Types

Slide Series 2

Content

Compile time and Runtime
Types, Type Systems and Type checking
Introducing types
Type compatibility, Conversion rules
Polymorphism
Override and overload
Variance
The Array loop hole
Runtime type information

Compile Time vs Runtime

Software development, in this course, have 3 major phases

1. Coding (incl. informal reasoning).
2. Compiling

   Operations performed during translation of the source code to Java byte code, **compile time**

   After compiling we have classes (.class)
3. Running (incl. testing). Operations during execution of the bytecode is **runtime**

   During execution we have objects (also objects representing the classes)

Data Type and Type System

Data type (type) = *"a classification ... that determines the possible values for that type; the operations that can be done on values of that type ... '* //Wikipedia

Also possible to think : A set with operations

Type system = *"a tractable syntactic framework for classifying phrases according to the kinds of values they compute [the types of the values]'* //Benjamin Pierce (MIT)

Why Types and Type Systems?

"The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program"

"In general, accurate type information at compile time leads to the application of the appropriate operations at run-time without the need of expensive tests."

//Luca Cardelli (famous computer scientist)

Type Systems in Imperative Programming

Remember: Have a lot of variables to handle

Typesystem will prevent us from confusing different kinds of values

- Can only put correct kinds of values in the variables
- Can only apply valid operations for objects
- ... some problems removed!

The Java Type System

"The Java programming language is a statically typed language, which means that every variable and every expression has a type that is known at compile time.
The Java programming language is also a strongly typed language, because types limit the values that a variable (§4.12) can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of the operations. Strong static typing helps detect errors at compile time." // JLS 4

Not 100% strong, we'll later look at a loophole in the system ...

Q: Why is that? ... A: (often) Because of the type system!

Method Expressions and Void

Every expression has a type that is known at compile time

Method call possible an expression or not
- If used as an expression, method has a declared return type, the type for the expression
- If not, has no result (acting as a statement). This is indicated with keyword **void** in method declaration (void is not a type)

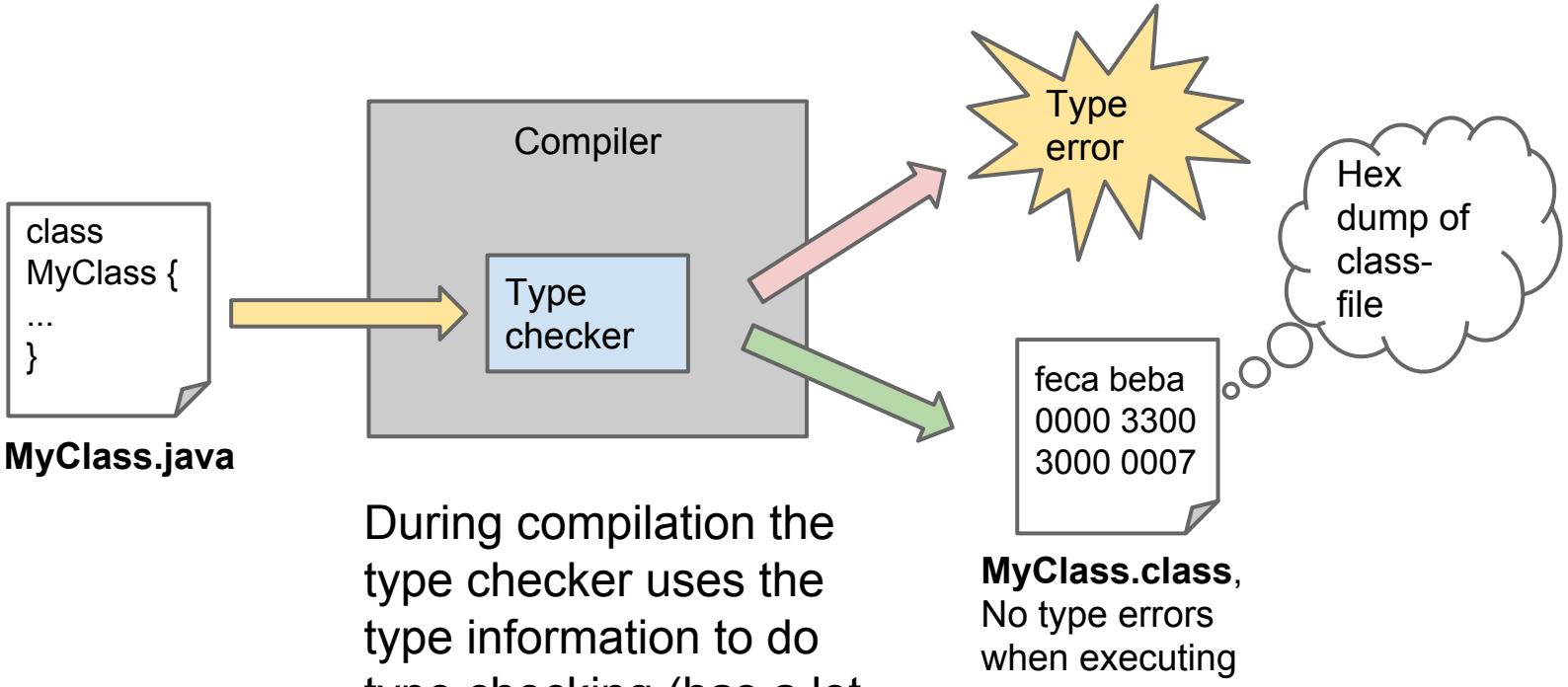
"...*uses the keyword void to indicate that the method does not return a value*" // JLS 8.4.5

Type Checking

"Static type-checking is the process of verifying the type safety of a program based on analysis of a program's text [the source code]. If a program passes a static type-checker, then the program is guaranteed to satisfy some set of type-safety properties for all possible inputs." // Wikipedia

Checking is done by the **type checker** (software, part of compiler)

Type Checking, cont.



