Java Programming 2 – Lecture #7 – Jeremy.Singer@glasgow.ac.uk

Static Methods

Recall that `static` methods are associated with a class rather than any particular instance. These `static` methods are generally utility methods. Examples include:

- `Math.random()` which returns a `double` value in the range [0,1)
- `System.exit(int status)` which terminates all threads and aborts the running JVM
- `Integer.parseInt(String s)` which tries to interpret the parameter `s` as a 32-bit integer value

Exceptions

When errors occur in program execution, `Exception` objects are *thrown*. All `Exception` objects belong to classes that are subclasses of `java.lang.Exception`[1]. Some `Exception` objects are subclasses of `RuntimeException` – these are *unchecked*. All other `Exception`s are *checked*, and if they may be thrown then they must be caught or declared in the enclosing method's `throws` clause.

An example of a checked exception is `FileNotFoundException`. An example of an unchecked exception is `ArrayIndexOutOfBoundsException`.

It is possible to instantiate and `throw` exceptions directly in your own code, i.e.

```
throw new Exception();
```

Customized exceptions can be created – either (1) by supplying an error message `String` in the `Exception` constructor (the String can be retrieved via the `Exception.getMessage()` instance method) – or (2) by extending the `Exception` class and possibly adding new instance fields.

Handling Exceptions

A `try` block should enclose code that may throw an `Exception` instance. A `try` block may be followed by one or more `catch` blocks, each of which takes a single `Exception` parameter. The `catch` blocks are evaluated in sequential order, and the first `catch` block whose parameter type matches the thrown exception is executed. A `try` block may also be associated with a `finally` block, which is executed either after the non-exceptional exit from the `try` block, or after any matching `catch` block has been executed. Example source code is shown below:

[1] See http://www.oracle.com/technetwork/articles/entarch/effective-exceptions-092345.html for a discussion of Exceptions in Java.

```
try { …
}
catch (Exception e) { …
}
finally { …
}
```

Abstract Classes and Methods

Some superclasses have 'holes' in them, which subclasses can 'fill in' when they extend the superclass. The `holey' superclasses are marked as `abstract` classes, which have `abstract` methods declared in them. The `abstract` class only defines a partial implementation. An `abstract` class cannot be instantiated. An `abstract` method only has a signature and no method body, thus it cannot be called. A subclass of an `abstract` class must supply an implementation for the inherited `abstract` methods, or the subclass itself must be marked as `abstract`.

The `abstract` method mechanism is a way to enforce that subclasses conform to a particular API. An example is shown below. All subclasses of `TwoDimensionalPoint` must implement the `distanceToOrigin()` method.

```
public abstract class TwoDimensionalPoint {
  double x;
  double y;
  public abstract double distanceToOrigin();
}

public class CartesianPoint extends TwoDimensionalPoint {
  public double distanceToOrigin() {
    return Math.sqrt(x*x+y*y);
  }
}

public class ManhattanPoint extends TwoDimensionalPoint {
  public double distanceToOrigin() {
    return Math.abs(x) + Math.abs(y);
  }
}
```

Questions

1) Can an `abstract` class have constructors? If so, why? If not, why not?
2) What is the relationship between an `abstract` class and an `interface`?