Data Structures

Augustin Cosse.

Spring 2021

February 11, 2021

Catching Exceptions

- If an exception occurs and is not handled, then the Java Runtime will terminate the program after printing after printing an appropriate message together with the trace of the runtime stack
- Before the program is terminated, each method on the stack trace has an opportunity to catch an exception
- Starting with the most deeply nested method in which the exception occurs, each method may either catch the exception, or allow it to pass through to the method that called it

Catching Exceptions

- For example, consider the following stack trace:

```
Exception in thread "main" java.lang.NullPointerException
  at java.util.ArrayList.toArray(ArrayList.java:358)
  at net.datastructures.HashChainMap.bucketGet(HashChainMap.java:35)
  at net.datastructures.AbstractHashMap.get(AbstractHashMap.java:62)
  at dsaj.design.Demonstration.main(Demonstration.java:12)
```

- In the stack trace above the **ArrayList.java** method had the first opportunity to catch the exception. Since it did not do so, the exception was passed upward to the **HashChainMap.bucketGet** method, which in turn ignored the exception, causing it to pass further upward to the **AbstractHashMap.get** method. The final opportunity to catch the exception was in the **Demonstration.main** method. But since it did not do so, the program terminated with the above diagnostic message.

Catching Exceptions

- The general methodology for catching exceptions is the **try-catch** construct in which a guarded fragment of code that might throw an exception is executed.
- If it throws an exception, then the exception is **caught** by having the flow of control jumping to a predefined **catch block** that contains the code to analyze the exception and apply an appropriate resolution.
- If no exception occurs in the guarded code, all catch blocks are ignored.

Catching Exceptions

- The typical syntax for the **try catch** is as follows

```
try{
// guarded body

} catch(exceptionType1 variable1) {
// remedy body 1
} catch(exceptionType2 variable2) {
// remedy body 2
}
```

- Each exceptionTypei is the type of an exception and each variablei is a valid Java variable name.

Catching Exceptions

- The Java Runtime environment starts by executing the block **guarded body**. If no exception is generated during the execution, the flow of control continues with the first statement beyond the last line of the entire **try catch statement**.
- If, on the other hand, the block bfseries guarded body generates an exception at some point, the execution of that block immmediately terminates and execution jumps into the **catch** block whose exceptionType most closely matches the exception thrown.
- The variable for this catch statement references the exception object itself, which can be used in the block of the matching **catch statement**.

Catching Exceptions

- Once execution of the **catch** block completes, control flows continues with the first statement beyond the entire **try catch** construct
- If an exception occurs during the execution of the block **guardedBody** that does not match any of the exception types characterized by the catch statements, that exception is rethrown in the surrounding context.

Catching Exceptions

- If you are willing to use the same error message for the two exceptional cases, we can use a single catch clause wit the following syntax

```java
}catch(ArrayIndexOutOfBoundsException |
        NumberFormatException e){
System.out.println("Using default value for n. ");
}
```

Throwing Exceptions

- Exceptions are generated when a piece of Java code finds some sort of problem during execution and **throws** and exception object
- This is done through the keyword **throw** followed by an instance of the exception type to be thrown.
- It can sometimes be convenient to instantiate an exception at the time the exception has to be thrown.
- A thrown statement typically takes the form

```java
throw new exceptionType(parameters);
```

- Where exceptionType is the type of the exception and the parameters are sent to that exception's constructor.

Throwing Exceptions

- Most exceptions offer a version of a constructor that accepts an error message string as a parameter
- As an example, consider the following code snippet

```java
public void ensurePositive(int n){
if (n<0)
    throw new IllegalArgumentException
                              ("That's not positive")
// ....
}
```

- The execution of a throw statement immediately terminates the body of a method.

