CSC8012 Software Development Techniques and Tools — Part 2

Dr Konrad Dabrowski
konrad.dabrowski@newcastle.ac.uk

School of Computing
Newcastle University

Link to AVD:
https://rdweb.wvd.microsoft.com/arm/webclient/index.html

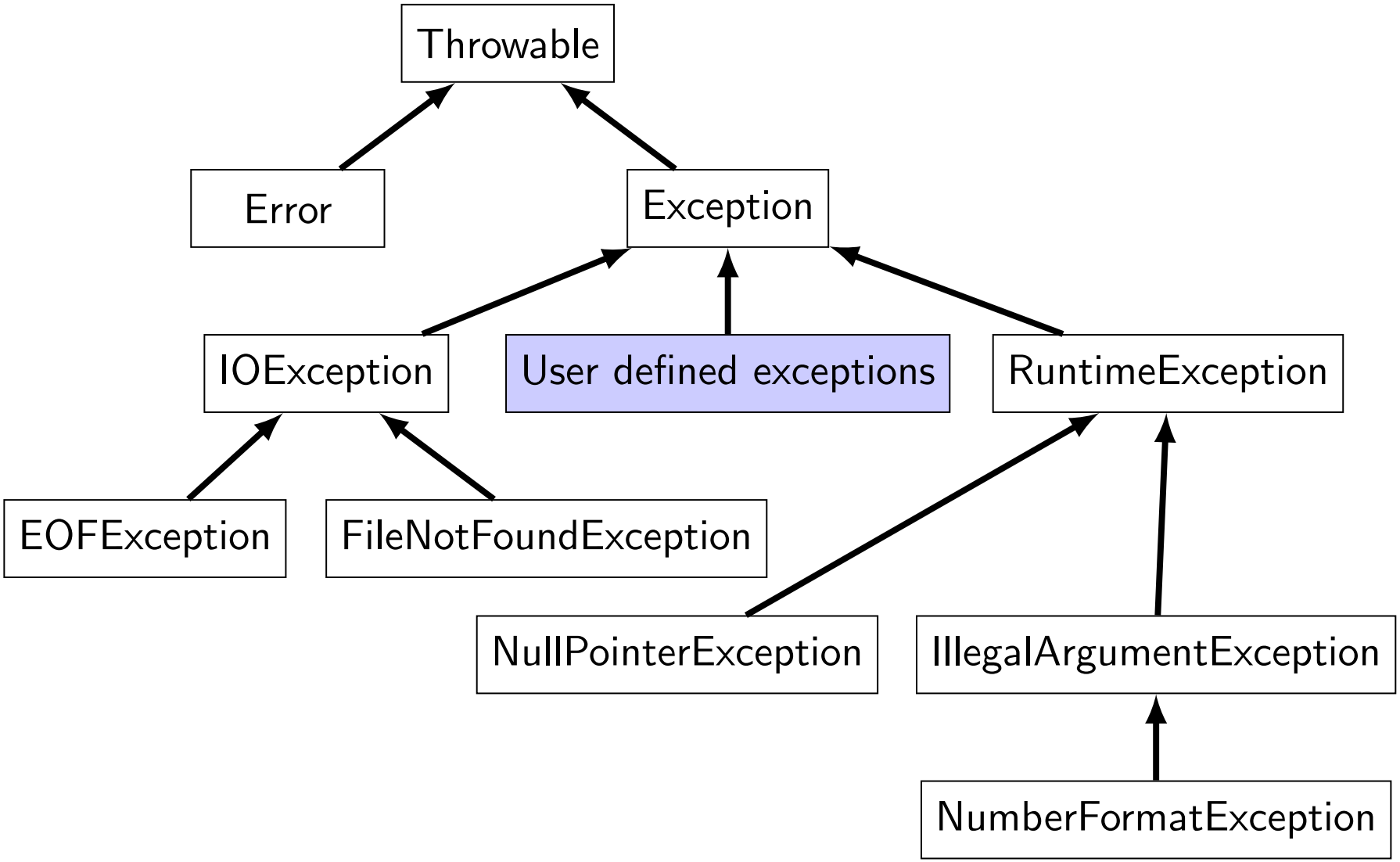
Topics

- Exceptions and exception handling
- Generic classes and methods

Exceptions

- An exception is an occurrence of a failure during program execution.
- Examples:
  - division by zero,
  - inputting a letter when a number is being read in,
  - referring to a file that cannot be found,
  - using array index which is less than 0 or greater than or equal to the length of the array.
- Java provides a number of exception classes for different types of exceptions. For the exceptions listed above, we have the following classes:
  - `ArithmeticException`
  - `NumberFormatException`
  - `FileNotFoundException`
  - `ArrayIndexOutOfBoundsException`
- Programmers are allowed to create their own exception classes.
- When an exception occurs in a method, an object of a specific exception class is created and "thrown". The exception can be either "caught" and handled by the method or propagated to a higher level.

