ETH Zürich
Chair of Software Engineering

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

C#: exceptions and genericity

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Exceptions

Exceptions

Exceptions are objects

- They are all descendants of **`System.Exception`**
- Raise with a **`throw`** **`ExceptionObject`** instruction

```
throw new AnExceptionClass("ErrorInfo");
```

- In C#, all exceptions are "unchecked" (using Java terminology)
  - May be handled, if desired
  - If the current block does not have an exception handler, the call stack is searched backward for an exception handler
  - If none is found, the unhandled exception terminates the current execution thread

Exception handlers

The scope of the exception handler is denoted by a `try` block

Every `try` block is immediately followed by zero or more `catch` blocks, zero or one `finally` block, or both. At least one of `catch` blocks and `finally` block is required (otherwise, the `try` would be useless)

```
public int foo(int b) {
    try { if ( b > 3 ) {
            throw new System.Exception();
          }
        } catch (System.Exception e) { b++; }
          finally { b++; }
    return b;
}
```

Exception handlers: catch blocks

`catch` blocks can be exception-specific:

```
catch (ExceptionType name) { /* handler */ }
```

- Targets exceptions whose type conforms to `ExceptionType`
- `ExceptionType` must be a descendant of `System.Exception`
- `name` behaves as a local variable inside the handler block
- A `catch` block of type `T` cannot follow a `catch` block of type `S` if `T` ≤ `S` (otherwise the `T`-type block would be shadowed)
- From within a `catch` block the exception being handled can be re-thrown with `throw` (no arguments)

Exception handlers: catch/finally blocks

When an exception of type `T` is thrown within a `try` block:

- control is transferred to the first (in textual order) catch block whose type `T` conforms to, if one exists
- then, the control is then transferred to the `finally` block (if it exists)
- finally, execution continues after the `try` block

When no conforming `catch` exists or an exception is re-thrown inside the handler:

- After executing the `finally` block, the exception propagates to the next available enclosing handler

When a `try` block terminates without exceptions:

- the control is transferred to the `finally` block (if it exists)
- then, execution continues after the `try` block

Exception handlers: catch/finally blocks

A `finally` block is **always** executed after the `try` block even if no exceptions are thrown

- Typically used to free resources

Control-flow breaking instruction (`return` , `break` , `continue`) inside a `finally` block are restricted.

- `return` statements cannot occur in `finally` blocks

- `goto`, `break`, and `continue` statements can occur in `finally` blocks only if they do not transfer control outside the `finally` block itself

These restrictions disallow tricky cases that are allowed in Java

Invalid `finally` blocks

Valid Java code but invalid C# code:

```
public int foo() {
    try { return 1; }
    finally { return 2; }
}

public void foo() {
    int b = 1;
    while (true) {
        try { b++; throw new Exception(); }
        finally { b++; break; }
    } b++;
}
```

(Examples from **Martin Nordio**)

Exceptions vs. assertions (contracts)

Exceptions and assertions have partially overlapping purposes: dealing with "special" behaviors

- invalid input
- errors in computations
- runtime failures (e.g., I/O or network errors)
- ...

Exceptions vs. assertions

The following guidelines are useful to choose when to use exceptions rather than assertions:

- exceptions define the actions to be taken in case of exceptional behavior, to restore a normal behavior
  - they define a “special” behavior that requires special handling
  - an exception occurring is a possible, if unusual, behavior
  - exceptions may occur even in correct programs
- assertions constitute a specification of what the implementation should achieve
  - they define a contract
  - an assertion violation is always an implementation error
  - if the program is correct, checking assertions should be completely useless

Exceptions vs. assertions: examples

A **`BankAccount`** class defines a public method **`FracBonus`** to add a fractional bonus to the **`Balance`**:

```
void FracBonus(int frac)
    // add 1/frac to Balance
```

Valid inputs: **`frac > 0`**

Exception or assertion?

