Lecture 05: Java Safari

May 12, 2022

Primitive Types in Java:

- `boolean`
- `byte`
- `short`
- `char`
- `int`
- `float`
- `long`
- `double`

(see slides for details for size, description, and ranges)

Characters

What is a `char`?

- Letters (including unicode)
- Digits
- Punctuation
- Whitespace

How can we use them?

- As a test of character
  - `Character.isLetter()` and `Character.isDigit()`
- As elements of strings
  - `String.charAt(int n)` and `String.indexOf(char c)`
- Arrays of Characters

When should we use Characters?

- Processing a String character by character
  - Tokenizing a String into words
- `Duration.format()`
- Representing text-like values that are always of length 1
  - ie. keystrokes in a GUI

Enumerations

What are they?

- Syntax: `enum Name {VAL1, VAL2, ...}`
- Examples:
  - `enum Status {PLAYING, STALEMATE, WON}`
  - `enum BinaryOperation {ADD, SUB, MUL, DIV, POW}`

Usage

- If-Statements

What they can be

Example:

```
enum USCoin {
    PENNY(1), NICKEL(5), DIME(10),
    QUARTER(25), HALF_DOLLAR(50), DOLLAR(100);


    private final int cents;

    public int getCentsValue() {
      return cents;
    }

    USCoin(int cents) {
      this.cents = cents;
    }
}
```

When should we use them?

- To represent a finite set
- The options are known at compile time
- The options aren't too many to list
- No need to extend another class

The Switch Statement

Don't write this:

```
if (c == 's') {
  //...
} else if (...) {
  // ...
} else if (...) {
  // ...
} else if (...) {
  // ...
} else if (...) {
  // ...
} else if (...) {
  // ...
} else if (...) {
  // ...
} else if (...) {
  // ...
}
```

Do write this:

```
switch (c) {
  case 's':
    //do something
    break;
  case: 'm':
    //do something
    break;
  case: "etc":
    // do something
    break;
  default:
    unrecognized();
}
```

Switch for enums

```
 switch (op) {
  case ADD: return a + b;
  case SUB: return a - b;
  case MUL: return a * b;
  // etc.
}
```

When should we use them?

- A multi-way branch that depends on specific values of a variable
- Works on primitive types, enums, and (as of Java 7) Strings

Arrays

- If `T` is a Java type, then an array of type `T[]` is a:
    - mutable
    - fixed length
    - constant-time indexed
    - sequence of values of type `T`

How To Use Them:

Construction:

```

int[] array1 = new int[] {2, 4, 6, 8};
int[] array2 = new int[64]; // autoinit'd to 0
int[] array3 = new String[21]; // autoinit'd to null

//OR

int[] array3 = {2, 4, 6, 8}
```

Aliasing

```
 int[] a1 = new int[16];
int[] a2 = new int[16];
int[] a3 = a1;

assertEquals(0, a1[7]);
assertEquals(0, a2[7]);
assertEquals(0, a3[7]);

a1[7] = 1;
assertEquals(1, a1[7]);
assertEquals(0, a2[7]);
assertEquals(1, a3[7]);

a2[7] = 2;

assertEquals(1, a1[7]);
assertEquals(2, a2[7]);
assertEquals(1, a3[7]);
```

Copying

```
 int[] a1 = new int[16];
int[] a2 = new int[16];

// create a copy
int[] a3 = new int[a1.length];
for(int i = 0; i < a1.length; i++) {
  a3[i] = a1[i];
}

//or
int[] a3 = Arrays.copyOf(a1, a1.length);
// creates a copy, so no more aliasing
```

Testing

```
 int[] a1 = new int[16];
int[] a2 = new int[16];
int[] a3 = a1;

assertEquals(a1, a2); //FAIL
assertEquals(a1, a3); //PASS
assertArrayEquals(a1, a2); //PASS
assertArrayEquals(a1, a3); //PASS
```