A Hierarchy of Exception Classes



Explaining the Hierarchy

- The `Throwable` class, which is derived from the `Object` class, is the superclass of all the exception classes in Java.
- The `Error` class indicates a disaster of some kind — these are usually so serious that it is almost never worth catching them.
- The `Exception` class is the root of all classes of exceptions which are worth catching.
- The `RuntimeException` class and its subclasses indicate exceptions that are almost always the results of programming errors.
- The `IOException` class and its subclasses indicate exceptions which are thrown when something in the environment behaves badly.
- Programmers are allowed to create their own exception classes, by extending the `Exception` class or one of its subclasses.

The `Throwable` Class — Some Useful Methods

- The `Throwable` class is contained in the `java.lang` package.
- It contains some useful methods which are inherited by its subclasses:
  - `public String getMessage()`

    returns the detailed message stored in `this` object.
  - `public String toString()`

    returns the detailed message stored in `this` object as well as the name of the exception class.
  - `public void printStackTrace()`

    prints the sequence of method calls when an exception occurs.

Checked and Unchecked Exceptions

- Java's predefined exceptions are divided into two categories: checked exceptions and unchecked exceptions.
- Any exception that can be detected by the compiler is called a checked exception. For example, `FileNotFountExceptions` and `IOExceptions` are checked exceptions.
- If a checked exception occurs in a method, then the method can either handle this exception or throw it to a higher level. In the latter case, it *must* include an appropriate `throws` clause in its heading.
- An exception that cannot be detected by the compiler is called an unchecked exception. For example, `RuntimeExceptions` are unchecked exceptions.
- If an unchecked exception occurs in a method, then the method *does not need* to include any `throws` clause in its heading when throwing this exception to a higher level.

Example

```
import java.io.*;
public class ReportingExceptions
{
    static BufferedReader k = new
    BufferedReader(new InputStreamReader(System.in));
    public static void main(String[] args) throws IOException
    {
        int number;
        System.out.println("Enter integer: ");
        number = Integer.parseInt(k.readLine());
        System.out.println("You entered: " + number);
  }
}
```

- Note that:
  - The `readLine` method throws `IOException`
  - The `parseInt` method throws `NumberFormatException`

Exception Handling Statement

- To handle exceptions within a program, Java provides the `try/catch/finally` block:

```
try
{
    // statements
}
catch(ExceptionClassName1 e1)
{
    // exception handler code
}
catch(ExceptionClassName2 e2)
{
    // exception handler code
}
  ...
finally
{
    // statements
}
```

try/catch/finally Block

- A `try` block contains the code you would normally write to carry out the standard task.
- Placing the code in the `try` block recognises that one or more possible exceptions may occur during its execution.
- The `try` block is followed by zero or more `catch` blocks.
- A `catch` block takes a parameter which defines the type of exception it is prepared to deal with, and contains a code to handle this type of exception.
- The last `catch` block may be followed by a `finally` block. If a `try` block has no `catch` block, then it must have a `finally` block.
- Executing the `try/catch/finally` block:
  - If no exception occurs in a `try` block, all `catch` blocks associated with the `try` block are ignored and program execution resumes after the last `catch` block.
  - If an exception occurs in a `try` block, the remaining statements in the `try` block are ignored. The program searches the `catch` blocks (in the order in which they appear after the `try` block) looking for an appropriate exception handler (a `catch` block with the parameter type matching the type of the thrown exception). If one is found, the code of that `catch` block is executed and the remaining `catch` blocks are ignored.
  - The `finally` block (if there is one) always executes regardless of whether an exception occurs or not.