Throwing Exceptions

- When a method is declared, it is possible to explicitly declare as part of the method's signature the possibility that a particular exception type may be thrown during a call to that method
- It does not matter whether the exception is directly thrown from from a **throw** statement in that method body or propagated upward from a secondary method call from within the body.
- the syntax for that particular situation relies on the keyword **throws** (with an **s**) such as

```
public static int parseInt (String s) throws NumberFormatException;
```

Throwing Exceptions

- The designation **throws** NumberFormatException warns the user about the possibility of an exceptional case so that they might be better prepared to handle an exception that may arise

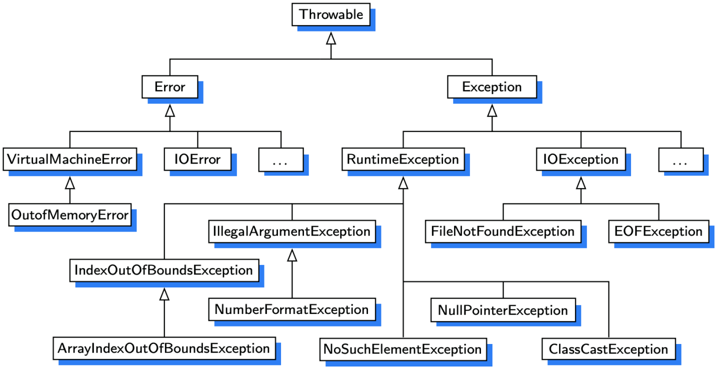
- If one of many exception types may possibly be thrown, all such types can be listed separated by commas.

- In short, if you are using a try-catch block for a particular exception, you are handling the exception there. If you specify throws on the method, you are declaring that the method can throw such an exception(and not handle it within itself) and it becomes the responsibility of the caller to handle that exception.

```
public static int parseInt (String s) throws NumberFormatException;
```

Throwing Exceptions

- Java defines a rich inheritance hierarchy of all objects that are deemed Throwable
- The two main branches of the hierarchy are errors and exceptions



Throwing Exceptions

- Errors are typically thrown by the Java Virtual Machine and designate the most serious situations that are unlikely to be recoverable, such as when the virtual machine is asked to execute a corrupt class file
- In contrast exception designate situations in which a running program might reasonably be able to recover, for example, when opening a data file.
- Java further provide a division between the RuntimeException class (which are officialy treated as checked exceptions) and a remaining set of other subclasses which are labelled as checked exceptions

Casting and Generics

- We have discussed casting for base type. Java also provides a way to do conversion between objects
- Just as for base types, we will talk about **widening conversion** when :
  - A type T is converted to a type U, T and U are class types and U is a super class of T
  - A type T is converted to a type U, T and U are interface types and U is a super interface of T
  - A type T is converted to a wider type U, T is a class that implements the interface U

Casting and Generics

- Widening conversion automatically occurs to store the result of an expression within a variable, without the need for explicit casting.
- In other words, as we saw, we can directly store the result of an expression of type T into a variable *v* of type U when the conversion from T to U is a widening conversion
- We already encountered such a situation when discussing polymorphism

```java
CreditCard card = new PredatoryCreditCard(...);
```

Casting and Generics

- A **narrowing conversion** occurs when a type T is converted into a narrower type S. Common examples include
  - A type T is converted to a type S, T and S are class types and S is a subclass of T
  - A type T is converted to a type S, T and S are interface types and S is a subinterface of T
  - A type T is converted to a type S, T is an interface implemented by class S

Casting and Generics

- In general, narrowing conversion of reference types requires an explicit cast.
- The correctness of a narrowing conversion may not be verifiable by the compiler. Hence its validity should be tested by the Java runtime environment during program execution

```
CreditCard card = new PredatoryCreditCard(...);  // widening
PredatoryCreditCard pc = (PredatoryCreditCard) card; // narrowing
```

Generics

- Java provides support for writing generic classes and methods that cna operate on a variety of data types while often avoiding the need for explicit casts.
- The generics framework allows use to to define a class in terms of a set of formal type parameters which can be used as the declared type for variables, parameters and return values within the class definition.
- Those formal type parameters are later specified when using the generic class as a type elsewhere in the program