Using Arrays

```
// Problem: find the max of some numbers
int max(int one, int two);
int max(int one, int two, int three);

// use an array
int max(int[] nums) {
  int maxN = nums[0];

  for(int n : nums) {
    if(n > maxN) {
      maxN = n;
    }
  }

  return maxN;
}

// usage
max(new int[]{3,4,5}); // returns 5
// would like to do:
max(3, 4, 5);
max(3, 6, 7, 8, 9, 10);

// varargs - variable # of arguments
// syntax (type... name)
int max(int... nums) {
  int maxN = nums[0];

  for(int n : nums) {
    if(n > maxN) {
      maxN = n;
    }
  }

  return maxN;
}
```

Static Methods

```
 List<T> Arrays.asList(T... elements);
int Arrays.binarySearch(int[] array, int key);
T[] Arrays.copyOfRange(T[] original, int from, int to);
boolean Arrays.equals(Object[] a1, Object[] a2);
boolean Arrays.deepEquals(Object[] a1, Object[] a2);

// see java.util.Arrays for more
```

When To Use Arrays

- An existing API requires it
- To ensure sequence length is fixed
- Efficiency, especially when implementing higher-level data structures

Primitive Types vs. Reference Types

- Boxed Types
- Objects and Primitives are treated differently

Equality

- With box types, `==` is fuzzy with boxed types
    - Use `.equals()` with boxed types
- Static versions of Object methods

When do use a primitive vs boxed

- Most Java programmers prefer primitives
    - Primitives are immediate, no null, no supertype, `==`
    - Goes against OO, but is useful
- Boxed data are references
-

Good Practices

Null Values

- What is `null`?
    - Use it only to represent the absence of a reference
    - Use it conscientiously and sparingly
    - Carefully document where it's allowed
    - Check for it and fail *FAST*

Equality

- Shallow vs Deep - same memory, but same memory contents?
- Intentional vs Extensional - Same *thing*, or calculates to the same thing?
- Nominal vs. Structural - are objects of two classes w/ same instance variables and values equal?
- Physical v. Logical - are HMSDuration and CompactDuration objects w/ the same duration equal?

`==` vs `.equals()`

- `==` is shallow, intentional, nominal, and physical
- `.equals()` is deep or shallow, extensional, as nominal or structural as you like, and logical and you choose the logic

Overriding `.equals()`

1. Fast path
2. Instance-of
3. Cast
4. Check fields

Rules of `.equals()`

- Reflexivity: `x.equals(x)`
- Symmetry: `x.equals(y) IFF y.equals(x)`
- Transitivity: `if x = y and y = z then x = z`

Rules of `.hashCode()`:

- Compatibility: if `x.equals(y)`, then `x.hashCode() == y.hashCode()`
- There's another one that I didn't get in

So what is `==`?

- Fundies 2: Don't use ==
- Now: Understand what it means and use it appropriately
    - `==` compares immediate memory locations

Understanding `instanceof`

- Fundies 2: `instanceof` is evil
- OOD: `instanceof` should be used sparingly but correctly

Necessary Example:

- Overriding `.equals()`
- Downcasts would be rare, and therefore `instanceof` would be rare too

Exceptions

- Alternative to checking if every line fails
- Try-Catch: `java try { // code } catch (Exception e) { // cleanup }`

Types of Exceptions

- **Checked Exceptions**

    - Extends `Exception`
    - Possibly recoverable
    - eg. Network error
    - Must appear in `throw` clauses
- **Unchecked Exceptions**
    - Extends `Error` or `RuntimeException`
    - Probably bail out
    - eg. Programming error
    - May appear in `throw` clauses

Basics

- Extend
    - `class IllegalMoveException extends IllegalStateException {}`
- Throw
    - `throw new IllegalMoveException(reason)`

Generics

- Generic Classes
    - `class BinTree<T> { ... }`
- Generic Methods
    - `<T> BinSearch() { ... }`