# Introduction to Type Systems for Object-Oriented Programming Languages

# The Purpose of Types

- Organize the untyped universe in different ways for different purposes
  - classification of objects in terms of the purposes
- Types arise naturally, even starting from the untyped universe
  - set of objects with uniform behavior may be named and are referred to as types
- Types provide implicit context
  - Compilers can infer more information, so programmers write less code
  - *E.g.* the expression `a+b` in Java may be adding two integer, two floats or two strings depending on the context
- Types provides a set of semantically valid operations
  - Compilers can detect semantic mistakes
  - *E.g.* `stacks` in support `push()` and `pop()`, but complex numbers do not

# What is a Type?

- Three points of view:
  - Denotational: a set of values sharing some properties
    - A value ***v*** has type ***t*** if ***v*** is an element of ***t***
  - Constructive: a type is built-in type or a composite type
    - Primitive types: booleans, integers, floats, chars ...
    - Composite types: created using type constructors for lists, tuples
      - E.g. In Java, `boolean` is a built-in type, while `boolean[]` is a composite type
  - Abstraction-based: a type is an interface that defines a set of consistent operations
    - E.g. `int` = set of all objects that are produced by integer-valued functions
    - E.g. Class type `Person` = set of all objects that understand the public methods of class `Person`
- These points of view complement each other

# Determining Types

- "Untyped languages may enforce safety by performing run time checks"
- "Typed languages may also use a mixture of run time and static checks"
- Is an untyped language that enforces safety comprehensively at run time equivalent to a typed language that uses run time checks exclusively?
  - Cardelli argues languages should be safe and typed but which typing
- Explicit typing
  - Function and variable types determined by declaration
  - Type (usually) invariant throughout execution
  - E.g., Pascal, Algol, C, C++, Java
- Implicit typing
  - Function and variable types are determined by usage
  - E.g., Prolog, Lisp, Smalltalk
- Mixed
  - Implicit typing by default, but allows explicit type declarations
  - E.g., Miranda, Haskell, ML

# Types in Java

- Implementation types (set of values)
  - Primitive like int, float, etc.
  - Composite like Array, String
  - User-defined types like Classes, Enumerations
- Interface types (set of consistent operations)
  - Abstract class types
  - Interface types in Java
- The study of type systems has important applications in
  - software engineering, language design, high-performance compilers, and security

| Java Types | Primitives | Composite |
| --- | --- | --- |
| Built-in | byte<br>short<br>int<br>long<br>float<br>double<br>char<br>boolean | Strings<br>Arrays |
| User defined | n/a | Classes<br>Interfaces |

# Static and Dynamic Typing

- A language is strongly typed if it is possible to ensure that every expression is type consistent based on the program text alone (e.g., Java)
  - All expressions and objects must have a type
  - All operations must be applied in appropriate type contexts
- A language is statically typed if it is always possible to determine the (static) type of an expression based on the program text alone (e.g., Pascal, C, Java)
- A language is dynamically typed if only values have fixed type (e.g., Lisp, Smalltalk)
  - Variables and parameters may take on different types at runtime, and must be checked immediately before they are used

# Type Checking

- Type Checking is the process of ensuring that the operands / arguments of an operator / procedure are consistent w.r.t. a set of rules associating a type with every expression of the language
  - These consistency/compatibility rules are known as the type system supported by most high-level languages
- Type consistency/compatibility may be assured by
  - compile-time type-checking, type inference, or
  - dynamic type-checking

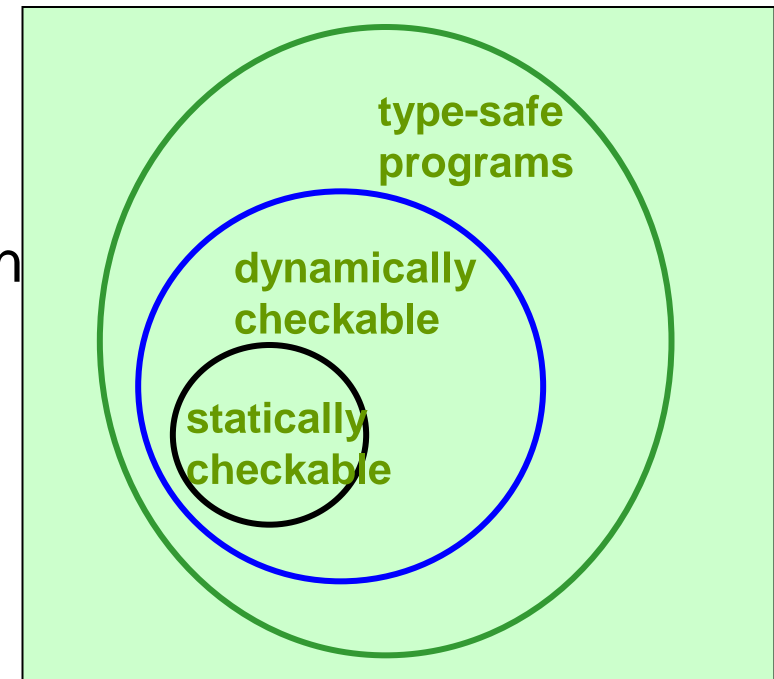
# Why Static Types?

- Static typing, based on a sound type system ("well-typed programs do not go wrong") is a basic requirement for robust systems programming
  - Static types are like proven assertions about a program

- Static types are useful for improving
  - Robustness: Early elimination of type errors
  - Verification: properties and invariants expressed in types are verified by the compiler ("a priori guarantee of correctness")
  - Readability: Types provides an excellent documentation
  - Efficiency: Type information allows optimizations
  - Design: Types provide a language and discipline for design of data structures and program interfaces
  - Abstraction: Support for interfaces, abstract data types, modules based on types
  - Scalability: support for orderly evolution of software since consequences of changes can be traced

# What is a Type System?

- A type system consists of
  - A mechanism for defining types and associating them with certain language constructs
  - A set of rules for type-checking:
    - Type compatibility: whether a value can be used in a certain context
    - Type inference: type of expression based on parts
    - Type equivalence: whether two values have same type




- Output of a type system:
  - There are type-errors (wrt type system) => Program is NOT type-safe
  - There are no type-errors (wrt type system) => Program is type-safe

# How to Type-Check?

- Definition: Type statements are of the form:

  **<expr> : <type>**

  meaning that an expression

  <expr> 'is-of-the-type' (the '**:**' symbol) <type>

- Examples:
  - `3 : int`
  - `3+4 : int`
  - `3.14 : real`
  - `"abc" : String`
  - `while (x < 5) {x++;} : Stmt`

# How to Type-Check?

- Definition: Type rules are of the form:

$$\frac{e_1 : t_1 \quad e_2 : t_2 \quad \ldots \quad e_n : t_n}{f\ e_1\ e_2\ \ldots e_n : t} \ (\text{rule name})$$

where each $e_i : t_i$ is a type statement, $n \geq 0$

The rule is interpreted as "**IF** $e_1$ is of type $t_1$ and … and $e_n$ is of type $t_n$ **THEN** $f\ e_1\ e_2\ \ldots e_n$ is of type $t$"

# How to Type-Check?

- Examples of type rules:
  - Rule for constants:

$$\frac{}{1 : \texttt{int}} \qquad \frac{}{2 : \texttt{int}} \qquad \frac{}{3 : \texttt{int}} \qquad \ldots\ldots$$

  - Rule for addition:

$$\frac{E_1 : \texttt{int} \qquad E_2 : \texttt{int}}{E_1 + E_2 : \texttt{int}} \ (+)$$

  - Rule for boolean comparison:

$$\frac{E_1 : \texttt{int} \qquad E_2 : \texttt{int}}{E_1 == E_2 : \texttt{bool}} \ (==)$$

# How to Type-Check?

- Examples of type rules:
  - Rule for assignment statement:

$$\frac{\texttt{x} : \texttt{T} \quad \texttt{E} : \texttt{T}}{\texttt{x := E;} \ : \ \texttt{Stmt}} \ (\texttt{:=})$$

  - Rule for if-statement:

$$\frac{\texttt{E}_1 : \texttt{Bool} \quad \texttt{S}_1 : \texttt{Stmt} \quad \texttt{S}_2 : \texttt{Stmt}}{\texttt{if (E}_1\texttt{) \{S}_1\texttt{\} else \{S}_2\texttt{\}} : \texttt{Stmt}} \ (\texttt{if})$$