Handling Exceptions (1)

```
import java.io.*;

public class HandlingExceptions1
{
    static BufferedReader k = new
    BufferedReader(new InputStreamReader(System.in));

    public static void main(String[] args) throws IOException
    {
        int number;
        try
        {
            System.out.println("Enter integer: ");
            number = Integer.parseInt(k.readLine());
            System.out.println("You entered: " + number);
        }
        catch (NumberFormatException e)
        {
            System.out.println(e);
        }
    }
}
```

Handling Exceptions (2)

```
import java.io.*;
public class HandlingExceptions2
{
    static BufferedReader k = new
    BufferedReader(new InputStreamReader(System.in));
    public static void main(String[] args) throws IOException
    {
        int number; boolean valid = false;
        while (!valid)
        {
            try
            {
                System.out.println("Enter integer: ");
                number = Integer.parseInt(k.readLine());
                System.out.println("You entered: " + number);
                valid = true;
            }
            catch (NumberFormatException e)
            {
                System.out.println("Incorrect format, try again");
            }
        }
    }
}
```

Handling Exceptions (3)

```
import java.io.*;
public class HandlingExceptions3
{
    static BufferedReader k = new
    BufferedReader(new InputStreamReader(System.in));
    public static void main(String[] args)
    {
        int number;
        try
        {
            System.out.println("Enter integer: ");
            number = Integer.parseInt(k.readLine());
            System.out.println("You entered: " + number);
        }
        catch (NumberFormatException e1)
        {
            System.out.println(e1);
        }
        catch (IOException e2)
        {
            System.out.println(e2);
        }
    }
}
```

Order of `catch` Blocks

- In our last example we had two `catch` blocks. They could have been written in any order, because their parameter types are classes which are not in a subclass - superclass relationship.
- In a sequence of `catch` blocks following a `try` block, `catch` blocks handling exceptions of a subclass type should be placed before `catch` blocks for handling exceptions of a superclass type.
- The following `try/catch` block will produce a compile time error.

```
try
{
    ...
}
catch(Exception e1)
{
    ...
}
catch(NumberFormatException e2)
{
    ...
}
```

Re-throwing and Throwing an Exception

- When an exception occurs in a `try` block, and is caught in one of the `catch` blocks, there are three possibilities to handle it:
  - re-throw the same exception for the calling environment to handle
  - partially process the exception, and then re-throw the same exception or throw another exception for the calling environment to handle
  - completely handle the exception
- To throw or re-throw an exception we use the `throw` statement:

```
throw exceptionReference;
```

- For example, when re-throwing the same exception we write:

```
catch (Exception e)
{
    throw e;
}
```

- To throw another exception we can use a statement of the form:

```
throw new ExceptionClassName(messageString);
```

The ThrowingExceptions Class

```
import java.io.*;

public class ThrowingExceptions
{
    static BufferedReader k = new
    BufferedReader(new InputStreamReader(System.in));

    public static void main(String[] args) throws IOException
    {
        int number;
        int i = 3;
        boolean valid = false;

        // while-loop
    }
}
```

The ThrowingExceptions Class - the while Loop

```
while (!valid)
{
    try
    {
        System.out.println("Enter integer: ");
        number = Integer.parseInt(k.readLine());
        System.out.println("You entered: " + number);
        valid = true;
    }
    catch (NumberFormatException e)
    {
        i--;
        if (i > 0)
        {
            System.out.println("Incorrect format, try again");
            System.out.println("You have " + i + " more chance(s)");
        }
        else
        throw new NumberFormatException("after 3 attempts");
    }
}
```

Defining New Exception Classes

- We can create new exception classes by extending either the `Exception` class or one of its subclasses.
- The `Exception` class is derived from the `Throwable` class, so it (and its subclasses) inherits the methods like `getMessage` and `toString`.
- Example:

```
public class MyException extends Exception
{
    public MyException(String s)
    {
        super(s);
    }
}
```

- Once the new exception class is defined, it can be used in a similar way to the predefined Java exception classes. However, the exceptions of this new type are not thrown by the Java Virtual Machine. They can only be thrown by using the `throw` statement.
- If the new exception class is a direct subclass of the `Exception` class or a direct subclass of an exception class whose exceptions are checked exceptions, then the exceptions of the new class are checked exceptions.

