

# Generics

## 1 Why do we need generics?

Think of a data structure that holds several items. For instance, a Bookcase may have 3 shelves. Each shelf can hold one object.

```
public class Bookcase {
    private Object topshelf;
    private Object middleshelf;
    private Object bottomshelf;

    public void addtop(Object item) { topshelf = item; }
    public void makemiddle(Object item) { middleshelf = item; }
    public void addbottom(Object item) { bottomshelf = item; }

    public Object gettop() { return topshelf; }
    public Object getmiddle() { return middleshelf; }
    public Object getbottom() { return bottomshelf; }
}
```

This class gives the programmer the flexibility to mix different data types in the same Bookcase or re-use the same code for a Book-filled Bookcase and a Toy-filled Bookcase. The disadvantage is that if you want just one data type in the list, you won't get any help from Java to enforce this. For example, if you wanted to insert only Book objects in the Bookcase, there is nothing to prevent other objects such as Shoes from being added to the Bookcase if it is declared to hold Objects. Additionally, you have to make a cast every time you take an item out of the Bookcase.

```
Bookcase shelves = new Bookcase();
shelves.addtop(new Book("Harry Potter"));
shelves.addbottom(new SmellySock()); // well, nothing stopped you...
Book book = (Book) shelves.gettop(); // and the cast is annoying
```

One possible solution to this problem is to declare the contents to be of type Book. This will force the Bookcase to hold only Book objects. The disadvantage is that you need a different implementation for each type of Bookcase that you want. You might also want to store Photo instances on a Bookcase.

This is where Java's generics can be extremely useful. Java's generics allow the programmer to define one data structure to be used for different data types in different applications.

Generics involve a bit of extra work when declaring or initializing a variable but remove all the annoying casts when using the generic variable later on. They also add some safety and allow Java to catch a few more code errors you might make.

## 2 How to make a class generic

This can be done in several easy steps

### 2.1 Make the overall class generic

Pick your favorite letter, I'll pick T. Modify the class declaration so it changes from

```
public class Bookcase { ... }
```

to

```
public class Bookcase<T> { ... }
```

This tells Java, the Bookcase will hold only Ts, whatever T may be. The T defined here is called a *type parameter* for the class. You can make both interfaces and classes generic this way.

```
public interface Shelf<T> { ... }  
public class Bookcase<T> implements Shelf<T> { ... }
```

If you need multiple different generic types (for instance, if you want to store different kinds of objects on each shelf), you just declare a bunch of type parameters.

```
public class Bookcase<S, T, U> { ... }
```

## 2.2 Make all the contents generic

You will now replace the contents (Object) with T. To do this, find all occurrences of Object and make them use T instead.

```
public void addtop(T item) { topshelf = item; }  
public void addmiddle(T item) { middleshelf = item; }  
public void addbottom(T item) { bottomshelf = item; }  
  
public T gettop() { return topshelf; }  
public T getmiddle() { return middleshelf; }  
public T getbottom() { return bottomshelf; }
```

## 3 How to use a generic type

To make and use a non-generic Bookcase of Books, you originally used

```
Bookcase shelves = new Bookcase();  
shelves.addtop(new Book("Harry Potter"));  
Book book = (Book) shelves.gettop();
```

Now, just like with methods, you need to provide a parameter when you want to use the generic class. In our example, you declare the Bookcase to hold only Books and you can drop the cast when you retrieve a Book.

```
Bookcase<Book> shelves = new Bookcase<Book>(); // notice we used <Book> twice  
shelves.addtop(new Book("Harry Potter"));  
Book book = shelves.gettop(); // no more casts!
```

A fully qualified generic type (like Bookcase<Book>) is called an *instantiated* type. When you want to use a generic type, you need to use the same instantiated type on the declared variable as you do in the new expression. If you do not use a type parameter, like in our original case, the Java compiler will warn you that you are using a raw type.

## 4 How to use a generic type inside another generic type

Generic classes frequently contain uses of other generic classes. For example, Russian babushka dolls are wooden painted dolls that contain smaller and smaller dolls. If the dolls also contained an object, a class for these might look like

```
public class Babushka {  
    private Object contents;  
    private Babushka next;  
}
```

The generic version of this class adds <T> to the class declaration and replaces Object with T but also uses the next field generically.

```
public class Babushka<T> {
    private T contents;
    private Babushka<T> next; // generic usage of a Babushka
}
```

Similarly, a method which takes and returns a Babushka now uses the generic <T> so Babushka methods

```
public void insert(Babushka doll) { next = doll; }
public Babushka open() { return next; }
```

become the generic methods

```
public void insert(Babushka<T> doll) { next = doll; }
public Babushka<T> open() { return next; }
```

## 5 How to use an array of generic types

When creating an array of generic data types, you still have to call the array constructor using Object but you also have to cast to your generic type (T).

```
Object[] data = new Object[size]; // non-generic array
T[] data = (T[]) new Object[size]; // generic array
```

A common error with arrays is to call the constructor using your generic type (T).

```
T[] data = (T[]) new T[size]; // This is a common error!
```

From this point, you can use the array as normal. You should not need to cast anything when retrieving it from the array.

