Subtyping and primitive types

Primitive types in Java

- `boolean`: `false` and `true`
- integral types (two's-complement, all signed except for `char`):
  - `byte` (1 byte)
  - `short` (2 bytes)
  - `char` (2 bytes, unsigned)
  - `int` (4 bytes)
  - `long` (8 bytes)
- floating-point types (IEEE 754)
  - `float` (4 bytes)
  - `double` (8 bytes)

Example of integer literals

- type `int`: `123_000_000` (base 10), `0xfff0ab` (base 16), `0XFFF0AB` (base 16), `07334` (base 8), `0b1100_0000_1100` (base 2)
- type `long`: `123_000_000L`, `0xfff0abL`, `0XFFF0ABL`, `07334L`, `0b1100_0000_1100L`

Subtyping and primitive types

Primitive types in Java

- **`boolean`**: **`false`** and **`true`**
- integral types (two's-complement, all signed except for **`char`**):
  - **`byte`** (1 byte)
  - **`short`** (2 bytes)
  - **`char`** (2 bytes, unsigned)
  - **`int`** (4 bytes)
  - **`long`** (8 bytes)
- floating-point types (IEEE 754)
  - **`float`** (4 bytes)
  - **`double`** (8 bytes)

Example of floating-point literals

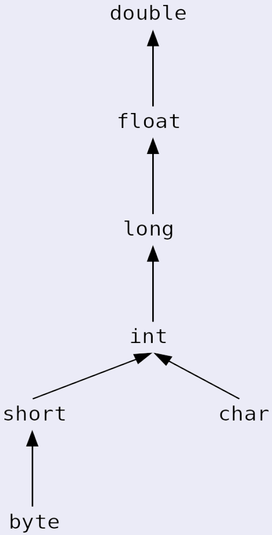
- type **`float`**: `1e1f 2.f .3f 0f 3.14f 6.022137e+23F`
- type **`double`**: `1e1 2. .3 0.0 3.14 1e-9d 1e137D`

Subtyping and primitive types

Subtyping between primitive types

Intuition: rules follow set inclusion

- **`int`** $\leq$ **`long`** $\leq$ **`float`** $\leq$ **`double`**
- **`byte`** $\leq$ **`short`** $\leq$ **`int`**
- **`char`** $\leq$ **`int`**



Conversions on primitive types

Remark

- a variable of reference type T can contain **`null`** or refer to an object of a subtype of T
- a variable of primitive type t can only contain a value of type t

Widening and narrowing primitive conversions

Widening:

- conversion from subtype T_1 to supertype T_2 ($T_1 \leq T_2$)
- allowed to be implicit
- not lossy, except for some cases

Narrowing:

- conversion which is not a widening
- must be explicit, with cast or `Math.round`
- lossy, in general

Conversions on primitive types

Example

```
int i = Integer.MAX_VALUE;
long l = Long.MAX_VALUE;

float f1 = i; // implicit widening primitive conversion
float f2 = l; // implicit widening primitive conversion

assert f1 == i; // implicit widening primitive conversion
assert f2 == l; // implicit widening primitive conversion

assert (int) f1 == i;  // narrowing conversion with cast
assert (long) f2 == l; // narrowing conversion with cast

assert Math.round(f1) == i;          // calls version int Math.round(float a)
assert Math.round((double) f2) == l;  // calls version long Math.round(double a)
```

Remarks for narrowing conversions

- cast more efficient, but less precise
- class method `Math.round` more precise, but less efficient

Wrapper classes of primitive types

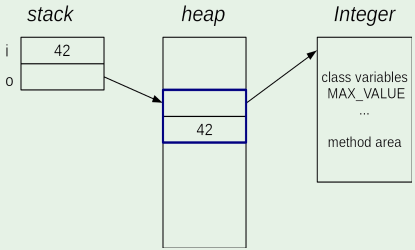
Each primitive type has a corresponding wrapper class

- wrapper classes are predefined classes in `java.lang`

  `Boolean`, `Byte`, `Character`, `Short`, `Integer`, `Long`, `Float`, `Double`
- Remark: objects of wrapper classes are immutable

Demo

```
int i = 42;
Integer o = Integer.valueOf(i); // returns an object representing 42
```



Class Integer

Some public methods

```java
// returns the wrapped integer
public int intValue()

// returns an Integer of value i, caches values at least in the range -128 to 127
public static Integer valueOf(int i)}

//  parses s and converts it as a signed decimal, may throw NumberFormatException
public static int parseInt(String s)

// decodes s into an Integer, radix 10, 2, 8, 16, may throw NumberFormatException
public static Integer decode(String s)
```

