Advanced Programming Techniques in Java

COSI 12B

Sets

Lecture 23

Class Objectives

- Sets (Section 11.2)

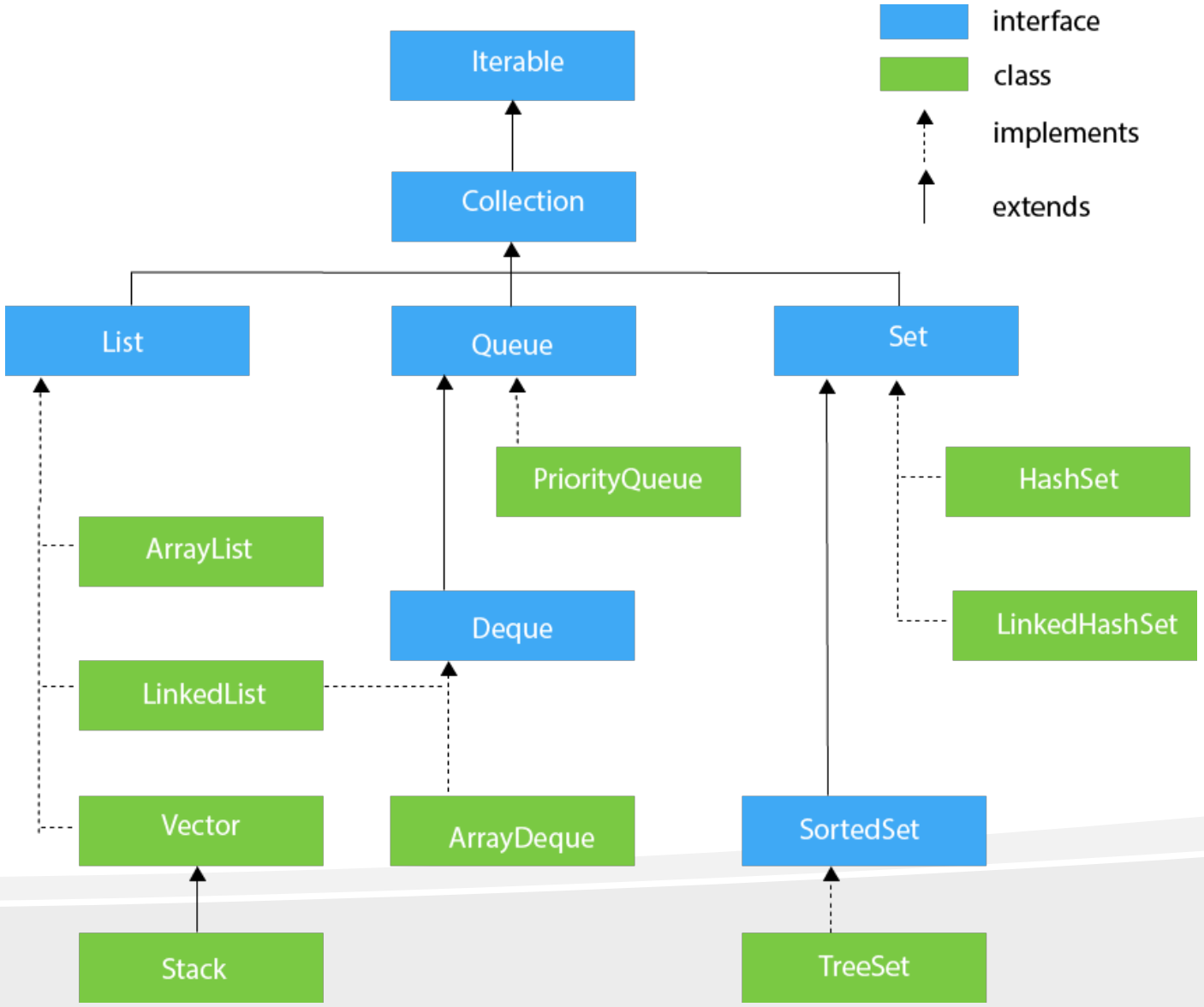
Review: Specification of the Stack Abstract Data Type

- Only the top element of a stack is visible; therefore, the number of operations performed by a stack are few
- We need the ability to
  - test for an empty stack (empty)
  - inspect the top element (peek)
  - retrieve the top element (pop)
  - put a new element on the stack (push)

Methods	Behavior
`boolean empty()`	Returns **`true`** if the stack is empty; otherwise, returns **`false`**.
`E peek()`	Returns the object at the top of the stack without removing it.
`E pop()`	Returns the object at the top of the stack and removes it.
`E push(E obj)`	Pushes an item onto the top of the stack and returns the item pushed.

