Java Types

Slide Series 2

Content

Compile time and Runtime
Types and Type Systems
Introducing types
Type compatibility
Polymorphism
Override and overload
Runtime type information
null

Compile Time vs Runtime

Software development, in this course, have 3 major phases

1. Coding (incl. reasoning)
2. Compiling

   Operations performed during translation of the source code to machine code (Java byte code), **"compile time"**
3. Running (incl. testing)

   Operations during execution is **"runtime"**

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 could 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)

Example: Classification by Syntax

```
if data type int defined as
{..,-2,1,0,1,2,...}, operations: +, -, *, /

int i;     // Variable of type int (syntax tells)
5765       // Classify as int because it's
           // composed of digits only (syntax tells)
21         // Also an int
i = 5765 + 21; // + defined for type int (2 int operands
               // result of type int). Type system says ok!
               // = defined for equal types (or
              compatible).
               // Type system says ok!
```

Put simply: By "reading" the program the type system can classify all kinds of values and check operations

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)

State is Problematic

Remember: Have a lot of "boxes" to handle

Types/typesystem will prevent us from confusing different kinds of data

- Can't put wrong kind in the box
- Can't apply undefined/wrong operations
- ... **one problem less**!

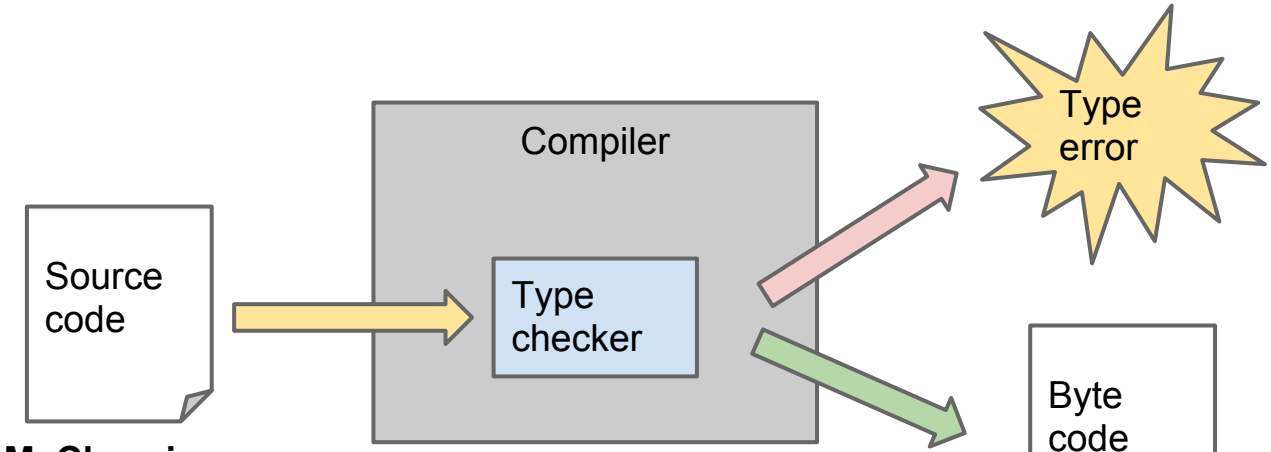
The Java Type System

"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!

