar to what's already there to invoke your new method:

```
public static void main (String[] args)
{
  System.out.println ("Using java.util.Date...");
  useJavaUtilDate ();
  System.out.println ();

  System.out.println
    ("Using java.time.LocalDate:");
  useLocalDateTime ();
  System.out.println ();
}
```

Creating and Printing Dates

DEMO

7. Run the application now, and see three differences between the behavior of **Date/Calendar** and **LocalDate**:

```
Using java.util.Date and java.util.Calendar:
  Now: Sat Oct 25 14:17:35 EDT 2014
  Specific date: Sat Mar 07 14:17:35 EST 2015
  Specific date: Sat Mar 07 14:17:35 EST 2015
  Same date? false
  Same date? true
  Future date: Mon Mar 16 19:00:00 EDT 2015
  Next month a Monday? false
  This time on that day:
    Thu Apr 16 14:17:00 EDT 2015

Using java.time.LocalDate:
  Now: 2014-10-25T14:17:35.439
  Specific date: 2/7/15
  Specific date: 2/7/15
  Same date? true
```

   - The default **toString** representation of **LocalDate** is more terse, and runs rigorously from **general to specific**, in a format that also fits nicely with **XML Schema** and various styles of web services.
   - Months are 1-based, so the date is now **February 7**, not March 7 – which is a good bit more intuitive, right?
   - **LocalDate** suffers no confusion about stray time information, and so the date is the date is the date – **equals** returns **true**.

Adjustment

- You can modify an existing temporal to accommodate whatever is expressed in a separate **TemporalAdjuster**.
  - **TemporalAdjuster.adjustInto** will "push" the adjuster's values into your **Temporal**.
  - But it's usually clearer and easier to work from the other side, and the **with** method exists for this purpose:

```
LocalTime noon = LocalTime.of (12, 0);
LocalDate noonToday = LocalDate.now ().with (noon);
```

  - You can also fix a specific field to a specific value, with a different overload of **with** and with strongly-typed variants:

```
LocalTime twelveThirty = noon.withMinutes (30);
```

- **TemporalAdjuster** is more general than this, too: the **adjustInto** method can do just about anything to the existing **Temporal**.
- See **TemporalAdjusters** for a number of prepared adjusters that can save you some time and trouble – these are factory methods:

```
lastDayOfMonth ()
firstDayOfNextMonth ()
next (DayOfWeek day)
nextOrSame (DayOfWeek day)
dayOfWeekInMonth (int weekNumber, DayOfWeek day)
```

Date Arithmetic

- All **Temporal** types offer weakly-typed methods to add or subtract by **ChronoUnit**:

```
LocalDate nextDay =
  someDay.plus (1, ChronoUnit.DAYS);
LocalTime fiveHoursAgo =
  LocalTime.now ().minus (5, ChronoUnit.HOURS);
```

- Then subtypes offer strongly-typed alternatives:

```
LocalDate nextDay = someDay.plusDays (1);
LocalTime fiveHoursAgo =
  LocalTime.now ().minusHours (5);
```

- Note that, while all arithmetic operations will return consistent results (dates consider lengths of months, leap years, etc.), there is no accounting for fields not expressed in the type you use.
  - For example if the hours **"roll over"** midnight when adding or subtracting on a **LocalTime**, it's up to you to capture the change in date if it matters to you.
- You can also **truncate** a time to a desired precision:

```
LocalTime justHours =
  myTime.truncatedTo (ChronoUnit.HOURS);
```

  - There is no analogous operation on **LocalDate**, because it's not really meaningful to "zero out" fields such as month or day.

ChronoField and ChronoUnit

- The two enumerations **ChronoField** and **ChronoUnit** are easy to confuse – as are the interfaces they implement, **TemporalField** and **TemporalUnit**. Keep this in mind:
  - **Fields** are parts of temporal objects, as they exist, and allow us to represent or to extract some meaningful **part of a whole**.
  - **Units** allow us to relate temporal objects arithmetically, and they can express the **difference** between two temporals.
  - All fields are measured in units, but there are **multiple fields with the same unit** that represent the unit within a scope.
- Here are some values for the two, aligned so as to highlight the conceptual difference between the types:

ChronoField	ChronoUnit
MONTH_OF_YEAR	MONTHS
DAY_OF_YEAR	DAYS
DAY_OF_MONTH	
DAY_OF_WEEK	
EPOCH_DAY	
AMPM_OF_DAY	HALF_DAYS
HOUR_OF_AMPM	HOURS
HOUR_OF_DAY	

- Note that there are also a few handy **ChronoUnits** for which there is no corresponding **ChronoField**:

```
DECADES
CENTURIES
MILLENIA
```

Date Arithmetic

DEMO

8. Continuing at the bottom of **useLocalDateTime**, now we'll do some date arithmetic. First, add two new formatters to the top of the method:

```
public static void useLocalDateTime ()
{
  final DateTimeFormatter shortDate =
    DateTimeFormatter.ofLocalizedDate
      (FormatStyle.SHORT);
  final DateTimeFormatter myDate =
    DateTimeFormatter.ofPattern
      ("MMMM dd, yyyy");
  final DateTimeFormatter longDateTime =
    DateTimeFormatter.ofPattern
      ("yyyy-MM-dd hh:mm:ss");
```

9. Now, at the bottom, find the date that is one week later than February 7, and then move forward until you find a Monday:

```
  System.out.println ("  Same date? " +
    (date.equals (date2)));

  date = date.plusWeeks (1)
    .with (TemporalAdjusters.nextOrSame
      (DayOfWeek.MONDAY));
  System.out.println ("  Future date: " +
    date.format (myDate));
```

10. Check to see if that date, one month later, is also a Monday:

```
  date = date.plusMonths (1);
  System.out.println ("  Next month a Monday? " +
    (date.getDayOfWeek () == DayOfWeek.MONDAY));
```

Date Arithmetic

DEMO

11.Now, get the system time, in hours and minutes only:

```
LocalTime time = LocalTime.now ()
  .truncatedTo (ChronoUnit.MINUTES);
```

12.Synthesize a date-and-time from our computed date and this time:

```
LocalDateTime dateTime =
  LocalDateTime.of (date, time);
```

13.Produce that date-time, in long form:

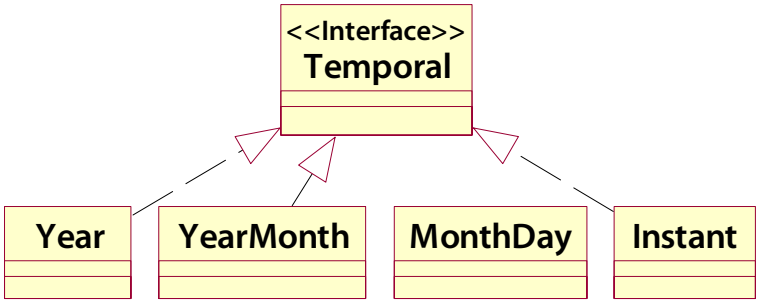
```
System.out.println ("  This time on that day: "
  + dateTime.format (longDateTime));
```

14.Run the application and see appropriate results:

```
Using java.time.LocalDate:
  Now: 2014-10-25T14:17:35.439
  Specific date: 2/7/15
  Specific date: 2/7/15
  Same date? true
  Future date: February 16, 2015
  Next month a Monday? true
  This time on that day: 2015-03-16 02:17:00
```

Other Temporal Representations

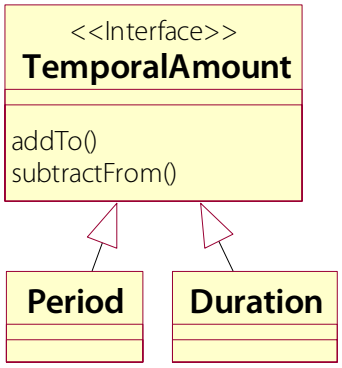
- **For special applications, you may want to take advantage of other Time API types:**



- **Most of these cover subsets of the LocalDate fields.**
  - **Year** represents only the year, and no other date information.
  - **YearMonth** and **MonthDay** focus on two fields each.
  - Note that **MonthDay** is not a **Temporal**: it fails the test of being "complete enough to be manipulated using plus and minus." What if you added a day to February 28? The answer depends on the year, which is not captured by the object.
  - It is, however, a **TemporalAccessor** and a **TemporalAdjuster**.
- **Instant represents an instant, to nanosecond precision.**
  - Unlike local temporals, it assumes a **time zone**, which is UTC.
  - It's most useful for creating and comparing precise **timestamps**.
  - It is the closest equivalent to **java.util.Date** – in that it works in **Epoch time**, measuring from midnight UTC on January 1, 1970.

Duration and Period

- A few types exist just to capture lengths of time:



  - **Period** relates to **date**, while **Duration** relates to **time**.
- You can use instances of **TemporalAmount** for date arithmetic – for example capturing a required gap between two dates.
  - Build durations and periods with **ofXXX** factory methods:

```
Period threeDays = Period.ofDays (3);
Period aYearAMonthAndADay = Period.of (1, 1, 1);
Duration minusTwoHours =
  Duration.ofHours (2).negated ();
Duration twoDaysInTime = Duration.ofDays (2);
```

  - Get the duration **between** two **Temporals**:

```
Period season = Period.between (summer, autumn);
```

  - Do arithmetic with temporal amounts:

```
date = threeDays.addTo (date);
date = date.minus (threeDays);
```

- Note that **ChronoUnit** also offers a **between** method, allowing you to measure gaps in specific units as you like.

Browsing a Datebook

DEMO

- We'll add code to complete a datebook application that can read data from a stream of strings and then let the user browse over events, grouping them by day.
  - Do your work in **Datebook/Step1**.
  - The completed demo is in **Datebook/Step2**.
- In **src/cc/datetime/Persistence.java** there is a placeholder implementation of persistence that "loads" the datebook and returns a list of strings:

```
public static List<String> loadDatebook()
{
  List<String> result = new ArrayList<>();

  result.add
    ("2015-02-23T09:00:00 Discuss pre-meeting");
  result.add("2015-02-23T09:30:00 Pre-meeting");
  ...
```

- The data type in **src/cc/datetime/Event.java** has two properties:
  - **when** is a **LocalDateTime**.
  - **what** is a **String**.
- In **src/cc/datetime/Datebook.java**, the datebook class is a **TreeMap<LocalDate, Event>**, and part of our work is going to be to implement the grouping of a flat list of events into a map with the date parts of the events' **when** properties as keys.

Browsing a Datebook

DEMO

1. In **Datebook.java**, create a static method **parse** to convert a string into an **Event**. Start by splitting the string over the first space character:

```
public static Event parse(String serialized)
{
  int separator = serialized.indexOf (' ');
```

2. Create a new event object, and set its **when** property by parsing the first part of the string:

```
  Event result = new Event ();
  result.setWhen (LocalDateTime.parse
    (serialized.substring (0, separator)));
```

3. Set the **what** parameter to the rest of the string:

```
  result.setWhat
    (serialized.substring (separator + 1));

  return result;
}
```

Browsing a Datebook

DEMO

4. Now, the class already offers a method to **buildSchedule**, but so far it just returns **null**. Implement it to build the map by processing the given **Stream<String>**, first by parsing to **Event** and then by grouping on the date part of the **when** property:

```
public void static Datebook buildSchedule
  (Stream<String> serializedEvents)
{
  Datebook datebook = new Datebook ();
  datebook.putAll
    (serializedEvents.map (Datebook::parse)
      .collect (Collectors.groupingBy
        ((ev) -> ev.getWhen ().toLocalDate ())));

  return datebook;
}
```

5. Open **src/cc/datetime/DatebookBrowser.java**. This class too is mostly implemented, with a formatter and printing method all ready to roll, and a constructor that stores a prepared **Datebook**. The **main** method calls your **buildSchedule** to populate a datebook, and then instantiates the browser class and asks it to **browse**:

```
public static void main (String[] args)
{
  Datebook datebook = Datebook.buildSchedule
    (Persistence.loadDatebook ().stream ());
  ...
  DatebookBrowser browser =
    new DatebookBrowser (datebook);
  browser.browse ();
}
```

Browsing a Datebook

DEMO

- The class is going to use the **Browser** utility that we last saw in an earlier chapter on functional programming.
- That means we'll need to keep track of our position in a list of "pages," which we model as distinct dates.
- And we'll need to plug in a number of functions to enable the generic browser to process the user's commands:
  - Functions to move to **next**, **previous**, **first**, and **last** pages
  - A function to **show** the current page (all events on this date)
  - A function to return a **label** for the page (a formatted date)

6. Add a field **currentDate** to the class:

```
private LocalDate currentDate;
```

7. Have the constructor initialize this to the first date in the given datebook:

```
public DatebookBrowser (Datebook datebook)
{
  this.datebook = datebook;
  currentDate =
    Collections.min (datebook.keySet());
}
```

8. Use this new field to initialize the variable **events** in the **show** helper method:

```
public void show ()
{
  List<Event> events = datebook.get (currentDate);
  ...
```

Browsing a Datebook

DEMO

9. Implement the empty **browse** method to create a **Browser**, with only some of the functions provided so far, and to call its **browse** method:

```
public void browse ()
{
  Browser browser = new Browser
    (
      () -> {},
      () -> {},
      () -> {},
      () -> {},
      this::show,
      () -> currentDate.format (formatter)
    );
  browser.browse (datebook.keySet ().size ());
}
```

   – The existing **show** function iterates the events for a given date and prints them out in a simple format.

   – The final lambda expression produces a label by formatting **currentDate**.

Browsing a Datebook

DEMO

10. If you test at this point, you'll get a working browser – or, one that doesn't crash, anyway – but it doesn't move off of the first page:

```
Page 1 of 5: Monday, February 23, 2015

    09:00  Discuss pre-meeting
    09:30  Pre-meeting
    09:50  Pre-meeting wrap-up
    10:00  Meeting
    10:15  Debriefing
    16:00  Follow-up conference call

Enter "next", "prev", or "quit".
next
Page 2 of 5: Monday, February 23, 2015

    09:00  Discuss pre-meeting
    09:30  Pre-meeting
    09:50  Pre-meeting wrap-up
    10:00  Meeting
    10:15  Debriefing
    16:00  Follow-up conference call

Enter "next", "prev", or "quit".
last
Page 5 of 5: Monday, February 23, 2015

    09:00  Discuss pre-meeting
    09:30  Pre-meeting
    09:50  Pre-meeting wrap-up
    10:00  Meeting
    10:15  Debriefing
    16:00  Follow-up conference call

Enter "next", "prev", or "quit".
quit
```

Browsing a Datebook

DEMO

11. Now you can implement the commands to move from page to page. Fill in the two easier of these first: just set **currentDate** to either the first or last date found in the map:

```
Browser browser = new Browser
  (
    () -> {},
    () -> {},
    () -> { currentDate =
      Collections.min (datebook.keySet ()); },
    () -> { currentDate =
      Collections.max (datebook.keySet ()); },
    this::show,
    () -> currentDate.format (formatter)
  );
```

   – If you test at this point, commands **first** and **last** should work.

12. The remaining two are a little more subtle. We don't want to bother to show an empty page when there are no events on a specific date, so we need to move from one date that has at least one event to the next. Use the **ceilingKey** and **floorKey** methods in **TreeMap** , along with simple date arithmetic, to make this easier:

```
Browser browser = new Browser
  (
    () -> { currentDate = datebook.ceilingKey
              (currentDate.plusDays (1)); },
    () -> { currentDate = datebook.floorKey
              (currentDate.plusDays (-1)); },
    () -> { currentDate = Collections.min... },
    ...
```

Browsing a Datebook

DEMO

13.Test again and see all commands working well:

```
Page 1 of 5: Monday, February 23, 2015

    09:00  Discuss pre-meeting
    09:30  Pre-meeting
    09:50  Pre-meeting wrap-up
    10:00  Meeting
    10:15  Debriefing
    16:00  Follow-up conference call

Enter "next", "prev", or "quit".
next
Page 2 of 5: Tuesday, February 24, 2015

    09:00  Work
    10:30  Break
    10:45  Work
    12:00  Lunch
    13:00  Work
    14:30  Break
    14:45  Work
    17:00  Off work
    19:30  Theatre tickets

Enter "next", "prev", or "quit".
last
Page 5 of 5: Sunday, March 01, 2015

    14:00  Band practice

Enter "next", "prev", or "quit".
quit
```

Rental Pricing

LAB 2

Suggested time: 30-45 minutes

In this lab you will implement the date and time arithmetic to support calculation of rental prices. Your method will be given starting and ending dates and times – or "rental and return" dates and times – and you will quote the best price by calculating weekly, daily, and hourly rates, and by applying break-even logic to, for example, quote the weekly rate if a six-day rental would be more expensive at the daily rate.

Optionally, you'll compute the earliest and latest return dates and times that will be acceptable without an early/late return fee.

Detailed instructions are found at the end of the chapter.

Time Zones and Offsets

- **The Time API also supports "zoned time" and time offsets.**
  - This allows management of **absolute time** by comparing local times in different time zones based on their offsets.
  - We can also **convert** a local time in one zone to local time in another zone – for example to represent our current time on a server to a client located elsewhere in the world.
- **We can calculate time offsets in one of two ways:**
  - Expressing a **fixed offset**, such as for a time that is always two hours ahead of UTC.
  - Identifying a **time zone**, in which there may be more complex rules that define the offset from UTC – for example in locales where there is a daylight-savings time such that the offset from UTC varies by date.
- **Time zones are identified in a variety of ways, and the Time API supports many of them by parsing strings of different formats:**
  - Offsets prefixed with "Z", "GMT", "UTC", or "UT"; and then expressing an offset with a **+/-** sign and a number of hours, or in some cases a duration expressed in hours and minutes.

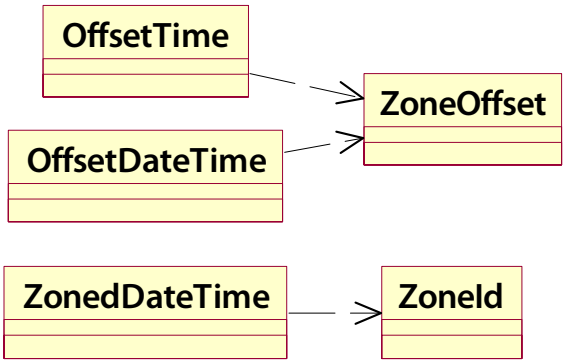
```
GMT+5
UTC-04:30
```

  - Descriptive identifiers expressed and region and locality, as managed in the IANA Time Zone Database.

```
Africa/Harare
America/Hermosillo
```

OffsetDateTime and ZonedDateTime

- The Time API models this problem by distinguishing between the levels of specificity in UTC offset information:



  - **OffsetDateTime** encapsulates times with fixed offsets, and only knows the numerical offset from UTC.
  - **ZonedDateTime** encapsulates full time-zone information and can apply whatever rules exist in that time zone in order to calculate the right time offset.
  - **ZoneOffset** captures the numerical data of a fixed offset, while **ZoneId** identifies a time zone and can use the **ZoneRules** therein.

OffsetDateTime and ZonedDateTime

- The API design and usage model of zoned and offset **Temporal**s are the same as for local dates and times: use **now**, **of**, **from**, **with**, **plus/minus**, etc., as you would with locals.
- To convert between local and offset/zoned times, use **of** and **toXXX** methods on the zoned/offset classes.
  - Create a **ZonedDateTime** or **OffsetDateTime** from a **LocalDateTime** with the factory method **of**, providing a **ZoneOffset** or **ZoneId**.

```
ZoneOffset offset = ZoneOffset.ofHours (-8);
OffsetDateTime offset =
  OffsetDateTime.of (localDateTime, offset);

ZoneId zone = ZoneId.of ("Europe/Paris");
ZonedDateTime zoned =
  ZonedDateTime.of (localDateTime, zone);
```

  - Reduce to local time with **toLocalDate/Time** methods. These don't offset into a different zone, or to UTC; they just render the "local part" of the offset or zoned date-time.

```
LocalDateTime local1 = offset.toLocalDateTime ();
LocalDateTime local2 = zoned.toLocalDateTime ();
```

Converting Between Time Zones

- If you need to calculate or convert a time in one time zone to a time in another, use one of the **withXXX** methods on either **OffsetDateTime** or **ZonedDateTime**.
- You can think of either of these classes as combinations of local times and offsets/zones – or as instants, which are fixed on UTC.
- So, converting means changing the time zone and deciding which thing you want to keep invariant:
  - Same local time in a different part of the world? Use **withOffset/ZoneSameLocalTime**, and the absolute **Instant** will change:

```
ZoneOffset cairo = ZoneOffset.ofHours (+1);
OffsetDateTime inCairo =
  offset.withZoneSameLocalTime (cairo);
```

  - Same moment in time, measured elsewhere? Use **withOffset/ZoneSameInstant**, and the local time will change:

```
ZoneId tahiti = ZoneId.of ("Pacific/Tahiti");
ZonedDateTime inTahiti =
  zoned.withZoneSameInstant (tahiti);
```

Cancellation Deadline, Translated

DEMO

- **Let's add one more feature to the Rental application: let the prospective renter know the date and time by which they would need to cancel their rental reservation without penalty.**
- **And, since they will be in their part of the world when that instant passes, let's put it in their local time rather than ours.**
  - Do your work in **Rental/Step2** or in your completed lab project.
  - The final version of the application is in **Rental/Step3**.

1. Open **src/cc/rental/Rental.java** and give the class a second formatter that will show the time zone:

```
private static final DateTimeFormatter
  FORMATTER_TRANSLATED = DateTimeFormatter
    .ofPattern ("MMMM dd, yyyy 'at' HH:mm zzzz");
```

2. Also define the time zone in which the rentals occur; let's say that's somewhere on the west coast of the United States:

```
private static final ZoneId RENTAL_ZONE =
  ZoneId.of ("US/Pacific");
```

3. Give the **rent** method one more parameter: a string called **zone**. This will allow the caller to identify their local time zone. (It will also break all the client code in the **main** method – but we'll get to that.)
4. At the bottom of the **rent** method, develop a zoned date-time representing the cancellation deadline, which is 24 hours before the start of the rental, in the rental company's locale:

```
ZonedDateTime cancel = ZonedDateTime
  .of (start.minusDays (1), RENTAL_ZONE);
```

Cancellation Deadline, Translated

DEMO

5. Now, translate that information to the client's stated time zone, as parsed from the new method parameter:

```
ZonedDateTime cancelTranslated =
  cancel.withZoneSameInstant (ZoneId.of (zone));
```

6. Write the information to the console:

```
System.out.println
  ("  If necessary, please cancel by "
    + FORMATTER_TRANSLATED.format
      (cancelTranslated));
System.out.println ();
```

7. Now you'll need to update **main** to supply time zones with each call to **rent**:

```
if (args.length > 3)
  rent (args[0], args[1], args[2],
    args[3], args[4]);
else if (args.length != 0)
  System.out.println ("Please provide ...");
else
{
  rent ("2015-01-22", "10:00:00", "2015-01-25",
    "15:30:00", "US/Eastern"); // $165
  rent ("2015-01-22", "09:00:00", "2015-01-25",
    "18:30:00", "GMT+3"); // $180
  rent ("2015-01-22", "10:00:00", "2015-01-31",
    "15:30:00", "Asia/Hong_Kong"); // $320
  rent ("2015-01-22", "10:00:00", "2015-02-03",
    "15:30:00", "UTC-06:30"); // $400
  rent ("2015-01-22", "10:00:00", "2015-02-02",
    "15:30:00", "America/Caracas"); // $400
  rent ("2015-01-22", "10:00:00", "2015-01-22",
    "22:30:00", "Pacific/Auckland"); //  $45
}
```

Cancellation Deadline, Translated

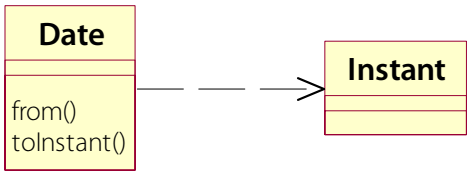
DEMO

8. Run the application and see cancellation times in various time zones:

```
Rental from January 22, 2015 at 10:00 AM
  to January 25, 2015 at 03:30 PM:
  ...
  If necessary, please cancel by
    January 21, 2015 at 13:00 Eastern Standard Time

Rental from January 22, 2015 at 09:00 AM
  to January 25, 2015 at 06:30 PM:
  ...
  If necessary, please cancel by
    January 21, 2015 at 20:00 GMT+03:00

Rental from January 22, 2015 at 10:00 AM
  to January 31, 2015 at 03:30 PM:
  ...
  If necessary, please cancel by
    January 22, 2015 at 02:00 Hong Kong Time

Rental from January 22, 2015 at 10:00 AM
  to February 03, 2015 at 03:30 PM:
  ...
  If necessary, please cancel by
    January 21, 2015 at 11:30 UTC-06:30

Rental from January 22, 2015 at 10:00 AM
  to February 02, 2015 at 03:30 PM:
  ...
  If necessary, please cancel by
    January 21, 2015 at 13:30 Venezuela Time

Rental from January 22, 2015 at 10:00 AM
  to January 22, 2015 at 10:30 PM:
  ...
  If necessary, please cancel by January 22, 2015
    at 07:00 New Zealand Daylight Time
```

The Time API in Other Java SE APIs

- There has been only minimal revision of the parts of the Java SE Core API that use **java.util.Date**.
  - For example, **java.security.KeyStore** still uses **Date** to identify the creation instant of a keystore.
- The primary strategy for interoperating is to take advantage of conversion methods on **Date**, to and from **Instant**.



  - Call the factory method **from** to convert to a new **Date**:

```
oldMethod (Date.from (existingInstant));
```

  - Call **toInstant** on a **Date** to get an equivalent **Instant**:

```
Instant instant = oldMethod ().toInstant ();
```

- Though there are no API changes, JDBC does support types from the Time API in binding to SQL statements and result sets.
  - Use the **getObject** and **setObject** methods to convert between the SQL and Java types shown below:

SQL	Java
DATE	LocalDate
TIME	LocalTime
TIMESTAMP	LocalDateTime
TIME WITH TIMEZONE	OffsetTime
TIMESTAMP WITH TIMEZONE	OffsetDateTime

SUMMARY

- **The Time API as introduced in Java 8 offers a powerful and intuitive means of managing date and time information.**
- **At first blush it seems quite dense, and the behavior of many concrete types is factored into several smaller interfaces that may seem to overlap in functionality.**
- **But the upshot is a set of usage patterns that holds nicely over the whole API, and ultimately gives you what you need, where you need it.**
  - Synthesize new objects with **of**; convert using **with**; etc.
  - Do arithmetic with **plus**, **minus**, **between**, etc.
  - If a type can be supported – for example you'd want to be able to adjust a date-time to a new month-and-day – it usually is.
- **Offset and zoned times are basically combinations of local times with offsets and time zones.**
  - Most of the complexity here is in understanding and identifying the time zones themselves.
  - The time arithmetic involved in translating from one zone to another is captured in the **withZoneSameXXX** methods.
- **Convert between older APIs that use java.util.Date using the from and toInstant methods, making the Instant class the gateway to the rest of the Time API.**

Rental Pricing

LAB 2

In this lab you will implement the date and time arithmetic to support calculation of rental prices. Your method will be given starting and ending dates and times – or "rental and return" dates and times – and you will quote the best price by calculating weekly, daily, and hourly rates, and by applying break-even logic to, for example, quote the weekly rate if a six-day rental would be more expensive at the daily rate.

Optionally, you'll compute the earliest and latest return dates and times that will be acceptable without an early/late return fee.

Lab project: **Rental/Step1**

Answer project(s): **Rental/Step2**

Files: * to be created
src/cc/rental/Rental.java

Instructions:

1. Open **Rental.java** and see that there are pre-defined rates for renting by the week, day, and hour. (We don't actually say what it is we're renting here - cars? boats? furniture? tuxedoes? - but it really doesn't matter to the logic of the application.)

   There is also a defined **GRACE_PERIOD**, an empty **rent** method, and a **main** method that drives the **rent** method a few times, with different test data and comments showing expected results.

2. Give the class a static, final field **FORMATTER** of type **DateTimeFormatter**, and initialize this based on a pattern of "MMMM dd, yyyy 'at' hh:mm a". The single quotes assure that the word "at" will appear literally, and not be mistaken for a formatting field itself. The 'h' characters are for a 12-hour modulus, and the 'a' character at the end will render as "AM" or "PM".
3. Start your implementation of the **rent** method by parsing the given start date and time into a **LocalDateTime** called **start**. (Easiest here is to **parse** each of the two strings separately, and then use **LocalDateTime.of** to combine the results.)
4. Do the same with the other two parameters to derive a **LocalDateTime** called **end**.
5. Validate the proposed rental by asserting that the **start** precedes the **end**. For this you can use **compareTo** or, more strongly typed and maybe clearer, **isBefore**. If the test fails, print an error message and **return**.
6. Print out the rental period in a user-friendly way, using the **FORMATTER** to produce the **start** and **end** date-times.

Rental Pricing LAB 2

7. Now, you need to figure out how long a rental this is. You could call **Period.between**, but only on the **LocalDate** parts of your date-times, and you wouldn't get sufficiently precise results.

   You could call **Duration.between**, but this would give you a duration in seconds and nanoseconds, and all the math to divide this down to weeks, days, and hours would be up to you.

   So the best approach is going to be to calculate one **ChronoUnit** at a time, and then adjust to get each smaller unit as the "remainder." First, call **ChronoUnit.WEEKS.between**, passing **start** and **end**, and capture the result as a long called **weeks**.
8. Now, get a **LocalDateTime** named **afterWeeks** by calling **start.plusWeeks** and passing **weeks**. This will have the effect of advancing the start time by an even number of weeks, leaving only days and smaller units to be measured.
9. Call **ChronoUnit.DAYS.between** – passing **afterWeeks** and **end**, this time – and capture the result as **days**.
10. Create another **LocalDateTime** called **afterDays**, and set it to **afterWeeks** plus **days** days.
11. Get a local variable **hours** as the number of hours between **afterDays** and **end**.
12. Now, you don't care about counting minutes and seconds, but you do want to round up. So derive a **LocalDateTime** called **afterHours**, and simply see if it still **isBefore** your **end** date-time. If so, increment **hours**.
13. Okay, now you have the numbers you need to express the length of the rental period as it pertains to your rates. You'll work from the smallest unit to the largest now, calculating the rate for a given unit. Capture **int** variables **costForHours**, **costForDays**, and **costForWeeks**, each the result of multiplying one of your duration measurements by the appropriate dollar rate.
14. Print the resulting rental rate, as the sum of these three numbers, to the console.

Rental Pricing LAB 2

15. Run the application and see how you do. Some of your rates should match the expected rates shown in the **main** source code at this point – but some will be off:

```
From January 22, 2015 at 10:00 AM to January 25, 2015 at 03:30 PM:
$165

From January 22, 2015 at 09:00 AM to January 25, 2015 at 06:30 PM:
$185 // Expected $180

From January 22, 2015 at 10:00 AM to January 31, 2015 at 03:30 PM:
$320

From January 22, 2015 at 10:00 AM to February 03, 2015 at 03:30 PM:
$455 // Expected $400

From January 22, 2015 at 10:00 AM to February 02, 2015 at 03:30 PM:
$410 // Expected $400

From January 22, 2015 at 10:00 AM to January 22, 2015 at 10:30 PM:
$ 65 // Expected $45
```

    If the above matches your results, then you're good so far ... but clearly not done. The discrepancies are all the results of applying a rate for one unit of time that turns out to be more than just one of the encompassing unit of time would cost – for example charging for 23 hours instead of one day. You'll fix this next.

16. Add code to the bottom of your method to itemize the cost for each rental: show the weekly, daily, and hourly contributions to the resulting rate.
17. Run again and you'll see the issue more plainly: for example, the last rental would be cheaper as one day than it is as thirteen hours. All four rentals computations that don't show as expected are failing to apply break-even logic.
18. This is why we wanted to compute hourly, then daily, then weekly. Right after calculating **costForHours**, check to see if it's greater than **DAILY_RATE**. If it is, set **costForHours** back to zero, and increment **days**. This will have the effect of adding a day's rental to whatever the **costForDays** would have been.
19. Do the same thing for **costForDays**, comparing to **WEEKLY_RATE** and adjusting **weeks** if necessary.

Rental Pricing LAB 2

20. Run again, and you should see that your rates for all test cases now line up:

```
From January 22, 2015 at 10:00 AM to January 25, 2015 at 03:30 PM:

    3 days  at $ 45/day  -- $135
    6 hours at $  5/hour -- $ 30
                            ---
                      Total $165

From January 22, 2015 at 09:00 AM to January 25, 2015 at 06:30 PM:

    4 days  at $ 45/day  -- $180
                            ---
                      Total $180

From January 22, 2015 at 10:00 AM to January 31, 2015 at 03:30 PM:

    1 weeks at $200/week -- $200
    2 days  at $ 45/day  -- $ 90
    6 hours at $  5/hour -- $ 30
                            ----
                      Total $320

From January 22, 2015 at 10:00 AM to February 03, 2015 at 03:30 PM:

    2 weeks at $200/week -- $400
                            ----
                      Total $400

From January 22, 2015 at 10:00 AM to February 02, 2015 at 03:30 PM:

    2 weeks at $200/week -- $400
                            ---
                      Total $400

From January 22, 2015 at 10:00 AM to January 22, 2015 at 10:30 PM:

    1 days  at $ 45/day  -- $ 45
                            ---
                      Total $ 45
```

Rental Pricing LAB 2

Optional Steps

21. The rental company loses money if property isn't returned fairly close to the agreed time – too early or too late. So, find the date and time that are **GRACE_PERIOD** later and earlier than **end**, and print them to the console as the boundaries around acceptable return window. For example the final output for the first and last test case would look like this:

```
Rental from January 22, 2015 at 10:00 AM to January 25, 2015 at 03:30
PM:

    3 days  at $ 45/day  -- $ 135
    6 hours at $  5/hour -- $  30
                           ----
                      Total $ 165

  A fee of $50 is owed if returned
    before January 25, 2015 at 12:30 PM or
    after January 25, 2015 at 06:30 PM.

...

Rental from January 22, 2015 at 10:00 AM to January 22, 2015 at 10:30
PM:

    1 days  at $ 45/day  -- $  45
                           ----
                      Total $  45

  A fee of $50 is owed if returned
    before January 22, 2015 at 07:30 PM or
    after January 23, 2015 at 01:30 AM.
```

Chapter 3

I/O Streams

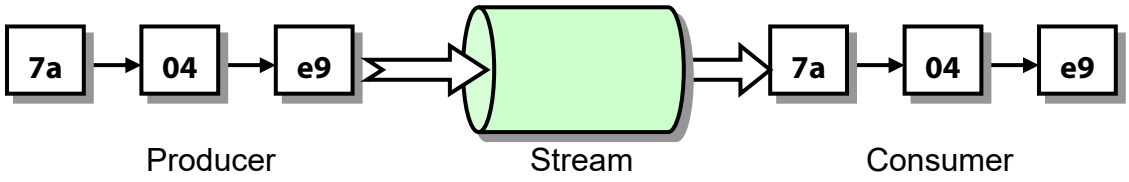
OBJECTIVES

After completing "I/O Streams," you will be able to:

- Describe the Java model for streaming data.
- Identify Java stream classes for input and output as delegating or non-delegating, and understand the uses of each.
- Use the **System** fields **in** and **out** to model the standard streams for input from and output to a console.
- Implement a delegating stream to perform filtering on output or input, regardless of source and destination media.

What is a Stream?

- **Many languages and platforms define the general concept of a stream: a first-in-first-out queue of bytes which is being (perhaps concurrently) fed and consumed.**



  - Java code is at one or both ends of the queue – as a **producer** of data and/or as a **consumer**.
  - The stream may be based in some medium such as a file or network connection, or perhaps just an array of bytes in memory or an array of characters – a/k/a a string.
- **Core API classes in java.io abstract ends of a stream.**
  - **Output streams** model the production end of a stream – so the producer **writes** to the output-stream object.
  - **Input streams** model the consumption end – so consumers **read** information back out of the stream.
- **It is the broad heterogeneity of these roles that makes the general concept of a stream so powerful.**
  - A stream isolates the calling code from details such as how the data is stored and retrieved, or transmitted and received, or filtered, or formatted, etc.
- **Do not confuse I/O streams with the newer library centered on java.util.stream – which models a stream of data elements for sequential or parallel processing.**

Java Streams

- **Generic stream behavior is modeled in base classes, one each for input and output.**
  - All stream classes extend one of these two classes: **InputStream** and **OutputStream**.
- **Some classes model specific types of stream endpoints, such as disk files or network ports.**
- **Many of the stream classes, however, have as their responsibility some modification of input or output behavior, and can only exist based on another input or output stream instance, known as the delegate.**
  - There are a base class and several subclasses just for the purpose of **filtering** either input or output in some way.
  - Similarly, the **Serialization API** is based on stream classes which delegate to any input/output stream instance.
- **Finally, there are completely separate hierarchies of classes to model character-based stream input and output – these are Readers and Writers.**

Advantages of Delegation

- **Java's delegation-based model allows each of many possible stream behaviors to be encapsulated in a different class.**
  - Instances of those classes can then be **chained** together, each delegating to the next, up to a terminal or non-delegating stream, to combine the desired behaviors.
  - Each stream in the chain contributes some desired feature, be it data formatting, buffering, or filtering out some unwanted subset of possible bytes.
- **There are many possible means of supporting a diverse set of behaviors, including basing the development of new behaviors in an inheritance hierarchy.**
- **Delegation offers a much more flexible infrastructure in this case.**
  - It is easy to choose exactly the features that will be needed from a particular stream (or chain of streams), because each stream object can be connected to any other.
  - In an inheritance-based model, combining three features would require the inheritance of three different classes, unless one or more of those features could reasonably be conceived as a specialization of another.
  - In Java, of course, only single inheritance is possible, so a delegation-based model seemed the obvious choice.

Standard Streams

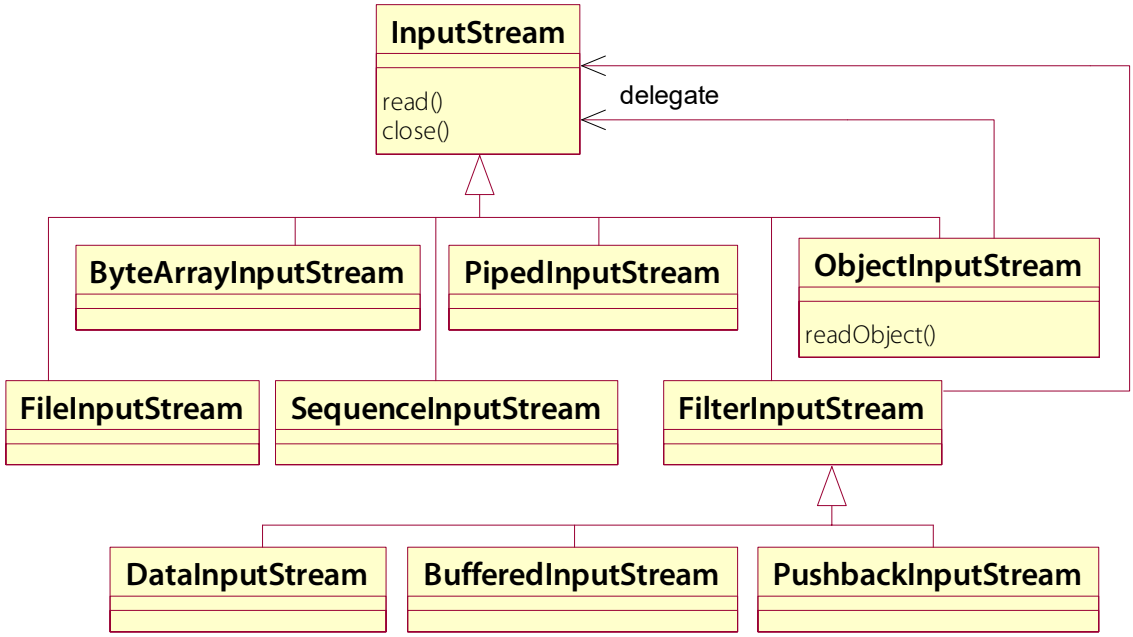
- **The java.lang.System class holds public fields which model standard streams:**
  - **System.in** represents the standard input stream, typically piped from the keyboard input of the interactive user.
  - **System.out** represents the standard output stream, for instance to print responses to a console – its concrete type is **PrintStream**.
  - **System.err** represents the standard error stream – it is also a **PrintStream**.
- **We've been using System.out throughout the course, as our only means of producing output (though that's about to change!).**

```
System.out.println ("Success in all endeavors.");
for (int i = 0; i < charArray.length; ++i)
  System.out.print (charArray[i]);
System.out.println ();
```

- **We've relied a bit on System.in as well.**
- **These stream references are to objects which are already initialized for you by the JRE, but which otherwise are like any other Java objects.**
  - They can be passed from method to method as parameters.
  - They can be used to initialize object fields.

Input Streams

- The various input stream classes extend **InputStream** according to the following hierarchy:



  - Note the **FilterInputStream** class, whose primary responsibility is simply to define a delegation relationship with some other input stream.
  - We'll consider the subtypes of **FilterInputStream** in a later chapter.
  - **ObjectInputStream** is the input end of the Serialization API, which we'll also study later.

InputStream

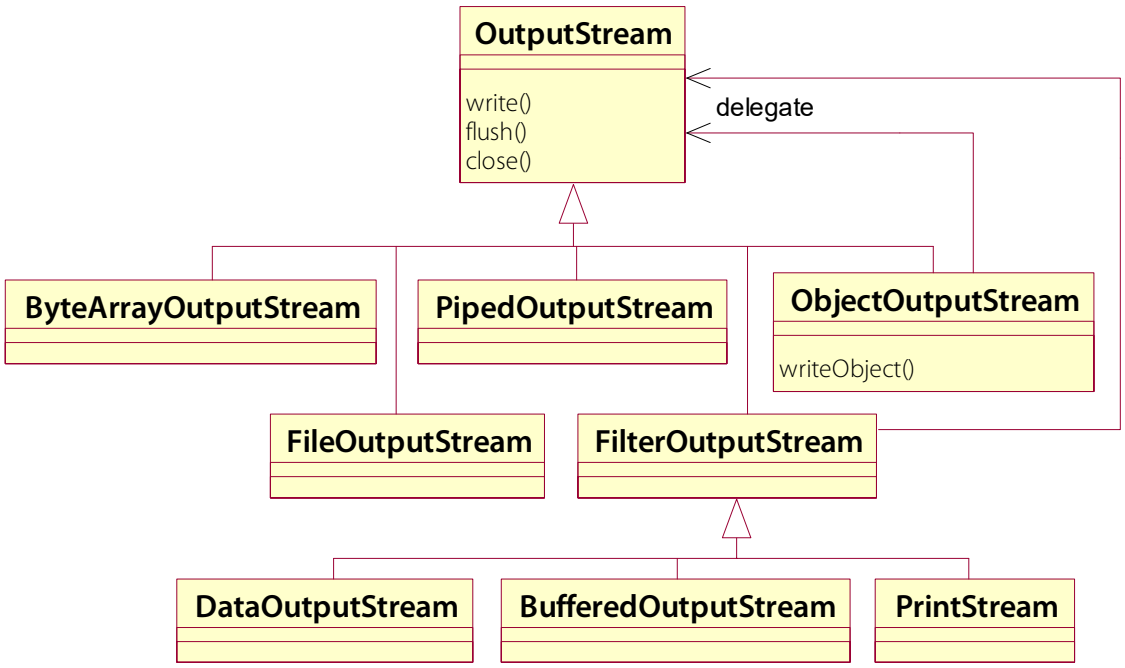
- Here is the public interface of the abstract **InputStream** class:

```
public abstract class InputStream
{
  public abstract int read () throws IOException;
  public int read (byte b[]) throws IOException;
  public int read (byte b[], int off, int len)
    throws IOException;
  public long skip (long n) throws IOException;
  public int available () throws IOException;
  public void close () throws IOException;
  public synchronized void mark (int readlimit);
  public synchronized void reset ()
    throws IOException;
  public boolean markSupported ();
}
```

- Note that the only abstract method is the first overload of **read**.
  - Everything else is either implemented in terms of **read** or is implemented trivially as a default.
  - This is a classic example of the abstract class as an almost-complete implementation: subclasses must define that **read** method as appropriate to their underlying medium or filtering behavior.
  - Many of the other methods, including **available** and **mark**/**reset**, are implemented differently or not at all on various subtypes.

