Design Patterns

- The Strategy Pattern
- The Factory Method
- **Generics**
- The Abstract Factory Pattern
- The State Pattern
- The Observer Pattern
- The Adapter Pattern
- The Composite Pattern
- The Iterator Pattern
- The Builder Pattern
- Fallen Patterns
  - The Singleton Pattern
  - The Visitor Pattern

Generics (Java) / Templates (C++)

- Generics and templates allow you to write code that works on different types without requiring inheritance
- You've seen this before:
  - ArrayList<type> in Java
  - std::vector<type> in C++
- Writing a generic class is like writing a regular class, except you use a "fake" type which is filled in when the generic is used
- "Generic" is roughly equivalent to "Template that isn't as powerful"
- C++ templates are turing-complete, Java generics are not
  - Turing complete: Can simulate any Turing machine

A Generic Class

```
public class Matrix<T extends Number> {
    private T[] cells;

    public void Add( Matrix<T> o ) {…}
    public void Mult( Matrix<T> o ) {…}
    public void Scale( T value ) {…}
    public T get( int i, int j ) {
        return cells[i];
    }
}
```

Generic Class Explanation

- "T" is the generic type variable
- When people use this class they will substitute a real type for T:
  - Matrix<Float> or Matrix<BigDecimal>
- "T" is an arbitrary name
  - It could be called "M" or "CoolType"
- In order to know how T can be used, we say that it "Extends" Number
  - The Number class contains methods to convert to specific types of Number

What's the Point?

- Why not just have `Matrix` be an abstract class?

- Generics are:
  - More convenient (don't need to subclass)
  - More type-safe (don't need to down-cast)
  - More performant (in some languages, not in Java)

- Consider `comparable` with and without generics

Generic Factories

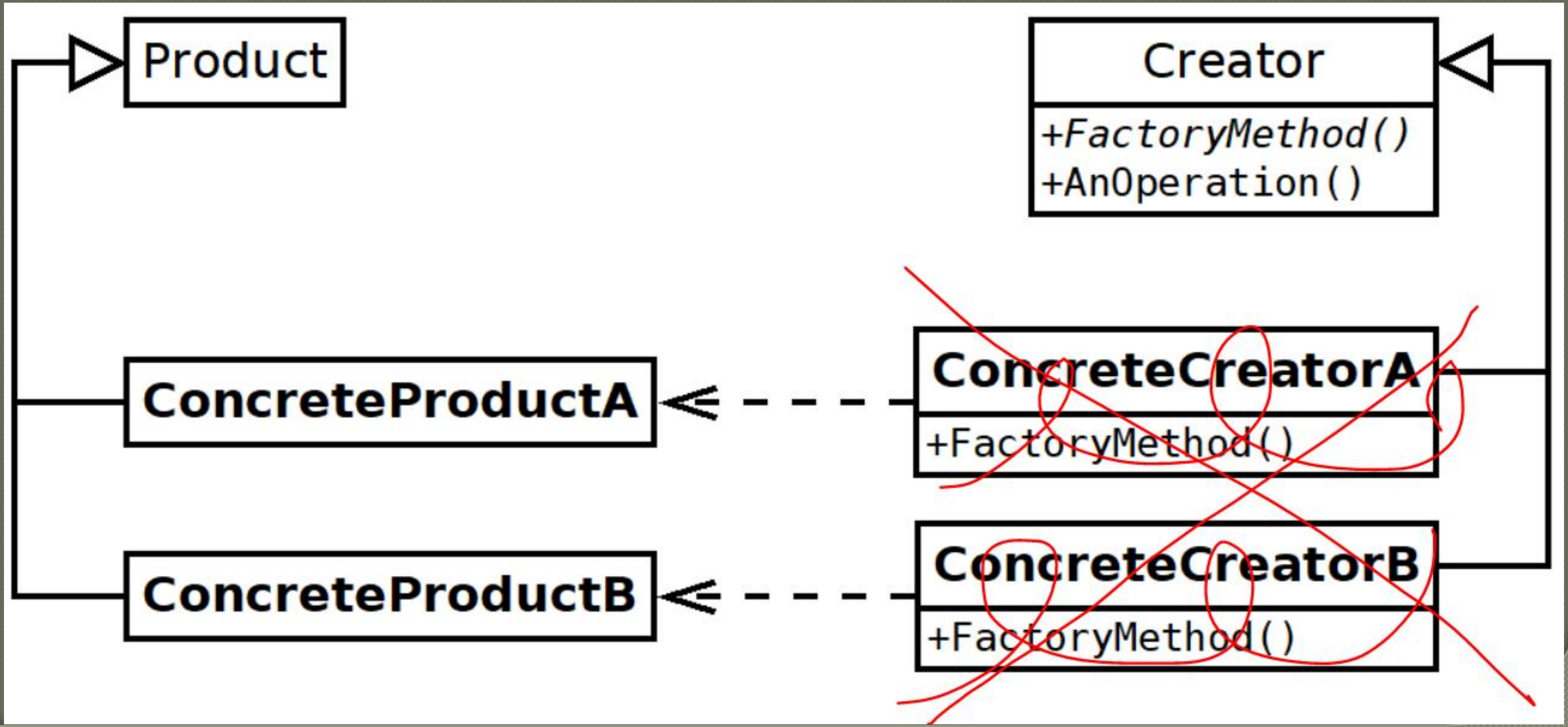
Why are these generic types useful?