Sets

Review: Collections Framework Diagram

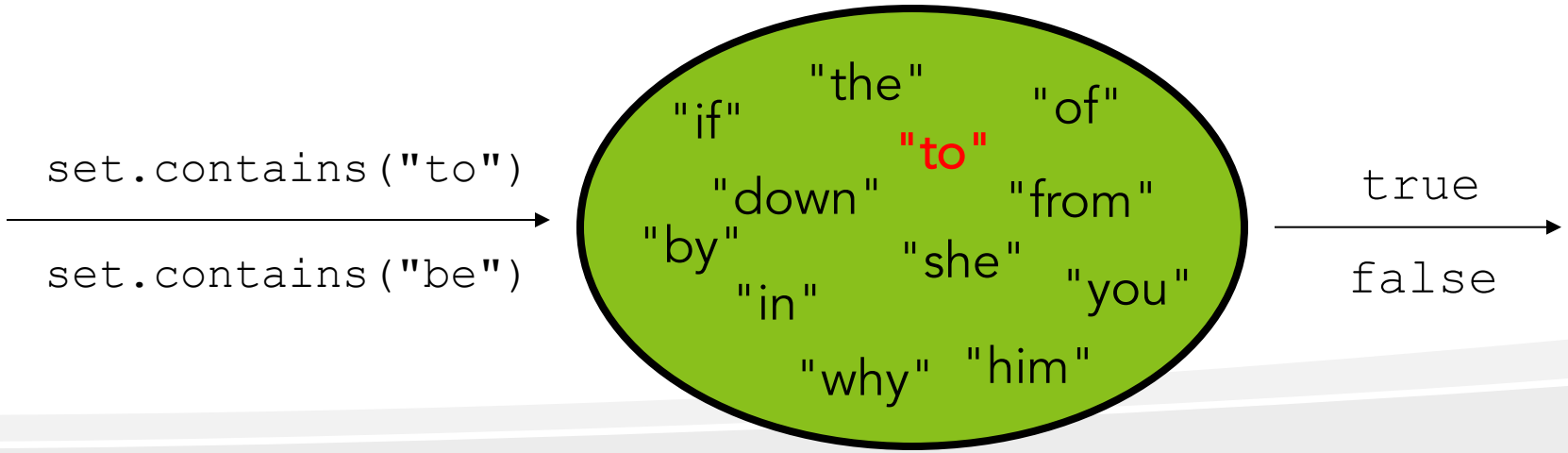


Words in a book

- Write an application that reads in the text of a book (say, Moby Dick) and then lets the user type words, and tells whether those words are contained in Moby Dick or not
  - How would we implement this with a List?

Sets

- **Set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - `add`, `remove`, search (`contains`)
  - We don't think of a set as having indexes; we just add things to the set in general and don't worry about order



Set

- Java has an interface named `Set<E>` to represent this kind of collection

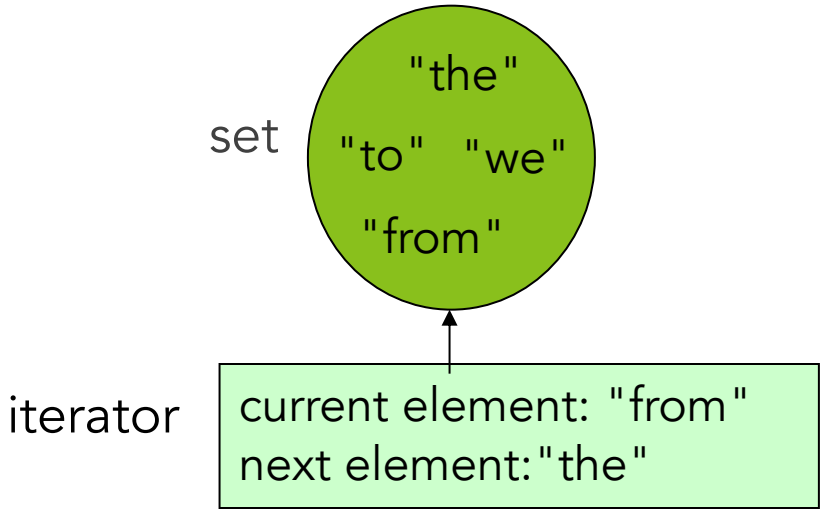
- We will discuss two `Set` implementations in Java: `TreeSet` and `HashSet`
  - Java's set implementations have been optimized so that it is very fast to search for elements in them