During compilation the type checker uses the type information to do type checking (has a lot of built-in knowledge, lot of rules)

Imagine you're the Type Checker!

The source code is all you have … how to check types?

```
int i     // Which type for i?.. Obvious!
1         // Which type?
1.0
'1'
"111"
4.0 + Math.sin(x)
System.out.println(...)
```

Type Inference

In Java we normally have to put explicitly type information in the code

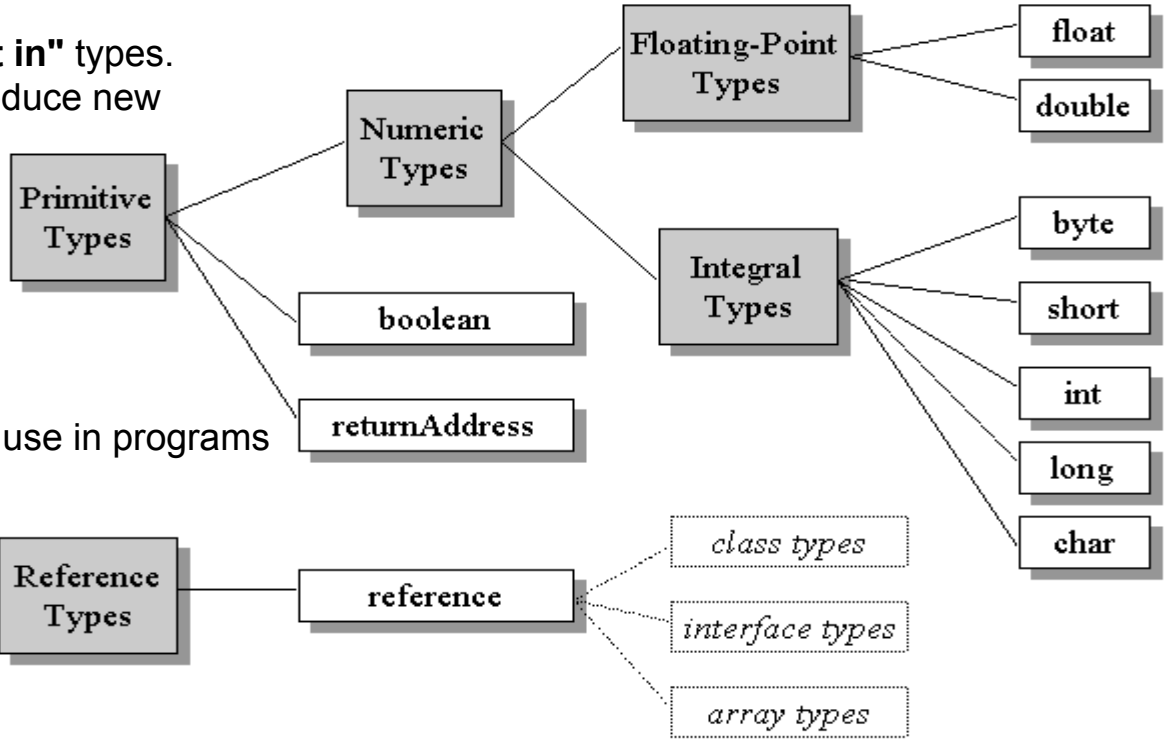
```
//Explicit type information
int i = 0;
```

Haskell is also statically and strongly typed but (mostly) no need to specify types. Why?

- Answer: Haskell uses type inference. Advanced technique to automatically deduce types for expressions
- In between Java also can infer, more later...

Java Types

Aka **"built in"** types.
Can't introduce new



Aka **"user defined"** types.
Can introduce new

Types in java.lang

java.lang.* provides reference types that are fundamental to the design of the Java programming language

Examples: **String, Integer, Boolean, System**, ...

Introducing New Types

Can't introduce new primitive types

New reference types introduced by

- Class
  - **enum**
- Interface
  - Annotation like @Override
- Array (of some existing type)

Introduce Type with Class*

Using a class we introduce a class type and an implementation

- Somewhat problematic, a type is not specified by it's implementation (it's specified by it's elements and the operation on the elements)

```
// Class declaration
public class MyClass {
    public  void doIt()
{
        //...
    }
    //...
}
```

```
// Have type can declare var.
MyClass m;

m = new MyClass();
m.doIt(); // OK
```

Introduce Type with Enum*

An enum introduces an enum type

- An enum type has no instances other than those defined by its enum constants. It is a compile-time error to attempt to explicitly instantiate an enum type
- Direct superclass Enum<E>
  - Can't extend anything, others can't extend enum, more later...
- Can't clone() and more, upcoming...
- Uniqueness for instances guaranteed by system
- Can have attributes, methods, private/package constructors, implement interface