Example

- Consider the following `monthName` method:

```
private static String monthName(int monthNumber)
{
    String name;
    switch(monthNumber)
    {
        case 1:  name = "January";    break;
        ...
        case 12: name = "December";   break;
        default: name = "No name";
    }
    return name;
}
```

- We do not want our method to decide what should be done in the case of exceptional behaviour. The method should rather inform the calling program that some exception might occur, and allow the program to determine the correct response.

The Altered monthName Method

```
private static String monthName(int monthNumber) throws
                                                InvalidDateException
{
    String name;
    switch(monthNumber)
    {
        case 1:  name = "January";    break;
        case 2:  name = "February";   break;
        case 3:  name = "March";      break;
        case 4:  name = "April";      break;
        case 5:  name = "May";        break;
        case 6:  name = "June";       break;
        case 7:  name = "July";       break;
        case 8:  name = "August";     break;
        case 9:  name = "September";  break;
        case 10: name = "October";    break;
        case 11: name = "November";   break;
        case 12: name = "December";   break;
        default: throw new InvalidDateException("Invalid month number:
                                                + monthNumber);
    }
    return name;
}
```

Defining the InvalidDateException Class

```
public class InvalidDateException extends Exception
{
    public InvalidDateException()
    {
        super("Invalid date");
    }

    public InvalidDateException(String s)
    {
        super(s);
    }
}
```

Using Our New Exception Class

```
import java.util.*;
public class TestingInvalidDateException
{
    static Scanner k = new Scanner(System.in);
    public static void main(String[] args)
    {
        int number;
        try
        {
            System.out.print("Enter integer: ");
            number = k.nextInt();
            System.out.println("Month name: " + monthName(number));
        }
        catch (InvalidDateException e)
        {
            System.out.println(e);
        }
    }
    private static String monthName(int monthNumber) throws
                                                InvalidDateException
    {...}
}
```

Generic Programming

- Generic programming is the creation of programming constructs that can be used with many different types.
- In Java, generic programming can be achieved with inheritance or with type variables (discussed later).
- *Inheritance* allows us to assign an object of any type to a reference variable of type `Object`.
- *Type variables* can be substituted with any class or interface type.
- However, you cannot treat values of primitive types as objects. Similarly, you cannot substitute any of the primitive types for a type variable. Java's solution to this problem is *wrapper classes*. For example:
  - The wrapper class for `int` is called `Integer`.
  - The wrapper class for `char` is called `Character`.
- Whenever a value of a primitive type is used in a context that requires a wrapper type, it is automatically converted to an appropriate wrapper object. This is called *autoboxing*. The reverse operation, *unboxing*, is also performed automatically.

Using the Object Class to Achieve Genericity (1)

```
public class PairOfObjects
{
    private Object first;
    private Object second;

    public PairOfObjects(Object firstElement, Object secondElement)
    {
        first = firstElement;
        second = secondElement;
    }
    public Object getFirst()
    {
        return first;
    }
    public Object getSecond()
    {
        return second;
    }
}
```

Using the PairOfObjects Class

```
public class UsingPairsOfObjects
{
    public static final int NUMBER_OF_LETTERS = 26;
    public static void main(String[] args)
    {
        PairOfObjects[] pairs = new PairOfObjects[NUMBER_OF_LETTERS];
        int i = 0;
        for (char c = 'a'; c <= 'z'; c++)
        {
            pairs[i] = new PairOfObjects(c,ordinalNumber(c));
            i++;
        }

        for (PairOfObjects p: pairs)
            System.out.println("Ordinal number of " + p.getFirst()
            + " is " + p.getSecond());
    }

    private static int ordinalNumber(char ch) { return (int) ch; }
}
```

Using Object Class to Achieve Genericity (2)

```
public class ArraysOfObjects
{
    public static void main(String[] args)
    {
        String[] strings = {"Adam", "Mary", "William", "Julia"};
        Integer[] integers = {2, 67, 45, 3, 18, -36};
        printObjectArray(strings);
        printObjectArray(integers);
        Object[] objects = {"Mary", 5, 7, "Robert"};
        printObjectArray(objects);
    }


    private static void printObjectArray(Object[] objects)
    {
        for (int i = 0; i < objects.length; i++)
            System.out.print(objects[i] + " ");
        System.out.println();
    }
}
```

Using Type Variables to Achieve Genericity

- In Java, generic programming can be achieved with *type variables*.
- The `java.util.ArrayList<E>` class is a generic class, which has been declared with a type variable `E`. The type variable denotes the element type:

```
public class ArrayList<E>
{
    public ArrayList() {...}
    public void add(E element) {...}
    ...
}
```

