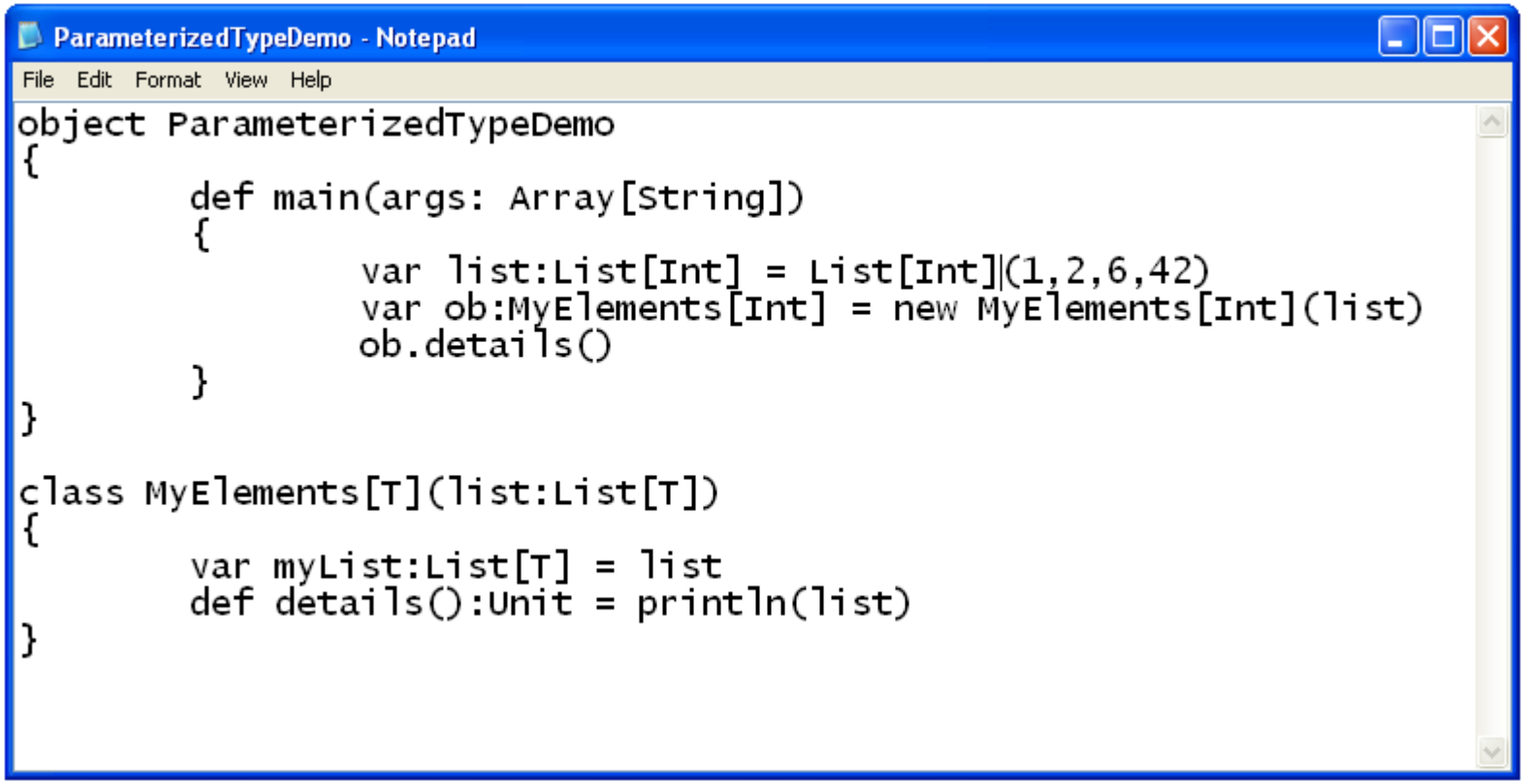
Type Parameterization

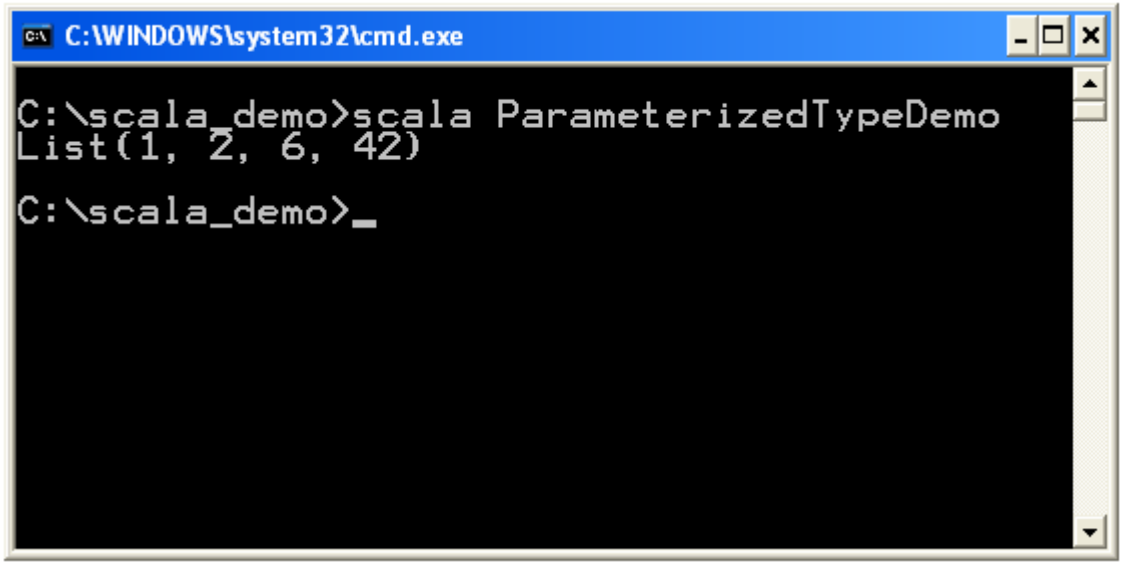
Introduction

- Using the type parameterization capability we can define generic classes and generic traits.
- We denote the parameterized type using square brackets.

Code Sample



Code Sample



Generic Functions

- Functions can have type parameters. Each time we call such function we should specify the missing type\s.

Generic Functions

```
object HelloSample
{
  def main(args:Array[String]):Unit =
  {
    val ob = getMyStack[Int](10)
  }
  def getMyStack[T](num:Int) =
  {
    new MyStack[T](num)
  }
}

class MyStack[T](size:Int)
{
  //...
}
```

Type Bounds

- When we define a function or a class and we choose to use generics we can set limits on the type.

  `S <: T` - means that S is a sub type of T

  `S >: T` - means that S is a super type of T

- It is also possible to mix between a lower bound and an upper one.

  `S >: T1 <:T2`

  S must be a super type for T1 and a sub type for T2.

Type Bounds

```
object HelloSample
{
  def main(args:Array[String]):Unit =
  {
    val ob = getMyStack[SportCar](10)
  }
  def getMyStack[T >: RacingCar <: Car](num:Int) =
  {
    new MyStack[T](num)
  }
}

class MyStack[T](size:Int)

class Car

class SportCar extends Car

class RacingCar extends SportCar
```

Covariance

- Covariance exists when two generic classes maintain the same relationship as the one between the types they are using.
- The Generic classes in Scala are not covariant with each other.

```
val a:Array[SportCar] = new Array[SportCar](10);
val b:Array[Car] = a; //compilation error
```

Generics Wildcard

- We can use '_' (AKA: 'wild card') instead of specifying the exact type when creating variables that should be assigned with a reference for a Generic type object.

Generics Wildcard

```
object HelloSample
{
  def main(args:Array[String]):Unit =
  {
    val a:MyStack[SportCar] = new MyStack[SportCar];
    val b:MyStack[_ <: Car] = a;
  }
}

class Car

class SportCar extends Car

class MyStack[T]
```