Implementing the Standard Methods
Loose Ends

• In implementing several kernel interfaces so far, you have been given code in the skeletons for the Standard methods.

• The code for these methods is very stylized and easy to adapt to a new situation, even if the code itself is hardly transparent!

  – Several new Java issues arise ...
newInstance

TF newInstance()

• Returns a new object with the same dynamic type as this, having an initial value. If the type TF has a no-argument constructor, then the value of the new returned object satisfies its contract. Otherwise, the value of the new returned object satisfies the contract of the constructor call that was used to initialize this.

• Ensures:

   is_initial(newInstance)
newInstance

TF newInstance()  
• Returns a new object with the same dynamic type as this, having an initial value. If the type TF has a no-argument constructor, then the value of the new returned object satisfies its contract. Otherwise, the value of the returned object satisfies the contract of the constructor call that was used to initialize this.  
• Ensures:  
  \textit{is\_initial}(newInstance)  

Throughout this set of slides, TF stands for the type family interface whose kernel you are implementing; it is not a generic type parameter!
newInstance(TF newInstance())

- Returns a new object with the same *dynamic* type as *this*, having an initial value. If the type TF has a no-argument constructor, then the value of the new returned object satisfies its contract. Otherwise, the value of the new returned object satisfies the contract of the constructor call that was used to initialize *this*.

- Ensures:

  \[ is\_initial(newInstance) \]
Implementing `newInstance`

- Java already offers a method to make a new instance that is “like” an existing object, but it is messy to call it
- The reason for the `Standard` method `newInstance` is simply to make it easy to call Java’s method for doing this, by wrapping the mess inside a method body
- The same code works for any type, so you can just copy the code that follows
Code for `newInstance`

```java
@SuppressWarnings("unchecked")
@Override
public final Queue<T> newInstance() {
    try {
        return this.getClass()
             .getConstructor().newInstance();
    } catch (ReflectiveOperationException e) {
        throw new AssertionError(
            "Cannot construct object of type ",
            + this.getClass());
    }
}
```
Code for `newInstance`

```java
@SuppressWarnings("unchecked")
@Override
public final Queue<T> newInstance() {
    try {
        return this.getClass()
            .getConstructor().newInstance();
    }
    catch (ReflectiveOperationException e) {
        throw new AssertionError(
            "Cannot construct object of type 
                    + this.getClass());
    }
}
```

The return type is whatever type family interface you are implementing; `Queue<T>` is shown here for definiteness.
Code for `newInstance`

```java
@SuppressWarnings("unchecked")
@Override
public final Queue<T> newInstance() {
  try {
    return this.getClass()
            .getConstructor().newInstance();
  }
  catch (ReflectiveOperationException e) {
    throw new AssertionError("Cannot construct object of type "+
                              this.getClass());
  }
}
```

This tells the compiler not to issue a warning about “unchecked conversion” ...
Code for `newInstance`

```java
@ SuppressWarnings("unchecked")
@ Override
public final Queue<T> newInstance() {
    try {
        return this.getClass()
            .getConstructor().newInstance();
    }
    catch (ReflectiveOperationException e) {
        throw new AssertionError(
            "Cannot construct object of type "+ this.getClass());
    }
}
```

... which it would otherwise warn you about on this statement (even though it cannot cause any trouble).
Code for newInstance

@SuppressWarnings("unchecked")
@Override
public final Queue<T> newInstance() {
    try {
        return this.getClass().getConstructor().newInstance();
    } catch (ReflectiveOperationException e) {
        throw new AssertionError("Cannot construct object of type 
                                + this.getClass()");
    }
}

The try/catch block and exceptions are Java constructs to be explained later in the course.
clear

void clear()

• Resets this to an initial value. If the type TF has a no-argument constructor, then this satisfies its contract. Otherwise, this satisfies the contract of the constructor call that was used to initialize #this.

• Clears: this
void clear()

- Resets this to an initial value. If the type TF has a no-argument constructor, then this satisfies its contract. Otherwise, this satisfies the contract of the constructor call that was used to initialize #this.

- Clears: this

So, the parameter mode **clears** means what it says above, i.e., *is_initial* is true for that parameter.
Implementing `clear`

• Because this code has to do the same thing as the no-argument (or maybe some other) constructor, it is a good idea to factor out the code that initializes the instance variables into a separate private method.

