Useful predefined classes

Class `java.lang.Object`

A short selection of public object methods:

- **`public int`** `hashCode()`

  returns the hash code of object **`this`**

- **`public boolean`** `equals(Object obj)`

  returns **`this`** `== obj`

- **`public`** `String toString()`

  returns
  `getClass().getName()+"@"+Integer.toHexString(hashCode())`

Useful predefined classes

Class `java.util.Objects`

A short selection of public class methods:

- **`public static`** `<T> T requireNonNull(T obj)`

  if `obj==`**`null`** then throws NullPointerException
  otherwise returns `obj`

- **`public static boolean`** `equals(Object a, Object b)`

  if `a!=`**`null`** then returns `a.equals(b)`
  otherwise returns `a==b`

- **`public static int`** `checkIndex(`**`int`** `index,` **`int`** `length)`

  if `index<0||index>=length` then throws IndexOutOfBoundsException
  otherwise returns `index`

- **`public static`** `String toString(Object o)`

  if `o!=`**`null`** then returns `o.toString()`
  otherwise returns `"null"`

Useful predefined classes

Example

```
import java.util.Objects;
import static java.util.Objects.requireNonNull;
import static java.util.Objects.checkIndex;
...
   name = requireNonNull(s);  // checks if s!=null
   r = Objects.equals(a,b);   // never throws NullPointerException
   index = checkIndex(i,100); // checks if 0 <= i < 100
   s = Objects.toString(o);   // never throws NullPointerException
```

Remarks

- `equals(Object)` and `toString()` are object methods in `java.lang.Object`
- `equals(Object,Object)` and `toString(Object)` are class methods in `java.util.Objects`
- do not use **`import static`** `java.util.Objects.equals/toString`: `equals/toString` in `Object` have the precedence

Generic methods

Some details

- `<T> T requireNonNull(T obj)` is a generic method
- `<T>` is the syntax to declare a type variable
- if needed, more type variables can be declared: `<T1,T2>`
- in OCaml the type of `requireNonNull` would be `'a → 'a`

Generic methods

Parametric versus subtype polymorphism

parametric polymorphism: `<T> T requireNonNull(T obj)`

subtype polymorphism: `Object noGenRequireNonNull(Object obj)`

Example

```
public static <T> T requireNonNull(T obj) {              // correct
    if (obj == null)
        throw new NullPointerException();
    return obj;
}

public static Object noGenRequireNonNull(Object obj) { // correct
    if (obj == null)
        throw new NullPointerException();
    return obj;
}
```

Remark

`noGenRequireNonNull()` is not very useful

Generic methods

Example

```
class Person {
   private final String name;
   public Person(String name){
      this.name = nonGenRequireNonNull(name); // error, Object≰String
      this.name = requireNonNull(name);       // ok, String≰String
   }
   ...
}
```

Useful predefined classes

Class `java.lang.StringBuilder`

A more efficient way to manipulate strings

- `String`: immutable objects
- `StringBuilder`: mutable objects

A short selection of public object methods

- `StringBuilder append(String str)` (and other overloaded versions)

  appends `str` to **`this`** and returns it
- `StringBuilder delete(int start, int end)`

  removes from **`this`** the characters from `start` to `end-1`, returns **`this`**
- `String toString()`

  returns a string converted from **`this`**
- **`char`** `charAt(int index)` and **`int`** `length()`

  as in `String`

Useful predefined classes

Example

```
StringBuilder sb = new StringBuilder("hello");
sb.append(" ").append("world"); // method chaining: append returns this
assert sb.toString().equals("hello world");
sb.delete(0, 6); // removes "hello "
assert !sb.equals("world"); // objects of different classes
assert sb.toString().equals("world");
assert sb.length() == 5;
assert sb.charAt(4) == 'd';
```

Remark: both `String` and `StringBuilder` implements `CharSequence`

Implicit string conversion

Example

```
class Point {
   private int x, y;
   @Override // overrides the method defined in Object
   public String toString() { return "(" + x + "," + y + ")"; }
}
// some tests
assert (1 + "2").equals("12");
assert ("1" + 2).equals("12");
assert ("1" + 2 + 3).equals("123"); // beware of associativity!
assert (1 + 2 + "3").equals("33");  // beware of associativity!
assert (null + "_string").equals("null_string");
assert ("string_" + null).equals("string_null");
assert (new Point() + "_string").equals("(0,0)_string");
assert ("string_" + new Point()).equals("string_(0,0)");
```

Details

- primitive types converted to wrapper classes, then `toString()` is called
- for non-null references `toString()` is called, **`null`** converted to `"null"`
- see `String valueOf(Object obj)` in `String`
- see also `print(Object)` and `println(Object)` in `PrintStream`