Generics

- The generics framework was not part of the original Java language. It was added as part of Java SE5.
- Prior to that, generic programming was implemented through the class **Object** which is the universal supertype of all objects
- In that "classic" style, a generic pair might be implemented as follows.

```
public class ObjectPair{
  Object first;
  Object second;
  public ObjectPair(Object a, Object b){
  first = a;
  second = b;}
  public Object getFirst() {return first; }
  public Object getSecond() {return second; }}
```

Generics

- An instance of the ObjectPair class thus stores the two objects that are sent to the constructor and provides individual accessors for each component of the pair
- With this definition, a pair can be declared and instantiated with the following command

```
ObjectPair bid = newObjectPair("ORCL", 32.07)
```

- This instantiation is legal because the parameters to the constructor undergo widening conversion. The first parameter "ORCL" is a string (and thus also an object) and the second is a double (but it is automatically boxed into a Double object)

Generics

- The drawback with the classical approach are the accessors, both of which formally return an Object reference
- Even if we know that the first object is a string, we cannot legally make the assignment

```
String stock = bid.getFirst(); // illegal
```

- This represents a narrowing conversion from the declared return type of Object to the variable of type String. In this case, an explicit cast is required

```
String stock = (String) bid.getFirst(); // illegal
```

Generics

- With Java's generic framework, we can implement a pair class using formal type parameters to represent the two relevant types in our composition

```java
public class Pair<A,B> {
A first;
B second;
public Pair(A a, B b) { // constructor
first = a;
second = b;
}
public A getFirst( ) { return first; }
public B getSecond( ) { return second;}}
```

Generics

- A new pair can then be instantiated as

```
bit = new Pair<>("ORCL", 32.07)
```

Using arrays

- As a first illustration of the use of arrays in Java, we consider the class GameEntry which stores the scores of a number of video games players

```java
public class GameEntry {
private String name; // name of the person earning this score
private int score; // the score value
/ Constructs a game entry with given parameters.. /
public GameEntry(String n, int s) {
name = n;
score = s;
}
/ Returns the name field. /
public String getName( ) { return name; }
/ Returns the score field. /
public int getScore( ) { return score; }
/ Returns a string representation of this entry. /
public String toString( ) {
return "(" + name + ", " + score + ")";
}}
```

Using arrays

- To keep track of the scores of all players, we can then define another class (which we will call **ScoreBoard**)
- The **ScoreBoard** class will consist of a limited number of scores that can be maintained. Once the limit has been reached, a new score only qualifies for the score board if it is strictly higher than the lowest score "high score" on the board.
- Within the **ScoreBoard** class, we will use an array to manage the scores

```
public class ScoreBoard{
  private int numEntries = 0;
  private GameEntry[] board;
  /** constructor */
 public Scoreboard(int capacity){
  board = new GameEntry[capacity];
}}
```

Using arrays

- All the entries in the array are intially set to **null**.
- One of the most basic improvement we want to make to the ScoreBoard class is to add a method that will add an entry in the board.

```
public void add(GameEntry e) {
int newScore = e.getScore( );
// is the new entry e really a high score?
if (numEntries < board.length ||
                 newScore > board[numEntries-1].getScore( )) {
if (numEntries < board.length) // no score drops from the board
numEntries++; // so overall number increases
// shift any lower scores rightward to make room for the new entry
int j = numEntries - 1;
while (j > 0 && board[j-1].getScore( ) < newScore) {
board[j] = board[j-1]; // shift entry from j-1 to j
j--; // and decrement j
}
board[j] = e; // when done, add new entry
}}
```

Using arrays

- When a new score is considered, the first step is to determine if it qualifies as a high score. This is always true if the board has not reached its full capacity yet

- If the board has attained its maximum capacity, we have to determine whether the new GameEntry belongs or not

```
public void add(GameEntry e) {
int newScore = e.getScore( );
// is the new entry e really a high score?
if (numEntries < board.length ||
                newScore > board[numEntries-1].getScore( )) {
if (numEntries < board.length) // no score drops from the board
numEntries++; // so overall number increases
// shift any lower scores rightward to make room for the new entry
int j = numEntries - 1;
while (j > 0 && board[j-1].getScore( ) < newScore) {
board[j] = board[j-1]; // shift entry from j-1 to j
j--; // and decrement j
}
board[j] = e; // when done, add new entry
}}
```

