Iterating objects

Comparing objects

Autoboxing, Unboxing

Exercises with collections

Iterators

- Iterators are individual objects for each collection object.
- Iterators are used to traverse elements through the collection.

```
ArrayList list = new ArrayList();
for(Iterator it = list.iterator(); it.hasNext();)
{
    Object obj = it.next();
    //do something with the object
}

ArrayList cars = new ArrayList();
for(Iterator it = cars.iterator(); it.hasNext();)
{
    //casting is needed, because next() returns Object
    Car car = (Car) it.next();
    //do something with the object
}
```

Checks availability of a next element

Getting the iterator object for this collection

Returns the next element in the collection

More on Iterators

- By using this iterator object, you can access each element in the collection, one element at a time.
- In general, to use an iterator to cycle through the contents of a collection, follow these steps:
  - Obtain an iterator to the start of the collection by calling the collection's **iterator( )** method.
  - Set up a loop that makes a call to hasNext( ). Have the loop iterate as long as **hasNext( )** returns true.
  - Within the loop, obtain each element by calling **next( ).**

Comparing

- Java objects can be compared
  - Equals() or „==“ return the equality of a reference but not the logical equality meant by the programmer
- Two types of comparing in Java

Comparable

- Comparable = java.lang.Comparable
  - Requires overriding of compareTo(Object o1)
  - CompareTo returns (by Convention)
    - -1 if o1 > this;
    - 0 if o1 == this;
    - 1 if o1 < this

```
public abstract class Car implements Comparable<Car>
{

    public abstract int getMaxSpeed();

    @Override
    public int compareTo(Car otherCar)
    {
        if(this.getMaxSpeed() > otherCar.getMaxSpeed())
            return 1;
        else
        if(this.getMaxSpeed() < otherCar.getMaxSpeed())
            return 1;
        return 0;
    }
}
```

Comparator

- Comparator = java.util.Comparator
  - Requires overriding of compare(Object o1, Object o2)
  - Compare returns (by Convention)
    - 1 if o1 > o2
    - 0 if o1 == o2
    - -1 if o1 < o2

```
public class CarComparator implements Comparator<Car>
{

    @Override
    public int compare(Car car1, Car car2)
    {
        if(car1.getMaxSpeed() > car2.getMaxSpeed())
            return 1;
        else
        if(car1.getMaxSpeed() < car2.getMaxSpeed())
            return 1;
        return 0;
    }

}
```

Primitive types and classes

- In Java we have primitive types
  - boolean, char, byte, short, int, long, float, double
  - The primitive types are not classes (but something more primitive)
    - No methods, no constructors, no inheritance, etc.
- Each primitive type has a wrapper class
  - Boolean, Character, Byte, Short, Integer, Long, Float, Double

Autoboxing

- **Autoboxing** is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed.

- **Auto-unboxing** is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.

Autoboxing

- From Java 5.0 on we don't need to explicitly convert from primitive type to wrapper type.
- Example
  - List<Integer> myList = new ArrayList<Integer>();
  - myList.add(47); // legal: 47 is automatically boxed
  - int a = myList.get(0); // legal: a is automatically unboxed
- Result
  - Fewer lines
  - Cleaner code

Autoboxing and methods

- Autoboxing automatically occurs whenever a primitive type must be converted into an object.
- Auto-unboxing takes place whenever an object must be converted into a primitive type.
- Autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method.

Autoboxing and Expressions

- Whenever we use object of Wrapper class in an expression, automatic unboxing and boxing is done by the JVM

```
Integer iOb;
iOb = 100;          //Autoboxing of int
++iOb;
```

- When we perform increment operation on Integer object, it is first unboxed, then incremented and then again reboxed into Integer type object.
    - This will happen always, when we use Wrapper class objects in expressions or conditions etc.

When autoboxing falls short

- Integer can have a null value, int cannot
  - Unboxing may result in a NullPointerException
    - Integer a = null;
    - Int res = 45 + a; // throws NullPointerException
- == has different semantics (meaning)
  - int b=3, c=3;
  - Integer d=new Integer(3), e = new Integer(3);
  - b == c true
  - d == e false
  - d.equals(e) true
  - b.equals(c)
    - Does not compile: b is primitive and cannot be deferenced.

Performance

- Autoboxing works well with collections.
- Don’t use a lot of autoboxing in fast-running aplications
  - scientific calculations, gaming, etc.
  - It takes too much time.
  - Try to stay with the primitive types
    - They are faster than the wrappers.
  - Don’t do this at home (in a fast-running application)
    - Integer a, b;
    - Integer result = a + b;
      - Unboxes a and b. Executes plus. Boxes result.
  - This is what you should do
    - int a, b;
    - int result = a + b;
      - No autoboxing

Example

```
class Test
{
 public static void main(String[] args)
 {
  Integer iob = 100;       //Autoboxing of int
  int i = iob;           //unboxing of Integer
  System.out.println(i+" "+iob);

  Character cob = 'a';        /Autoboxing of char
  char ch = cob;             //Auto-unboxing of Character
  System.out.println(cob+" "+ch);
 }
}
```

```
Output :
100 100
a a
```

Tasks

1. Write a program that fills Employee objects into a Company. The company has a name and a collection of Employees separated in different departments. Each employee has name, age, salary and an ID number. Departments are known by their name only. Write a demo that creates the company and adds employees into different departments. Then list all employees that the company has. The output of the program should be a list of departments and a sublist of employees for each department.
2. Write a program that sorts the Employee objects in the company getter:
    - Based on highest salary;
    - Based on their names alphabetically;
    - Based on their age.

Tasks

3. Write a method in the above program that takes the collection of employees by department as an argument and returns a collection of all employees that work in the company. The collection must be sorted by their names.
3. Write a method in the above program that eliminates the duplicate Employees. Duplicate employees have identical names and age.
4. Add functionality that sets salaries for a particular month for each Employee. Add a method that prints the salaries for each month.
5. Write a program that reads a piece of programing code and tells if the curly braces “{” and “}” are all put. This means that each “{” should have a corresponding “}”