Java: Compile and Type Check



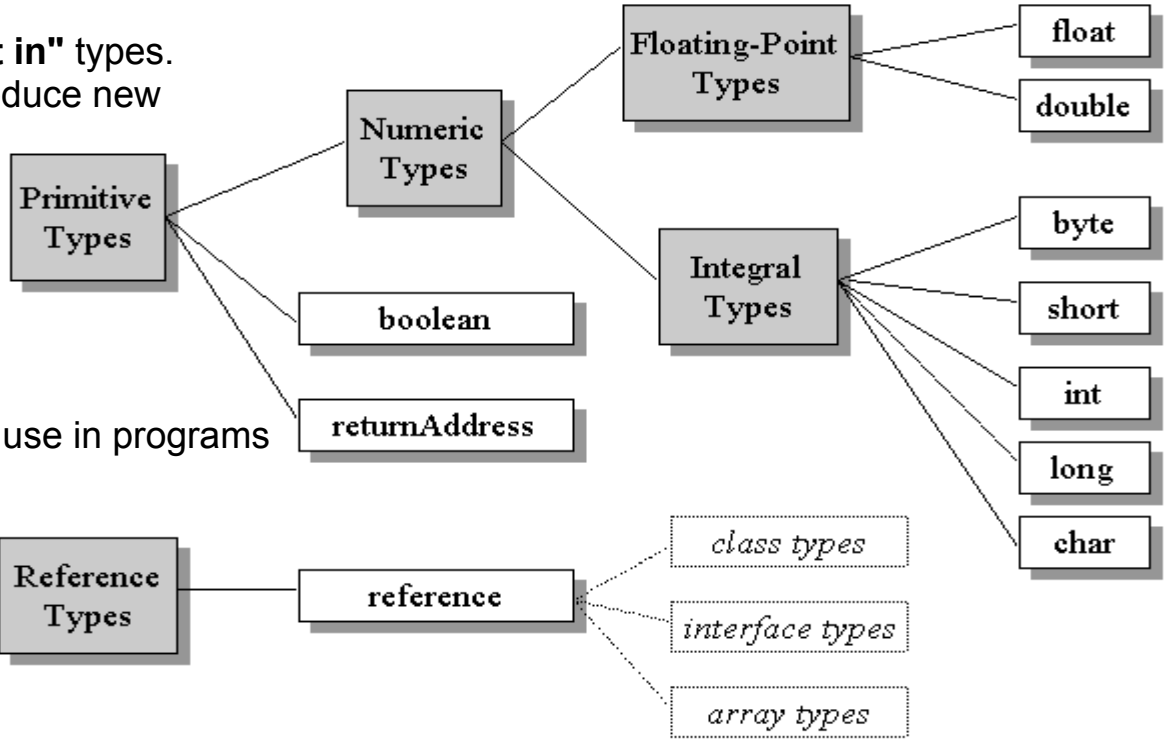
During compilation the type checker uses the type information to do type checking (also has built-in knowledge, a lot of rules)

MyClass.class, type information used during compilation deleted

Eclipse compiles continuously in background, that's how the notifications (icons) are produced

Java Types

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



Can't use in programs

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

Introducing New Types

New reference types introduced by

- Class
  - **enum** special kind of class
- Interface
  - Annotation type is a special kind of interface, used for annotations: @Override
- Array (of some existing type)

Introduce Type with Class*

Using a class we introduce a type and an implementation

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

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

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

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

Enum

Special kind of class (so introduce a 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
- Uniqueness for instances guaranteed by system
- Direct superclass Enum<E>
  - So can't extend ..upcoming (others can't extend enum).
- Can implement interface
- Can't clone ... and more ... (more later)

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
// 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 = -1; // 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 and safe...
Order o = new Order();
o.setStatus(Order.OrderStatus.ACCEPTED); //Defined in
class
```

Introduce Type with Interface*

New type but no implementation

- Separation of type and implementation
- Type as a **contract** (to be full filled by something)
- My naming convention: Leading uppercase "I" in name
- Usage: Much 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();

// There are other ways...
```

Annotation Type

- 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();
}
```

Annotations

"The purpose of an annotation is simply to associate information with the annotated program element." // JLS 9

We only use predefined annotation types in course (@Override)

```
// (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

- The direct superclass of an array type is Object (no Array type).
- Possible with arrays (collections) of interfaces, very useful
- All array types have public final field length

```
// Array declaration, instantiation
public class MyClass {
    String[] strs;  // Type and variable, depth 1, no
array!
    strs = new String[5]; // Create array with five nulls!

}
```

Type Equivalence

Equivalence by name

- Simple : Compare two names

Equivalence through definition “structural type equivalence"

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

Type Equivalence In Java

Two reference types are the same compile-time type if they have **the same binary name** (§13.1) and their type arguments, if any, are the same, applying this definition recursively. // JLS 4.3.4

Two reference types are the same run-time type if (**NOTE**: Possible different type depending on compile or run-time!)