Using arrays

- If the capacity of the array has not been attained, the first thing we should do is increase this capacity by 1
- When the array is full, the addition of a new entry necessarily implies removing an entry

```
public void add(GameEntry e) {
int newScore = e.getScore( );
// is the new entry e really a high score?
if (numEntries < board.length ||
                newScore > board[numEntries-1].getScore( )) {
if (numEntries < board.length) // no score drops from the board
numEntries++; // so overall number increases
// shift any lower scores rightward to make room for the new entry
int j = numEntries - 1;
while (j > 0 && board[j-1].getScore( ) < newScore) {
board[j] = board[j-1]; // shift entry from j-1 to j
j--; // and decrement j
}
board[j] = e; // when done, add new entry
}}
```

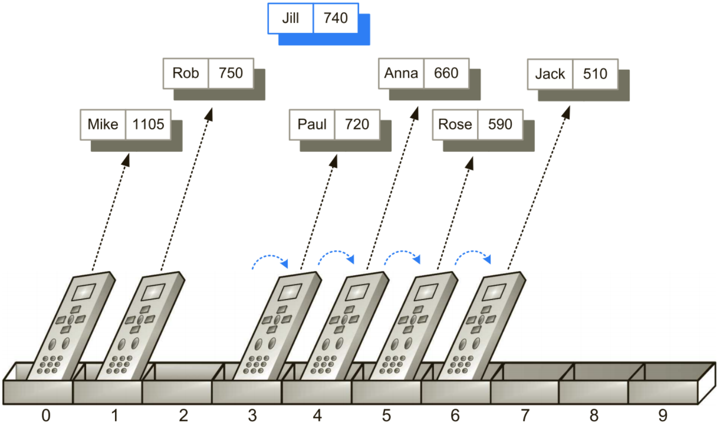
Using arrays

- To update the array, we start from the end. While the GameEntry score we need to add is higher than the score at entry j, we shift score j by one.

```
public void add(GameEntry e) {
int newScore = e.getScore( );
// is the new entry e really a high score?
if (numEntries < board.length ||
                newScore > board[numEntries-1].getScore( )) {
if (numEntries < board.length) // no score drops from the board
numEntries++; // so overall number increases
// shift any lower scores rightward to make room for the new entry
int j = numEntries - 1;
while (j > 0 && board[j-1].getScore( ) < newScore) {
board[j] = board[j-1]; // shift entry from j-1 to j
j--; // and decrement j
}
board[j] = e; // when done, add new entry
}}
```

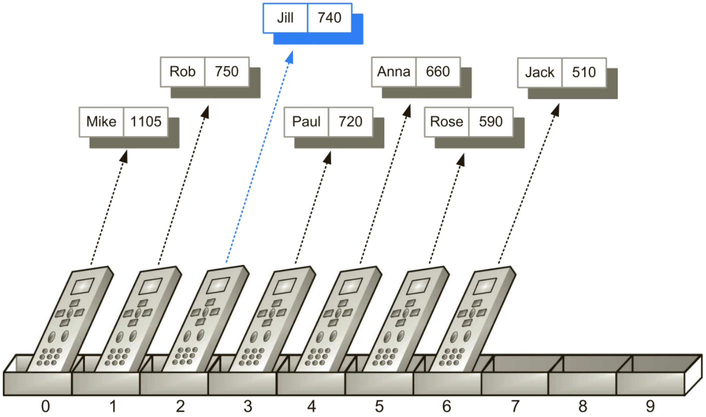
Using arrays

- The idea of the **add** method can be summarized by the figure below



Using arrays

- The idea of the **add** method can be summarized by the figure below

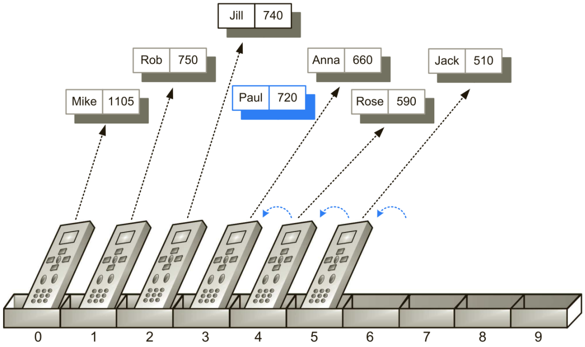


Using arrays

- How would you adapt the methods in a situation where you don't need to preserve the ordering of the entries ?