• For the no-argument constructor (usual case):

```java
private void createNewRep() {
    // initialize instance variables
}
```
Code for `createNewRep`

- Again, code from `Queue2<T>` is shown:

```java
private void createNewRep() {
    this.preFront = new Node();
    this.preFront.next = null;
    this.rear = this.preFront;
    this.length = 0;
}
```
Code for `clear`

```java
@Override
class final void clear() {
    this.createNewRep();
}
```
Code for `clear`

```java
@Override
public final void clear() {
    this.createNewRep();
}
```

If there isn’t a no-argument constructor, then `createNewRep` needs some parameters (like a constructor with parameters); see a `SortingMachine` implementation for an example.
transferFrom

void transferFrom(TF source)

• Sets this to the incoming value of source, and resets source to an initial value; source must have the same dynamic type as this. If the type TF has a no-argument constructor, then source satisfies its contract. Otherwise, source satisfies the contract of the constructor call that was used to initialize #source.

• Replaces: this
• Clears: source
• Ensures:
  
  \[ this = #source \]
Implementing `transferFrom`

- This code simply copies the values of all the instance variables from `source` to `this`, and then re-initializes `source`
  - For an instance variable of a reference type, the reference value is copied—but aliases are eliminated before `transferFrom` returns
@Override

public final void transferFrom(Queue<T> source) {
    assert source != null : "Violation of:
    + " source is not null";
    assert source != this : "Violation of:
    + " source is not this";
    assert source instanceof Queue2<?> : "" + "Violation of: source is of dynamic" + " type Queue2<?>";
    ...
}
@Override
public final void transferFrom(Queue<T> source) {
    assert source != null : "Violation of: source is not null";
    assert source != this : "Violation of: source is not this";
    assert source instanceof Queue2<?>> : "Violation of: source is of dynamic type Queue2<??>";
    ...
}
Code (Pt 1) for `transferFrom`

```java
@Override
public final void transferFrom(Queue<T> source) {
    assert source != null : "Violation of:
    + " source is not null";
    assert source != this : "Violation of:
    + " source is not this";
    assert source instanceof Queue2<?> :
    + "Violation of: source is of dynamic
    + " type Queue2<??>>
    ...
}
```

First two `assert` statements check for normal problems: `null` and repeated arguments.
@Override
public final void transferFrom(Queue<T> source) {
    assert source != null : "Violation of: source is not null";
    assert source != this : "Violation of: source is not this";
    assert source instanceof Queue2<?> : "Violation of: source is of dynamic type Queue2<?>";
    ...
}
Because of a problem in Java called **type erasure**, you can check only that the dynamic type is `Queue2<?>`, meaning “Queue2 of something”; more on this later.
Code (Pt 2) for `transferFrom`

```java
@Override
public final void transferFrom(Queue<T> source) {
    ...
    Queue2<T> localSource = (Queue2<T>) source;
    this.preFront = localSource.preFront;
    this.rear = localSource.rear;
    this.length = localSource.length;
    localSource.createNewRep();
}
```
Code (Pt 2) for `transferFrom`

```java
@Override
public final void transferFrom(Queue<T> source) {
    ...
    Queue2<T> localSource = (Queue2<T>) source;
    this.preFront = localSource.preFront;
    this.rear = localSource.rear;
    this.length = localSource.length;
    localSource.createNewRep();
}
```

This *cast* cannot fail since the `assert` above would have stopped execution in that case; so, `source` *must be* of dynamic type `Queue2<?>`, and the `? must be T` or the call would not have compiled.
The Cast and Why It Works Here

Some abstract value of type Queue<T>
The compiler thinks things look like this picture because of the declared types of `this` and `source`, shown below.
The initialization of `localSource` claims that the **dynamic** type of `source` is `Queue2<T>`, as shown here—and it is!
How `transferFrom` Works

Start here ...
How `transferFrom` Works

... update this ...
How `transferFrom` Works

- `localSource`
- `source`
- `preFront`
- `rear`
- `length`
How `transferFrom` Works

... and return.
To check your understanding: could this code be used in place of the last line shown in the body?

```
source.clear();
```

```java
@Override
public final void transferFrom(Queue<T> source) {
    ...
    Queue2<T> localSource = (Queue2<T>) source;
    this.preFront = localSource.preFront;
    this.rear = localSource.rear;
    this.length = localSource.length;
    localSource.createNewRep();
}
```