## 6 Bounded Generic Types

While it is nice to have Bookcase instances that only store books, it might be even nicer to require that all Bookcase instances only store books. Our current implementation, however, allows you to create a Bookcase that stores Socks.

```
Bookcase<Book> bookBookcase = new Bookcase<Book>(); // allowed! yay!
Bookcase<Sock> sockBookcase = new Bookcase<Sock>(); // also allowed! boo!
```

As a practical reason why we might want to prevent creating Bookcases of Socks, imagine that we wanted to add a new method to Bookcase that printed out all of the titles and authors of Books on the shelves. This makes sense for a Book object, which has the methods getTitle and getAuthor, but would be an error for a Sock, which does not have these methods.

```
public class Bookcase<T> {
    ...
    public void browseShelves() { // errors on getTitle() and getAuthor()!
        System.out.println("Top Shelf:");
        System.out.println(topshelf.getTitle() + " by " + topshelf.getAuthor());
        System.out.println("Middle Shelf:");
        System.out.println(middleshef.getTitle() + " by " + middleshef.getAuthor());
        System.out.println("Bottom Shelf:");
        System.out.println(bottomshelf.getTitle() + " by " + bottomshelf.getAuthor());
    }
}
```

To fix this issue, you can use a *bounded* type parameter for your class! In this example, that would mean you only want to accept types that can be used as Books. Modify your class header so that it changes from

```
public class Bookcase<T> {
```

to

```
public class Bookcase<T extends Book> {
```

You do not have to change anything else about the class to get this to work! All of the places you used T before behave exactly the same and the issue with `browseShelves` is fixed! As a result, you cannot create a `Bookcase` of `Socks` anymore.

```
Bookcase<Book> bookBookcase = new Bookcase<Book>(); // still allowed! yay!  
Bookcase<Sock> sockBookcase = new Bookcase<Sock>(); // not allowed! yay!
```

One thing to remember is that using a type parameter `<T>` is the same as using the bounded type parameter `<T extends Object>`.

## 7 Wildcards

Occasionally you may want to use generics to enable you to use the same code for all instances of a generic classes. Because of the way types work in Java, it is not safe to just cast an instantiated type with a particular parameter to an instantiated type with the parameter's superclass as its parameter. For example, imagine the `Book` class has two subclasses: `PaperbackBook` and `HardcoverBook`.

```
Bookcase<PaperbackBook> myBookcase = new Bookcase<PaperbackBook>();  
Bookcase<Book> myCastBookcase = (Bookcase<Book>) myBookcase; // not allowed!  
myCastBookcase.addtop(new HardcoverBook()); // this is why it is not allowed!
```

In the above example, if it were allowed to cast to the instantiated type with the superclass of the parameter, you could store a `HardcoverBook` on the top shelf of a `PaperbackBook` bookcase! This is clearly not the correct behavior.

But, what if you wanted to be able to write some code that worked for any kind of `Bookcase`? You cannot just cast because of the previous problem. For this purpose, Java provides some syntax called the *wildcard*, and denoted by `'?'`. You can use a wildcard instead of an actual type parameter.

```
public void browseBookcases(Collection<Bookcase<?>> c) {  
    for(Bookcase<?> b : c) {  
        b.browseShelves();  
    }  
}
```

In this example, you can still invoke methods like `browseShelves` on each of the `Bookcase` objects, but you do not have any information about what is actually in them. As a result, you would not be allowed to use these references to add any new `Books`.

```
Bookcase<PaperbackBook> myBookcase = new Bookcase<PaperbackBook>();  
Bookcase<?> myCastBookcase = (Bookcase<?>) myBookcase; // allowed!  
myCastBookcase.addtop(new HardcoverBook()); // not allowed!
```

Wildcards are among the most advanced, and least understood, of Java's features. It is easy to get yourself into a corner where the compiler will not let you do anything with wildcards, so use them sparingly.