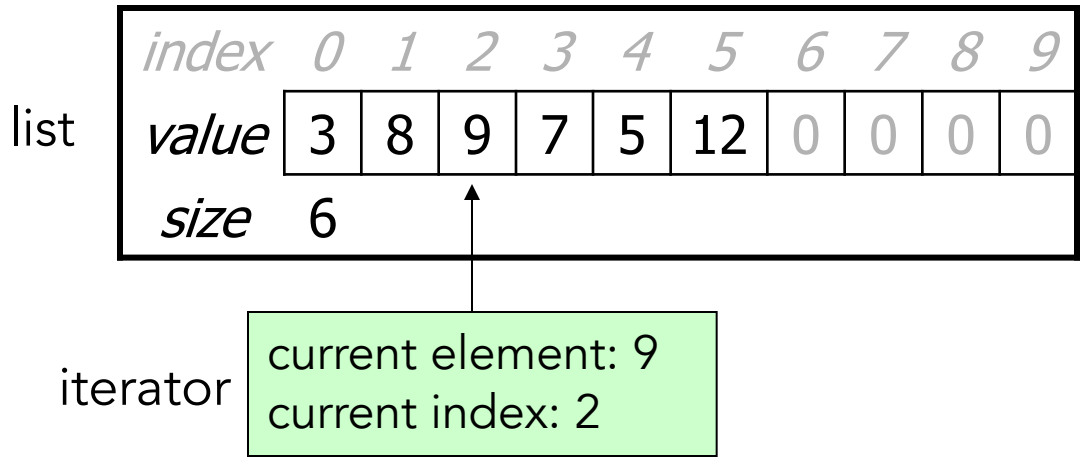
Java `Set` interface

- Interface `Set` has exactly the methods of the `Collection` interface
- `TreeSet` and `HashSet` classes implement the `Set` interface
  - `Set<Integer> set1 = new TreeSet<Integer>();`
  - `Set<Integer> set2 = new HashSet<Integer>();`
- Notice: The following `List` methods are missing from `Set`:
  - `get(index)`
  - `add(index, value)`
  - `remove(index)`

Java `Set` interface

- To access each element of a set we need to use its `iterator()` method





Set usage example

- The following code illustrates the usage of a set:

```
Set<String> strings= new HashSet<String>();
strings.add("Larry");
strings.add("Moe");
strings.add("Curly");
strings.add("Moe");    // duplicate, won't be added
strings.add("Shemp");
strings.add("Moe");    // duplicate, won't be added
System.out.println(strings);
```

- Output:

```
[Moe, Shemp, Larry, Curly]
```

- Notice that the order of the strings doesn't match the order in which they were added, nor is it the natural alphabetical order

Set methods

```
List<String> list = new ArrayList<String>();
…
Set<Integer> set1 = new TreeSet<Integer>();     // empty
Set<String> set2 = new HashSet<String>(list);
```

- Can construct an empty set, or one based on a given collection

`add(`**value**`)`	adds the given value to the set
`contains(`**value**`)`	returns `true` if the given value is found in this set
`remove(`**value**`)`	removes the given value from the set
`clear()`	removes all elements of the set
`size()`	returns the number of elements in list
`isEmpty()`	returns `true` if the set's size is 0
`toString()`	returns a string such as `"[3, 42, -7, 15]"`

`Set` concepts

- The set can be searched incredibly quickly
  - `contains` method often needs to examine just one element
- `HashSet` is implemented using a special internal array called **hash table**
  - Places elements into specific positions based upon integers called hash codes
  - Don't need to know the details only that you can `add`, `remove` and `contains` very quickly **O(1)**
- Drawbacks stores elements in unpredictable order

