Northeastern University

Generics, I/O & Regular Expressions

Maria Zontak

Credits: CS143 course I taught in North Seattle College
CS5004 course built by Dr. Therapon Skotiniotis here in Northeastern
Slides on Regular Expressions by Yuri Pekelny (my husband)

Prelude: `ArrayList` vs. array

- Arrays Drawback: Fixed length, once created

1. Cannot change the size of the arrays
2. Cannot accommodate an extra element in an array
3. Memory is allocated to an array during its creation only (before the actual elements are added to it).

- **`ArrayList` Advantage: `ArrayList` is resizable:**

1. Can grow and shrink dynamically
2. Elements can be inserted/deleted at/from a particular position.
3. Has many methods to manipulate the stored objects.
4. **Today we will see one more advantage!!!**

Review: ADTs as interfaces

Abstract Data Type (ADT): a specification of a collection of data and the operations that can be performed on it.

- Describes *what* a collection does, not *how* it does it.

- Java's collection framework uses interfaces to describe ADTs:
  - `Collection`, `List`, `Map`, `Set`,…

- An ADT can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList` implement `List`
  - `HashSet` and `TreeSet` implement `Set`

Which one to use?

```
(1)    List<String> list = new ArrayList<String>(); or
(2)   ArrayList<String> list = new ArrayList<String>();
```

- Almost always (1) should be preferred over (2)

Why?

- The implementation of the `List` can change (to a `LinkedList` for example), without changing the rest of the code.
- Difficult task if `ArrayList` is used because:
  - Need to change `ArrayList` to `LinkedList` everywhere
  - Not compatible if specific methods of `ArrayList` were used

Set your variable Static/Compile-time type to be Interface

Using ADT interfaces

- It is considered a good practice (and a MUST in this course) to declare collection variables using the corresponding ADT interface type:

```
List<String> list = new ArrayList<String>();
```

- Methods that accept/return a collection as a parameter should also declare the parameter using the ADT interface type:

```
public void doSomething(List<String> list) {
    ...
}
public List<String> getMyList() {
    ...
}
```

Type Parameters (Generics)

```
List<Type> name = new ArrayList<Type>();
```

$\uparrow$

type parameter: type of elements `ArrayList` will contain

- Allows the same `ArrayList` class to store lists of different types.
- Also called *generic* class.

```
List<String> names = new ArrayList<String>();
names.add("Maria Zontak");
names.add("Your name");
```

Primitives as type parameter?

- The type parameter must be an object type

→ Setting a primitive type is ILLEGAL:

```
~~List<int> list = new ArrayList<int>();~~
```

- Instead use special classes - *wrapper* classes (hold primitive values):

```
// creates a list of ints
List<Integer> list = new ArrayList<Integer>();
```

Primitive Type	Wrapper Type
`int`	`Integer`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Type Parameters (Generics)

```
List<Type> name = new ArrayList<>();
```

Since Java 7:

– **Diamond Operator** can be used on the right hand side of an assignment

– Java compiler will auto populate each type parameter from the compile-time **Type** on the left hand side of the assignment.

```
List<String> names = new ArrayList<>();
names.add("Maria Zontak");
names.add("Your name");
```

Implementing generics

```
// a parameterized (generic) class
public class Name<T> {
    ...
}
```

T stands for Type

- By putting the **T** in < >, you are demanding that any client that constructs your object must supply a type parameter.
  - You can require multiple type parameters separated by commas.

- The rest of this class's code can refer to that type by name **T**

Implementing generics

Let’s try design a class that can hold **any pair of objects**.

For example:

1. First Name & Last Name
2. Birth Month (Jan ... Dec) & Birth Day (1 … 31)
3. X and Y coordinates in `Point2D`

Any suggestions?