# How to Type-Check?

- Rules of Type Checking
  - Type of value => known in advance
  - Type of variable => known in the declaration
  - Type of function => known from the type of arguments (in declaration) and type of result (also in declaration)
  - Type of expression => inferred from sub-expression
- Example:
  - Values have static types defined by the programming language
  - Variables and expressions have dynamic types determined by the values they assume at run-time

*declared*, static type is `Applet`

```
Applet    myApplet    =    new GameApplet();
```

actual dynamic type is `GameApplet`

static type of value is `GameApplet`

# How to Type-Check?

- Given the program:

```
int x;
…
x := x+1;
…
```

A program/expression is type-safe if we can construct a derivation tree to give a type for that program/expression

…And Given the rules:

$$\frac{}{1 : int} \quad \frac{}{2 : int} \quad \frac{}{3 : int} \quad \cdots\cdots$$

$$\frac{E_1 : int \quad E_2 : int}{E_1 + E_2 : int} \; (+)$$

$$\frac{E_1 : int \quad E_2 : int}{E_1 == E_2 : bool} \; (==)$$

$$\frac{x : T \quad E : T}{x := E; \; : \; Stmt} \; (:=)$$

$$\frac{E_1 : Bool \quad S_1 : Stmt \quad S_2 : Stmt}{if \; (E_1) \; \{S_1\} \; else \; \{S_2\} : Stmt} \; (if)$$

$$\frac{\frac{}{x : int} \quad \frac{\frac{}{x : int} \quad \frac{}{1 : int}}{x+1 : int} \; (+)}{x:=x+1; \; : \; Stmt} \; (:=)$$

# How to Type-Check?

- Given the program:

```
int x;   float y;
...
if (x == 3) {
   y := x;
} else {
   x := x+1;
}
...
```

A program/expression is type-safe if we can construct a derivation tree to give a type for that program/expression

Follow the rules! Try to build tree.
Cannot build tree=> Not type safe

…And Given the rules:

$$\frac{}{\texttt{1: int}} \quad \frac{}{\texttt{2: int}} \quad \frac{}{\texttt{3: int}} \quad \ldots\ldots$$

$$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int}}{E_1 + E_2 : \texttt{int}} \;(+)$$

$$\frac{E_1 : \texttt{int} \quad E_2 : \texttt{int}}{E_1 == E_2 : \texttt{bool}} \;(==)$$

$$\frac{x : T \quad E : T}{x := E; \;:\; \texttt{Stmt}} \;(:=)$$

$$\frac{E_1 : \texttt{Bool} \quad S_1 : \texttt{Stmt} \quad S_2 : \texttt{Stmt}}{\texttt{if } (E_1)\ \{S_1\} \texttt{ else } \{S_2\} : \texttt{Stmt}} \;(\texttt{if})$$

$$\frac{\dfrac{\dfrac{}{x : int} \quad \dfrac{}{3 : int}}{x==3 : Bool}(==) \quad \dfrac{???}{y:=x; : Stmt}(:=) \quad \dfrac{\dfrac{}{x : int} \quad \dfrac{\dfrac{}{x : int} \quad \dfrac{}{1 : int}}{x+1 : int}(+)}{x:=x+1; : Stmt}(:=)}{if\ (x==3)\ \{y:=x;\}\ else\ \{x:=x+1;\} : Stmt}(if)$$

# Monomorphism vs. Polymorphism

- Polymorphism = poly (many) + morph (form)
- Languages like Pascal have monomorphic type systems: every constant, variable, parameter and function result has a unique type
  - good for type-checking
  - bad for writing generic code
    - it is impossible in Pascal to write a generic sort procedure
- Monomorphism: every value belongs to exactly one type
- Polymorphism: a value can belong to multiple types
- Mostly Monomorphic . . . Mostly Polymorphic
  - One or the other characterizes individual languages

# Polymorphism

- Polymorphism, as it relates to:
  - values and variables: may have more than one type
  - functions: arguments can be of more than one type
  - types: operations are applicable to operands of more than one type
- Polymorphism can be classified into 2 categories
  - Ad-hoc Polymorphism: is obtained when a function works, or appears to work on several different types which may not exhibit a common structure and may behave in unrelated ways for each type
  - Universal Polymorphism: is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure

# Polymorphism: A Taxonomy (Cardelli&Wegner 1985)

**Universal: infinite number of types with common structure**

- Polymorphism
  - Universal
    - Parametric: uniformity of type structure is achieved by type parameters
    - Inclusion: object can belong to many different classes that need not be disjoint (subtypes & inheritance)
  - Ad Hoc
    - Overloading: same name used to denote different functions. Use determined from context
    - Coercion: a semantic operation required to convert an argument to a type expected by a function

**Ad Hoc: finite set of potentially unrelated types**

# Ad Hoc Polymorphism

- Example
  - `3 + 4`
  - `3.0 + 4`
  - `3 + 4.0`
  - `3.0 + 4.0`
- Overloading:
  - + is of `int*int->int` and `float*float->float`
  - constants get coerced from `int` to `float`
- Coercion:
  - Widening: coercing a value into a larger type
    - e.g., `int` to `float`, subclass to superclass
  - Narrowing: coercing a value into a smaller type
    - looses information, e.g., `float` to `int`

# Coercion

- A coercion is a operation that converts the type of an expression to another type
  - It is done automatically by the language compiler
- If the programmer manually forces a type conversion, it's called casting

$$\frac{E : \mathtt{int}}{E : \mathtt{float}} \quad \text{(Int-Float Coercion)}$$

```
int x; float y;
...
y = x;
...
```

# Example of the use of Coercion

```
int x;   float y;
…
if (x == 3) {
   y := x;
} else {
   x := x+1;
}
…
```

Typing rules:

$$\frac{}{1: int} \quad \frac{}{2: int} \quad \frac{}{3: int} \quad \ldots$$

$$\frac{E_1 : int \quad E_2 : int}{E_1 + E_2 : int} \; (+)$$

$$\frac{E_1 : int \quad E_2 : int}{E_1 == E_2 : bool} \; (==)$$

$$\frac{x : T \quad E : T}{x := E; \; : \; Stmt} \; (:=)$$

$$\frac{E_1 : Bool \quad S_1 : Stmt \quad S_2 : Stmt}{if\ (E_1)\ \{S_1\}\ else\ \{S_2\} : Stmt} \; (if)$$

Add in new rule…

$$\frac{E : int}{E : float} \; \text{(Int-Float Coercion)}$$

$$\frac{\dfrac{\dfrac{}{x : int} \quad \dfrac{}{3 : int}}{x==3 : Bool}(==) \quad \dfrac{\dfrac{}{y : float} \quad \dfrac{\dfrac{}{x : int}}{x : float}(Coercion)}{y:=x; : Stmt}(:=) \quad \dfrac{\dfrac{}{x : int} \quad \dfrac{\dfrac{}{x : int} \quad \dfrac{}{1 : int}}{x+1 : int}(+)}{x:=x+1; : Stmt}(:=)}{if\ (x==3)\ \{y:=x;\}\ else\ \{x:=x+1;\} : Stmt}(if)$$

# Coercion

## Coercion

### Widening

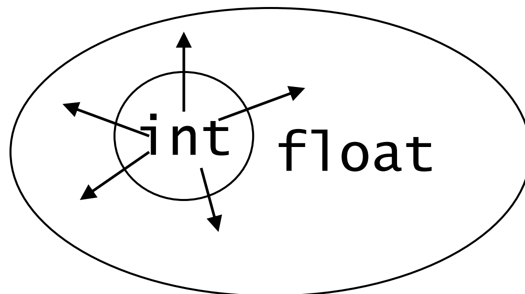
Widening coercion converts a value to a type that can include (at least approximations of) all of the values of the original type

Widening is safe most of the time. It can be unsafe in certain cases

### Narrowing

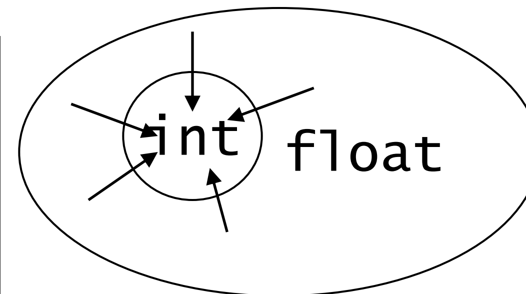
Narrowing coercion converts a value to a type that cannot store (even approximations of) all of the values of the original type