- Type variables can be instantiated with class or interface types, or wrapper types of primitive types. For example:

```
ArrayList<Triangle> triangles = new ArrayList<Triangle>();
ArrayList<Polygon> polygons = new ArrayList<Polygon>();
ArrayList<Integer> integers = new ArrayList<Integer>();
```

Defining a Generic Class

- We can define a *generic class* with fields and methods that depend on type variables using the following syntax:

```
accessSpecifier class GenericClassName<TypeVar1, TypeVar2, ...>
{
    fields
    constructors
    methods
}
```

- The type variables `TypeVar1`, `TypeVar2` etc. can be used for the types of generic fields, method parameters, and the return types of value-returning methods.

Example - the Pair Class

```
public class Pair<T,S>
{
    private T first;
    private S second;

    public Pair(T firstElement, S secondElement)
    {
        first = firstElement;
        second = secondElement;
    }

    public T getFirst()
    {
        return first;
    }

    public S getSecond()
    {
        return second;
    }
}
```

Generics and Arrays

- Arrays and generics do not mix well. Consider the program below:

```
public class UsingPairs
{
    public static final int NUMBER_OF_LETTERS = 26;
    public static void main(String[] args)
    {
        Pair<Character,Integer>[] pairs = new Pair[NUMBER_OF_LETTERS];
        // the above statement gives compiler warnings
        // Pair<Character,Integer>[] pairs = new
        //           Pair<Character,Integer>[NUMBER_OF_LETTERS]; the above
        // statement produces compilation error: generic array creation
        int i = 0;
        for (char c = 'a'; c <= 'z'; c++)
        {
            pairs[i] = new Pair<Character,Integer>(c,ordinalNumber(c));
            i++;
        }
        for (Pair<Character,Integer> p: pairs)
            System.out.println("Ordinal number of " + p.getFirst() + " is "
                                                    + p.getSecond());
    }
    private static int ordinalNumber(char ch){ return (int) ch; }
}
```

Using ArrayList Instead

```
import java.util.*;
public class ArrayListOfPairs
{
    public static void main(String[] args)
    {
        ArrayList<Pair<Character,Integer>> pairs =
        new ArrayList<Pair<Character,Integer>>();
        for (char c = 'a'; c <= 'z'; c++)
        {
            Pair<Character,Integer> pair =
            new Pair<Character,Integer>(c,ordinalNumber(c));
            pairs.add(pair);
        }

        for (Pair<Character,Integer> p: pairs)
            System.out.println("Ordinal number of " + p.getFirst()
            + " is " + p.getSecond());
    }
    private static int ordinalNumber(char ch) { return (int) ch; }
}
```

Defining a Generic Method

- Generic methods can be defined inside ordinary and generic classes.
- We can define a *generic method* that depends on type variables using the following syntax:

```
modifiers <TypeVar1, TypeVar2, ...> returnType
                                        methodName(parameters)
{ body }
```

- For example:

```
public static <E> void print(E[] a)
{
    for(E e: a)
        System.out.print(e + " ");
    System.out.println();
}
```

- Type variables declared at the class level can only be used by non-static members and methods. So generic methods are particularly useful for defining static generic methods.
- When calling a generic method, it is necessary to instantiate the type variables.

Using Generic Methods

```
public class GenericArrays
{
    public static void main(String[] args)
    {
        String[] strings = {"Adam", "Mary", "William", "Julia"};
        Integer[] integers = {2, 67, 45, 3, 18, -36};
        printArray(strings);
        printArray(integers);
    }

    private static <E> void printArray(E[] a)
    {
        for (E elem: a)
            System.out.print(elem + " ");
        System.out.println();
    }
}
```

Constraining Type Variables

- Sometimes it is necessary to put constraints on types that can be substituted for type variables.
- The place to specify constraints, for both generic classes and methods, is between the angle brackets, where the type variables are declared.
- For example we can add the following method to the `Pair<T,S>` generic class:

```
public <E extends Number> void operation(E value)
{
    if (value instanceof Integer) System.out.println(first);
        else
            if (value instanceof Double) System.out.println(second);
                else
                    System.out.println(first + " " + second);
}
```

- The reserved word `extends` in the declaration `<E extends ClassOrInterface>` means "extends or implements" in this context.