Output Streams

- The various output stream classes extend **OutputStream** according to the following hierarchy:



  - For most of the input classes there are opposite numbers here.
  - The **PrintStream** is the one exception; this models a character-only output stream suitable for human reading.
  - This is a bit of a legacy in the Core API; it has been there since Java 1.0 (fifteen years ago as of this writing!) and has since been overtaken by the **Writer** hierarchy.
  - It lives in such fundamental places – such as **System.out** – that it persists in parallel with **Writer**s.

OutputStream

- Here is the public interface of the abstract **OutputStream** class:

```
public abstract class OutputStream
{
  public abstract void write (int b)
    throws IOException;
  public void write (byte b[]) throws IOException;
  public void write (byte b[], int off, int len)
    throws IOException;
  public void flush () throws IOException;
  public void close () throws IOException;
}
```

- As with **InputStream**, only one method is abstract, in this case the simplest overload of **write**.
  - The subclass must define an implementation for this method.
  - The other **write** methods delegate to the abstract one.
  - **flush** and **close** are implemented trivially and are likely to be overridden in various subclasses as appropriate.
  - Note the distinction between **flush**, which is not applicable to input, and **close** – **flush** assures the **durability** of all write operations by flushing any buffers between the Java process space and the ultimate physical destination, such as a file or network connection.

Non-Delegating Streams

- **The input and output stream types which do not delegate to other streams are each targeted to some specific endpoint or type of behavior:**
  - File I/O is modeled through the **File*Stream** classes; this will be the focus of our study in the next chapter.
  - You can model a chunk of memory as a stream for input or output using the **ByteArray*Stream** classes.
  - If you want **pipe** information between two threads, you can use the **Piped*Stream** classes. (Note – these do **not** map to any underlying "pipes" support in any given platform. They are for Java thread to Java thread, in a single JVM process.)
- **Though not found in the java.io package, there are many other medium-specific types, which are usually derived by factory methods on other APIs:**
  - For instance, one can open an input or output stream on a TCP/IP socket by calling methods in the **java.net** package.
  - The exact runtime type of the stream is hidden, which is appropriate to the streams pattern – the caller only needs to model the returned object as a stream.
- **We'll look at file-based streams in the next chapter.**
- **We'll consider network sockets, and streams that they provide, in a later chapter.**

Closing Streams

- **Because many media-based streams manage resources outside the JVM – such as files or TCP sockets – they must be closed explicitly when no longer needed.**
  - Garbage collection is not sufficient.
  - A call to **close** is needed, or the external resource will not be managed correctly: the file will be locked, the port unavailable, etc.
- **The traditional best practice for working with streams, then, has been to try code that opens a stream and then assure that close is called in an associated finally block.**
- **As of Java 7, the "try-with-resources" syntax is a better option:**

```
try ( FileOutputStream out =
        new FileOutputStream ("NewFile.txt"); )
{
  out.write (...);
}
```

  - The variable(s) declared as **resources** must be of types that implement **AutoCloseable**.
  - The **close** method will be called on each, as if it had been called in a **finally** block.
- **For some output-stream types, including socket streams, it's also important to remember to call flush, even if it's not yet time to call close.**
  - Output can wait in a local buffer until flushed.
  - This can leave your code deadlocked, e.g. waiting for a response to a request that it hasn't actually sent.

Byte-Array Streams

- Often it is convenient to model some data set in memory as an array of bytes.
- A pair of stream classes facilitate integration of streaming code with application code that models data in byte arrays.
  - **ByteArrayInputStream**
  - **ByteArrayOutputStream**
- The byte array essentially acts as a buffer, but since it's directly exposed to application code it can be used for much more than a basic performance booster.
- To read an existing byte array as a stream, wrap it in a **ByteArrayInputStream**:

```
InputStream in = new ByteArrayInputStream (array);
InputStream in =
  new ByteArrayInputStream (array, offset, length);
```

- Create a new buffer for output using a **ByteArrayOutputStream**.
  - A nice feature here is that you needn't decide the size of the buffer in advance – although you can if you know it.

```
try (ByteArrayOutputStream out = new
      ByteArrayOutputStream (); )
{
  out.writeInt (5);
  out.writeUTF ("Hello");
  byte[] buffer = out.toByteArray ();
}
```

Byte-Array Streams

EXAMPLE

- In **IOStream**, see **src/cc/io/ReadinAndRitin.java**, which illustrates basic usage of **InputStream** and **OutputStream**.
  - The **main** method first allocates an in-memory stream as a resource, and writes bytes to it, using each of the three overloads of **write**:

```
final byte[] alphabet =
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ".getBytes ();

try
(
  ByteArrayOutputStream out =
    new ByteArrayOutputStream ();
)
{
  out.write (97);
  out.write (98);
  out.write (99);
  out.write (' ');
  out.write (alphabet);
  out.write (' ');
  out.write (alphabet, 10, 10);
  out.write (' ');
  out.write (120);
  out.write (121);
  out.write (122);
```

  - It's then possible to derive the complete output, as a byte array:

```
byte[] buffer = out.toByteArray ();
```

Byte-Array Streams

EXAMPLE

– Now, in a nested **try** block, we open a stream interface to that same block of memory, and read it back out – first as individual integers (where some stream types use -1 to indicate end-of-file):

```
byte[] destination = new byte[26];
try
(
  ByteArrayInputStream in =
    new ByteArrayInputStream (buffer);
)
{
  System.out.println ("First three ...");
  for (int i = 0; i < 3; ++i)
    System.out.print ((char) in.read ());
  System.out.println ();
  in.read ();
```

– Note the single call to **read**, just to "burn" a byte in the stream that we know will be there but don't need to capture.

– Then we read into an allocated buffer, reading as many characters as the buffer can hold and are available:

```
  System.out.println ("Uppercase alphabet:");
  in.read (destination);
  System.out.println
    (new String (destination));
  in.read ();
```

– Now we set a bookmark in the stream, at the current position. (The argument to **mark** would limit how many bytes past the mark we can read; it's irrelevant for byte-array streams.)

```
  if (in.markSupported ())
    in.mark (-1);
```

Byte-Array Streams

EXAMPLE

– We read more characters and produce them as we go:

```
System.out.println ("Next ten characters:");
for (int i = 0; i < 10; ++i)
  System.out.print ((char) in.read ());
System.out.println ();
in.read ();
```

– Then we read some characters into an existing buffer, overwriting values at a given offset and for a given length:

```
in.read (destination, 23, 3);
System.out.println ("Mixed alphabet:");
System.out.println
  (new String (destination));
```

– Now, we go back to our bookmark, and read the next ten characters from the stream a second time, converting to lowercase as we go:

```
if (in.markSupported ())
{
  in.reset ();

  System.out.println ("Same ten, modified:");
  for (int i = 0; i < 10; ++i)
    System.out.print
      ((char) (in.read () + 32));
}
```

Byte-Array Streams

EXAMPLE

- **Run this class as a Java application and see the content parceled out in these various stages:**

```
First three characters:
abc
Uppercase alphabet:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Next ten characters:
KLMNOPQRST
Mixed alphabet:
ABCDEFGHIJKLMNOPQRSTUVWxyz
Same ten characters, modified:
klmnopqrst
```

  - Notice the "xyz" overwriting "XYZ" at the end of the alphabet, the second time it is printed. This is the effect of the call to **read** that takes a byte array, offset, and length.
  - And see that our bookmark worked, as we re-read and then demote to lowercase "klmnopqrst".

System.in, System.out, and System.err

- The standard streams provided by various operating systems are modeled as streams in the Core API, as static fields on the **System** class.
- **System.out** and **System.err** are of type **PrintStream**, which defines the **print** and **println** methods.
- **System.in** represents the user's keyboard input as a plain old **InputStream**.
  - One quirk is that you won't be able to read any input until the user hits **Enter** or **Return** to submit a complete line of text; this is actually a design choice in the various operating systems.
  - It means that a call to **System.in.read** can still be waiting after the user hits a key. So it's best to prompt the user to hit **Enter** or **Return** as a trigger to start or to cancel a procedure.
  - Also, **System.in.available** is meaningful: you can test this in order to avoid blocking with a call to **System.in.read**.

```
while (someCondition)
  try
  {
    if (System.in.available () != 0)
      doSomethingWith (System.in.read ());
  }
  catch (IOException ex) { ... }
```

  - We'll see other **System.in** techniques in later chapters, as well.
- Finally, note that there is no need to call **close** on any of these streams – though it's harmless to do so, for example in a general-purpose method that does stream processing.

Delegating Streams

- **Some streams can only function by delegating to others.**
- **This reflects the design principle of decoupling features that are ultimately independent.**
  - One stream class may be responsible for interacting with the file system, and another may know how to read and write network connections.
  - Orthogonal to these choices are features like buffering, formatting various data types as streams of bytes, or filtering certain kinds of data.
- **For instance the Data*Stream classes provide methods to read or write each of the Java primitive types, plus some support for reading and writing strings.**
  - These streams are useless without delegates.
  - On the other hand, they can be attached to any other stream type.
  - This means that the feature of data formatting can be connected to a file, or a network connection, or a byte array to be used in memory in some other way.
  - It can also be chained to some other delegating stream, such as a buffering stream.
- **We'll look at these sorts of streams, and see what you can do with them, in a later chapter.**

Chaining Streams

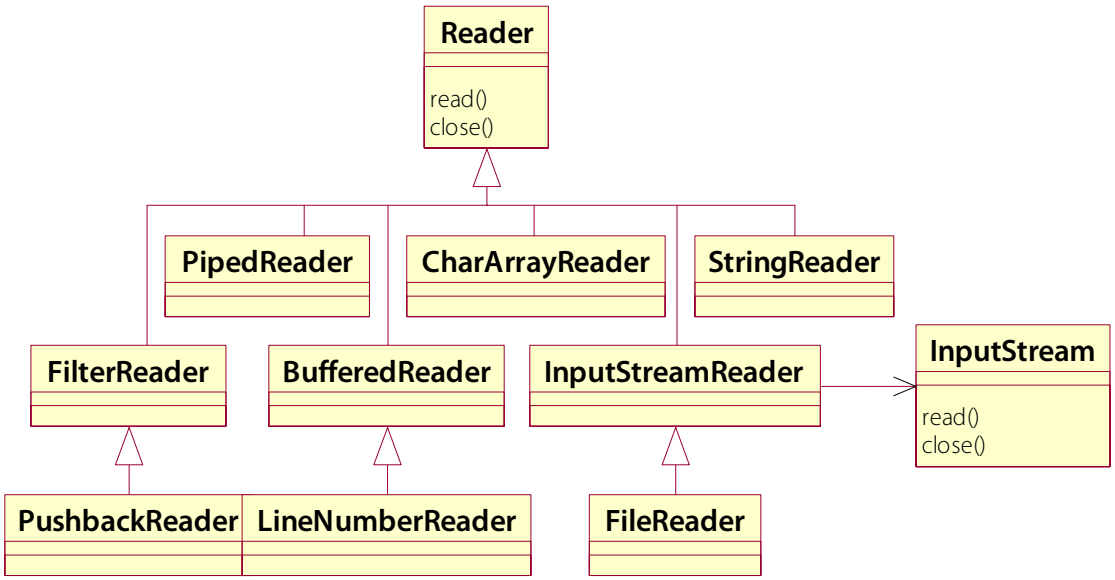
- As an example of chaining delegating streams together, consider the following code:

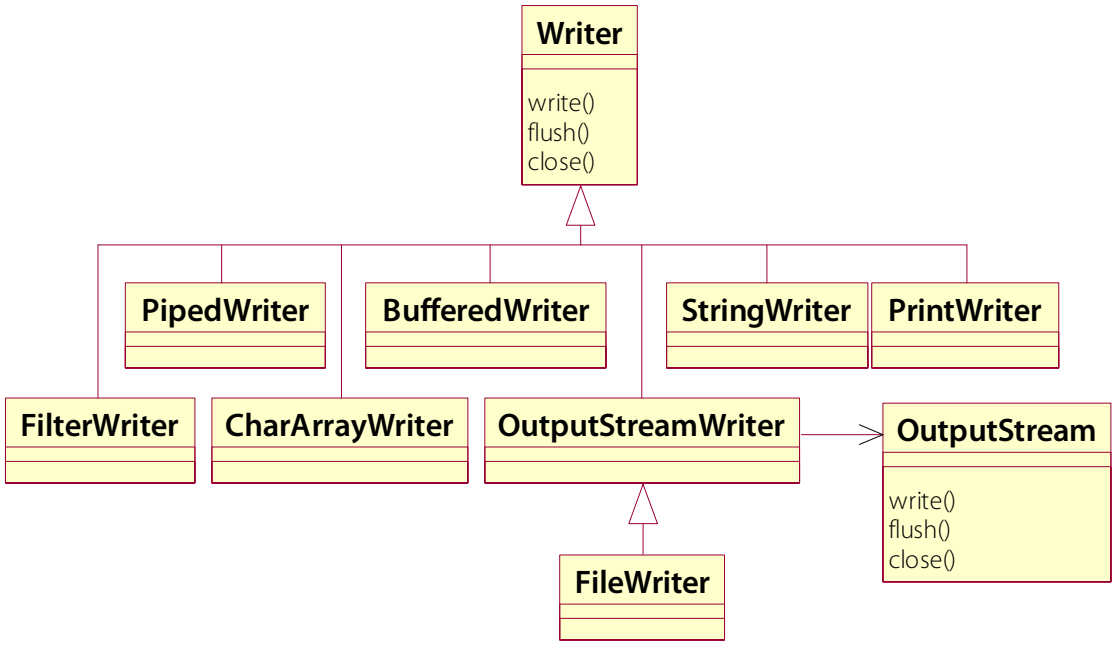
```
public static void main (String[] args)
{
  try
  (
    DataOutputStream dos = new DataOutputStream
      (new BufferedOutputStream
        (new FileOutputStream ("myfile.dat"));
  )
  {
    dos.writeInt (5);
    dos.writeUTF ("Hello, UTF!");
  }
  catch (IOException ex)
  {
    System.out.println ("Bad things happened.");
  }
}
```

- This builds a buffering stream based on a data-formatting stream which is based in turn on a file output stream.
- The result is that each of the individual features (buffering, automatic data formatting per type, and file-targeted output) is offered by the combination.

Readers and Writers

- Hierarchies of classes known as **Readers** and **Writers** are dedicated to character-based streams.





- Some of these delegate to raw stream objects, and some manage their underlying media directly.

Pretty Bad Privacy

LAB 3

Suggested time: 45-60 minutes

In this lab you will implement a filtering stream class that converts English into Pig Latin. Your class will extend **OutputStream** and will chain to any **PrintStream**; as characters are written to your stream you will cache them appropriately, and when you encounter whitespace you will write the modified tokens out to the delegate stream.

In optional steps, you will also explore resource management: the classic try/finally approach to assuring that streams and files are reliably closed, and then the try-with-resources feature of Java 7 for the sake of comparison.

Detailed instructions are found at the end of the chapter.

SUMMARY

- The Core API defines a simple but powerful model for stream programming.
- Many provided classes offer useful features that can be chained to any type of stream implementation.
- Many extensions of this system are therefore possible, each new extension implicitly leveraging a broad feature set.
- Some of the classes in the **java.io** package directly encapsulate file I/O, and these will be treated in the next chapter.
- Some extensions implemented elsewhere in the Core API include network communication streams, which are covered in another course, and I/O from compressed archives, in the **java.util.zip** package.

Pretty Bad Privacy

LAB 3

In this lab you will implement a filtering stream class that converts English into Pig Latin. Your class will extend **OutputStream** and will chain to any **PrintStream**; as characters are written to your stream you will cache them appropriately, and when you encounter whitespace you will write the modified tokens out to the delegate stream.

In optional steps, you will also explore resource management: the classic try/finally approach to assuring that streams and files are reliably closed, and then the try-with-resources feature of Java 7 for the sake of comparison.

Lab project: **PigLatin/Step1**

Answer project(s): **PigLatin/Step2** (intermediate)
PigLatin/Step3 (intermediate)
PigLatin/Step4 (final)

Files: * to be created
src/cc/language/Translator.java

Instructions:

1. Open the source file **src/cc/language/Translator.java** and review the starting contents: it imports the **java.io** and **java.util** packages and defines the public class **Translator**, with an inner **enum** that we'll use later to test characters to see if they are vowels.
2. First, make the class extend **OutputStream**.
3. Add a field **delegate** of type **PrintStream**, and initialize it in a constructor that takes a **PrintStream** argument.
4. Implement a dummy override of **write** (the overload that takes an **int**): just **print** the character to the **PrintStream**. (Note that an output stream and a print stream have different ways of treating non-printable bytes, so this is not the most robust filtering stream in creation, but hey – for Pig Latin, it'll do.)
5. Implement an application method **main**. Use try-with-resources to create an instance of **Translator**, passing **System.out** as the delegate stream, and **catch** any **IOException**. Run an infinite loop that reads a byte from **System.in** and echoes it by calling **write** on the translator instance. But, if the byte is the character **=**, break out of the loop so as to end the program. Build and test at this point. You should see that when you type a line of text it is echoed to you. (Each character is not echoed immediately because the standard input stream implementation automatically echoes by itself, and only unlocks the output stream line-by-line.) Hit [=] and **[ENTER]** to quit.

Pretty Bad Privacy LAB 3

6. Add String fields **beginning**, **middle**, and a **static final** called **ENDING**. Initialize the first two to be empty strings (not **null**) and make the last one "**ay**".
7. Add a boolean field **foundMiddle**, initializing it to **false**.
8. Now we're ready to try translating. Empty out your **write** implementation and start again. First get the output as a **char** by doing a conversion on the method argument. Call the local variable **character**.
9. Test if **character** is a letter in the alphabet – use the static method **Character.isLetter**. If it is not, you are ready to write a new token – a word translated into Pig Latin – to the delegate. Just write an empty code block for the moment; we'll look at how the incoming characters are parsed before we consider how to write the modified token.
10. If **character** is a letter (**else** and a second code block), then the job is to find the first character of the new word, which is the first vowel encountered; then you will chop off the **beginning** consonants, put them after what used to be the **middle** of the word, and append the **ENDING** string. So first, if **foundMiddle** is still **false**, see if the vowels collection **contains** the character, and if it does set **foundMiddle** to **true**:

```
if (!foundMiddle && Vowel.valueSet.contains (character))
    foundMiddle = true;
```

11. Now – still in the **else** block – you can append the current character to either the **beginning** or **middle** strings, according to the value of **foundMiddle**.
12. Now go back to the empty block under **if**. When we encounter a non-alphabetic character, we know the whole word, and we've accumulated its characters into two segments. We will rearrange the order of those segments and add a third, which is the **ENDING** string. So call **delegate.print** for each segment: **middle**, **beginning**, **ENDING**.
13. Print the **character** itself, so that the spaces and punctuation that end words are not lost.
14. In the same block, re-initialize **beginning**, **middle**, and **foundMiddle** to their original values in order to be ready to parse the next word. Test at this point – see sample output below.

```
this is a test of the emergency broadcast system
isthay isay aay esttay ofay ethay emergencyay oadcastbray emsystay
Hello – is anybody out there?
elloHay - isay anybodyay outay erethay?
=
```

This is the intermediate answer in the **Step2** directory.

Pretty Bad Privacy LAB 3

Optional Steps

15. Though working with the system input and output streams in a console application doesn't really bring a resource-management problem to bear, we can use this as an early opportunity to explore and develop good practices for handling files, streams, and other non-garbage-collectable resources. First, add a method **close** to your class, and have it delegate to the underlying stream, after writing a line to standard output, so we'll know that the method has been called.
16. Now add a **finally** block to the **main** method, in which you **close** your **filter** object.
17. You'll find that, for this to compile, you need to do two things. First, wrap your call to **close** in a **try/catch** system, since it's signed to throw the **IOException** itself! Your catch can do nothing, but it has to be there.
18. Second, declare **filter** before the **try** block, so that it is in scope for the **catch** and the **finally** blocks as well. The trick is to declare it outside, and initialize it inside:

```
OutputStream filter = null;
try
{
    filter = new Translator (System.out);
    while (true)
    ...
```

19. Test, and see that when you finish a run, your **close** method is indeed invoked:

```
try this
isthay
=
Closing underlying stream.
```

This is the intermediate answer in the **Step3** directory.

Pretty Bad Privacy LAB 3

20. That was a bit of a fuss, though, wasn't it? Now try this: remove the **finally** block completely.
21. Initialize **filter** formally as a "resource" for the **try** block:

```
try ( OutputStream filter = new Translator (System.out); )
{
    while (true)
    ...
```

22. Build and test again, and see results identical to those for the **Step3** version.

This is the final answer in **Step4**, and it is a pretty good example of the motivation for the new feature in Java 7, and it applies equally well to files. The stream type implements **AutoCloseable** – or there would have been a compiler error – and the runtime knows to call **close** through that interface, and exactly when to call it in various success and failure scenarios.

Also famously difficult are the connection, statement, and result-set objects in **JDBC**, the Java SE API for working with SQL and databases. They too implement **AutoCloseable** and are a lot easier to use now.

CHAPTER 4

FILES AND FILE SYSTEMS

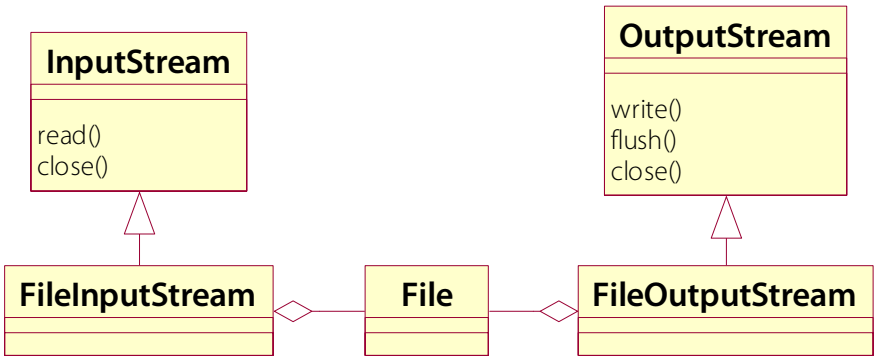
OBJECTIVES

After completing "Files and File Systems," you will be able to:

- Interact with the local file system using the file classes in the Core API.
- Use the **File** class to read directories and to get statistics on individual files, such as file length.
- Read and write files using the appropriate stream classes.
- Describe the function of the **RandomAccessFile** class for random, as opposed to sequential, file access.
- Navigate, walk, and process directory trees in local file systems.
- Apply **java.util.stream** processing to sets of files, and to file contents at the text-line and byte level.

Files and I/O Streams

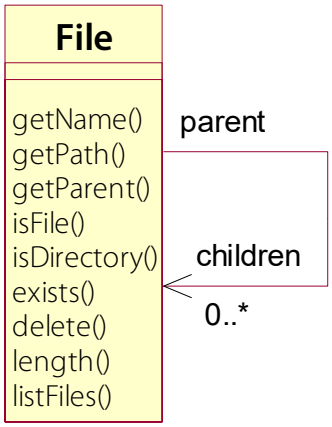
- Several of the **java.io** stream classes are designed to read and to write local files.
- They rely on the **File** class, which is the central abstraction for interactions with real files and directories on the local system.



- **File** can be used by itself to manage files and directories without reading or writing their contents:
  - **Create** and **delete** files
  - Read file **attributes** such as length and last-modified stamp
  - Convert between **relative and absolute paths**, and map path syntax to the underlying platform
  - **Navigate** directory trees

The File Class

- The class **java.io.File** models files and directories on the local file system.



- As with many other parts of the Core API, the **File** and associated classes provide a portable interface to functionality which ultimately depends on the platform/OS.
- There are of course security issues involved with file system access and manipulation.
  - The security policy in force will either grant or deny various permissions to the running code as it attempts operations on files and file streams.
  - We do not cover security at a detailed level in this course.

Managing Files

- **Create a File instance by providing a filename to the constructor.**
  - Both **absolute and relative paths** can work; anything that can be evaluated by the underlying file system.
  - Note that there are **portability issues** here, since different platforms have quite different file systems.
  - In at least some simple ways the **File** class and company hide these platform differences – most obviously the difference in separators between tokens in a path.
  - Always use **forward slashes** as your separator when describing file paths in Java. The native implementation will map that character to the correct separator for the OS.

```
File myFile = new File ("sub/sub2/file.txt");
    // This will map correctly in Unix or
    //  in Windows
```

- **You can query the File instance for information about the underlying file.**
  - Call **canRead** and **canWrite** to determine access rights.
  - Call **length** to get the file size.
  - Call **lastModified** to check the date and time of the most recent update. This returns a long, which can be converted to a **Date** or **Calendar** object, or can be used directly for simple comparisons, dependency checking, etc.
  - Call **delete** to delete the file.

Managing Directories

- **File also models directories or folders in the file system.**
- **You can perform directory queries and management through an instance of File.**
  - Create an instance on a path string that terminates with the directory of interest.
  - Check that you have a handle to a directory and not a file by calling **isDirectory** (or check that it's a file with **isFile**).
  - You can list the contents of the directory by calling **list**. This returns an array of strings.
  - **listFiles** is better for recursive processing – it returns an array of **File** objects.
  - **mkdir** will create a new directory, and **delete** can be applied to a directory as well as a file.

A Directory Listing

DEMO

- In **Directory/Step1** there is a simple application that prints a list of the objects in a given directory.
  - See **src/cc/files/DirJava.java**:

```
File currentDir = new File
  (args.length != 0 ? args[0] : ".");
if (currentDir.isDirectory ())
{
  String[] contents = currentDir.list ();
  for (String filename : contents)
    System.out.println (filename);
}
```

1. Run the starter version and see a simple directory listing of the project directory.

```
.classpath
.project
build
compile.bat
compile
doc
run.bat
run
src
```

2. Open **src/cc/files/DirJava.java** and change **contents** to an array of **File**s. Initialize this by calling **listFiles**.

```
File[] contents = currentDir.listFiles ();
```

3. Change the loop to define a **File** iterator, not a **String**:

```
for (File file : contents)
```

A Directory Listing

DEMO

4. Change the **println** to a **format** call and produce a long listing of the file's last-modified date and time:

```
for (File file : contents)
  System.out.format ("%s  %12d  %s%n",
    new java.util.Date (file.lastModified ()),
    file.length (),
    file.getName ());
}
```

5. Test again:

```
Fri Feb 06 10:43:32 EST 2015       316  .classpath
Fri Feb 06 10:43:32 EST 2015       334  .project
Mon Feb 09 17:09:35 EST 2015         0  build
Fri Feb 06 10:43:07 EST 2015        72  compile
Fri Feb 06 10:43:07 EST 2015       114  compile.bat
Fri Feb 06 10:43:07 EST 2015        56  run
Fri Feb 06 10:43:07 EST 2015        61  run.bat
Fri Feb 06 10:43:17 EST 2015         0  src
```

   – The completed demo is found in **Directory/Step2**.

File Streams

- Four classes in **java.io** rely on **File** to connect to a local file:
  - **FileInputStream**
  - **FileOutputStream**
  - **FileReader**
  - **FileWriter**
- **Their semantics differ slightly, but all are built with reference to a file or filename.**
  - You can create a stream instance on an existing File instance.
  - There is a shortcut constructor, widely used, that takes a filename and thus saves you a step.
  - Thus the following are equivalent:

```
File myFile = new File ("myFile.txt");
FileOutputStream fos =
    new FileOutputStream (myFile);

FileOutputStream fos2 =
    new FileOutputStream ("myFile.txt");
```

Working with Text Files

- Although raw streams can read and write text content perfectly well, the **Reader** and **Writer** types are dedicated to text content.
- Common techniques:
  - To read a line at a time from a text file:

```
try (BufferedReader in = new BufferedReader
      (new FileReader ("SomeFile.txt")); )
{
  String line;
  while ((line = in.readLine ()) != null)
    doSomethingWithTheString (line);
}
```

  - To read a command from the standard input stream:

```
try (BufferedReader in = new BufferedReader
      (new InputStreamReader (System.in)); )
{
  String command = in.readLine ();
  if (command.equalsIgnoreCase ("list"))
    ...
}
```

  - To write text to a file in segments or whole lines:

```
try (PrintWriter out = new PrintWriter
      (new FileWriter ("SomeFile.txt")); )
{
  out.print ("Starting task ... ");
  doSomething ();
  out.println ("Done.");
}
```

Analyzing a Directory Tree

LAB 4A

Suggested time: 30-45 minutes

In this lab you will implement a recursive method that can find the aggregate number of files under a root directory, and their total size in bytes. The starter class **Analysis** holds a single **File** object as a delegate, and its own **bytes** and **files** properties. You will implement the method **runAnalysis** to walk the directory tree and to capture running totals as it does so.

In an optional section at the end of the lab, you may also enhance the application to break down the disk usage for each immediate child of the directory as a percentage of the total for that directory.

Detailed instructions are found at the end of the chapter.

Random Access

- **Thus far we have discussed interactions with files through stream classes.**
- **Here input and output are naturally sequential.**
  - One must write the file contents in a precise order, and read them back in that same order.
  - If one wants to write or read at a certain known point in the file one must use the **InputStream** or **OutputStream** methods to advance the pointer to the desired location.
- **The java.io package also provides a means of working with random access files.**
  - These are formatted in fixed-length blocks, or **records**, making it possible to move to an arbitrary record number reliably, for either input or output.
  - This formatting is sometimes wasteful of file space, since not every record will need every available byte for storage of values, but performance is often much better.
- **The class RandomAccessFile models such files as arrays of bytes.**
  - In addition to the usual stream-based operations (the class implements both data-formatting interfaces), the class offers methods to get and set the file pointer to any position.
  - This class does not capture record length; calculating the offset of a particular record is up to you.

Working with File Systems

- **Over several recent versions of Java, and especially with functional-programming enhancements in Java 8, the language has acquired increasingly sophisticated utilities for common activities in file handling:**
  - **Listing** files in a directory
  - **Walking** a directory tree
  - **Searching** for files
  - **Processing** file contents
- **Much of this newer utility code is found in packages `java.nio.file`, and many methods return some sort of `java.util.stream.Stream`.**
- **There is some duplication of capabilities on `java.io.File`.**

The Path Interface

- Whereas the **File** class provides some convenience methods for interpreting and converting file paths, the newer **Path** interface offers richer functionality.
- A **Path** is a structured sequence of name elements, and for example can be processed, token-by-token, in a **for** loop.
- Here is a partial listing:

```
public interface Path
  extends Comparable<Path>, Iterable<Path>, ...
{
  public boolean endsWith (Path other);
  public boolean endsWith (String other);
  public Path getParent ();
  public Path normalize ();
  public Path resolve (Path other);
  public boolean startsWith (Path other);
  public boolean startsWith (String other);
  public Path subpath (int start, int end);
  public File toFile ();
  public String toString ();
}
```

The Path Interface

- Get a **Path** via a variety of factory methods on other types, but most simply using the class utility **Paths**:

```
Path workingDir = Paths.get (".");
Path deepFile =
  Paths.get ("child", "grandchild", "filename");
```

- **Path**s are immutable.
  - This makes them naturally **thread-safe**.
  - But beware for example calls to **normalize** or **relativize**: you may intuit that these will change the path in place, but they are only useful if you capture their return values.
- Another significant pitfall: don't call the **endsWith** method thinking that you can test for a file extension.
  - This won't work:

```
if (myPath.endsWith (".xml")) { ... }
```

  - **startsWith** and **endsWith**, along with the ability to iterate over the path, are entirely about the whole **name components** or tokens that make up the path.
  - In other words, it's not a **substring** test; **endsWith** will return **true** if the given string or path accounts for the entire final component.
  - To search for a file extension or suffix, convert the **Path** to a string:

```
if (myPath.toString ().endsWith (".xml")) { ... }
```

The Files Class Utility

- The **Files** class offers a large body of static utility methods, for functions as diverse as searching for files, copying and moving files, deleting, and reading individual file contents using **java.util.stream.Stream**s.
- Some of the highlights are shown here – all methods shown are **static** and **public**:

```
public class Files
{
  long copy
    (Path source, Path target, CopyOption...);
  Path createDirectories
    (Path path, FileAttribute<?>... attrs);
  Path createFile
    (Path path, FileAttribute<?>... attrs);
  Stream<Path> find (Path root, int maxDepth,
    BiPredicate<Path,BasicFileAttributes> matcher,
    FileVisitOption... options);
  Stream<String> lines (Path path);
  Stream<Path> list (Path directory);
  long move
    (Path source, Path target, CopyOption...);
  Stream<Path> walk
    (Path root, FileVisitOption... options);
}
```

  - Note that **createDirectories** will create a chain of parent and child directories, if necessary to establish the path that you require.
  - **File.mkdir** will not do this; you must create one at a time or be ready to handle **IOException**s.
  - **lines** opens a **BufferedReader** on the file at the given path, and provides a stream representing the result of each call to **readLine**.

Searching for a File

EXAMPLE

- In **Files/Step1**, see **src/cc/files/Search.java**, which implements a search for a given filename under a root directory:

```
long start = System.currentTimeMillis ();
try
{
  Files.walk (path)
       .parallel ()
       .peek (Search::countFilesAndThreads)
       .peek (Search::showStatus)
       .filter (p -> p.endsWith (filename))
       .peek (p -> ++filesFound)
       .forEach (p -> System.out.println
                        (CLEAR + "Found " + p));
}
```

  - We use the consolidated **Stream<Path>** given to us by the **Files.walk** method to seek files of a given name.
  - The core logic resides in the calls to **filter** and **forEach**.
  - See that **filter** uses **endsWith** as designed, to test for equivalence to the entire filename.
  - The **peek** calls allow us to profile the code, counting thread usage and files found, and to keep the user up to date on the progress of a longer search process.
  - We take advantage of the ability to **parallelize** the processing, which is built into the **java.util.stream** API.

Searching for a File

EXAMPLE

- **You can run this class as a Java application, to search for itself (!) as a source file within the project directory ...**

```
Searching for file:  Search.java
From root directory: .

Found .\src\cc\files\Search.java
Time: 235 msec
Files searched: 31
Threads used: 1
```

  - This may look better in a command console than in the console view of your IDE, because it uses a carriage-return control character to clear output and re-write the same text line.

- **Or, pass program arguments to search for other files in other locations.**
- **You can try searching larger areas of the file system, and see how the stream performs, and you might try commenting out the call to parallel to see what sort of win there is in parallelizing the search.**
  - You will find that, for very small searches, it's no win at all; the ability to thread and parallelize never develops a time benefit that outweighs the overhead cost of setting it up.
  - For larger searches it will indeed be a win.

Making Backups

DEMO

- **Let's explore this API a bit further, by completing a utility class that creates backup copies of certain files.**
  - Do your work in **Files/Step1**.
  - The completed demo can be found in **Files/Step2**.

6. Open **src/cc/files/Backup.java** and see that instances of this class store a **suffix** by which they recognize a class of files.

```
private static final String
  BACKUP_SUFFIX = ".bak";
private String suffix;

public Backup (String suffix)
{
  this.suffix = suffix;
}
```

7. The **backup** method makes a backup copy of a given file:

```
public void backup (Path path)
{
  try
  {
    Files.copy (path, Paths.get
      (path.toString () + BACKUP_SUFFIX));
  }
  catch (IOException ex)
  {
    System.out.println ("Couldn't copy ...");
  }
}
```

Making Backups

DEMO

8. There is also a method **backupAll** that is meant to recurse under a root path, and recognize files that need to be backed up, calling **backup** for each one. Start your implementation of this method by processing a directory list, filtering to files (not directories) and to files that end with the configured **suffix**, calling **backup** on each:

```
public void backupAll (Path path)
{
  try
  {
    Files.list (path)
         .filter (p -> p.toFile ().isFile ())
         .filter (p -> p.toString ()
                       .endsWith (suffix))
         .forEach (this::backup);
  }
  catch (IOException ex)
  {
    System.out.println
      ("Couldn't list files under " + path);
  }
}
```

- The **main** method calls **backupAll**, looking for all files with the extension **.dat** under the working directory:

```
new Backup (suffix).backupAll
  (Paths.get (root));
System.out.println ("Created backups of all " +
  suffix + " files under " + root + ".");
```

Making Backups

DEMO

- **A tree of files is prepared, for testing purposes:**

```
Files
Step1
  files
    irrelevant.txt
    morefiles
      dontcopy
      three.dat
    one.dat
    two.dat
```

9. Test this out now:

```
Created backups of all .dat files under '.'.
```

10. Check the **files** folder to see if any backups were made. You won't see any new files. Why not?
    - So far you are not recursing from the main directory; you are only processing a flat list of files at that top level, and there were no qualifying files in **Files/Step1** itself.
11. If you change the code or pass a program argument "files" to the application, then you should see two backups at that level of the folder structure – but still no recursion from there:

```
Files
  Step1
    files
      irrelevant.txt
      morefiles
        dontcopy
        three.dat <-- missed this one
      one.dat
      one.dat.bak
      two.dat
      two.dat.bak
```

Making Backups

DEMO

12. Let's add recursion to **backupAll**. At the bottom of the method, add code to process directories, rather than files, and to call **backupAll** recursively for each one:

```
    try
    {
      Files.list (path)
           .filter (p -> p.toFile ()
                          .isDirectory ())
           .forEach (this::backupAll);
    }
    catch (IOException ex)
    {
      System.out.println
        ("Couldn't recurse under " + path);
    }
  }
```

13. If you have created any backup files at this point, delete them now.

14. Test again. Now the method recurses and you get all three backup copies, as designed:

```
Files
  Step1
    files
      irrelevant.txt
      morefiles
        dontcopy
        three.dat
        three.dat.bak
      one.dat
      one.dat.bak
      two.dat
      two.dat.bak
```

Making Backups

DEMO

15. Now take a look at **src/cc/files/Cleanup.java**. See that it uses a similar code structure to remove backup copies:

```
public static void cleanup (Path path)
{
  try
  {
    Files.delete (path);
  }
  catch (IOException ex) { ... }
}

public static void cleanupAll (Path path)
{
  try
  {
    Files.list (path)
         .filter (p -> p.toFile ().isFile ())
         .filter (p -> p.toString ()
                        .endsWith(BACKUP_SUFFIX))
         .forEach (Cleanup::cleanup);
  }
  catch (IOException ex)  { ... }

  try
  {
    Files.list (path)
         .filter (p -> p.toFile ().isDirectory())
         .forEach (Cleanup::cleanupAll);
  }
  catch (IOException ex) { ... }
}
```

16. Test this application out now, and see that it does indeed remove all three of the **.bak** files that you just created.

Making Backups

DEMO

17. But it turns out that this class, as well as **Backup.java**, could be written a little more cleanly. To wit, **Files.walk** will do the recursion for us that we've been doing ourselves, feeding us just the paths that it finds. So, let's tap into that – start by removing the **cleanupAll** method completely.

18. Now, in **main**, remove the call to **cleanupAll** and replace with simpler path-streaming code:

```
try
{
  Cleanup.cleanupAll (Paths.get (root));
  Files.walk (Paths.get (root))
       .filter (p -> p.toFile ().isFile ())
       .filter (p -> p.toString ()
                       .endsWith(BACKUP_SUFFIX))
     .forEach (Cleanup::cleanup);
  System.out.println
    ("Removed backups from " + root + ".");
}
```

19. Test again, by running first **Backup** and then **Cleanup**, and see that the files appear and then disappear as they should.

File Processing

- The ability to process whole files as lines of text, using **java.util.stream.Stream**, opens up new implementation strategies for many common operations.
- To get a **String** at a time from a text file, either
  - Open a **BufferedReader**, and call **lines**, or
  - Simply call **Files.lines**, providing the **Path**.
- For example, to read file content while squelching blank lines:

```
Files.lines (Paths.get ("myfile.txt"))
     .filter (line -> line.length () != 0)
     .forEach (aStringBuilder::append);
```

- To process a file one byte or character at a time, you need one additional trick, which is to get a stream of bytes from each string, and then to flatten that "stream of streams" into a single stream.
- For example, to read file content while squelching any non-printable-ASCII characters:

```
Files.lines (Paths.get ("myfile.dat"))
     .flatMapToInt (line -> line.chars ())
     .filter (c -> c > 31 && c < 127)
     .forEach (aStringBuilder::append);
```

Global Replacement and Checksums

LAB 4B

Suggested time: 30 minutes

In this lab you will build two additional file-processing applications. The first will perform a global replacement of all double-quote characters with the XML/HTML escape sequence `&quot;` and produce the resulting text.

The second application will derive a simple checksum from a given file, processing the file byte-by-byte and feeding each byte into a simple reducing formula.

Detailed instructions are found at the end of the chapter.

SUMMARY

- Management of file systems through the **File** class and related classes is straightforward.
- Code portability is maintained, here as in other branches of the Core API, through platform-specific implementations in the JRE.
- Reading and writing character streams and simple formatted data streams can be accomplished quite easily using the simpler stream classes.
- The **Reader** and **Writer** trees make it easier to work with printable-text files.
- The **Files** class utility and related types in **java.nio.file** can greatly simplify file-management tasks such as searching, moving, and copying, and provide some nice shortcuts to operations otherwise done with **File** and various types of streams, readers, and writers.

Analyzing a Directory Tree

LAB 4A

In this lab you will implement a recursive method that can find the aggregate number of files under a root directory, and their total size in bytes. The starter class **Analysis** holds a single **File** object as a delegate, and its own **bytes** and **files** properties. You will implement the method **runAnalysis** to walk the directory tree and to capture running totals as it does so.

In an optional section at the end of the lab, you may also enhance the application to break down the disk usage for each immediate child of the directory as a percentage of the total for that directory.

Lab project: **Analysis/Step1**

Answer project(s): **Analysis/Step2** (intermediate)
Analysis/Step3 (final)

Files: * to be created
src/cc/files/Analysis.java

Instructions:

1. Open **Analysis.java** and see that it already implements read-only JavaBeans properties **bytes**, **files**, and **target**. There is a constructor that initializes **target** and then calls the method **runAnalysis**, which will be responsible for finding **bytes** and **files** by a recursive process.
2. Begin your implementation of **runAnalysis** by checking if **target** is a file. If it is, you won't recurse any further: set the **bytes** property to the length of the file and the **files** property to one, and return from the method.
3. Now, you know you're working with a directory, so get an array of strings called **contents** by calling **target.list**.
4. Initialize both **bytes** and **files** to zero.
5. Create an array of **Analysis** references called **childProps** and initialize to the same length as **contents**.

Analyzing a Directory Tree LAB 4A

6. For each element in **contents**, create a corresponding **File** object. Then instantiate an **Analysis** based on that file, and assign it to the next element in **childProps**. (This is the point of recursion, because the constructor for **Analysis** will call its own **runAnalysis**, which will get child elements and create new **Analysis** objects, etc.) Read the **bytes** and **files** properties from each child as it's created and add these numbers to your own totals.
7. You may want to **build** at this point to check for compile errors.
8. Implement the **main** method to create a single **Analysis** object based on the one command-line argument. (You may want to implement a default that operates on the current directory, as the **DirJava** demo does.)
9. Print the total **bytes** and **files** to the console, using **System.out.format**.
10. Test, and if you run in the lab directory itself you should get something like the following (the size is partially based on the size of the source file you just built, so it may vary a bit):

```
Size :      6884
Files:         7
```

This is the intermediate answer in **Step2**.