Using arrays

- We now need to remove an entry from the array. We consider the function below
- Here again, when a score is removed, all the lower scores must be shifted towards the left as shown by the figure below



Using arrays

- Our method remove (below) will involve a loop that starts from the index *i* that we want to remove and then gradually shifts (i.e. board[i] = board[i+1]) all the entries on the left.

- There is not entry to shift in cell board[numEntries-1] so we return that cell to **null** just after the loop.

```
/ Remove and return the high score at index i. /
public GameEntry remove(int i) throws IndexOutOfBoundsException {
if (i < 0 || i >= numEntries)
throw new IndexOutOfBoundsException("Invalid index: " + i);
GameEntry temp = board[i]; // save the object to be removed
for (int j = i; j < numEntries - 1; j++) // count up from i (not down)
board[j] = board[j+1]; // move one cell to the left
board[numEntries -1 ] = null; // null out the old last score
numEntries--;
return temp; // return the removed object
}
```

Using arrays

- The methods for adding and removing object in the array of high scores are simple. Nevertheless, they form the basis of techniques that are used repeatedly to build more sophisticated data structures.
- These other data structures, however will be more general than the array structure which we just covered and they will often have more operations they will be able to perform than just add or remove but studying the array data structure is a very good starting point.

Sorting an array

- Another important function when working with arrays is **sorting**
- As a warm up, we will consider the simple **insertion-sort** algorithm. We want this algorithm to proceed by placing the new elements in the correct order with respect to those which came before it.

```
// Input: An array A of n comparable elements
// Output: The array A with elements rearranged
for k from 1 to n-1 do
  Insert A[k] at its proper location within A[0],
                                        A[1],...A[k]
```

Sorting an array

- The correct approach is to start form the first element (for which the array is trivially sorted). Then proceed with the second element. If it is smaller than the first, we swap them.

```java
/** Insertion-sort of an array of
characters into nondecreasing order */
public static void insertionSort(char[ ] data) {
int n = data.length;
for (int k = 1; k < n; k++) { // begin with second character
  char cur = data[k]; // time to insert cur=data[k]
    int j = k; // find correct index j for cur
    while (j > 0 && data[j-1] > cur) { /* thus, data[j-1]
                                          must go after cur */
         data[j] = data[j-1]; // slide data[j-1] rightward
         j--; // and consider previous j for cur
     }
    data[j] = cur; // this is the proper place for cur
}}
```

Sorting an array

- We then proceed with the third element, swapping leftwards until it reaches its proper position and we continue loke that with all the subsequent elements.

```
/** Insertion-sort of an array of
characters into nondecreasing order */
public static void insertionSort(char[ ] data) {
int n = data.length;
for (int k = 1; k < n; k++) { // begin with second character
  char cur = data[k]; // time to insert cur=data[k]
    int j = k; // find correct index j for cur
    while (j > 0 && data[j-1] > cur) { /* thus, data[j-1]
                                         must go after cur */
         data[j] = data[j-1]; // slide data[j-1] rightward
         j--; // and consider previous j for cur
     }
    data[j] = cur; // this is the proper place for cur
}}
```

Sorting an array

- You can see that the function **insertion-sort** moves a new element to its proper location starting from the far right and gradually swapping this element with the entries on its left whenever they are larger (largest is on the right)

```java
/** Insertion-sort of an array of
characters into nondecreasing order */
public static void insertionSort(char[ ] data) {
int n = data.length;
for (int k = 1; k < n; k++) { // begin with second character
  char cur = data[k]; // time to insert cur=data[k]
    int j = k; // find correct index j for cur
    while (j > 0 && data[j-1] > cur) { /* thus, data[j-1]
                                        must go after cur */
         data[j] = data[j-1]; // slide data[j-1] rightward
         j--; // and consider previous j for cur
     }
    data[j] = cur; // this is the proper place for cur
}}
```

