Java Interfaces
Conceptual Framework

• A firm conceptual foundation for understanding and designing modern software recognizes:
  – Modern software consists of potentially large numbers of components that are composed into “larger” components/systems
  – In Java, one ordinarily thinks of a class as a component, with its client-visible specification in the interface(s) it implements
An interface contains a description of what software does.

A firm conceptual foundation for understanding and designing modern software recognizes:

- Modern software consists of potentially large numbers of components that are composed into “larger” components/systems.
- In Java, one ordinarily thinks of a class as a component, with its client-visible specification in the interface(s) it implements.
Conceptual Framework

A firm conceptual foundation for understanding and designing modern software recognizes:

- Modern software consists of potentially large numbers of components that are composed into “larger” components/systems
- In Java, one ordinarily thinks of a class as a component, with its client-visible specification in the interface(s) it implements

An interface contains a description of what software does.

A class contains a description of how the software does it.
Anatomy of an Interface

/**
 * An informal Javadoc comment.
 * [additional Javadoc comments as needed; in our
 * case: the mathematical model, contract(s) for
 * constructor(s), optional contract for iterator]
 */

public interface I {

    // contract(s) for method(s)
}

6 April 2016 OSU CSE
Anatomy of an Interface

/**
 * An informal Javadoc comment.
 * [additional Javadoc comments as needed; in our
 * case: the mathematical model,
 * constructor(s), optional constructor/iterator]
 */

public interface I extends I1, I2 {

    // contract(s) for method(s)

}
Anatomy of an Interface

/**
 * An informal Javadoc comment.
 * [additional Javadoc comments as needed; in our
 * case: the mathematical
 * constructor(s), optional contract for Iterator]
 */

public interface I<T> extends I1<T>, I2 {

    // contract(s) for method(s)
}

An interface may be **generic**, as seen here; formal parameter T may have any name, but T is typical (for “type”).
Technicalities

- Interfaces may be used to define:
  - instance methods
  - static final variables ("constants")
  - Not constructors
  - Not static methods
  - Not instance variables
Technicalities

• Interfaces may be used to define:
  – *instance methods*
  – *static final variables* ("constants")
  – Not constructors
  – Not static methods
  – Not instance variables

All instance methods in an interface are automatically *public abstract*. 
Technicalities

• Interfaces may be used to define:
  – *instance methods*
  – *static final variables* (“constants”)
  – Not constructors
  – Not static methods
  – Not instance variables

 Variables are automatically `public static final`; it is unusual for an interface to define variables.
Technicalities

• Interfaces may be used to define:
  – *instance methods*
  – *static final variables* ("constants")
  – Not constructors
  – Not static methods
  – Not instance variables

A constructor always has the name of a class, and is not an instance method.
Technicalities

• Interfaces may be used to define:
  – *instance methods*
  – *static final variables* ("constants")
  – Not constructors
  – Not static methods
  – Not instance variables

As of Java 8, interfaces allow static methods *with their implementations*. But providing code in interfaces goes against our use of interfaces for client views (this biases designs to instance methods).
Technicalities

• Interfaces may be used to define:
  – *instance methods*
  – *static final variables* (“constants”)
  – Not constructors
  – Not static methods
  – Not instance variables

This is a good thing: instance variables are not client-view information!
Example: Queue Component Family

```
QueueKernel
  Standard
  extends

Queue
  Iterable
  extends
```

Example: **Queue** Component Family

![Diagram showing the inheritance relationships between Queue, QueueKernel, Standard, and Iterable]

Which methods are in **Standard**, and why?
Example: **Queue Component Family**

We call the “core interface” in this design style the **kernel interface**.
Example: *Queue* Component Family

- Standard
- QueueKernel
- Queue

We call the “most powerful interface” in this design style the *enhanced* interface.
Let’s Examine...

- Standard
- QueueKernel
- Queue

Ask about details, e.g.:
- Design choices (math model, constructor, kernel methods, other methods)
- Javadoc (compare code in the Java interface with the generated documentation)
Interface Design

• The **kernel** interface defines:
  – The mathematical model for the type
  – Contract(s) for the constructor(s)
  – Contract(s) for kernel methods
  – Contract(s) for methods inherited from Java library interfaces that do not have their own contract specifications (if applicable; e.g., an iterator or a comparator)

• The **enhanced** interface defines contracts for all other methods for the type
Kernel Design Guidelines

• Kernel methods generally should be:
  – A *minimal set* of methods that is *functionally complete*, i.e., powerful enough to:
    • Give a variable of the type any allowable value
    • Determine the value of a variable of the type
  – Powerful enough to allow a *client* to:
    • Implement `equals` and `toString`
    • Check every kernel method’s precondition
Kernel Design Guidelines

• Kernel methods generally should be:
  – A *minimal set* of methods that is *functionally complete*, i.e.
    • Give a variable of the type any allowable value
    • Determine the value of a variable of the type
  – Powerful enough to allow a client to:
    • Implement `equals` and `toString`
    • Check every kernel method’s precondition