Example 1

- Find the unique words in a file
- This code ignores duplicate words in the file

```
Set<String> words = new HashSet<String>();
Scanner in = new Scanner(new File("test.txt"));
while(in.hasNext()) {
          String word = in.next();
          word = word.toLowerCase();
          words.add(word);
}
System.out.println("Number of unique words =" + words.size()););
```

Example 1 (using `HashSet<E>(list)`)

- `HashSet<E>(list)`
  - This constructor that accepts another collection as a parameter and puts all the unique elements from that collection into the `Set`
  - We use this constructor to find out whether a list contains any duplicates

```
public static boolean hasDuplicates(List<Integer> list) {
    Set<Integer> set = new HashSet<Integer>(list);
    return set.size() < list.size();
}
```

Drawback of a `Set`

- Does not store elements by indexes
- This code does not compile because there is no `get(index)` method

```
// remember: Set<String> words = new HashSet<String>();

for (int i=0; i< words.size(); i++) {
    String word = words.get(i); //error
    System.out.println(word);
}
```

Iterators on `Set`

- The following version works correctly

```
// remember: Set<String> words = new HashSet<String>();

Iterator<String> itr= words.iterator();
while (itr.hasNext()){
            String word = itr.next();
            System.out.println(word);
}
```

- Shorter alternative:

```
for (String word: words) {
            System.out.println(word);
}
```

TreeSet

- We can use a `TreeSet` for the previous code

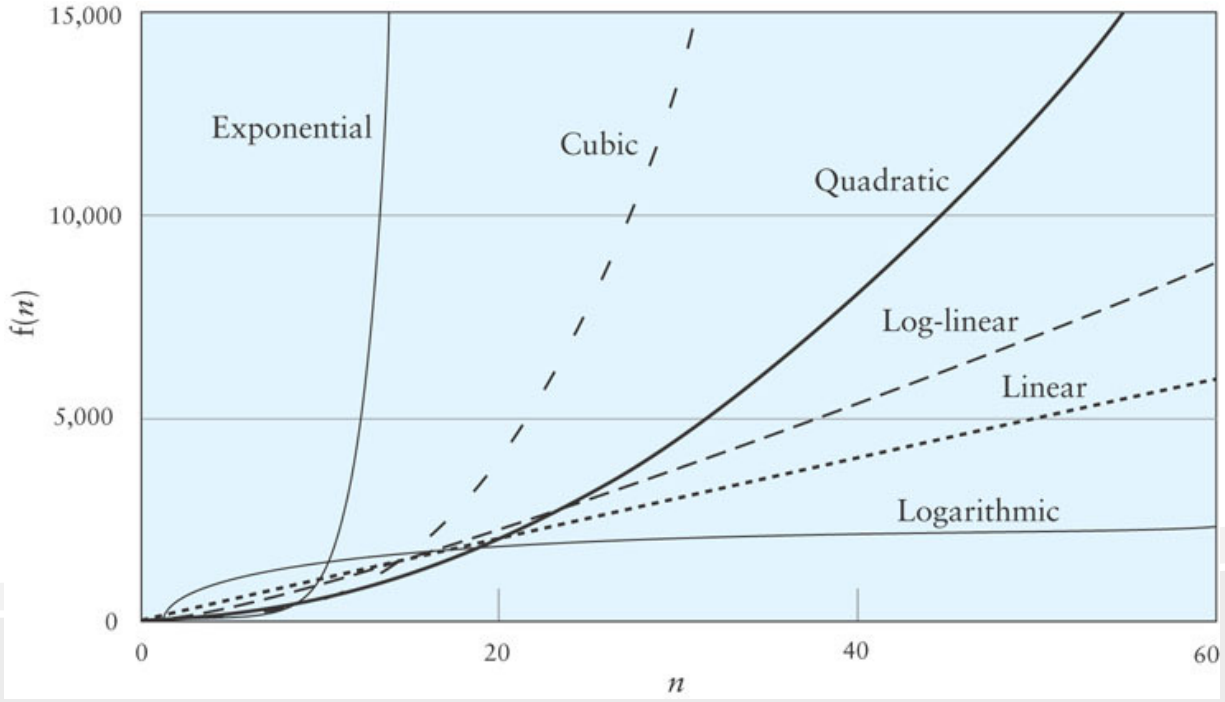
```
Set<String> strings= new TreeSet<String>();
strings.add("Larry");
strings.add("Moe");
strings.add("Curly");
strings.add("Moe");    // duplicate, won't be added
strings.add("Shemp");
strings.add("Moe");    // duplicate, won't be added
System.out.println(strings);
```