Sorting an array

- A nice feature of the **insertionSort** method is that once the array is sorted, the inner loop does only one comparison., determines that there is no swap needed and then get back to the outer loop.

```
/** Insertion-sort of an array of
characters into nondecreasing order */
public static void insertionSort(char[ ] data) {
int n = data.length;
for (int k = 1; k < n; k++) { // begin with second character
  char cur = data[k]; // time to insert cur=data[k]
    int j = k; // find correct index j for cur
    while (j > 0 && data[j-1] > cur) { /* thus, data[j-1]
                                          must go after cur */
         data[j] = data[j-1]; // slide data[j-1] rightward
         j--; // and consider previous j for cur
     }
    data[j] = cur; // this is the proper place for cur
}}
```

Illustration of the **insertionSort** method

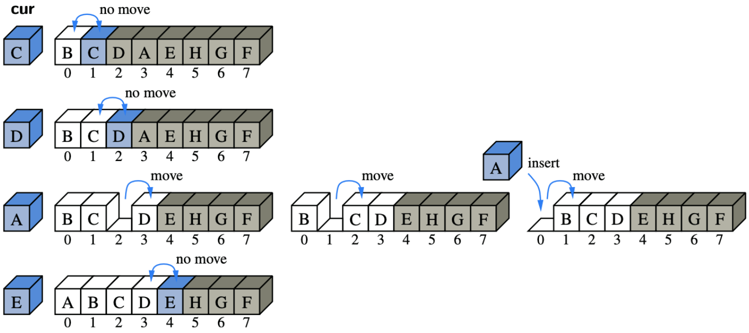
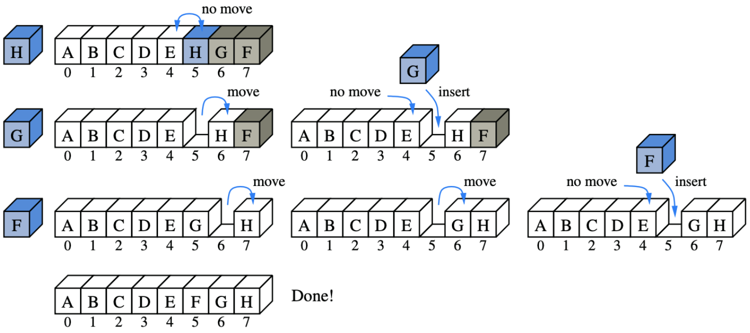


Illustration of the **insertionSort** method



Java.util Methods for Arrays and Random numbers

- Because arrays are so important, Java provides a class **java.util.Arrays** with a number of nuilt in static methods for performing common tasks

equals(A, B)	Returns true if and only if the array A and the array B are equal. Two arrays are equal if they have the same number of elements and every corresponding pair of elements in the two arrays are equal.
fill(A, x)	Stores value x in every cell of an array A, provided the type of array A is defined so that it allows to store the value x

Java.util Methods for Arrays and Random numbers

copyOf(A, n)	Returns an array of size n such that the first k elements of A are copied from A, where $k = \min(n, A.length)$. If $n >$ A.length then the last n-A.length elements in this arraywill be padded with default values, e.g. 0 for an array of **int** and **null** for an array of objects.
copyOfRange(A, s, t)	Returns an array of size (t-s) such that the elements of this array are copied in order from A[s] to A[t-1], where s<t, padded with **copyOf** if t>A.length

Java.util Methods for Arrays and Random numbers

toString(A)	Returns a string representation of the Array A, beginning with [, ending with] and with elements of A displayed separated by string ",". The string representation of and element A[i] is obtained using String.valueOf(A[i]), which returns the string "null" for a **null** reference and otherwise calls A[i].toString().

