# Java Exceptions; equals/hashCode; enums

“Orange juice” by Gloria García
https://flic.kr/p/4Vz65X

# Exceptions

When an error occurs in program execution, an `Exception` is **thrown**

*(Exceptions are also Java objects like any other; parent class is `java.lang.Exception`)*

Unless the exception is **caught**, the entire program will crash

## Checked and unchecked exceptions

### UNCHECKED EXCEPTIONS

Do not need to be explicitly handled

- Program will still compile and run without any special handling

Generally indicate **programming/logic bugs** that an application cannot reasonably recover from

Example: `ArrayIndexOutOfBoundsException`

### CHECKED EXCEPTIONS

Must be explicitly handled

- Program will not compile unless you deal with them somehow

Generally indicate conditions that a well-written application should anticipate and recover from

Example: `FileNotFoundException`

## Handling exceptions #1: Catching

Wrap a `try {}` block around any code that might throw an Exception

Must be followed by one (or more) `catch {}` blocks

- First one whose parameter matches the thrown exception is executed

Optional `finally {}` block

- Executed after entire rest of the try block

```
try {
        // code that might
        // throw Exception
} catch (Exception ex) {
        // deal with it
} finally {
        // clean up
}
```

## Handling exceptions #2: Passing on

If you do something that might throw an exception, you can add that exception to the `throws` clause of the current method

Then anyone who calls your method will need to handle the exception (by catching or passing on)

```
public void doSomething()
     throws IOException
{
     // code that might
     // throw IOException
}
```

## Throwing an Exception

Use the **throw** keyword:

```
  throw new Exception ("Invalid input");
```

You can throw an Exception at any point in your code

String parameter indicates the message (available through `ex.getMessage()`)

If you throw a checked Exception, you also need to add it to the header of your method with the **throws** keyword

```
  public String processInput (String input) throws Exception { … }
```

## Advantages of using Exceptions

1. Separating out error-handling code
   - Instead of a series of if/then/else statements
   - Just “assume” that things will work and deal with errors elsewhere
2. *Propagating* errors up the call stack
   - i.e., sending errors along until they reach a method that is prepared to handle them
3. Grouping error types
   - Exception is a class, and can be subclassed
   - => different types of Exceptions can be conceptually grouped together
   - (I/O exceptions, for example)

# Methods of `java.lang.Object`

```
protected Object clone()
boolean equals (Object obj)
protected void finalize()
public Class<?> getClass()
public int hashCode()
public void notify() / notifyAll()
public String toString()
public void wait() / wait(long timeout) / wait(long
timeout, int nanos)
```

# java.lang.Object.equals() documentation

Indicates whether some other object is “equal to” this one

The `equals` method implements an equivalence relation on non-null object references:

- It is ***reflexive***: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is ***symmetric***: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is ***transitive***: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is ***consistent***: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

# Default implementation of equals()

"The most discriminating possible equivalence relation on objects"

- Returns true **if and only if** x and y refer to the **same** object (i.e., x == y is true)

Gives the correct result for primitive types (int, double, char, etc.)

Does not check if objects are **equivalent** – i.e., if their contents are the same

```
ArrayList<Integer> l1 = new ArrayList<>();
l1.add (1);
ArrayList<Integer> l2 = new ArrayList<>();
l2.add (1);
boolean result = l1.equals(l2); // Default would return false
```

# java.lang.Object.hashCode() documentation

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

## Default implementation

“As much as is reasonably practical, the `hashCode` method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)”

## equals() and hashCode()

"If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects **must produce the same integer result.**"

So if you override `equals()`, **you must also override `hashCode()`**

# Enumerated types

An enum type is a special data type that allows a variable to be one of a set of predefined constants

Common examples:

- Compass directions (NORTH, SOUTH, EAST, WEST)
- Days of week, months of year, etc.

## Declaring an enum in Java

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY,
  }
```

Note: values are constants ==> conventionally written in ALL_CAPS

You use the `enum` keyword **instead of** `class`

An enum called Day should be in a class Day.java

# An enum is a special class

- It has **methods**
  - Built-in static method values() that returns an array of all values
  - Built-in static method valueOf() that parses a string into an enum constant
  - Appropriate definitions of compareTo(), equals(), hashCode(), toString()
  - Other methods:
    - ordinal() -- returns the position of this constant in the list
    - name() -- returns the name of this constant
  - Any other methods that you define
- You can define **fields** as well if necessary