Object Oriented Programming

Wrapper classes, Java Type System, The Object class

Number Types

- `int`: integers, no fractional part

```
1, -4, 0
```

- `double`: floating-point numbers (double precision)

```
0.5, -3.11111, 4.3E24, 1E-14
```

Number Types

- A numeric computation overflows if the result falls outside the range for the number type

```
int n = 1000000;
System.out.println(n * n); // prints -727379968
```

- Java: 8 primitive types, including four integer types and two floating point types

Primitive Types

Type	Description	Size
`int`	The integer type, with range -2,147,483,648 . . . 2,147,483,647	4 bytes
`byte`	The type describing a single byte, with range -128 . . . 127	1 byte
`short`	The short integer type, with range -32768 . . . 32767	2 bytes
`long`	The long integer type, with range -9,223,372,036,854,775,808 . . . -9,223,372,036,854,775,807	8 bytes

Continued…

Primitive Types

Type	Description	Size
`double`	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits	8 bytes
`float`	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits	4 bytes
`char`	The character type, representing code units in the Unicode encoding scheme	2 bytes
`boolean`	The type with the two truth values `false` and `true`	1 byte

Number Types: Floating-point Types

- Rounding errors occur when an exact conversion between numbers is not

```
double f = 4.35;
System.out.println(100 * f); // prints 434.99999999999994
```

- Java: Illegal to assign a floating-point

```
double balance = 13.75;
int dollars = balance; // Error
```

Number Types: Floating-point Types

- `Casts`: used to convert a value to a different t

```
int dollars = (int) balance; // OK
```

Cast discards fractional part.

- `Math.round` converts a floating-point

```
long rounded = Math.round(balance); // if balance is 13.75, then
                                    // rounded is set to 14
```

Cast

```
(typeName) expression
```

Example:

```
(int) (balance * 100)
```

Purpose:

To convert an expression to a different type

Primitives & Wrappers

- Java has a *wrapper* class for each of the eight primitive data types:

Primitive Type	Wrapper Class	Primitive Type	Wrapper Class
boolean	Boolean	float	Float
byte	Byte	int	Integer
char	Character	long	Long
double	Double	short	Short

Use of the Wrapper Classes

- Java's *primitive* data types (boolean, int, etc.) are not classes.
- Wrapper classes are used in situations where objects are required, such as for elements of a Collection:

```
List<Integer> a = new ArrayList<Integer>();
methodRequiringListOfIntegers(a);
```

Value => Object: Wrapper Object Creation

- ***Wrapper*.valueOf() takes a value (or string) and returns an object of that class:**

```
Integer i1 = Integer.valueOf(42);
Integer i2 = Integer.valueOf("42");

Boolean b1 = Boolean.valueOf(true);
Boolean b2 = Boolean.valueOf("true");

Long n1 = Long.valueOf(42000000L);
Long n1 = Long.valueOf("42000000L");
```

Object => Value

- Each wrapper class Type has a method typeValue to obtain the object's value:

```
Integer i1 = Integer.valueOf(42);
Boolean b1 = Boolean.valueOf("false");
System.out.println(i1.intValue());
System.out.println(b1.booleanValue());

=>
42
false
```

String => value

- The Wrapper class for each primitive *type* has a method parse*Type*() to parse a string representation & return the literal value.

```
Integer.parseInt("42")          => 42
Boolean.parseBoolean("true")    => true
Double.parseDouble("2.71")      => 2.71
//…
```

- Common use: Parsing the arguments to a program:

Parsing argument lists

```
// Parse int and float program args.
public parseArgs(String[] args) {
  for (int i = 0; i < args.length; i++) {

    …println(Integer.parseInt(args[i]));
}
}
```

Each Number Wrapper has a MAX_VALUE constant:

```
byteObj = new Byte(Byte.MAX_VALUE);
shortObj = new Short(Short.MAX_VALUE);
intObj = new Integer(Integer.MAX_VALUE);
longObj = new Long(Long.MAX_VALUE);
floatObj = new Float(Float.MAX_VALUE);
doubleObj = new Double(Double.MAX_VALUE);

printNumValues("MAXIMUM NUMBER VALUES:");
```

MAX values (output from previous slide):

=>

```
Byte:127
Short:32767
Integer:2147483647
Long:9223372036854775807
Float:3.4028235E38
Double:1.7976931348623157E308
```

Many useful utility methods: Integer

```
        int    hashCode()
static int     numberOfLeadingZeros(int i)
static int     numberOfTrailingZeros(int i)
static int     reverse(int i)
static int     reverseBytes(int i)
static int     rotateLeft(int i, int distance)
static int     rotateRight(int i, int distance)
static String  toBinaryString(int i)
static String  toHexString(int i)
static String  toOctalString(int i)
static String  toString(int i, int radix)
```

Double & Float: Utilities for Arithmetic Operations:

- Constants POSITIVE_INFINITY & NEGATIVE_INFINITY
- Constant NaN = Not-a-Number (NaN) value.
- Methods isNaN(), isInfinite()

Class `Object`

- `Object` is the root of the class hierarchy
  - Every *class* has `Object` as a superclass
- All classes inherit the methods of Object
  - But may override them

TABLE 3.2
Methods of Class `java.lang.Object`

Method	Behavior
`Object clone()`	Makes a copy of an object.
`boolean equals(Object obj)`	Compares this object to its argument.
`int hashCode()`	Returns an integer hash code value for this object.
`String toString()`	Returns a string that textually represents the object.

The Method `toString`

- You should always override **`toString`** method if you want to print object state

- If you do *not* override it:
  - **`Object.toString`** will return a **`String`**
  - Just not the **`String`** you want!

    Example: **`ArrayBasedPD@ef08879`**

    ... The name of the class, `@`, instance's hash code

Always override toString()

"When practical, the toString method should return all of the interesting info contained in the object."

Note that toString should never print anything

toString() called automatically

```
System.out.println( "Answer = "  +  42 );
```

toString() method is called automatically

```
System.out.println( d1 );

System.out.println( d1.topFace() );

System.out.println( d1.toString() );
```

unnecessary, adds clutter

Overriding toString()

It is recommended that you specify the format of the return value for classes associated with a “value”, and to document your intentions.

examples:

phone number format (XXX) YYY-ZZZZ

address format Flat No.: XXX Door No.: YYY

Street, Landmark, City, District, State, Country, Pincode