The “Common” Methods from 

*Object*

equals, hashCode, toString
Methods of `Object`

- Recall that every Java class implicitly extends the class `Object`, which defines the following methods and provides default implementations that you should override:
  
  ```java
  boolean equals(Object obj)
  int hashCode()
  String toString()
  ```
- Recall that every Java class implicitly extends the class `Object`, which defines the following methods and provides default implementations that you should override:

  ```java
  boolean equals(Object obj)
  int hashCode()
  String toString()
  ```

Default implementation of `equals`:
“for any non-null reference values `x` and `y`, this method [i.e., `x.equals(y)`] returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).”
• Recall that every Java class implicitly extends the class `Object`, which defines the following methods and provides default implementations that you should override:

  ```java
  boolean equals(Object obj)
  int hashCode()
  String toString()
  ```

Default implementation of `hashCode`:
“typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.”
• Recall that every Java class implicitly extends the class Object, which defines the following methods and provides default implementations that you should override:

  ```java
  boolean equals(Object obj)
  int hashCode()
  String toString()
  ```

  Default implementation of `toString`:
  “returns a string consisting of the name of the class of which the object is an instance, the at-sign character ‘@’, and the unsigned hexadecimal representation of the hash code of the object.”
Why Override The Defaults

• For `equals` consider the default:

```java
NaturalNumber n1 = new NaturalNumber2();
NaturalNumber n2 = new NaturalNumber2();
boolean b = n1.equals(n2);
```

• What is the value of `b` with the default implementation of `equals`?
Why Override The Defaults

• For `hashCode` consider the default:

```java
NaturalNumber n1 = new NaturalNumber2();
NaturalNumber n2 = new NaturalNumber2();
Set<NaturalNumber> s = new Set4<>();
s.add(n1);
boolean b = s.contains(n2);
```

• What is the value of `b` with the default implementation of `hashCode`?
Why Override The Defaults

• For `hashCode`

```java
NaturalNumber n1 = new NaturalNumber2();
NaturalNumber n2 = new NaturalNumber2();
Set<NaturalNumber> s = new Set4<>();
s.add(n1);
boolean b = s.contains(n2);
```

• What is the value of `b` with the default implementation of `hashCode`?

Recall: `Set4` uses `hashCode` to organize elements into “buckets”, and calls `equals` to decide whether an element is in a bucket.
Why Override The Defaults

• For `toString` consider the default:
  ```java
  NaturalNumber n = new NaturalNumber2();
  String s = n.toString();
  ```

• What is the value of `s` with the default implementation of `toString`?
Why Override The Defaults

• For `toString`

```java
NaturalNumber n = new NaturalNumber2();
String s = n.toString();
```

• What is the value of `s` with the default implementation of `toString`?

You can’t say exactly, but it’s something like:

"NaturalNumber2@9A6F49D0"
The Crux of the Problem

- **Object** cannot possibly know *anything* about the *abstract mathematical model values* (i.e., the object values) of variables

- **equals**, **hashCode**, and **toString** should *all* behave in ways that depend on these abstract mathematical model values
The Idea Behind the Solution

• Recall that we implement these methods in abstract class **XYZSecondary**
  – To write code that deals with abstract mathematical model values (object values), you should *layer* the code for **equals**, **hashCode**, and **toString** on top of the **kernel** methods from interface **XYZKernel**
An OSU CSE Component Family

- Standard
- XYZKernel
- XYZ
- XYZSecondary
- XYZ1
- XYZ2

- implements
- extends
- extends
- extends
- extends
Layer implementations of XYZSecondary methods by calling methods from XYZ (including those from XYZKernel).
Warning: When you call XYZ methods from here, beware of mutual recursion: these methods may call each other!
An OSU CSE Component Family

Note: If you call *only* `XYZKernel` methods from here, then there is no potential problem with mutual recursion.
Implementing `equals(Object obj)`

- There are a couple ways to approach this, but **best practice** suggests a multi-step “filtering” approach that first weeds out special cases
Implementing `equals(Object obj)`

- There are a couple ways to approach this, but **best practice** suggests a multi-step “filtering” approach that first weeds out special cases.

