Object and primitive types

No subtyping between primitive and object types

Example:

- a variable of type **`int`** cannot contain an instance of `Integer`
- a variable of type `Integer` cannot contain a value of type **`int`**

Implicit conversion between primitive and object types

- boxing: from primitive to object type
- unboxing: from object to primitive type

Example

```
Integer o = 42; // boxing, OutOfMemoryError may be thrown
int i = o;      // unboxing, NullPointerException may be thrown
```

Object and primitive types

Motivations

- object types allow values to be managed uniformly through references
- boxed primitive values follow the "everything is an object" OO vision
- instance variables of boxed primitive types can be optional with **`null`**

Details on boxing/unboxing

Contexts where code for boxing/unboxing conversions is generated:

- assignment
- argument passing
- casting
- numeric promotion

Remarks

- `OutOfMemoryError` may be thrown during boxing conversion
- `NullPointerException` may be thrown during unboxing conversion

Numeric promotion

In a nutshell

- unboxing and widening implicitly applied for some arithmetic operators
- subtypes of `int` are always promoted

Example

```
assert 5 / 2 == 2;
assert 5 / 2. == 2.5; // int -> double
Integer i = 5; // int -> Integer
assert i * 2 == 10; // Integer -> int
assert i * i == 25; // Integer -> int
assert i / 2. == 2.5; // Integer -> int -> double
```

Boxing, unboxing, and efficiency

Example

```
public static int sum (Integer[] ints) {
    int s = 0; // ok
    for (int n : ints) { s += n; }
    return s;
}
public static Integer sumInteger(Integer[] ints) {
    Integer s = 0; // inefficient
    for (Integer n : ints) { s += n; }
    return s;
}
public static void main(String[] args) {
    assert sum(new Integer[] { 1, 2, 3, 4 }) == 10; // int -> Integer
    sum(new int[]{1,2,3,4}); // compile-time error: int[] ⋠ Integer[]
}
```

Subtyping and array types

Java rules

- T_1 and T_2 reference types: $T_1 \leq T_2 \Rightarrow T_1[\,] \leq T_2[\,] \leq$ `Object`
- T primitive type: $T[\,] \leq$ `Object`
- T primitive type: the only array type compatible with $T[\,]$ is $T[\,]$

Examples

- `String[]` $\leq$ `Object[]` $\leq$ `Object`
- **`int`**`[]` $\leq$ `Object`
- **`int`**`[]` $\not\leq$ `Integer[]`
- **`int`**`[]` $\not\leq$ `Object[]`
- the only array type compatible with **`int`**`[]` is itself

Subtyping and array types

Remark

- array subtyping is not sound in Java
- consequence: array assignment requires a dynamic type check

Example

```
Object[] o;
String[] s = {"a", "b"};
o = s; // ok: String ≤ Object ⇒ String[] ≤ Object[]
o[0] = 42; // throws ArrayStoreException at runtime
assert s[0].length() == 1; // never executed
```

Remarks

`o[0]=42` is type correct for boxing and subtyping

More details on Java class `String`

String literals in a nutshell

- immutable objects of `String`
- delimited by `"`
- contained in a single line

Escape sequences (a selection)

- `\b` (backspace)
- `\t` (horizontal tab)
- `\n` (linefeed)
- `\f` (form feed)
- `\r` (carriage return)
- `\"` (double quote)
- `\'` (single quote)
- `\\` (backslash)

More details on Java class `String`

String comparison

String must be compared with instance methods

- **int** `compareTo(String anotherString)`
- **boolean** `equals(Object anObject)`

Example

```
String s1 = ""; // the empty string
assert s1.length() == 0;
String s2 = "\"\\\n";
assert s2.length() == 3;
assert s2.charAt(0) == '"'; // double quote char
assert s2.charAt(1) == '\\'; // backslash char
assert s2.charAt(2) == '\n'; // linefeed char
s2 = "ab";
assert s2 != s1 + s2; // do not use == and != on immutable objects!
assert s2.compareTo(s1 + s2) == 0;
assert s2.equals(s1 + s2);
```