Exceptions vs. assertions: examples

A **`BankAccount`** class defines a public method **`FracBonus`** to add a fractional bonus to the **`Balance`**:

```
void FracBonus(int frac)
    // add 1/frac to Balance
```

Valid inputs: **`frac > 0`**

Exception or assertion?

assertion:
this is a requirement imposed on clients of the method

Exceptions vs. assertions: examples

Using exceptions:

- In class **BankAccount**:

```
void FracBonus(int frac) {
    if (frac <= 0)
        throw new Exception("Wrong input");
    Bonus = Bonus * 1/frac;
}
```

- In clients of **BankAccount**:

```
BankAccount ba;
int x;
// ...
try { ba.FracBonus(x) }
    catch (Exception e) {
        if (e.Message == "Wrong input") {
            x = -x + 1; ba.FracBonus(x);
    } }
```

Exceptions vs. assertions: examples

Using assertions:

- In class **BankAccount**:

```
void FracBonus(int frac) {
    Assert(frac > 0);
    Bonus = Bonus * 1/frac;
}
```

- In clients of **BankAccount**:

```
BankAccount ba;
int x;
// ...
if (!(x > 0)) { x = -x + 1; }
ba.FracBonus(x);
```

Exceptions vs. assertions: examples

A `BankAccount` class defines a public method `LoadBalance` to read a new value of `Balance` from file:

```
void LoadBalance(String fileName);
     // read a new value
     // of Balance from fileName
```

Valid inputs:

- `fileName` is the name of an existing file
- the file can be opened correctly
- the content reads as an integer
- ...

Exception or assertion?

Exceptions vs. assertions: examples

A `BankAccount` class defines a public method `LoadBalance` to read a new value of `Balance` from file:

Valid inputs:

- `fileName` is the name of an existing file
- the file can be opened correctly
- the content reads as an integer
- ...

Exception or assertion?

- exception:
  an invalid input is a runtime error that requires extra measures but doesn't depend on the implementation being incorrect

Exceptions vs. assertions: examples

Using assertions:

- In class **BankAccount**:

```
void LoadBalance(string fileName) {
  Assert(fileName != null);
  Assert(fileName != "");
  Assert(System.IO.File.Exists(fileName));
  TextReader tr = new StreamReader(fileName);
  int result;
  bool ok = Int32.TryParse(tr.ReadLine(),out result);
  Assert(ok);
  return result;      }
```

- In clients of **BankAccount**:

```
BankAccount ba; string fn;
// read file name from user into fn
// redo the checks and notify user if they go wrong
if (fn == "") {
    Console.WriteLine("Invalid filename"); }
// ...
```

Exceptions vs. assertions: examples

Using exceptions:

- In class **BankAccount**:

```
void LoadBalance(string fileName) {
    TextReader tr = new StreamReader(fileName);
    return Convert.ToInt32(tr.ReadLine());
}
```

- In clients of **BankAccount**:

```
BankAccount ba;
string fn;
// read file name from user into fn
// catch assertions and notify user accordingly
try { ba.LoadBalance(fn) }
    catch (ArgumentException e) {
          Console.WriteLine("Invalid filename"); }
// ...
```

Java and C# in depth

Carlo A. Furia, Marco Piccioni, Bertrand Meyer

Genericity in C#

Generics

C#’s genericity mechanism, available since C# 2.0

Most common use:

- Use (and implement) generic type-safe containers

```
List<String> safeBox = new List<String>();
```

- Compile-time type-checking is enforced

More sophisticated uses:

- Custom generic classes and methods
- Bounded genericity

```
public T test <T> (T x) where T:Interface1, Interface2
```

Generic classes

A generic class is a class parameterized w.r.t. one or more generic types.

```
 public <T> class Cell {
    public T Val { get; set;}
}
```

To instantiate a generic class we must provide an actual type for the generic parameters.

```
Cell<String> c = new Cell<String>();
```

Generic classes

The generic parameters of a generic class may constrain the valid actual types.

```
public class Cell<T> where T:S { ... }
```

The following is valid only if `X` is a subtype of `S`:

```
Cell<X> c = new Cell<X>();
```

The constrains may involve multiple types.

```
public class C<T> where T: A, IB
```

The following is valid only if `Y` is a subtype of both `A` and `IB`:

```
C<Y> c = new C<Y>();
```