Note that the parameter to `equals` is an `Object`! Your code is supposed to **override** the default implementation of `equals`, so its signature must match this one **exactly** or your code will instead **overload** `equals`. 
Implementing `equals(Object obj)`

- Step 1: If the reference values of `obj` and `this` are equal (i.e., they are aliases), then their object values are equal.
  - Why is this true?
Implementing `equals(Object obj)`

- Step 1: If the reference values of `obj` and `this` are equal (i.e., they are aliases), then their object values are equal.
  
  – Why is this true?

Ordinarily, we would consider this a **repeated argument violation**. But for `equals`, repeated arguments must be allowed—and dealt with according to the (informal but clear) description of `equals` found in `Object`.
Implementing `equals(Object obj)`

- Step 2: If `obj` is null, then its object value does not equal the object value of `this`.
  - Why is this true?
Implementing `equals(Object obj)`

• Step 3: If the types of `obj` and `this` are not the same, then their object values are not equal.
  – Why is this true?
Implementing `equals(Object obj)`

- Step 3: If the types of `obj` and `this` are not the same, then their object values are not equal.
  - Why is this true?

This is far from obvious! Consider:

```java
Stack<Integer> s = new Stack1L<>();
Queue<Integer> q = new Queue1L<>();
What do we want `s.equals(q)` to return?
Implementing `equals(Object obj)`

- Step 3: If the types of `obj` and `this` are not the same, then their object values are not equal.
  - Why is this true?

Even though \( s = <> \) and \( q = <> \) (i.e., their mathematical model types are both `string of integer` and both have the math value `<>`), in Java we must choose to say these two variables’ values are not equal.
Implementing `equals(Object obj)`

- Step 3: If the types of `obj` and `this` are not the same, then their object values are not equal.
  - Why is this true?

If you think about this question very carefully, you’ll find that there is really no choice here: it would be impossible to implement `equals` to do the right thing if we decided that only the mathematical model types mattered.
“... types ... are not the same”?

• So, what does this mean?
  – Declared types?
  – Dynamic (object) types?
“... types ... are not the same”?

• So, what does this mean?
  – Declared types?
  – Dynamic (object) types?

Declared types are a compile-time notion in Java, so in the body of `equals` it seems we cannot check whether the declared types of `this` and `obj` are the same.
“... types ... are not the same”?

- So, what does this mean?
  - Declared types?
  - Dynamic (object) types?

If we based the answer on *dynamic (object) types* being the same, then an \texttt{XYZ1} variable could never be reported equal to an \texttt{XYZ2} variable; not a disaster, but we can do better...
Good News About Java

• Java gives us a way to check a slightly more general interpretation of “types are not the same”, which is better for this purpose than checking whether dynamic types are the same

• We can ask whether a variable’s dynamic type implements a particular interface, e.g., XYZKernel; or extends a particular class, e.g., XYZSecondary
To tell whether `this` and `obj` might possibly be equal, we can ask whether their dynamic types both implement `XYZ`; e.g., one might be an `XYZ1` and the other an `XYZ2`, and this is OK!
The `instanceof` Method

- This code implements Step 3 in the `equals` method:
  ```java
  if (!(obj instanceof XYZ)) {
    return false;
  }
  ```
The `instanceof` Method

• This code implements Step 3 in the `equals` method:

```java
if (!(obj instanceof XYZ)) {
    return false;
}
```

Since `XYZ` is an interface, this code checks whether the dynamic type of `obj` (a class) `implements` `XYZ`. And because this code for `equals` is in the class `XYZSecondary`, we know that the dynamic type of `this` implements `XYZ`. 
The `instanceof` Method

- This code implements Step 3 in the `equals` method:

```java
if (!(obj instanceof XYZ)) {
    return false;
}
```

Note that, for this particular component design, we could just as well ask:

- `obj instanceof XYZKernel`
- `obj instanceof XYZSecondary`
What About Generics?

• The previous code works fine when $XYZ$ is something like `NaturalNumber`, but what if it’s a generic type like `Queue<T>`?

  – The answer to this question reveals a serious problem with Java (compared to similar languages, e.g., C#), so bear with us through a couple twists and turns...
Type Erasure

• For generic types, the JVM keeps track of the \textit{raw type} (e.g., `Queue1L` or `Queue2`) of each variable as its dynamic type, but does not keep track of any generic type parameters (e.g., for this variable $T$ is `Integer`, for that variable $T$ is `String`)

• This mechanism is called \textit{type erasure}
  – Effectively (but not technically), the type parameter is replaced by `Object`
Some Consequences

• Built-in arrays of parametric types cannot be created, so this is a compile-time error:

\[
T[] \text{ myArray} = \text{new } T[50];
\]

• In code like `equals`, we cannot distinguish between related but (we would like to think, different) dynamic types like:

\[
\text{Queue1L<Integer> Queue1L<String>}
\]
**instanceof** With Generics

• This code implements Step 3 in the `equals` method if `XYZ` is a generic type with one parameter:

```java
if (!(obj instanceof XYZ<?>)) {
    return false;
}
```
instanceof With Generics

• This code implements Step 3 in the \texttt{equals} method if \texttt{XYZ} is a generic type with one parameter:

\begin{verbatim}
if (!(obj instanceof XYZ<?>)) {
    return false;
}
\end{verbatim}

This checks whether \texttt{obj} has a dynamic type that implements \texttt{“XYZ of unknown”}, which is all the JVM knows about.
Filter Code for Special Cases

```java
if (obj == this) {
    return true;
}
if (obj == null) {
    return false;
}
if (!(obj instanceof XYZ<?>)) {
    return false;
}
```
Filter Code for Special Cases

```java
if (obj == this) {
    return true;
}
if (obj == null) {
    return false;
}
if (!(obj instanceof XYZ<?>)) {
    return false;
}
```

What comes next?
What do we know here?
Casting

• We now know that `obj` and `this` are:
  – Not aliases
  – Not null
  – Have the same raw type `XYZ` (of something)

• We may now cast `obj` from its declared type `Object` to the most specific dynamic type we know it must have: `XYZ<?>`
The Rest of `equals`

```java
XYZ<?> x = (XYZ<?>) obj;
```

Now, the compiler would allow us to call `XYZ` methods on `x` and `this`. However, as mentioned before (i.e., to avoid concern about possible mutual recursion), we limit ourselves to using `XYZKernel` methods to determine whether their mathematical model values are equal.
The Rest of \texttt{equals}

\texttt{XYZ<?> x = (XYZ<?>>) obj; }

This code is like other code you’ve written before for \textit{layered} implementations that call kernel methods, except for one thing: \texttt{this} is of type \texttt{XYZ<T>}, but \texttt{obj} (hence \texttt{x}) might be of type \texttt{XYZ<T1>}!
Example: Queue

Queue<?> q = (Queue<?>) obj;
if (this.length() != q.length()) {
    return false;
}
Iterator<T> it1 = this.iterator();
Iterator<?> it2 = q.iterator();
while (it1.hasNext()) {
    T x1 = it1.next();
    Object x2 = it2.next();
    if (!x1.equals(x2)) {
        return false;
    }
}
return true;
Example: Queue

Queue<?> q = (Queue<?>) obj;
if (this.length() != q.length()) {
    return false;
}
Iterator<T> it1 = this.iterator();
Iterator<?> it2 = q.iterator();
while (it1.hasNext()) {
    T x1 = it1.next();
    Object x2 = it2.next();
    if (!x1.equals(x2)) {
        return false;
    }
}
return true;

If $q$ and $\text{this}$ are not the same length, then they are not equal.
Example: Queue

```java
Queue<?> q = (Queue<?>>) obj;
if (this.length() != q.length()) {
    return false;
}
Iterator<T> it1 = this.iterator();
Iterator<?> it2 = q.iterator();
while (it1.hasNext()) {
    T x1 = it1.next();
    Object x2 = it2.next();
    if (!x1.equals(x2)) {
        return false;
    }
}
return true;
```

We can use `iterator` here because it is a kernel method for `Queue`. 
Example: **Queue**