Optional Steps

11. Add a new read-only field to the class: a **Map** from **String** to **Long** called **shares**. Initialize this to a new **TreeMap** and create the accessor method.
12. At the bottom of **runAnalysis**, loop over the **contents** array again. For each element, call **shares.put**, passing the filename itself as the key, and for the value a percentage based on the corresponding **childProps** element's **bytes** value, multiplied by 100, then divided by the total **bytes**.
13. At the bottom of **main**, print out the values in **shares** as a breakdown of total usage.

Analyzing a Directory Tree LAB 4A

14. Test, and you should see output like the following. This is the final answer in **Step3**, and again the exact results will vary a bit based on the size of your source file.

    Running on the project directory ...

```
Size :      9213
Files:         7
Breakdown:
  62% build
   0% compile
   1% compile.bat
   0% run
   0% run.bat
  34% src
```

    Running on the parent directory ...

```
Size :     20369
Files:        21
Breakdown:
  20% Step1
  33% Step2
  45% Step3
```

Global Replacement and Checksums

LAB 4B

In this lab you will build two additional file-processing applications. The first will perform a global replacement of all double-quote characters with the XML/HTML escape sequence `&quot;` and produce the resulting text.

The second application will derive a simple checksum from a given file, processing the file byte-by-byte and feeding each byte into a simple reducing formula.

Lab project: **Files/Step2**

Answer project(s): **Files/Step3**

Files: * to be created
src/cc/files/Replace.java *
src/cc/files/Checksum.java *

Instructions:

1. Create the new class **cc.files.Replace** and give it a **main** method.
2. Derive a path **source** from the first command-line argument, or "src/cc/files/Replace.java" as a default.
3. Derive a path **target** from the second argument or "output/Replaced.html".
4. Open a **try/catch** system against all **Exception**s, and print an error message and stack trace in the **catch** block.
5. In the **try** block, print the contents of the **source** file to standard output, by calling **Files.lines** and then **forEach**, just using **System.out.println** for each line of text.
6. Test this and see that you can print the contents of your own source file, verbatim.
7. Now, instead of printing each string, print the results of a call to **replace** on the string, so that all double-quote characters are replaced with the sequence **`&quot;`**.
8. Test again and see your replacements in use.
9. Now, let's do the same thing, but to an output file. Start by creating the directory that will hold the file identified by **target**. This means getting the parent path and calling **Files.createDirectories** on that.

Global Replacement and Checksums LAB 4B

The simple, stream-processing approach you used to print to standard output will not work as well for file output, because there's no static such as **System.out** that you can use so easily in a **forEach** method. You could define a **FileOutputStream**, or a **BufferedWriter**, etc. But because the Streams API works by deferred processing, you can't identify a local variable as the target of a **forEach** method. There are certainly ways around this, but the simplest approach will be to depart from stream processing and put code ina traditional **for** loop.

10. Start by opening a nested **try** block, defining a **PrintWriter** called **out** as a resource and initializing based on a new **FileWriter**, passing the string representation of **target** to the constructor of the writer.
11. In the **try** block, call **out.println** a number of times, producing an HTML document skeleton such as is shown below:

```
<html><head>
<title>Replace.java</title>
<style>p { font-family: monospace;
           white-space: pre;
           margin: 0; }</style>
</head><body>
</body></html>
```

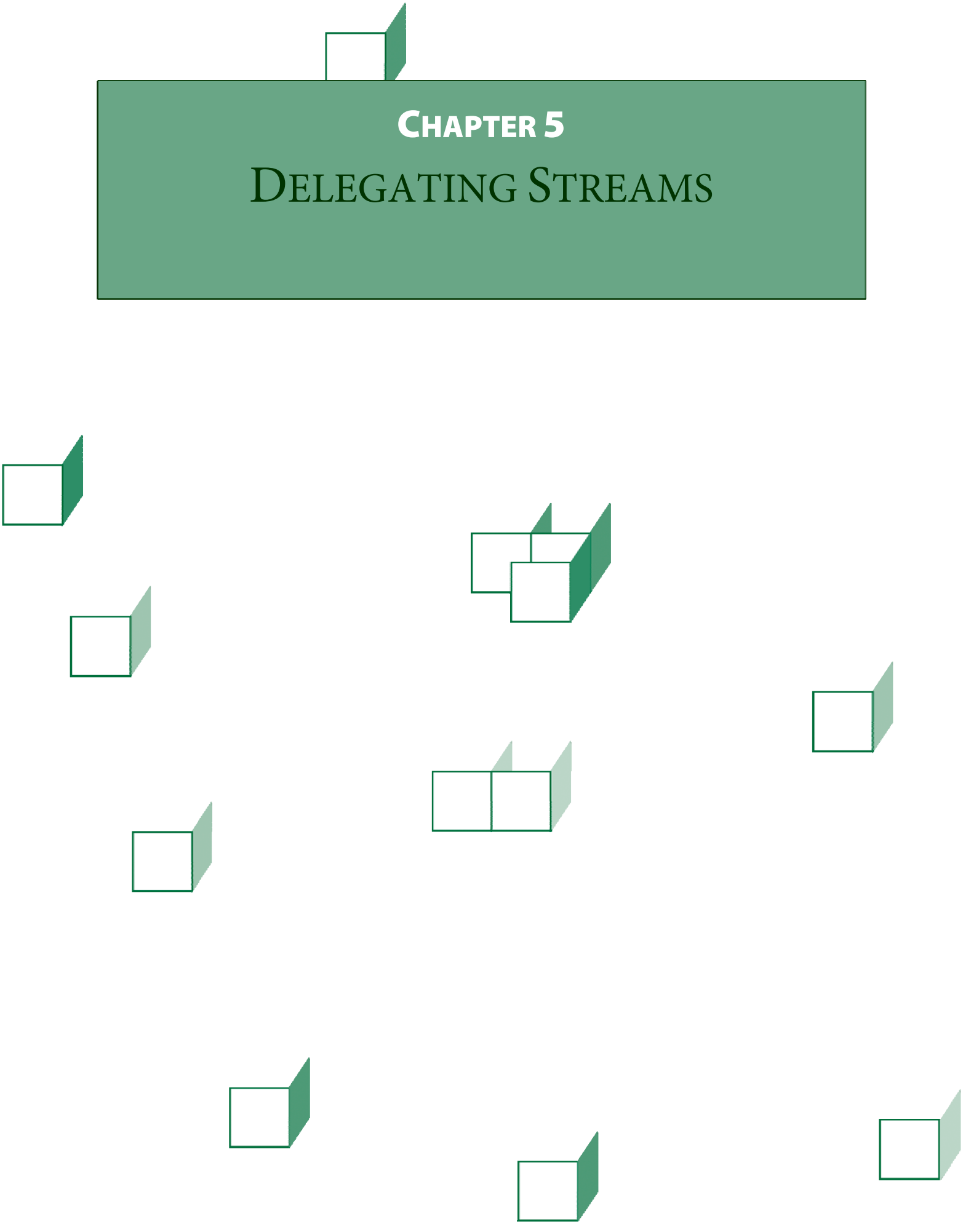
12. Before you print that final line – so, as the body content of the document – enter a loop over the lines of the file. To do this, you will again call **Files.lines**, but then use this as the basis for a call to the **iterator** method, and then use that as the **Iterable** on the right side of the colon in your loop ... like this:

```
for (String line :
    (Iterable<String>) Files.lines (source)::iterator)
```

13. In the loop, just **out.println** a ** ** tag with the text line as its content – but call **replace** on the line, just as you did earlier for console output, to replace all quotation marks.
14. Run your application again. You should now see a new file **Replaced.html** in the **output** folder (which will be created the first time you run the application); and if you open the file in a browser you should see that it shows your source code in a fixed-width font and with quotes represented correctly.

Global Replacement and Checksums LAB 4B

15. Now create the class **cc.files.Checksum**, with a **main** method.
16. Set a string **source** to the first command-line argument or a default value of "src/cc/files/Checksum.java".
17. **try** the following code, printing an error message and a stack trace in case of any **Exception**.
18. Call **Files.lines** on your **source** filename.
19. On that, call **flatMapToInt**, passing a lambda expression that returns the results of a call to **chars** on the given string – as shown on the page previous to the announcement page for this lab.
20. On that, call **reduce**, passing another lambda that simply adds the two arguments and casts the result to **byte**. This will have the effect of keeping a running total of all the byte values, modulus 256.
21. On that, call **getAsInt** to retrieve the final checksum. Print this value to standard output.
22. Test your application and see that you can derive a deterministic checksum for the **source** file. Try adding a comment to the source file and running again, and see the checksum change.



Chapter 5

Delegating Streams

OBJECTIVES

After completing "Delegating Streams," you will be able to:

- Use data streams to read and to write primitive values and strings in a compact binary format.
- Use buffering to improve streaming performance.
- Use push-back parsing to control flow between objects responsible for different sections of content in an input stream.
- Use streams to pipe information between threads.
- Use byte-array streams to connect streams and streaming logic to in-memory byte arrays.
- Use string readers and writers to connect streams and streaming logic to ordinary Strings.

Buffering

- **For most media that can support stream I/O, read and write operations are relatively expensive.**
  - That is, relative to reading a value out of the JVM's memory/address space, getting a byte from a file as modeled by the operating system is quite slow.
  - The same can be said of a network connection.
  - The saving grace is that the cost is mostly in getting there and back, so to speak; getting many bytes in a pass is not much slower than getting one.
- **This leads to the technique of buffering, by which a process reads or writes a chunk of bytes all at once.**
  - An array of bytes in memory acts as a **buffer** between the streaming code and the physical medium.
  - Then, byte-by-byte operations are very fast.
- **Four classes support buffering – they all chain to other classes, naturally:**
  - **BufferedInputStream**
  - **BufferedOutputStream**
  - **BufferedReader**
  - **BufferedWriter**

Buffering File I/O

DEMO

- **In Buffering/Step1, we'll observe the relatively poor performance of un-buffered file I/O, and then "tune up" the code by adding buffering and see the improvement at runtime.**

1. Review the code in **src/cc/files/OutAndBack.java**, which writes a file of a fixed length (1,000,000 bytes), closes it, and reads it back in. It clocks its own performance for each operation:

```
final int LENGTH = 1000000;
long start;
...
start = System.currentTimeMillis ();
System.out.print ("Writing file ... ");
try (OutputStream out =
      new FileOutputStream ("test.dat"); )
{
  for (int i = 0; i < LENGTH; ++i)
    out.write (i);
}
System.out.println ("Done in " +
  (System.currentTimeMillis () - start) +
    " milliseconds.");

start = System.currentTimeMillis ();
System.out.print ("Reading file ... ");
try (InputStream in =
      new FileInputStream ("test.dat"); )
{
  for (int i = 0; i < LENGTH; ++i)
    in.read ();
}
System.out.println ("Done in " +
  (System.currentTimeMillis () - start) +
    " milliseconds.");
```

Buffering File I/O

DEMO

2. Run the application – it takes a few seconds each way:

```
Writing file ... Done in 3265 milliseconds.
Reading file ... Done in 2353 milliseconds.
```

3. Copy the code that writes and reads the file (marked between the two lines on the previous page) and clone it below the original code.

4. Add a **BufferedOutputStream** to the output chain:

```
System.out.print
  ("Writing file with 4k buffer ... ");
try (OutputStream out = new BufferedOutputStream
      (new FileOutputStream ("test.dat"), 4096); )
```

5. Add a **BufferedInputStream** to the input chain:

```
System.out.print
  ("Reading file with 4k buffer ... ");
try (InputStream in = new BufferedInputStream
      (new FileInputStream ("test.dat"), 4096); )
```

6. Test again, and the performance advantage is obvious:

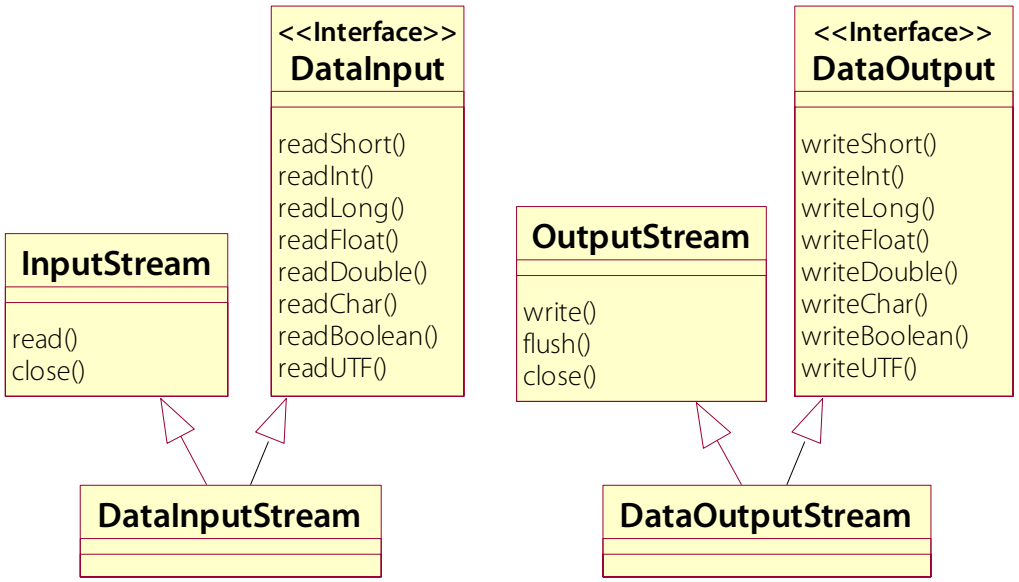
```
Writing file ... Done in 3284 milliseconds.
Reading file ... Done in 2434 milliseconds.
Writing file with 4k buffer
             ... Done in 0 milliseconds.
Reading file with 4k buffer
             ... Done in 0 milliseconds.
```

Push-Back Parsing

- **Two classes offer a simple push-back feature for reading streams:**
  - **PushbackInputStream**
  - **PushbackReader**
- **Each offers unread methods that put some number of bytes back to the stream, logically speaking.**
  - This implies some level of buffering, although it is not done so much for performance as for code simplicity.
  - For instance a parser may be decomposed into components that are variously responsible for reading keywords, operators, braces, etc.
  - The keyword component might read bytes until it hits a delimiter.
  - It then knows that its job is done, but it needs to make sure that the delimiter will be consumed by some other component.
  - So it calls **unread** on the stream to put that character back.

Data Formatting

- The **DataInput** and **DataOutput** interfaces define methods for reading or writing basic data types in a portable, efficient binary format.
- Two stream classes implement these interfaces: **DataInputStream** and **DataOutputStream**.
  - ???Each of these is a filtering stream as well, which means that they can be chained to other streams.



Binary Representations

- **Common uses of data streams are:**
  - Chained to file-based stream to manage compact file formats
  - Chained to a network stream to send and receive data efficiently over a network – in many enterprise-level specifications this is called **marshaling** object data
- **Note that there is a dilemma hidden in here: how should the data be formatted?**
  - For instance, when a **DataOutputStream** writes an integer, what byte ordering should it use?
  - There are different conventions used on different platforms and in other programming languages.
  - So there is a tension between **portability**, which we get if Java uses a consistent ordering of its own, and wanting the most natural byte ordering for a specific platform.
  - It is far more important that a file written by Java code on one platform be read by Java code (perhaps the very same code!) running on another platform – anything less would be an unacceptable compromise of Java's portability.
  - Thus so the byte ordering is the same for all platforms; as it happens the choice was to use "network ordering," which represents numbers with most significant bytes first.
  - This is usually no problem, but if you are sharing such a formatted file with code written in C, for instance, beware that standard stream routines there may assume the opposite ordering, giving you unwanted results.

The DataOutput Interface

- Write formatted values using the **DataOutput** interface:

```
public interface DataOutput
{
  public void writeDouble (double);
  public void writeFloat (float);
  public void write (int);
  public void writeByte (int);
  public void writeChar (int);
  public void writeInt (int);
  public void writeShort (int);
  public void writeLong (long);
  public void writeBoolean (boolean);
  public void write (byte[]);
  public void write (byte[],int,int);
  public void writeBytes (String);
  public void writeChars (String);
  public void writeUTF (String);
}
```

  - Specific methods are provided for various primitive types.
  - To write string values, use **writeUTF**. This will encode the string in UTF-8 format, which is more efficient than the flat Unicode representation of a **String** in memory.

```
out.writeInt (5);
out.writeUTF ("Hello");
out.writeInt (length);
Iterator each = myCollection.iterator ();
while (each.hasNext ())
  out.writeInt
    (((Integer) each.next ()).intValue ());
```

The DataInput Interface

- Read formatted values using the **DataInput** interface:

```
public interface DataInput
{
  public byte readByte ();
  public char readChar ();
  public double readDouble ();
  public float readFloat ();
  public int readInt ();
  public int readUnsignedByte ();
  public int readUnsignedShort ();
  public long readLong ();
  public short readShort ();
  public boolean readBoolean ();
  public int skipBytes (int);
  public void readFully (byte[]);
  public void readFully (byte[], int, int);
  public String readLine ();
  public String readUTF ();
}
```

  - There are corresponding methods here for the **writeXXX** methods on **DataOutput**.
  - Use **skipBytes** to ignore values you know are there but don't need to read.

```
int five = in.readInt ();
String hello = in.readUTF ();
int length = in.readInt ();
int[] numbers = new int[length];
for (int n = 0; n < length; ++n)
  numbers[n] = in.readInt ();
```

String Readers and Writers

- Strings function in the same catch-all way as byte arrays do, but specifically for character content.
- Two classes provide easy integration between strings and streaming behavior:
  - **StringReader**
  - **StringWriter**
- To produce a formatted report to a string instead of a console or file, you can "print" lines into a **StringWriter**:

```
PrintWriter out = new PrintWriter
  (new StringWriter ());
produceReport (out);
StringBuffer buffer = out.getBuffer ();
String report = out.toString ();
```

- To read lines of text from a String – perhaps this was pulled from a relational database as a large object – chain a **BufferedReader** to a **StringReader**:

```
BufferedReader in = new BufferedReader
  (new StringReader (completeListing));
String line1 = in.readLine ();
```

BufferedReader

- **In addition to the basic feature of buffering input, the BufferedReader class provides a higher-level API for managing text input, somewhat on par with PrintStream and PrintWriter.**
- **Specifically, it offers the single method readLine, which will process a full line of text from the delegate reader.**
  - It returns the complete string, in one shot.
  - It will treat either the **\r\n** sequence, or just **\n**, as the line delimiter, based on the operating system – whatever is found as the value of the system property "line.separator".
  - The line separator will not appear in the returned string.
  - The method will return an **empty string** when encountered.
  - It will return **null** when it encounters an end-of-file or not-available event in the delegate reader.

Processing User Commands

EXAMPLE

- In **Cars/Step1** (a case-study application you'll start using more heavily in the following chapter), see the utility class defined in **src/cc/tools/UserInput.java**.
- This class wraps a **BufferedReader** based on **System.in**, so that it can read one line at a time from standard input:

```
public class UserInput
{
  private static BufferedReader reader =
    new BufferedReader
      (new InputStreamReader (System.in));

  public static String getString ()
  {
    try
    {
      return reader.readLine ();
    }
    catch (IOException ex) {}

    return "EXCEPTION";
  }
}
```

  - Client code could then do something like this:

```
String command = null;
while (!(command = UserInput.getString ())
          .equalsIgnoreCase ("quit"))
  processCommand (command);
```

- You'll also see code similar to this in the upcoming lab exercise, threaded into the **DNA.java** source file directly.

Persistent DNA

LAB 5

Suggested time: 30-45 minutes

In this lab you will write persistence code to save and reload records of laboratory experiments. The starter code is based on the **DNA** application, seen earlier, but the algorithms to encode and decode the RNA strings using a compressed byte array have been broken into individual, private methods. Also there is a new method for gathering new records from the user that include the DNA sequence itself, test name, date and time, and a numerical result; also a method to print the values in a record.

You will implement methods **readFromFile** and **writeToFile** that manage individual files as the persistent storage for test records.

Detailed instructions are found at the end of the chapter.

SUMMARY

- **Buffering offers significant performance improvement for many situations.**
  - It may seem strange that things like file streams are not buffered by default.
  - This is a side effect of the delegation approach used in the streams model.
  - It's a good habit to at least consider chaining a buffer onto any streams that use physical media like hard disks.
- **Data streams are generally useful for binary formatting, for flat files or especially over network connections.**
- **Push-back parsing is supported with an input stream and a reader.**
- **In-memory streaming using a byte-array stream or a string reader/writer can simplify code design, especially by allowing easy integration of existing stream code with other application logic.**

Persistent DNA

LAB 5

In this lab you will write persistence code to save and reload records of laboratory experiments. The starter code is based on the **DNA** application, seen earlier, but the algorithms to encode and decode the RNA strings using a compressed byte array have been broken into individual, private methods. Also there is a new method for gathering new records from the user that include the DNA sequence itself, test name, date and time, and a numerical result; also a method to print the values in a record.

You will implement methods **readFromFile** and **writeToFile** that manage individual files as the persistent storage for test records.

Lab project: **PersistentDNA/Step1**

Answer project(s): **PersistentDNA/Step2** (intermediate)
PersistentDNA/Step3 (final)

Files: * to be created
src/DNA.java

Instructions:

1. Open **DNA.java** and review the code there. This is based on previous versions of the application, but the code has been rearranged significantly:
   - **DNA** is now an instantiable class with fields for **codons**, **name**, **whenSequenced**, and **results**.
   - The algorithms for encoding/decoding between the **codons** string and a compressed representation in a **byte[]** have been factored into methods **encode** and **decode**.
   - The class can communicate with the user through methods **readFromUser** and **print**. **readFromUser** asks a series of questions to initialize the four fields, and then asks the user for a filename in which to store the record. It calls **writeToFile** to create the file – this method is not yet implemented.
   - **main** either calls **readFromUser** or gets a filename from the command line and calls **readFromFile** and **print** directly.
2. Start with **writeToFile**, which accepts the filename as a parameter. Create a **DataOutputStream** chained to a new **FileOutputStream** based on that **filename**. Wrap this initialization in parentheses as the start of a try-with-resources block, and do the rest of your work in that block.
3. Write the **name** string using **writeUTF**.

Persistent DNA LAB 5

4. Write the **whenSequenced** field as a **long**, by calling **getTime** on the field. (Most dates can be mapped one-to-one to long numbers, which represent the number of seconds since midnight on January 1, 1970.)
5. Write the length of the **codons** string as an integer. This will be necessary when decoding, because you're going to encode **codons** to a byte array that won't make the exact length of the original string clear by itself.
6. Now write the result of a call to **encode** as a byte array – you can do this with the **write** method defined on the base **OutputStream**.
7. Write the **results** value as a **double**.
8. Test at this point. You should be able to enter valid values at the prompts and when the application terminates you should see a file created in the lab directory.

```
Creating a new test record - April 3, 2004, 3:45 PM
Test name:
Test one
Codons:
GAACGGATTCGG
Results:
9
Filename:
TestOne.dat

dir or ls
       build
   101 build.bat
    59 build.sh
 5,681 DNA.java
       Docs
    48 run.bat
    43 run.sh
    33 TestOne.dat
```

Persistent DNA LAB 5

9. Now implement **readFromFile**. Start by opening a new **DataInputStream in** on the provided filename. Again, try with resources, so that you're certain your stream and file will be closed cleanly.
10. Read the **name** and **whenSequenced** – for the latter, just create a new **Date** and pass **in.readLong** directly to the constructor.
11. Read the length of the codon string as an integer **length**.
12. Create a new **byte[]** called **compressed** and initialize to a new array of **(length + 3) / 4** bytes.
13. Pass this array to **in.read**.
14. Call **decode**, passing **compressed** and also **length**, so that the algorithm will know when to stop when processing the last byte in the array.
15. Finally, read the **results** field as a **double**.
16. Test with "TestOne.dat" as a program argument, and you should be able to read your record back from the file:

```
Test "Test one"
  Sequenced January 17, 2005, 3:45 PM
  Codons: GAACGGATTCGG
  Results: 9.0
```

17. You can experiment with other values from here. You might try passing a file created on your machine to another student to make sure that their application can read it; there should be no problem doing this as long as you've both encoded information the same way using the streams.

    This is the intermediate answer in **Step2**.

Persistent DNA LAB 5

Optional Steps:

18. If you test creating a record and enter any characters for the codon string other than 'A', 'C', 'G', or 'T', you will get strange results. The application doesn't crash ...

```
Creating a new test record - April 3, 2004, 3:54 PM
Test name:
Test two
Codons:
attgccattgcc
Results:
5
Filename:
TestTwo.dat
```

... but when you read the value back (run with "TestTwo.dat" as the argument) it's not what you typed in:

```
Test "Test two"
  Sequenced January 17, 2005, 3:54 PM
  Codons: ACCCACCCACCC
  Results: 5.0
```

19. If you look at the encode and decode methods, you can see the problem: there is a map of the four acceptable characters, but when encoding there is no guard against failing to find a codon character in the map: **find** winds up as 4, which doesn't get encoded in the correct bit segments in the byte array. So every byte comes out as binary 01010100 (the fourth letter always winds up rotated to 256, which rotates right out of range for the byte and so is lost). Then **decode** works correctly and writes out "ACCC" for every byte in the array.

20. Ultimately, how the algorithm fails is not important! The point is that bad data like that should be caught by more defensive code: we should check that **find** is not out of range when the loop ends in **encode**. What to do in this case? Let's work with exceptions a bit more: throw an **IllegalArgumentException** with an appropriate message.

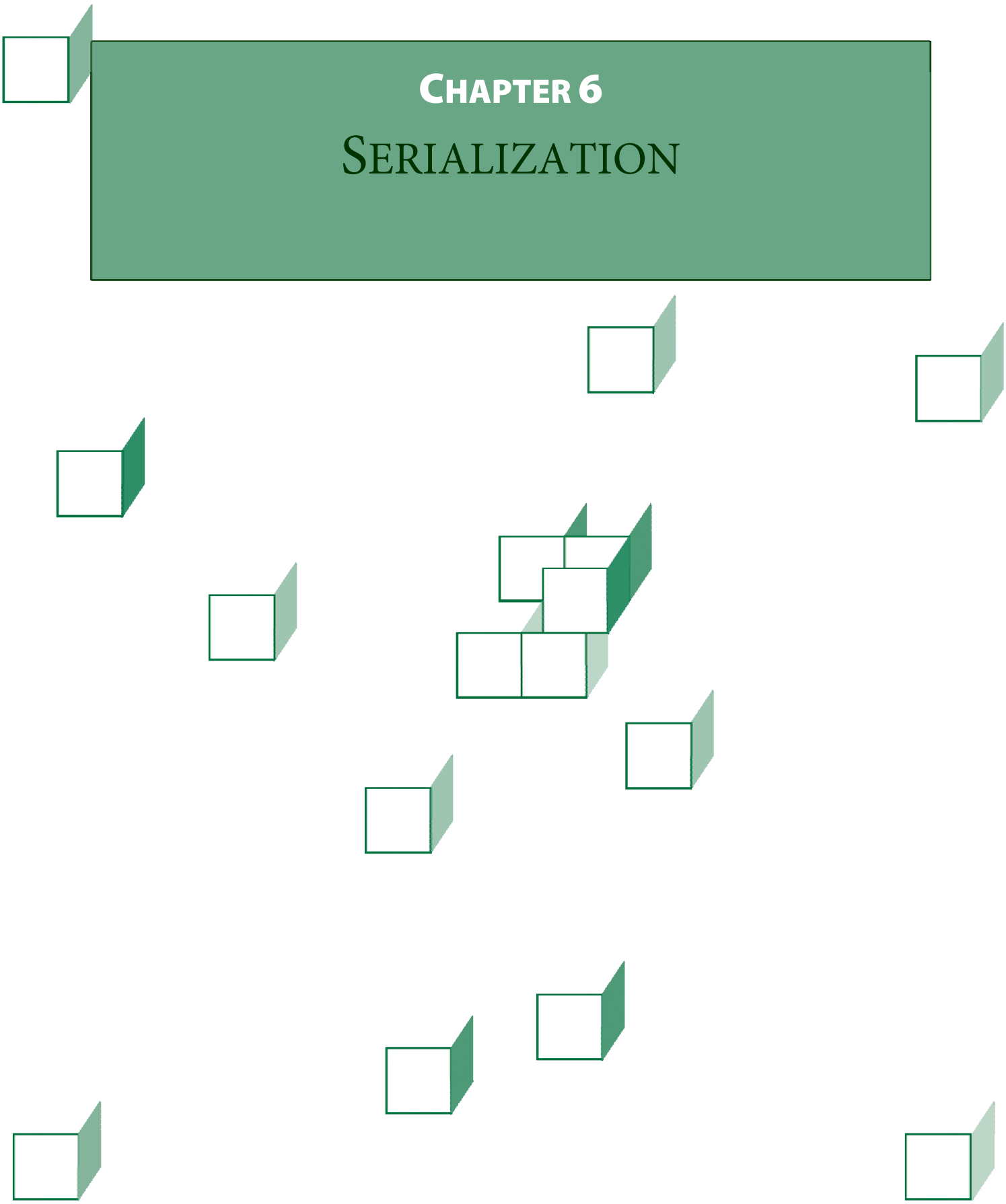
21. In **writeToFile**, after the existing **try** block, **catch** the **IllegalArgumentException** and delete the file, so as not to leave an ill-formed record hanging on the disk. You might want to print an error message to the console, as well.

22. Build and test and assure that your application handles bad data correctly. You might want to go so far as to promote lowercase letters to uppercase before looking through the **map**, just to be a little more user-friendly.

    This is the final answer in **Step3**.

Chapter 6

Serialization



OBJECTIVES

After completing "Serialization," you will be able to:

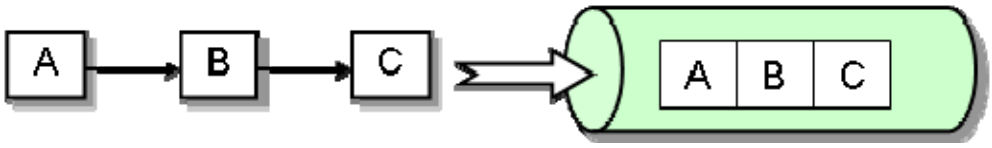
- Describe the behavior of the Java Serialization API, and understand how it can robustly and generically manage the reading and writing of object graphs of arbitrary complexity.
- Implement a system of classes to be Serializable and make appropriate choices about transient fields and re-initialization.
- Make use of an object stream to serialize such objects.
- Describe the function of Java externalization as distinct from serialization.

Object Serialization – Who Needs It?

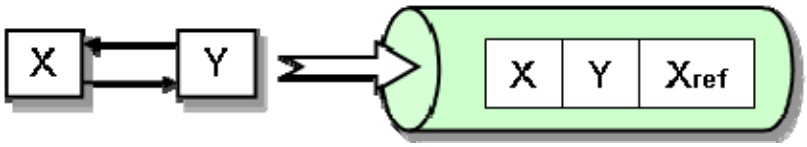
- **Consider the car dealership application: what if we wanted to make the entire data set persistent?**
  - The application would save its state before closing.
  - It would reload it on startup, and things like the sold flag on a car, or the available quantity of a part, would persist from previous sessions.
- **How might we do that?**
  - We could add code to the **Application** class to read and write at the correct times.
  - We'd need code in each persistence-aware class to read and write the state of that class' instances, probably using the data-formatting streams.
  - In short, the problem is solvable, but requires quite a lot of boilerplate code: write an int, write another, write a string ...
- **Could we generalize a solution – something that would work as well for the PersistentDNA application as for the car dealership?**

Serialization Frameworks

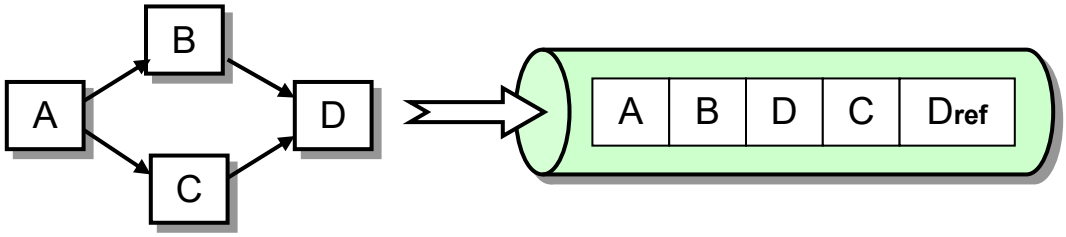
- **The problem of object serialization is a complex one.**
  - **Serialization logic** should be separate from **core logic**, because these are distinct concerns that may vary independently.
  - Thus an **outside class** has to do the work – how will it see the state of a Java class reliably, when that state is often private?
  - We'd need **rich meta-data**, so as to choose serialized formats for boolean vs. integer vs. string, etc., dynamically.
  - It would need to be able to represent complete **graphs of objects**, which means handling fields of non-primitive types.



  - What about **cycles of object references**? We'd need to avoid **infinite recursion** while assuring that the complete state of all objects is preserved.

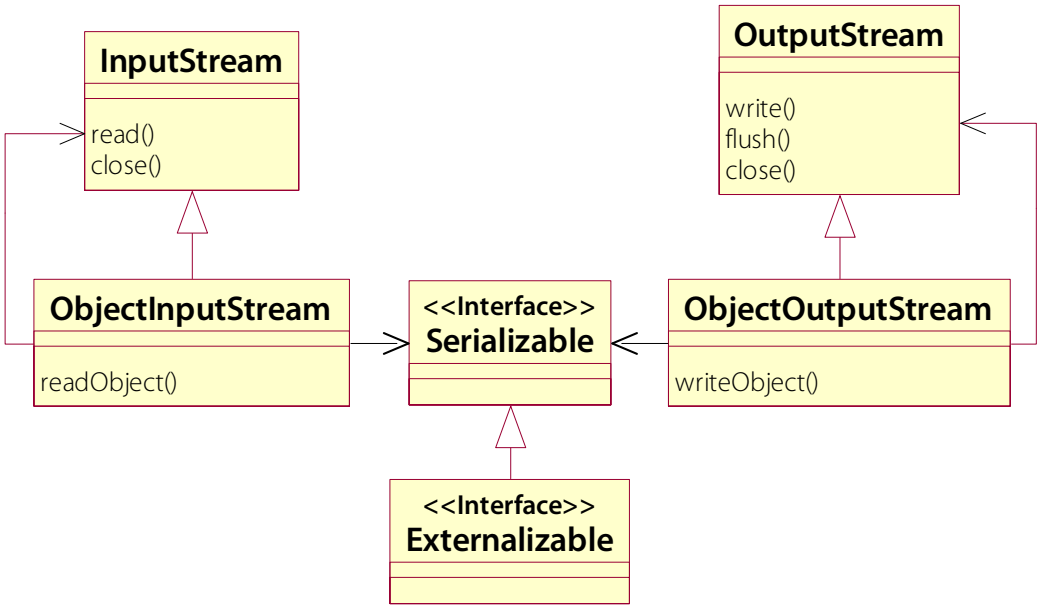


  - If an object is referenced by multiple paths, we want to avoid writing it in duplicate, because then we'd wind up reading extra objects back into memory. That is, we need to preserve **object identity** in the serialized state.



Java Serialization API

- The Java Serialization API provides all of the functionality described on the previous page.
- It is an entirely generalized solution to the problem of object serialization, and it is fully automatic.
- ???The Serialization API is implemented in the **java.io** package, in the form of:



  - A set of classes working behind the facade of the **ObjectInputStream** and **ObjectOutputStream** classes.
  - An interface **Serializable** to be implemented by classes whose instances are potential serialization targets.

Basic Use of the Serialization API

- There are two roles in the object serialization process: the object(s) and the stream.
- Implement the **Serializable** interface on one or more classes whose instances you want to write and read.
  - Implementing this interface is often as simple as declaring it implemented; **it has no methods**.
  - This is an example of a fairly common technique, in the Core API and elsewhere, called a **tagging interface**, in which the implementation of an interface is just a flag that can be recognized by the compiler.
  - By implementing **Serializable** you are formally declaring a class as a potential target for read and write operations

```
public class MyClass
    implements Serializable
{
}
```

- Create an **ObjectOutputStream** to write objects to some other allocated stream, such as on a file, then call **writeObject**, passing a reference to your object(s).

```
ObjectOutputStream out = new ObjectOutputStream
  (new FileOutputStream ("test.ser"));
out.writeObject (myObject);
```

- That's it! The same approach will work for an **ObjectInputStream**, with the additional wrinkle that you must catch the **ClassNotFoundException**.

Capabilities of the Object Streams

- **When you take the above steps, you plug your object and your stream into different parts of a quite complex and powerful engine.**
- **The code supporting object streams will perform the following steps in response to the read/writeObject calls:**
  - Perform **reflection** on the passed object and determine its fields and their types, and read or write them appropriately.
  - Recurse when it encounters fields which are references to other objects, calling **read/writeObject** on them.
  - Write the class name for each object written, and use that to instantiate the correct type at read time. (Hence the need to handle the **ClassNotFoundException** at read time.)
  - Encode **null** references uniquely.
  - Determine when objects encountered have already been written to the stream, and – instead of writing duplicate representations which would be misconstrued as multiple instances on read – **encode references** to the objects.
  - When reading such references, find the object in memory and use it, instead of creating a new one.
- **Thus it is possible, with proper class design and implementation of Serializable, to read or write a very complex graph of object instances with a single call to Object*Stream.read/writeObject.**

Serializable Types

- **The Serialization engine knows how to handle all the Java primitives.**
- **In the Core API, many common types are Serializable:**
  - **String** and **StringBuffer**
  - **Date** and **Calendar**
  - Boxing types – **Integer**, **Long**, **Double**, etc.
  - All the concrete types in the **Collections API** – this means that a **Vector** or **HashMap** of serializable objects can be written or read with just one call
  - Many others we've not studied in this course, including all the window and control types in the GUI libraries AWT and JFC
- **An application dictates which of its classes are serializable by explicitly implementing Serializable on them.**
  - A class that extends a **Serializable** class is automatically **Serializable** itself – this is in the nature of the Java inheritance model.

Transient Fields

- **By default, the Serialization engine will serialize all non-static fields on a class.**
  - Note that visibility is irrelevant to this process: private and public fields will be written in exactly the same way.
  - There are serious security concerns here, and in fact the Serialization API was recoded almost completely between versions 1.1 and 1.2 to properly recognize those issues.
  - Now the object stream code (properly) makes attempts to access non-public fields on a target object as **privileged** actions, subject to the security policy in play.
- **What if you want a class to be Serializable, but there are fields you don't want to be serialized?**
- **Declare the fields in question to be transient – this is a Java keyword defined for this precise purpose.**
  - This will exempt those fields only from the work of the Serialization engine.
  - Mark fields as **transient** when they capture only information about the current session, such as the last known mouse coordinates in a drawing application.
  - Also, use **transient** to avoid redundant storage, as when one field can be calculated from another.
  - Use **transient** to avoid recursion through references to objects which perhaps are not **Serializable**, or are not managed by this class and should be left alone.

Serializing Statistics

EXAMPLE

- In **Stats/Step1** is a new, **Serializable** version of the statistics class that holds a **Vector** of test scores.

```
public class Stats
  implements Serializable
{
  ...
  private static final String FILENAME =
    "Stats.ser";
  private Vector scores = new Vector ();
  ...
}
```

  - It offers a command-line interface that will add a score at a time to the collection – or initialize a default data set.

```
if (args.length != 0)
  try (ObjectInputStream in = new ObjectInputStream
    (new FileInputStream (FILENAME)); )
  {
    stats = (Stats) in.readObject ();
    stats.addScore (Integer.parseInt (args[0]));
  }
else
{
  stats = new Stats ();
  stats.addScore (100); ...
}
...
try (ObjectOutputStream out = new ObjectOutputStream
  (new FileOutputStream (FILENAME)); )
{
  out.writeObject (stats);
}
```

Serializing Statistics

EXAMPLE

- Run the application:

```
Scores: 100 62 80 93 82
Sample size: 5
Mean score: 83.4
```

- Note the new file **Stats.ser** in the example directory.
- Test again, passing "30" as a program argument to add a score to the data set:

```
Scores: 100 62 80 93 82 30
Sample size: 1
Mean score: 30.0
```

- … and again, this time with "50" as the new score:

```
Scores: 100 62 80 93 82 30 50
Sample size: 1
Mean score: 50.0
```

Serializing Statistics

EXAMPLE

- **Good news and bad news here!**
  - The list of scores is clearly persistent – no problem there.
  - The sample size and mean score are wrong after the first run.
  - These are each fields on the class – **sampleSize** and **mean** – which are updated with each new added score – a technique known as **eager evaluation**.
  - They are both marked **transient** because they can be derived from other data on the class.
  - This is fine, but so far nothing is being done to restore their values when the **Stats** object is de-serialized.

readObject

- **Some transient fields really do not require re-initialization when the object's state is read from the stream, for instance last mouse coordinates.**
- **Often, however, a transient field must be reinitialized somehow for the reconstituted object to be viable.**
- **When an object is instantiated by the object input stream and read from the stream, the constructor for the class is not called.**
  - This highly counter-intuitive omission can get you in all sorts of trouble – remember that when you don't use **new**, the constructor won't be invoked.
  - For a **Serializable** class with **transient** fields, it is important to assure that there is a common method for initializing the **transient**s; call this method from the constructor for objects created with **new**.
  - You must also implement a well-known method signature on your class as an entry point for completing an object's re-initialization.
  - The object input stream uses reflection to find it, and if it is found it will be called.

```
private void readObject (ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

  - There is a corresponding method **writeObject** which can be implemented as well, though usually for different reasons.

Implementing readObject

- **Firstly, be sure to get the signature right!**
  - There is no interface to implement; the method is found by reflection or not at all.
  - Thus there is no way for the compiler to warn you if you haven't gotten it right.
- **readObject, if found, is called in lieu of performing reflection on the object and reading its fields.**
  - If you implement this method, it is almost always the best thing to call **defaultReadObject** on the provided **ObjectInputStream** object.
  - If you do not, you are responsible for replacing the default behavior with whatever is appropriate for your object.

Serializing Statistics

EXAMPLE

- In **Stats/Step2** the **Stats** application has been fixed to persist everything correctly.
  - There is a new method **readObject** which assures that **sampleSize** and **mean** are kept up to date with the recently-read **scores** collection.