- Output:

```
[Curly, Larry, Moe, Shemp]
```

TreeSet

- `TreeSet`: implemented using a "binary search tree"; pretty fast **O(log n)** for all operations elements are stored in sorted order
- Stores elements in sorted order using an internal linked data structure called a binary search tree



`TreeSet` vs. `HashSet`

- A `TreeSet` stores its elements in the natural order
- `TreeSet` can only be used with elements with an ordering
  - Any class type that implements the `Comparable` interface
  - You cannot use it for elements that do not implement the `Comparable` interface .You will get a runtime error
- `TreeSet` is slightly (often not noticeably) slower than `HashSet`

Sets and ordering

- `HashSet` : elements are stored in an unpredictable order

```
Set<String> names = new HashSet<String>();
names.add("Jake");
names.add("Robert");
names.add("Marisa");
names.add("Kasey");
System.out.println(names);
// [Kasey, Robert, Jake, Marisa]
```

- `TreeSet` : elements are stored in their "natural" sorted order

```
Set<String> names = new TreeSet<String>();
...
// [Jake, Kasey, Marisa, Robert]
```

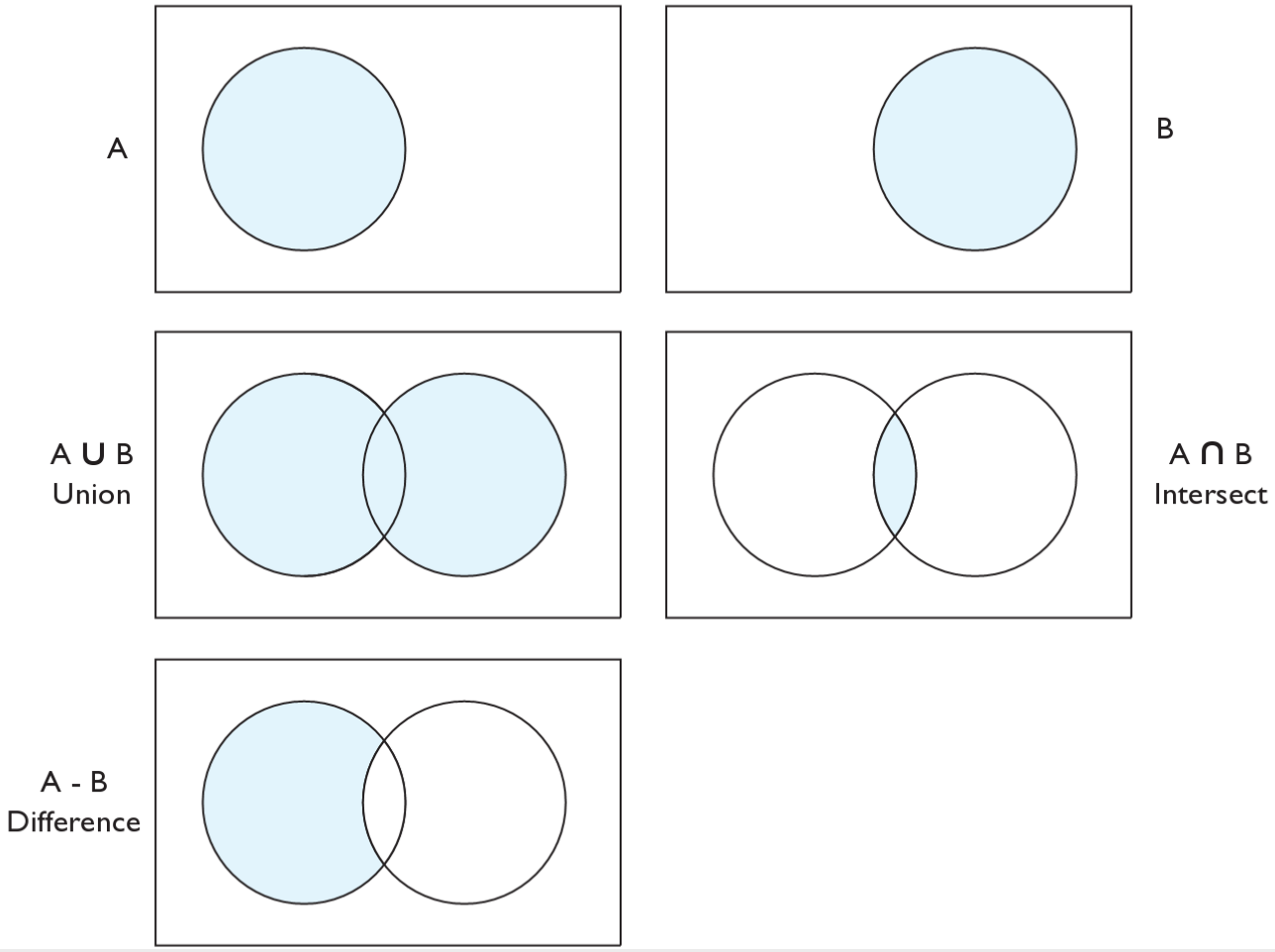
Strengths

- `HashSet`
  - Extremely fast performance for add, remove, contains
  - Can be used with any type of objects as its elements

- `TreeSet`
  - Elements are stored in sorted order
  - Must be used with elements that can be compared

Set operations

- Sets support common operations to combine them with, or compare them against, other sets:



Typical set operations

- Sometimes it is useful to compare sets:
  - **Subset**: S1 is a subset of S2 if S2 contains every element from S1
    - `containsAll` tests for a subset relationship
- It can be useful to combine sets in the following ways:
  - **Union**: S1 union S2 contains all elements that are in S1 or S2
    - `addAll` performs set union
  - **Intersection**: S1 intersect S2 contains only the elements that are in both S1 and S2
    - `retainAll` performs set intersection
  - **Difference**: S1 difference S2 contains the elements that are in S1 that are not in S2
    - `removeAll` performs set difference

Write a lottery program

- Generate at random a winning lottery ticket of 6 numbers and prompt the user to enter 6 lotto numbers. Depending on how many numbers match, the player wins various cash prizes