Type inference: implicit types

When creating an instance of a generic class, the compiler is often able to infer the generic type from the context. In such cases, we can omit the type and use `var` instead.

```
var c = new Cell<String>();
```

is equivalent to:

```
Cell<String> c = new Cell<String>();
```

In general, `var` can be used for every variable declaration where the compiler can figure out the types.

```
var x = new String[12];
var y = 12;
var z = 12.4 + 5 + "OK"; // String "17.4OK"
```

Generic methods

A generic method is a method parameterized w.r.t. one or more generic types.

```
public U downcast <U> (T x) where U:T {
    return (U) x;
}
```

Notice the different position of the generic parameter:

- C#: `public T foo <T> (T x);`
- Java: `public <T> T foo (T x);`

Generic methods

Clients must provide actual types for the generic parameters only when the compiler cannot infer them from context.

```
public U downcast <U> (T x) where U:T

Person p = new Person();

Employee e = downcast(p); // error: which type
  // among all subtypes of Employee?
Employee e = downcast<Employee>(p);  // OK
var e = downcast<Employee>(p);  // OK

public static void a2c <G> (G[] a, IList<G> c)
a2c(new String[8], new List<String>()); // OK
```

Generics: features and limitations

Unlike Java, genericity is supported natively by .NET bytecode

Hence, basically all limitations of Java generics disappear:

- Can instantiate generic parameter with value types
- At runtime you can tell the difference between `List<Integer>` and `List<String>`
- Exception classes can be generic classes
- Can instantiate a generic type parameter
  - provided a clause `where T : new()` constrains the parameter to have a default constructor
- Can get the default value of a generic type parameter
  `T t = default (T);`
- Arrays with elements of a generic type parameter can be instantiated
- A static member can reference a generic type parameter

- Another consequence is that raw types (unchecked generic types without any type argument) don't exist in C#

Generics and inheritance

- Let S be a subtype of T (i.e. S ≤ T)

In general, there is no inheritance relation between:

`SomeGenericClass<S>` and `SomeGenericClass<T>`

In particular: the former is not a subtype of the latter

However, let AClass be a non-generic type:

- `S<AClass>` is a subtype of `T<AClass>`

There's no C# equivalent of Java's wildcards, but C#'s full-fledged genericity mechanisms normally provide alternative ways to achieve the same designs

However, C# doesn't have lower-bounded genericity

Why subtyping with generics is tricky

Consider a method of class **F**:

```
public static void foo(List<Vehicle> x){
      // add a Truck to the end of list 'x'
      x.Add(new Truck());
}
```

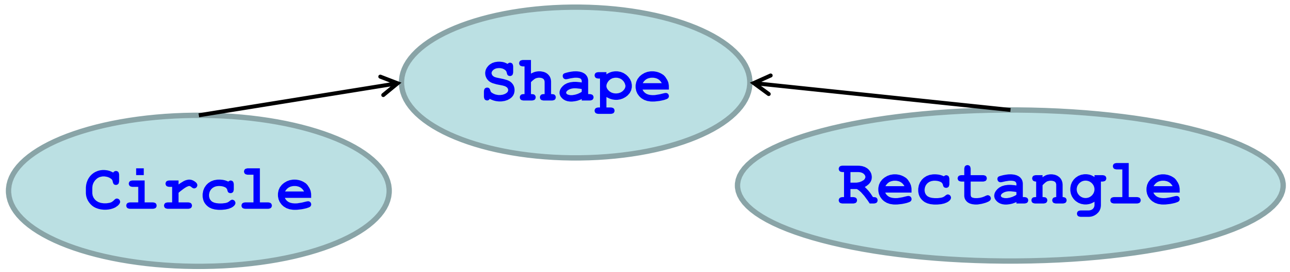
If **List<Car>** were a subtype of **List<Vehicle>**, this would be valid code:

```
var cars = new List<Car>();
cars.Add(new Car());
F.foo(cars);
```

But now a **List<Car>** would contain a **Truck**, which is not a **Car**!

Replacing wildcards in C#: example

Consider the following hierarchy of classes:



What should be the signature of a method **drawShapes** that takes a list of **Shape** objects and draws all of them?