- They are both class or both interface types, are defined by the same class loader, and have the same binary name (§13.1), in which case they are sometimes said to be the same run-time class or the same run-time interface.
- They are both array types, and their component types are the same run-time type (§10). // JLS 4.3.4

Type Compatibility

Type equivalence is often a too hard restriction

Type compatibility: Rules to decide when it's safe to use one type instead of another

- Safe = no runtime errors

Using double (1.0) for integer (1) is always safe

- But not other way round! Compatibility has a direction (some exception)

Compatibility is specific to each programming language.

Type Compatibility in Java*

Defined by conversion rules (if compatible => automatic conversion)

Conversion depends on where in code (5 different context) the (11 different) types of conversions apply.
Conversions examples (compatible types)

- Widening primitive: int to long, float, or double, ... (18 more)
- Boxing: primitive types from/to reference type, i.e. int to Integer, Boolean to boolean, ...
- String conversion, any type can be converted to a string (in a specific context)!

Reference conversions .... upcoming...

Subtyping*

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)

... (in Java class and type are the same, more later...)

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

- All classes and interfaces: A:>A (super :> sub)
- If A:>B and B:>C then A:>C

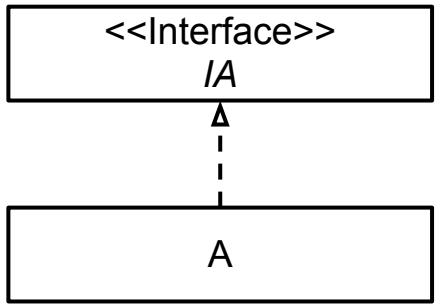
Subtyping, cont

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

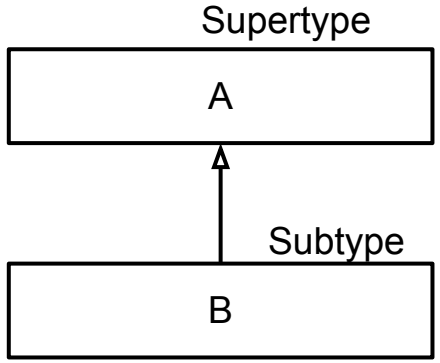
So subtype defined from supertype

UML for Subtyping

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



Arrow head
hollow.
Dashed line
for implements




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

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

Java Reference Conversion*

Widening Reference 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

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 Conversion, cont

Narrowing Reference Conversion

- From supertype to subtype
- Not type safe, operations in subtype possible not in supertype
- 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?!?!
```

The Array Loophole*

Java has decided that: if A:>B then A[] :> B[]

- 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

Casting*

Casts are explicit type conversion

```
// Is o really an integer??? We'll see...no compiler check
Integer i = (Integer) o;
```

A cast will bypass the compile time type checking and possible result in runtime exception. Casts may sometimes add code to a program:

- Code to actually perform the conversion
- Code to perform semantic checks on the conversion result
- Compile time error if cast never can succeed (not a subtype of)

AVOID!

Polymorphism

Subtyping implies that a variable/value (expression) can have many types (poly = many)

- Java is a **polymorphic language**, variables/values may have more types (possible both extends and implements)
- Contrast: (mostly) Monomorphic (C, Pascal,...)

Note: Polymorphism, polymorphic, etc. used different by different authors

Aside: Class Loading and Object Instantiation

Steps to create an object of type A

1. Need the class, look for A.class (file)
2. Load class A from file into memory, say A'. Use A' as template to construct an object "a" of type A in memory (return reference to)

   Object "a" will have a reference back to an (read only) object representing the type A (i.e. "a" knows it's type (class) and all associated type information)

   Note: All methods are shared between all objects of type A and located in A'

Runtime Type

What's happening here?