```
private void readObject (ObjectInputStream in)
    throws IOException, ClassNotFoundException
{
  in.defaultReadObject ();

  sampleSize = scores.size ();
  long sum = 0L;
  Iterator eachScore = scores.iterator ();
  while (eachScore.hasNext ())
    sum += ((Integer) eachScore.next ())
      .intValue ();

  mean = ((double) sum) / sampleSize;
}
```

- Test this version out, and you'll see better behavior:

```
Scores: 100 62 80 93 82
Sample size: 5
Mean score: 83.4
```

  - Running a second time with an additional score of "30":

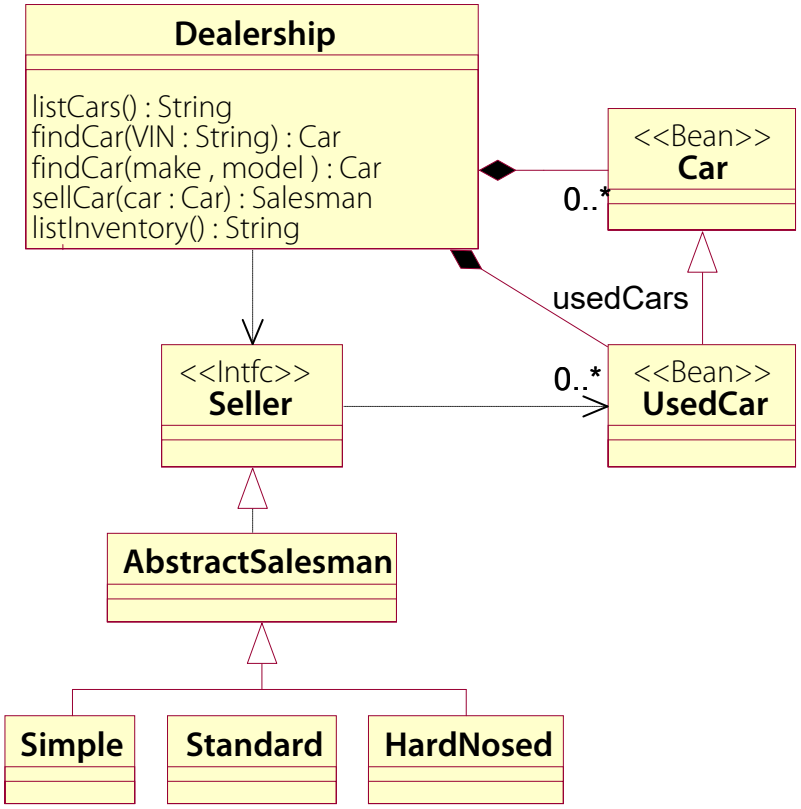
```
Scores: 100 62 80 93 82 30
Sample size: 6
Mean score: 74.5
```

Externalizable Interface

- A subinterface of **Serializable** is **Externalizable.**
- **When an object implements Externalizable, it can still participate in a serialization process, with a few important differences in behavior.**
  - The **Externalizable** interface has methods **read/writeExternal**, which (of course) must be implemented in the class code.
  - The class name is written and read, but after that the entire persistent form of the object is up to the code in the **Externalizable** implementation.
  - There is no implicit management of supertype fields, which in ordinary serialization there is.
  - There is no implicit traversal of object references, nor any recursion through those references.
- **Externalizable can be useful when you need complete control over the persistent expression of object state.**
  - Perhaps you need to implement a part of a resulting file to match a pre-existing specified format.
  - You may wish to encrypt pieces of an object's state.
  - These are all features that can conceivably be managed through **Serializable** and implementations of **read/writeObject**, but if this is being done liberally it makes sense to declare that by implementing **Externalizable** and taking more complete control of the process.

The Car-Dealership Case Study

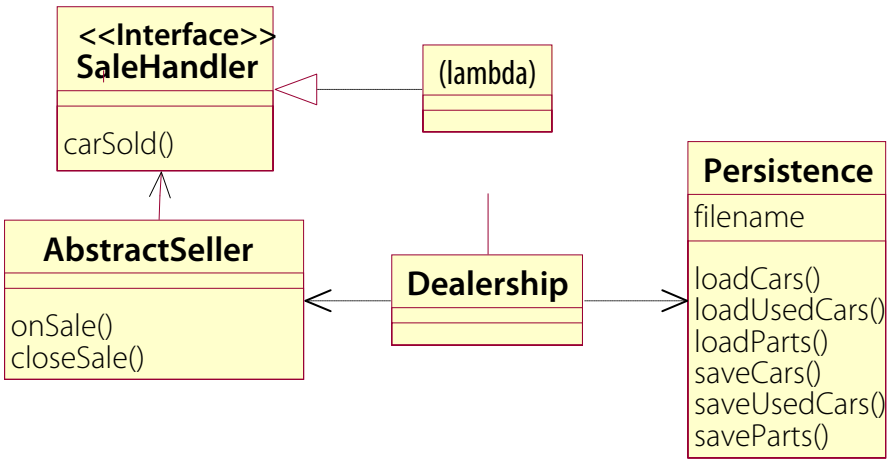
- We'll get our first look in an upcoming lab exercise at a case-study application, carried over from our intermediate Java course: the Cars application manages an inventory of new and used cars, plus a parts department, for a car dealership.



- The **Application** class provides a command-line user interface to the **Dealership** that lets the user
  - **List** cars currently on the lot
  - **Find** and price individual cars, by VIN (vehicle identification number) or make and model
  - Attempt to **buy** a car by entering into price negotiations with a selected **S**eller

The Car-Dealership Case Study

- **The starter version of the application uses hard-coded data – lists of cars, used cars, and parts.**
  - A **Persistence** class is in place but thus far only returns this static data set.
- **In the upcoming lab you will implement a simple save-and-load feature for the application, mostly by adding code to that class.**



  - **Load methods**, already called, will look for local data files, and if found will use them; otherwise it will revert to hard-coded data.
  - The **Dealership** will need to invoke new **save methods** on any change to the inventory.
  - For this purpose you'll establish a callback interface **SaleHandler**, to which various salesmen will fire notifications when they close sales. This will allow the **Dealership** to set a trigger to save state on any completed sale.

Persistent Car Dealership

LAB 6

Suggested time: 45 minutes

In this lab you will add persistence to the car dealership application. You will add code to the **Persistence** class to save and load files, using the Serialization API to format the data. You will define the **SaleHandler** interface, add notification code to the **AbstractS**eller and trigger notifications in each of the concrete subclasses, and handle these notifications in the **Dealership** by calling the appropriate **saveXXX** method.

Various classes in the application already observe the **sold** flag on a **Car**, and once you have persistence in place you'll see sold cars flagged as such in listings, and as it should the application will refuse to sell an already-sold car.

Detailed instructions are found at the end of the chapter.

Criticism of Java Serialization

- **The Serialization API – or the ObjectInput/OutputStream classes, really – need to be able to read and write fields in order to deliver on the promise of saving total object state.**
- **To do this, they must break the rules of OO visibility – that is, they must be able to read and write even private fields.**
- **It's arguable whether this is really a proper OO solution.**
- **Practically, it means that the Serialization API must carry out certain privileged actions in the Reflection API (more to come on that in a later chapter) that would ordinarily be subject to security checks by the access controller.**
- **And that, in turn, means that ... wait for it ... Serializable objects are basically non-secure – at least to the extent that they no longer practice data-hiding.**
  - If an **ObjectOutputStream** can read a **Serializable** object's state, then anyone can read that object's state, whether it's marked as **private** or not.
  - It's as easy as serializing the object to a **ByteArrayOutputStream** and then getting the bytes.
- **The serialization format is also Java-specific.**
  - This may not be a big deal, and the format is published, so code in other languages could use it, if interoperability is important.
- **Consider more modern APIs for state-serialization, such as the Java API for XML Binding, or JAXB, which uses a portable format and doesn't need to run afoul of security policies.**

SUMMARY

- The Serialization API takes on a daunting task in providing a robust, generic solution to making objects and graphs of objects persistent.
- Use of the API is quite simple and straightforward, largely thanks to the power of the Reflection API.
- Careful thought must go into class design, however: where do you want serialization processes to begin, and where is it important that they stop?
- Java Serialization is not right for every persistence requirement: in particular when pre-existing file formats must be respected, it is sometimes necessary to code support for those formats entirely by hand.
- There are some knocks on the choices made in the Serialization API code base, one especially having to do with the perceived inefficiency of writing the fully-qualified class name into the stream for each instance.
- There are also some thorny issues with object versioning, but in fairness this is one of the trickiest problems in dealing with durable data, and JavaSoft can be credited with an approach to the problem that while not perfect is realistic and well-considered.

Persistent Car Dealership

LAB 6

In this lab you will add persistence to the car dealership application. You will add code to the **Persistence** class to save and load files, using the Serialization API to format the data. You will define the **SaleHandler** interface, add notification code to the **AbstractS**eller and trigger notifications in each of the concrete subclasses, and handle these notifications in the **Dealership** by calling the appropriate **saveXXX** method.

Various classes in the application already observe the **sold** flag on a **Car**, and once you have persistence in place you'll see sold cars flagged as such in listings, and as it should the application will refuse to sell an already-sold car.

Lab project: **Cars/Step1**

Answer project(s): **Cars/Step2**

Files: * to be created
src/cc/cars/Persistence.java
src/cc/cars/Dealership.java
src/cc/cars/InventoryItem.java
src/cc/cars/SaleHandler.java *
src/cc/cars/Seller.java
src/cc/cars/sales/AbstractSeller.java

Instructions:

1. Review the starter application and note that the **Dealership** won't let you try to buy a car that's already sold. You can test this by running the **Application** class a few times, and trying to buy the Kia Sonata, which even in the starter data is marked as sold. First, run with program argument "list" – or, run with no arguments and answer the prompt given interactively by the application:

```
ED9876: 2015 Toyota Prius (Silver) $28,998.99
PV9228: 2015 Subaru Outback (Green) $25,998.99
...
UI4456: 2014 Volkswagen Jetta TDI (Green) $23,899.99
WE9394: 2015 Kia Sonata (White) SOLD
XY1234: 2015 Ford F-150 (Black) $28,999.99
...
KM0962: 2004 Saab 9000 (Silver) -- USED -- $11,498.99
HL4735: 2003 Saturn Ion (Plum) -- USED -- $4,598.99
```

   Then run with "buy WE9394" ...

```
Sorry -- that one's sold.
```

2. However, if you actually buy a car, it doesn't show up as sold in the next listing. This is because the data set is re-initialized on every run.

Persistent Car Dealership LAB 6

3. Open **Persistence.java** and add a string field **filename**.
4. Add a constructor that initializes the filename based on a string parameter.
5. Add a no-argument constructor that sets a default filename of "Dealership".
6. Define a private, parameterized helper method **loadList**: let it take a string parameter **filename** (this will be based on the field **filename**, but not the same value) and a **Class<T>** object. The return type of the method will be a **List<T>**. This way your other code can load cars, used cars, and parts by the same basic logic, varying filename and type T in multiple calls.
7. Implement the method to create an **ObjectInputStream** based on a **FileInputStream** for the given **filename**, call **readObject** on it, and return that single object, downcast to a **List<T>**.

   You will need to do this in a **try** block, with the stream as a resource, and you will need to catch **IOException** as well as **ClassNotFoundException**. Write a **SEVERE**-level message to the built-in **LOG** in case of exception, and return **null**.
8. Implement a parameterized helper method **saveList** as well: this one can take a string **filename** and a **List<T>** as parameters, and return nothing.
9. Implement this method just about symmetrically to the previous one: use output streams instead of input streams, call **writeObject**, and you will only need to catch the **IOException** since you're not loading objects of unknown types as in the load method.
10. Modify the existing **loadCars** method to build a filename based on the configured **filename** and with the extension ".cars".
11. Check for the existence of a file with this name in the working directory. If this file is found, call **loadList**, passing the composed filename. If this returns a non-**null**, simply return that from the method as the loaded list of cars.

    If either the file doesn't exist or **loadList** returns **null**, proceed to build the list from hard-coded data as the method does currently.
12. Make similar changes to the **loadUsedCars** and **loadParts** methods – use file extensions ".usedCars" and ".parts".
13. Add a method **saveCars** that takes a **List<? extends Car>** parameter. This is just a pass-through to **saveList**, setting the filename to use the ".cars" extension.
14. Create similar methods **saveUsedCars** and **saveParts**.

Persistent Car Dealership LAB 6

15. Open **Dealership.java** and add code to the bottom of the constructor to call each of your three **saveXXX** methods on the **persistence** object, right after calling the **loadXXX** methods. This is just for early testing, and you'll rip this code out later in the lab.
16. Test by running the **Application** class again, passing the "list" command as the only program argument. Whoops ...

```
SEVERE: Couldn't write to Dealership.cars
java.io.NotSerializableException: cc.cars.Car
    at java.io.ObjectOutputStream.writeObject0(...)
    ...
```

    Right – the Serialization API requires that objects be **java.io.Serializable** in order to be serialized and de-serialized with object streams. **ArrayList<E>** is already **Serializable**, but your element types **Car**, **UsedCar**, and **Part** are not.

17. You could implement the interface in each of these classes, but of course it is inherited, so the simplest thing will be to make the common base type **InventoryItem** implement **Serializable**.
18. You will see new files in the working directory. These have very little in them, because of the error you've now fixed, and they will gum up your next test, so delete them:

```
Dealership.cars
Dealership.usedCars
Dealership.parts
```

19. Now, when you test, you should see the same listing as before, and these three files re-created, with no exceptions logged.
20. Run a second time and be sure that you now load these data files, in lieu of the hard-coded data. Actually, it's hard to be sure, just yet! because the data is all the same. But if you get successful runs through your three load and three save methods, you're in good shape – set breakpoints in the debugger or instrument the code to be sure.
21. Remove the calls to **saveXXX** methods in the **Dealership** constructor. You're going to put a system in place by which this object saves data whenever its state changes. Right now, the only possible state change is the sale of a car ...
22. Define a new interface **cc.cars.SaleHandler**, with a single method **carSold**. The method will take a single parameter, of type **Car**, and return nothing.
23. Add a method **onSale** to the **S**eller interface, taking a single parameter of type **SaleHandler** and returning nothing. Seller will use this to register "listeners" for the notifications that they will fire when they sell cars.

Persistent Car Dealership LAB 6

24. Implement this method in **AbstractS**eller, by setting a field of type **SaleHandler** to the given object. (Note that we only support a single listener – a so-called **unicast event model**. This is a bit limited but will be enough for this course's lab exercises.)
25. Now, in the helper method **closeSale**, add code so that, if there is a registered listener, it calls that listener's **onSale** method, passing **car**. All of the implemented seller types already call this helper method when they complete a sale.
26. It just remains to handle this event when fired. Back in **Dealership**, find the **sellCar** method, which follows a heuristic involving system properties and possibly pseudo-random selection to instantiate some type of **S**eller and return that to the **Application** for ongoing user interaction. After deriving the **s**eller and before returning the object, call **onSale** on it.

    Pass an implementation of the **SaleHandler** interface that calls either **persistence.saveUsedCars** or **persistence.saveCars**, depending on whether the given car is an **instanceof UsedCar** or not. You can implement this in a variety of ways, including a nested class, local class, anonymous class, or a lambda expression; the answer code goes with a lambda expression.
27. Test again: first list the inventory and see that no files are created, since nothing changed when you ran that command.
28. Then run the application to buy one of the cars, and negotiate with the seller such that you succeed in buying the car. (You can have some fun trying to get the seller's price down, but a simple way to go for this lab is just offer at or above the asking price, and you're on your way.)

    Now you should see either a **Dealership.cars** or **Dealership.usedCars** file in the project directory, depending on what sort of car you bought.
29. Run a list again, and see the car marked as SOLD.
30. Run again and try to buy the same car. The application should refuse to sell, because the car is already sold.

CHAPTER 7

SOCKETS

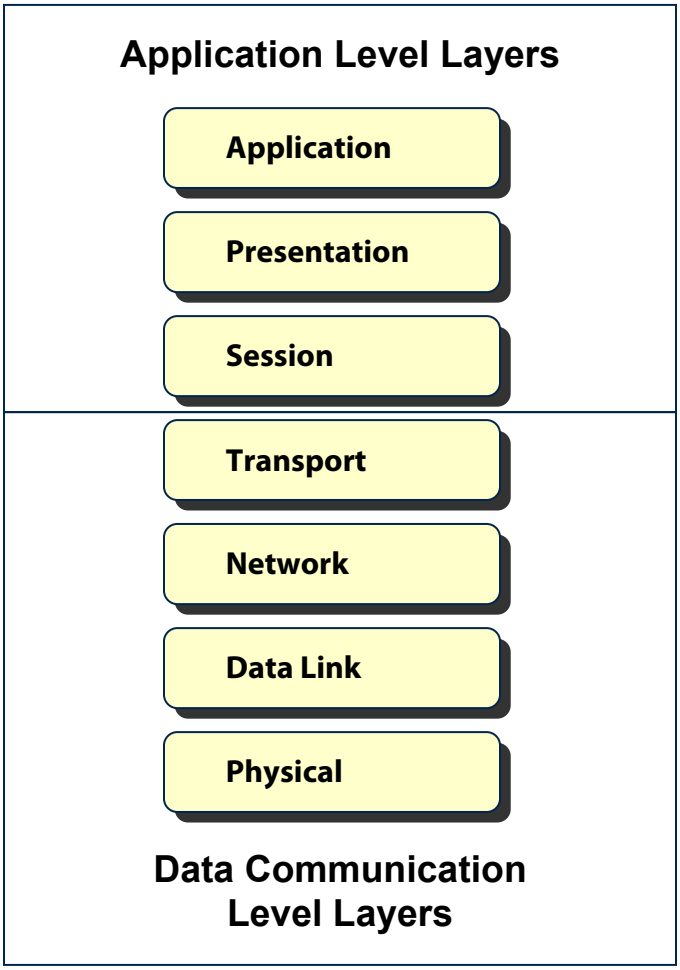
OBJECTIVES

After completing "Sockets," you will be able to:

- Identify the role of the OSI/RM, and the protocols associated with it, in the development of a communication's applications.
- Define **Sockets**.
- Use classes in the **java.net** package.
- Discuss the use of a Java client using **HttpURLConnection**.
- Explore the use of **Socket** and **ServerSocket** classes for building HTTP clients and servers.
- Build simple UDP client and server applications using **DatagramPacket** and **DatagramSocket** classes.

The OSI Reference Model

- The **Open System Interconnect Reference Model** (**OSI/RM**) is defined by the International Standards Organization (ISO).
- The **OSI stack** is a framework of logical layers that allows programmers to develop modular applications without having to worry about the low level networking nuts and bolts.
- There are seven layers:



Reference Model Layers

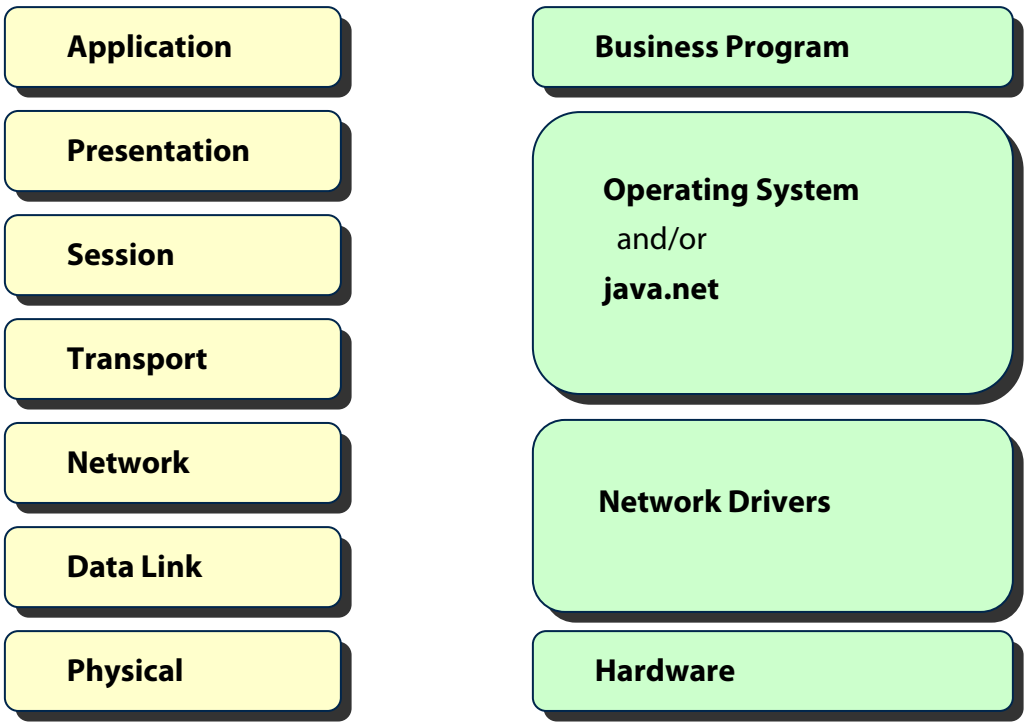
- **The Application Layer provides services to an application program (the interface to the end user) that ensure successful communication with another application program in the network.**
  - It accepts requests and passes them down to the Presentation Layer.
  - A web browser is an example of an application that uses this layer's services.
  - As programmers we're utilizing most of the classes in the **java.net** package at this layer, however these class methods operate on the lower levels.
- **The Presentation Layer provides any structure the data might need to support the user interface above and the communication layers below.**
  - It transforms data from the Application Layer into the Session Layer formats or vice versa.
  - For example, the domain name typed into the address bar is decoded into a decimal format eventually needed by the Transport Layer.
- **The Session Layer establishes and manages connections between participating applications.**
  - It takes care of creation, coordination and disconnection of the application's session.

Reference Model Layers

- **The Transport Layer provides transparent connection and disconnection from host to host.**
  - It also supports end-to-end error recovery and flow control at the host level.
  - Packet creation starts here.
- **The Network Layer forwards and routes packets.**
  - It ensures packets are sent in the right direction to the correct destination on the sending end, and receives incoming packets on the other end.
- **The Data Link Layer provides reliable data frame transmission.**
  - It supports synchronization, error control and data flow control across the physical communication path.
  - This is the hardware driver.
  - The hardware address is resolved here.
- **The Physical Layer transmits electrical impulses over a physical link.**
  - The mechanical and electrical properties needed to establish, maintain and deactivate the physical link lie here.
  - Hardware error checking (for example, CRC) is done here.

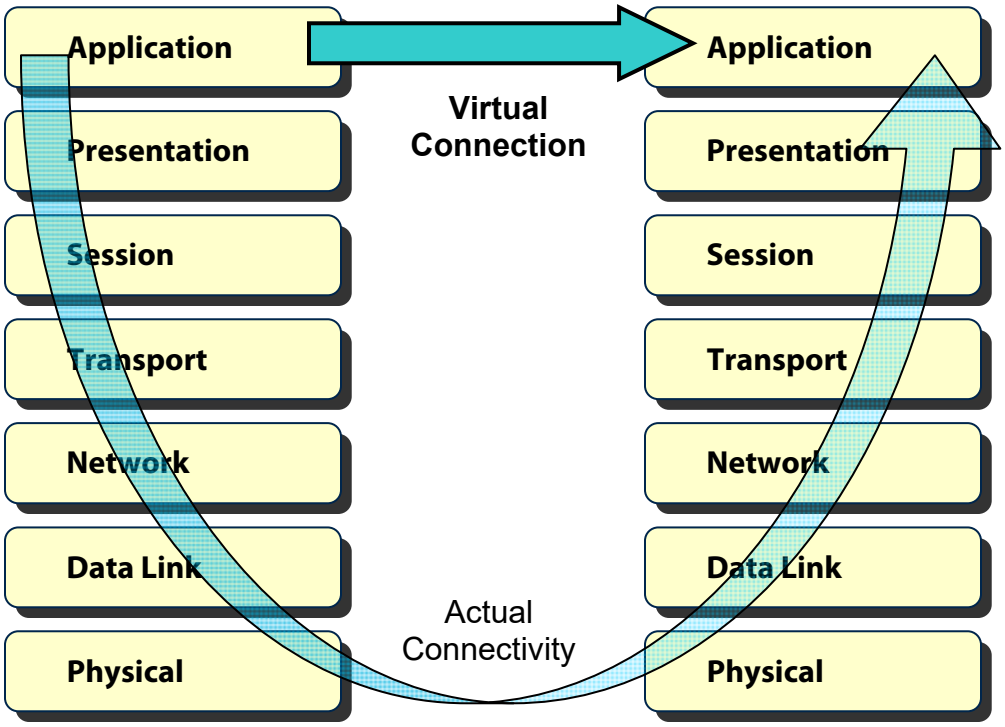
Using the OSI/RM

- **The OSI stack is a common reference model for application communication development.**
  - Each Layer has its own set of logical functions.
  - While most applications may be described in terms of all seven Layers, actual implementation will sometimes combine the functions of two or more Layers into one module.
- **As a Java programmer, you will usually be writing code in the Application Layer.**
  - For example, the functionality of the other layers could be combined in the following way, although the actual implementation could vary:



Using the OSI/RM

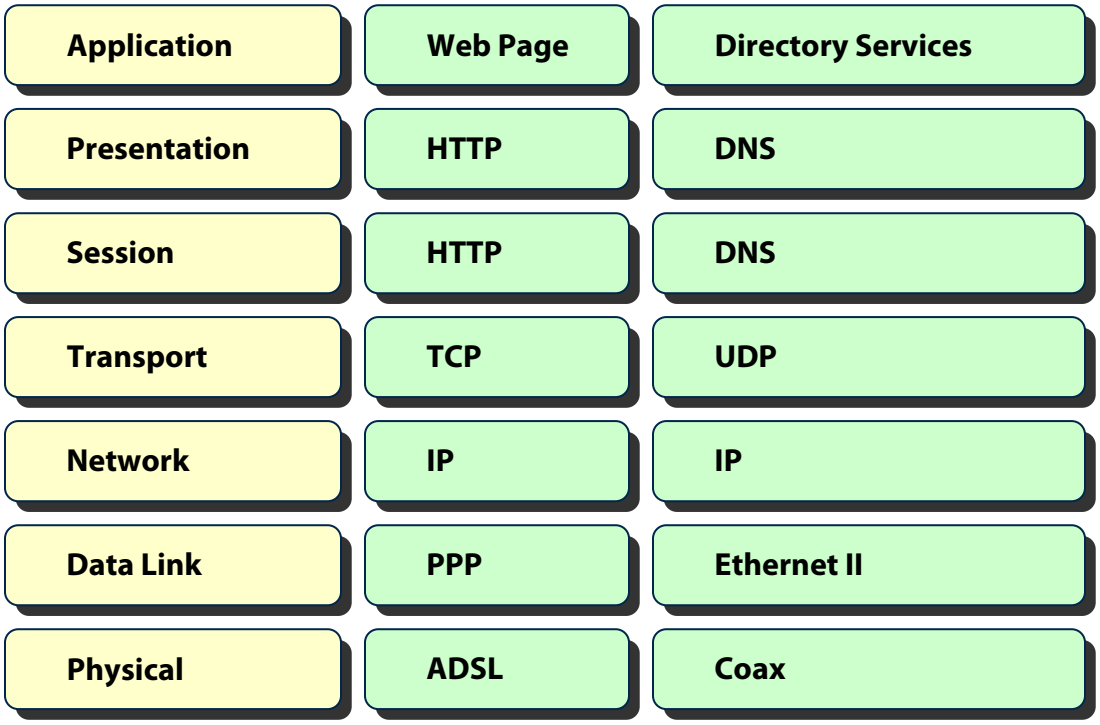
- **When a program sends a message to an application program on another machine, data flows down the OSI stack and at the other end flows up the OSI stack to the destination program.**



-
  - As the message travels down the stack, each layer wraps the message with its own information.
  - As the message travels up the stack, each layer peels off its wrapper much as you would peel a layer off an onion.
- **The information that is wrapped around the message is specific to the protocol in use at each layer.**

Protocols

- **A protocol is a set of rules for establishing and maintaining communication between two entities.**
- **For example, as human beings we have protocols for greeting one another.**
  - In Japan, people greet one another with bows; the number and depth of the bows would depend on the situation as well as the social standing of the two people greeting each other.
  - In the USA, people greet one another with a handshake.
- **Each level of the OSI stack has different protocols associated with it.**
  - The diagram shows two possible combinations using some of the Internet protocols:



Protocol Definitions

- **Hypertext Transfer Protocol (HTTP) is the set of rules for transferring files, and for linking to those files, on the World Wide Web (WWW).**
  - HTTP is often seen at the Application, Presentation or Session Layers; HTTP calls made at the Application Layer will be implemented at the Presentation or Session Layers.
  - Web browsers such as Firefox and Internet Explorer are HTTP clients.
  - Supported files include text, graphics, and multimedia.
  - The **Uniform Resource Locator (URL)** is the unique address for identifying the location of each file on the Internet.
- **Internet Protocol (IP) is the set of rules at the Network Layer that handles the sending and receiving of data.**
  - Each host (computer) is uniquely identified on the Internet by one or more IP addresses.
  - **Domain Name Service (DNS)** is a directory service used to map the domain name in a URL to an IP address.
  - IP is a connectionless protocol; IP sends each packet independently so the connection does not need to be maintained between the end points of an IP communication.

Protocol Definitions

- **Transmission Control Protocol (TCP) is a set of rules used with IP to track packets at the Transport Layer.**
  - At the source end it divides a message into packets, numbering each one and forwarding it to IP in the Network Layer.
  - At the destination end it receives each packet and waits for all of them to arrive before reassembling the message and sending it to the Session Layer.
  - Because a receipt acknowledgment for each packet is sent from the destination back to the sender, TCP is considered connection-oriented or stateful.
- **User Datagram Protocol (UDP) is a set of rules used with IP to send and receive packets at the Transport Layer.**
  - The source end does not divide packets (datagrams) so this is most often used when small amounts of data are involved.
  - However if UDP is used and if the datagram is too large, it will be broken down into smaller sizes at the Network Layer.
  - The destination end does not re-sequence the datagrams; if necessary and desirable, the application program itself is responsible for that.
  - Because no receipt acknowledgments are sent from the destination back to the sender, UDP is considered connectionless or stateless.

Java Socket Classes

- **The Java Socket API is contained in the java.net package.**
- **A socket is an endpoint for communication.**
  - A host IP address and port number define the endpoint.
  - Java encapsulates the idea of an IP address with **InetAddress**.
  - Each port number can represent either a UDP or TCP port.
- **There are connection-oriented and connectionless sockets.**
  - **TCP** sockets maintain a connection between the host and client. A web browser and web server might represent this type of connection.
  - **UDP** sockets, also know as datagram sockets, do not maintain a connection and communicate by means of datagram packets.

Java Socket Classes

- **Socket and ServerSocket represent TCP sockets.**
  - A **Socket** is a client socket; it can attempt connections to specific hosts and ports and then send and receive data.
  - A **ServerSocket** waits for connections and responds to incoming requests.
- **The web has mandated the need for specialized classes for handling HTML pages and related files.**
  - The **URL** class represents the URL and **URLConnection** represents the connection.
- **Streams are used to represent both sides of the conversation on connection.**
  - A client sends requests over an **OutputStream** provided by the connected **Socket**.
  - Responses can be read using an **InputStream**, again provided by the connected **Socket**, or by a **URLConnection**.
  - A **ServerSocket** works the same way, but from the server's perspective: an input stream to read the request and an output stream to write the response.
  - As usual for Java streams, these basic abstractions can be aggregated by higher-level ones such as **DataXXXStream**, **PrintStream**, etc.
  - The timing of writing to one stream and reading from another is defined by the protocol in use: HTTP, FTP, SMTP, etc.

Server Sockets

- A **ServerSocket** waits for requests on the network.

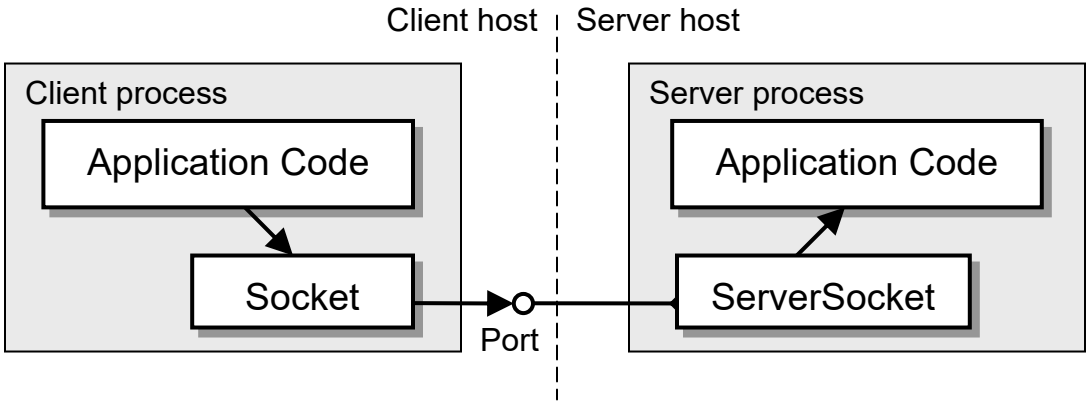
```
ServerSocket serverSocket =
  new ServerSocket(port, 1);
Socket socket = serverSocket.accept();
InputStream is = socket.getInputStream();
OutputStream os = socket.getOutputStream();
...
os.close();
is.close();
socket.close();
```

  - The **accept** method blocks waiting for an incoming request; when it comes the method returns a **Socket**.
  - Call **getInputStream** on this object to read the request.
  - Write responses using the output stream returned by **getOutputStream**.
- **The example above will respond to exactly one request, and then close.**
  - More commonly the **accept** method is the boundary condition for a loop, and often the body of the loop spawns a thread or assigns the socket to a thread from a pool.
- **At a higher level, the fields of HttpURLConnection are useful for categorizing and returning values in response headers.**

Client Sockets

- **A client Socket makes a single connection to a given host and port.**

```
Socket socket = new Socket("acme.com", port);
InputStream is = socket.getInputStream();
OutputStream os = socket.getOutputStream();
...
os.close();
is.close();
socket.close();
```



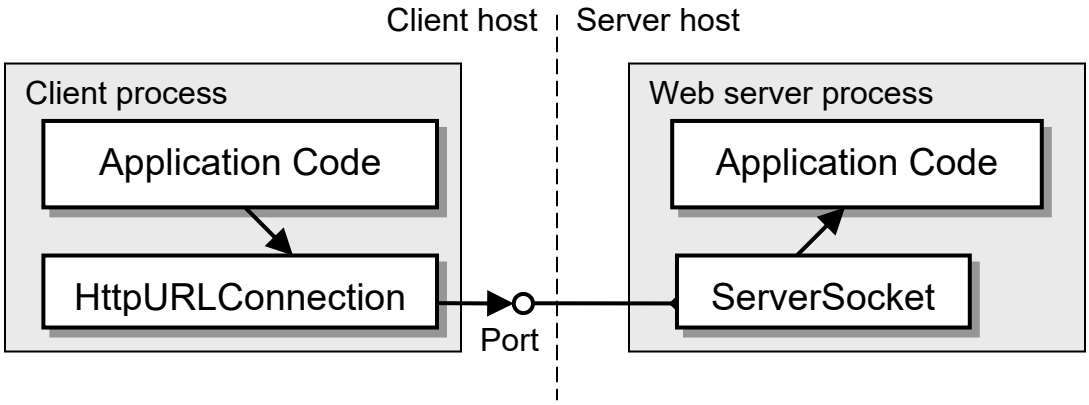
  - Again, use streams to communicate over the connection, but now the output stream is for writing the request and the input stream is for reading the response.

- **For both servers and clients, the exact timing of writing and reading is critical to proper function.**
  - It's surprisingly easy to attempt a read prematurely, and the typical outcome is a deadlock, resolved by a network timeout and an exception.

Connecting via URLs

- **URL** and **HttpURLConnection** provide broader abstractions supporting retrieval from a website:

```
URL url = new URL(urlString);
HttpURLConnection huc =
  (HttpURLConnection) url.openConnection();
huc.connect();
InputStream is = huc.getInputStream();
is.close();
huc.disconnect();
```



  - When the **URL** object is created, the URL string will be validated to see that it is well-formed.
  - The **HttpURLConnection** manages interaction with the web server, beginning with a call to **openConnection** – which creates a private **Socket**.
  - **connect** and **disconnect** methods control the actual connection lifespan.
  - Now there are also HTTP-specific methods available, to modify and read setup parameters and to manage HTTP header information more formally.

HTTP Server and Client

LAB 7A

Suggested time: 45-60 minutes.

In this lab you will complete work on an HTTP server, and test it with a local Web browser. Then you will write a client using **URL** and **HttpURLConnection**. You will be able to test it using the favorite URL of your choice, and against your own server. As a final step, you will complete a new client that uses the **Socket** class.

Detailed instructions are found at the end of the chapter.

Connected Communication

- **Though TCP supports connected communication, HTTP doesn't take full advantage of this: it is connectionless, itself.**
  - This means that each new connection accepted by an HTTP socket will receive a single request, answer it, and close.
  - Another request? A whole new socket connection.
  - Later versions of HTTP offer optimizations that allow for connections to be "kept alive."
- **For connected protocols – by which multiple request/response cycles might transpire over a single connection – the algorithm is a little different than what you used in the preceding lab.**
  - Instead of processing a single message for each call to **accept** on the server socket, you'll generally loop, handling a request at a time and sending back responses.
  - Break out of the loop when you receive a terminating command.
  - Also, be ready to catch the **SocketException** in your calls to **accept**. This will be thrown when the client terminates the connection.
  - It will also be thrown when you call **close** on the server socket – and so that is your cleanest way to shut your server down, for example in response to a shutdown command.
  - Also, be careful to call **ready** on the socket's input stream, before trying to process requests. The first request on a newly **accept**ed connection will usually be right there; but you may have to wait for later ones, and correct usage is to call **ready** before trying to **read** more bytes from the stream.

A Monitoring Station

EXAMPLE

- A case-study application will mostly develop over later chapters, in which a TCP service can answer requests about the status of certain hardware at a remote site, by a home-grown protocol.
- Because we develop much of the code for this application in later chapters on testing, test-driven development, and mocking, it would be premature in this chapter to delve into all the code.
- But, in **Station/Step5**, it might be worth a sneak peek at **src/cc/monitor/net/StationService.java**, which runs the TCP service.
- The code structure has some things in common with the **HttpServer** you just built – but does some things differently:
  - Instead of looking to a boolean flag, it runs **while (true)**, and simply catches **SocketException** as a way of recognizing that it's time to shut down – either due to client disconnect or a **close** of the server socket on our end.
  - For each **accept**ed connection, it loops until it receives a "DONE" request.
  - It calls **ready** on the input stream before reading the next request over the connection. (It could perform better here, though: the **while** loop with no body will chew up a lot of processor time, and a simple **Thread.sleep** would go a long way in this regard.)
- You will see this code in more detail later in the course.

Datagram Servers

- **DatagramSocket** represents a UDP socket and **DatagramPacket** represents a packet.

```
datagramSocket = new DatagramSocket(8000);
byte[] buffer = new byte[4096];
DatagramPacket inPacket = new
  DatagramPacket(buffer, buffer.length);
datagramSocket.receive(inPacket)page;
String inData = (new String(inPacket.getData(), 0,
  inPacket.getLength()));
...
DatagramPacket outPacket = new DatagramPacket
  (outData, outData.length,
    inPacket.getAddress(), inPacket.getPort());
datagramSocket.send(outPacket);
datagramSocket.close();
```

  - A datagram server begins by creating a **DatagramSocket** and waiting for a request via the **receive** method.
  - This method returns the received **DatagramPacket** – which is a simple byte array, to be parsed by the application.
  - The request is processed and a packet is returned to the client as a response using the **send** method.
  - Typically, the server then returns to the top of its wait loop to receive another packet.

Datagram Clients

- **The client and server code is very similar when using datagrams; only the order of events changes.**

```
datagramSocket = new DatagramSocket();
DatagramPacket outPacket = new DatagramPacket
  (message, message.length,
    InetAddress.getByName("localhost"), port);
datagramSocket.send(outPacket);
byte[] buffer = new byte[4096];
DatagramPacket inPacket = new
  DatagramPacket(buffer, buffer.length);
datagramSocket.receive(inPacket);
String inData = new String
  (inPacket.getData(), 0, inPacket.getLength());
datagramSocket.close();
```

  - The client creates a **DatagramSocket**, but does not usually specify a port to the constructor.
  - The client creates a **DatagramPacket** with the packet content and a target IP address, including host and port.
  - Then the **send** method transmits the packet, and the client's call to **receive** blocks while the server processes and returns a response packet.

- **Blocking in a client is not generally a problem, and this also means a loop is not necessary in order to poll for a reply.**
  - UDP and the datagram classes also support **one-way** messaging, rather than the request/response patterns we've seen so far, and here blocking is a non-issue.

Datagram Server and Client

LAB 7B

Suggested time: 45 minutes

In this lab you will create a client and server that use UDP datagrams. The server can accept date and time requests. It can also provide configuration information for testing purposes. You will build a datagram client and use it to test the server.

Detailed instructions are found at the end of the chapter.

SUMMARY

- Java programmers don't usually need to deal with low-level details when building a socket-based application.
- When two applications on two different computers need to communicate, one solution is Java Sockets.
- A Java Socket is a logical abstraction of the physical communication connection between two machines.
- While there is not a need to fully understand the low-level communication details, a high-level grasp of the modular framework provided OSI/RM may facilitate the effective use of Java Sockets.
- We have built a primitive HTTP server and client that illustrate the basics of communication between a web server and a browser.
- UDP packet based servers and clients might be used to allow communication between a Java program and a legacy system.

HTTP Server and Client

LAB 7A

In this lab you will complete work on an HTTP server, and test it with a local Web browser. Then you will write a client using **URL** and **HttpURLConnection**. You will be able to test it using the favorite URL of your choice, and against your own server. As a final step, you will complete a new client that uses the **Socket** class.

Lab project: **HTTP/Step1**

Answer project(s): **HTTP/Step2**

Files: * to be created
src/cc/sockets/HttpServer.java
src/cc/sockets/HttpUrlClient.java
src/cc/sockets/HttpSocketClient.java

Instructions

1. Review the starter code: several Java classes partially implemented, and a **webroot** directory holds resources for the web server you are about to build and test.
2. Open **HttpServer.java** and see that you have a few fields ready to use and a partially-implemented **main** method.
3. Define a private method **init** that takes no parameters and returns nothing.
4. In the **init** method, create the **ServerSocket** object for the **serverSocket** reference declared in the class above. Use the int variable, **port**, as part of this creation.

   You will need to **try** this code and **catch** the **IOException**. In your **catch** block, use the pre-defined **LOG** to write a **SEVERE**-level error message along with the exception object itself.
5. After creating the server socket, write an **INFO**-level message to the **LOG**, saying that the server is running on the given port.
6. The **main** method already instantiates the class. Right after this, you can now call your **init** method.
7. Run the class as a Java application, and see that you can create the server socket.

```
HttpServer running on port 80
```

   On non-Windows systems, you may need to adjust the port number so that your process will be granted permission to listen on that port – or run as superuser at port 80. For examples:

```
./run HttpServer 8000
  or
sudo ./run HttpServer
```

HTTP Server and Client LAB 7A

8. Now define a public method **service**, again taking no parameters and returning nothing. Define a **while** loop whose condition is **!shutdown**; all the rest of the method code will go inside this loop.
9. Open a **try** block, and in the resource initializer, call **serverSocket.accept** to derive the simple **Socket** as a local variable, and then get references **is** and **os** to its input and output streams. Remember that the **Socket** is created from **ServerSocket**.
10. Define two **catch** blocks: one for **IOException**, in which you should log a warning, and one for **Exception**, in which you should log a severe error and **return** from the whole method (thereby stopping the server).
11. In the **try** block itself, create **HttpRequest** and **HttpResponse** objects, passing the **is** and **os** streams, respectively, to the constructors. This would be a good time to examine **HttpRequest.java** and **HttpResponse.java**. You won't have to modify these files, but you should understand their functionality: the request type parses the request from the socket input stream, and the response stands ready to produce response content to the output stream.
12. Derive a string **path** by calling **getContextPath** on the request object.
13. Write an **INFO**-level message to the log, showing the **path**.
14. Check to see if this string equals the prepared **SHUTDOWN** URL – you might test this insensitive to case – and if so, set **shutdown** to **true**.
15. Otherwise, call **response.sendResponse**, passing the prepared **WEBROOT** location in which to find requested resources, and the **path**.
16. Now, in **main**, add a call to **service** after your existing call to **init**.
17. Run again, and see that your process stays up and running.

```
HttpServer running on port 80
```

HTTP Server and Client LAB 7A

18. Test HTTP service using a browser. In the address bar of the browser, type:

```
http://localhost
```

19. If all goes well, the browser should display:

... and you should see something like the following output in the server console:

```
Responding to: /index.html
Responding to: /images/logo.gif
```

20. Run **HttpUrlClient.java** as a Java application. This class will make a connection to the home page and show the contents as text. It also uses the **HttpURLConnection** API to read out some details of the HTTP connection and resource.

    If you set up your server on a port other than 80, pass the program argument "http://localhost:*port*/index.html" to the application.