Narrowing is unsafe. Information may be lost during conversion of type




Theoretically speaking,

int ⊆ float



# Overloading

- An overloaded operation has different meanings, and different types, in different contexts

$$\frac{E_1 : \texttt{int} \qquad E_2 : \texttt{int}}{E_1 + E_2 : \texttt{int}} \ (\texttt{+-int})$$

$$\frac{E_1 : \texttt{float} \qquad E_2 : \texttt{float}}{E_1 + E_2 : \texttt{float}} \ (\texttt{+-float})$$

# Example of the use of Overloading

```
int x,y,z;   float a,b,c;
…
if (x == 3) {
    x := y + z;
} else {
    a := b + c;
}
…
```

$$\frac{}{1:\ int} \quad \frac{}{2:\ int} \quad \frac{}{3:\ int} \quad \ldots\ldots$$

$$\frac{E_1 : int \quad E_2 : int}{E_1 + E_2 : int}\ (+)$$

$$\frac{E_1 : int \quad E_2 : int}{E_1 == E_2 : bool}\ (==)$$

$$\frac{x : T \quad E : T}{x := E; \ : \ Stmt}\ (:=)$$

$$\frac{E_1 : Bool \quad S_1 : Stmt \quad S_2 : Stmt}{if\ (E_1)\ \{S_1\}\ else\ \{S_2\} : Stmt}\ (if)$$

Add in new rule…

$$\frac{E_1 : float \quad E_2 : float}{E_1 + E_2 : float}\ (+\text{-}float)$$

$$\frac{\dfrac{\dfrac{}{x : int} \quad \dfrac{}{3 : int}}{x==3 : Bool}(==) \quad \dfrac{\dfrac{}{x : int} \quad \dfrac{\dfrac{}{y:int} \quad \dfrac{}{z:int}}{y+z : int}(+)}{x:=y+z; : Stmt}(:=) \quad \dfrac{\dfrac{}{a : float} \quad \dfrac{\dfrac{}{b:float} \quad \dfrac{}{c:float}}{b+c : float}(+\text{-}float)}{a:=b+c; : Stmt}(:=)}{if\ (x==3)\ \{x:=y+z;\}\ else\ \{a:=b+c;\} : Stmt}(if)$$

# Inclusion Polymorphism

- Model subtypes and inheritance
  - A type **b** is a subtype of type **p** if values of type **b** can be used in any context where type **p** is expected without introducing errors
  - Types = Sets of Values, Subtyping = Subsets of Values
- Embodied by a subtype hierarchy
  - Inclusion polymorphism = Subtype polymorphism
- Note that
  - Subclassing $\neq$ Subtyping
  - Subclassing $\Rightarrow$ Subtyping
  - Subclassing $<\neq$ Subtyping
- Subtyping is about substitutability
- Subtyping without subclassing (structural subtyping) in Java:
  - interfaces
  - arrays
  - parametric types

# Subtyping

- Subtype-relationships
  - `int <: float`
  - `C <: B if C extends B`
  - `C <: I if C implements I`
  - `s <: t if s is a structural subtype of t`
    - `if s <: t then s[] <: t[] (arrays` more latter)
- Additional Rules
  - If `t` is a type, then `t[] <: Object and t[] <: Object[]`
  - If `C` is a class, then `null <: C`
  - If `s <: t` and `t <: r`, then `s <: r`
- Note that interfaces can have multiple subtypes (implements, extends)

# Widening Conversions: Java Subtypes

- Can convert from S to T if,
  - if S is a subclass of T
  - if S implements interface T
  - if S is NULL and T is any class type, interface type or array type
  - if S is a subinterface of T
  - if S is an interface and T is ‘`Object`’
  - if S is an array and T is ‘`Object`’
  - if S is an array and T is `Clonable`
  - If S is an array `SC[]`, `T` is an array `TC[]`, and
    - SC and TC are reference types (more latter), and widening conversion from SC to TC
    - What if SC and TC are not reference types?

# Narrowing Conversions : Java Supertypes

- Can convert from S to T if,
  - S is superclass of T
  - S is a class, T is an interface, S is not a final class
  - S is ‘Object’ T is any array or interface type
  - S is an interface, T is not a final class
  - S is interface, T is final class , and T implements S
  - S and T are interfaces, but S is not a subinterface of T and they don’t declare incompatible methods
  - S and T are array types and there is a narrowing conversion from their element types

- Narrowing conversions require run-time type tests
  - What are the tests involved in the last narrowing rule?

# Named and Anonymous Types

- Difference between type names and anonymous type names

- The type of a variable is either described through:
  - A type name:
    - ❶ those names defined using a type definition command. (eg. 'type' for Pascal, 'typedef' for C, class/interface for Java), or…
    - ❷ the primitive numeric types (e.g. int, float)
  - Or directly through a type constructor (e.g. array-of, type parameter)
    - In this case, the variable has an anonymous type name

# Structural vs. Nominal Type Equivalence & Subtyping

- Some languages define subtyping (or type equality) implicitly, based on the structure of the data
  - Terminology: structural equality or structural subtyping
  - Example: `{x:int; y:int; z:int}` <: `{x:int; y:int}`
  - Advantage: relationships among types can be inferred from the structure; More flexible, yet the flexibility can be bad too (different names for the same types)
  - Drawback: Harder to implement since entire structures must be compared
    - Other issues to consider: eg. Arrays, records or enumerations with same sizes but different subscripts – are they the same type?
- Some languages define subtyping (or type equality) explicitly, based on names for the types
  - Terminology: nominal equality or nominal subtyping
  - Of course declared subtypes are still subject to structural constraints
  - Advantage: Easy to implement checking, since we need only compare the name; types can make finer distinctions of the same types
  - Drawback: Very restrictive, inflexible

# Named and Structural Subtyping in Java

- Named subtyping in Java

```
class B {int foo() {...}}
class C extends B {
          int foo() {...}}
B p = new C();
int i = p.foo();
        // executes C.foo
```

- Implementation
  - method foo is looked up in a dispatch table at run time

- Subtype relationship not declared in class declaration

```
interface I {int foo();}
class C {int foo () {...}}
I p = new C();
int i = p.foo();
         // executes C.foo
```

- Same type check as for ‘implements I’

- Same table look up as before

# Type Equivalence

- When are two types equivalent ($\equiv$)?

Rule 1: For any type name T, $T \equiv T$

Rule 2: If C is a type constructor and $T_1 \equiv T_2$, then $CT_1 \equiv CT_2$

Rule 3: If it is declared that type name = T, then name $\equiv$ T

Rule 4 (Symmetry): If $T_1 \equiv T_2$,then $T_2 \equiv T_1$

Rule 5 (Transitivity): If $T_1 \equiv T_2$ and $T_2 \equiv T_3$, then $T_1 \equiv T_3$

- What rules do you want to use?

# Type Equivalence

- When are two types equivalent ($\equiv$)?

Rule 1: For any type name T, $T \equiv T$

Rule 2: If C is a type constructor and $T_1 \equiv T_2$, then $CT_1 \equiv CT_2$

Rule 3: If it is declared that type name = T, then name $\equiv$ T

Rule 4 (Symmetry): If $T_1 \equiv T_2$,then $T_2 \equiv T_1$

Rule 5 (Transitivity): If $T_1 \equiv T_2$ and $T_2 \equiv T_3$, then $T_1 \equiv T_3$

- Structural Equivalence will use all the rules to check for type equivalence

# Type Equivalence

- When are two types equivalent (≡)?

Rule 1: For any type name T, T ≡ T

~~Rule 2: If C is a type constructor and $T_1 \equiv T_2$, then $CT_1 \equiv CT_2$~~

~~Rule 3: If it is declared that type name = T, then name ≡ T~~

~~Rule 4 (Symmetry): If $T_1 \equiv T_2$,then $T_2 \equiv T_1$~~

~~Rule 5 (Transitivity): If $T_1 \equiv T_2$ and $T_2 \equiv T_3$, then $T_1 \equiv T_3$~~

- (Pure) Name Equivalence will use only the first rule
  - Unless the two variables have the same type name, they will be treated as different type

# Type Equivalence

- When are two types equivalent (≡)?