```
// IA interface type
public void doIt( IA a ){
    // a can be any subtype of IA
    // Which code to run? Can't know at compile time
    // Possible implementation doesn't exist yet
    a.call();
}
```

Must be able to find the type (class) of the object runtime, the **runtime type,** to be able to find the code to run (i.e. in which class is the method)

Polymorphism, cont

Selecting code to run based on object runtime type is a polymorphic behavior

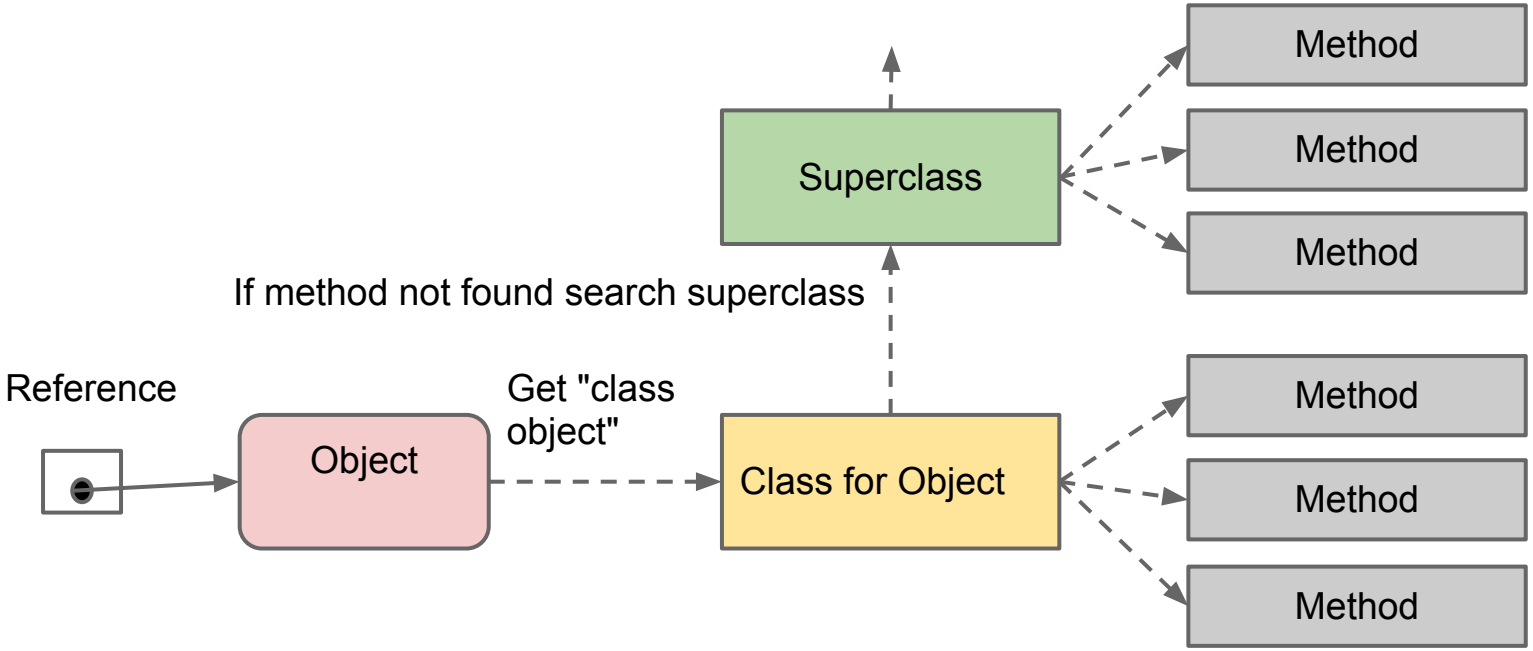
- Aka "late binding" or "dynamic dispatch"
- Must be a sub/supertype relationship
- All methods must have same **signature** and return type
  - Signature = name and parameter list
  - May not have more methods with same signature in same class
- This is known as **overriding**

Overriding*

- Default behaviour for methods in Java
- Can't override static methods (it's about instances)

Overriding, cont

During compilation method is "marked" as runtime. During execution search like this:



If fact it's very complicated to find the code to run

15.12. **Method Invocation Expressions (what to run, like this in code: o.someMethod())**
15.12.1. **Compile-Time Step 1:** Determine Class or Interface to Search
15.12.2. **Compile-Time Step 2:** Determine Method Signature
15.12.2.1. Identify Potentially Applicable Methods
15.12.2.2. Phase 1: Identify Matching Arity Methods Applicable by Subtyping
15.12.2.3. Phase 2: Identify Matching Arity Methods Applicable by Method Invocation Conversion
15.12.2.4. Phase 3: Identify Applicable Variable Arity Methods
15.12.2.5. Choosing the Most Specific Method
15.12.2.6. Method Result and Throws Types
15.12.2.7. Inferring Type Arguments Based on Actual Arguments
15.12.2.8. Inferring Unresolved Type Arguments
15.12.3. **Compile-Time Step 3:** Is the Chosen Method Appropriate?
15.12.4. **Run-time Evaluation of Method Invocation**
15.12.4.1. Compute Target Reference (If Necessary)
15.12.4.2. Evaluate Arguments
15.12.4.3. Check Accessibility of Type and Method
15.12.4.4. Locate Method to Invoke
15.12.4.5. Create Frame, Synchronize, Transfer Control // JLS

Usage Overriding

Overriding is a key feature of OO-programs

- Separate contract (interface from implementation (class). Implementation can change without affecting other parts. Will reduce dependencies, more to come...
- Object knows what to do, reduced need for selection (if, switch)
- One way to implement the Open Closed principle (next slide)
- Also: Reuse code, eliminate duplicate code

Always put annotation **@Override** on overriden methods, more to come...

Open Closed Principle*

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification" R.C. Martin

- I.e. we should not modify tested (proved) code to extend functionality, instead we should add new code

Clearly this is supported by subclassing. We extend by adding new subclasses, not changing existing

Explicit use Of Runtime Type*

Every object knows it's type (runtime type information, RTTI)

Possible to use explicit in code

- Operator **instanceof**
- 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<? extends 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).
The type of void.class (§8.4.5) is Class<Void>. "// JLS 15.8.2