Usage Enum

Used when a few constant values needed (weekdays, colors, ...)

```
//Colors as int's is bad!!!
public static final int RED = 0;
public static final int BLUE = 1;
...
//Later
int color = BLUE; // Ok

// Later
// Compiler accepts, it’s an int!
color = -1;  // Runtime error
```

```
//Colors as type safe enums, good!!!
enum Color{
// Enum constants, upper case
    RED, BLUE, ...
}

//Later
Color color = Color.BLUE; // Ok

// Later
color = “BLUE”; // Compile error!
```

Usage Enum, cont

Use enum to define possible choices

```
public static class Order{
    private OrderStatus status;
    public static enum OrderStatus {  // Inner enum
        ACCEPTED, REJECTED, PAID, SHIPPED;
    }
}

// Simple to use and safe...
Order o = new Order();
o.setStatus(Order.OrderStatus.ACCEPTED); //Defined in
class
```

Introduce Type with Interface*

New interface type but no implementation

- Separation of type and implementation
- Type as a contract (aka as a specification)
- Usage: More to come...

```
// Interface declaration
public interface IMyIface
{
    // Implicit abstract
    public void doIt();
}
```

```
// Have type can declare
IMyInterface m;

// Can't instantiate, error
m = new IMyInterface();
```

My naming convention: Leading uppercase "I" in name

Implementing the Interface

An interface is a specification. The implementation of the specification is done by some class that **implements** the interface

```
// Class implements interface, must have method
doIt()
// Compiler checks (see previous slide)
public MyClass implements IMyIface {
    @Override
    public void doIt(){
        ...// code to run
    }
}
```

Marker Interface

Some Java interfaces are **marker interfaces**

Very different use of interface

- New type with no operations (interface empty)
- Examples: Serializable, Cloneable, ...
- Used for technical reasons, extra information to compiler or JVM
- Strange, bad, don't use ... (nowadays annotations are used)

Introduce Type with Annotation

Similar to interfaces, in this course we don't use any

- Annotations may be present only in source code, or they may be present in the binary form of a class or interface
- Direct superinterface of an annotation type is always java.lang.annotation.Annotation
- Some predefined: @Override, @SuppressWarnings, ...

```
// Annotation type
@interface Quality {
    enum Level { BAD, INDIFFERENT, GOOD }
    Level value();
}
```

Using Annotations

"The purpose of an annotation is simply to associate information with the annotated program element."

// JLS 9

```
// (Non predefined) Annotation usage @Quality(Quality.
Level.GOOD)
public class Karma {
...
}
```

Introduce Type with Array

"An array type is written as the name of an element type followed by some number of empty pairs of square brackets []. The number of bracket pairs indicates the depth of array nesting." // JLS 10

- Array type based on existing type introduced
- The direct superclass of an array type is Object (no Array type).
- All array types have public final field length (which is not in super type!), see JLS 10.7

```
// Array declaration, instantiation
public class MyClass {
    String[] strs;  // Type and variable, no array yet!
    strs = new String[5]; // Create array with five nulls!
}
```

Subtype

*"In programming language theory, **subtyping** (also **subtype polymorphism** or **inclusion polymorphism**) is a form of type polymorphism in which a **subtype** is a datatype that is related to another datatype (the **supertype**) by some notion of substitutability, meaning that program elements, typically subroutines or functions, written to operate on elements of the supertype can also operate on elements of the subtype. If S is a subtype of T, the subtyping relation is often written **S <: T**, to mean that any term of type S can be safely used in a context where a term of type T is expected. The precise semantics of subtyping crucially depends on the particulars of what "safely used in a context where" means in a given programming language. The type system of a programming language essentially defines its own subtyping relation, which may well be trivial."* // Wikipedia

Subtype in Java

Subtype defined from supertype

"The subtypes of a type T are all types U such that T is a supertype of U, and the null type. // JLS 4.10

Supertype in Java

In Java a type S is a direct supertype of T if

- S is the direct superclass of S (T **extends** S)
- S is a direct superinterface of S (T **implements** S)

The supertypes of a type are obtained by reflexive and transitive closure over the direct supertype relation

- All classes and interfaces: A:>A
- If A:>B and B:>C then A:>C

Supertype accessible from subtype using keyword **super**

Multiple Super and Subtypes*

A type may have multiple supertypes by extend (zero or one occurrence) and implements (zero to n occurrences)

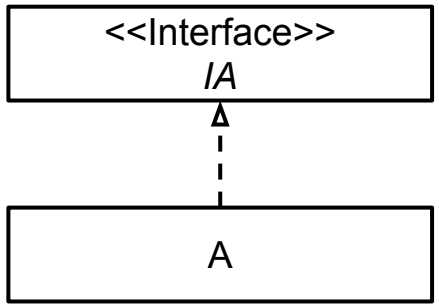
- More to come...