Class Integer

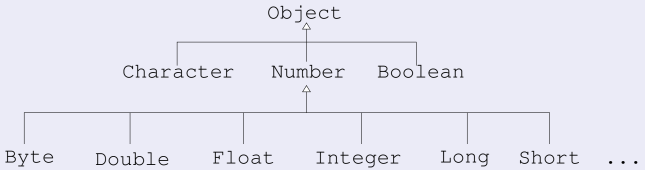
Reminder: avoid `==` or `!=` with immutable objects

```
int i = 4242; // value not in the range -128 to 127

Integer o1 = Integer.valueOf(i);
Integer o2 = Integer.valueOf(i);

// o1 and o2 refer to different objects that represent the same integer
assert o1 != o2;
assert o1.intValue() == o2.intValue();
assert o1.equals(o2);
```

Subtyping relation for wrapper classes



`Byte` `Double` `Float` `Integer` `Long` `Short` subtypes of `Number`

Remark

- no subtyping between wrapper classes of primitive types
- example: `Integer` $\not\leq$ `Float` even though **`int`** $\leq$ **`float`**
- motivation: no value is changed in conversions between reference types

```
int i1 = 4242;
float f = i1; // widening primitive conversion, value is changed

Integer i2 = Integer.valueOf(4242);
Number n = i2; // widening reference conversion, reference is unchanged
```

Primitive and reference types

Recall: no subtyping between primitive and reference types

Example: **`int`** $\not\leq$ `Integer` and `Integer` $\not\leq$ **`int`**

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

Implicit conversion between primitive and reference types

Since Java 5:

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

Demo

```
Integer o = 42; // boxing, same as 'Integer o=Integer.valueOf(42)'
int i = o;      // unboxing, same as 'int i=o.intValue()'
```

Primitive and reference types

Motivations

- reference types allow values to be managed uniformly through references
- boxed primitive values follow the approach "everything is an object"
- fields of boxed primitive types can be optional with **`null`**

Details on boxing/unboxing

Contexts for boxing/unboxing conversions:

- assignment, argument passing, casting, numeric promotion

Remarks

- `OutOfMemoryError` may be thrown during boxing conversion
  `Integer.valueOf(int i)` is a factory method that may create objects
- `NullPointerException` may be thrown during unboxing conversion
  Example: `o.intValue()` with `o` containing **`null`**

Numeric promotion

In a nutshell

- unboxing and widening implicitly applied for arithmetic operators, including comparison and equality
- subtypes of **`int`** are always promoted

Example

```
assert 5 / 2 == 2; // no conversion
assert 5 / 2. == 2.5; // widening int -> double
Integer i = 5; // boxing int -> Integer
assert i  == 5; // unboxing Integer -> int
assert i > 2; // unboxing Integer -> int
assert i * 2 == 10; // unboxing Integer -> int
assert i * i == 25; // unboxing Integer -> int
assert i / 2. == 2.5; // unboxing and widening Integer -> int -> double
```

Boxing, unboxing, and efficiency

Example

```
public static int sum (Integer[] ints) { // efficient version
    int s = 0; // no conversion
    for (int n : ints) { s += n; } // 1 unboxing per iteration
    return s; // no conversion
}
public static Integer sumInt(Integer[] ints) { // inefficient version
    Integer s = 0; // 1 boxing
    for (Integer n : ints) { s += n; } // 2 unboxing+1 boxing per iteration
    return s; // no conversion
}
public static void main(String[] args) {
    assert sum(new Integer[] { 1, 2, 3, 4 }) == 10; // 4 boxing
    assert sumInt(new Integer[] { 1, 2, 3, 4 }) == 10; // 4 boxing+1 unboxing
}
```

Modularization for large-scale programming

Two different levels of modularization

- Modules define and export logically related packages (since Java 9)
- Packages define and export logically related classes

Remark

- for our purposes packages are sufficient for structuring code
- Java projects can be based on the unnamed module

Packages

Logical view of a program structured into packages

Program

package p1

public class C1
public class C2
class D1
class D2

Remarks:
C1, C2 can be used *outside* p1,
with names p1.C1, p1.C2
D1, D2 can *only* be used inside p1,
with names D1, D2

package p2

public class C3
public class C4
class D3
class D4

Remarks:
C3, C4 can be used *outside* p2,
with names p2.C3, p2.C4
D3, D4 can *only* be used inside p2,
with names D3, D4

Packages

Physical view of a program structured into packages

folder src of the program

subfolder p1

file C1.java:

```
package p1;
import p2.C3;
import p2.C4;
public class C1 {...}
class D1 {...}
```

file C2.java:

```
package p1;
import p2.C3;
import p2.C4;
public class C2 {...}
class D2 {...}
```

subfolder p2

file C3.java:

```
package p2;
import p1.C1;
import p1.C2;
public class C3 {...}
class D3 {...}
```

file C4.java:

```
package p2;
import p1.C1;
import p1.C2;
public class C4 {...}
class D4 {...}
```

Packages

Main features

- packages contain classes and subpackages
- public classes can be used outside their package
- non public classes can only be used within their package
- packages are hierarchical namespaces