Implementing generics

```
/**
 * Represents a pair that holds two values.
 * @param <X> type for the first value.
 * @param <Y> type for the second value.
 */
public class Pair<X,Y> {
  private X first;
  private Y second;

  /**
   * Creates a pair from the two given values.
   * @param first value/element of the pair.
   * @param second value/element of the pair.
   */
  public Pair(X first, Y second) {
    this.first = first;
    this.second = second;
  }
```

`Pair<X,Y>` can be read as **Pair of X and Y**.

The scope of `X & Y` is till the end of the class

`X & Y` can be used as **field types**

`X & Y` can be used as **argument types**

Implementing generics

```
/**
 * Get the first element of the pair.
 * @return the first element
 */
 public X getFirst() {
   return first;
 }


  /**
   * Set the first element of the pair.
   * @param first the new value for the first element.
   */
 public void setFirst(X first) {
   this.first = first;
 }
… //do the same with second
```

X & Y
can be used as
return type

X & Y
can be used as
argument types

Using Generic Pair<X,Y>

```
public static void main(String[] args) {
   Pair<String,Integer> myBirthday = new Pair<>("January",9);

   String  month = myBirthday.getFirst();
   Integer day = myBirthday.getSecond();

   Pair<Pair<String,Integer>,Integer> myFullBirthday =
                                new Pair<>(myBirthday, 1983);

   Pair<String,Integer>  date = myFullBirthday.getFirst();
   Integer year = myFullBirthday.getSecond();
 }
```

Better abstract date → create a class `Date` with field **`Pair<Integer, String>`**
OR… What else can you do?

Using Generic Pair<X,Y>

What about Point2D? What options do we have?

1. Point2D **has a** Pair of coordinates
2. Point2D **is a** Pair of coordinates

```
public class Point2D extends Pair<Integer, Integer> {

 public Point2D(Integer x, Integer y) {
     super(x, y);
   }
   /**
    * @return Value for property 'x'.
    */
   public Integer getX() {
     return super.getFirst();
   }
   /**
    * @param x Value to set for property 'x'.
    */
   public void setX(Integer x) {
     super.setFirst(x);
   } … //do the same with Y
```

Wrapper class allows **code reuse** along with **abstraction**

Wrapper class allows more **meaningful names**

Using Generic Pair<X,Y>

- What about additional behavior?

```
// @return distance between this point and another point
public double calcDistance(Point2D other) {
    return Math.sqrt(Math.pow(this.getX() - other.getX(), 2) +
        Math.pow(this.getY() - other.getY(), 2));
  }
```

- What if we wanted to write this method using static method?

```
public static double calcDistance(Point2D first, Point2D second) {
    return Math.sqrt(Math.pow(first.getX() - second.getX(), 2) +
        Math.pow(first.getY() - second.getY(), 2));
  }
```

- Can we have BOTH methods in the same class?
- What about code reuse?

YES
This is
Overloading

```
public Double calcDistance(Point2D other) {
    return calcDistance(this,other);
}
```

Overloading Rules

Two (or more methods) are overloaded if:

1. the method name is the same
2. the argument list differs in:
   – the number of arguments
   – the types for the arguments
   – the order of the arguments

You CANNOT have two or more methods with the same signature:

- the same name and
- the same number of arguments with the same argument order and the same argument types

A method's return type is **not** considered when resolving overloaded method

→ You cannot declare two methods with the same signature even if they have a different return type.

Generics and static methods

```
public class PairUtil {
    public static <X,Y> boolean pairEquals(Pair<X,Y> p1, Pair<X,Y> p2) {
        return p1.getFirst().equals(p2.getFirst())
            && p1.getSecond().equals(p2.getSecond());
    }
}
```

Static methods ARE SHARED among ALL the instances of the same class → CANNOT depend on class type parameter →MUST define their own type parameter/s

Let's use it:

```
Pair<String, Integer> p1 = new Pair<>("A", 1);
Pair<String, Integer> p2 = new Pair<>("B", 2);
PairUtil.<String, Integer> pairEquals(p1, p2);
```

OR:

```
PairUtil.pairEquals(p1, p2);
```

Java compiler will infer the type based on the arguments

Generics and arrays

```
public class Foo<E> {
    private E myField;                          // ok

    public void method1(E param) {
        myField = new E();                      // error
        E[] a = new E[10];                      // error

        myField = param;                        // ok
        E[] a2 = (E[]) (new Object[10]);        // ok
    }
}
```

E stands for Element

Generally, AVOID using GENERIC arrays

- You cannot create objects or arrays of a parameterized type.
- You can create variables of that type, accept them as parameters, return them, or create arrays by casting from `Object[]`.