A type may have multiple subtypes by classes or interfaces extending or implementing

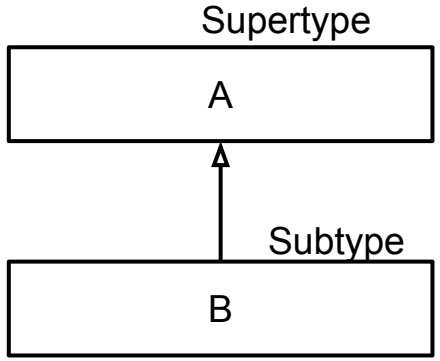
- Many interface/classes can implements/extends another interface
- Many classes can extends some class

UML for Super/Sub Types

<<Interface>> is optional (a stereotype),
name should be in italics



Arrow head
hollow.
Dashed line
for implements




```
class A implements IA {
...
}
```

```
class B extends A{
...
}
```

The Null Type

The literal **null** is a value representing "not a valid object" (it's not an object)

- null is the only value in the **Null type**
- null == null always true

"The direct supertypes of the null type are all reference types other than the null type itself" //* JLS 4.10.2

*) Hmm, seems like null extends many classes. Not possible for other types in Java!

Object*

java.lang.Object is the (implicit, not visible in code) supertype for all class enum or array types and null

- All are possible to convert to the Object type
- Object has no supertype

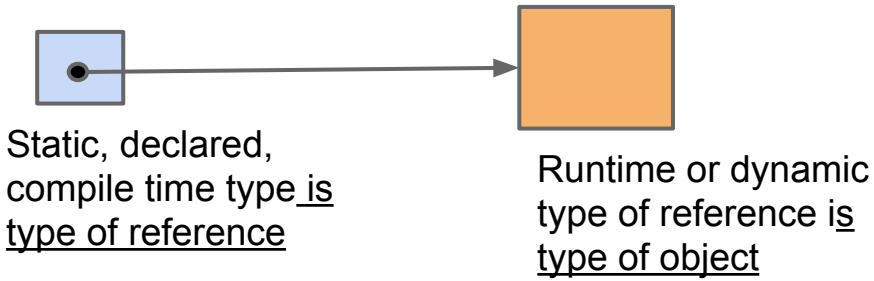
Object is not super type to any interface type but ...

- ... any interface type is assignable to type Object
- ... all Objects methods present in any interface (as abstract methods)...

Compile Time vs Run time Type

Sadly we have many confusing notions of type …

- Type of reference (the variable) often referred to as **static type, declared type** or **compile time type**
- **Runtime type** or **dynamic type** of reference refers to the type of the object the reference is referencing



Runtime/dynamic type of reference = object type

Type Equivalence

Equivalence by name

- Simple : Compare two names (syntactic)

Equivalence through definition “structural type equivalence"

- Harder : Structure must be compared (compared in what way...?)
- Recursively defined type (class Node)!

Type Compatibility

Type equivalence is often a too hard restriction

Type compatibility: Rules to decide when it's safe (no runtime errors) to use one type instead of another

Compatibility is specific to each programming language

Conversion

In Java type compatibility defined by conversion rules

" *… rather than requiring the programmer to indicate a type conversion explicitly, the Java programming language performs an implicit conversion from the type of the expression to a type acceptable for its surrounding context.*" // JLS 5

"*A specific conversion from type S to type T allows an expression of typeS to be treated at compile time as if it had type T instead.*" // JLS 5

Conversion Rules

Eleven broad conversion categories (Assignment conversion, Boxing conversions, ...)

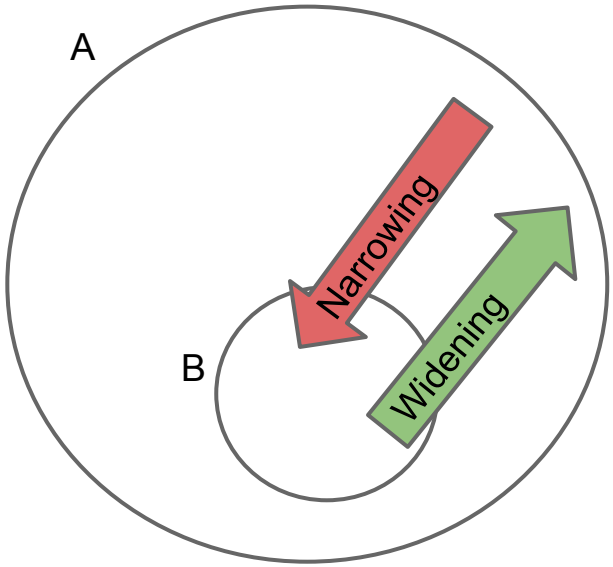
- Specific conversions divided into 13 categories

If a conversion is applicable depends on a "context"

- There are 5 different contexts (one is when using +)

Widening and Narrowing

Rules refer to **widening** and **narrowing**



Types at Set's
- Narrowing: From superset to subset (general to special)
- Widening: From subset to superset (special to general)

Narrowing not allowed by compiler. Loses information or type safety (have to cast, more to come…)
Widening normally non-problematic (some issues)

Some Java Conversion*

- Widening primitive conversion (int to long, float, double)
- Boxing/Unboxing (int to Integer)
- String (using +, any type can be converted to String)
- Numeric promotions, applied to operands of arithmetic operators (widening, unboxing)

See JLS 5.