```
<html>
  <head>
    <title>Capstone Courseware</title>
  </head>
  <body>
    <img src="./images/logo.gif"
      alt="Capstone Courseware Logo" />
    <br />
    Welcome to Capstone Courseware!
  </body>
</html>

URL information:
   Content Type: text/html
   Content Encoding: null
   Content Length: 201
   Date: Wed Dec 31 19:00:00 EST 1969
   Last Modified: Wed Dec 31 19:00:00 EST 1969
   Expiration: Wed Dec 31 19:00:00 EST 1969
   Request Method: GET
   Response Message: OK
   Response Code: 200
```

HTTP Server and Client LAB 7A

21. Now you'll build a socket-based client: open **HttpSocketClient.java** and see that it is prepared for work much as **HttpServer.java** was, with a field and partially-implemented **main** method.
22. Define a public method **retrieve** that takes a string **URL** as a parameter and returns a string, which will be the contents of the addressed resource.
23. Declare a local variable **result** and initialize to a new **StringBuilder**.
24. Open a **try/catch** system against **IOException**. In the **catch** block, just print an error message and the exception stack trace to standard output.
25. Open a resources section for the **try** block, and in the parentheses declare a **socket**, initialized to a new **Socket**, passing "localhost" and the value of **port**.
26. Still in the resources section, get a **PrintWriter** based on the socket's output stream, and get the socket's **InputStream** as well.
27. In the body of the **try** block, use the **PrintWriter** to **println**, passing a string formed as shown below, with the value of your **URL** parameter shown as a placeholder in italics:

```
GET URL HTTP/1.1
```

28. Then print a blank line.

HTTP Server and Client LAB 7A

29. Now enter a **while** loop in which you call **read** on the **InputStream** until it returns -1. In the body of the loop, cast the result of the call to **read** to type **char**, and pass that to a call to **append** on your **StringBuilder**.
30. At the bottom of the method – after the whole **try/catch** system – return the results of calling **toString** on your **StringBuilder**.
31. Add a call to **retrieve** to the bottom of your **main** method, passing the already-derived **URL** string, and print the results to the console.
32. With your server running, run the client class as a Java application. The output should be as follows – note that the header information is now read directly through the socket, not abstracted by the URL connection object, and that certain other raw stream information is reproduced in the message body.

    If you need to adjust the port, for this client you will pass first the relative URL and then the port number ... for example:

```
./run HttpSocketClient /index.html 8000

HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: text/html
--------------: ---

BF
<html>
  <head>
    <title>Object Innovations</title>
  </head>
  <body>
    <img src="./images/logo.gif"
      alt="Capstone Courseware Logo" />
    <br />
    Welcome to Capstone Courseware!
  </body>
</html>

0
```

Datagram Server and Client

LAB 7B

In this lab you will create a client and server that use UDP datagrams. The server can accept date and time requests. It can also provide configuration information for testing purposes. You will build a datagram client and use it to test the server.

Lab project: **Datagram/Step1**

Answer project(s): **Datagram/Step2**

Files: * to be created
src/cc/sockets/DatagramServer.java
src/cc/sockets/DatagramResponse.java
src/cc/sockets/DatagramClient.java

Instructions

1. Open **DatagramServer.java** and see a code structure similar to the **HttpServer** from the previous lab.
2. Define a private method **init** that takes no parameters and returns nothing.
3. In this method, initialize the field **datagramSocket** to a new **DatagramSocket**, passing the **port** field, which is initialized by the **main** method at startup.

   You will have to **try** this code and **catch SocketException**. Use the prepared **LOG** to write an error message along with the exception, and shut down the JVM. (The answer code catches the base type **IOException**.)

   You may also want to **catch BindException**, a subtype of **SocketException** specific to a failure to bind at a specific port. This is typically thrown in cases of port conflict, and so in this **catch** block you might log a different message – but it's still a fatal error, so shut down the JVM here, too.
4. In the **try** block, after initializing **datagramSocket**, write an **INFO**-level log message saying that the server has started.
5. Much of the **service** method is already implemented, but you need to get each incoming packet. After the byte array **buffer** is allocated, instead of setting **inPacket** to **null**, create a new **DatagramPacket**, passing the already-defined **buffer** and **buffer.length**.
6. Call **datagramSocket.receive**, passing **inPacket**.

   At this point, you may want to add a **catch** block just for the **IOException**, which can be thrown by the **receive** method. This will allow you to print a warning message, but avoid the shutdown that will happen for exceptions generally.

Datagram Server and Client LAB 7B

7. The next passage of code delegates to a **DatagramResponse** object to process the packet and generate a response, checks if it is a shutdown message, and if it is, sets a flag to shut down the server. (You will work in this helper class in later steps.)
8. After this, write the code to send the response to the client. Call it **outPacket** and use **outData** and **inPacket** to construct it. Send the DatagramPacket using **datagramSocket** and **outPacket**.
9. After the loop terminates, **close** the **datagramSocket**.
10. In **DatagramResponse.respond**, retrieve the **inPacket** into a **String** called **inData**. You will be using **inPacket.getData()** to do this. The rest of the method is written, as this is just ordinary string parsing, not really networking code. Un-comment the chain of if/else statements that parses and responds to different commands, and you may want to review this code briefly before moving on.
11. Now you can add calls to **init** and **service** to the bottom the **try** block in your **main** method.
12. Run the class as a Java application now. Initial output should look like this:

```
HttpServer running on port 8000
```

    Leave the server running for now.

13. Open **DatagramClient.java** and find the first TODO: comment, in the **request** method. Create the **DatagramSocket** and call it **datagramSocket**.
14. After the necessary byte array **message** is initialized with the command string, create a **DatagramPacket** called **outPacket** and construct it using **message**.
15. Finally, send the request using **datagramSocket** to send **outPacket**.

Datagram Server and Client LAB 7B

16. Run the client class. Each time you run, you will need to provide a command to send to the server, as a program argument. Sample output is shown below:

    Run with the command "date":

```
May-03-2005
```

    Run with the command "time":

```
10:53:24
```

    Run with the command "echo":

```
Host:   /127.0.0.1
Port:   1203
Length: 4
Data:   Echo
```

    Run with the command "shutdown":

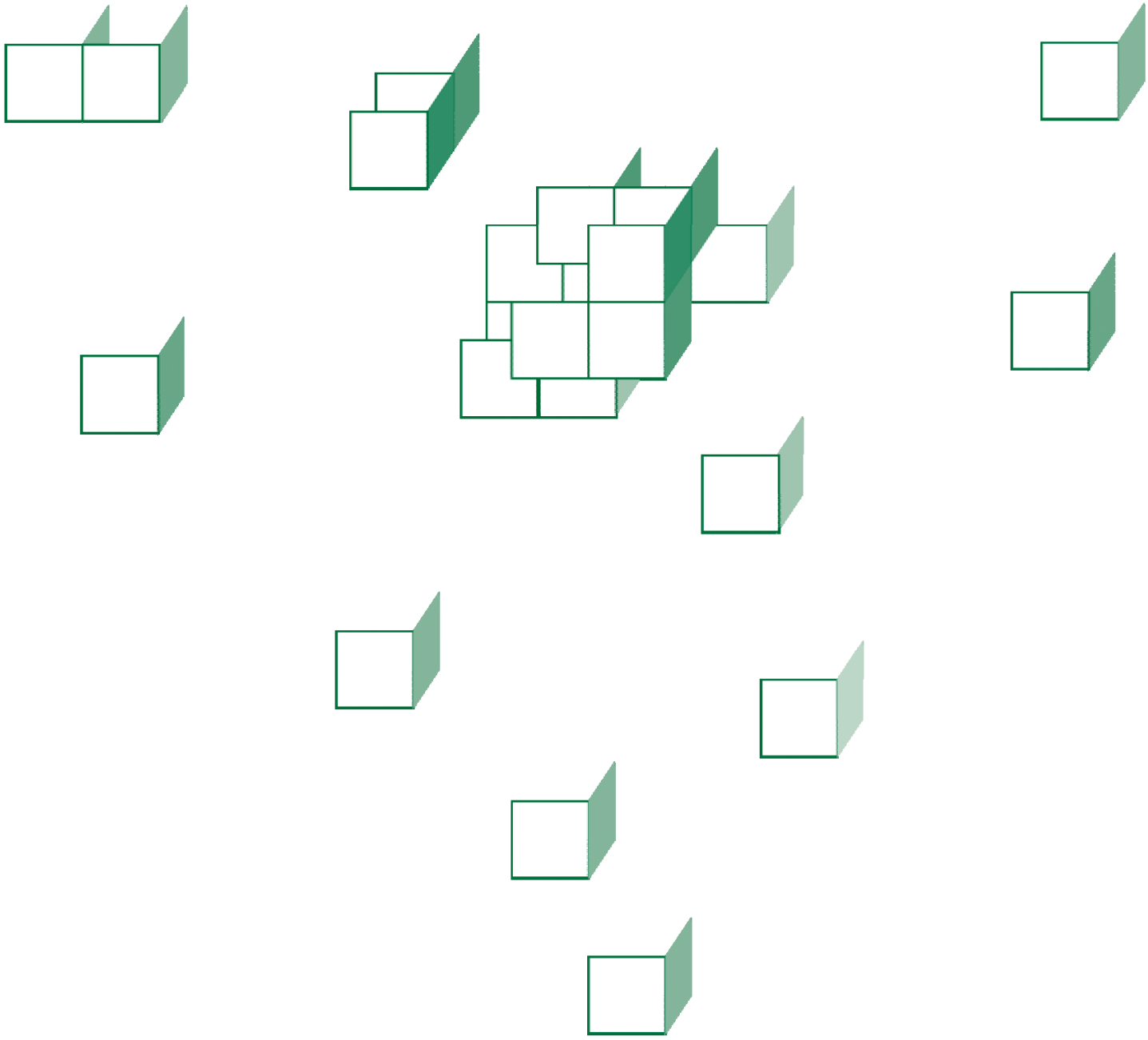
```
Server shutdown
```

    DatagramServer should show corresponding diagnostic lines in its console, and on the final command it should shut itself down and return to the command prompt.

```
DatagramServer running on port 8000
Responding with: Feb-11-2015
Responding with: 18:00:01
Responding with: Host:  /127.0.0.1
Port: 58592
Length:     4
Data: echo

Responding with: Server shutdown
```

CHAPTER 8

THREADS



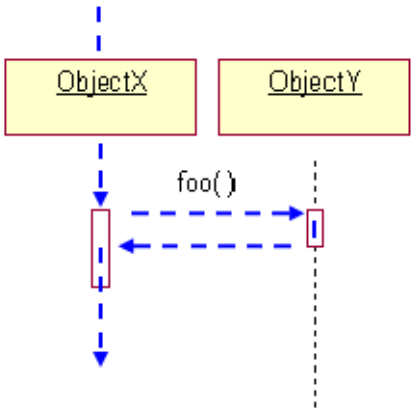
OBJECTIVES

After completing "Threads," you will be able to:

- Describe the role of threads of execution in a Java application or component.
- Use multiple threads in your Java code.
- Describe the organization of Java threads into thread-groups.
- Synchronize threads which may need to share one or more objects, so as to protect against concurrent access to and modification of those objects.
- Use **join**, **wait** and **notify** to further control the behavior of multiple threads in a JVM instance.

A Virtual CPU

- A **thread** is variously described as a **unit of execution**, or perhaps most intuitively, as a **virtual CPU**.
- The thread is the abstraction by which we understand how processing is performed by software and hardware.
- A thread might best be described by enumerating its properties.
  - A thread has a starting point and a stopping point.
  - From its start it performs work as assigned; the means by which work is assigned boils down to a combination of sequences of processor instructions in memory and the behavior of a thread scheduler.
  - A thread can only be doing one thing at a time.
  - In most modern operating systems, threads can be **suspended**, which means put in a state in which they cannot do work, and later **resumed**, or put back in a runnable state.



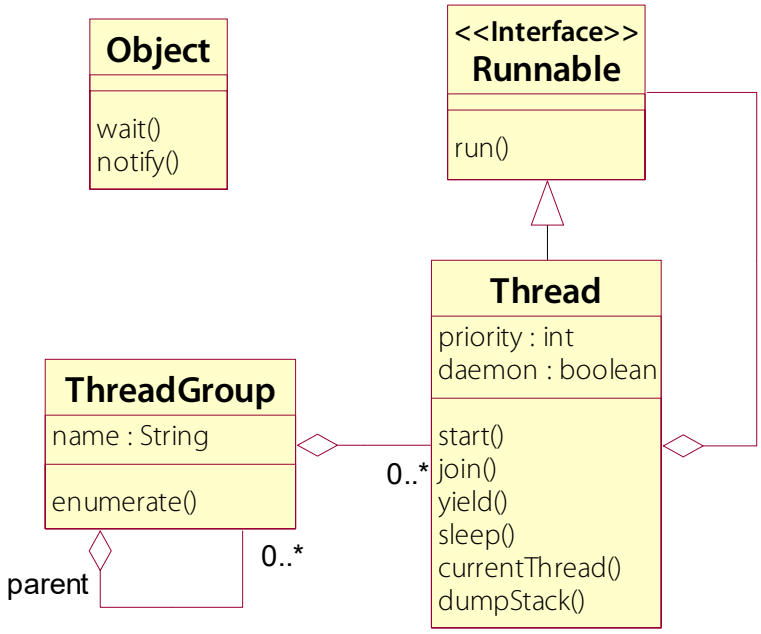
- There can be many threads in a running **process** in most operating systems.

Threads – Who Needs 'Em?

- **Everything discussed in this course thus far assumes a single thread of execution.**
- **You never work without threads, but you can work with just one.**
- **The advantages of using multiple threads for an application can be many:**
  - If there is a **user interface** that can be used to start time-consuming jobs, it is important to **avoid blocking the user** from continuing to interact with the interface; this is solved by using separate threads for the UI and the job.
  - Sometimes a more **complex task** can be optimized by using several **parallel threads**, especially when some subtasks require resources the use of which would cause a thread to block for a time.
  - In a **distributed context**, an application may well have to serve many clients, and therefore many client requests. If each request must be processed serially by a single thread **performance** will scale very poorly.

The Java Thread Model

- The Java Virtual Machine, which maps to a process in the native OS, manages threads, usually leveraging the thread model of the OS itself.
- The Core API provides a window to the JVM threads in the form of a class, **java.lang.Thread**.



  - There is a one-to-one mapping between instances of the **Thread** class and actual threads modeled by the JVM.
  - The JVM sets up a few threads for tasks that your code will never see, such as garbage collection.

Threads and Thread Groups

- The API also includes a **ThreadGroup** class, used to organize hierarchies of **Threads**.
  - A **ThreadGroup** can contain multiple **Threads** as well as other **ThreadGroup**s.
  - Certain features of thread behavior and security attach to thread groups, making administration of a multiple-threaded application easier.
- Threads themselves have lifecycles that run from being **started** through periods of **activity** and **inactivity**, and eventually they **die**.
- Threads can either be **daemon** threads or **non-daemon** threads.
  - Mark this state with the **setDaemon** method.
  - The significance of this marker is that the JVM will terminate (as a process) when all **non-daemon** threads have terminated (unless **System.exit** is called).
- A thread also has a **priority** attribute which determines the frequency with which the thread scheduler will allow the thread to run.

Thread Priority

- **The possible thread priorities vary from platform to platform, so they are captured in static fields on the Thread class:**
  - **MIN_PRIORITY** and **MAX_PRIORITY** define the boundaries.
  - **NORM_PRIORITY** defines the default thread priority.
- **When a thread of a higher priority than the current one enters a runnable state, the thread scheduler can pre-empt the running thread by suspending it and running the higher-priority thread.**
- **Threads of equal priority are treated differently on different platforms.**
  - In particular, Windows platforms use a technique called **time slicing** which allows many threads of the same priority to be allotted bursts of time to run, in rotation.
  - Non-Windows systems for the most part do not do this, with the effect that one thread can keep another of the same priority waiting.
  - There is a method **Thread.yield** which you can call to assure that same-priority threads get a chance to run during a long stretch of processing.

Identifying the Current Thread

- **Many threads may be used at various times to run the same span of code.**
- **You can get a reference to the thread on which your code is currently running using a static method on the Thread class:**

```
Thread current = Thread.currentThread ();
```

- **Threads do behave as ordinary Java objects.**
  - You can reference and act upon them.
  - You can add them to collections.
  - After a **Thread** has died, you can check other attributes to determine what happened during its lifetime.
- **The lifespan of the thread object exceeds that of the actual thread of execution.**
  - You create a thread object, and later explicitly **start** it.
  - The object lives as long as references to it exist – normal garbage-collection rules – which means it may persist well after the thread of execution has died.

Viewing JVM Threads

EXAMPLE

- **In ViewThreads there is a simple application that shows the full group/thread hierarchy of the running JVM.**
  - It starts with the current thread and walks up to group, parent group, etc.:

```
ThreadGroup group =
  Thread.currentThread ().getThreadGroup ();
while (group.getParent () != null)
  group = group.getParent ();
```

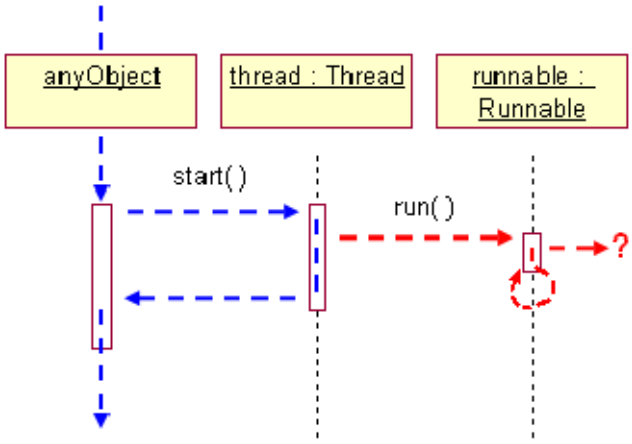
  - A recursive method **showGroup** then produces an indented tree of threads and groups, using **showThread** for the leaf nodes of the tree (the threads themselves).
- **Run the application, and you will see the following output.**

```
system
  Reference Handler
  Finalizer
  Signal Dispatcher
  Attach Listener
  main
  * main
```

  - Note that the current thread is marked with an asterisk; **showThread** accomplishes this by testing the passed thread object for identity with the current thread.

Spawning Threads

- To spawn a new thread in the JVM, you create a new instance of **Thread** and set it running.
- However, **Thread** itself doesn't know what to do when it's started; it needs a reference to a **Runnable** object, and we will look at how to develop one in a moment.




```
Thread worker = new Thread (someRunnable);
worker.start ();
```

- You can manipulate the state of a thread once you've created it (or work on any thread once you have a reference to it).
  - Call **yield** to ask the thread scheduler to let other threads run.
  - Call **sleep** on the thread to suspend it for a finite time.
- Methods that forcibly **suspend**, **resume**, and **stop** another thread have been deprecated, as they are susceptible to deadlocks.
- The better practice uses flags that are observed by the thread object but made public so that outside actors can tell the thread to pause, resume, or stop.

Monitoring Thread State

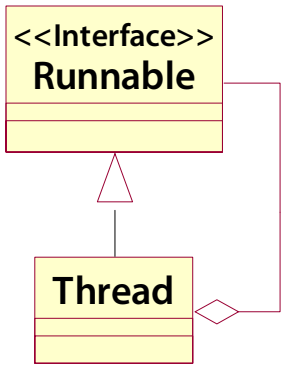
- You can check thread priority with **getPriority**.
- You can see whether or not a thread is somewhere between its **start** and **stop** points by calling **isAlive**.
  - "Alive" is not the same as "currently running."
- The more general-purpose method is **getState**, which returns a value from the enumeration **Thread.State**:

```
enum State { NEW, RUNNABLE, BLOCKED, WAITING,
             TIMED_WAITING, TERMINATED        };
```

- For diagnostic purposes, you can print a current stack trace for a thread using **dumpStack**.
  - Method-call stacks occur per thread of execution, and so a thread can produce a trace of this stack to a stream.
  - The exception stack traces we've seen here and there are examples of this. They reflect the state of the thread of execution on which the exception occurred.
  - Use **getAllStackTraces** as a way of checking the complete thread/stack status at a given moment.

Defining Thread Behavior

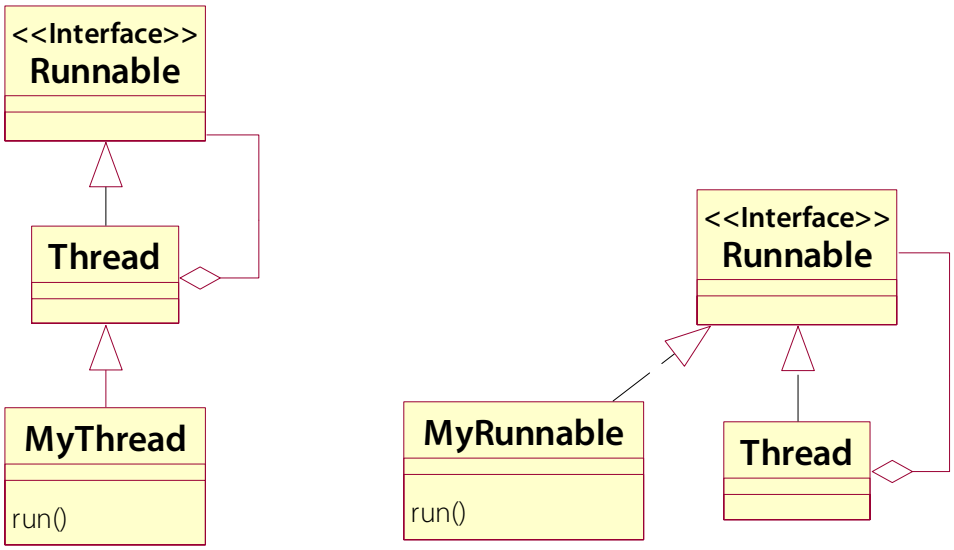
- There are two means of defining what a given thread should do when it runs.
- When **Thread.start** is called, it in turn calls the **run** method on the **Runnable** object, and the code in this method determines the work of the thread.
- When (if) **run** returns, the thread dies.
- However, the **Thread** constructor has many overloads; in particular:
  - Constructors that take a **Runnable** object, as observed.
  - Also constructors that take no **Runnable** object reference – what then?
  - The trick is that **Thread** both **references** and **implements Runnable**, or in other words **is a Runnable** object.



  - Thus a thread can call its own **run** method instead of relying on a provided object reference.

Creating Thread Classes

- **There are two approaches to defining thread behavior.**
  - You can **extend** the **Thread** class to get everything in one place; then you can create a new instance of your class and call start on it (shown below on the left).

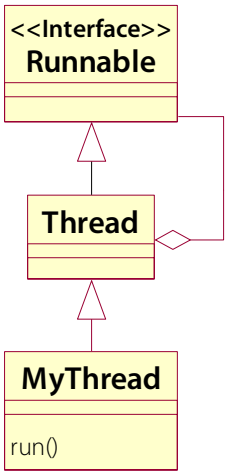


  - You can **implement** the **Runnable** interface, and pass an instance of your object to a new **Thread** instance (shown at right).
- **Neither approach builds in any limitations on the thread's behavior.**
- **However, in Java, you can only extend one other class, while you can implement as many interfaces as you like; thus in some cases extending Thread will not be an option, or not desirable.**

Subclassing Thread

- Subclass **Thread** to create a new thread class that can function on its own.
- Here is a thread class that counts to twenty million before it dies:

```
public Counter
    extends Thread
{
  public void run ()
  {
    for (int x = 1; x <= 20000000; ++x)
    {
      System.out.println ("" + x);
      if (x % 10000 == 0)
        Thread.currentThread ().yield ();
    }
  }
}
```



- Use the thread like this:

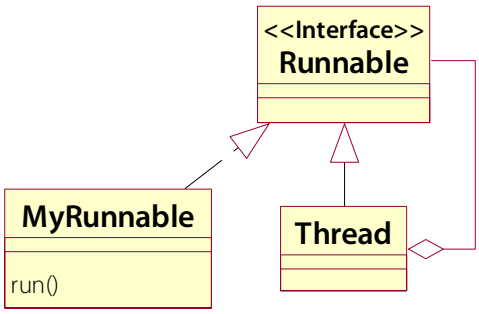
```
Thread counter = new Counter ();
counter.start ();
```

  - Do not call **run**! That would be just like calling any other method on any other object – to wit, it would not cause the desired behavior to run on a different thread.
  - Only the **start** method can do this. **start** calls **run**, but only after creating a new thread of execution in the JVM.

Implementing Runnable

- Implement **Runnable** to create a class that defines the work that a thread might do, but not the total behavior of a JVM thread.
- Here is a **Runnable** class that monitors the standard input stream for user interaction, perhaps while other threads are working on a lengthy chore:

```
public Monitor
  implements Runnable
{
  public void run ()
  {
    int key = System.in.read
();
    switch ((char) key)
    {
    case ‘x’:
    case ‘X’:
      return;
    }
  }
}
```



- Use the thread like this:

```
Runnable monitor = new Monitor ();
new Thread (monitor).start ();
```

join and sleep

- You can make the current thread block until another completes (or dies) using the **join** call on that thread.
- If an application were to kick off two parallel threads to manage a computation, and wanted to print a report of the results, it would wait for completion:

```
Thread thread1 = new ComputingThread ();
Thread thread2 = new ComputingThread ();
thread1.start ();
thread2.start ();
try
{
    thread1.join ();
    thread2.join ();
    printReport ();
}
catch (InterruptedException ex) {}
```

- Make a thread suspend for a fixed amount of time by calling **sleep** on that **Thread**:

```
try
{
    Thread.currentThread.sleep (5000);
      // ~5000 milliseconds – not exact
}
catch (InterruptedException ex) {}
```

Interrupting a File Search

DEMO

- **In Search/Step1 we'll use a second application thread to allow the user to cancel a search for a specific filename.**

1. Review the application code in **src/cc/threads/Search.java**, which parses command line arguments and calls the static method **searchForFile**.
2. This method recursively searches a given directory for a given filename, writing the full path of any matches to the console.
   - This example may not work perfectly in the IDE.
   - It uses a trick to update the console output that doesn't work so well in some console views, resulting in verbose output.
   - It will be better to code this in the IDE, and then test externally.
3. Give it a quick try:

```
compile
run Search.java
Searching for file: Search.java
From root directory: /Capstone/JavaAdv/Search/Step1
Hit ENTER to cancel ...
Found /Capstone/JavaAdv/Search/Step1/
  src/cc/threads/Search.java
```

4. Now try it on a larger tree of files, and note that there is no good way to stop the process except to crash the process with **Ctrl-C**:

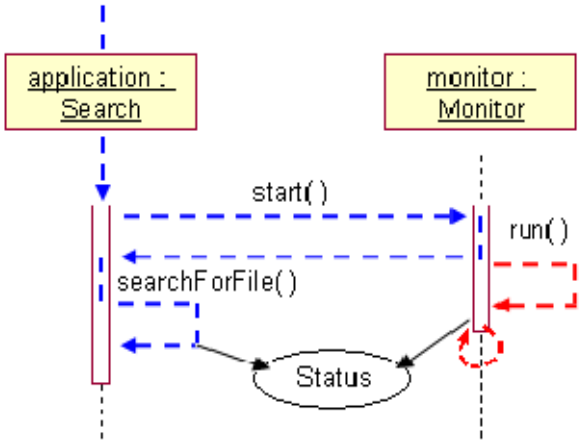
```
run Search.java /
...
```

- **How could we give the user better control?**

Interrupting a File Search

DEMO

5. We'll define a status flag that will tell the running search code whether it should continue. This can be set externally by a new thread that monitors user input.



6. First, we need to define a status flag that will tell the running search code whether it should continue. Start by creating an enumerated type **Status**:

```
public enum Status { SEARCHING, CANCELED, DONE };
```

7. Define a static field of this type:

```
private static Status status;
```

8. In **searchForFile**, start each recursion by checking this flag, and if we are either **DONE** or **CANCELED**, quit:

```
public static File searchForFile
    (String filename, File path)
  throws IOException
{
  File candidate = null;

  if (status != Status.SEARCHING)
    return null;

  System.out.print (CLEAR);
  ...
```

Interrupting a File Search

DEMO

9. In **main**, set the status to **SEARCHING** before calling **searchForFile**, and add a **finally** clause to the system that assures that the status is correctly set to **DONE**:

```
status = Status.SEARCHING;
try
{
  File found = searchForFile (filename, path);
}
catch (IOException ex)
{
  System.out.println ("IOException ... ");
}
finally
{
  status = Status.DONE;
}
```

- **This is all well and good, but how can the status change while the search is ongoing?**

10. Create a new inner class **Monitor** that **extends Thread**:

```
private static class Monitor
  extends Thread
{
  public void run ()
  {
  }
}
```

Interrupting a File Search

DEMO

11. Implement **run** to poll for keyboard input, so long as the status is still **SEARCHING**. When **Enter** is hit, set the status to **CANCELED**:

```
public void run ()
{
  while (status == Status.SEARCHING)
    try
    {
      if (System.in.available () != 0)
      {
        int key = System.in.read ();
        status = Status.CANCELED;
      }
    }
    catch (IOException ex)
    {
      System.out.println ();
      System.out.println
        (ex.getClass ().getName ());
      System.out.println ();      }
}
```

12. Create and **start** a new **Monitor** before calling **searchForFile**:

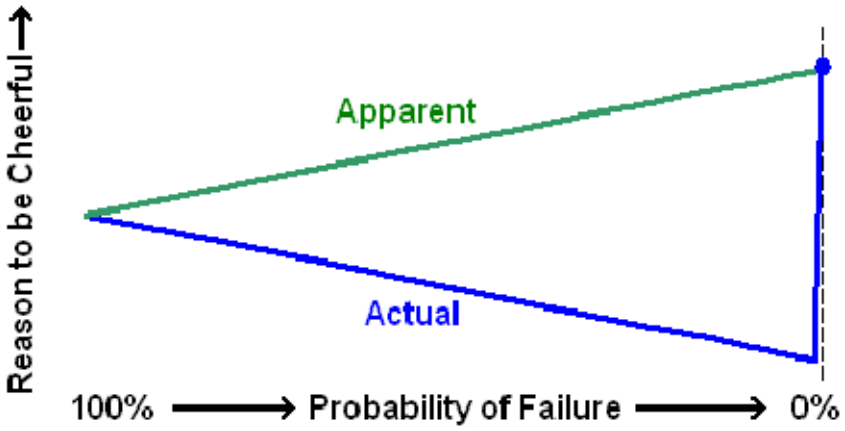
```
status = Status.SEARCHING;
new Monitor ().start ();
try
{
  File found = searchForFile (filename, path);
}
```

13. Build and test at this point. You should now be able to let the search run to completion, or to stop it by hitting ENTER.

- The completed demo is found in **Search/Step2**.

Thread Synchronization

- **It can be useful or essential to use multiple threads in a JVM to solve a programming problem.**
- **One they're there, however, multiple threads can ruin your life.**
- **If threads share data – primitive values or object references – there is a particular danger.**
  - Whatever your thread-priority choices, it is possible that any given thread might be interrupted during processing.
  - This can happen at **any** point, even in the middle of what looks in source code like a single instruction.
  - If while one thread is **interrupted** another makes a state change on an object, the original thread might be resumed only to proceed with processing based on bad assumptions. This is a nasty sort of bug known as a **race condition**.



  - What's nasty about it? It is **intermittent**, and if anything the less likely it is to occur (short of zero probability), the worse the problem it poses:

Working with Single Values

- **Most primitive values are managed atomically – that is, the operation of reading or writing their values cannot be meaningfully interrupted.**
- **Reads and writes of object references are also atomic.**
- **Still, some surprising effects can occur:**
  - There is no guarantee of atomicity over **64-bit values**. Thus a read of a **long** could produce neither the starting value nor the ending value of a concurrent write!
  - The compiler can optimize method code in some ways that throw most assumptions of sequentiality out the window. It can **reorder** certain statements, seemingly arbitrarily, for the sake of performance.
  - It can **cache** a value in a local register if it sees the value as **non-volatile**; a loop boundary is a good example.
  - It can substitute a local variable that captures the results of an expression for that expression as used later in the code – this is called **forward substitution**.
- **Believe it or not, even code like the following is susceptible to race conditions:**

```
public foo ()
{
  int a = x;
  y = 1;
}
```

```
public bar ()
{
  int b = y;
  x = 2;
}
```

  - Both `a == 2` and `b == 1` can be true in the same run!

volatile Fields

- **This seemingly impossible result is enabled first by statement reordering, and then by interruption.**
  - The reordered code might be as follows; the compiler doesn't see a problem with this since within either method the two assignments are unrelated:

```
public foo ()          public bar ()
{                      {
  y = 1;                 x = 2;
  int a = x;             int b = y;
}                      }
```

  - … and then a simple interruption of one method by the other would give us this surprising result.
- **One can avoid this and other compiler-generated surprises by declaring shared fields volatile:**

```
private volatile int x, y;
```

- **This informs the compiler that the field value may change at any time due to an external action.**
- **In response the compiler assures that**
  - It will read the value of the field "fresh" each time it is referenced in an expression.
  - It will not reorder, substitute, or otherwise optimize based on an assumption of a stable value for the field.
- **In practice this is of limited usefulness, and higher-level synchronization of code blocks makes it moot.**

Working with Arrays

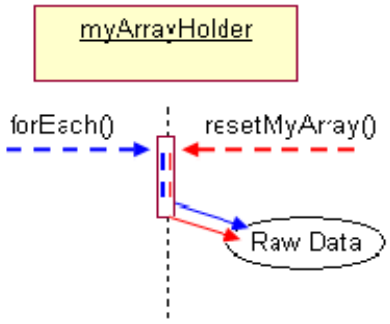
- Consider a method that iterates over the elements of an array:

```
public void forEach ()
{
  for (int i = 0; i < myArray.length; ++i)
    myArray[i].doSomething ();
}
```

- What if there is another method on the class that allows for the array field to be reset?

```
public void resetMyArray (SomeClass[] value)
{
  myArray = value;
}
```

- If one pre-empts the other, there will be trouble!



  - Thread one, for instance, might be in the midst of a call to **forEach**, maybe on the 12th array element, when interrupted.
  - Thread two, given the green light by the thread scheduler, calls **resetMyArray,** plugs in a whole new collection of only ten **SomeClass** instances, returns and is suspended.
  - Thread one resumes, attempting to **doSomething** on lucky 13, and things go very wrong.

synchronized Blocks

- The problem described above can be solved by **synchronizing** the two methods.
- Any block of code can be declared as **synchronized**, which causes the JVM to enforce a locking policy on that block of code.

```
synchronized (myArray)
{
  for (int i = 0; i < myArray.length; ++i)
    myArray[i].doSomething ();
}
```

  - In parentheses after the keyword, specify a **lock**, which is any object reference.
  - The lock acts as an identifier for a **synchronization group**, which is any number of **synchronized** blocks identifying the same lock.
  - Only one thread can be active at any moment in any of the blocks in a synchronization group: once a thread is in one of these blocks, any and all other threads will wait for that thread to leave, before one of them is allowed to enter its assigned code block.
  - The lock can be anything; especially, it doesn't need to be involved in the synchronized processing, although often it is involved.
  - **this** is a common choice of lock object, and more on that later.

synchronized Methods

- As used as a modifier on a method declaration, the **synchronized** keyword accomplishes something very similar.
- In fact, this is really just a shorthand for treating the whole method body as a synchronized block, with **this** as the lock.

```
public synchronized void forEach ()
{
  for (int i = 0; i < myArray.length; ++i)
    myArray[i].doSomething ();
}

public synchronized void resetMyArray
  (SomeClass[] value)
{
  myArray = value;
}
```

- This then is a limited technique, but still surprisingly useful, since so often the state that's being shared between threads is exactly that state encapsulated by a class.
- Note that a synchronized method is a member of the synchronization group defined by its lock – **this** – and this can be a "mixed group" of whole methods and narrower code blocks that identify **this** as their lock explicitly.

Synchronized Code

EXAMPLE

- In **Synchronization/Step1**, **src/cc/thread/RunInSynch.java** exercises the synchronization feature of the JVM.
  - The class allocates two **java.lang.Object**s, just to serve as locks.

```
public class RunInSynch
{
  private Object lock1 = new Object ();
  private Object lock2 = new Object ();
```

  - A helper method **workForThreeSeconds** will be called from various methods on various threads. It prints a given ID and traces its progress through three iterations, sleeping for a second in each:

```
  private void workForThreeSeconds (String ID)
  {
    System.out.println (ID + " running ...");
    for (int i = 1; i <= 3; ++i)
    {
      System.out.println ("  " + ID + " " + i);
      try
      {
        Thread.sleep (1000);
      }
      catch (InterruptedException ex)
      {
        System.out.println ("... strange ...");
      }
    }
    System.out.println (ID + " done.");
  }
```

Synchronized Code

EXAMPLE

- A trio of methods participate in a synchronization group based on **this** as a lock, each calling the helper method. The third method calls it once before synchronization, once during, and once after.

```
public synchronized void ownLockMethodA ()
{
  workForThreeSeconds ("ownLockMethodA()");
}

public void ownLockMethodB ()
{
  synchronized (this)
  {
    workForThreeSeconds ("ownLockMethodB()");
  }
}

public void ownLockMethodC ()
{
  workForThreeSeconds
    ("ownLockMethodC() prelude");
  synchronized (this)
  {
    workForThreeSeconds ("ownLockMethodC()");
  }
  workForThreeSeconds ("ownLockMethodC() coda");
}
```

Synchronized Code

EXAMPLE

– Down a page or two, a method **runTwoOfMyOwn** calls each of the first two methods on separate threads:

```
public void runTwoOfMyOwn ()
{
  Thread[] threads =
    {
      new Thread (this::ownLockMethodA),
      new Thread (this::ownLockMethodB)
    };
  for (Thread thread : threads)
    thread.start ();
  for (Thread thread : threads)
    try
    {
      thread.join ();
    }
    catch (InterruptedException ex)
    {
      System.out.println ("... strange ...");
    }
}
```

– Another method **runThreeOfMyOwn** follows exactly the same code structure, but it runs all three methods, each on a separate thread.

Synchronized Code

EXAMPLE

- Back up a ways in the source file, a pair of methods works similarly, over **lock1** – and a second pair does the same thing over **lock2**.

```
public void lock1MethodA ()
{
  synchronized (lock1)
  {
    workForThreeSeconds ("lock1MethodA()");
  }
}

public void lock1MethodB ()
{
  synchronized (lock1)
  {
    workForThreeSeconds ("lock1MethodB()");
  }
}
```

- Below the two- and three-thread driver methods is another, **runFourOnLocks1and2**, which kicks off a thread on each of this pair of pairs of methods.
- The **main** method runs all three of the driver methods – in sequence, since each driver method **join**s the threads it creates:

```
public static void main (String[] args)
{
  RunInSynch tester = new RunInSynch ();
  tester.runTwoOfMyOwn ();
  ...
  tester.runThreeOfMyOwn ();
  ...
  tester.runFourOnLocks1and2 ();
}
```

Synchronized Code

EXAMPLE

- **Okay! Run the application to observe the effects of the various uses of the synchronized keyword:**

```
Testing two threads ...
ownLockMethodA() running ...
  ownLockMethodA() 1
  ownLockMethodA() 2
  ownLockMethodA() 3
ownLockMethodA() done.
ownLockMethodB() running ...
  ownLockMethodB() 1
  ownLockMethodB() 2
  ownLockMethodB() 3
ownLockMethodB() done.
```

  - So each of methods A and B runs in sequence, with no overlap in time – a/k/a no concurrent execution.

Synchronized Code

EXAMPLE

```
Testing three threads ...
ownLockMethodA() running ...
  ownLockMethodA() 1
ownLockMethodC() prelude running ...
  ownLockMethodC() prelude 1
  ownLockMethodC() prelude 2
  ownLockMethodA() 2
  ownLockMethodA() 3
  ownLockMethodC() prelude 3
ownLockMethodC() prelude done.
ownLockMethodA() done.
ownLockMethodC() running ...
  ownLockMethodC() 1
  ownLockMethodC() 2
  ownLockMethodC() 3
ownLockMethodC() done.
ownLockMethodB() running ...
ownLockMethodC() coda running ...
  ownLockMethodB() 1
  ownLockMethodC() coda 1
  ownLockMethodC() coda 2
  ownLockMethodB() 2
  ownLockMethodB() 3
  ownLockMethodC() coda 3
ownLockMethodC() coda done.
ownLockMethodB() done.
```

- This output at first seems more scrambled. But methods A and B – and the **synchronized** part of C – still all run in a clean sequence.
- It's only the **prelude** and **coda** sections of method C that run on their own – since those calls are not synchronized.
- Neither overlaps with the middle of C, but that's just because they are all running on a single thread!

Synchronized Code

EXAMPLE

```
Testing four threads ...
lock1MethodA() running ...
  lock1MethodA() 1
lock2MethodA() running ...
  lock2MethodA() 1
  lock1MethodA() 2
  lock2MethodA() 2
  lock1MethodA() 3
  lock2MethodA() 3
lock1MethodA() done.
lock2MethodA() done.
lock1MethodB() running ...
lock2MethodB() running ...
  lock2MethodB() 1
  lock1MethodB() 1
  lock2MethodB() 2
  lock1MethodB() 2
  lock1MethodB() 3
  lock2MethodB() 3
lock1MethodB() done.
lock2MethodB() done.
```

- In this final test we see that, while the pairs of methods that share lock objects run in sequence, they overlap each other as pairs.
- This shows the independence of synchronization groups: if you don't share a lock object, then you are not synchronized.

wait and notify

- **For greater control over thread synchronization, you can use the wait and notify calls on an object.**
- **wait will suspend the calling thread until a notify call is made on the called object.**
- **Use wait to cause a thread to suspend until a condition is met.**
  - Always call **wait** in a loop whose boundary condition is that which must be met before processing can proceed.
  - Do this because **wait** and **notify** are at bottom just optimizations; it is possible to awaken from a wait to find the condition unmet, and in that case you just **wait** again.
  - For instance you might **wait** for a count to come to zero, and **notify** every time the count were decremented.

```
while (resourcesBusy != 0)
    wait ();
```

- **Always call wait and notify from a synchronized method or code block.**
  - The object's **monitor** is the target of the **wait/notify** calls.
  - Running in **synchronized** code acquires ownership of the monitor, which is necessary for the **wait/notify** to work.
- **In many modern Java applications, wait and notify have given way to various concurrency utilities in java.util.concurrent.**
- **We'll consider these in the following chapter.**

Suspend/Resume and Cancel

LAB 8

Suggested time: 30 minutes

In this lab you will enhance the threading code in the file-search application, to allow the user to suspend the search and then to choose to resume it or to cancel it altogether. This will require more than just a status flag and concurrent checks on status in the two threads: you will need to synchronize certain stretches of code to assure that they are waiting on each other.

Detailed instructions are found at the end of the chapter.

The Worst Thing You Can Do

- **Procedures naturally emerge in multi-threaded systems whose job it is to wait for certain conditions to be met.**
- **The best way to be notified of changing circumstances is by way of a method that lets the JVM wake you up:**
  - **wait** to be notified, and check your condition; if not satisfied **wait** a bit longer.
  - **join** one or more threads, and be awakened when they are terminated.
- **Sometimes there is no such natural mechanism – no one to notify you if you wait, for example.**
- **The worst way to monitor changing conditions is to loop, with no check on your thread's usage:**

```
while (!done);
```

  - This will chew up a whole lot of processor time!
- **If you must write a polling loop such as this, be sure to do one of two things – or both:**
  - **Lower the priority** of the polling thread.
  - **Sleep** periodically, and then check again.
- **The former is more elegant, but beware that it can wind up starving your thread, unless your higher-priority threads either yield periodically or use system resources that make them non-runnable from time to time.**

Getting Your Sleep

EXAMPLE

- The answer code to the preceding lab includes two extra lines of code that are quite important:

```
while (status == Status.SEARCHING)
  try
  {
    if (System.in.available () != 0)
      synchronized (status)
      {
        // Read user input and possibly cancel
      }
    else
      Thread.sleep (100);
  }
  catch (IOException | InterruptedException ex)
  { ... }
```

- At this stage, you probably won't notice a difference in the way the application runs.

Getting Your Sleep

EXAMPLE

- Consider **Search/Step4**, though, which does some simple timing of the search process from the **main** method.