Rule 1: For any type name T, $T \equiv T$

~~Rule 2: If C is a type constructor and $T_1 \equiv T_2$, then $CT_1 \equiv CT_2$~~

Rule 3: If it is declared that type name = T, then name $\equiv$ T

Rule 4 (Symmetry): If $T_1 \equiv T_2$,then $T_2 \equiv T_1$

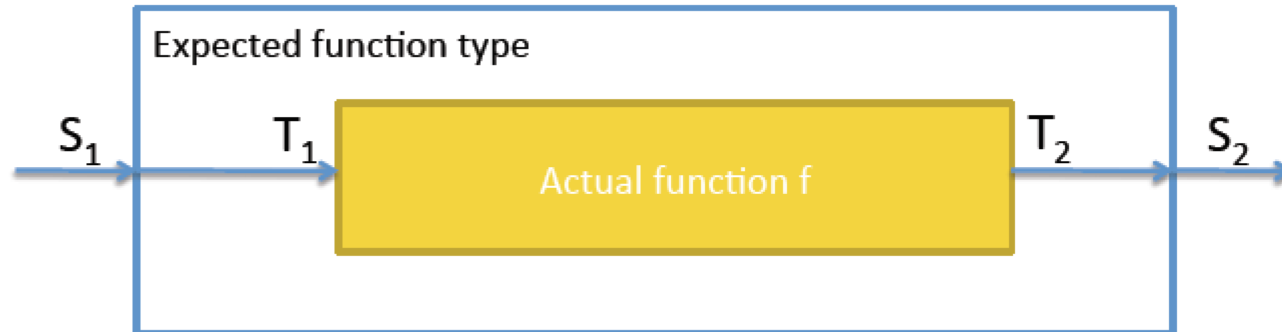
Rule 5 (Transitivity): If $T_1 \equiv T_2$ and $T_2 \equiv T_3$, then $T_1 \equiv T_3$

- Declarative Equivalence will leave out the second rule

# Subtyping for Java Method Types

- Signature Conformance
  - Subclass method signatures must conform to those of superclass
- Three ways signature could vary
  - Argument types
  - Return type
  - Exceptions
- How much conformance is needed in principle?
- What are the Java conformance rule
  - Java 1.1: method arguments and returns must have identical types as base method, may remove exceptions
  - Java 1.5: covariant method return type specialization

# Subtyping for Method Types

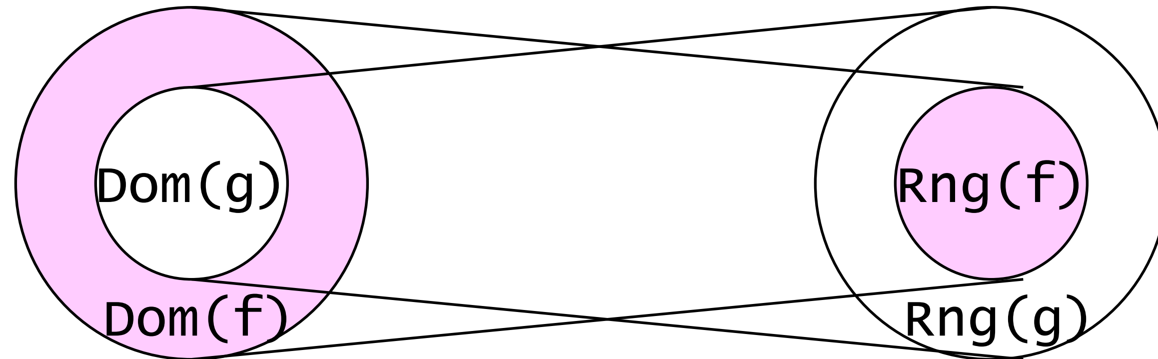



- Substitutability relationship
  - an instance of a subtype T can stand in for an instance of its supertype S: `T <: S`
  - we want the same functionality for method types (aka overriding): `(T1 -> T2) <: (S1 -> S2)`
- Consider a function $f: T_1 \rightarrow T_2$ as substitute of a function $g: S_1 \rightarrow S_2$
  - Need to convert (without casting) an $S_1$ to a $T_1$ and $T_2$ to $S_2$, so the argument type is contravariant and the return type is covariant

$(T_1 \rightarrow T_2) <: (S_1 \rightarrow S_2)$ iff $S_1 <: T_1$ and $T_2 <: S_2$

# Method Substitutability: Co, Contra & In Variance

**f**: $T_1$ -> $T_2$
**g**: $S_1$ -> $S_2$




- Contravariance of argument types: a relation between an overridden method (**f**) in a subclass and the original method (**g**) in which the type of each argument either remains unchanged or becomes generalised, `Dom(f)` can be substituted by `Dom(g)`
- Covariance of return type: a relation between an overridden method (**f**) in a subclass and the original method (**g**) in which the return type either remains un-changed or becomes specialised, `Rng(g)` can be substituted by `Rng(f)`
- Type invariance: a relation between an overridden method (**f**) in a subclass and the original method (**g**) in which the type of each argument (or return type) neither can be specialized and generalized
  - $(T_1 \rightarrow T_2)$ <: $(S_1 \rightarrow S_2)$ iff $T_1 = S_1$ and $T_2 = S_2$

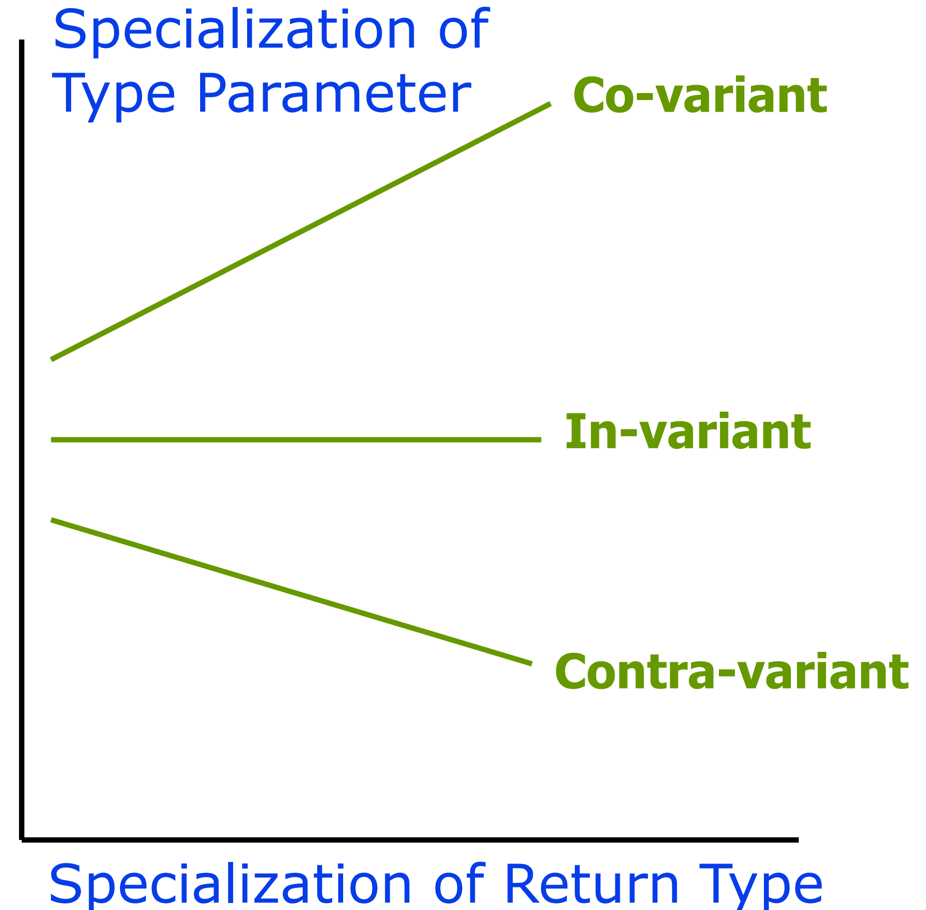
# Basic Terminology: Co, Contra, & In Variance

- variance ⇒ change
- co- ⇒ together
- contra- ⇒ opposite
- in- ⇒ absence

---

- co-variance ⇒ change together
- contra-variance ⇒ change opposite
- in-variance ⇒ no change

- Java is invariant for the most part ..
  - Method return types are covariant (Java 1.5)
  - Arrays are covariant



# Problems in Java Method Types

- Consider class **C** and method **m** having the type:

  **m : B X $P_1$ X … X $P_n$ -> R**

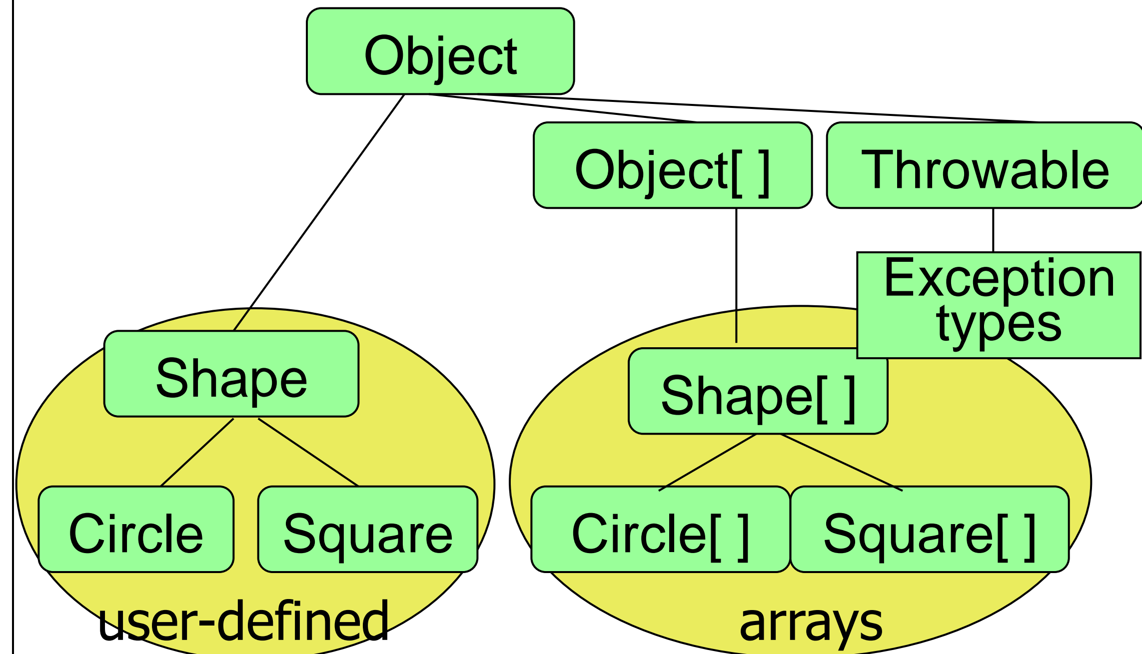
  - **R** is the return type and therefore covariant
    - But Java (1.1) requires it to be invariant (changed in 1.5)
  - Clearly, since **C, $P_1$,…$P_n$** are arguments, they must be contravariant
    - Java (1.1. and 1.5) requires them to be invariant
- Consider the following example, where **$P_i$ <: $P'_i$**:

```
class Base {                      class Sub extends Base
  {R m(P1 par1… Pn parn);}          R' m(P'1 par1… P'n parn);}
```

  - Java does not allow this overriding
    - The return values in overriding can be covariant
    - One can define contravariant argument types, but the Java typing system will consider this declaration as **overloading** instead of **contravariant overriding**

# Java Literal and Reference Types

- Recall: Two general kinds of Java types
  - Literal types – *not* objects have value semantics
    - Assignment is copy
  - Reference types have reference semantics (no need for pointers)
    - No syntax distinguishing `Object*` from `Object`
    - Assignment is aliasing
- Arrays and strings are somewhat anomalous
  - reference semantics but not classes
  - awkward syntax for string operations



# Java Array Types

- Automatically defined
  - Array type `T[ ]` exists for each class, interface type `T`
  - Cannot extended array types (array types are final)
  - Multi-dimensional arrays are arrays of arrays: `T[ ] [ ]`
- Treated as reference type
  - An array variable is a pointer to an array, can be null
  - Example: `Point[ ] x = new Point[array_size]`
  - Anonymous array expression: `new int[ ] {1,2,3, ... 10}`
- Every array type is a subtype of `Object[ ]`, `Object`
  - Length of array is not part of its static type
- Arrays have parametric polymorphism:
  - `Array<E>`: Example `Array<Point>, Array<ColouredPoint>`
  - "array" is the generic type: the type of things in the array is the type parameter

# Java Array Subtyping

- Java arrays are said to have covariant subtyping
  - if `S <: T` then `S[ ] <: T[ ]`
- Standard type error

```
class T {…}
class S extends T {…}
S[ ] sArray = new S[10];
T[ ] tArray = sArray;//considered OK since S[]<:T[]
tArray[0] = new T(); //compiles, but run-time error
                     //raises ArrayStoreException
```

- If you try to store an object of type `T` into an array of type `S` via an alias array of type `T[]`, you get a class cast exception
  - If the type `S` were parameterized, this protection could not be enforced during run-time, as parameterized types are new in Java 5
  - So to prevent this, you are not allowed to create an array of a parameterized type! Sorry!

# Java Generic Programming (GJ)

- Java has a root class `Object`
  - Supertype of all object types allowing for “inclusion polymorphism”
    - Can apply operation on class `T` to any subclass `S <: T`
- Java 1.0–1.4 does support “parametric polymorphism” (like C++ templates)
  - Many consider this the biggest deficiency of Java: in collections
    - cannot specify the exact type of elements
    - must cast to specific classes when accessing
- Generic Programming allows programming with classes and methods parameterized with types: more abstractions over types in Java 1.5!!!
  - Classes, interfaces and methods can have a type parameter
    - Any class type can be plugged in
  - The class (interface) definition is compiled just like any other class (interface)
    - Once compiled, it can be used like any other class, except that the code must specify a class type to replace the parameter

# Java 1.0 vs Java 1.5

```
class Stack {
  void push(Object o) {
  ... }
  Object pop() { ... }
  ...}

String s = "Hello";
Stack st = new Stack();
...
st.push(s);
...
s = (String) st.pop();
```

```
class Stack<A> {
  void push(A a) { ... }
  A pop() { ... }
  ...}

String s = "Hello";
Stack<String> st = new
      Stack<String>();
...
st.push(s);
...
s = st.pop();
```

- Java Community proposal (JSR 14) incorporated into Java 1.5

# Why Generic Programming?

- Generic types are a powerful tool to write reusable object-oriented components and libraries
  - however, the generic language features are not easy to master and can be misused
    - their full understanding requires the knowledge of the type theory of programming languages
    - especially *covariant* and *contravariant* typing
- Why no Generics in Early versions of Java?
  - Basic language goals seem clear
  - Details take some effort to work out
    - Exact typing constraints
    - Implementation
      - Existing virtual machine?
      - Additional bytecodes?
      - Duplicate code for each instance?
      - Use same code (with casts) for all instances

# Java Generics are Typechecked

- In principle, Java generics supports statically-typed data structures
  - early detection of type violations
    - cannot insert a `String` into `Stack <Number>`
  - also, hides automatically generated casts (auto boxing/unboxing)
    - `st.push(12);` vs `st.push(new Integer(12));`
- Superficially resembles C++ templates (aka generic declarations)
  - C++ templates are factories for ordinary classes and functions
    - a new class is always instantiated for given distinct generic parameters (type or other)
- Java generic types are factories for compile-time entities related to types and methods
  - Java Generics changes the way one programs

# Non-Generic Pair Objects

```
package nonGeneric;
/**
 * A class representing pairs of objects using Object types
 */
public class Pair {
   private Object first;
   private Object second;
   public Pair(Object first, Object second) {
      this.first = first;
      this.second = second; }
   public Object getFirst() {
      return first; }
   public Object getSecond() {
      return second; }
   public String toString() {
      return "(" + first + "," + second + ")"; }
};
```

# Using Non-Generic `Pair` objects

- Define pairs of strings

```
Pair p1 = new Pair("Fred", "Jones");
System.out.println(p1);
```

- Use `getFirst()` and `getSecond()` methods (typecasting is necessary)

```
String first = (String) p1.getFirst();
String second = (String) p1.getSecond();
System.out.println(first);
System.out.println(second);
```

- Define pairs of names and ages

```
Pair p2 = new Pair("Fred", 18); // autoboxing for 18
String name = (String) p2.getFirst();
int age = (Integer) p2.getSecond()
System.out.println("Name = "+name+", age = "+age);
```