Java Reference Widening Conversion*

"*A widening reference conversion exists from any reference type S to any reference type T, provided S is a subtype (§4.10) of T.*" // JLS 5.1.5

From subtype to supertype I.e. subtype compatible with supertype

```
// If class A implements interface IA (A subtype to IA)
IA a = new A();  // A compatible with IA.
                 // Automatic conversion
                 // (nothing happens to the object)
```

Java Reference Narrowing Conversion

- From supertype to subtype
- Not type safe, operations in subtype possible not in supertype
- Common case : From Object to some subtype
- Compiler rejects (must use casting, ...)

```
// If class B extends A (compiler will reject the code)
B b = new A();  // B possible can handle more than A!
b.doIt();       // Not sure the method implemented in A?!?!
```

Casting Conversions*

"Casting conversion [casting] is applied to the operand of a cast operator (§15.16): the type of the operand expression must be converted to the type explicitly named by the cast operator."* // JLS 5.5

```
// Casting using cast operator and operand expression o
Object o = …
Integer i = (Integer) o;     // Casting expression
```

Used for narrowing conversions

*) It's just the type nothing happens to the object

Casting Expressions

"A cast expression converts, at run time, a value of one numeric type to a similar value of another numeric type; or confirms, at compile time, that the type of an expression is boolean; or checks, at run time, that a reference value refers to an object whose class is compatible with a specified reference type." // JLS 15.16

Reference Type Casting Conversion*

"*Given a compile-time reference type S (source) and a compile-time reference type T (target), a casting conversion exists from S to T if no compile-time errors occur due to the following rules.*" // JLS 5.5.1

A few casting rules, many more see JLS ...

If S class type and
- T class type, must have S :> T or T :> S
- T interface type, then casting always legal (a bit simplified)
- T array type, S must be type Object

If S interface type and
- T class type then casting always legal (a bit simplified)

Can't cast to void (void not a type)

Casting is Dangerous

Even if there exist a casting conversion during compilation no guarantees it will work runtime

- The responsibility is on the programmer…
- **Always try to avoid casting**

```
// Is o really an Integer?
// If not, runtime exception (ClassCastException)
Integer i = (Integer) o;
```

Polymorphic Objects

Subtyping implies that an object may have many types
the objects are polymorphic (= many forms)

```
// Polymorphic object
Number c = o ;        // If o a Number ...!
Integer i = o;        // … then possible also an Integer
```

Aka **inclusion polymorphism** or simply polymorphism

Polymorphic Types

Polymorphic Types

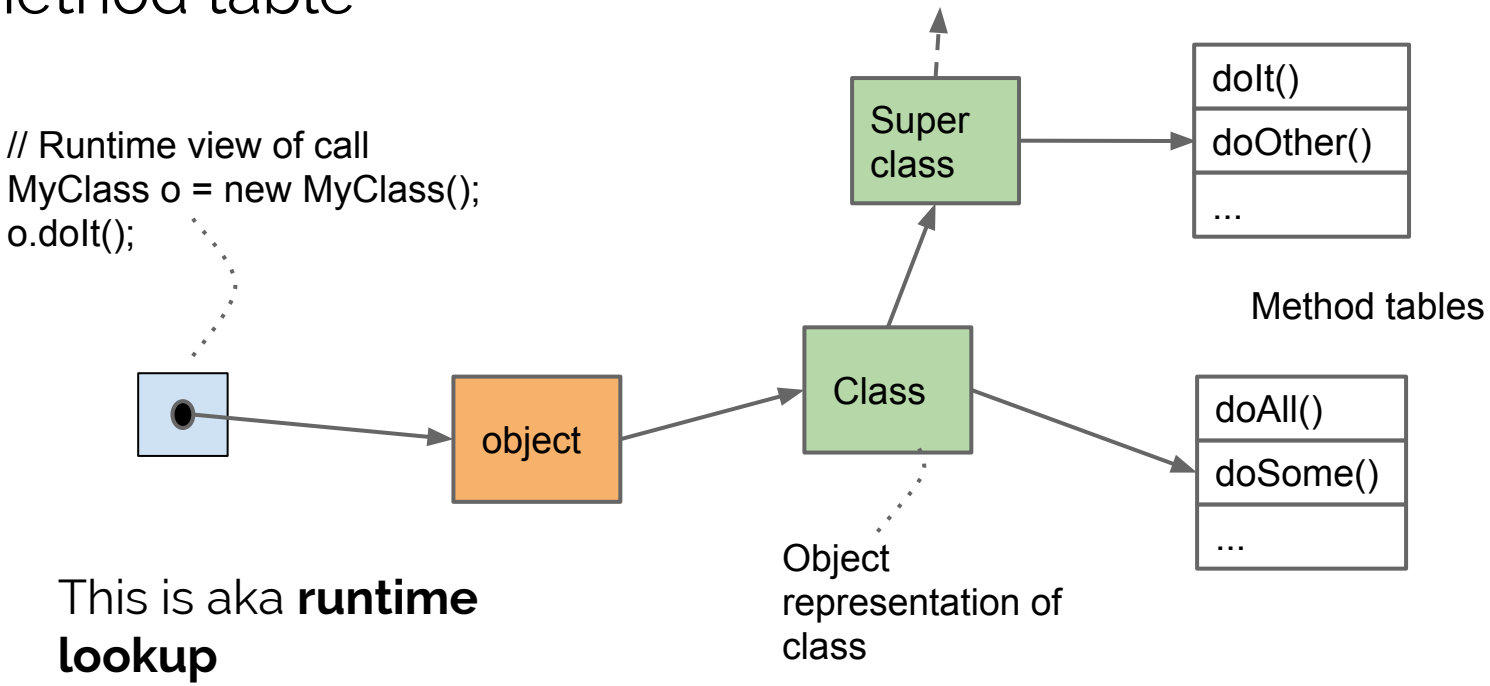
- The Java collection class ArrayList is a **polymorphic type**
- All methods can be applied to values of more than one type (ArrayList<Integer>, ArrayList<String>,...)