```
// Get class object for object o (of type A)
Class<? extends A> c = o.getClass();
```

"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

Static type vs Runtime Type

The static type (declared type) of an variable is the type given at the declaration

- Possible to type check compile time

Again: Runtime type is type during execution

- Not possible to check compile time

Polymorphism, cont, cont

```
// Methods not same signature but same name
public class A {
    public void doIt( int i ){...}
    public void doIt( double d ){...}
}
// Which method to run?
A a = new A();
a.doIt(1);
a.doIt(1.0);
```

This is possible to decide at compile time, choose closest match from name and parameters (number of, ordering and type)! This is known as **overloading**

Overloading*

Same name for different methods

- Overloading is decided compile time by object's static type and parameter list (possible conversions to match)
- Return type not involved
- Another polymorphic behaviour
- Convenient for programmer (doesn't need different names for closely related operations)
- Conversion and overloading sometimes blurs

Usage Overloading

Same name for logically "same" methods

- Example: add(int, int) and add(float, float)
- All overloaded methods should be in same class
- Again: Put @Override on overridden methods, possible to get overload by mistake
- Constructor overloading common

Constructor Overloading*

Constructor overloading common

- Constructor with most parameters is the base constructor

Overloaded constructors use base constructor with predefined values for some parameters

- A service to the user

Polymorphism, cont, cont, cont*

Variables have nothing with polymorphism to do!

- Only declared (static) type matters for variables
- Polymorphism is (for now) that implementation can vary
- Implementation of variables can't vary

Polymorphic Types*

Polymorphic Types

- All List methods can be applied to values of more than one type (List<Integer>, List<String>,...)
- More later ...

Aka parametric polymorphism or generic types

Classification of Polymorphism

Universal

- Override and polymorphic types are universal polymorphism, works uniformly and general for any type

Ad-hoc (specific)

- Overload and conversion, specified by special cases. Not general, ad-hoc polymorphism

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 annotations, possible more later?)

Null

Invented by C.A.R. Hoare (computer science gigant)

*"I call it my **billion-dollar mistake**. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."*

Nullable Types

Java allows any reference variable to be set to null

- null-value, represented by the literal **null**, is the only value in the Null-type
- null is a value representing "not a valid object" (but it's not an object)
- *"The direct supertypes of the null type are all reference types other than the null type itself" //* JLS 4.10.2
- null == null always true
- ```
  // In practice we have this ...
  StringOrNull s = ...;
  IntegerOrNull i = ...;
  ```
  ```

NullPointerException (NPE)

RuntimeException, easy to locate, but sometimes hard to find cause

```
// Must check before ... but what if we forget?
if( map.containsKey("svea")){
    Person p = map.get("svea");

// Or check after ... but what if we forget?
Person p =  map.get("svea"))
if( p != null ){
```

Handling null's*

Have a type (null) not "handled" by the type system

What to do?