“Minimal” means if any proposed kernel method were left out, you could not satisfy one of the completeness tests—though sometimes an “extra” kernel method is so parsimonious it is included in the kernel anyway!
public interface QueueKernel<T> extends Standard<Queue<T>>, Iterable<T> {
    ...
}

public interface Queue<T> extends QueueKernel<T> {
    ...
}
Notice: Circularity

public interface QueueKernel<T> extends Standard<Queue<T>>, Iterable<T> {
    ...
}

Using Queue<T> is necessary because it is the return type of newInstance and the interface type we always use to declare Queue variables.

public interface Queue<T> extends QueueKernel<T> {
    ...
}
What “Mentions” What?

Standard

“mentions”

QueueKernel

“mentions”

Iterable

“mentions”

Queue

“mentions”
Interface = Type

• Java permits the circularity here because:
  – A Java interface defines a type, i.e., the type name (which we consider to denote a set of mathematical model values that certain variables might have) along with the set of its instance methods
  – An interface type may be used in Java even if there is no implementation of it in scope
Interfaces: Only At Compile-Time

- Interfaces are a *compile-time* construct
  - Used by the Java compiler for *type-checking* with declared types: to make sure variables are used only where they make sense
    - Recall the rules for declared types and object (dynamic) types
  - Once a Java program compiles, only object types are kept at *run-time*
    - Declared types literally “disappear” in the JVM
• The declared type of a variable *may be* an interface type
• The object type *can never be* an interface type, because you *may not* instantiate a variable using `new` followed by an interface type
• If the declared type is an interface type, then the object type (a class) *must* implement the declared type (an interface)

are used where they make sense

  • Recall the rules for declared types and object (dynamic) types

    – Once a Java program compiles, only object types are kept at *run-time*

      • Declared types literally “disappear” in the JVM
Javadoc Tags

• Recall that the short one-sentence informal overview, and the following standard Javadoc tags, are “expected”:
  - @param
  - @return
Javadoc Tags

- Recall that the short one-sentence informal overview, and the following standard Javadoc tags, are “expected”:
  - @param
  - @return

Checkstyle warns you if they are missing when they should be there.
Custom Javadoc Tags

- Javadoc supports *custom tags* that can be introduced to document various aspects not considered when Javadoc was introduced
  - Example: *contracts*
# OSU CSE Custom Contract Tags

<table>
<thead>
<tr>
<th>Custom Tags for Interfaces</th>
<th>Custom Tags for Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>@mathsubtypes</td>
<td>@convention</td>
</tr>
<tr>
<td>@mathdefinitions</td>
<td>@correspondence</td>
</tr>
<tr>
<td>@mathmodel</td>
<td></td>
</tr>
<tr>
<td>@initially</td>
<td></td>
</tr>
<tr>
<td>@iterator</td>
<td></td>
</tr>
</tbody>
</table>
# OSU CSE Custom Contract Tags

## Custom Tags for Constructors and Methods
- @aliases
- @updates
- @replaces
- @clears
- @requires
- @ensures
- @decreases

## Custom Tags for Loops
- @maintains
- @decreases
Packages

• Each OSU CSE component family is bundled into its own package, i.e., a grouping of interfaces and classes that the designer thinks “belong together” for logical reasons
  – Example: the Queue-family components are all in the package components.queue
Packages Are Useful

• A package provides:
  – Logical structuring: packages are hierarchical, i.e., you may have packages within packages
  – A namespace: units in different packages may have the same name without conflict
    • See also import statements
  – Another level of access control between public and private
Packages Are Useful

• A package provides:
  – Logical structuring: packages are hierarchical, i.e., you may have packages within packages
  – A namespace: units in different packages may have the same name without conflict
• See also import statements
– Another level of access control between public and private

But any apparent “control” is easily circumvented and illusory, so we do not recommend using it.
Package Declaration

```java
class components.queue;
import components.standard.Standard;
public interface Queue<T> ...
```

- A Java file must be in a file system directory matching the package name
  - Eclipse handles this correspondence for you
- At most one package declaration per file
  - If there is no package declaration, interface/class is in unnamed default package
package components.queue;
import components.standard.Standard;
public interface Queue<T> ...

• A Java file must be in a file system directory matching the package name
  – Eclipse handles this correspondence for you
• At most one package declaration in a file
  – If there is no package declaration, interface/class is in unnamed default package

This line declares that the interface (or class) in the current file belongs to the package components.queue.
package components.queue;
import components.standard.Standard;
public interface Queue<T> ...

• A Java file must be in a file system directory matching the package name
  – Eclipse handles this correspondence for you
• At most one package declaration in a file
  – If there is no package declaration, interface/class is in unnamed default package

This line brings the interface Standard into scope, from the package components.standard.
Resources

• *Big Java Late Objects*, Section 9.6: Interface Types

• javadoc — *The Java API Documentation Generator*
  – http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html