Packages reflect the structure of the file system

- package = folder containing
  - subpackages (=subfolders)
  - compilation units (=files) declaring classes and interfaces
- a package name corresponds to the path of its folder. Example:
  - `javax.swing.tree` corresponds to `javax/swing/tree`
  - `com.sun.source.util` corresponds to `com/sun/source/util`
  - `javax.swing.tree` and `com.sun.source.util` are different namespaces:

    `javax.swing.tree.TreePath` and
    `com.sun.source.util.TreePath` are different classes

Packages

Simple and fully qualified names of classes and interfaces

Classes and interfaces have both a simple and a fully qualified name
Example:

- `TreePath` is the simple name, usable inside `javax.swing.tree` or `com.sun.source.util`
- `javax.swing.tree.TreePath` and `com.sun.source.util.TreePath` are the fully qualified names, useful outside their packages

Compilation unit

Example

```java
// file ColoredLine.java must be placed in directory shapes
package shapes;         // optional package declaration

// use 'Color' as an abbreviation for 'java.awt.Color'
import java.awt.Color; // optional import declarations

// top level class declarations start
// 'Point' not visible outside 'shapes'
class Point {
    ...
}

// 'ColoredLine' visible outside 'shapes'
public class ColoredLine {
    private Point a;
    private Point b;
    private Color color = Color.BLACK;
    ...
    public Color getColor() { return this.color; }
    public void setColor(Color color) { this.color = color; }
}
```

Compilation unit

A compilation unit consists of three parts

1. package declaration:
   - specify the package which all classes in the unit belongs to
   - if not specified, the classes of the unit belong to the unnamed package
2. import declarations, to access classes of other packages with simple names
3. top level class declarations

Remarks

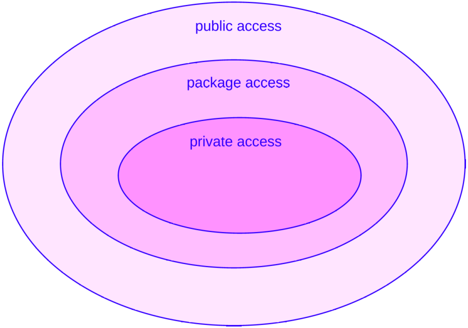
- a compilation unit can contain more classes
- only one class per compilation unit can be public
- the name of the file must be the same as its public class (if any)
- classes can be nested in other classes and methods
- for simplicity we will not consider nested classes

Package access

A new access level for declarations in classes

- private access: declaration only accessible in the class
- package access: declaration only accessible in the package
- public access: declaration accessible everywhere the class is accessible

Inclusion of the three access levels:



Package access

How package access is declared?

- there is no keyword for package access
- package is the default access

Example

```
private void privateMeth(){...} // private method
void packageMeth(){...}          // package method
public void publicMeth(){...}    // public method
```

Remark: package access allowed for object/class fields and methods

Package access

Correct access example

```
package p; // file C.java in folder p
public class C {
    void m() { ... } // package method
}

package p; // file Test.java in folder p
public class Test {
    public static void main(String[] args) {
        C c = new C();
        c.m(); // correct!
    }
}
```

Package access

Illegal access example

```java
package p; // file C.java in package p
public class C {
    void m() { ... } // package method
}

package q; // file Test.java in package q
import p.C;

public class Test {
    public static void main(String[] args) {
        C c = new C();
        c.m(); // compilation error!
    }
}
```

Package access

No visibility rules between packages and their subpackages

Example:

- `javax.swing.tree` is a subpackage of `javax.swing`
- components of `javax.swing.tree` with package access are not visible in `javax.swing`
- components of `javax.swing` with package access are not visible in `javax.swing.tree`

API modules and packages

Documentation

- documentation on the Java API available on the official web site
- documentation also accessible through the IDEs (Eclipse, IDEA)
- we will mainly use packages of the `java.base` module

Remarks

- API = Application Programming Interface
- IDE = Integrated Development Environment

Imports

Useful feature to access a class of another package with its simple name

Remarks

1. all public classes in package `java.lang` of module `java.base` can be automatically accessed with their simple names
2. useless imports are ignored. Example: importing a class of the same package

Single imports and on demand imports

- `import java.util.Scanner;`
  the single class `Scanner` is imported
- `import java.util.*;`
  all public classes and interfaces of `java.util` are imported if needed
- single imports take precedence in case of conflicts
- single imports must avoid name conflicts

Static imports

Single static imports and on demand static imports

- static imports are used for abbreviating names of class fields and methods
  Example:

```
import static java.lang.System.out;
```

  - the single class field `out` is imported from `System`
  - the abbreviated name `out` can be used instead of `System.out`

```
import static java.lang.System.*;
```

  - all accessible class fields and methods of `System` are imported if needed