A bit of history

Before Java 5.0:

- The **static type** of the elements of a Collection was **Object**.

→ Usually required a type cast upon getting/removal of an element

```
List emps = new ArrayList();
emps.add(new Employee("Michael"));
Employee emp = (Employee)emps.get(0);
```

Object as Argument Type

- Allowed any type of reference to be added to a collection

→ potentially leading to errors.

[Let's illustrate with Arrays → see code example]

New in Java 5.0: Generics

Beginning with Java 5.0, the static type of the elements may be specified using a **type parameter**.

```
List<Employee> emps = new ArrayList<Employee>();
```

or

```
List<Employee> emps = new ArrayList<>();
```

- With the type parameter, the *compiler* ensures that we use the collection with objects of a compatible type only
- Object type errors are now detected **at compile time**, rather than throwing casting exceptions at runtime.
- Warnings generated if type parameter not included
- NO need to cast to get an object:

```
Employee emp = emps.get(0);
```

Careful: Generics and Polymorphism

Collections of a type are NOT polymorphic on the type.

- A `List<String>` CANNOT be assigned to a reference variable of type `List<Object>`
- Similarly, if `Employee` is a subclass of `Person` a `List<Employee>` cannot be assigned to a `List<Person>`.

→ Both will result in a compiler error.

- By extension: CANNOT pass a `List<String>` as an argument to a method whose parameter is type `List<Object>`.

For example: `printCollection (List<Person> personList),`

will accept `List<Person>` collections as arguments.

- Why?

If allowed, objects in the collection could be manipulated through a more "generic" typed reference variable → troubles→ see code

Java 5.0: Autoboxing and Auto-Unboxing

- Before Java 5.0, a primitive type could not be directly added to a collection. Instead, it needed to be manually 'boxed' using a wrapper class:

```
 int num = 5;
myIntegerList.add(new Integer(num));
```

- In addition, it was necessary to manually extract the primitive value from the wrapper object to use it:

```
int x = ((Integer)myIntegerList.get(0)).intValue();
```

// ugly!

Examples

Before

```
ArrayList numList = new ArrayList( );
int num = 2;
// numList.add( num ); // syntax error
numList.add( new Integer( num ) );
int x =
    ( (Integer)numList.get( 0 ) ).intValue( );

ArrayList stringList = new ArrayList( );
stringList.add( "Hello" );
stringList.add( Color.RED ); // legal –
                        /* potential bug */
```

After

```
ArrayList<Integer> numList =  new ArrayList<>( );
int num = 2;

numList.add( num ); // autoboxing
int x = numList.get(0); //genercis (& autoboxing)


ArrayList<String> stringList =  new ArrayList<>( );
stringList.add( "Hello" );
// stringList.add( Color.RED );  // genercis→
            /* syntax error - compiler helps catch
bugs */
```

From EJ: Prefer primitives to boxed primitives, and watch out for unintentional autoboxing.

```
//Hideously slow program!
//Can you spot the object creation?
public static void main(String[] args) {
        Long sum = 0L;
        for (long i = 0; i < Integer.MAX_VALUE; i++) {
                sum += i;
        }
        System.out.println(sum);
}
```

The variable sum is declared as a `Long` instead of a `long`,
→ the program constructs about 2^{31} unnecessary `Long` instances
Profile on your machine…

Nested classes

Inner Class:

- Is associated with an instance of the enclosing class
- Has direct access to the enclosing class' fields and methods **(including non-public!)**
- Inner classes can not define any static members.
- For example: Iterators

