Enterprise Application Systems (EAS)

CSharp Fundamentals - Generics

Daniel Fink

Institute for Computational Physics

October 19, 2022

What is the Problem?

```
// Bob wants to buy 10 apples in an online shop
```

What is the Problem?

```
// Bob wants to buy 10 apples in an online shop
Apple[] shoppingCard = new Apple[10];
```

What is the Problem?

```
// Bob wants to buy 10 apples in an online shop
Apple[] shoppingCard = new Apple[10];

// ...

// Now, Bob also wants to buy 5 bananas
```

What is the Problem?

```
// Bob wants to buy 10 apples in an online shop
Apple[] shoppingCard = new Apple[10];

// ...

// Now, Bob also wants to buy 5 bananas
Banana[] shoppingCard2 = new Banana[5];
```

What is the Problem?

```
// Bob wants to buy 10 apples in an online shop
Apple[] shoppingCard = new Apple[10];

// ...

// Now, Bob also wants to buy 5 bananas
Banana[] shoppingCard2 = new Banana[5];

// ...

// Yet, Bob find out he is broke, he can only buy 2 apples
```

What is the Problem?

```
// Bob wants to buy 10 apples in an online shop
Apple[] shoppingCard = new Apple[10];

// ...

// Now, Bob also wants to buy 5 bananas
Banana[] shoppingCard2 = new Banana[5];

// ...

// Yet, Bob finds out he is broke, he can only buy 2 apples
shoppingCard = new Apple[2];
```

What is the Problem?

```
// Bob wants to buy 10 apples in an online shop
Apple[] shoppingCard = new Apple[10];

// ...

// Now, Bob also wants to buy 5 bananas
Banana[] shoppingCard2 = new Banana[5];

// ...

// Yet, Bob finds out he is broke, he can only buy 2 apples
shoppingCard = new Apple[2];
```

- We create a new (fixed-sized) array every time a change is made.
- We need to handle multiple shopping cards - one per item type.
- Obviously, we can/should not do this in a real-world application.

What is the Solution?

- Use (dynamically-sized) lists:

```
// Bob wants to buy 10 apples in an online shop
List<Apple> shoppingCard = new List<Apple>();
for (int i = 0; i < 10; i++)
{
    shoppingCard.Add(new Apple());
}
```

What is the Solution?

- Use (dynamically-sized) lists:

```
// Bob wants to buy 10 apples in an online shop
List<Apple> shoppingCard = new List<Apple>();
for (int i = 0; i < 10; i++)
{
    shoppingCard.Add(new Apple());
}

// ...

// Yet, Bob finds out he is broke, he can only buy 2 apples
shoppingCard.Remove(index: 0, count: 8);
```

What is the Solution?

- Use (dynamically-sized) lists:

```
// Bob wants to buy 10 apples in an online shop
List<Apple> shoppingCard = new List<Apple>();
for (int i = 0; i < 10; i++)
{
    shoppingCard.Add(new Apple());
}

// ...

// Yet, Bob finds out he is broke, he can only buy 2 apples
shoppingCard.Remove(index: 0, count: 8);
```

- How about the bananas?

What is the Solution?

- Create a *generalized* object "Item":

```
class Item { ... }
class Apple : Item { ... }
class Banana: Item { ... }

// ...

List<Item> shoppingCard = new List<Item>();

shoppingCard.Add(new Apple());
shoppingCard.Add(new Banana());
```

With generics you...

- can implement logic for “to-be-specified-later” types.
  - Even for types you are not considering at the moment.
- generally need less code.
- mostly write faster code.
- are more flexible compared to only static types (C++).
- code type-safe compared to interpreters (Python).

Comparison to C++ templates and Python

- C++ templates are resolved at *compile-time*.
- CSharp generic types are resolved at *run-time*.
- Python only knows *run-time* anyway.

Comparison to C++ templates and Python

```
// CSharp

// The function signature contains a "placeholder" for the type
float CalculatePrice<TItem>(List<TItem> shoppingCard)
{
    float totalPrice = 0.0;

    foreach (TItem item in shopping_card)
    {
        totalPrice += item.price;
    }

    return totalPrice;
}
```

- This won't compile since TItem might not have a "price" property.

Comparison to C++ templates and Python

```
// CSharp

// The "placeholder" must be of type Item
float CalculatePrice<TItem>(List<TItem> shoppingCard) where TItem : Item
{
    float totalPrice = 0.0;

    foreach (TItem item in shopping_card)
    {
        totalPrice += item.price;
    }

    return totalPrice;
}
```

- This will compile since we ensure TItem has a “price” property.

Comparison to C++ templates and Python

```cpp
// C++

template <class TItem>
double CalculatePrice(std::list<TItem> shoppingCard)
{
    float totalPrice = 0.0;

    for (auto it = shoppingCard.begin(); it != shoppingCard.end(); ++it){
        totalPrice += it->price;
    }

    return totalPrice;
}
```

- This will compile as long as all calls to CalculatePrice() put in a list of items that have the price property.

Comparison to C++ templates and Python

```
def calculate_price(shopping_card):
    total_price = 0.0

    for item in shopping_card:
        total_price += item.price

    return total_price
```

- This would throw an AttributeError if the property price does not exist.

Summary

- Generic types enrich a strongly-typed language with great flexibility.
- The type is resolved at run-time and thus used type properties/methods must be known at compile-time.
- It can be seen as somewhere in-between Python and C++ in terms of flexibility and performance.

Outlook

- EAS libraries make heavy use of generics, e.g., middlewares.
- The software architecture can be defined and enforced via generics (and interfaces), e.g., pre-define services that only allow certain types to be used.
- Software patterns and principles can be used more restrictively depending on the context, e.g., dependency injection, mediator, reflection, ...