Aka parametric polymorphism or **generic types**

More to come later...

Method Call and Dispatching

Method call and dispatching (= search for method to run) in Java **always** starting from the object, follow its class and superclass "references" until a method is found in a method table



This is aka **runtime lookup**

Polymorphic Behaviour

Polymorphic behaviour refers to the ability of objects to behave in possible different way depending on involved types (this is also called polymorphism)

Polymorphic behaviour in course

- **Method Overriding,** method to run decided by runtime type
- **Method Overloading,** method to run decided from parameters (method possible in some superclass)

Overriding*

"*An instance method m1, declared in class C,* ***overrides*** *another instance method m2, declared in class A iff all of the following are true:*
- C is a subclass of A.
- The signature of m1 is a sub signature (§8.4.2) of the signature of m2 [incl. they have the same signature] […]

- […] m2 is public, protected, or declared with default access in the same package as C" // JLS 8.4.8.1

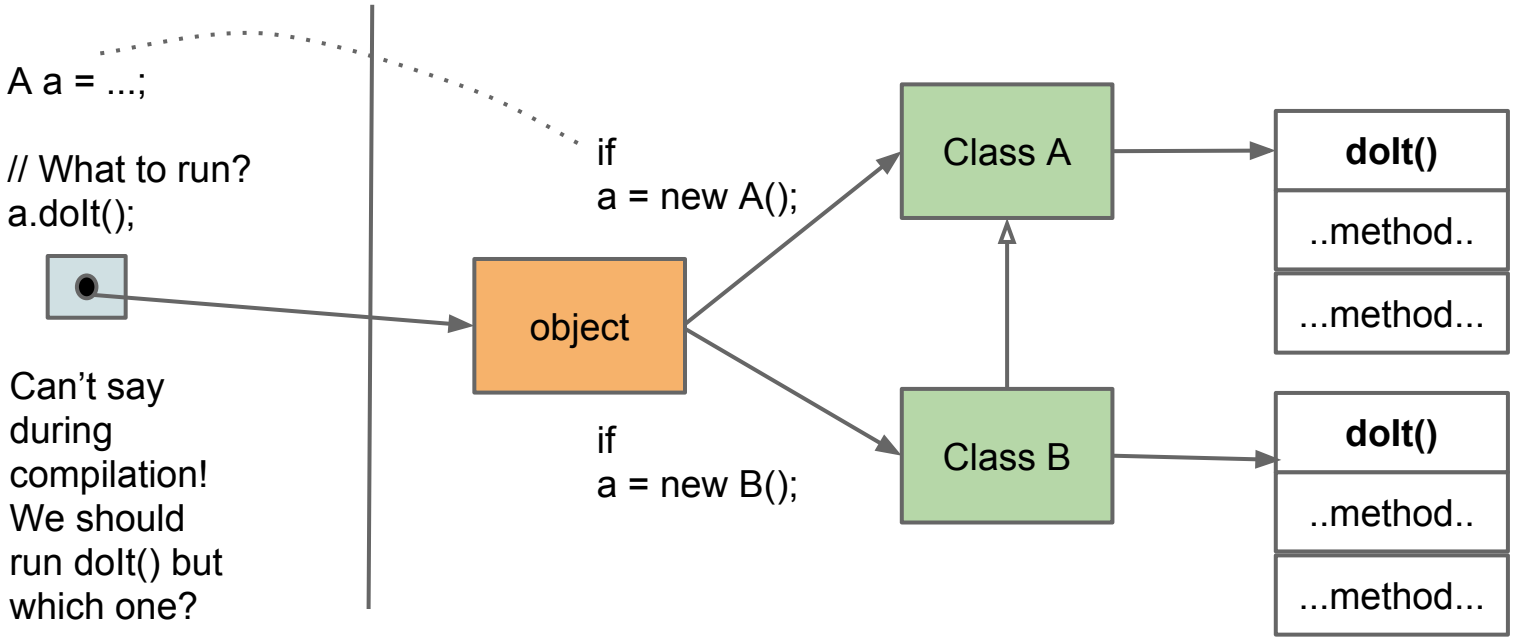
"*If a method declaration d1 with return type R1 overrides […] the declaration of another method d2 with return type R2, then d1 must be return-type-substitutable (§8.4.5) for d2, or a compile-time error occurs.*"

// JLS 8.4.8.3

Method signature = name and parameter list, number of params and type for each (return type not part of signature)

Overriding and Method Dispatch

Dispatch when overriding is present (A:>B)



We get behavior depending on type of object (not type of reference)! A key feature in OO programming!

Inheritance

"*A class C inherits from its direct superclass and direct superinterfaces all abstract and non-abstract methods of the superclass and superinterfaces that are public, protected, [...]* " //JLS 8.4.8

A method need not be declared in the actual class, it's possible inherited from superclass (invisible in sub class code)

More to come later ...