- **DrawShapes(List<Shape> shapes)**
  - this doesn’t work on a **List<Circle>**, which is not a subtype of **List<Shape>**

Replacing wildcards in C#: example

What should be the signature of a method **drawShapes** that takes a list of **Shape** objects and draws all of them?

First solution: use a helper class with bounded genericity

```
class DrawHelper <T> where T: Shape {
    public static void DrawShapes( List<T> shapes)
}
```

Client usage:

```
DrawHelper<Shape>.DrawShapes(listOfShapes);
DrawHelper<Circle>.DrawShapes(listOfCircles);
```

The compiler may be able to infer the generic type argument from context.

Replacing wildcards in C#: example

What should be the signature of a method **drawShapes** that takes a list of **Shape** objects and draws all of them?

Second solution: use a generic method inside **Shape**

```
public static void DrawShapes <T> (List<T> shapes)
    where T:Shape
```

Client usage:

```
Shape.DrawShapes<Shape>(listOfShapes);
Shape.DrawShapes<Circle>(listOfCircles);
```

The compiler may be able to infer the generic type argument from context.

Replacing wildcards in C#: example

What should be the signature of a method **`drawShapes`** that takes a list of **`Shape`** objects and draws all of them?

Third solution: use an **`out`** generic parameter, which declares that objects of generic type will only be read (and hence passing a collection of a subtype is type safe). Typically done using the **`IEnumerable<out T>`** interface.

```
public static void DrawShapes
                        (IEnumerable<Vehicle> shapes)
```

Client usage:

```
DrawShapes(listOfShapes);
DrawShapes(listOfCircles);
```

Conversion from **`List`** to **`IEnumerable`** is implicit, but the signature guarantees that **`DrawShapes`** only reads the list while iterating.

Covariant generic parameters

If **`S`** is subtype of **`T`** (i.e. **`S`** ≤ **`T`**) and generic interface **`I`** is declared as covariant: **`IC<out G>`**, then:

`IC<S>` is a subtype of **`IC<T>`**

That is: instances of classes implementing **`IC<S>`** can be attached to references of type **`IC<T>`**

Covariant **`out`** generic parameters have restrictions that conservatively ensure type safety:

- they can only be used in interfaces and delegates
- they can only be use as return types (not as argument type)
- they cannot be used as genericity constraint

Contravariant generic parameters

Consider a method `SameArea` that takes a list of `Circles` and counts how many have the same area as a given `Circle`:

```
public static int SameArea
      (IEnumerable<Circle> clist, Circle c,
       IEqualityComparer<Circle> cmp)
```

A comparator of areas of generic shapes should also work to compare the area of circles. In fact, the `IEqualityComparer<in T>` interface allows us to pass a comparator for a supertype of `Circle`.

Client usage:

```
IEqualityComparer<Shape> shapeComparer = ...
SameArea(listOfCircles, circle, shapeComparer);
```

Contravariant generic parameters

If `S` is subtype of `T` (i.e. `S` ≤ `T`) and generic interface `I` is declared as contravariant: `IC<in G>`, then:

`IC<T>` is a subtype of `IC<S>`

That is: instances of classes implementing `IC<T>` can be attached to references of type `IC<S>`

Contravariant `in` generic parameters have restrictions that conservatively ensure type safety:

- they can only be used in interfaces and delegates
- they can only be used as argument type (not as return types), and not for `out` or `ref` arguments

(Contravariant genericity may be unintuitive to use in general.)

Collections

A classic example of separating interface from implementation

Some library interfaces from `Systems.Collections.Generic`:

- `ICollection<E>`
  - `int Count;`
    - number of elements in the collection
  - `void add(E item)`
  - `bool remove(E item)`
    - returns whether the collection actually changed
  - `IEnumerator<E> GetEnumerator()`
- `IEnumerator<E>`
  - `bool MoveNext()`
    - Moves to the next element; returns `false` if the enumerator has passed the end of the collection
  - `E Current`
    - returns the current element in the enumeration

Collections: some implementations

- List: indexed, dynamically growing
- LinkedList: doubly-linked list
- HashSet: unordered, rejects duplicates
- TreeSet: ordered, rejects duplicates
- Dictionary: key/value associations
- SortedDictionary: key/value associations, sorted keys