Many patterns can be extended through the use of generics

Factories and strategies are two of them

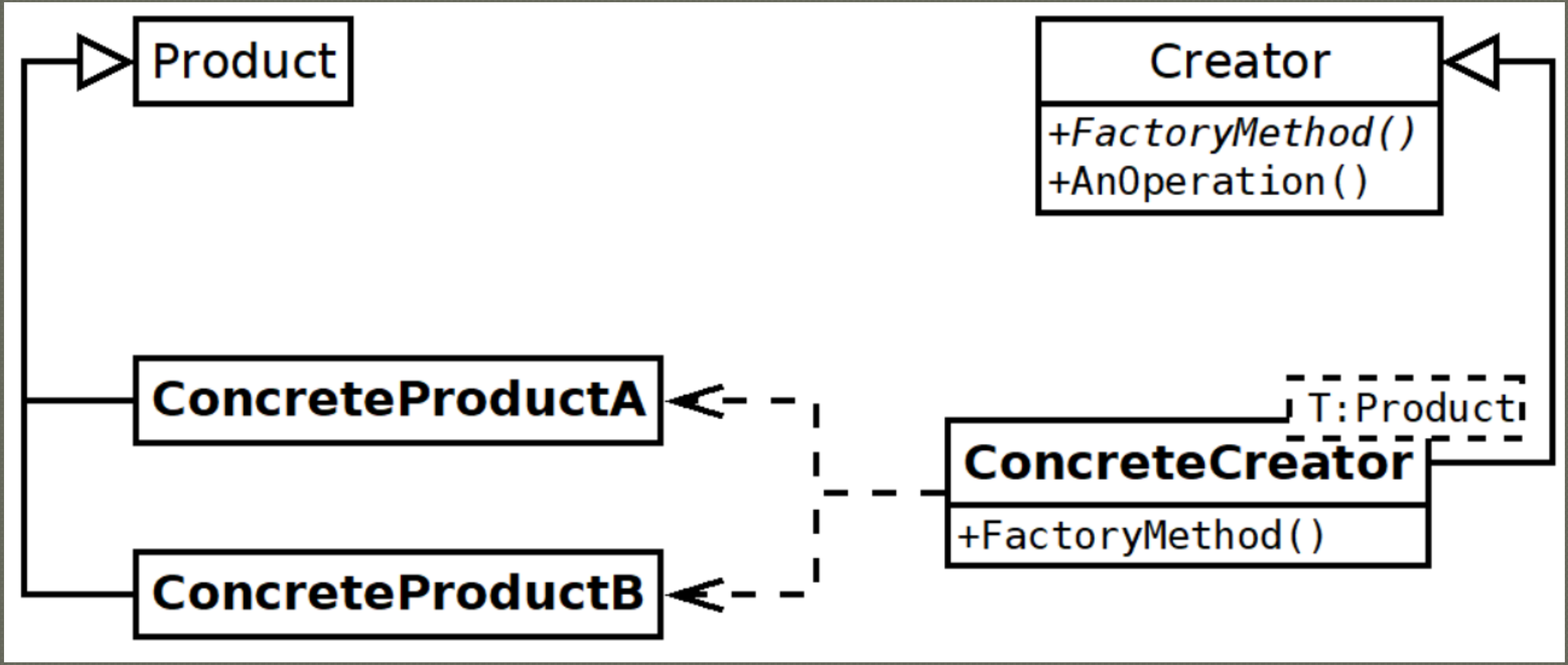
Generic Factories

- In theory, we can remove concrete factories



Generic Factories

- The UML would look like this



Is it this easy?

```
class SpecificCreator<T extends Product>
                         extends Creator {
    Product createProduct() {
        return new T();
    }
}
```

NO!

Java doesn’t allow constructors to be called on generic types

This is because constructors are never inherited

Java therefore doesn’t know anything about the constructor (number of arguments, types, etc.)

Solution: Lambdas

- Lambdas offer a solution
- Lambdas are anonymous functions
- We can pass a constructor as a lambda
- If you don’t know about lambdas, you should learn them:
  - https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- Most languages support them now

Tradeoffs

- Good:
  - Don’t need to add new factories when we add new products
- Bad:
  - Rarely is the complexity worth it in Java. (in other languages it works great)
- Remember YAGNI: You Ain’t Gonna Need It
  - Only refactor to a generic factory if you’re sure you’ll need a group of factories with trivial constructors

C++ Template Factory: it works (unlike Java)

```
template <class ConcreteProduct>
class TemplateCreator: public Creator {
public:
    virtual Product* CreateProduct();
};

template <class TheProduct>
Product* TemplateCreator<TheProduct>::
CreateProduct () {
    return new TheProduct;
}
```

Duck Typing Languages

- Duck-typing languages do not support generics, because they don’t need them

- “Factories” can just be free functions

- Remember, if you are using a function as a factory method, you need to annotate it in UML as such
    - Same as the Strategy pattern

C++ Duck Typing

- Templates in C++ actually support a form of duck typing
- The compiler checks that methods exist at compile time
- This enables extreme flexibility
- The downside: horrific error messages:
  - https://codegolf.stackexchange.com/questions/1956/generate-the-longest-error-message-in-c