Overloading

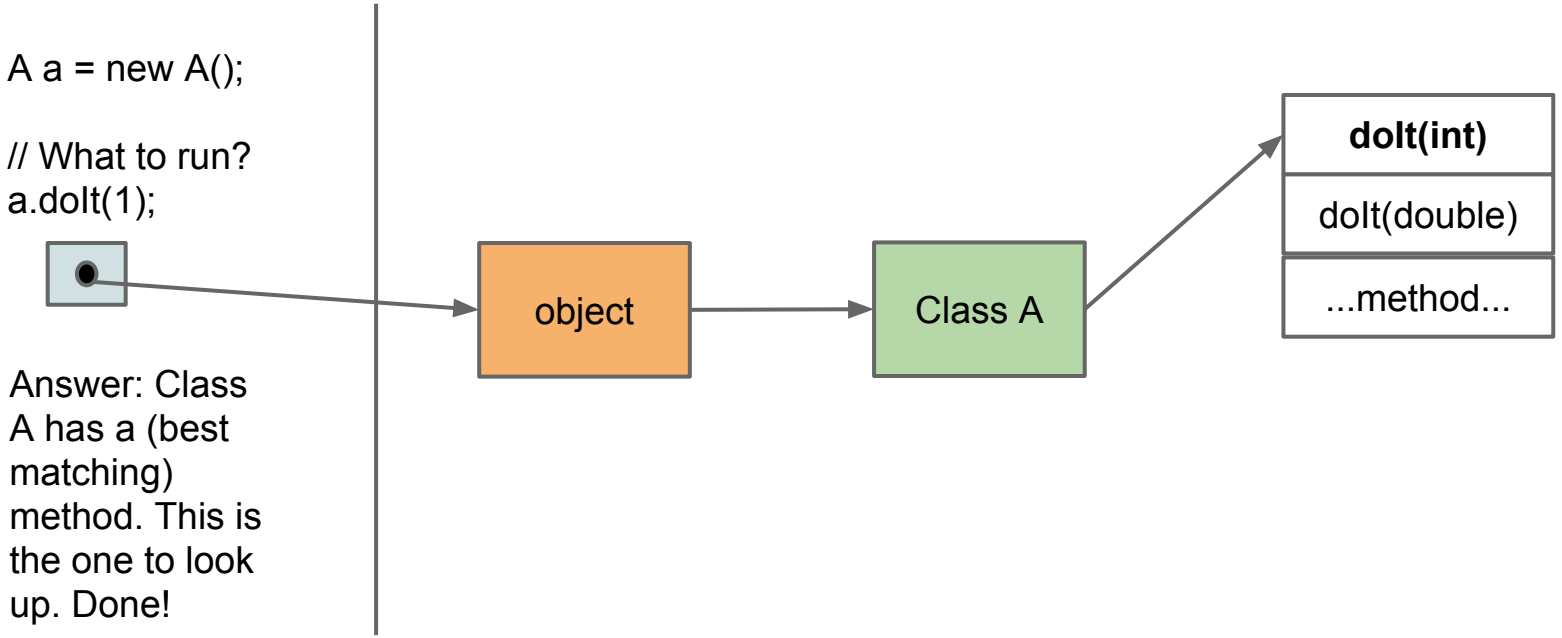
"*If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but signatures that are not override-equivalent, then the method name is said to be* ***overloaded***.

[...]

When a method is invoked (§15.12), the number of actual arguments (and any explicit type arguments) and the compile-time types of the arguments are used, at compile time, to determine the signature of the method that will be invoked (§15.12.2). If the method that is to be invoked is an instance method, the actual method to be invoked will be determined at run time, using dynamic method lookup (§15.12.4). // JLS 8.4.9