  - User should enter unique numbers (no duplicates)
  - Number of lotto can be up to 40

- We will use sets for storing the winning lotto numbers & player's numbers
  - No duplicates (lotto numbers are not duplicated)
  - Fast search (search if a player's number is in the winning set)

Winning numbers

- Write a method to generate the winner numbers

```
public static final int NUMBERS = 6;
public static final int MAX_NUMBER = 40;


public static Set<Integer> createWinningNumbers() {
        Set<Integer> winningNumbers = new TreeSet<Integer>();
        Random r = new Random();
        while (winningNumbers.size() < NUMBERS) {
            int number = r.nextInt(MAX_NUMBER) + 1;
            winningNumbers.add(number);
        }
        return winningNumbers;
}
```

Player's numbers

- Write a method to read the player's numbers

```
// reads the player's lottery ticket from the console
public static Set<Integer> getTicket() {
	Set<Integer> ticket = new TreeSet<Integer>();
	Scanner console = new Scanner(System.in);
	System.out.print("Type your " + NUMBERS + " unique lotto numbers: ");
	while (ticket.size() < NUMBERS) {
		int number = console.nextInt();
		ticket.add(number);
	}
	return ticket;
}
```

Check for winners?

Check for winners

- Option 1: search the winning number set to see whether it contains each number from the player's ticket

Check for winners

- Option 2: Find the intersection between the winning and the player's ticket

```
Set<Integer> winningNumbers = createWinningNumbers();
Set<Integer> ticket = getTicket();

// keep only the winning numbers from the user's ticket
Set<Integer> intersection = new TreeSet<Integer>(ticket);
intersection.retainAll(winningNumbers);
System.out.println("You had" + intersection.size() +  "matching numbers.");
```

Calculate prize

```
if (intersection.size() > 0) {
    double prize = 100 * intersection.size();
    System.out.println("The matched numbers are " + intersection);
    System.out.println("Your prize is $" + prize);
}
```

```
Type your 6 unique lotto numbers: 2 8 15 18 21 32
Your ticket numbers are [2, 8, 15, 18, 21, 32]
The winning numbers are [1, 3, 15, 16, 18, 39]
You had 2 matching numbers.
The matched numbers are [15, 18]
Your prize is $200.0
```