- Object (class)-internal null's accepted
- Incoming null's (method-parameters)
- Outgoing null's (method-results)

Handling Incoming null's

Hard, not much to do ...

- Checks doesn't help much
  - Accept NullPointerException
  - Throw IllegalArgumentException (with a possible better error message)
- Always check for null if value should be stored in some collection or sent along to other object
  - Throw IllegalArgumentException (more to come how to...)

Handling Outgoing null's

Never (or at least very rarely) return nulls

If result is a Collection

- Return empty Collection (String return "")

If result is a single value. Hard..

- ... quick look at Haskell way... (next slide)

Haskell Maybe Monad

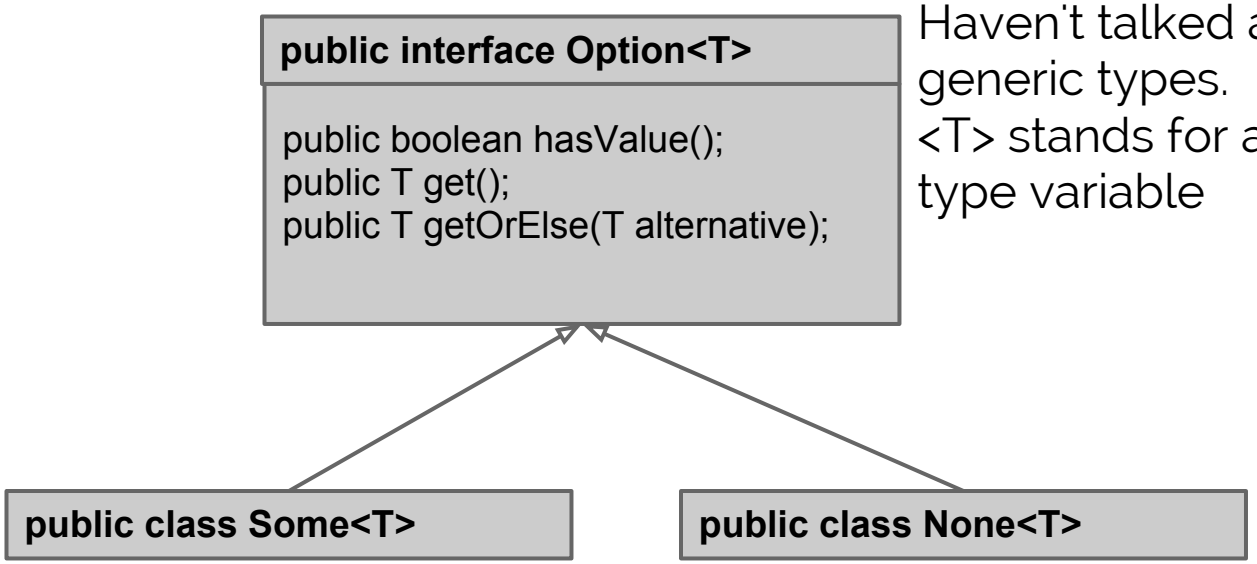
The Maybe monad represents computations which might "go wrong", in the sense of not returning a value

```
// Definition of Maybe
data Maybe a = Just a | Nothing
     deriving (Eq, Ord)

// Usage, Bob possible not in phonebook, what to do?
> lookup "Bob" phonebook
Just "01788 665242"
> lookup "Blblblb" phonebook
Nothing
```

Option<T>*

Possible to do something similar to Maybe in Java



Haven't talked about generic types.
<T> stands for any type, a type variable

Usage Option<T>

Can't use returned reference directly, check enforced by type system

```
// Impossible to forget to check
Option<SomeClass> o = m.getIt();
SomeClass value = o.getOrElse(...); //Possible default
value
```

Null: Arrays and Collections*

"An array created using new Object[10] has 10 null pointers. That's 10 more than we want, so use collections instead, or explicitly fill the array at initialisation"

Still can get NPE from collections

```
// Hmm..
List<Integer> is = new ArrayList(4);
int i = is.get(2);
```

Type Inference

In Java we mostly have to explicitly put 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...

Terminology

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

- Overloading is 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 data
- 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
- Null was a mistake, avoid