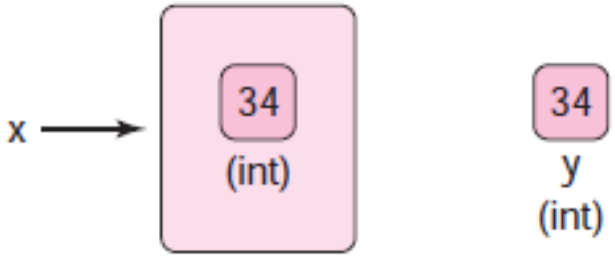
Wrapper Classes

Wrapper Class

- A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.



Reference variable *x* refers to an *Integer* object; *y* is the name of a primitive variable. Both hold the value 34.

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 Array List and Vector, store only objects (reference types) and not primitive types.
- An object is needed to support synchronization in multithreading.

Wrapper Classes in JAVA

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

Properties of Wrapper Classes

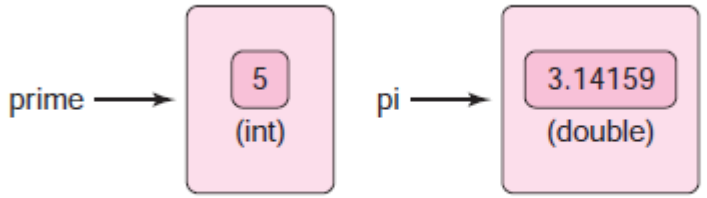
- A wrapper class comes packaged with constructors. Except for the Character class, each wrapper class has two constructors.
  - Each numeric class (Integer, Long, Short, Byte, Double, and Float) has a one-argument constructor that accepts an argument of the corresponding primitive type.
    - Integer(int value) //constructor
    - Integer integer = new Integer(4); //initialization
  - Each wrapper class, except Character , has a second constructor that accepts a String argument.
    - Double x = new Double ("234.56");
    - Integer y = new Integer("12345");
    - Boolean z = new Boolean("true");

Continue..

- The Character class has a single constructor:
    - Character(char ch);
- Surprisingly, the wrapper classes have no default or 0-argument constructors. So, the statement
    - Integer x = new Integer(); // ILLEGAL Integer has no default constructor
- generates a compilation error.

Auto-boxing and unboxing

- The automatic conversion of a primitive type to its corresponding wrapper (reference) type is called *automatic boxing* or simply *autoboxing*. Similarly, the conversion of a wrapper object to its corresponding primitive type is called *automatic unboxing* or *unboxing*. Thus, converting from int to Integer is *autoboxing* and from Integer to int, *unboxing*.
    - Integer prime = 5;
    - Integer x = 5; // or Integer x new Integer(5). Note that *x* is a reference.
    - int y = x; // Notice that *x* is an object reference and *y* is a primitive.



An *Integer* object and a *Double* object; *prime* and *pi* are references

Wrappers inherit and Wrappers Implement

- The wrapper classes override equals(...) and toString() so that equals(...) compares the *values* inside two wrapper objects, and toString() returns the value of a wrapper object as a String reference.
    - Integer x = new Integer(5);
    - Integer y = new Integer(5);
    - System.out.println(x.equals(y)); // compares *values* not references
    - System.out.println(x.toString());
- All wrapper classes, except Boolean, implement the Comparable interface.

Wrappers and Expressions

- Conveniently, references to wrapper objects can be used in arithmetic expressions.
    - Integer x = 10; // x, y, and z are *references* not primitives;
    - Integer y = 20;
    - Integer z = x * y; // Is this multiplication of references?
  - Although x and y are indeed references, the expression x * y is evaluated as follows:
  - Variable x is unboxed and its primitive value (10) retrieved.
  - Variable y is unboxed and its primitive value (20) retrieved.
  - The value 10 * 20 = 200 is calculated.
  - A new Integer object with value 200 is instantiated, boxed, and referenced by z
- As you can see from this seemingly innocuous code segment, using wrapper references in an arithmetic expression incurs a bit of processing overhead.
- “Classes are very convenient when a method requires an object, however, when performing basic arithmetic, opt for primitives.”

Wrapper Objects are Immutable

- Like String objects, an object belonging to a wrapper class is immutable.
- Once a wrapper object has been instantiated, its value cannot be changed. Of course, this does not mean that a *reference* to a wrapper object cannot be reassigned.