```
long start = System.currentTimeMillis ();
status = Status.SEARCHING;
new Monitor ().start ();
try
{
  searchForFile (filename, path);
}
catch (IOException ex) { ... }
finally
{
  status = Status.DONE;
}

System.out.println ("Time: " +
  (System.currentTimeMillis () - start) + " msec");
```

- **You may want to experiment with this.**
  - Run on a larger directory tree – the **../..** path should do it – and check the search time.
  - Remove that call to **sleep** in the **Monitor.run** method, test again, and see if there is any effect.
- **In rough testing we have found that failing to sleep the monitoring thread increases search time by as much as 20%.**
  - In fact it would probably be much worse, if the main worker thread didn't already spend so much time in a non-runnable state, waiting for the file system to report.
  - It's those file-system delays that take most of the time.

A Single-Threaded Server

EXAMPLE

- Finally, we'll take a quick look at server threading, by way of alternative implementations of the HTTP server you built in the previous chapter's lab exercise.
- Both have been enhanced slightly with code that dumps the thread ID while processing each new HTTP request, to illustrate how the different approaches behave at runtime.
- The key source file for each is **src/cc/sockets/HttpServer.java**.
- In **HTTP/Step3**, the only new code is found in the **service** method, where we write a line to system output:

```
System.out.println("Responding to " + path +
  " on thread " + Thread.currentThread ().getId());
```

A Single-Threaded Server

EXAMPLE

- **Run the server, and try hitting it frequently and from many sources – if your classroom is networked, you might try having everyone make requests of the instructor's running server.**
  - For non-Windows systems, again, you may need to adjust the server's port number or 'sudo' the script – see notes in the previous chapter's lab exercise.
  - You'll see predictable results as to the thread ID:

```
HttpServer running on port 80
Responding to /index.html on thread 1
Responding to /images/Logo.gif on thread 1
Responding to /favicon.ico on thread 1
...
```

A Multi-Threaded Server

EXAMPLE

- In **HTTP/Step4**, the server code is refactored, with a new **Runnable** class **RequestHandler** that can carry out the request/response cycle for a single accepted request:

```
public class RequestHandler
  implements Runnable
{
  ...
  public void run ()
  {
    try
    (
      InputStream is = socket.getInputStream();
      OutputStream os = socket.getOutputStream();
    )
    {
      HttpRequest request = new HttpRequest(is);
      HttpResponse response = new HttpResponse(os);

      String path = request.getContextPath();
      System.out.println("Responding to " + path +
        " on thread " +
          Thread.currentThread ().getId ());
      if (path.equalsIgnoreCase(SHUTDOWN))
        shutdown = true;
      else
        response.sendResource(WEBROOT, path);
    }
    ...
  }
}
```

A Multi-Threaded Server

EXAMPLE

- By contrast the **service** method is much reduced:

```
while (!shutdown)
{
  try
  {
    (new Thread (new RequestHandler
      (serverSocket.accept()))).start ();
  }
  ...
```

  - So, for each new request, we create a new **Thread** object and let it carry out the work of handling that request.
  - When the response is sent, the thread's work is done, and it dies.

- Try this version out, and see that you will get new threads *ad infinitum* as new requests come in:

```
HttpServer running on port 80
Responding to /index.html on thread 9
Responding to /images/Logo.gif on thread 10
Responding to /favicon.ico on thread 11
Responding to /index.html on thread 12
Responding to /images/Logo.gif on thread 13
...
```

  - As long as the individual threads don't tarry too long in serving content (or delve into partial responses and Ajax push and that sort of thing), this threading policy can work well.
  - There is the ongoing cost of creating threads, but the need for concurrently running threads is not out of proportion, and if it were, then process and host clustering would be indicated.

- In the next chapter we'll consider a thread-pooling approach.

SUMMARY

- Threads of execution play a fundamental role in a running Java Virtual Machine, whether your code explicitly uses and controls them or not.
- Threads in the Java threading model are organized into thread groups for administrative purposes.
- Threads as modeled by the Core API map to threads as modeled by the JVM, which usually means direct use of thread support in the operating system.
- Nevertheless, some synchronization capability exists, largely through the use of **synchronized** methods and code blocks, and of **wait**, **notify**, and **join** methods.

Suspend/Resume and Cancel

LAB 8

In this lab you will enhance the threading code in the file-search application, to allow the user to suspend the search and then to choose to resume it or to cancel it altogether. This will require more than just a status flag and concurrent checks on status in the two threads: you will need to synchronize certain stretches of code to assure that they are waiting on each other.

Lab project: **Search/Step2**

Answer project(s): **Search/Step3**

Files: * to be created
src/cc/threads/Search.java

Instructions:

1. Open **Search.java** (which is exactly the completed version from the earlier demo) and find the **run** method of the inner class **Monitor**.
2. We're going to switch to using a **BufferedReader** to process keyboard input, because we may now need to read the standard input stream several times, and reading just the first byte of a text line isn't going to cut it. Before the main **while** loop, initialize a variable **in** to a new **BufferedReader**, based on a new **InputStreamReader**, based on **System.in**.
3. Remove the two-line body of the conditional block that's executed when input is **available**.
4. In the now-empty code block, call **in.readLine**, to clear the full line of text from the input stream.
5. Print a prompt to the user: do they indeed want to cancel? Print a blank line before the prompt, so that your text doesn't get lost in the ongoing status output produced by the main search method.
6. Read another line of text and capture this as the string **answer**.
7. If **answer** is not an empty string, and the first character is upper- or lower-case Y, then set the status to **CANCELED**.

Suspend/Resume and Cancel LAB 8

8. Build and test – not quite what we were hoping for! You should see that the search continues even while the user is asked to hit a yes-or-no key, which is not the desired behavior and certainly junks up the console UI:

```
run Search.java c:\Capstone
Searching for file: Search.java
From root directory: c:\Capstone
Hit ENTER to cancel ...
Found
C:\Capstone\JavaAdv\Examples\Search\Step1\src\cc\threads\Search.java
Searching C:\Capstone\JavaAdv\Labs\Lab2A
  (Hit ENTER here but prompt got swamped!)
Found C:\Capstone\JavaAdv\Labs\Lab2A\src\cc\threads\Search.java
Searching C:\Capstone\JavaAdv\Labs\Lab5A
  (Hit 'Y' here and program does terminate.)
```

9. The problem, of course, is that while **searchForFiles** is checking the status flag on each recursive call, it is not stopping to wait for the UI conversation to occur. Before this was not a problem since the status was changed immediately, with no perceptible delay in shutdown for the user. Now, we need to make the search method wait for the UI conversation, once it starts, to conclude.

10. Solve this problem by synchronizing the two threads over the key code blocks. First, enclose your UI conversation code in a **synchronized** block. Use the **status** value itself as the lock target.

    Don't enclose the whole loop! – just the things that happen after the user hits the first ENTER. Think about that: what would happen if you synchronized the **if** statement, or the whole **while** loop?

Suspend/Resume and Cancel LAB 8

11. What is the appropriate passage of code to synchronize in **searchForFiles**? Think about this one a moment, too: maybe just the test of the **status** flag and conditional **return**? Maybe more? The whole method?

    For basic suspend/resume/cancel behavior, just the test of **status** would do it. However we'll get cleaner console output if we also include the code that tests for the file in the given directory and prints out the search location and/or the found filename; if this code is interruptible by the monitor, then it could be part-way through printing a line when the monitor takes over and prompts the user. Synchronizing the whole method would be as bad as synchronizing the whole **while** loop in the monitor: it would basically remove the benefits of independent threads.

    So, place everything but the final loop that causes recursion in a **synchronized** block.

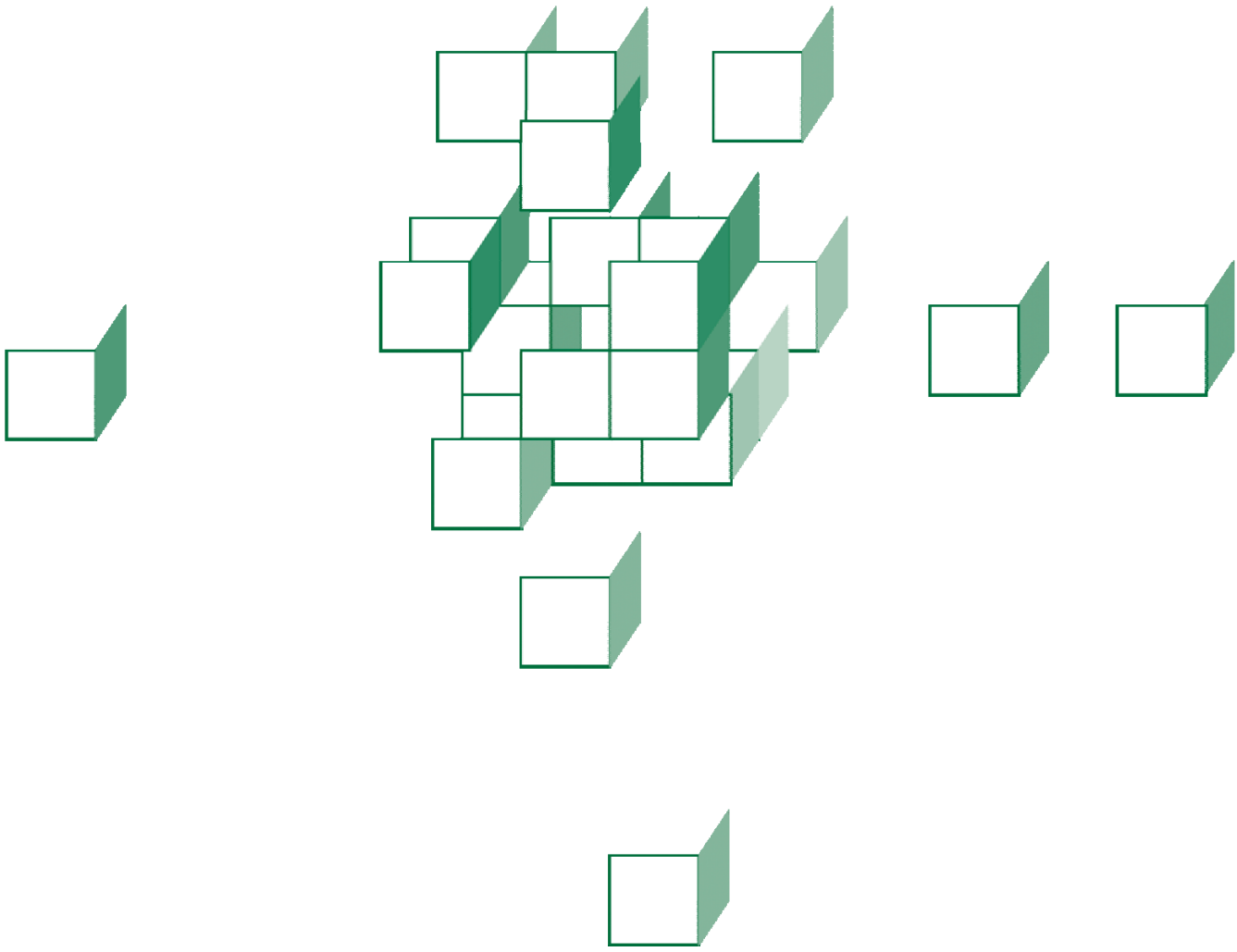
12. Rebuild and test again. You should now see a clean UI and proper control of the search thread, as in:

```
run Search.java c:\Capstone
Searching for file: Search.java
From root directory: c:\Capstone
Hit ENTER to cancel ...
Found
C:\Capstone\JavaAdv\Examples\Search\Step1\src\cc\threads\Search.java
Searching C:\Capstone\JavaAdv\Labs\Lab2A
Cancel search? n
Found C:\Capstone\JavaAdv\Labs\Lab2A\src\cc\threads\Search.java
Searching C:\Capstone\JavaAdv\Labs\Lab5A
Cancel search? y
```

    The answer code has one additional enhancement, which we will discuss after the lab.

CHAPTER 9

CONCURRENCY



OBJECTIVES

After completing "Concurrency," you will be able to:

- **Use synchronizers defined in the Concurrency API to manage concurrent access to shared state more easily and effectively.**
- **Make informed decisions about thread-safety strategies when multiple threads may share dynamic collections.**
  - Use **Collections.synchronizedXXX** methods for simple, top-to-bottom synchronization of calls.
  - Use Concurrency API collection types such as **CopyOnWriteArrayList** and **ConcurrentHashMap** for a balance of thread safety and performance.
  - Apply your own synchronization logic in classes that encapsulate collections.
- **Use atomic types and atomic operations.**
- **Implement thread pools using the ExecutorService.**

The Concurrency API

- **Found in package java.util.concurrent and subpackages are utilities for managing threads and the data they may share, at a higher level than the basic java.lang support for thread objects.**
  - There are **synchronizers** such as a **Semaphore** class that facilitate coordination of complex tasks between threads
  - **Thread-safe collections** implement the basic Collections API interfaces in ways that allow them to be safely shared.
  - **Atomic operations** over certain simple values can now be performed without complex locking logic.
  - **Thread pools** are available for parallel processing or for scalable request processing.
  - Various **queues** are implemented to facilitate sharing or piping of data between producer and consumer threads.
  - Many of the intermediate-level **locking objects** that make the above features possible are also available directly to application code, to build other, specific high-level abstractions.

Synchronizers

- **Using synchronized blocks or methods is one straightforward way to keep multiple threads out of one another's hair; careful use of wait and notify can be another.**
- **For more sophisticated concurrency scenarios, use synchronizers of a few stripes:**
  - A **Semaphore** can allows a finite number of threads concurrent access to some protected resource.
  - **CountDownLatch**, **CyclicBarrier**, and **Phaser** function in various ways as catch-up points for groups of threads, such that when all have finished some phase of operations, they can all proceed to the next one.
  - An **Exchanger** allows one thread in a pair to wait for the other, and for the two threads to swap data when ready.
- **Encapsulate concurrent-access logic and rules, either in the class that models the shared resource, or in another class dedicated to managing access to the resource. Don't leave it to the caller!**

An Object Pool

DEMO

- **We'll look now at a quick demonstration of the use of a Semaphore to control access to a pool of objects by a larger pool of threads.**
- **We'll see a pool of objects that is currently not protected from concurrent use, and see that as a result it is possible to crash trying to get an object once they are all checked out from the pool.**
- **Then we'll fix things up with a semaphore.**
  - Create the semaphore with some finite number of **permits**.
  - Code that wants to use the resource will first **acquire** the semaphore – an action which blocks while the semaphore is acquired by other threads equal to the number of permits.
  - When another thread **release**s the semaphore, a caller waiting to **acquire** it is allowed to continue.
- **Do your work in Synchronization/Step1.**
  - The completed demo is found in **Synchronization/Step2**.

1. Open **src/cc/threads/ConnectionPool.java** and see that this class manages an array of a fixed number of objects, and lets callers "check out" one object at a time and "check in" when done. Code is shown on the following page.

An Object Pool

DEMO

```
public class ConnectionPool
{
  private static final int POOL_SIZE = 10;

  private Object[] connections =
    new Object[POOL_SIZE];
  private boolean[] inUse = new boolean[POOL_SIZE];
  ...
  public synchronized Object checkOut ()
  {
    for (int i = 0; i < POOL_SIZE; ++i)
      if (!inUse[i])
      {
        inUse[i] = true;
        return connections[i];
      }

    throw new IllegalStateException ("...");
  }

  public synchronized void checkIn
    (Object connection)
  {
    for (int i = 0; i < POOL_SIZE; ++i)
      if (connections[i] == connection)
      {
        if (!inUse[i])
          throw new IllegalArgumentException("...");

        inUse[i] = false;
        return;
      }

    throw new IllegalArgumentException ("...");
  }
```

An Object Pool

DEMO

2. The class functions as its own client, for purposes of simple testing. The method **useAConnection** checks out an object, simulates a task that takes 5 seconds, and then checks the object back in:

```
public void useAConnection ()
{
  Object connection = checkOut ();
  String ID = Thread.currentThread ().getName ();
  System.out.println (ID + " checked out.");

  try
  {
    Thread.sleep (5000);
  }
  catch (InterruptedException ex) { ... }

  checkIn (connection);
  System.out.println (ID + " checked in.");
}
```

3. The **main** method runs twice as many threads over this method as there are objects in the pool.

```
public static void main (String[] args)
{
  ConnectionPool pool = new ConnectionPool ();
  for (int i = 0; i < POOL_SIZE * 2; ++i)
  {
    new Thread (pool::useAConnection).start ();
    try
    {
      Thread.sleep (250);
    }
    catch (InterruptedException ex) { ... }
  }
}
```

An Object Pool

DEMO

4. So, what will happen when we start 20 threads, one every ¼-second, and each thread occupies one of the 10 "connection" objects for 5 seconds?
5. Run the class and see that it doesn't hold up too well:

```
Thread-0 checked out.
Thread-1 checked out.
...
Thread-8 checked out.
Thread-9 checked out.
Exception in thread "Thread-10"
  java.lang.IllegalStateException:
    No connections available.
  at cc.thread.ConnectionPool.checkOut(...)
...
Thread-0 checked in.
Thread-1 checked in.
...
Thread-8 checked in.
Thread-9 checked in.
```

   - When the 11th thread tries to check out, it encounters an **IllegalStateException** because there are no available objects.
   - Now, the caller could deal with this externally, by entering a polling loop and taking other precautions.
   - But better for the pool to incorporate concurrency logic ...

An Object Pool

DEMO

6. Add a semaphore as a field of the class:

```
private Semaphore semaphore =
  new Semaphore (POOL_SIZE);
```

7. In the **checkOut** method, add code to acquire a permit for the requested object:

```
public Object checkOut ()
{
  try
  {
    semaphore.acquire ();
  }
  catch (InterruptedException ex)
  {
    System.out.println ("Interrupted?!?");
  }

  for (int i = 0; i < POOL_SIZE; ++i)
```

8. Add a call to release the permit when **checkIn** is called:

```
public synchronized void checkIn
  (Object connection)
{
  for (int i = 0; i < POOL_SIZE; ++i)
    if (connections[i] == connection)
    {
      if (!inUse[i])
        throw new IllegalArgumentException ("...");

      inUse[i] = false;
      semaphore.release ();
      return;
    }
```

An Object Pool

DEMO

- Now, one concern: if you leave the method **synchronized**, then a waiting call to **acquire** will eventually prevent the very call to **release** on which it is waiting – i.e. you'll have a deadlock.
- Test this, if you like – you'll see that as of the 11th call to **useAConnection**, everything freezes.
- You still want the synchronization, because even when one thread is free to check out, if another is free to check in, there will be a race over the array of available objects.
- A clean way to deal with this will be to reduce the scope of the **synchronized** block to cover just the check-in process.

9. Remove the modifier from the method signature:

```
public synchronized Object checkOut ()
```

10. Instead, wrap the **for** loop in a synchronized block:

```
synchronized (this)
{
  for (int i = 0; i < POOL_SIZE; ++i)
    if (!inUse[i])
    {
      inUse[i] = true;
      return connections[i];
    }
}
```

An Object Pool

DEMO

11. Test now, and see that threads 10 through 19 now wait their turns by virtue of calls to **semaphore.acquire**:

```
Thread-0 checked out.
Thread-1 checked out.
...
Thread-8 checked out.
Thread-9 checked out.
  (pause for about 2.5 seconds)
Thread-0 checked in.
Thread-10 checked out.
Thread-1 checked in.
Thread-11 checked out.
...
Thread-8 checked in.
Thread-18 checked out.
Thread-9 checked in.
Thread-19 checked out.
  (pause for about 2.5 seconds)
Thread-10 checked in.
Thread-11 checked in.
...
Thread-18 checked in.
Thread-19 checked in.
```

- Each is granted a permit after about a 2½-second wait, as the first group of 10 starts to finish up, so that the second group of 10 gets rolling and once again starts are staggered every ¼-second.
- And, any sort of usage pattern now should be safe: 50 threads all rushing to the door at once, or a trickle of threads using objects but some tasks taking very long and choking off availability ... etc.

Working with Collections

- **All data sharing between threads poses concurrency issues, but collections are especially problematic:**
  - They are pervasive in most complex application code.
  - They suffer many moments vulnerable to race conditions in common use – similar to what we saw for arrays earlier in this chapter.
- **There are three main strategies for sharing collections of data between threads:**
  - **Do it yourself**, using techniques we've seen.
  - Derive **synchronized collections** using **Collections** utility methods such as **synchronizedList**.
  - Use **thread-safe collection** implementations provided in the Concurrency API.

Synchronized Collections

- The **Collections** class utility in **java.util** provides a set of methods for adapting an existing collection with a wrapper all of whose methods – readers and writers alike – are **synchronized**.

```
static <T> Collection<T> synchronizedCollection
  (Collection<T> backingCollection);
static <T> List<T> synchronizedList
  (List<T> backingList);
static <K,V> Map<K,V> synchronizedMap
  (Map<K,V> backingMap);
static <T> Set<T> synchronizedSet
  (Set<T> backingSet);
static <K,V> SortedMap<K,V> synchronizedSortedMap
  (SortedMap<K,V> backingMap);
static <T> SortedSet<T> synchronizedSortedSet
  (SortedSet<T> backingSet);
```

- The reference derived by calling any of these methods can be shared between threads – with some caveats, as follows.
- It is essential to provide this same "wrapped" collection to all users of the underlying collection; don't mix and match use of the backing collection with the wrapper, or you lose the benefit.

Synchronized Collections

- **Don't confuse "synchronized" with "thread-safe:" for example state can change between calls to synchronized methods.**
  - It's unsafe for one call to a synchronized method to rely on results of another. For example, without the use of a synchronized block, the following code is not guaranteed to avoid overwriting a key:

```
synchronized (stringList)
{
  if (!synchroMap.contains (key))
    synchroMap.put (key, value);
}
```

  - It follows that any use of iterators derived from the synchronized collection must itself be synchronized, as in:

```
synchronized (stringList)
{
  Iterator<String> it = stringList.iterator ();
  while (it.hasNext ())
    buffer.append (it.next ());
}
  or
synchronized (myStringList)
{
  for (String s : myStringList)
    buffer.append (s);
}
```

- **Finally, be aware that the inherent, total-synchronization strategy can perform poorly, as it serializes all use of the object and will even prevent concurrent reads.**

Thread-Safe Collections

- **A few thread-safe collection implementations are provided; each is designed perform well under certain assumptions.**
- **ConcurrentHashMap will perform much better than a synchronizedMap wrapper, but compromises on perfect synchronization.**
  - It allows **concurrent reads**.
  - It **locks data partitions** around **write operations**, and so only other writes within the same partition will be blocked.
  - But **concurrent writes** to different partitions are allowed.
  - An important implication is that there is no guarantee of **up-to-date reads**, because there is no certainty about the ordering of any given pair of read and write operations.
  - But you are guaranteed **consistency**: a read operation will get the results of the most recent, completed write operation.
  - A **concurrencyLevel** argument to the constructor can give you more partitions – which means more use of memory and slower lookups, but less chance of contention over writes.
  - You generally tune this to be roughly the **number of threads** you expect to be using the map.

Thread-Safe Collections

- The **CopyOnWriteArrayList<E>** and **CopyOnWriteArraySet<E>** address situations where reads are much more frequent than writes.
  - The strategy is to create a **new copy** of the underlying array on each write operation, and to use that as the data going forward.
  - Any threads **currently reading** information will complete their work on the **old copy** of data, and therefore will be entirely safe from surprises brought on by that write operation.
  - **New read operations** will get their data from the **new copy**.
  - This strategy performs well generally, but especially so when reads greatly outnumber writes – which is a common case.
- **The other safe collection types are all various sorts of queues.**
  - The **first-in, first-out** (**FIFO**) **queue** is especially important in multi-threaded programming – this model supports passage of data and also task scheduling.
  - Does processing of the queue **block** under certain conditions, or can it always be processed concurrently?
  - Various implementations are available to support different strategies: examples are **ArrayBlockingQueue<E>** vs. **ConcurrentLinkedQueue<E>**.

Atomic Operations

- **A common need in multi-threaded programming is to combine a few simple operations on a single value – such as a boolean or integer – into a single atomic operation.**
- **Classes in package java.util.concurrent.atomic encapsulate certain types of data to be manipulated via such operations.**
  - For instance **AtomicInteger** allows the caller to **get**, **set**, **compareAndSet**, **addAndSet**, **getAndIncrement**, **incrementAndGet**, and so on
  - The available methods may seem like overkill, but remember that our idea of a minimal interface must be different here, since calling two atomic operations in sequence would not itself be an atomic operation!
  - These methods typically take advantage of similar features in the operating system, and so avoid the performance penalties of synchronization in the JVM.
  - **Booleans**, **integers**, and **long integers** are supported, as well as object references via **AtomicReference<V>**.
  - There are **atomic arrays** of integers, longs, and objects.
  - Utilities such as **AtomicIntegerFieldUpdater<V>** facilitate managing fields that are not themselves atomic objects.

Checking Membership IDs

EXAMPLE

- In **ID/Step1**, consider **src/cc/threads/Members.java**, which wraps an **ArrayList** of numeric IDs and offers one mutating and one non-mutating method:

```
public class Members
{
  private static final int TYPICAL_SIZE = 1000;
  private List<Integer> IDs =
    new ArrayList<Integer> (TYPICAL_SIZE * 5 / 4);
  ...
  public void updateMembership ()
  {
    if (Math.random () >=
      ((double) IDs.size () / TYPICAL_SIZE / 2))
    {
      IDs.add (nextID);
      nextID = (nextID + 1) % (TYPICAL_SIZE * 2);
    }
    else
    {
      IDs.remove ((int)
        (Math.random () * IDs.size ()));
    }
  }

  public boolean findID (int ID)
  {
    for (int test : IDs)
      if (ID == test)
        return true;

    return false;
  }
}
```

Checking Membership IDs

EXAMPLE

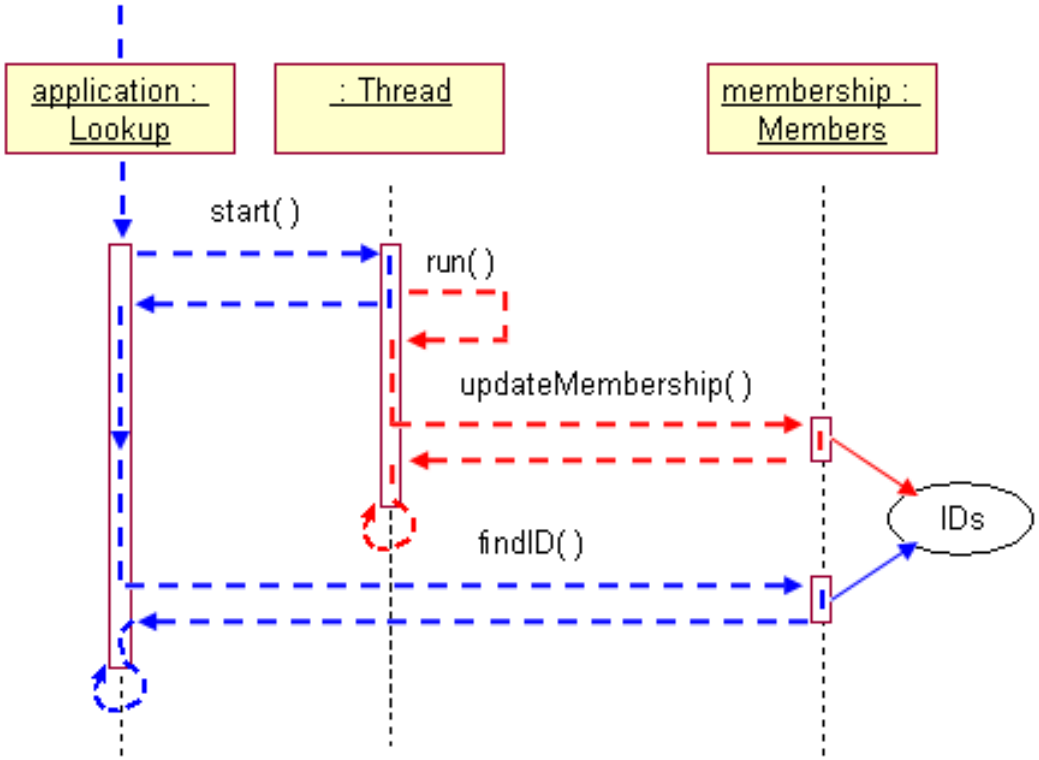
- In **src/cc/threads/Lookup.java**, we get two threads going, one repeatedly calling the mutating method, and one repeatedly calling the non-mutating method:

```
public class Lookup
{
  private static Members membership;

  public static void main (String[] args)
  {
    int limit = args.length != 0
      ? Integer.parseInt (args[0])
      : Integer.MAX_VALUE;
    membership = new Members ();
    Thread updater = new Thread (new Runnable ()
      { public void run ()
        { while (true)
            membership.updateMembership (); } } );
    updater.setDaemon (true);
    updater.start ();

    int attempts = 0;
    try
    {
      long start = System.currentTimeMillis ();
      int roller = 0;
      while (attempts++ < limit)
      {
        membership.findID (1100);
        if (++roller > 1000000) {/* print a dot */}
      }
      System.out.println ("Completed in " + ...);
    }
    catch (Exception ex) { ... }
  }
}
```

Checking Membership IDs

EXAMPLE

- So, we have set the stage for collisions in the **ArrayList** methods:



- What happens if **findID** tries to search the list while **updateMembership** is adding to or removing from it?
- We'll try this and several variants, and see what sorts of implementation strategies give what sorts of results.

Checking Membership IDs

EXAMPLE

- **ID/Step1** fails almost immediately, because it uses a simplified **for** loop, and the underlying **Iterator** is written to **fail fast** when multiple threads are active on the collection.
- Run the **Lookup** class and see this:

```
Failed after 4 attempt(s).
java.util.ConcurrentModificationException
  at java.util.ArrayList$Itr.checkForComodification
  at java.util.ArrayList$Itr.next
  at cc.threads.Members.findID
  at cc.threads.Lookup.main
```

  - You'll doubtless bump into this exception in your own multi-threaded coding – and, often, it will seem like an unnecessary nuisance.
  - But it is just the canary in the coal mine – again, see the "Reasons to be Cheerful" chart from the previous chapter – and almost always indicates a programming error.

Checking Membership IDs

EXAMPLE

- **ID/Step2** uses a classic **for** loop, and manages its own array index.

```
public boolean findID (int ID)
{
  int size = IDs.size ();
  for (int i = 0; i < size; i++)
    if (ID == IDs.get (i))
      return true;

  return false;
}
```

- Now we'll see how we do when we don't have those pesky co-mod exceptions blocking the door ... test this version now:

```
Failed after 6 attempt(s).
java.lang.IndexOutOfBoundsException:
    Index: 994, Size: 1003
  at java.util.ArrayList.rangeCheck
  at java.util.ArrayList.get
  at cc.threads.Members.findID
  at cc.threads.Lookup.main
```

  - So we do a little better, but we're really just "failing slow" as the array index runs out of bounds thanks to a concurrent **remove** operation.

Checking Membership IDs

EXAMPLE

- **ID/Step3** reduces the chance of failure by re-checking the collection size each time through the loop:

```
public boolean findID (int ID)
{
  for (int i = 0; i < IDs.size (); i++)
    if (ID == IDs.get (i))
      return true;

  return false;
}
```

- This does narrow the window in which a concurrent change might leave us with nothing to read. But it still fails.
  - This time, it may be an index-out-of-bounds, as before ...

```
Failed after 32 attempt(s).
java.lang.IndexOutOfBoundsException: Index: 998,
Size: 1010
        at java.util.ArrayList.rangeCheck
        at java.util.ArrayList.get
        at cc.threads.Members.findID
        at cc.threads.Lookup.main
```

  - ... or it may be a null-pointer exception, if we catch the moment after the size is increased but before a new element is placed in the new slot:

```
Failed after 4172 attempt(s).
java.lang.NullPointerException
        at cc.threads.Members.findID(
        at cc.threads.Lookup.main
```

Checking Membership IDs

EXAMPLE

- **ID/Step4** uses the **contains** method on the collection itself, instead of re-inventing it:

```
public boolean findID (int ID)
{
  return IDs.contains (ID);
}
```

- **contains** uses the underlying array, and other private fields.
- That array is usually larger than the collection size, so this further reduces the risk of an overrun.
- Test it now, and you will probably run out of patience waiting for a failure. The application prints a dot for every million successful calls to **findID**:

```
. . . . . .
. . . . . .
. . . . . .
```

- Still, the risk here is not zero, and you can bet that this code would wait until it were deployed at a client site before it would suddenly crash.

Checking Membership IDs

EXAMPLE

- **Partly to prove that the contains method is not a satisfactory solution, ID/Step5 intentionally increases the risk of failure.**
  - When it adds, it adds 20 elements at once; and when it removes, it **removes the last 20 elements at once**.
  - It then calls **trimToSize** on the collection, greatly increasing the (still fairly poor) odds that **contains** will get caught using an index that no longer exists on the array.
- **Even this version will not fail very frequently, but give it a test or two.**

```
....Failed after 4585911 attempt(s).
java.lang.ArrayIndexOutOfBoundsException: 1080
  at java.util.ArrayList.indexOf
  at java.util.ArrayList.contains
  at cc.threads.Members.findID
  at cc.threads.Lookup.main
```

  - On average we've seen this fail after around 10 million attempts.

Checking Membership IDs

LAB 9A

Suggested time: 30 minutes

In this lab you will try a few strategies for achieving thread safety in the Membership application. The starting step is just like **ID/Step1**, with a simple **for** loop that will fail fast. You will successively apply several thread-safety techniques:

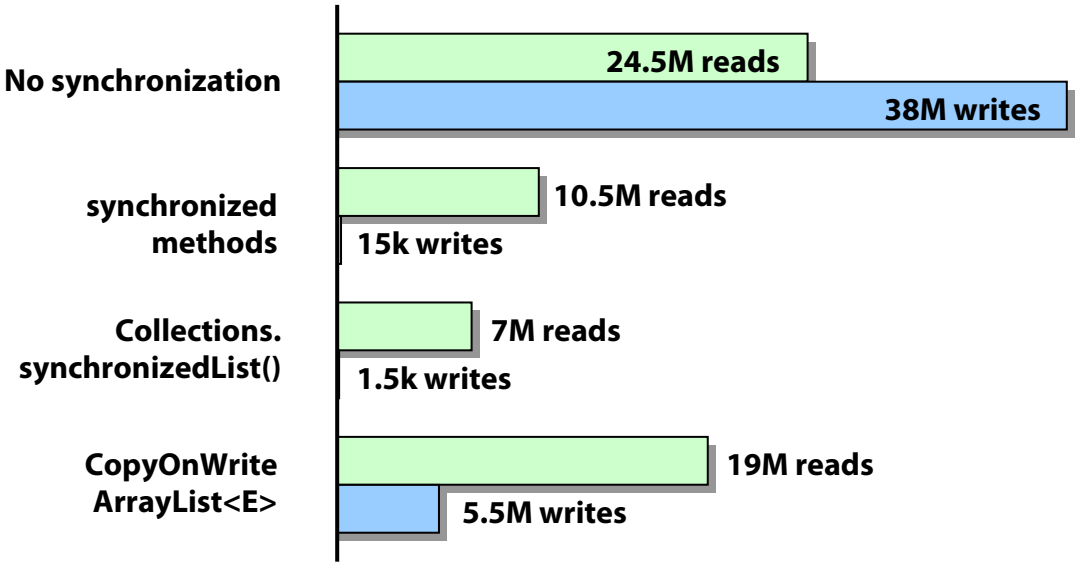
- Synchronizing your own methods
- Using wrapper given by **Collections.synchronizedList**
- Using a **CopyOnWriteArrayList**

The application has been altered slightly to perform crude benchmarking, so that you can compare not only the safety but the performance of the solutions.

Detailed instructions are found at the end of the chapter.

Performance Comparison

- **Here are some rough numbers from past testing using the variants of the Membership application.**
  - We measure throughput for five reader threads and one writer thread, running concurrently, over a single list, for ten seconds.
  - We add the "add" and "remove" operations together and call them "writes" where the lookup operations are the only sort of "reads.



- Analysis of the results:
  - **No synchronization** at all naturally provides the best throughput – not a safe strategy, but we can consider this a theoretical limit.
  - Using **synchronized** application methods, and relying on a wrapper given by **Collections.synchronizedList**, both manage a lot less work in the same unit of time.
  - The **CopyOnWriteArrayList** is the surprise winner – surprising not because it puts more read operations through, which we'd expected, but because it also allows a lot more writes to happen.

Thread Pools

- **For sheer performance or for scalability when implementing a server, a common programming problem is the thread pool.**
- **The Concurrency API provides high-quality thread-pool implementations that can be instantiated and put to work with a minimum of hassle.**
  - Call one of the factory methods on **java.util.concurrent.Executor**: **newCachedThreadPool**, **newFixedThreadPool**, etc.
  - The returned object implements **ExecutorService**.
  - This system isolates the calling code from implementation details such as the strategy for scheduling tasks, creating and recycling threads, and so on.
  - Simply **execute** a given **Runnable** task on the service, and trust that it will be performed, efficiently, in the future.
  - You can monitor progress, wait for certain tasks to complete, shut down the pool, and wait for all tasks to complete after shutdown.
- **Fine-grained management of tasks under a thread pool is achieved by use of different queue types.**
- **It is also possible to subclass the basic pool and queue implementations to take complete control of the scheduling and management policies.**

Multi-Threaded Search

EXAMPLE

- **We'll look at three alternate implementations of our file-searching utility, and compare their threading strategies:**
  - **Single-threaded**
  - Multi-threaded using a **thread pool** of recycled threads
  - Multi-threaded using **new threads** and not recycling them
- **Each of these new examples has crude benchmarking code built in so we can compare their performance.**
- **In Search/Step4, the pre-existing single-threaded implementation is enhanced with just that benchmarking code.**
  - To let the application run more efficiently, we've also removed the console output that shows status as what folder is being searched; now all we'll see are "hits" where the file has been found.
- **Try this one on a moderately large tree, as a baseline – and again, this application will be easier to use in a real command shell:**

```
compile
run Search.java /Capstone
...
Found /Capstone/JavaAdv/Search/Step1/src/cc/threads
  /Search.java
...
Time: 9911 msec
```

  - Exact results will vary, of course – even from run to run.

Multi-Threaded Search

EXAMPLE

- Now take a look at **Search/Step5**, and see that a thread pool has been put in place.
  - See **src/cc/threads/Search.java**:

```
private static final int POOL_SIZE = 20;
private static ExecutorService threadPool =
  Executors.newFixedThreadPool (POOL_SIZE);
private static AtomicInteger workerCount =
  new AtomicInteger (0);
private static AtomicInteger pathsThatSpawned =
  new AtomicInteger (0);
private static AtomicInteger pathsThatDidNotSpawn
  = new AtomicInteger (0);
```

- Dividing a tree search up for purposes of parallel processing is not straightforward.
- But the approach shown here gives pretty good load-balancing, and trades some precision for efficiency since calculating the work per worker would cost more than we lose in the form of unequal loads.

Multi-Threaded Search

EXAMPLE

– We take the opportunity to schedule new tasks on open threads as we drill down into child directories:

```
File[] children = path.listFiles ();
int count = children.length;
for (File child : children)
  if (child.isDirectory ())
  {
    if (--count == 0 ||
          workerCount.get () >= POOL_SIZE)
    {
      pathsThatDidNotSpawn.incrementAndGet ();
      searchForFile (filename, child);
    }
    else
    {
      pathsThatSpawned.incrementAndGet ();
      threadPool.submit
        (new Searcher (filename, child));
    }
  }
```

– The inner class **Searcher** is a **Runnable** that just calls **searchForFile**, continuing the recursion – but of course on a separate thread.

Multi-Threaded Search

EXAMPLE

- But we cap that task-submission behavior, waiting for a worker to run out of sub-tree to process.
- Each worker increments a global count, works, and then decrements the count before dying. This lets a new worker tackle whatever part of the tree comes up for searching next:

```
public void run ()
{
  workerCount.incrementAndGet ();

  try
  {
    searchForFile (filename, path);
  }
  catch (IOException ex)
  {
    System.out.println
      ("IOException while searching.");
  }

  if (workerCount.decrementAndGet () == 0)
  {
    System.out.println ();
    ViewThreads.showAllThreads ();
    System.out.println ();

    threadPool.shutdown ();
  }
}
```

- When the worker count comes to zero, we know the search is completed, so we call **shutdown** on the thread pool – which otherwise would sit waiting for new tasks to run.

Multi-Threaded Search

EXAMPLE

- The code that launches the process now does so on a thread – scheduling the first of many **Searcher** tasks – and then awaits the termination of the thread pool, as initiated by the **shutdown** call we just saw.

```
status = Status.SEARCHING;
new Monitor ().start ();
threadPool.submit
  (new Searcher (filename, path));
threadPool.awaitTermination
  (1, TimeUnit.HOURS);
status = Status.DONE;
```

- We also keep count of how many nodes in the tree submit new tasks to the thread pool vs. continue recursion on the current thread.

Multi-Threaded Search

EXAMPLE

- **Try this version out, and in most cases you'll see a significant speed advantage.**
  - You'll also notice that we're dumping the thread tree – using old friend **src/cc/thread/ViewThreads.java** – and you can see how the thread pool is organized.

```
compile
run Search.java c:\Capstone
Using a pool of 20 threads.
...
Found C:\Capstone\JavaAdv\Labs\Lab2A\src\cc\threads
  \Search.java

  system
    Reference Handler
    Finalizer
    Signal Dispatcher
    Attach Listener
    main
      main
      Thread-0
      pool-1-thread-1
      pool-1-thread-2
      ...
      pool-1-thread-19
      pool-1-thread-20

Time: 3528 msec
Paths that submitted new tasks: 195
Paths that didn't submit tasks: 1246
```

Multi-Threaded Search

EXAMPLE

- **We won't investigate Search/Step6 quite so closely, but it may be worth a quick tour and a test.**
  - We still keep a **count and cap** on the number of tasks that are active concurrently.
  - But now when we create a task it is run on its own new thread – **no pooling**.
  - When the work is done, the thread **dies**, and makes room for another to search elsewhere in the file system.
- **So the tradeoff, for performance purposes, is:**
  - **No overhead costs** for managing a fixed-size pool of threads and recycling them to new tasks.
  - Ongoing cost of **creating new thread objects**, and cleaning them up when they die.
- **If you test this version, you'll see little or no speed advantage one way or the other.**
  - Some testing with very large target trees has shown an increasing edge for a well-tuned thread pool.
  - But the real win of pools is usually in avoiding a thread-per-task policy when the task is long-running – such as managing an open-ended user or remote-client conversation.
- **You might also try tuning the size of the pool, in either version: try edge cases such as one or two threads, or much larger pools.**

Parallel Processing

- **Threading, and attention to concurrency issues, are two fundamental skill sets for enterprise Java programmers.**
- **Especially as multi-processor and multi-core machines become ever more common, the question naturally occurs how best to use them – or, how best to map the abstractions we know as threads to such very concrete things as CPUs and internal cores?**
- **A prior question might be, whose problem should this be ...**
  - The application programmer's?
  - The JVM's (or his or hers who built the JVM)?
  - The operating system's?
- **Historically, it has largely been left to the operating system to manage a JVM instance, in a JRE, as a process – and thus to decide what threads in that process run where.**
  - Many Java-EE server vendors handle their performance, scalability, and load-balancing concerns at this level.
- **With Java 8, the JVM begins to assume some responsibility for this, where the OS API allows it, and it exposes an API for this purpose, which is the Streams API rooted in `java.util.stream`.**
  - Streams can be **sequential** or **parallel**, and in the latter case they may be mapped to threads and physical cores and processors.
  - The API does not expose the logic of assigning to cores, though – instead the programmer is encouraged to define "parallelizable" logic and to let the JVM manage threading and parallelism from there.

A Multi-Threaded Server

LAB 9B

Suggested time: 15-30 minutes

In this lab you will convert the multi-threaded HTTP server shown in the previous chapter to use a fixed-size thread pool, instead of an endless series of throwaway threads.

Optionally, you will build a new client application that fires a lot of requests at the server, to see how it behaves, and try some other experiments.

Detailed instructions are found at the end of the chapter.

SUMMARY

- **synchronized**, **wait/notify**, and **join** have their places in application code, but for many common concurrency problems there is a better, pre-packaged solution in the Concurrency API.
- Synchronized collection wrappers as handed out by the **Collections** class utility provide a high degree of thread safety, but still require some attention in how they are used, and often don't perform well.
- Concurrent collection types such as **CopyOnWriteArrayList** and **ConcurrentHashMap** are often better choices; but always start with an analysis of the usage pattern, and decide from there.
- The **ExecutorService** provides
  - A key encapsulation that allows multi-threading strategy to be separated from the act of scheduling tasks
  - Several high-quality thread-pooling implementations, ready to go

Checking Membership IDs

LAB 9A

In this lab you will try a few strategies for achieving thread safety in the Membership application. The starting step is just like **ID/Step1**, with a simple **for** loop that will fail fast. You will successively apply several thread-safety techniques:

- Synchronizing your own methods
- Using wrapper given by **Collections.synchronizedList**
- Using a **CopyOnWriteArrayList**

The application has been altered slightly to perform crude benchmarking, so that you can compare not only the safety but the performance of the solutions.

Lab project: **ID/Step6**

Answer project(s): **ID/Step7** (synchronized methods)
ID/Step8 (synchronizedList)
ID/Step9 (contains method)
ID/Step10 (CopyOnWriteArrayList)
ID/Step11 (no synchronization, for comparison)

Files: * to be created
src/cc/threads/Members.java
src/cc/threads/Lookup.java

Instructions:

1. Review the starter version of **Lookup.java**, and see that we've changed a few things. We now run one updater thread and five lookup threads; and instead of doing a specific number of lookups, we run all threads for a fixed time. Also we have added counters to **Members.java**, and **Lookup**'s **main** method prints out the counts of lookups, adds, and removes.
2. Test, and see that you get an immediate failure, as observed in **ID/Step1**, and for the same reason: concurrent modification through an iterator. It looks a bit different, because (a) each of the five reader threads gets the exception, and (b) the main thread no longer encounters the exception directly, and the application continues to run for ten seconds, because the writer thread is unaffected.
3. Make both methods **updateMembership** and **findID synchronized**.

Checking Membership IDs LAB 9A

4. Build and test again, and you should see that there are no failures. And, your results will naturally vary from those shown below, but make note of the counts shown in the console: how many lookup operations, how many add and remove operations. This gives us a general sense of the "throughput" of the application.