# Definition of a Simple Generic Class `Pair`

```
class Pair <T> {
  public T first;
  public T second;
  public Pair (T f, T s) { first = f; second = s; }
  public Pair () { first = null; second = null; }
}
```

- **T** can be used as a method argument type, a field member, or anywhere that a type can be used in a class definition
- You instantiate the generic class by substituting actual types for type variables, as: `Pair<String>`
- You can think the result as a class with a constructor `public Pair(String f, String s)`, etc..
- You can then use the instantiated generic class as it were a normal class (almost):
  - `Pair<String> pair = new Pair<String>("1","2");`

# Multiple Type Parameters are Allowed

- You can have multiple type parameters

```
class Pair<T,U> {
  public T first;
  public U second;
  public Pair (T f, U s) { first = f; second = s; }
  public Pair () { first = null; second = null; }
}
```

- To instantiate: `Pair <String, Number>`
- `Pair` is generic: Same behavior for any possible `T` and `U`
  - Single source: only one `pair.java`
    - Low maintence
  - Single binary: only one `pair.class`
    - No binary explosion (but imposes restrictions)

# Generic Pair Objects

```
package generic;
/**
 * A class representing pairs of objects of generic types
 * @param <T> the type of the first object in pair
 * @param <U> the type of the second object in pair
 */
public class Pair<T,U> {
   private T first;
   private U second;
   public Pair(T first, U second) {
      this.first = first;
      this.second = second; }
   public T getFirst() {
      return first; }
   public U getSecond() {
      return second; }
   public String toString() {
      return "(" + first + "," + second + ")"; }
};
```

# Using Generic Pair Objects

- Define pairs of strings

```
Pair<String,String>p1=new Pair<String,String>("Fred","Jones");
System.out.println(p1);
```

- Define pairs of strings & Integers (autoboxing/unboxing is used)

```
Pair<String,Integer> p2=new Pair<String,Integer>("Fred",18);
int age = p2.getSecond();
System.out.println(p2);
```

- Without autoboxing/unboxing we would need to use

```
Pair<String,Integer> p2 =
   new Pair<String,Integer>("Fred", new Integer(18));
int age = p2.getSecond().intValue();
System.out.println(p2);
```

- Define pairs of doubles

```
Pair<Double,Double> p1=new Pair<Double,Double>(1.23, 3.14);
System.out.println(p1);
```

- Note that `Pair<double,double>` or `Pair<int,int>` are illegal since only `Object` types can be used

# Generic Methods: Static Case

- A generic class may use operations on objects of a parameter type aka generic methods
- You can define generic methods both inside ordinary classes and inside generic classes

```
class Algorithms { // some utility class
  public static <T> T getMiddle(T[ ] a) {
   return a [ a.length / 2 ];
  }
. . .
}
```

- When calling a generic method, you can specify type

```
String s = Algorithms.<String>getMiddle(names);
```

- but in most cases, the compiler infers the type:

```
String s = Algorithms.getMiddle(names);
```

# Generic Types and Subtyping

- Consider the method `void method(Collection<Object> c)`
- Is `LinkedList<Object>` subtype of `Collection<Object>`?
  - Yes, covariant row types
- Is `LinkedList<Float>` subtype of `Collection<Float>`?
  - Yes, but not a subtype of `LinkedList<Number>`
- Is `LinkedList<Float>` subtype of `LinkedList<Object>`?
  - No! generic types are NOT covariant

```
LinkedList<Object> objLst =
    new LinkedList<Float>(); //compile error
```

- Why? Generics are a Java 5 add-on, and the class files must be Java 1-compatible and forgetting about the parameter types
  - To avoid abuse, subtyping ignores type parameters (invariant)

# Unbounded and Bounded Wildcard Types

- So, how do we declare method to work on any collection?
  `void method(Collection<?> c);`
  - Collection of "unknown": read only usage in method body!
    - `c.get()` returns an `Object`
    - `c.add(o)` is not allowed
    - Exception: `null`, which is a member of every type
- Upper bound: `<? extends Foo>`
  - Wildcard can be any subclass of `Foo` including `Foo` itself
  `void method(Collection<? extends Foo> c);`
    - As with unbounded wildcards, `c` is read-only (immutable)
- Lower bound: `<? super Foo>`
  - Wildcard can be any superclass of `Foo` including `Foo` itself
  `void method(Collection<? super Foo> c);`
    - `c` can be updated (mutable)
- Get and Put Principle: use an `extends` (`super`) wildcard when you only get (put) values out of an object
  - Don't use wildcards for both getting and putting values from the same object

# Wildcard Types Example

- Copy from one list to another:

```
public static<T> void copy(List <? extends T> src,
                           List <? super T> dst) {
 for (int i=0; i<src.length(); i++) {
     dst.set(i, src.get(i));
 }
}
```

- Getting elements :

```
public static double sum(Collection<? extends Number>
                                                nums) {
 double s=0.0;
 for (Number num: nums) s += num.doubleValue();
 return s;
}
```

# Wildcard Types Example

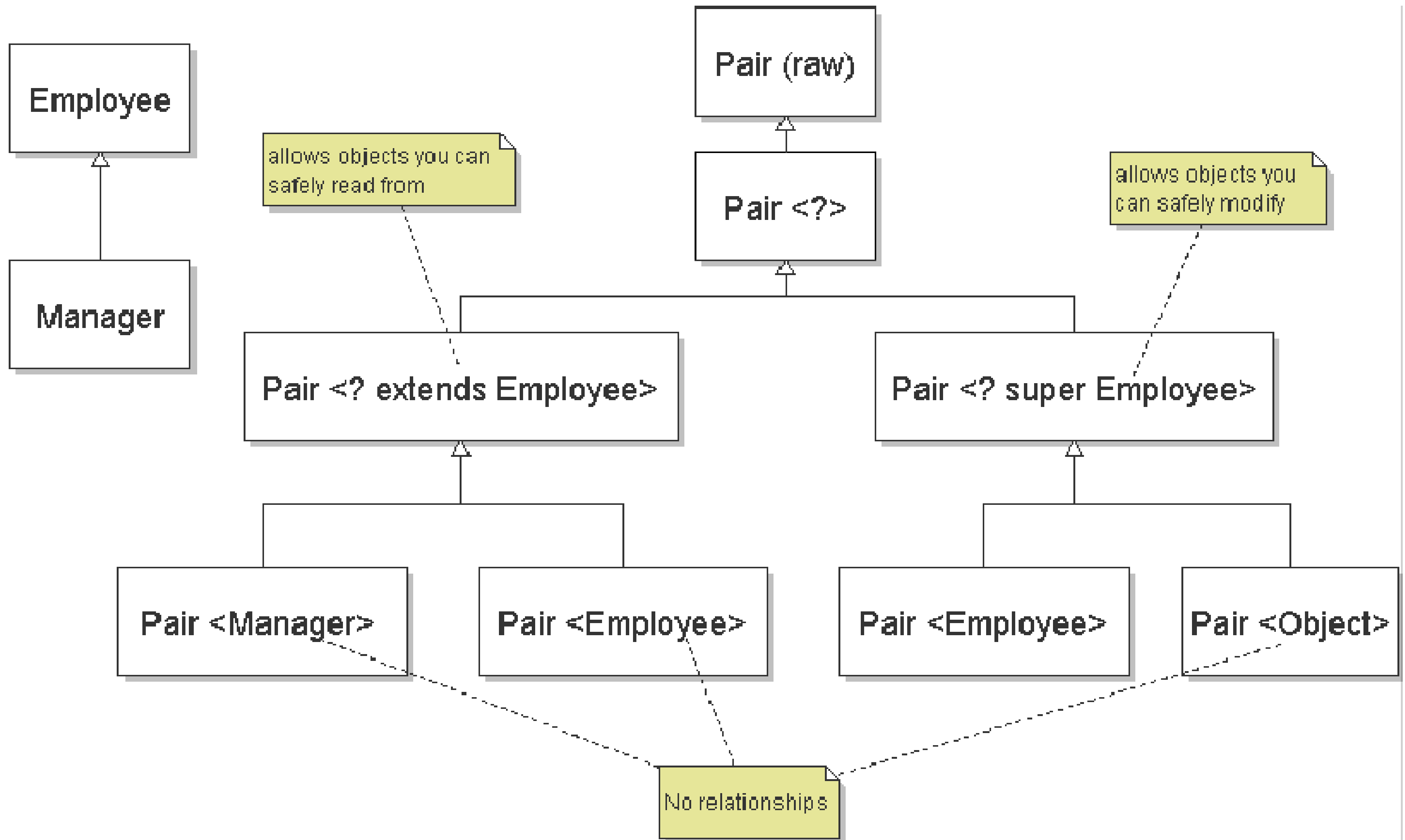
- Putting elements :

```
List<Object> objs = Arrays.<Object>asList(1,"two");
List<? super Integer> ints = objs;
ints.add(3); //OK
double dbl = sum(ints); // compile-time error
```