Using Generic Non-static Methods

```
public class TestPairOperation
{
    public static void main(String[] args)
    {
        Integer i = 5;
        Double d = -3.5;
        Byte b = 0x07;
        String s = "test";
        Pair<Character,String> pair =
        new Pair<Character,String>('a',"aa");
        pair.operation(i);
        pair.operation(d);
        pair.operation(b);
        pair.operation(s); // compile-time error
    }
}
```

Constraining Type Variables in Generic Classes

```
public class NumberPair<T extends Number, S extends Number>
{
    private T first;
    private S second;

    public NumberPair(T firstElement, S secondElement)
    {
        first = firstElement;
        second = secondElement;
    }

    public Double add()
    {
        return  first.doubleValue() + second.doubleValue();
    }
}
```

The Comparable<T> Interface

- Comparable<T> is a generic interface containing only one method:

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

- This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*.
- The compareTo method compares this object with the specified object, other, for order. It returns a negative integer, zero or a positive integer if this object is less than, equal to or greater than the other object, respectively.

A New Person Class

```
public class Person implements Comparable<Person>
{
    private String firstName;
    private String lastName;
    private int age;

    public Person(String firstName, String lastName, int age)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        if (age < 0) throw new IllegalArgumentException();
        this.age = age;
    }

    public int compareTo(Person p)
    {
        int lnCmp = lastName.compareTo(p.lastName);
        if (lnCmp !=0) return lnCmp;
        int fnCmp = firstName.compareTo(p.firstName);
        if (fnCmp != 0) return fnCmp;
        else return age - p.age;
    }

    public String toString() { ... }
}
```

The Person Class - the toString Method

```
public class Person implements Comparable<Person>
{
    private String firstName;
    private String lastName;
    private int age;

    ...

    public String toString()
    {
        return firstName + " " + lastName + " - " + age;
    }
}
```

Ordering Person Objects

```
import java.util.*;
public class OrderPersons
{
    public static void main(String[] args)
    {
        Person mary = new Person("Mary", "Williams", 25);
        Person john = new Person("John", "Brown", 28);
        Person michael = new Person("Michael", "Brown", 72);
        Person littleMary = new Person("Mary", "Williams", 1);
        ArrayList<Person> persons= new ArrayList<Person>();
        persons.add(mary); persons.add(john);
        persons.add(michael); persons.add(littleMary);
        Collections.sort(persons);
        System.out.println(persons);
    }
}
```

Type Variables Restricted to Comparable Types

```
public static <E extends Comparable<E>> E min(ArrayList<E> a)
{
    E minSoFar = a.get(0);
    for (E elem: a)
        if (elem.compareTo(minSoFar) < 0) minSoFar = elem;
    return minSoFar;
}
```

- We can weaken the constraints on the type variable `E` in the above method to widen the set of valid types:

  `<E extends Comparable<? super E>>`

- The above type declaration means that `E` implements the `Comparable` interface or extends a class that does this.

Wildcard Types

- *Wildcards* are type arguments in the form of `?`, possibly with an upper or lower bound.

Name	Syntax	Meaning
Wildcard with upper bound	`? extends E`	Any subtype of `E`
Wildcard with lower bound	`? super E`	Any supertype of `E`
Unbounded wildcard	`?`	Any type

Using Wildcards

```
import java.util.*;
public class ListsOfNumbers
{
    public static void main(String[] args)
    {
        List<Integer> myInts = Arrays.asList(1,2,3,4);
        List<Double> myDoubles = Arrays.asList(3.14, 6.28);
        List<Object> myObjs = new ArrayList<Object>();
        copy(myInts, myObjs);
        copy(myDoubles, myObjs);
        printList(myObjs);
    }

    private static void copy(List<? extends Number> source,
                             List<? super Number> destination)
    {
        for(Number number : source) destination.add(number);
    }

    private static <E> void printList(List<E> a) { ... }
}
```

ListsOfNumbers - the printList Method

```
import java.util.*;

public class ListsOfNumbers
{
    public static void main(String[] args) { ... }

    private static void copy(List<? extends Number> source,
                             List<? super Number> destination)
    { ... }

    private static <E> void printList(List<E> a)
    {
        for (E elem: a)
        System.out.print(elem + " ");
        System.out.println();
    }
}
```