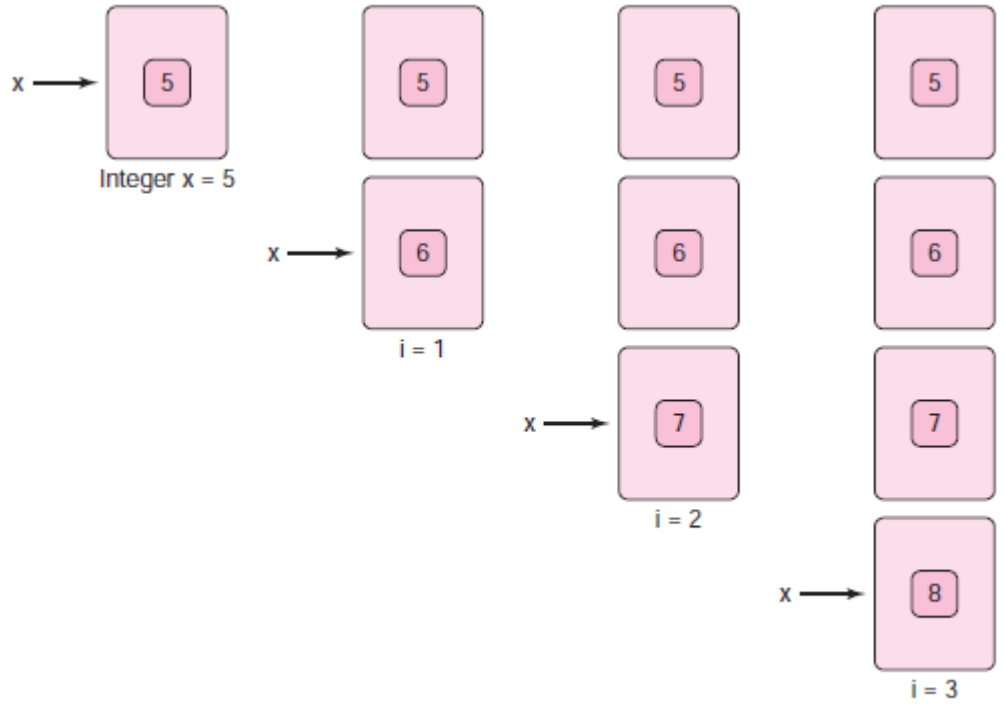


FIGURE 14.5 Wrapper objects are immutable

Continue..

- The wrapper classes override the equals(Object o) method inherited from Object so that a.equals(b) compares *values* and not references. In contrast, the operator compares references. No unboxing takes place.
    - Integer x = new Integer(5);
    - Integer y = new Integer(5); // a second object is instantiated
    - System.out.println(x == y);
  - prints
    - false.
- However, it may surprise you that the segment
    - Integer x = 5;
    - Integer y = 5;
    - System.out.println(x == y);
  - prints
    - true;

Continue..

- In the second case, because Integer objects are immutable, Java deems it unnecessary to create two distinct objects with the value 5. So, in fact, the references x and y both refer to the same object. By not creating two separate objects, the compiler saves memory. Java does this for integer values between 128 and 127, inclusive. If we change the value in the two preceding assignment statements to 555 , then x y evaluates to false. Although autoboxing blurs the line between primitives and wrappers, an Integer is not an int , and an int is not an Integer . Use autoboxing cautiously.

Some Useful Methods

- The wrapper classes also implement a number of handy static methods.

Method	return type	Description	Example
Integer.valueOf(String s)	Integer	Returns reference to an Integer object initialized to the numeric value of s	Integer x = Integer.valueOf("345");
Double.valueOf(String s)	Double	Returns a reference to a Double object initialized to the numeric value of s	Double x = Double.valueOf("3.14159");
Integer.parseInt(String s)	int	Returns the numeric value of s as a primitive	int x = Integer.parseInt("345");
	double	Returns the numeric value of s as a primitive	double x = Double.parseDouble("3.14159");
Integer.toString(int x)	String	Returns the integer x as a String	String s = Integer.toString(123);
Double.toString(double x)	String	Returns the double x as a String	String s = Double.toString(3.14159);

FIGURE 14.6 Some static methods of the *Double* and *Integer* classes. Similar methods are defined for *Byte*, *Long*, and *Float*.

Continue..

Method	return type	Description	Example
Character.isDigit(char ch)	boolean	Returns true if ch is a digit	Character.isDigit('w') returns false
Character.isLetter(char ch)	boolean	Returns true if ch is a letter	Character.isLetter('w') returns true
Character.isLettorOrDigit(char ch)	boolean	Returns true if ch is a letter or a digit	Character.isDigit('$') returns false
Character.isLowerCase(char ch)	boolean	Returns true if ch is a lower case letter	Character.isLowerCase('w') returns true
Character.isUpperCase(char ch)	boolean	Returns true if ch is an uppercase letter	Character.isUpperCase('w') returns false
Character.isWhitespace(char ch)	boolean	Returns true if ch is a blank, a tab, a form feed, or a line separator	Character.isWhitespace('x') returns false
Character.toLowerCase(char ch)	char	Returns the lowercase version of ch if ch is an alphabetical character, otherwise returns ch	Character.toLowerCase('a') returns 'A' Character.toLowerCase('#') returns '#'
Character.toUpperCase(char ch)	char	Returns the uppercase version of ch if ch is an alphabetical character, otherwise returns ch	Character.toUpperCase('r') returns 'R' Character.toUpperCase('#') returns '#'

FIGURE 14.7 A few static methods of the *Character* class

Continue..

- In addition to the static methods of the wrapper classes, each wrapper class (except Boolean) defines two static constants, MIN_VALUE and MAX_VALUE, that represent the largest and smallest value of the corresponding primitive type.
- For example, Integer.MAX_VALUE is 2147483647, Integer.MIN_VALUE is -2147483648, and Byte.MAX_VALUE is 127.