Java.util Methods for Arrays and Random numbers

sort(A)	Sorts the array A based on natural ordering of its elements, which must be comparable
binarySearch(A), x	Searches the sorted array A for value x, returning the index where it is found , or else the index where it could be inserted while maintaining the sorted order.

Java.util Methods for Arrays and Random numbers

- As static instances, the methods java.util.Arrays class are invoked directly on the class, not on any particular instance of the class.
- For example, if **data** were an array, we could sort it wiht syntax java.util.Arrays.sort(data), or with the shorter syntax Arrays.sort(data) if we import the Arrays class

Java.util Methods for Arrays and Random numbers

- Another feature built in Java which is often useful when testing programs dealing with arrays is the ability to generate pseudo random numbers
- Java has a built in class, java.util.Random whose instances are pseudorandom numbers generators, that is objects that compute a sequence of numbers that are statistically random
- The sequence are not actually random in the sense that it is possible to predict the next number in the sequence given the past list of numbers.
- Indeed, a popular pseudorandom number generator is to generate the next number, **next**, from the current number **cur** as **next** = (a***cur** + b) % n where a, b are appropriately chosen integers

Java.util Methods for Arrays and Random numbers

- An idea along this line is used by the java.util.Random objects, with $n = 2^{48}$.
- It turns out that such a sequence can be proven to be statistically uniform, which is usually good enough for most applications requiring random numbers such as games
- For applications such as computer security settings where unpredictable security sequences are needed, this king of formula should not be used.
- Instead one should consider a sample coming from a physical source that is actually random such as atmospheric noise.

Java.util Methods for Arrays and Random numbers

- Since the next number in a pseudorandom generator is determined by the previous numbers, such a generator always needs a place to start, which is called the **seed**
- The sequence of numbers generated from a same seed will always be the same.
- The seed for an instance of the **java.util.Random** class can be set in its constructor, or with its **SetSeed()** method.
- One common trick to get a different sequence, each time a program is run is to use a seed that will be different for each run.

Java.util Methods for Arrays and Random numbers

- The methods of the **java.util.Random** class are the following:

nextBoolean()	Returns the next pseudorandom **boolean** value
nextDouble()	Returns the next pseudorandom **double** value between 0.0 and 1.0
nextInt()	Returns the next pseudorandom **int** value
nextInt(n)	returns the next pseudorandom **int** value in the range from 0 to but not including n
setSeeds(s)	Sets the seed of this pseudorandom number generator to the **long** s

PseudoRandom Numbers Generators

```
import java.util.Arrays;
import java.util.Random;
/** Program showing some array uses. */
public class ArrayTest {
public static void main(String[ ] args) {
int data[ ] = new int[10];
Random rand = new Random( );
// a pseudo-random number generator
rand.setSeed(System.currentTimeMillis( ));
// use current time as a seed
// fill the data array with pseudo-random
// numbers from 0 to 99, inclusive

// ....
```

PseudoRandom Numbers Generators

```
import java.util.Arrays;
import java.util.Random;
/** Program showing some array uses. */
public class ArrayTest {
public static void main(String[ ] args) { ...
for (int i = 0; i < data.length; i++)
      data[i] = rand.nextInt(100);
      // the next pseudo-random number
int[ ] orig = Arrays.copyOf(data, data.length);
// make a copy of the data array
System.out.println("arrays equal before sort: "
           +Arrays.equals(data, orig));
Arrays.sort(data);
// sorting the data array (orig is unchanged)
System.out.println("arrays equal after sort: "
           + Arrays.equals(data, orig));
System.out.println("orig = " + Arrays.toString(orig));
System.out.println("data = " + Arrays.toString(data));}}
```

Java.util Methods for Arrays and Random numbers

- By using a pseudo random number generator to determine program values, we can get a different input to our program each time we run it
- This feature is in fact what makes pseudorandom number generator useful for testing code, aprticularly when dealing with arrays.
- Even so, we should not use random tests as a replacement for reasoning about our code, as we might miss important special cases in test runs