```java
Queue<?, ?> q = (Queue<?, ?>) obj;
if (this.length() != q.length()) {
    return false;
}
Iterator<T> it1 = this.iterator();
Iterator<?> it2 = q.iterator();
while (it1.hasNext()) {
    T x1 = it1.next();
    Object x2 = it2.next();
    if (!x1.equals(x2)) {
        return false;
    }
}
return true;
```

Iterate over both and ask, one by one, if the corresponding entries are equal.
Example: Queue

Queue<?> q = (Queue<?>) obj;
if (this.length() != q.length()) {
    return false;
}
Iterator<T> it1 = this.iterator();
Iterator<?> it2 = q.iterator();
while (it1.hasNext()) {
    T x1 = it1.next();
    Object x2 = it2.next();
    if (!x1.equals(x2)) {
        return false;
    }
}
return true;

We know the entries of this are of type T, but the only thing we can say about the entries of q is that they are Objects (because everything is).
Example: Queue

```java
Queue<?> q = (Queue<?>) obj;
if (this.length() != q.length()) {
    return false;
}
Iterator<T> it1 = this.iterator();
Iterator<?> it2 = q.iterator();
while (it1.hasNext()) {
    T x1 = it1.next();
    Object x2 = it2.next();
    if (!x1.equals(x2)) {
        return false;
    }
}
return true;
```

Note that we may call equals here with the argument `x2`, because (recall) equals accepts any `Object` as the argument.
Example: Queue

```java
Queue<?> q = (Queue<?>) obj;
if (this.length() != q.length()) {
    return false;
}
Iterator<T> it1 = this.iterator();
Iterator<?> it2 = q.iterator();
while (it1.hasNext()) {
    T x1 = it1.next(),
    Object x2 = it2.next();
    if (!x1.equals(x2)) {
        return false;
    }
}
return true;
```

Fortunately, the first call to equals for type T will return false if the types of the first entries of q and this are not the same!
Queue<?> q = (Queue<?>) obj;
if (this.length() != q.length()) {
    return false;
}
Iterator<T> it1 = this.iterator();
Iterator<?> it2 = q.iterator();
while (it1.hasNext()) {
    T x1 = it1.next();
    Object x2 = it2.next();
    if (!x1.equals(x2)) {
        return false;
    }
}
return true;

If the loop finishes checking all corresponding entries (i.e., all are reported equal), then q (hence obj) and this have the same mathematical model value.
Two Remaining Problems

• What if the loop doesn’t execute at all because $q$ and this are both empty (but with different entry types)?
  – Sadly, this code reports they are equal, even though they are not even of the same type
  – There is no apparent way around this in Java!

• What happens with an “unordered” math model type, e.g., Set or Map?
  – A slightly bigger mess, but it can be handled
equals, compareTo, compare

• Recall the interfaces:
  – Comparable (with the method compareTo)
  – Comparator (with the method compare)

• Informal documentation says “strongly recommended but not strictly required”:
  \[(x.\text{compareTo}(y) == 0) == x.\text{equals}(y)\]
  \[(\text{compare}(x, y) == 0) == x.\text{equals}(y)\]

• Don’t believe it: for all practical purposes, these properties are required
**hashCode and toString**

- Fortunately, these are far more straightforward than `equals` because the only parameter is `this`, and we know exactly what type it is.

- In each case, it’s just like writing code (layered on the *kernel*) for other methods or application software, as you have done throughout this course sequence.
Caveats About `hashCode`

- If `x.equals(y)` is `true`, then `x.hashCode() == y.hashCode()` must be `true` – But not vice versa; think about it
- `hashCode` should be “fast”, so you generally do not want to process all entries in a (possibly large) collection to compute its `hashCode`
Resources

• *Effective Java, Second Edition*, Item 8 (which is valid as far as it goes)