```
Running ...
Lookups:      6900406
Adds:            2216
Drops:           2238
```

   Note that you can provide a number of seconds in which to run all the threads. Ten seconds is the default but you may prefer a shorter test as you move through the lab: just set a program argument of "3" or "5" or whatever you prefer.

   This is the intermediate answer in **ID/Step7**.

5. We could be a little more precise by synchronizing only the **remove** operation in **updateMembership** – try this if you like – but let's move on to another approach, which is to use a synchronized collection object. This transfers the responsibility for synchronization to the collection itself, so calls to **remove** and **contains** are guaranteed to be safe. First, remove your **synchronized** modifiers from the two methods.

6. At the bottom of the file, where the **IDs** collection is initialized, wrap the **ArrayList** in a call to **Collections.synchronizedList**.

7. Build and test – oops! What's wrong with using the synchronized list object?

```
Failed after 71 attempt(s).
java.util.ConcurrentModificationException
        at java.util.AbstractList$Itr.checkForComodification(...)
```

8. Recall that any code that uses iterators on a synchronized collection must still **synchronize** its use of the iterator. So place a synchronized block around the loop, using the collection itself as the lock target.

9. Test and now you get success.

```
Running ...
Lookups:      6255819
Adds:             133
Drops:            134
```

   This is the intermediate answer in **ID/Step8**.

10. Now forget the for-each loop entirely: replace the whole contents of the **findID** method with a call to **contains**. This does not have to be **synchronized**.

11. Test again and see similar results. This is the intermediate answer in **Step9**.

Checking Membership IDs LAB 9A

12. Now let's try one of the concurrent collection types. Remove the call to **Collections.synchronizedList**, and instead of creating an **ArrayList**, create a **CopyOnWriteArrayList**. This class has no constructor that takes a prepared size, so drop the argument that we've been passing to the **ArrayList** constructor.
13. Test again, and for the first time you should see a significant jump in throughput.

```
Running ...
Lookups:     18930253
Adds:         2677540
Drops:        2677552
```

    Again, exact results will vary, for a great many reasons; processor speeds, number of processors or processor cores, hyperthreading, other loads on your machine at the moment ... it's borderline-impossible to do useful benchmarking on a typical personal computer. But you should see a clear relationship between all the previous solutions and this one. The count of lookups performed in a fixed time will have jumped by a factor of two or three – that's a huge performance win. And, **CopyOnWriteArrayList** is supposed to perform at its best when reads greatly outnumber writes – and you can see that they do, in all tests so far, though if anything the number of write operations has increased with this newest strategy. So all six threads seem to have gotten better access to processors. We might expect **CopyOnWriteArrayList** even further to outperform **Collections.synchronizedList** if the ratio of reads to writes were greater.

    This is the final answer in **ID/Step10**.

14. Finally, if you like, remove the **synchronizedList** wrapping from the initialization of **IDs**. This basically takes us back a few steps, to something like **ID/Step4** but with a six-thread test instead of that step's two-thread test. Run the application to get an idea of the performance penalty for using any sort of synchronization: this version outperforms all previous ones, even though we also know it's a bit of a ticking time bomb. This is the alternate answer in **ID/Step11**.

```
Running ...
Lookups:     23396088
Adds:        18600098
Drops:       18600088
```

A Multi-Threaded Server

LAB 9B

In this lab you will convert the multi-threaded HTTP server shown in the previous chapter to use a fixed-size thread pool, instead of an endless series of throwaway threads.

Optionally, you will build a new client application that fires a lot of requests at the server, to see how it behaves, and try some other experiments.

Lab project: **HTTP/Step4**

Answer project(s): **HTTP/Step5**

Files: * to be created
src/cc/http/HttpServer.java
compile or **compile.bat** (if working from the command line)

Instructions:

1. Run the starter version of the server as a baseline. For non-Windows systems, remember that you may need to adjust the server's port number or 'sudo' the script – see the instructions to Lab 07A for notes on how to do this.
2. Use a browser or the **HttpSocketClient** to make requests at **http://localhost** (port 80). You'll see that it uses a new thread in order to handle each successive HTTP request.

```
HttpServer running on port 80
Responding to /index.html on thread 9
Responding to /images/Logo.gif on thread 10
Responding to /favicon.ico on thread 11
Responding to /index.html on thread 12
Responding to /images/Logo.gif on thread 13
...
```

3. Shut down the server – you can request the **/shutdown** URL, or just terminate the server process.
4. Open **HttpServer.java** and add a field **threads**, of type **int**. Initialize this to a small number, so that it will be easy to observe the behavior of the thread pool. (Essentially, you will have to make threads + 1 HTTP requests to see the pool re-use a thread.) The answer code sets a pool size of three.

A Multi-Threaded Server LAB 9B

1. At the top of the **service** method, initialize a local **ExecutorService** variable **threadPool** to the results of a call to **Executors.newFixedThreadPool**. Pass your **threads value** to this method to determine the pool size.
2. In the **while** loop and **try** block, instead of creating a new **Thread** object and **start**ing it, simply call **threadPool.submit**, passing your new **RequestHandler**.
3. Test the server again, and see that it now recycles the threads you've allocated to the pool as new requests come in.

```
HttpServer running on port 80
Responding to /index.html on thread 9
Responding to /images/Logo.gif on thread 10
Responding to /favicon.ico on thread 11
Responding to /index.html on thread 9
Responding to /images/Logo.gif on thread 10
...
```

From here, you're in a position to manage and tune the performance of the server process by changing the size of the thread pool; by using a different pooling or executor-service strategy; or by implementing your own and plugging it in – and this of course is the point of the **ExecutorService** abstraction.

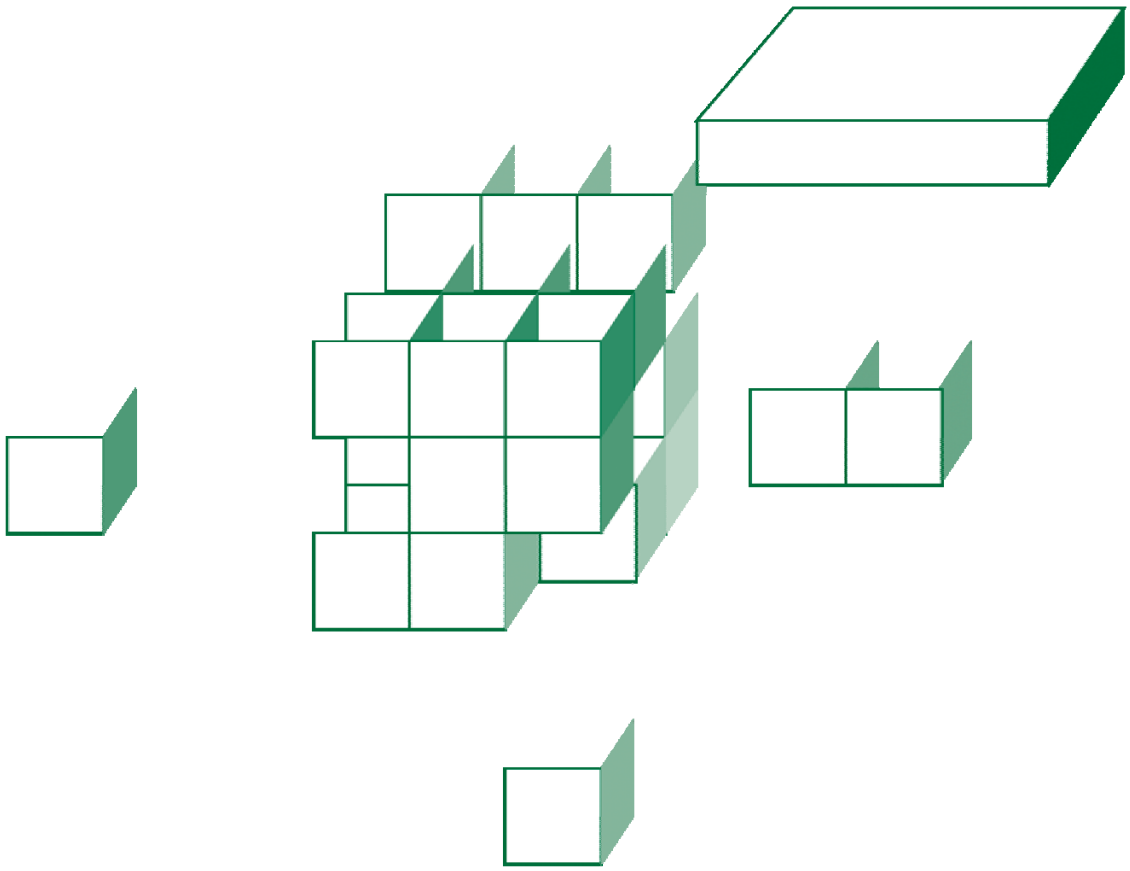
Optional Steps

4. Write a new client application to stress-test the server. Give it its own thread pool, and submit a long series of requests to URLs supported by the server. Write the client to shut down gracefully when all requests are answered – you can see the technique for checking if all work is complete in the multi-threaded search example.

   You can find one such client in the answer project: **cc.sockets.StressTestClient**.
5. Try adding code to the server to vary the response time for certain URLs – for example before getting the **favicon.ico** file you might force a brief sleep. See how this affects overall behavior and performance, and see if you can then improve things by tuning the pool size.
6. Assure that your server shuts down cleanly when it gets the remote command to do so: do requests in process at that moment get served completely?

Chapter 10

Reflection



OBJECTIVES

After completing "Reflection," you will be able to:

- **Describe the importance of meta-data and late-binding frameworks for complex software systems.**
- **Use the Java Reflection API to browse type information based on any class in the JVM:**
  - Walk inheritance hierarchies.
  - Read type parameters for generics.
  - Read collections of fields and methods.
  - Get type information on those, and recurse back through classes to other fields and methods.
- **Use Reflection to dynamically manipulate objects at runtime, where type information may not have been available at build time:**
  - Instantiate a class using meta-data.
  - Read and write fields on the resulting object.
  - Invoke methods on the object, passing parameters, reading return values, and catching exceptions.
  - Create or cast objects of a specific type using parameterized factory methods.

Early vs. Late Binding

- **Java offers two major approaches to interacting with objects, both of which follow patterns found elsewhere in the software industry.**
- **One can encode strong type information into a source file, and compile the code into a class that knows exactly what to expect at runtime.**
  - This is known as **early binding**, because the class is bound to the type information at build time.
  - A compiler can **validate source code** against known type information – almost all compile errors we've encountered in this course express this form of validation.
  - The resulting code is **narrowly targeted**.
- **One can write source code that adapts to whatever object types are encountered at runtime.**
  - This is known as **late binding**, because only at runtime can the virtual machine check to see that a desired method exists, offers the right signature, etc.
  - The means to this approach is the **Java Reflection API**.
  - Compilers cannot validate Reflection source code, because it speaks only in terms of **Object**s and other generic constructs.
  - Thus at runtime, there is a **greater risk of failure**.

Use Cases

- **The ability to read type information at runtime lets applications and frameworks offer a range of sophisticated features.**
- **The Java Serialization API uses Reflection to determine the serial form of an object, and to instantiate the right objects when reading a stream of serialized data.**
- **Other forms of serialization are also Reflection-based, and most also take advantage of the JavaBeans conventions for "getters and setters" to recognize state elements:**
  - The **Java API for XML Binding**, or **JAXB**, can translate Java object graphs to **XML** representations and vice-versa.
  - The **Java Persistence API**, or **JPA**, can carry out SQL operations against **relational databases** for Java classes known as **entities**.
- **Web technology including JavaServer Pages (JSP) and JavaServer Faces (JSF) use an expression language to identify specific properties in complex object models, and these expressions are evaluated dynamically using Reflection.**
- **Enterprise components especially are likely to be deployed unpredictably, and asked to work in different ways in different times and places.**
  - Many use some sort of **dependency injection**, by which an outside agent uses **external configuration** to choose what types of objects are created and what objects connect to what other objects.

Disassembly

- **The Reflection API is just the code-visible face of the type model that the virtual machine is using internally anyway.**
- **This type information is so important and so prevalent that many of the nicer features of the Java environment are based on it.**
- **A simple example of reflection can be found in the Java disassembler that's available with the SDK: this is called javap.**
  - Point this tool at any class name on the current class path:

```
javap java.lang.Object
Compiled from "Object.java"
public class java.lang.Object{
    public native int hashCode();
    static {};
    public java.lang.Object();
    protected void finalize();
       throws java/lang/Throwable
    public final native void notify();
    public final native void notifyAll();
...
```

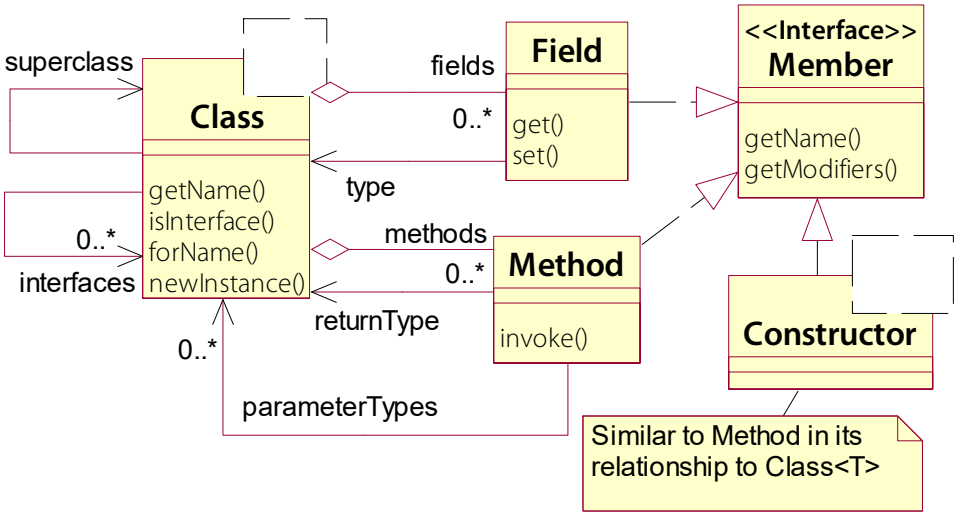
  - Not the most user-friendly output!
- **Popular IDEs use this same feature and put a much more sophisticated skin on it.**
  - When a tool offers an **auto-complete** feature when you're writing code, this is where it comes from.
  - Likewise the tooltip you might see with quick read on the **signature of a method** called from your code.

Documentation

- **Another great example of navigating type information is the javadoc utility, also available with the SDK.**
  - The **javadoc** system relies on the special "doc comment" syntax we discussed at the beginning of the course.
  - It extracts comments for classes, fields, methods, constructors, inner classes, etc., and formats them into an HTML page that documents the class.
  - Even better, it follows all those **Class** references and generates HTML links to similar pages for those types – this is the part of the utility that is enabled by Reflection.
- **Again, other tools go farther in developing features based on this capability.**
  - An IDE will typically offer some sort of automatic navigation to documentation based on mouse interaction over a symbol in source code.
  - It may allow you to navigate directly to the definition of a field, method, or class that you're looking at in your editor.

The Reflection API

- The **Java Reflection API** is deployed primarily in the package **java.lang.reflect**.
  - One class in the API, called **Class<T>**, is in **java.lang**, for both functional and historical reasons.



- **The API is firstly a way to navigate a type model.**
  - Usually starting from a **Class<T>** object, one can browse the inheritance tree (**superclass** and **interfaces**) and read the sets of **fields**, **methods**, and **constructors**.
  - On any of these members, one can read basic type information, such as field type or method signature.
  - This actually leads back to **Class<T>**!

The Class<T> Class

- The **Class<T>** class encapsulates meta-data about a single type **T** known to the JVM at runtime.
- As with the rest of the Reflection API, this class is used internally by the JVM to manage type information.
  - This is a big part of the reason that dynamic invocation through the Reflection API is not much slower than compiled method calls – they are essentially the same system.
- Derive a reference to the **Class** for a particular type using the static method **forName**, and pass the name of the type you want:

```
Class myClass = Class.forName ("com.me.MyClass");
```

  - If the class in question is not already loaded, it will be – sometimes **forName** is called just for this purpose.
  - If the class cannot be found, the method throws a **ClassNotFoundException**.
- A type known to the compiler at build time can also be referenced as a **Class**, by appending **.class** to the type name:

```
Class myClass = com.me.MyClass.class
```

- Finally, any object reference can be queried for its type, using the method **Object.getClass**.

```
Class myClass = myObject.getClass ();
```

Class<T> as a Generic

- You've noticed that **Class<T>** is a generic, but we're not parameterizing it in the examples thus far.
- **Class<T>** is a strange bird, by comparison to more obvious uses of generics like the Collections API.
- It is generic because it can exhibit some strongly-typed behavior dedicated to type **T**:

```
public T cast (Object obj);
public T newInstance ();
```

- But unlike collections, **Class<T>** is difficult to derive in a strongly-typed form.
  - Code that is truly reading type information dynamically must fall back to runtime captures of actual types, using **Class<?>**.

```
Class<?> myClass = Class<?>.forName ("MyClass");
Class<?> someClass = someObject.getClass ();

MyClass myObj = myClass.newInstance ();
  // error -- need to explicitly cast
```

- For this reason most reflection code just uses **Class**.
- **Constructor<T>** is also generic, and suffers similar practical limitations.

Walking Inheritance Hierarchies

- **Interrogate a Class object for whatever type information interests you.**
  - The most basic usage is to call **getName**.
- **Call getSuperclass or getInterfaces to find generalizations of the type.**
  - These too will be represented as **Class**es – **getInterfaces** returns an array of them.
- **The following loop will walk from the given type to the highest base class in the tree – and thus it will always produce "java.lang.Object":**

```
public void findMostBasic (Class<?> someType)
{
  while (someType.getSuperclass () != null)
    someType = someType.getSuperclass ();

  System.out.println (someType.getName ());
}
```

  - … or we could re-implement the **instanceof** operator:

```
public boolean instanceOf
  (Object test, Class<?> type)
{
  Class<?> testType = test.getClass ();
  while (type != null && testType != type)
    type = type.getSuperclass ();

  return type != null;
}
```

The Field Class

- **There are various ways to ask a Class what fields it holds as members.**
  - The simplest is **getFields**, which returns an array of **Field** objects for all the public fields.
  - **getDeclaredFields** is a bit more useful in most cases, since public fields are uncommon – this method returns a **Field** for every field declared on the class, regardless of visibility.
  - There are singular versions of each of these, **getField** and **getDeclaredField**, that retrieve a single **Field** by name.
- **As a subtype of Member, Field models the member name and can report modifiers such as private.**
- **It also provides the type of the field, as a Class.**

```
System.out.println (field.getName () + ": " +
  field.getType ().getName ());
```

The Method Class

- **Class** provides a similar suite of methods for finding all the **methods** on the modeled class.
- These all return objects of type **Method**, singly or in arrays.
- This type also offers **Member** semantics.
- Type information for methods is more complex than for fields, though:
  - Method names don't have to be unique! They can be overloaded. So finding a method means providing name as well as an array of **Class** objects representing the parameter list.
  - Get the return type of the method with **getReturnType**.
  - Get its parameter list as a **Class[]** with **getParameterTypes**.
  - The exception signature can be read, too: **getExceptionTypes**.

A Java "Reflector"

DEMO

- **In Reflector/Step1, we'll build a simple command-line tool that will report some basic type information.**
  - The completed demo is found in **Reflector/Step2**.

1. Open **src/cc/reflection/Reflector.java** and see that the class is already stubbed out with a **main** method that reads one argument from the command line and calls it **className**.
2. Start by loading the class into the JVM and getting its **Class** object:

```
String className = args[0];

Class<?> targetClass = Class.forName (className);
System.out.println
  ("Class name: " + targetClass.getName ());
```

3. Now walk a loop and report the supertypes:

```
Class<?> superClass = targetClass;
String indent = "  ";
while (superClass != null)
{
  superClass = superClass.getSuperclass ();
  if (superClass != null)
    System.out.println (indent + "Superclass: " +
      superClass.getName ());
  indent = indent + "  ";
}
```

A Java "Reflector"

DEMO

4. Test at this point – ask the tool for type information on the **Method** class itself by passing "java.lang.reflect.Method" as a program argument:

```
Class name: java.lang.reflect.Method
  Superclass: java.lang.reflect.AccessibleObject
    Superclass: java.lang.Object
```

5. Now add code to get an array of the public methods on the target class:

```
Method[] methods = targetClass.getMethods ();
for (int m = 0; m < methods.length; ++m)
  System.out.println ("  Method name: " +
    methods[m].getName ());
```

6. Run again, on "java.lang.reflect.Method" ...

```
Class name: java.lang.reflect.Method
  Superclass: java.lang.reflect.AccessibleObject
    Superclass: java.lang.Object
  Method name: hashCode
  Method name: getModifiers
  Method name: invoke
  Method name: equals
  Method name: getName
  Method name: toString
  ...
```

A Java "Reflector"

DEMO

7. ... and on "cc.math.Ellipsoid" (which class is included in the project):

```
Class name: cc.math.Ellipsoid
  Superclass: java.lang.Object
  Method name: getType
  Method name: getA
  Method name: setA
  Method name: getB
  Method name: setB
  Method name: getC
  Method name: setC
  Method name: getVolume
  Method name: getDefinition
  ...
```

- **We could go as far as we wanted with this.**
- **Imagine the various applications of type information as we're able to read it at this point:**
  - **Documentation**, as in **javadoc**, **javap**, and slicker IDE tools
  - **Code generation** – could you create a Java class that would generate a starter source file for a new class that implements a target interface? Absolutely.
- **What about dynamic invocation?**
  - The Reflection API is not just a read-only type browser.
  - **Class**, **Field**, **Method**, and other Reflection types offer methods that allow you to dynamically interact with objects at runtime.
  - Use this part of the API to build generic utilities, automated testing tools, and to parameterize your own applications.

Dynamic Instantiation

- **Create a new object of a given type in one of two ways:**
  - **Class.newInstance** can create a new object, but only for classes with public default constructors.
  - To pass arguments to an object's constructor, you need the **Constructor** object, which is a lot like a **Method** and can be derived with a call to **Class.getConstructors**, or a related method.
  - This latter approach is more work because you also have to build the argument list as an array of **Object**s.
  - We'll look at this process as it's applied to **Method**s a bit later.
- **In either case, two exceptions are common and must be caught:**
  - **InstantiationException**, which is the equivalent of the compiler refusing to instantiate an abstract type
  - **InvocationTargetException**, which wraps any exception thrown by the constructor when invoked.

```
try
{
  Object newObject = myClass.newInstance ();
}
catch (Exception ex) {}
```

- **To make life a little easier, since Java 7 all of these exceptions derive a common base, ReflectiveOperationException.**

Reading and Writing Fields

- **With an object of a given type at hand, and meta-data about that type, you can dynamically manipulate the object:**
  - Invoke methods on the object, passing arguments and reading return types.
  - Read and write values for fields on the object
- **These are all subject to accessibility constraints; Reflection isn't an end-run around member visibility!**
- **All of these behaviors work in terms of Objects and Classes, and so the compiler has no idea what you're doing – it checks correct use of the Reflection API, but after that you're on your own.**
- **Read a field using Field.get or getXXX – this is the only case in the API of such extensive overloading to provide you with an easier interface.**
  - As with all dynamic invocation, there's an odd inversion here: where we're used to **object.field**, here we call a method on the field object and pass the object reference:

```
double d = field.getDouble (object)
int i = ((Integer) field.get (object)).intValue ();
```

Dynamic Method Invocation

- **Invoking a method is more complicated.**
  - First, you need an array of **Class** objects just to find the method in question. This is true even if you know the target method is not overloaded.
  - Then, to invoke it, you must build an array of **Object**s. The first array was **parameter types**; this one is **argument values**.
  - This means promoting any primitive values to their box types, as you do when placing primitives in a collection.
  - Then the return value is an **Object** and must be downcast.
  - You must **catch** the **InvocationTargetException** and then unpack the exception object to find the real exception thrown by the method.

```
Class<?> dCls =
  Class.forName ("cc.cars.Dealership");
Class<?>[] params = { String.class, String.class };
Method method = dCls.getMethod ("findCar", params);
Object[] args = { "Toyota", "Prius" };
Object returnValue =
  method.invoke (someDealership, args);
```

  - Note that you can move in and out of dynamic invocation as it suits your purposes. We can take the newly-derived object, cast it appropriately, and then write ordinary code that the compiler can check against **Car**:

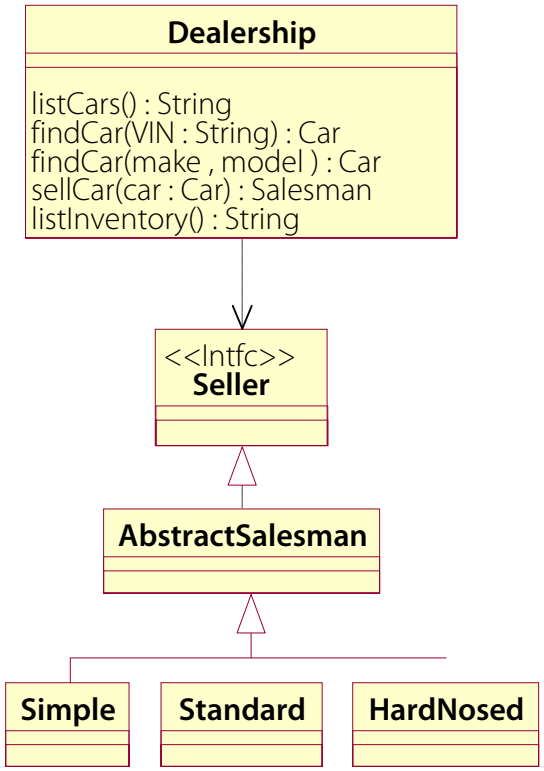
```
Car car = (Car) returnValue;
car.testDrive ();
Seller seller = dealer.sellCar (car);
```

Security Concerns

- **As we get deeper into the Reflection API's capabilities, you may well have begun to wonder about the security implications.**
  - Up to a point, we've just seen another way to read type information on a class – including for example whether a given field or method is **private**, **public**, etc.
  - Now we've started talking about being able dynamically to call a method or, say, read the value of a field – what if that member is **private**, or more generally wouldn't be visible to the code that's using the Reflection API?
- **Is Reflection a way around OO visibility rules?**
- **It isn't ... but then perhaps it is ... and ultimately it isn't.**
  - For starters, if you try to read/write a field or invoke a constructor or method that would not otherwise be visible to you, the call will fail with an **IllegalAccessException**.
  - But, **Field**, **Method**, and **Constructor** all extend the **AccessibleObject** class, which includes a read/write property **accessible** – so, you can call **setAccessible** and pass **true** ... and then your read, write, or invocation will succeed!
  - ... unless there's a **security manager** in force, because tweaking visibility rules on the fly in this way is obviously a concern. So there's a special permission that must be granted to your class in the **policy file**, or you will get an **AccessControlException**.

The Car Dealership

- In the Cars application, the **Dealership** selects a **S**eller when the user expresses interest in buying a specific car.
  - Several different types of **S**eller are modeled, each with different approaches to getting the best price and closing the sale.
  - Following the **Strategy** design pattern, **S**eller abstracts the semantics of selling a car, and different concrete strategies implementing the interface.



- Modeling the real world, in a way, the **Dealership** will choose a seller at random when the user "walks in."
- But, especially for testing purposes, it's important to have the ability to pre-select an implementation type from outside the source code.
- Currently the **Dealership** reads a system property and makes a runtime choice if that property is set – but only among certain known classes, and using a **switch/case** to select between statically-typed choices.

Forcing a Seller Type

LAB 10A

Suggested time: 30 minutes

In this lab you will improve the code that allows the type of **S**eller to be configured via a system property, so that it uses Java Reflection instead of the current, more limited approach.

Detailed instructions are found at the end of the chapter.

Dynamic Invocation

LAB 10B

Optional

Suggested time: 30 minutes

In this lab you will enhance the **Reflector** to allow you to invoke any method on any class from the command line – well, it will have to be a method that takes no parameters. You will look up a specific public method and call **invoke**, printing a string conversion of whatever is returned.

Detailed instructions are found at the end of the chapter.

Generics

- **Of course, the Reflection API must take cognizance of generic classes and methods.**
- **However, thanks to type erasure, there is really not much to learn about generics at runtime!**
- **Class<T> can be queried for its type parameters.**
  - Only the existence of type parameters in the declared class, and their bounds, can be learned.

```
TypeVariable<?>[] params =
  myGeneric.getTypeParameters ();
for (TypeVariable<?> param : params)
{
  System.out.print (param.getName ());
  for (Type bound : param.getBounds ())
    System.out.print (" extends " +
      ((Class<?>) bound).getName ());
}
```

  - Type arguments are unknown to the runtime; this is essentially what is erased by the compiler.
- **The same semantics are available on the Method class, allowing generic methods to be understood more completely, but with the same limitations.**

Reflecting Generics

EXAMPLE

- In **Reflector/Step4** is a final version of the reflector that shows generic class names to include their type arguments, if any.
  - Calls to **getName** on a class object are replaced by calls passing that class object to the new method **fullName**:

```
public static String fullName (Class<?> target)
{
  StringBuilder result =
    new StringBuilder (target.getName ());
  TypeVariable<?>[] typeParams =
    target.getTypeParameters ();
  if (typeParams.length != 0)
  {
    result.append ('<');
    boolean firstParam = true;
    for (TypeVariable<?> param : typeParams)
    {
      if (!firstParam)
        result.append (',');
      result.append (param.getName ());

      // We go looking for type bounds here ...

      firstParam = false;
    }
    result.append ('>');
  }

  return result.toString ();
}
```

Reflecting Generics

EXAMPLE

- Build and test the utility on a few Core API types:
  - **java.util.List**:

```
Class name: java.util.List<E>
...
```

  - **java.util.Map**:

```
Class name: java.util.Map<K,V>
...
```

  - **java.util.concurrent.DelayQueue**:

```
Class name: java.util.concurrent.DelayQueue
          <E extends java.util.concurrent.Delayed>
  Superclass: java.util.AbstractQueue<E>
    Superclass: java.util.AbstractCollection<E>
      Superclass: java.lang.Object
...
```

  - The utility is not fully tooled – it stops short of investigating all possible boundary types, such as where **T extends G<T>**.
  - **java.lang.Enum**:

```
Class name: java.lang.Enum<E extends **NYI**>
...
```

Parameterized Factories

- **We discussed a possible use of generics in an earlier chapter: strongly-typed factories for other objects.**
- **Reflection is required to make this practicable.**
  - In a generic class or method, it is impossible to invoke **new** directly, because the type argument that satisfies the parameter **T** is not known when the code is compiled.
  - But **Class<T>** and **Constructor<T>** provide strongly-typed methods that can create new instances of class **T** or cast an object reference to **T** and return it.
  - This is possible because specific objects dedicated to specific classes can be defined, using the construction **MyClass.class**.
  - It will not work on class objects known to the compiler only as **Class<?>** or **Constructor<?>** - which accounts for most reflection code.
- **Thus the typical approach is to create a generic factory method, letting a parameter to the method define the type argument explicitly:**

```
public static <T> T newObject (Class<T> cls)
{
  return cls.newInstance ();
}
```

  - Typically there will be some additional logic in such a method; if the caller has a **Class<SpecificClass>** on hand, the caller can invoke the **newInstance** method itself, without needing a wrapper method just to accomplish this.

Interpreting a Stream

EXAMPLE

- In **Factory** there is an application that can interpret a stream of objects written using the Java Serialization API.
  - In **src/cc/reflection/Factory.java**, see the **readObjects** method:

```
public static <T> Collection<T> readObjects
    (String source, Class<T> desiredType)
  throws IOException, ClassNotFoundException
{
  ObjectInputStream in = new ObjectInputStream
    (new FileInputStream (source));
  int size = in.readInt ();
  Collection<T> result = new ArrayList<T> (size);
  while (--size != -1)
    result.add
      (desiredType.cast (in.readObject ()));
  in.close ();

  return result;
}
```

Interpreting a Stream

EXAMPLE

- The **main** method exercises **readObjects** by asking it to read two files, one of strings and one of integers.
  - The files were created using **cc.reflection.Generate**.
  - The parameterized method calls can treat the resulting collections as strongly-typed, since the method has downcast the original **Object** references.

```
System.out.println ("Text is:");
for (String s :
    readObjects ("Strings.ser", String.class))
  System.out.print (s + " ");
System.out.println ();

int total = 0;
for (int i :
    readObjects ("Numbers.ser", Integer.class))
  total += i;

System.out.format
  ("Total number of non-space characters is %d.",
    total);
```

- Test the application and see the console output:

```
Text is:
Once more into the breach, dear friends!
Total number of non-space characters is 34.
```

SUMMARY

- **Reflection is expensive not so much at runtime but in code development.**
  - It is laborious.
  - Many exceptions can arise on each method in the API, so handling those meaningfully becomes a major chore.
- **For some applications it's overkill, certainly.**
- **For very generic tools, it is absolutely essential.**
  - For instance, imagine trying to write an IDE component of some sort – without Reflection, how would you provide some of the features we've discussed in this chapter – parse the source code manually?
  - Tools that use JavaBeans properties – including GUI builders and JSP containers – use Reflection to do so.
  - Distributed systems also make heavy use of Reflection for dynamic invocation and code generation – again, usually at a tools level.
- **Somewhere in the middle there, application problems exist that can be solved neatly with a few lines of Reflection code.**
  - The dynamic choice of a **S**eller implementation at runtime in the car dealership is a classic example.
  - It is a limited use of Reflection in which the desired interface is known, but the implementation might be configurable or even user-selectable.

Forcing a Seller Type

LAB 10A

In this lab you will add a feature to the car dealership that allows it to force a particular **S**eller subtype into use, based on a system property value. The **sellCar** method will check this property, and if it is defined the method will attempt to instantiate the appropriate class, cast the object to **S**eller, and return that object.

Lab project: **Cars/Step2**

Answer project(s): **Cars/Step3**

Files: * to be created
src/cc/cars/Dealership.java

Instructions:

1. Open **Dealership.java** and find the **sellCar** method. The method looks to a system property for the name of a seller type, and if not found it chooses either **Standard** or **HardNosed** at random.

   Then, it **switch**es over the type name, and in each **case** it sets the **s**eller to a new instance of a specific concrete class.

2. Remove the **switch/case** construct, leaving the rest of the method intact.
3. In place of the **switch/case**, create a **try/catch** system against the **ReflectiveOperationException**. In case of exception, log a **SEVERE** error.
4. In the **try** block, start by loading the named class, using **Class.forName** and passing **type**. Store the result in a **Class<?>** variable called **s**eller**Class**.
5. Construct an array **params** of **Class<?>** references with just **Car.class** as the contents.
6. Pass this array to a call to **s**eller**Class.getConstructor**, so that you get a reference to the constructor that takes only a **Car**. Capture this as a **Constructor<?>** called **ctor**.
7. Construct an array **args** of **Object** references – this time you're building the list of arguments for an actual call to the constructor, and so the only element in the array will be the **car** object that's passed to the **sellCar** method in the first place.
8. Now you're ready to create a new object of the configured type: call **ctor.newInstance**, passing your **args** array, and capture the results in an **Object** reference called **obj**.
9. Now set **s**eller to the value of **obj**, downcast to **S**eller. So, we don't know the exact runtime type, but we know that whatever it is it will have to implement this interface in order to do the job. At this point if an unrelated type were configured, you'd encounter a **ClassCastException**.

Forcing a Seller Type LAB 10A

10. Test the new feature, first by running normally with "buy ED9876" as the program arguments – or choose any VIN you see in the dealership's inventory.

    Whoops! What happened?

```
SEVERE: Couldn't instantiate configured seller type: Standard
java.lang.ClassNotFoundException: Standard
```

    Since the old system used simple class names in the **switch/case** system, the random-selection code produces these simple strings as well. But now, we try to interpret them as full class names, and that fails. This is also indicative of what would happen if you were to provide the full name of a class that, for some reason, couldn't be found in the class path at runtime.

11. Add the package specification to each of the literal strings "Standard" and "HardNosed" as used earlier in the **sellCar** method.

12. This will clean it up. Now, try this a few times, and you'll see a more or less 50/50 selection between the **Standard** and **HardNosed** seller, as designed for cases where you haven't configured a type explicitly:

```
Seller is class cc.cars.sales.HardNosed
"I can sell the car for $26,099.09."
quit

Seller is class cc.cars.sales.Standard
"I can sell the car for $26,099.09."
quit
```

13. Now, try the same test, but with the system property set.

    From the command line, you can run as follows:

```
java –classpath build -Dcc.cars.Seller.type=cc.cars.sales.Simple
  cc.cars.Application buy ED9876
```

    In an IDE, just add "-Dcc.cars.Seller.type=cc.cars.sales.Simple" – but be sure to set this in the "VM arguments" section of the run configuration, and not along with the "program arguments."

    You should see that you can control the Dealership's choice of seller, every time.

14. You've seen what it looks like when you supply the name of a non-existent class. Try configuring a real class, but one that isn't a seller – maybe "cc.cars.Part".

15. Test this and see that it still fails, but later in the process, as it was able to use the Reflection API to instantiate, but then couldn't cast **Part** to **S**eller.

16. You might want to experiment with other exceptions by changing the source code so as to make bad requests: wrong parameters for the constructor, wrong arguments to **newInstance**, cast the return to the wrong type, etc.

Dynamic Invocation

LAB 10B

Optional

In this lab you will enhance the **Reflector** to allow you to invoke any method on any class from the command line – well, it will have to be a method that takes no parameters. You will look up a specific public method and call **invoke**, printing a string conversion of whatever is returned.

Lab project: **Reflector/Step2**

Answer project(s): **Reflector/Step3**

Files: * to be created
src/cc/reflection/Reflector.java

Instructions:

1. Open **Reflector.java** and after the **className** is read from **args[0]**, see if a second argument has been provided. If it has, initialize a second string **methodName** to that value.
2. Place the last paragraph of code – the loop that prints out all method names – in a conditional block so that it only executes if **methodName** is not provided.
3. If method name is provided, start by constructing an array of **Class<?>** objects called **params**, and initializing it to an empty array (not to **null**).
4. Declare a local **Method** reference **targetMethod** and initialize it by calling **targetClass.getMethod** – pass your **params** array.
5. Instantiate the target class and capture the object reference in a new variable **targetObject**.
6. Build an empty array of **Object**s as your method arguments – call this **arguments**.
7. Invoke the method on the target object by calling **targetMethod.invoke**, passing **targetObject** and the **arguments** array. Print the result to the console.

Dynamic Invocation LAB 10B

8. Test, and you should be able to call any non-static method that takes no arguments. For example, run with the arguments "cc.math.Ellipsoid getType":

```
Class name: cc.math.Ellipsoid
  Superclass: java.lang.Object
Sphere
```

Try "cc.math.Ellipsoid getDefinition"?

```
Class name: cc.math.Ellipsoid
  Superclass: java.lang.Object
Where A = B = C, offering perfect rotational symmetry in all three
dimensions.
```

Try "java.util.Date toString" ...

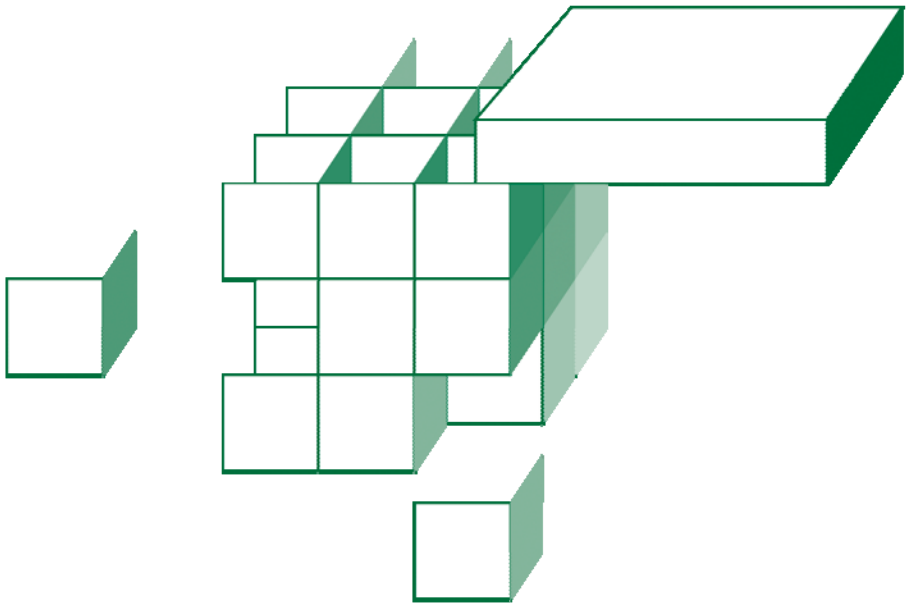
```
Class name: java.util.Date
  Superclass: java.lang.Object
Wed Jun 01 13:07:45 EDT 2005
```

Here's one that won't work – try "cc.math.Ellipsoid setType" ...

