Generics

Abstraction mechanism on type parameters

Polimorfism
==========
(categorization: James Coplien)
- Universal
  - Subtype/inclusion (Altípusos) Liskov-subtitutional-principle OOP
  - Parametric --> Generics used here
 (- raw (structural subtyping))
- Ad-hoc
  - name based (overloading)
  - type cast (coersion)

No Polimorfism: monomorphic type system
   (mostly: Pascal, C (but void*))

Generics (templates)
types and algoritms

Constrained generics (Sablon-szerzodés model)
we can set constraints against the type parameters
 - Ada "with"
 - Eiffel: subtyping, Java, C#: subtyping + interfaces
 - C++20: Concepts
Unconstrained generics
 - Duck typing (C++ before C++20)

examples:
  sort<T implements Comparable<T>>(vector<T> v)
  f<T extends ClassX>(T x)

  C++ before C++20:
    template <typename RandomAccessIterator>
    void sort(RandomAccessIterator begin, RandomAccessIterator end);
  C++ after C++20:
    template <typename It> requires RandomAccessIterator<It>
    void sort(It begin, It end);

When it happens?
==========

(Preprocessor time)
  C macros: does not co-operate with the type system
  Not Turing complete
  #define MAX(a,b) ((a)>(b)?(a):(b))

Compile-time
  Most of the modern programming languages
  Strong type checking

Run-time
  Python, Javascript, Lua

Implementation
==========

Type erasure (Java) no specialization
 All generics are mapped to some Base type, e.g. Java "Object"
 + Small size of generated code
 - No specialization

Instantiation (Ada, Eiffel, C++)
 + More flexible for specialization
 - Possible Code-blow

Ada: Generics: Explicit instantiation of generics to Packages
C++: implicit + explicit

Funcprog: ML, Haskell, Clean
 Haskell type classes

Generics + Inheritance
 Mixin
  C++: template <typename T>
      class Mixin : public T { ... };

 CRTP Couriously Recurring Template Pattern
      -> F-bounded polymorphism
      class Derived : public Base<Derived> { ... }
       static polymorphism, enable_shared_from_this, ...