Overloading and Dispatch

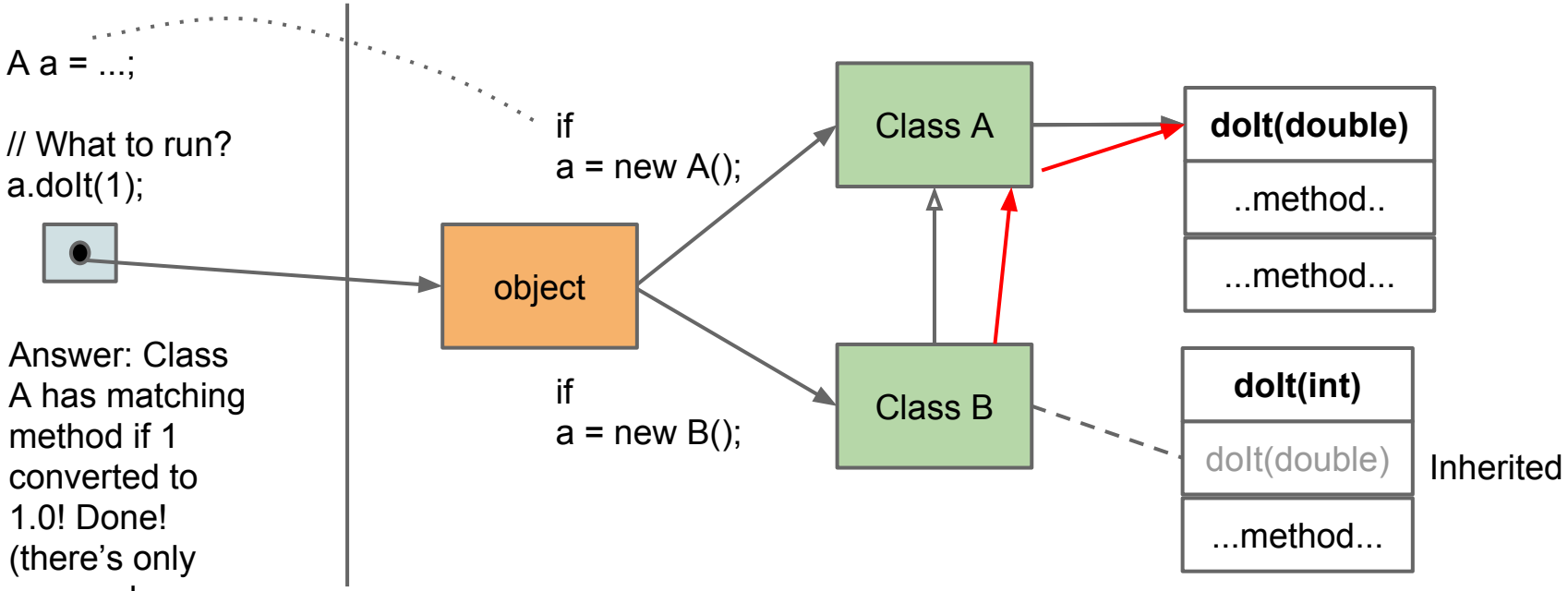
Dispatch when method overloaded in same class*



*) Overloaded methods should be in same class, but can't force programmer

Overloading and Dispatch, cont

Dispatch when method overloaded with and super/sub classes (A :> B)



Method signature fixed during compilation
No other signature considered during dispatch!

Usage Overloading

- Logically "same" methods with different type, example add(int, int), add(float, float). Convenient for programmer (doesn't need to find different names for closely related operations i.e. same name for different methods)
- Methods should do “the same”
- Constructor overloading common. Constructor with most parameters is the base constructor. Other have default values for some parameter (a service to the user)

Compile vs Run-time Polymorphism

Overriding is runtime polymorphism (aka late binding)

The type of the object for which the method is called, during execution, governs the behavior

Overloading is compile time polymorphism (aka static binding). The name of the method and the number and static types of the parameters will govern, at compile time, which code to run

Variance*

Possible to relax type checking for overridden methods

Covariance

- Subclass method may return subtype of superclass method returntype (Co: subclass -> sub return type)
- Possible in Java

Contra variance

- Subclass method can take parameters of supertype of superclass method parameters (Contra: subclass -> super param type)
- Not in Java, will end up as overload

Invariance = neither of above applies

The Array Loophole*

Java has decided that: if A:>B then A[] :> B[] (if B subtype to A than array of B subtype array of A)

- A, B reference types
- Will break typesystem!!!

Java fix

- Any insertion into an array is typechecked runtime
- Motivation (?): A cost but it is better to check stores at runtime than to copy the entire array when a conversion from A[] to B[] is desired

Runtime Type Information*

Every object knows it's class. Possible to retrieve at runtime as an object representing the class (runtime type information, RTTI)

- Using operator **instanceof**
- Using method getClass() or **.class** (an object literal i.e. will get an object without using new)

Instanceof*

```
// Using instanceof
if( o instanceOf A ){    // Check compatibility
    // True -> Conversion safe
    A a = (A) o;  // Safe
}
```

"At run-time, the result of the instanceof operator is true if the value of the RelationalExpression [object] is not null and the reference could be cast (§15.16) to the ReferenceType without raising a ClassCastException. Otherwise the result is false." // JLS 15.20.2

.getClass() and .class*

```
// Get class object for class A
Class<A> c = A.class;
```

"The type of C.class, where C is the name of a class, interface, or array type (§4.3), is Class<C>.The type of p.class, where p is the name of a primitive type (§4.2), is Class<B>, where B is the type of an expression of type p after boxing conversion (§5.1.7). "

// JLS 15.8.2

```
// Get class object for object o (of type A)
// This will give the runtime type for o!
Class<? extends A> c = o.getClass(); // ? = the unknown type
```

"The type of a method invocation expression of getClass is Class<? extends |T| where T is the class or interface searched (§15.12.1) for getClass." // JLS 4.3.2

Terminology

Sadly the nomenclature doesn't mean the same in Java and Haskell

- Overloading is same
- Parametric type same
- No subtype polymorphism in Haskell
- Different: (Type) class, override, ...

Type Theory

Behind much of this is Type theory

"type theory can refer to the design, analysis and study of type systems, although some computer scientists limit the term's meaning to the study of abstract formalisms such as typed λ-calculi".

Very strong area of research at D&IT

Summary

- The type system is a fundamental part of any (typed) language
- The type system stops us from confusing different kind of dat
- Type conversion rules and casting
- We have static (compile time) and dynamic (runtime) types
- Variables and values (expressions) can have more types
- Polymorphic behavior (behaviour depends on type)
- We have overloading, overriding and type conversions