- Two Bounds? Not legal though plausible

```
double aumcunt(Collection<? super Integer, extends
 Number> coll, int n) // not legal
```

# More on Generic SubTyping



# Comments on Generic Subtyping

- `Pair<Manager>` matches `Pair<? extends Employee>`
  pair of elements of any subtype of `Employee` including itself
  => subtype relation (covariant typing)
- `Pair<Object>` matches `Pair<? super Employee>`
  pair of elements of any supertype of `Employee` including itself
  => subtype relation (contravariant typing)
- `Pair<Employee>` can contain only `Employees`, but `Pair<Object>` may be assigned anything (`Numbers`)
  => no subtype relation
- also: `Pair<T> <: Pair<?> <: Pair` (*raw*) `<: Object`

```
Pair pair1 = . . .;
pair1.first = new Double(10.0);
Pair <?> pair2 = . . .;
pair2.first = new Double(10.0); //ERROR
```

# More on Type Variables Bounds

- Recall the min algorithm: find the smallest item in a given array of elements
- To compile this, must restrict `T` to implement the `Comparable` interface that provides `compareTo()`

```
public static <T extends Comparable> T min(T[ ] a){
                                   //this is almost correct
  if (a.length == 0) throw new
                               InvalidArgumentException(…);
  T smallest = a[0];
  for (int i = 1; i < a.length; i++)
    if (smallest.compareTo(a[i]) > 0)//T constraint
        smallest = a[i];
  return smallest;
}
```

# Multiple Type Variables Bounds

- However, `Comparable` is itself a generic interface (in Java 1.5)
  - moreover, any supertype of T may have extended it

```
public static <T extends Object & // bounding class
  Comparable <? super T>> T min(T[ ] a) { . . .
                            // the more general form
  T smallest = a[0];
  for (int i = 1; i < a.length; i++)
      if (smallest.compareTo(a[i]) > 0)//T
 constraint
          smallest = a[i];
  return smallest;
}
```

- cannot inherit multiple different instantiations of the same generic type (class or interface) an inherited generic type is fixed for subtypes, too

# Implementing Java Generics

- Generics processing in Java
  1. Check: type correctness
  2. Erase: all generic type information
  3. Compile: to byte code
- Java generic types are compile-time entities which are type checked
  - in C++, instantiations of a class template are compiled separately as source code, and tailored code is produced for each one
    - Link and see if all operations can be resolved
  - In Java, a generic type definition is compiled once for all, and a corresponding raw type is produced without linking
    - the name of the raw type is the same but type variables are removed (erased)
    - this requires programmer to give information about type parameter e.g. `<T extends ...>` `<... super T >`

# Generic Type Erasure

- Erased type variables are replaced by their bounding types (or `Object` if no bounds); e.g. the raw type for `Pair<T>` is `Pair`
  - Bytecode has some generic info, but objects don't
- `Pair<String>` and `Pair<Employee>` use the same bytecode generated as the raw class `Pair`
  - when translating generic expressions, such as

```
Pair <Employee> buddies = new Pair < . . .;
Employee buddy = buddies.first;
```

  - the compiler uses the raw class and automatically inserts a cast from `Object` to `Employee`:

```
Employee buddy = (Employee)buddies.first;
```

    - in C++, no such casts are required since class instantiations already use specific types

# Generic Type Erasure Rules

- If a class doesn't use type parameters in its definition, erasure doesn't change the class definition
- Parameterized types "drop" their type parameters

```
class Foo<T extends Number> {
  T m(Set<? Extends T> s){}
};
```

```
class Foo {
  Number m(Set s){}
};
```

- Every type parameter is mapped to the strongest type the compiler can reasonably assert
  - In the case of `<T>`, `T` is mapped to `Object`
  - In the case of `<T extends Comparable>`, `T` is mapped to `Comparable`
  - In the case of `<T extends Object&Comparable>`, `T` is mapped to `Object` (the first parameter)
- Casts are inserted wherever necessary to ensure code compilation

# Benefits and Drawbacks of Type Erasure

- Benefits:
  - Binary compatibility with older libraries: `List<String>` is translated to raw type `List`
    - code compiled using a "pre-generics" class works correctly
    - code written using a "pre-generics" class still compiles and work
- Drawbacks
  - Generic type information is not known at runtime
    - `List<Integer>` and `List<String>` refer to raw type `List`
  - More latter

# Overriding of Generic Methods

- Consider a generic class with a non-final method:

```
class Pair <T> { // parameter T is erased from code
  public void setSecond (T s) { second = s; } . . .
```

- To override such type-erased methods, the compiler must generate extra bridge methods:

```
class DateInterval extends Pair <Date> {
 public void setSecond (Date high) {// override
  if (high.compareTo (first) < 0) throw new . .
      second = high; // otherwise OK
 }
 public void setSecond (Object s) {// bridge method
  setSecond ((Date)s); // generated by compiler
 }
```

# Restrictions and Limitations

- Primitive type parameters (`Pair<int>`) not allowed
  - in C++, both classes and primitive types allowed
- Objects in JVM have non-generic classes:

```
Pair<String> strPair = new Pair<String> . .;
Pair<Number> numPair = new Pair<Number> . .;
b = strPair.getClass() == numPair.getClass();
assert b == true; //both of the raw class Pair
```

  - but byte-code has reflective info about generics
- Instantiations of generic parameter T are not allowed (in new expr)

```
new T()  //ERROR: whatever T to produce?
new T[10]
```

- Arrays of parameterized types are not allowed

```
new Pair <String>[10]; // ERROR
```

  - since type erasure removes type information needed for checks of array assignments
- Static fields with type parameters are not allowed

```
class Singleton<T> {
 private static T singleOne; // ERROR
```

  - since after type erasure, one class and one shared static field for all instantiations and their objects

# Wildcard Types Capture

- The wildcard type ? cannot be used as a declared type of any variable

```
Pair <?> p = new Pair <String> ("one", "two"); . .
p.first = p.second; // ERROR: unknown type
```