Static Nested Class:

- Must have `static` in signature
- Has access to the static members of the enclosing class.
- Static Nested Classes interacts with its enclosing class in the same manner as ANY other class → more general scope
- For example: Comparators

Generics and inner classes

```
public class Foo<E> {

    private class Inner<E> {}    // incorrect

    private class Inner {}       // correct
}
```

- Inner classes can (and should) use the type parameter declared by outer class.
- Inner class should NOT redeclare the type parameter. (If it does → second type parameter with the same name will be created)

The Comparable interface

- The standard way for a Java class to define a comparison function for its objects is to implement the `Comparable` interface.
- `Comparable` provides **natural ordering**: Rules governing the relative placement of all values of a given type.

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

Generic Interface

- A call of **A**.compareTo(**B**) should return:
  a value < 0 if **A** comes "before" **B** in the ordering,
  a value > 0 if **A** comes "after" **B** in the ordering,
  or exactly 0 if **A** and **B** are considered "equal" in the ordering

→ `Comparable` MUST be compatible with `equals()`

Bounded type parameters

Upper Bound: `<Type extends SuperType>`

- Accepts the given super- type or any of its **sub**types (MUST be a subtype of SuperType)
- Works for multiple superclass/interfaces with `&` :

  `<Type extends ClassA & InterfaceB & InterfaceC & ...>`

Lower Bound: `< SuperType super Type>`

- Accepts the given type or any of its **super** types.

- Examples:

```
// can be instantiated with any animal type
public class Nest<T extends Animal> {
    ...
}
...
Nest<Bluebird> nest = new Nest<Bluebird>();
```

```
public class VeryGenericExample<T extends Comparable<? super T>>{

    private int n;
    private T[] arr;
}
```

Despite that Comparable is an interface, we must say `extends`

So How Are We Doing? Not So Well, Unfortunately☹

```
Enum<E extends Enum<E>> { ... }

<T extends Object & Comparable<? super T>> T
   Collections.max(Collection<? extends T>)

public <V extends Wrapper<? extends Comparable<T>>>
    Comparator<V> comparator() { ... }

error: equalTo(Box<capture of ?>) in Box<capture of ?>
   cannot be applied to (Box<capture of ?>)

    equal = unknownBox.equalTo(unknownBox)
Arrays.asList(String.class, Integer.class) // Warning!
```

See Angelia Langer's 427-page (!) Java Generics FAQ for more:

http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.pdf

Next week:
More on Bounded Parameters
Type Erasure in Java

Taken from The Closures Controversy
By Joshua Bloch

Regular Expressions

A simple and powerful way to find/replace substring patterns

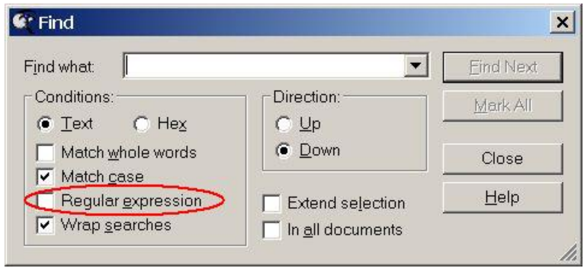
Where can I use Regular Expressions?

- Web:
  - Google Search
- Unix/Linux:
  - Command line utilities: grep, less, sed, awk
  - Editors: emacs, ed, vi
- Scripts:
  - Perl, tcl, Python, Matlab, PHP, Javascript, …
- Programming:
  - C++ libraries (boost::regex)
  - .NET (System.Text.RegularExpressions)
  - Java (we'll see today)

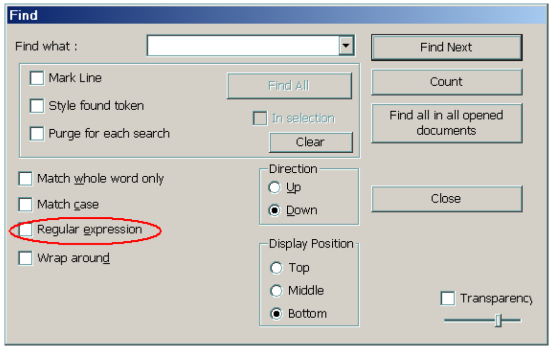
Northeastern University

Where can I use Regular Expressions?

