The Object Class and Wrapper Clases

The Object Class

Outline

- The Object Class
    - Methods common to all objects
    - `equals`
    - `toString`
- Wrapper Classes

Method common to all objects

- Although `Object` is a concrete class, it is designed primarily for extension.
- All of its ***non-final*** methods have explicit general contracts because they are designed to be overridden.
  - `protected Object clone() throws CloneNotSupportedException;`
  - `public boolean equals(Object obj);`
  - `public int hashCode();`
  - `public String toString()`
- It is the responsibility of any class overring these methods to obey their general contracts; failure to do so will prevent other classes that depend on the contracts (such as HashMap and HashSet) from functioning properly in conjunction with the class.

Method	Description
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public boolean equals(Object obj)	compares the given object to this object.
public int hashCode()	returns the hashcode number for this object.
public String toString()	returns the string representation of this object.

Two kinds of object equality…

- Often in Java programs you need to compare two objects to determine if they are equal or not. It turns out there are two different kinds of equality one can determine about objects in Java, ***reference equality*** or ***logical equality***.
- the equality operator (==) compares the references (addresses in memory) of the two variables- this is known as ***reference equality***.
- ***Logical equality*** compares the data of the objects instead of the value of the references.

When not to override `equals`

- The `equals` for class `Object` implements the most discrimination possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns true if and only if `x` and `y` refer to the same object.(***reference equality***)
- Overriding the `equals` method seems simple, but there are many ways to get it wrong. The easiest way to avoid problems is not to override the `equals` method:
  - When we don't care whether the class provides a "content equality"(***logical equality***) test.
  - When a superclass has already overridden `equals` and the superclass behavior is appropriate for this class.
  - When the class is private or package-private, and you are certain that its `equals` method will never be invoked.

How to override equals

- We override `equals` when a class has a notion of content equality that differs from mere object identity, and a superclass has not already overridden equals to implement the desired behavior. This is common for value classes, such as Rational numbers.

```
public class Rational{
...
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true; // an optimization
        else if (obj instanceof Rational) {
            Rational y = (Rational)obj;
            return (this.num == y.num) && (this.denom == y.denom);
        } else return false;
    }
}
```

```java
public class TestObject {

  public static void main(String[] args) {

    Rational r1 = new Rational(3, 7);
    Rational r2 = new Rational(4, 5);
    Rational r3 = r1;
    Rational r4 = new Rational(3,  7);
    System.out.println(r1.equals(r2));
    System.out.println(r1.equals(r3));
    System.out.println(r1.equals(r4));

    Complex c1 = new Complex(10,15);
    Complex c2 = new Complex(3.4,1.5);
    Complex c3 = new Complex(10.0,15.0);
    Complex c4 = c3;
    System.out.println(c1.equals(c2));
    System.out.println(c3.equals(c4));
    System.out.println(c1.equals(c3));

  }

}
```

```java
public class Rational {
private int num, denom;

public Rational(int num, int denom) {
this.num = num;
this.denom = denom;
}

@Override
    public boolean equals(Object obj) {
        if (this == obj) return true; // an optimization
        else if (obj instanceof Rational) {
            Rational y = (Rational)obj;
            return (this.num == y.num) && (this.denom == y.denom);
        } else return false;
    }
}
```

Always override `toString`

- Providing good `toString` implementation makes your class more pleasant to use.
- The `toString` method is automatically invoked when an object is passed to `println`, `printf`, the string concatenation operator, or assert, or printed by a debugger.
- When practical, the `toString` method should return all of the interesting information contained in the object.
- Provide programmatic access to all of the information contained in the value returned by to `toString`. If you fail to do this, you force the programmers who need this information to parse the string
- Example: `toString` of the Rational Class

```java
public class TestObject {

public static void main(String[] args) {

Rational r1 = new Rational(3, 7);
Rational r2 = new Rational(4, 5);
Rational r3 = r1;
Rational r4 = new Rational(3,  7);
System.out.println(r1.oldToString());
System.out.println(r1.toString());

Complex c1 = new Complex(10,15);
Complex c2 = new Complex(3.4,1.5);
Complex c3 = new Complex(10.0,15.0);
Complex c4 = c3;
System.out.println(c1.oldToString());
System.out.println(c1.toString());

}

}
```

```java
public class Complex {
  private double re, im;

  public Complex (double re, double im) {
    this.re = re;
    this.im = im;
  }

  @Override
  public String toString() {
    return String.format(re + " +i" + im);
  }

  public String oldToString() {
    return super.toString();
  }
}
```

The Wrapper Class

Need of Wrapper Classes

- They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
- The classes in `java.util` package handles only objects and hence wrapper classes help in this case also.
- Data structures in the Collection framework, such as `ArrayList` and `Vector`, store only objects (reference types) and not primitive types.
- An object is needed to support synchronization in multithreading.

```java
ArrayList<int> myNumbers = new ArrayList<int>(); // Invalid
```

```java
ArrayList<Integer> myNumbers = new ArrayList<Integer>(); // Valid
```

Wrapper Classes

The table below shows the primitive type and the equivalent wrapper class:

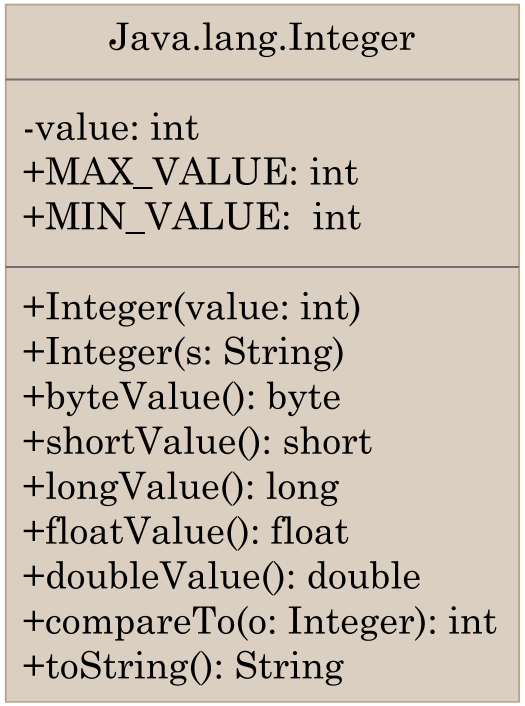
Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

NOTE

- The wrapper classes do not have no-arg constructors.
- The instances of all wrapper classes are immutable, i.e., their internal values cannot be changed once the objects are created.

The `Integer` and `Double` Classes

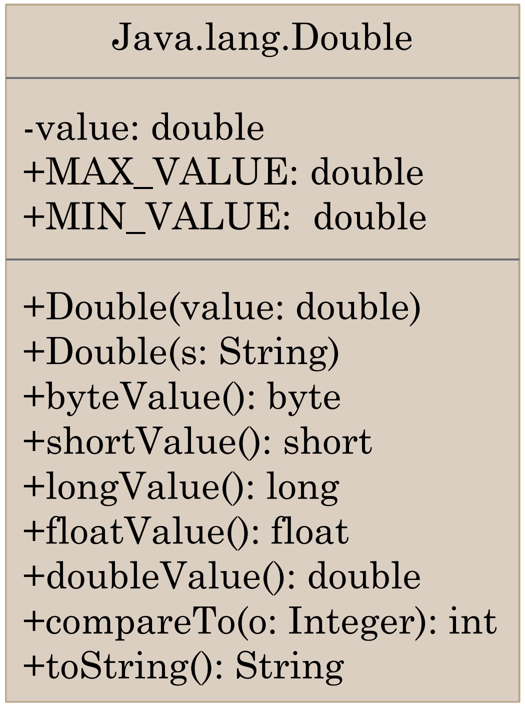
- Each wrapper class overrides the toString, equals, and hashCode methods defined in the Object class.



Java.lang.Integer

-value: int
+MAX_VALUE: int
+MIN_VALUE: int

+Integer(value: int)
+Integer(s: String)
+byteValue(): byte
+shortValue(): short
+longValue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Integer): int
+toString(): String



Java.lang.Double

-value: double
+MAX_VALUE: double
+MIN_VALUE: double

+Double(value: double)
+Double(s: String)
+byteValue(): byte
+shortValue(): short
+longValue(): long
+floatValue(): float
+doubleValue(): double
+compareTo(o: Integer): int
+toString(): String

https://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html

The `Integer` and `Double` Classes

- Constructors
- Class Constants `MAX_VALUE, MIN_VALUE`
- Conversion Methods

Numeric Wrapper Class Constructors

- You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value. The constructors for Integer and Double are:

```
// The constructor Integer(int/String) is deprecated
Integer myInt = new ~~Integer(34);~~
Integer myInt = new ~~Integer("34");~~

Double mydouble = new ~~Double(5.6);~~
Double mydouble = new ~~Double("5.6");~~
```

The Static `valueOf` Methods

- The numeric wrapper classes have a useful class method:
  - `valueOf(String s)`
- This method creates a new object initialized to the value represented by the specified string. For example:

```
Integer myInt1 = Integer.valueOf("34");

Integer myInt2 = Integer.valueOf(34);


Double mydouble1 = Double.valueOf(5.6);
Double mydouble2 = Double.valueOf("5.6");
```

Numeric Wrapper Class Constants

- Each numerical wrapper class has the constants `MAX_VALUE` and `MIN_VALUE`.
- `MAX_VALUE` represents the maximum value of the corresponding primitive data type.
- `MIN_VALUE` represents the minimum value of the corresponding primitive data type.
- For `Float` and `Double`, `MIN_VALUE` represents the minimum positive `float` and `double` values.

```
System.out.println(myInt.MAX_VALUE);
System.out.println(myInt.MIN_VALUE);

System.out.println(mydouble.MAX_VALUE);
System.out.println(mydouble.MIN_VALUE);
```

Conversion Methods

- Each numeric wrapper class implements the abstract methods doubleValue, intValue, longValue, and shortValue, which defined in the Number class. These methods “convert” objects into primitive type values.

```
int primiInt = myInt.intValue();

double primiDouble = myInt.doubleValue();
```

Automatic Conversion between Primitive Types and Wrapper Class Types

- Converting a primitive value to a wrapper object is called *boxing*. The reverse conversion is called *unboxing*.
- The compiler will automatically box a primitive value that appears in a context requiring an object, and will unbox an object that appears in a context requiring a primitive value.

```
Integer IntObj = new Integer(34);
Integer IntObj = 2;     //auto-boxing


Integer IntObj = Integer.valueOf("34");
int primiInt = myInt;   //auto-unboxing
```