- But, some operations (accessors) have no type constraints

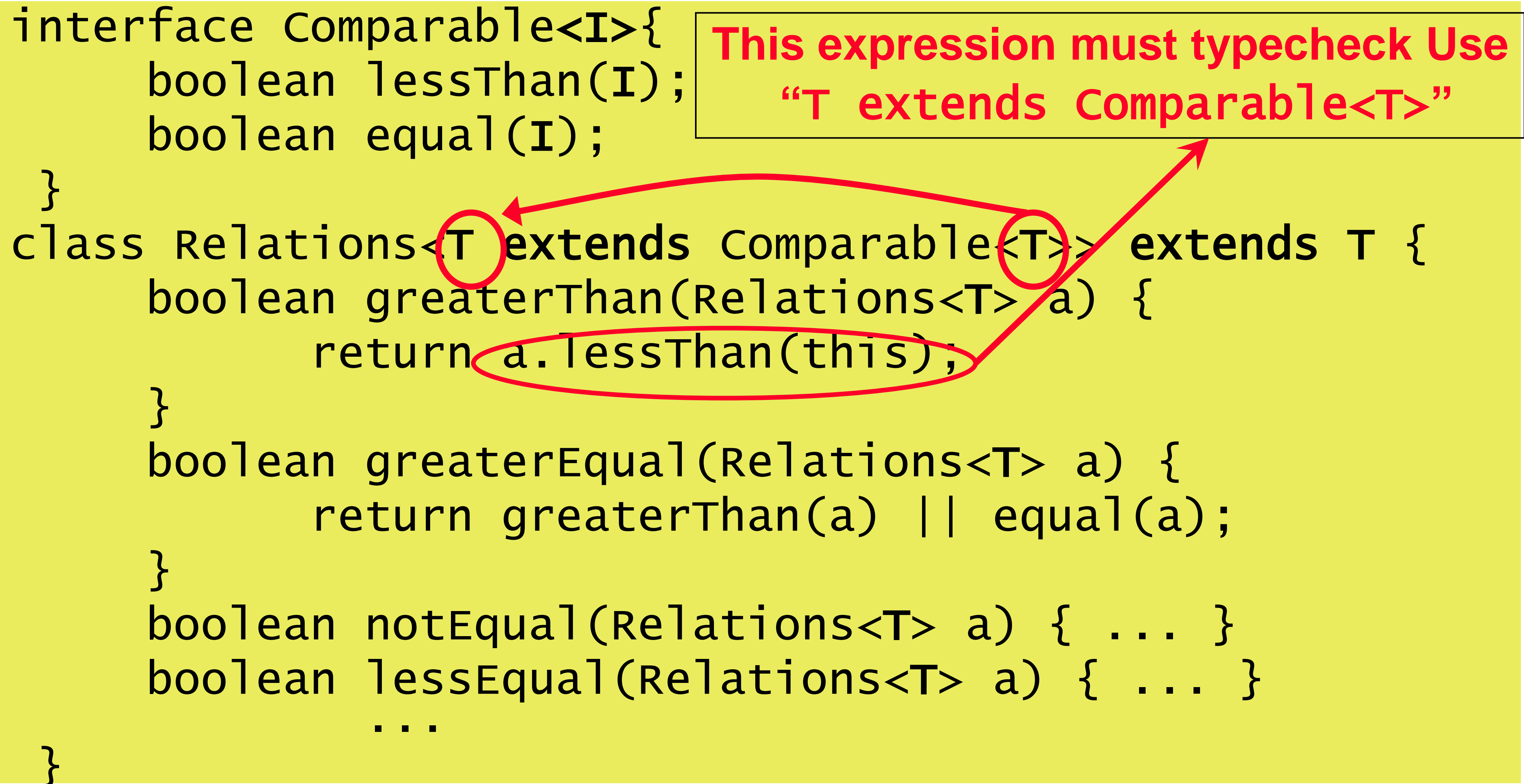
```
public static boolean hasNulls(Pair <?> p) {
 return p.first == null || p.second == null;
}
```

- Alternatively, you could provide a generic method to capture the wildcard:

```
public static <T> void rotate (Pair <T> p) {
  T temp = p.first; p.first = p.second;
  p.second = temp;
}
```

- The compile checks that such a capture is legal
  - e.g., the context ensures that T is unambiguous

# Interfaces Can Be Also Generic




Why is this form needed? `lessThan`: $T \times T \rightarrow$ `bool` is invariant in T

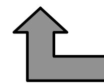
# Generic Types and Iterators

- Goal: design a minimal interface that you need
  - e.g., for max, implement to take any Collection

```
public static <T extends Object &
                Comparable <? super T>>
                T max(Collection <? extends T> c) {
// a hypothetical implementation:
Iterator <T> it = c.iterator();
  T largest = it.next(); //or throws NoSuchElement
  while (it.hasNext()) {
  T val = it.next();
  if (largest.compareTo(val) < 0) largest = val;
 }
 return largest;
}
```

# Cardelli on Type Systems

- Type system
  - purpose is to prevent occurrence of *execution errors* during runtime
- Type Sound Language
  - absence of execution errors holds for all program runs that can be described in a programming language
- Typechecker
  - method for determining if type errors occur
  - ambiguities in language specifications often lead to different type checker implementations, hampering language soundness.
- Type
  - “Upper bound” (maximal set) on range of values a variable can take on
- Typed Language
  - one in which variables can be given (nontrivial) types



How about “can assume”?

# More Cardelli on Types

- Explicit / implicit typing
  - as names suggest…
- Trapped errors
  - execution error when computation stops "immediately"
- Untrapped errors
  - execution errors that go unnoticed and cause arbitrary behavior
- Safe program fragment
  - one that does not (cannot?) cause untrapped errors to occur
- Safe language
  - one in which all program fragments are safe

# Off on Good Behavior

- Forbidden errors
  - all untrapped errors plus some trapped errors
    (what trapped errors might be included?)
- Good Behavior (well behaved)
  - no forbidden errors occur
  - a well behaved program fragment is safe
- Strongly checked language
  - One in which all (legal) program fragments have good behavior
    - no untrapped errors occur
    - none of the specified trapped errors occur
    - other trapped errors may occur - programmer must avoid them
  - (notice avoidance of “strongly typed”)

# Bounded Wildcards Example

```
public abstract class Shape {
  public abstract void draw(Canvas c);
}
```

```
public class Circle extends Shape {
      private int x, y, radius;
      public void draw(Canvas c) { ... }
}
```

```
public class Rectangle extends Shape {
      private int x, y, width, height;
      public void draw(Canvas c) { ... }
}
```

```
public void drawAll(
      List<? extends Shape> shapes) {
//...
}
```

# Function Types

- Function types allow one to deduce the types of expressions without the need to evaluate them:

  *fact :: Int -> Int*

  *42 :: Int ⇒ fact 42 :: Int*

- Curried types:

  *Int -> Int -> Int ≡ Int -> (Int -> Int)*

  and

  `plus 5 6 ≡ ((plus 5) 6)`

  so:

  *plus::Int->Int->Int ⇒ plus 5::Int->Int*

# List Types

- A list of values of type a has the type [a]:
  - *[ 1 ] :: [ Int ]*

- NB: All of the elements in a list must be of the same type!
  - `['a', 2, False]--` *this is illegal! can't be typed!*

# Record (Tuple) Types

- If the expressions $x_1, x_2, ..., x_n$ have types $t_1, t_2, ..., t_n$ respectively, then the tuple $(x_1, x_2, ..., x_n)$ has the type $(t_1, t_2, ..., t_n)$
  - *(1, [2], 3) :: (Int, [Int], Int)*
  - *('a', False) :: (Char, Bool)*
  - *((1,2),(3,4)) :: ((Int, Int), (Int, Int))*

- The unit type is written () and has a single element which is also written as ()

# Variant Record (Union) Types

- **data** Temp = Centigrade Float | Fahrenheit Float
- *freezing :: Temp -> Bool*
- freezing (Centigrade temp)= temp <= 0.0
- freezing (Fahrenheit temp)= temp <= 32.0

# Structural Subtyping

- Record types
  - `{k: Int, l: String}`
- Variant record types
  - `[k: Int, l: String, m: Bool]`
- Function types
  - `{k: Int, l: String} -> {m: Int}`

# Types

- A type declaration introduces a name (and scope) and a definition of the type of the value to which name is bound
  - `Int = {0,-1,1,-2,2,...}`
  - `String = {"abc", "0xzA%", ...}`
  - `String -> Int = {functions from String to Int}`
  - `Int x String = {(i,s)` | i is an Int and s is a String}
  - `Class Type C = {o | o is an instance of` **C** or of a subclass of **C**}
- Type errors:
  - `? 5 + [ ]`
  - `ERROR: Type error in application`
  - `*** expression : 5 + [ ]`
  - `*** term : 5`
  - `*** type : Int`
  - `*** does not match : [a]`
- Typeful programming is just a special case of program specification

# Overloading

- Multiple definition of the same name
- Method selection at compile time

```
class D extends C { ... }
int foo (C x) { return 0; }
int foo (D x) { return 1; }
C p = new D();
int i = foo(p); // executes C.foo
```

# Multimethods

- Multiple definition of the same name
- Method selection at run time

```
class D extends C { ... }
int foo (C x) { return 0; }
int foo (D x) { return 1; }
C p = new D();
int i = foo(p); // executes D.foo
```

# Assignment Rules

- `a: A = b: B` is allowed when
  - `B = A`
  - `B` can be converted to `A` using widening conversions (i.e., `B <: A`)
  - For primitives, sometimes a narrowing conversion is allowed

# Self and its Type

- Most OO languages have a (pseudo) variable **self** that refers to the current instance
  - In Java we have the reserved word **this**
- By the type theory, the type of **self** depends on the context of the code

```
class Base {int m() {… this …} }
class Sub extends Base {
      int x() { … m(); … } }
```

- When referring **m()** in **Sub**, **this** has the type of **Sub**
- However, in Java we have strictly static typing, where the type of **this** is defined by the strictly closed enclosing compile time context
  - thus, **m()** in **Sub**, **this** has the type of Base