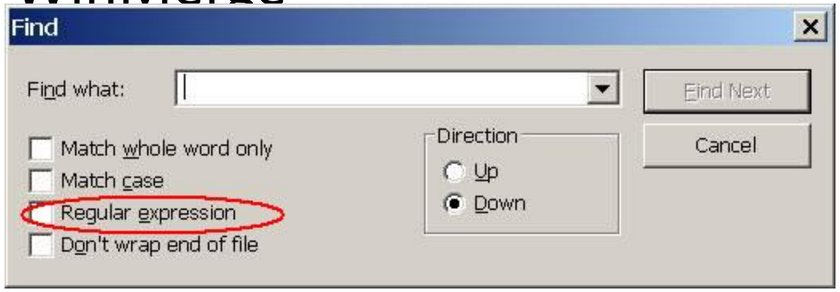
- TextPad


Find
Find what:
Conditions: [✓ Text] Hex
☐ Match whole words
☑ Match case
☐ Regular expression
☑ Wrap searches
Direction: Up [✓ Down]
☐ Extend selection
☐ In all documents

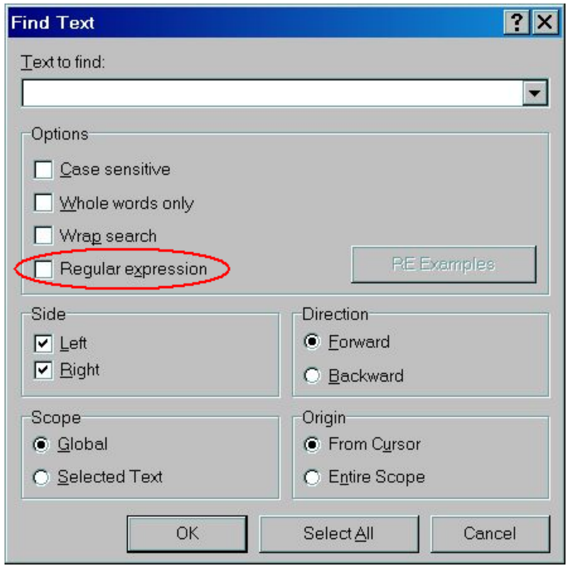
- Notepad++


Find
Find what :
☐ Mark Line
☐ Style found token
☐ Purge for each search
☐ In selection
☐ Match whole word only
☐ Match case
☐ Regular expression
☐ Wrap around
Direction: Up [✓ Down]
Display Position: Top Middle [✓ Bottom]
☐ Transparency

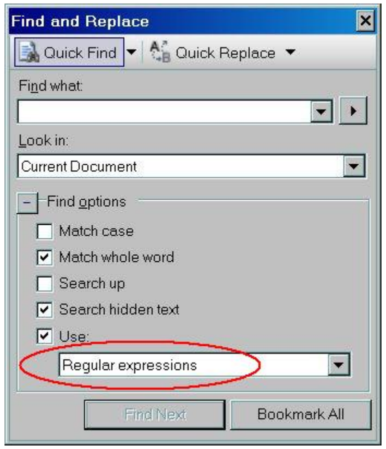
- WinMerge


Find
Find what:
☐ Match whole word only
☐ Match case
☐ Regular expression
☐ Don't wrap end of file
Direction: Up [✓ Down]

- Beyond Compare


Find Text
Text to find:
Options
☐ Case sensitive
☐ Whole words only
☐ Wrap search
☐ Regular expression
Side: ☑ Left ☑ Right
Direction: [✓ Forward] Backward
Scope: [✓ Global] Selected Text
Origin: [✓ From Cursor] Entire Scope

- Visual Studio


Find and Replace
Quick Find | Quick Replace
Find what:
Look in: Current Document
Find options
☐ Match case
☑ Match whole word
☐ Search up
☑ Search hidden text
☑ Use: Regular expressions

Regular Expression

What?