```
Class name: cc.math.Ellipsoid
  Superclass: java.lang.Object
java.lang.NoSuchMethodException: cc.math.Ellipsoid.setType()
        at java.lang.Class.getMethod(Class.java:1581)
        at cc.reflection.Reflector.main(Reflector.java:57)
```

Chapter 11

Dynamic Proxies



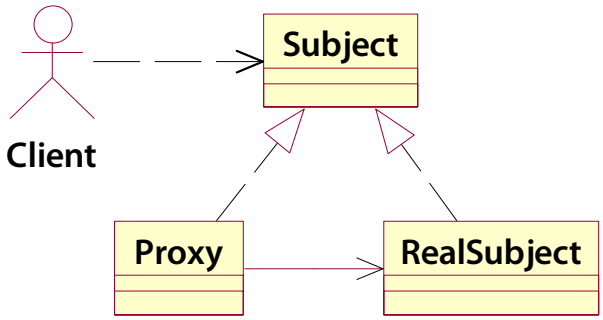
OBJECTIVES

After completing "Dynamic Proxies," you will be able to:

- **Describe the usefulness of proxies in various software design patterns and scenarios.**
- **Implement dynamic proxies as supported by the Java Core API.**
- **Use proxies to implement**
  - Mock objects
  - Filters
  - Caches

The Proxy Pattern

- The **Proxy** design pattern identifies a role for an object whose job it is to pass calls along to another object, known as the "real subject" or "delegate."



- The proxy class implements an interface also implemented by the delegate object's class.
- But its methods – for starters at least – all simply pass through to the delegate, and pass its return back as their own.
- From there, however, a proxy is in position to control access to the delegate, and to add features to the existing object.
  - It may **defer creation** of the delegate, where creation is expensive.
  - In distributed computing, the communication between a proxy and delegate may be over a **remote connection**.
- A proxy can also serve as a **single point of reference** for an object that actually changes or turns over from time to time.
  - **Dependency injection** systems use proxies to allow an object at a larger/longer scope easy reference to an object at a smaller/shorter scope, and it's the proxy's job to connect to different real subjects as they may change over time.

Dynamic Proxies in Java

- **Of course you can build proxies as static classes, by hand.**
- **But it's such a common requirement in enterprise systems – and anywhere dependency injection is used – that Java provides a means of building proxies dynamically.**
- **On the class java.lang.reflect.Proxy, call the factory method newProxyInstance; this generates an anonymous class, on the fly, and instantiates it:**

```
public static Object newProxyInstance
  (ClassLoader, Class<?>[], InvocationHandler);
```

  - Provide a **class loader** to use when loading the generated class; this will usually be the class loader for whatever class is about to use the interface, but in enterprise systems there may be multiple class loaders from which to choose more carefully.
  - Specify the **interfaces** for the generated class to implement. This is often just one interface, but you can generate a single proxy object that can manage multiple interfaces, so you pass an array here.
  - The final argument is a reference to an **invocation handler**. This is the implementation for your proxy – but not as a class that implements the interface.
  - Rather it is a dynamic handler that uses the **Reflection API** to respond to method calls – in a style similar to that used to invoke methods in the previous chapter.

The Invocation Handler

- **The invocation handler must implement the InvocationHandler interface; all calls to a proxy that uses this handler, for any of the methods on the "proxied" interface, will be translated to calls to the invoke method.**

```
public Object invoke
    (Object proxy, Method method, Object[] args)
throws Throwable
```

  - The **method** and **args** parameters are exactly as they would be in a call to **Method.invoke**, and your return will be treated as the return value of the proxied method.
  - The **proxy** argument ... well, on one hand it's exactly what it appears to be, which is a reference to the proxy itself.
  - But given that proxies often have **delegate** objects, it's all to easy to start thinking that this argument actually refers to a delegate.
  - It doesn't, and this means that you must **store the delegate** in the invocation handler, somewhere, if you mean to make calls on it.
- **Your invoke implementation will be the one and only implementation of the proxy class that uses this handler.**
  - If you leave this method un-implemented, the proxy will **do nothing** – it will not even make calls on a delegate.
  - A stock implementation would **pass all calls through**, and do nothing else.
  - Then, the typical implementation will pass calls through as a basis, but then **alter behavior** from there.

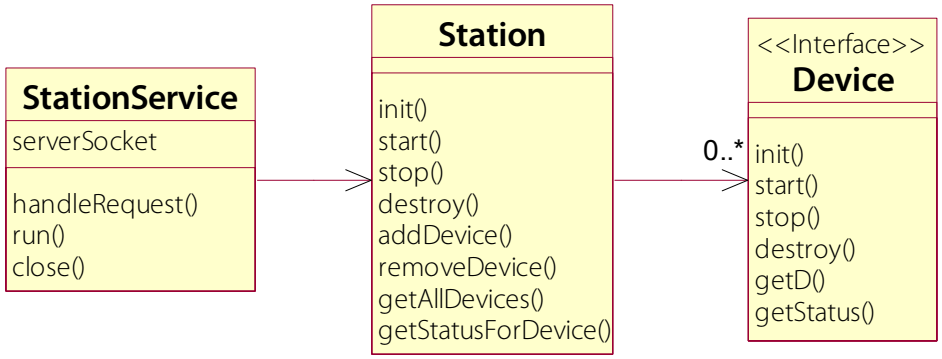
Use Cases

- **We use Java dynamic proxies for a wide range of purposes.**
  - **Masking** or **filtering** calls to certain methods on an object – for example to make an object **unmodifiable**, as with **Collections.unmodifiableXXX**
  - **Synchronizing** methods – e.g. **Collections.synchronizedXXX**
  - **Caching** results
  - **Controlling access** based on access control lists or security roles, or based on **parameter validation**
  - **Managing transactions** around method calls
  - **Translating** parameters and/or return types
  - **Intercepting** calls in order to carry out tasks **before** and/or **after** invocation – and this in turn leads to **aspect-oriented programming**, or **AOP**
  - **Implementing** an interface with many methods, when we really only have interesting behavior for one or two of them – and this leads to the notion of **mocking** as part of **testing** and test-driven development, which we'll discuss later in the course.

An Implementation and a Filter

DEMO

- See **Station/Step1**, in which a design for an MMC (management, monitoring, and control) station is partly implemented.



  - The **Station** can handle any number of hardware **Device**s.
  - Both **Station** and **Device** have a similar **lifecycle interface**, and the station will drive lifecycle calls on its devices.
- **Station** is implemented, but only the interface for **Device**s is defined; at this stage of development we have yet to start stamping out definitions for various sorts of hardware.
  - And we'll see the **StationService**, which provides a TCP interface to the station, in a later chapter.
- We'll look at some simple test code for the **Station**, and complete the implementation so that we can run the tests.
  - The completed demo is in **Station/Step2**.

An Implementation and a Filter

DEMO

1. Open **src/cc/monitor/TestStation.java**, and see that there's a **main** method that means to put a **Station** instance through a typical lifecycle – but with **Device**s that have yet to be initialized:

```
public static void main (String[] args)
{
  Station station = new Station ();
  Device device1 = null;
  Device device2 = null;

  station.addDevice (device1);
  station.init ();
  station.addDevice (device2);
  station.start ();
  String status = "Status of " + device1.getID ()
    + " is " + device1.getStatus () ...
  station.stop ();
  station.destroy ();
  ...
}
```

2. Add a proxy-factory method that uses this class' loader, implements the **Device** interface, and for the moment does nothing:

```
public static Device createDevice ()
{
  return (Device) Proxy.newProxyInstance
    (TestStation.class.getClassLoader (),
     new Class[] { Device.class },
     (target, method, args) -> {});
}
```

   – Another nice thing about **InvocationHandler** is that it can be treated as a **functional interface**, so here we use a simple **lambda expression** to implement the proxy's logic.

An Implementation and a Filter

DEMO

3. In **main**, use this method to create your devices:

```
Device device1 = createDevice ();
Device device2 = createDevice ();
```

4. Run the class as a Java application. You'll get no output, yet, but it will run cleanly. So, you're able to create objects that are considered to be **Device**s and can satisfy calls through a reference to a **Device** – even if as yet your proxy won't do anything when called.

5. Add a parameter to the factory method, and build a real implementation of **invoke** in the body of the lambda expression:

```
public static Device createDevice (String ID)
{
  return (Device) Proxy.newProxyInstance
    (TestStation.class.getClassLoader (),
     new Class[] { Device.class },
     (target, method, args) ->
      {
        System.out.println ("Called: " +
          method.getName () + " on " + ID);
        switch (method.getName ())
        {
        case "getID":     return ID;
        case "getStatus": return Status.RUNNING;
        default:          return null;
        }
      });
}
```

   – Now, we're implementing two of the methods on the interface, in short form, by returning the **ID** we're given or a stock status value – and tracing all calls through the proxy while we're at it.

An Implementation and a Filter

DEMO

6. Run again and see that your code is called, and works as designed:

```
Called: getID on Device1
Called: init on Device1
Called: getID on Device2
Called: init on Device2
Called: start on Device1
Called: start on Device2
Called: getID on Device1
Called: getStatus on Device1
Called: getID on Device2
Called: getStatus on Device2
Called: stop on Device1
Called: stop on Device2
Called: destroy on Device1
Called: destroy on Device2
Called: getID on Device1
Called: getID on Device2

Status of Device1 was RUNNING.
Status of Device2 was RUNNING.
```

An Implementation and a Filter

DEMO

- **So we've used a dynamic proxy in lieu of a static class, as a quick-and-dirty implementation of an interface, where most of the methods don't really require any logic.**
- **Now we'll create a second proxy that filters method calls.**

7. Add another factory method, this time taking another **Device** as a delegate:

```
public static Device
  createStartStopSuppressor (Device delegate)
{
  return (Device) Proxy.newProxyInstance
    (delegate.getClass ().getClassLoader (),
     new Class[] { Device.class },
     (target, method, args) -> {});
}
```

8. Implement **invoke**, this time to pass calls along to the delegate – and see how simple this is using the Reflection API in this context, because we've already been given a representation of the method invocation, in reflective terms:

```
  return (Device) Proxy.newProxyInstance
    (delegate.getClass ().getClassLoader (),
     new Class[] { Device.class },
     (target, method, args) ->
        method.invoke (delegate, args));
```

An Implementation and a Filter

DEMO

9. In **main**, wrap **device2** in a call to the new method – so you'll have a proxy to a proxy:

```
station.addDevice
  (createStartStopSuppressor (device2));
```

10. Run again, and see that the output is the same – so your additional proxy is passing calls through and is, for the moment, completely transparent.

11. Now add the following code, to make your proxy act as a "suppressor" or filter over the delegate, by refusing to forward calls to the **start** and **stop** methods:

```
return (Device) Proxy.newProxyInstance
  (delegate.getClass ().getClassLoader (),
   new Class[] { Device.class },
   (target, method, args) ->
    method.getName ().equals ("start") ||
        method.getName ().equals ("stop")
      ? null
      : method.invoke (delegate, args));
```

   – We might want to use this filter, on real devices, in order to shield them from expensive cleanup and re-initialization operations, perhaps in cases in which the station as a whole were going to go through a quick diagnostic or reboot.

An Implementation and a Filter

DEMO

12. Test again and see that the **start/stop** calls still go through to **device1**, but are never seen on **device2**:

```
Called: getID on Device1
Called: init on Device1
Called: getID on Device2
Called: init on Device2
Called: start on Device1
Called: getID on Device1
Called: getStatus on Device1
Called: getID on Device2
Called: getStatus on Device2
Called: stop on Device1
Called: destroy on Device1
Called: destroy on Device2
Called: getID on Device1
Called: getID on Device2

Status of Device1 was RUNNING.
Status of Device2 was RUNNING.
```

Proxy Classes

- **Proxy.newProxyInstance** is actually a shortcut for ...
  - Generating the **proxy class**
  - **Instantiating** the proxy class, by reflective calls upon it
- **For every requested proxy, there is a generated type.**
  - You can get this on its own by calling **Proxy.getProxyClass**.
  - You can also call **getClass** on any proxy, just as you can with ordinary objects.
  - In either case you get a **Class<?>** reference.
  - Proxy classes are **anonymous** – by which we mean that they don't have names that you can use in your code.
  - They do have names, as any class must have a name; but the return from **getName** on a proxy will be of a generated form that is JVM-implementation-specific.
- **A proxy class isAssignableFrom any of its supported interfaces, and a proxy can be tested accurately with instanceof.**

A Dynamic Cache

LAB 11

Suggested time: 30 minutes

In this lab you will build a **Cache** class that can provide dynamic proxies that will offer in-memory caching of values returned from certain methods, as keyed by certain arguments. So a client can use your class to improve performance over an interface of its choice, without the **Cache** having to be built with that interface in mind.

Detailed instructions are found at the end of the chapter.

Caching HTTP Responses

EXAMPLE

- In **HTTP/Step6**, we prove out the genericity of the **Cache<T>** class, by applying it to the results of HTTP requests as made by the **HttpSocketClient**.
- That class now implements an interface, found in **src/cc/sockets/Retriever.java**:

```
public interface Retriever
{
  public String retrieve (String URL);
}
```

- In **src/cc/sockets/LessStressfulTest.java**, a new class runs a long series of requests to the **HttpServer** – as we've seen **StressTestClient.java** do, but a little more simply, on one thread, and using the **HttpSocketClient**.

```
private static final String[] URLs =
  { "/index.html", "/logo.gif", "/favicon.ico" };
private static final int HOW_MANY_REQUESTS = 100;
```

- More interesting is that it also wraps the client object in a cache:

```
HttpSocketClient client = new HttpSocketClient();
Retriever cached = Cache.createProxy
  (Retriever.class, client, "retrieve", 0);
for (int t = 0; t < HOW_MANY_REQUESTS; ++t)
  cached.retrieve (URLs[t % URLs.length]);
```

Caching HTTP Responses

EXAMPLE

- If you run the **HttpServer**, and then the **LessStressfulTest**, you'll see that the client completes, very quickly (although even without the cache this is a fairly fast process).
- More tellingly, the log output from the server makes it clear that only three HTTP requests were actually served:

```
Feb 20, 2015 5:15:58 PM cc.sockets.HttpServer init
INFO: HttpServer running on port 80

cc.sockets.HttpServer$RequestHandler run
INFO: Responding to /index.html on thread 11

cc.sockets.HttpServer$RequestHandler run
INFO: Responding to /logo.gif on thread 12

cc.sockets.HttpServer$RequestHandler run
INFO: Responding to /favicon.ico on thread 13
```

- Terminate the server process.

SUMMARY

- The Reflection API is quite powerful, but it can be a bit unwieldy.
- Dynamic proxies, wrapping some of the Reflection API for a specific purpose, are a bit friendlier, even if one still needs some Reflection skills in order to implement them.
- Given those skills, they're easy to whip up as you need them, and the potential for dynamic and more general-purpose logic is tremendous.
- Many of the more sophisticated APIs that you will use, or are already using, in your applications, take advantage of dynamic proxies, somewhere under the hood.

A Dynamic Cache

LAB 11

In this lab you will build a **Cache** class that can provide dynamic proxies that will offer in-memory caching of values returned from certain methods, as keyed by certain arguments. So a client can use your class to improve performance over an interface of its choice, without the **Cache** having to be built with that interface in mind.

Lab project: **Inventory/Step1**

Answer project(s): **Inventory/Step2** (intermediate)
Inventory/Step3 (final)

Files: * to be created
src/cc/retail/Inventory.java
src/cc/retail/InventoryFileImpl.java
src/cc/retail/TestInventoryFileImpl.java
src/cc/retail/InventoryGenerator.java
src/cc/cache/Cache.java *

Instructions:

1. Review the **Inventory** interface, which lets the caller get the quantity of a given part, a batch of quantities for a batch of part numbers, or query for any parts "in short supply," meaning with quantities below a given threshold.
2. The **InventoryFileImpl** class implements **Inventory** by reading a backing data file. The particulars aren't important to the lab, but you may find some of the I/O and utility stream code interesting.
3. Run **TestInventoryFileImpl**, which will instantiate **InventoryFileImpl** and ask the instance to look up a series of part numbers.

```
Part    Quantity
SFLACC      3887
CPREMY        21
SFLACC      3887
OTBASD      6290
```

   The initial test runs against a small data file, and should run very quickly.

4. Run **InventoryGenerator** as a Java application. This will take a moment to create a much larger inventory file – about 5 megabytes in size.

```
Generating inventory/inventory_medium.dat ... done.
```

5. See the new file **inventory_medium.dat** in the **inventory** folder. **inventory_small.dat** is there, too – you just searched this file in your earlier test.

A Dynamic Cache LAB 11

6. Run **InventoryFileImpl** again, now with the program argument "medium" to tell it to look for different values in the bigger file. This will take noticeably longer, and you'll see each requested quantity as each part number is found:

```
Part    Quantity
SYQWTH      8857
CUCNDE      4457
SYQWTH      8857
ATPOOX      6626
YROINP      4562
CUCNDE      4457
```

   Notice that we've pointedly asked for some part numbers more than once – simulating in a small way the likelihood of multiple clients of this utility asking for the same information, or, maybe even more probable, that the same client might come asking for the same information again.

7. Create a new class **Cache<T>** that extends **HashMap<Object,Object>**.
8. Give the class a field **delegate**, of type **T**; a string field **methodName**, and an **int** field **keyIndex**. The idea is that we will create a proxy over type **T**, and when we pass calls to a delegate object, we'll store return values in our map, under a key which is assumed to be one of the arguments passed in the method call, and we'll know which argument by its ordinal position, a/k/a index.
9. Give the class a constructor that takes three parameters with which it can initialize all three of these fields.
10. Make the class implement **InvocationHandler**.
11. Implement **invoke** simply to pass all calls to the **delegate** – exactly as we did at first with the filtering proxy in this chapter's demonstration.
12. In **TestInventoryFileImpl**, change the name of the **inventory** variable to **fileInventory** – and don't use refactoring tools to do so; just change the name and let the existing references to the variable go sour.
13. Initialize a variable **cache**, of type **InvocationHandler**, to a new instance of **Cache<Inventory>**, passing **fileInventory**, "getQuantity", and the index zero, to indicate that the first (and only) parameter to the **getQuantity** method should be treated as a key into the cache.
14. Now initialize a new variable called **inventory**, of type **Inventory**, by calling **Proxy.newProxyInstance**. Pass the class loader (which you can derive from any existing object or from the **TestInventoryFileImpl.class**), an array with just **Inventory.class** in it; and the **cache**.

    The errors in the rest of the code should clean up again at this point.

A Dynamic Cache LAB 11

15. Test again, and you should see no difference – but now your proxy is in play.
16. Back in the **Cache** class, add code to the top of **invoke** to see if the **name** property of the given **Method** equals the cache's configured **methodName**.
17. If it does, then check to see if the cache **containsKey**, passing the element of the given arguments array at the configured **keyIndex**. If this returns **false**, then pass the method call along to the **delegate**, as you do at the bottom of the method – but pass the results as the second argument in a call to **put**, with that key argument as the first argument.
18. Now, whether the value was in the map before or it wasn't, it is now. So either way you can just return the results of a call to **get,** passing the key argument once more.

    So there's your actual caching behavior: you are intercepting calls to the configured method, and assuring that you only call once per value of the configured key argument, and storing the results in the map. So later calls – to that method, with that same key argument – will not be forwarded to the delegate, but answered directly by getting the value back from the map.
19. Anyway, so goes the thinking! Let's try it: run **TestInventoryFileImpl** again, passing "medium" to be sure that you hit a file large enough to make the cache's performance obvious.

    You should see exactly the same output ... but note that the second fetch of any part number now happens right away, instead of after a big delay while the file is searched. This tells you that the cache is indeed working.

```
Part    Quantity
SYQWTH      8857
CUCNDE      4457
SYQWTH      8857 (fast)
ATPOOX      6626
YROINP      4562
CUCNDE      4457 (fast)
```

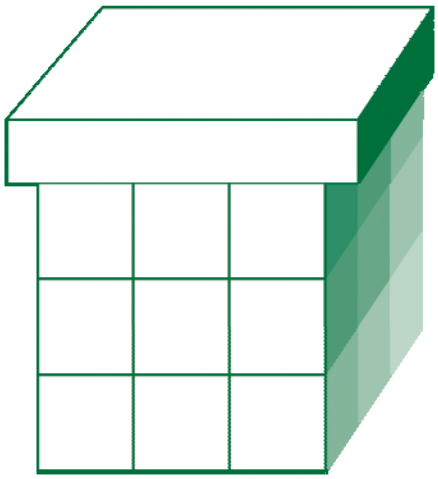
20. You could try for instance cloning the loop at the end of the **main** method, and pasting a second copy in, to make all six queries a second time. Run this and find that the whole second pass happens very quickly.

    This is the intermediate answer in **Cache/Step2**.

    If you like, carry out one last improvement on the **Cache<T>** class, which is a factory method that absorbs the process of calling Proxy.newProxyInstance with appropriate arguments; and adjust **TestInventoryFileImpl** to use this factory method. This enhancement can be seen in the final answer code, in **Inventory/Step3**.

Chapter 12

Annotations



OBJECTIVES

After completing "Annotations," you will be able to:

- Define **aspect-oriented programming** and describe its relationship to Java software.
- Use **annotations** to declare aspects of Java classes, methods, fields, etc.
- Enumerate the built-in annotations in Java SE.
- Use the Reflection API to process annotations on a target class.

Aspect-Oriented Programming

- **In recent years the idea of aspect-oriented programming (or AOP) has challenged the way we think about software design.**
- **Object-oriented ideology tries to address the problem of extreme software complexity by encouraging designers to encapsulate key concepts in classes.**
  - OO seeks to **decouple** the pieces of a larger system into solvable, manageable, maintainable chunks.
  - It defines boundaries around these chunks based on more or less natural marriages of related **state** and **behavior**.
- **AOP takes a different view of the same problem.**
  - It suggests that a complex system can be sliced up in two dimensions: **business logic** and **common services**.
  - Services are basic features of software components that **"cut across"** the business domain boundaries.
  - These features might be as conceptually simple as persistence for a field on a Java class, or as practically complex as transactionality for a suite of methods on an EJB.
- **Though the basic concept of AOP is simple enough, it has encountered significant obstacles to adoption.**
  - It is a fundamental idea requiring native support in programming languages to really succeed.
  - But most languages these days are strongly object-oriented.

AOP and Java

- **AOP and OO can co-exist, and indeed can complement each other nicely.**
  - AOP encourages the declaration (and then the support) of **aspects** on elements of software, so that programmers can focus on business logic and leave common services to runtimes and application servers.
  - OO provides natural organization over which aspects can be declared: classes, operations, attributes, parameters, and other metamodel concepts in OO are perfect homes for aspects.
- **Historically, Java software has flirted with AOP in a number of ways – all of them invented ad-hoc for a particular purpose:**
  - The **transient** keyword is a good example of an aspect.
  - The **synchronized** keyword, as used on a method, is another.
  - Java EE **deployment descriptors** exist primarily to declare aspects of a particular component to its intended container, as in "Component X is an entity bean that should be accessible only to administrators and requires a transaction to be in force when any of its methods is called."
  - JavaBeans' **BeanInfo** system is a means of declaring aspects on a Java component: how to treat various parts of a Java class as properties, methods, and event sources; what sorts of graphical tools to associate with the component for design purposes, etc.

Native Annotations

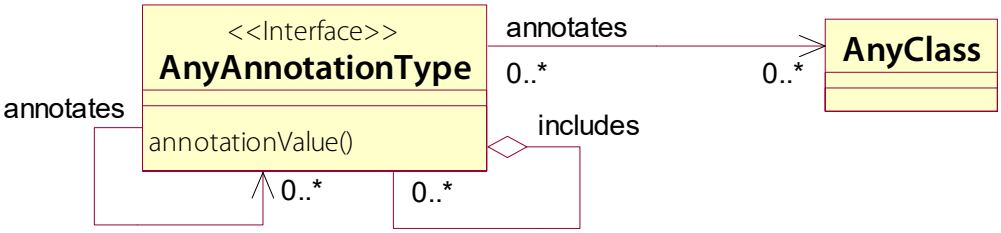
- **Java offers formal AOP support as a native language feature – the annotation.**
- **Annotations declare aspects on elements of an application's type model – packages, classes, methods, fields, and so on.**
  - The character @ is a new token in the Java language, and is used to indicate the use or definition of an annotation.
  - Like ordinary modifiers, annotations precede the things to which they are to be applied:

```
public @deprecated void suspend ();
```

- **The aspects an annotation declares are external to the Java language, meta-model, and Core API.**
  - Annotations are processed by external tools: code generators, deployment tools, validators, documentation generators, etc.
  - There are a handful of **built-in annotations**.
  - Most annotations will be defined at some higher level: for a Java EE standard technology, an open-source framework, or a single application.

The Java Annotations Model

- **Annotations are objects, of a sort.**
  - They have **annotation types**, which are not classes exactly: they are defined using almost exactly the syntax for a Java interface, yet they function more like data structures.
  - Though they use a familiar syntax for **methods**, these methods are really definitions of state elements: they are not behavioral, can't take parameters, but only return values.
- **So there are a few key relationships:**



  - **Annotations** are instances of **annotation types**, and apply to various Java constructs: classes, methods, fields, etc.
  - Annotations can be **complex**, meaning they hold whole instances of other annotation structs within them. This is the "includes" relationship-to-self in the diagram.
  - Annotations can be applied to annotation types! This is called a **meta-annotation** and is represented by the "annotates" relationship-to-self.
- **In this chapter's diagrams, arrows show that an annotation type can be applied to something else.**
  - This isn't quite proper UML, but seems more useful here.

Compiling Annotations

- **Annotation types are defined in Java source files.**
  - In fact they look a fair amount like Java **interfaces**.
  - They exist in the Java type model – they live in **packages**, they have their own **metamodel** that includes methods and complex data structures.
  - The Java source for an annotation must be in the class path for that annotation to be applied in another source file.
- **The programmer defines annotation types and/or applies annotations.**
- **The compiler checks annotations that it encounters, but does not consume them in any meaningful way.**
  - This is something like a **validating parser** asserting that an **XML** document is valid without knowing or caring what the information in the XML document means.
  - As it happens, XML is one of the other major ways to declare aspects! We'll compare Java and XML as AOP languages at the end of this chapter.

Annotation Processing

- Ultimately some other **annotation processor** must consume the programmer's annotations to make them useful.
- This processing can be done at runtime, using enhancements to the Java Reflection API.
  - On java.lang.Class<T>:

```
<A> A getAnnotation (Class<A> annotation);
Annotation[] getAnnotations ();
boolean isAnnotation ();
boolean isAnnotationPresent (Class<A> aClass);
```

  - On java.lang.reflect.AccessibleObject:

```
Annotation[]getDeclaredAnnotations ();
boolean isAnnotationPresent (Class<A> aClass);
```

- It's also possible to process annotations at build time – perhaps to generate new source code that will work with or on behalf of an annotated class at runtime.
  - The JDK comes with a generic processor called **apt**. However this has been deprecated as of JDK 7.
  - As of JDK 6, the tool of choice is the **JSR 269**, Pluggable Annotation Processing API, which is built into the JDK.
  - **javac** has also been enhanced with annotation-processing switches **-processors**, **-proc**, and **-processorpath**.

Annotation Types

- **Define an annotation type by almost the same rules as you would a Java interface.**
  - It must be defined in a source file that bears the same name as the annotation type, with the suffix **.java**.
  - Declare a public type but use the @ token before the keyword **interface**.

```
public @interface MyAnnotationType {}
```

- **Define methods (really attributes) within the body of the annotation type, with no method body (as if they were abstract):**

```
public @interface MyAnnotationType
{
  public int param1 ();
  public String param2 ();
}
```

- **Annotation types are "dumbed down" in many ways:**
  - Neither the type nor its methods may be generic.
  - No **extends** clause is allowed; all annotation types actually declare interfaces that extend **java.lang.annotation.Annotation**.
  - Methods cannot take parameters or declare exceptions.
  - Methods may only return primitive types, **String**s, **Class** objects, other annotation types, or arrays of the preceding.

Annotations

- **Annotations themselves also use the @ token, but now in conjunction with an annotation type name:**

```
@MyAnnotation class MyClass {}
```

- **Informally, there are three sorts of annotation types; these are really just progressive simplifications of the same basic concept and grammar:**
  - The **normal annotation** (just quoting the Java specification, here!) defines multiple values as defined by its type.

```
@MyAnnotation ( param1=5, param2="Summary" )
```

  - The **single-value annotation** defines just one value, and takes advantage of a convention that this value will be defined by a method **value** in the annotation type, and a default method name of **value** in the annotation itself:

```
@OtherAnnotation ( "TheOnlyValueDefined" )
```

  - The **marker annotation** defines no values, and is used as a flag to say that a type construction is a certain thing – if we were to re-invent modifiers such as **transient** and **synchronized**, this is how we would do it.

```
@SwellFellow
```

Annotations

- Annotations can be applied to:
  - **Packages**
  - **Classes** (and hence **enums**)
  - **Interfaces** (hence **annotation types** – the meta-annotation)
  - **Methods** and **constructors**
  - **Fields** (hence **enum constants**)
  - Method **parameters**
  - **Local variables**
- Annotations can be combined in series – just string them along:

```
public @SwellFellow
  @OtherAnnotation ( "AndAnotherThing" )
    void foo ();
```

- However they cannot be used in duplicate; that is, no two annotations on the same target can be of the same annotation type.

```
@MyAnnotation ( param1=5, param2="Good" )
@MyAnnotation ( param1=1, param2="Better" ) //error
public class MyClass
...
```

Built-In Meta-Annotations

- A few annotations are basic to the language.
- Four of them are designed as meta-annotations:

```
@Documented
@Target ({ ElementType.METHOD, ElementType.FIELD })
@Retention ( RetentionPolicy.RUNTIME )
@Inherit
public @interface MyAnnotation { ... }
```

  - **@Documented** means that annotations of this type should be processed as part of javadoc generation.
  - **@Target** defines the array of language elements to which a given annotation type is meant to be applied. Values are things like **PACKAGE** and **FIELD** and are defined in the enumeration **java.lang.annotation.ElementType**.
  - **@Retention** informs the compiler and runtime as to how far an annotation of this type should be propagated: available only from the **SOURCE** file, carried along to the **CLASS** (the default behavior), and/or available to reflection code in the **RUNTIME**. (This is another enum, **java.lang.annotation.RetentionPolicy**.)
  - **@Inherited** marks the annotation type so that annotations of this type on a class are implicitly inherited by its subclasses.
- Find documentation for these in the **java.lang.annotation** package of the Core API.

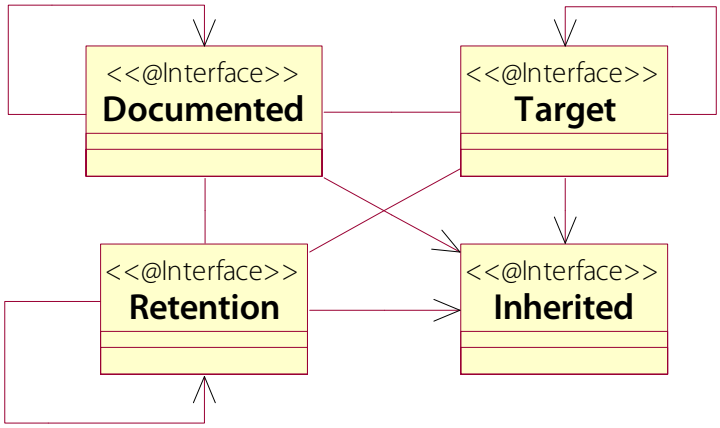
Built-In Meta-Annotations

- The declarations of the built-in meta-annotations are themselves instructive – here's **@Target:**

```
@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Target
```

  - It is to be documented as it occurs.
  - It can be discovered, where it occurs, at all times.
  - It is intended strictly for use as a meta-annotation, since its only stated target is the annotation type.

- All of the built-in meta-annotations, except **@Inherited**, are used on each of the others:



Built-In Annotations

- **Three more are to be applied to other language elements:**
  - **@Override** asks the compiler to apply extra checks to a method that is intended to override something in a base type. If a method so annotated does not in fact override anything – usually because of a mistype in the signature somewhere – the compiler will throw an error where it would not normally.

```
@Override public boolean equals (MyType other);
  // Which would ordinarily slip by the compiler
```

  - **@SuppressWarnings** asks the compiler to ignore conditions it would normally treat as warnings.

```
@SuppressWarnings ({ "unchecked" })
List legacyList = getArrayListOfIntegers ();
```

  - Names given to various warnings, and how they are handled by the compiler, are not themselves standardized facts just yet. So **@SuppressWarnings** does not yet enjoy broad support and is not terribly useful.
  - **@Deprecated** formally flags an element as deprecated. Compilers should produce warnings when they encounter such elements – unless the following annotation is in force:

```
@SuppressWarnings ({ "deprecation" })
```

Legacy Code

EXAMPLE

- In **Legacy** there is an **Algorithms** application that exercises algorithms from the **java.util.Collections** utility.
- It simulates – in a very compressed way – the common relationship between legacy code that derives a non-parameterized or "raw" collection type and Java 5+ code that provides parameterized references.
- The **@SuppressWarnings** annotation is affixed to these declarations to inform the compiler – and anyone reading the code – that the mix of raw and parameterized types is unavoidable.

```
@SuppressWarnings({"unchecked"})
public static void main (String[] args)
{
  ...
}

@SuppressWarnings({"unchecked"})
public static void printList
  (String label, List list)
{
  ...
}
```

Legacy Code

EXAMPLE

- If working from the command line, **compile** this project, and see that **javac** acknowledges this built-in annotation:
- Remove the annotation and try again – or just see the error immediately in the IDE's syntax checker:

```
Note: src\Algorithms.java uses unchecked or unsafe
  operations.
Note: Recompile with -Xlint:unchecked for details.
```

- See the exact warning by doing as the compiler suggests:

```
javac -d build -Xlint src\*.java
src\Algorithms.java:18: warning: [unchecked]
unchecked conversion
found   : java.util.List
required: java.util.Collection
                        <? super java.lang.Integer>
    Collections.addAll (numbers, 1, 2, 3, 4, 5);
                        ^
...
```

Associating Aspects with Java Code

- **Annotations represent one way of attaching aspects to code elements – an inline or compositional approach.**
  - That is, the annotations are inserted into the code itself.
  - They become part of the source file, package, etc.
- **Another important strategy is associative in nature: defining aspects in separate files that declare specific code elements and attach aspects to them from outside the source file.**
  - **Properties files** have been one way of doing this.
  - **XML** is a more powerful language for this process.
- Java EE **deployment descriptors** are the most prevalent example of associating aspects with Java code.
  - XML files name Java classes and methods by a simple convention, and then declare aspects about those elements:

```
<session>
  <display-name>DJ</display-name>
  <ejb-name>DJ</ejb-name>
  <home>cc.music.DJHome</home>
  <remote>cc.music.DJ</remote>
  <ejb-class>cc.music.DJEJB</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
</session>
```

Annotations vs. Descriptors

- **Which strategy is better: annotations or descriptors?**
- **Each has its advantages for certain classes of aspects.**
- **Annotations live right next to their targets, so:**
  - They are **simple** to write and easy to read.
  - They can be very **fine-grained**, attaching even to local variables or method parameters.
  - They become **integral** to the thing being annotated. For instance if a class needs to be refactored from a top-level to an inner class, the same cut-and-paste process that works in general will carry the annotations along in the transition; no need to re-code anywhere else.
  - Thus annotations are generally a good choice for aspects that are themselves integral to the target – fairly basic features that are not likely to change after the target is defined.
- **External descriptors enjoy the opposite benefits as they are decoupled from their targets:**
  - If only the aspects of a code element need to change, the descriptor can be modified **independent** of the source file.
  - Multiple annotations can attach to a target, but descriptors can attach **one aspect to many targets**, or **many to many**.
  - This makes external descriptors a better choice for aspects that are likely to change independently of the source code – hence the Java EE term "deployment descriptor."

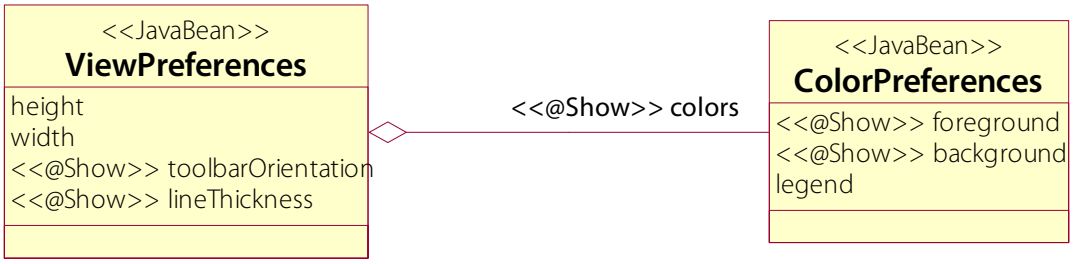
Filtering Data for a Dynamic GUI

LAB 12

Suggested time: 45 minutes

In this lab you will define an annotation type that can be used to flag fields of a complex data structure as appropriate for presentation in a user interface. Imagine a large tree of information that represents user preferences for an application – colors, arrangements of windows, available tools, last files open, etc. – and a preferences dialog that is built dynamically based on names and types of things to edit. This allows the application to grow in complexity without new preference support having to be added in two places, as the GUI will adapt to the underlying data structure as it evolves.

However, not all elements in that data structure should be shown for the user's direct editing: color choices probably should, but window placement probably shouldn't. How to get the best maintainability via the dynamic GUI while showing only the appropriate data? Your new **@Show** annotation will provide the necessary information to the GUI-builder at runtime. You will also add reflection code to detect this annotation; you won't create a full-fledged JFC GUI-builder, but will prove the concept of reading out an arbitrary tree of information, including the presence or absence of your annotation.



Detailed instructions are found at the end of the chapter.

SUMMARY

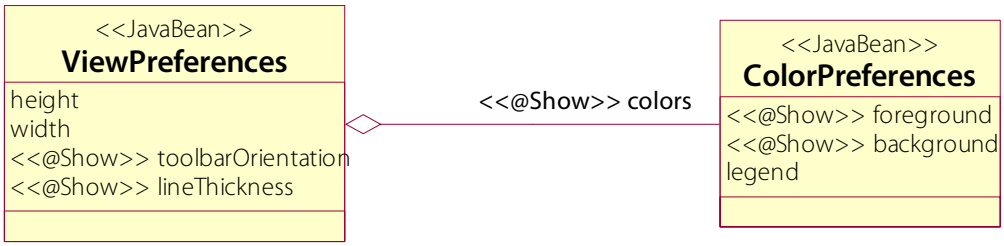
- **Aspect-oriented programming is both revolutionary and strangely familiar.**
  - The idea of organizing common software features as "cross-cutting" aspects is new.
  - But we have been defining various sorts of aspects for many years in the Java world – Java EE especially.
- **Making some form of AOP native to the Java language is a bold move.**
- **It pays off heavily in more recent editions of Java EE.**
  - XML deployment descriptors are a powerful tool, but they have proven to be a bit unwieldy.
  - Bringing some of the most fundamental Java EE aspects into the source code simplifies development tremendously.
- **There is perhaps a danger of overuse.**
  - Aspects are not the right fit for every problem.
  - Annotations are not the right way to express every aspect!
  - One can imagine an over-proliferation of annotations in source code: so many that source files become very difficult to maintain, and a language/meta-language with so many levels to it as to be incomprehensible.

Filtering Data for a Dynamic GUI

LAB 12

In this lab you will define an annotation type that can be used to flag fields of a complex data structure as appropriate for presentation in a user interface. Imagine a large tree of information that represents user preferences for an application – colors, arrangements of windows, available tools, last files open, etc. – and a preferences dialog that is built dynamically based on names and types of things to edit. This allows the application to grow in complexity without new preference support having to be added in two places, as the GUI will adapt to the underlying data structure as it evolves.

However, not all elements in that data structure should be shown for the user's direct editing: color choices probably should, but window placement probably shouldn't. How to get the best maintainability via the dynamic GUI while showing only the appropriate data? Your new **@Show** annotation will provide the necessary information to the GUI-builder at runtime. You will also add reflection code to detect this annotation; you won't create a full-fledged JFC GUI-builder, but will prove the concept of reading out an arbitrary tree of information, including the presence or absence of your annotation.



Lab project: **UIFlags/Step1**

Answer project(s): **UIFlags/Step2** (intermediate)
UIFlags/Step3 (intermediate)
UIFlags/Step4 (final)

Files: * to be created
src/cc/preference/Show.java *
src/cc/preference/ViewPreferences.java
src/cc/preference/ColorPreferences.java
src/cc/reflection/UITree.java

Filtering Data for a Dynamic GUI LAB 12

Instructions:

1. Review the starter code, and note the that **ViewPreferences** and **ColorPreferences** classes form a small information tree.
2. Create the annotation in a new source file **src/cc/preference/Show.java**. Declare it much as you would declare an empty interface: do write a **package** declaration at the top of the file, and just remember to use the **@** prefix before the **interface** keyword.
3. Annotate select fields on **ViewPreferences** and **ColorPreferences** with your new anntation, as shown in the UML diagram on the previous page.
4. Open the **UITree** source file and see that the class is stubbed out for you, with a constructor based on some **Class** that represents the root of the preferences hierarchy. The class extends **TreeMap<String,Object>** so that it can map preference names to either simple types (by way of other **Class** objects) or complex types (by recursive use of **UITree** itself).
5. First, let's get mapping of simple types as pairs of **String**s and **Class** objects. Call **getDeclaredFields** on **rootType**, and iterate over every such **Field** object. For each one, simply **put** the name and type of the field into this map. You'll need to **try** this code against **SecurityException**s; just print an error message in your **catch** block.
6. Test at this point. The **TestUITree** class points a new **UITree** at the **ViewPreferences** type, and then uses a utility method of its own to write out the resulting map. You should see the following output at this stage:

```
colors : cc.preference.ColorPreferences
height : int
lineThickness : int
toolbarOrientation : cc.preference.ViewPreferences$Compass
width : int
```

7. This accurately reflects the fields defined in **ViewPreferences**: names and types. It does not drill down into the **colors** field, though, and without this recursion into **ColorPreferences** we'll never have more than a flat map of names and values. We'll address that a bit later. (This is the intermediate answer in **Step2**.)
8. Now let's start filtering out the things that shouldn't be shown to the user, by detecting the presence of the **@Show** annotation on each tested field. Back in your **UITree** constructor, for each field object, call **isAnnotationPresent**, passing **cc.preference.Show.class** as the argument. Only if this returns true should you **put** the field's information into the map.
9. Test again – what do you get?

Filtering Data for a Dynamic GUI LAB 12

10. Nothing? Hmm. So none of the fields have the annotation declared? Double-check the source file for **ViewPreferences** and confirm that some of the fields are indeed annotated in this way. Why isn't the Reflection API picking up on this?
11. Recall that unless we declare otherwise, the retention policy for our annotation will be **RetentionPolicy.CLASS**, and that this means the annotation is carried into the class' bytecode, but is not loaded into the runtime representation of the class. Thus reflection code cannot "see" it.
12. Modify **Show.java** to define the retention policy to be **RUNTIME**. While you're at it, clarify that the annotation is only for fields. You will need to import a few things from the **java.lang.annotation** package for these new annotations to compile.

```
@Target (FIELD)
@Retention (RUNTIME)
public @interface Show {}
```

13. Build and test again, and you should now see proper filtering – your annotation is in effect!

```
colors : cc.preference.ColorPreferences
lineThickness : int
toolbarOrientation : cc.preference.ViewPreferences$Compass
```

    This is the intermediate answer in **UIFlags/Step3**.

14. Finally, let's extract the full tree of information in the **UITree** constructor. If the annotation is present, instead of just automatically **put**ting the name and type into the map, open a code block and start by capturing the type as a **Class<?>** called **fieldType**.
15. Test two things: if **fieldType.isPrimitive** or the **SIMPLE_TYPES** collection contains **fieldType**, then you can treat it as a simple type – this is where your pre-existing call to **put** the name and **fieldType** can go.
16. Otherwise, though, you'll consider this to be a complex type, and recursively process it by creating a new **UITree** object, passing **fieldType** to the constructor. Now **put** that in the current map as the value for this field's name, instead of the **fieldType** itself.
17. Test this, and you should now see a tree of information, filtering by the **@Show** annotation at each level. (This is the final answer in **Step4**.)

```
colors
  background : java.awt.Color
  foreground : java.awt.Color
lineThickness : int
toolbarOrientation : cc.preference.ViewPreferences$Compass
```