Lecture 24

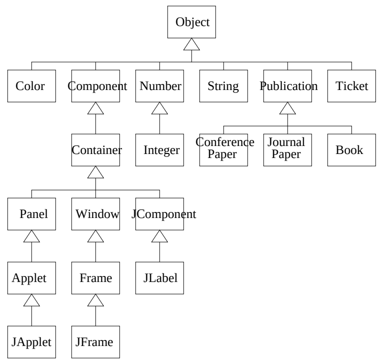
The topic of this lecture is Java's class hierarchy.

We discuss the class `Object`: the root of the Java class hierarchy.

We look at the methods that all classes inherit from the `Object` class. This deepens our understanding of how Java prints objects (using `toString`) and how it compares them for equality (using `equals`).

We also revisit the distinction between reference types and primitive types. We note that primitive types can have corresponding classes using Java's wrapper classes.

Part of the class hierarchy



(Note, each class has only one parent: *single inheritance*)

The `Object` class

- There is one class, `Object`, that *every* other class is a direct or indirect subclass of
- So, when there is no explicit `extends` clause, implicitly you are saying `extends Object`
- When you are defining your own classes, you need to consider how objects will be given a string representation, compared for equality, given a hash code, copied and disposed of
- `Object` already contains default definitions for all these, which will be inherited
- However, the defaults often don't do what you want, and so will need to be overridden

`Object`'s interface

- `public String toString()`
  Returns a string representation of this object
- `public boolean equals(Object anObject)`
  Returns true iff parameter is equal to this object
- `public int hashCode()`
  Returns a hash code value for this object
  - Hash codes are integers used as keys in hash tables
  - If two objects are equal according to `equals`, then calling `hashCode` on them should produce the same integer
  - So, if you override `equals`, you should override `hashCode` (or avoid Java's hash tables)

`Object`'s interface, cont'd

- `protected void finalize()`
  Called exactly once by the garbage collector
  - The default definition in `Object` does nothing and is rarely overridden
- `protected Object clone()`
  Creates and returns a copy of this object
  - Assigns contents of all instance variables into the new object (*shallow copy*)
  - Sometimes override it to invoke `clone` on referenced objects (*deep copy*)
  - (Complicated by `Cloneable` interface and `CloneNotSupportedException`)
- Others include `getClass()` (returns the class of the object) and `writeObject/readObject`

`toString`: the true story

- `println` automatically invokes `toString`
- Every class has a definition of `toString` because...
  - `Object` includes a `toString` method, which will be inherited
  - But this version of `toString` simply causes the reference ('pointer') to be printed
  - So we override it with a definition that returns a more meaningful string
- E.g. add to `Publication`

```
public String toString()
{  return author.getFullName() + " " +
      title + " " + yearOfPub;
}
```

Equality testing: the true story

- `==` compares 'pointers'; to do something other than this, use a method, e.g. one called `equals`
- Every class has a definition of `equals` because...
  - `Object` includes an `equals` method, which will be inherited
  - But this version of `equals` simply compares 'pointers' (i.e. it does the same as `==`)
  - So we override it with a definition that does a more meaningful comparison
- Coming up with a proper definition to override `equals` is complicated

Example definition of `equals` for `Publication`

```
public boolean equals(Object anObject)
{  if (this == anObject)
   {  return true;
   }
   else if (anObject == null ||
      getClass() != anObject.getClass())
   {  return false;
   }
   Publication other =
                  (Publication) anObject;
   return (author.equals(other.author) &&
      title.equals(other.title) &&
      yearOfPub == other.yearOfPub);
}
```

It compares each instance variable for equality. (This requires that `PersonName` has an `equals` definition; it uses `String`'s `equals` definition; and it uses `==` for `yearOfPub` because this has a primitive type)

Where do `int`, etc. fit into the hierarchy?

- Reference types: `byte, short, int, long, float, double, boolean` and `char`
- Values of these types are not objects; they are not part of the class hierarchy
- These values have direct machine representations, so we don't need class definitions; they are 'primitives' (built into the language)
- Arrays are objects, but they are built-into Java, so there is no class definition and they are also not part of the class hierarchy (but they do support all of `Object`'s methods)

Wrapper classes

- Some parts of a program might only accept objects. Java provides *wrapper classes* so that 'primitives' can be dressed up as objects: `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` and `Character`
- E.g. the value 3 (an `int`) can be operated on by, e.g. `+`, `-`, `*`, etc. But you can't send it any messages (it's not an object)
- But you can create an object from it:

  ```
  Integer obj = new Integer(3);
  ```

- And you can re-extract the `int` from the object by sending a message:

  ```
  int i = obj.intValue();
  ```


Primitive types	**Reference types**
Examples	
`int, double, boolean, char`	arrays, `String`, `Color`, `Integer`, `Fraction`, `Publication`
Contents	
Contains a value	Contains a reference ('pointer') to an object
System.out.println(...)	
Displays the value	Runs `toString` • `toString` inherited from `Object` displays the pointer • override `toString` to display something more meaningful
Assignment	
Copies the value	Copies the reference (hence two pointers to same object)

Primitive types	**Reference types**
Equality testing	
`==` tests for equal values	• `==` tests for equal references (pointing to same object) • `equals`, inherited from `Object`, is like `==` • override `equals` or define your own method to compare the values of the instance variables
Passing as a parameter to a method	
Copies the value, so changes made in the method do not affect the original	Copies the reference (hence two pointers to same object), so changes to the object made in the method do affect the original