- A pattern of characters used to match strings in a search

How?

- Metacharacters - a set of special characters that have special meaning
  - **. * ? +** …
  - **\***char* – escapes the special meaning of metacharacter following \\
- Any non-metacharacter matches itself
  - **A-Z a-z 0-9 _ space …**

Metacharacters

.	Any character, except new line
[a-z**]**	Any ONE (single) character from the set
[^a-z**]**	Any ONE (single) character not in the set
*****	Zero or more of preceding character
+	One or more of the preceding characters
| or (|)	Or

Examples:

(Mr|Mrs) X	“Mr X” or “Mrs X”
[0-9A-Fa-f]	Hex digit
0x[0-9A-Fa-f]+	Hex number
\[[^\]]*\]	Text in square brackets

Anchors

^	Beginning of line
$	End of line
< or \<	Beginning of word
> or \>	End of word

Examples:

^[\t]*$	Blank line
<A[A-Za-z]*>	A word that starts with capital ‘A’
^//.*	Comment line (in C++/Java)
[0-9]*\.[0-9]*0+>	Float number with unnecessary trailing zeros

- Usually there is more than one way to do it.
- Try to use the strictest possible pattern.

Be creative - What is this one doing?

- Find all functions that took more than 1 second:

```
\] [1-9][0-9][0-9][0-9]+\.
```

- Result:

```
15:21:36:884 Ext Dbg 0160 [CFusProjectData::LoadIniFiles] 2833.1
15:21:39:259 Ext Dbg 0160 [fIPCAgent::RunServer] 1376.3
15:21:39:462 Ext Dbg 0160 [CBaseApp::InitInstance()] 5421.1
15:21:48:041 Ext Dbg 0160 [CSQLDMOService::ConnectDatabase] 1501.2
15:21:48:056 Ext Dbg 0160 [CFusDBService::InitDatabase] 2374.0
15:21:48:056 Ext Dbg 0160 [CBaseEditLogic::ConnectToDatabase()] 2375.1
15:21:49:947 Ext Dbg 0160 [CSQLDMOService::GetSpaceInfo] 1898.0
15:21:49:947 Ext Dbg 0160 [CFusDBService::IsEnoughDbSpace] 1898.3
15:21:53:681 Ext Dbg 0160 [CBaseConfigSettings::LoadIniFile()] 1746.9
15:21:53:697 Ext Dbg 0160 [CSVATProtocolModule::Init] 1764.4
15:22:25:619 Ext Dbg 0160 [CBaseEditLogic::InitializeDatbaseData] 10994.1
15:22:25:619 Ext Dbg 0160 [CEntranceLogic::SetSfResumeTreat] 18126.1
…
```

Find & Replace metacharacters

( ) or \(\)	Grouping stores matched characters in a buffer

See

http://www.ccs.neu.edu/home/skotthe/classes/cs5004/Spring/2017/InstructorNotes/09/notes.html#_regular_expressions

I/O

- http://www.ccs.neu.edu/home/skotthe/classes/cs5004/Spring/2017/InstructorNotes/09/notes.html#_file_and_input_output_i_o

Many ways to read/write your data

Reading/writing Bytes

→ use `FileInputStream`/ `FileOutputStream` classes

https://docs.oracle.com/javase/7/docs/api/java/io/FileInputStream.html

Reading/writing charachters

→ use `FileReader`/ `FileWriter` classes

http://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html

Reading/writing lines

→ **Buffer your read/write operations with**

`BufferReader`/ `BufferWriter`

Dismiss `finally`

Use:

```
try (<INITIALIZE-RESOURCES>) {

      // code that uses initialized resources

  }  // automatically closes and cleanup performed by (<…>)
```