Generics

* The basic purpose behind using generics is to enable you to mark your intent of using a class, method, or interface with a particular data type. Generics add compile-time safety to collections.
* ***Removing explicit casts***—prior to generics, you needed to add casts when you had a list with strings and you wanted to get a string out of the list. With generics this isn’t needed anymore.
* ***Developing generic algorithms***—just as you need not hard-code values when you work with methods and can accept them as method parameters, generics help you parameterize over data types and develop algorithms that work with multiple data types.
* A type parameter can be used in the declaration of classes, variables, method parameters, and method return types.

**Generic classes**

**Extending classes**

* When **generic class is extending another generic class** a type *argument* must be passed to the *type parameter* of a base class. You can do so while extending the base class or while instantiating the derived class.

class Parcel<**T**> {}

class GenericBookParcel<**X**> extends Parcel<**T**> {}

* **When nongeneric class is extending a generic class it does not define any type parameter**, but only passes type parameters to generic base class.

class Parcel<T>{}

class NonGenericPhoneParcel extends Parcel<Phone> {}

NonGenericPhoneParcel<String> v = new NonGenericPhoneParcel<>(); -- won’t compile

**Extending Interface**

interface MyMap<K, V>{

void put(K key, V value);

V get(K key);

}

* When a **nongeneric class implements a generic interface, the type parameters don’t follow the class name**. For the implemented interface, the type parameters are replaced by actual types:

class **MapLegendNonGeneric** implements **MyMap<String, Integer>** {

public void put(String s, Integer i) {}

public Integer get(String s) { return null; }

}

* When a generic class implements a generic interface, the type parameter must follow the class name. For the implemented interface, the type parameters are replaced by actual types:

class **MapLegendGeneric** implements **MyMap<String, Integer>** {

public void put(String s, Integer i) {}

public Integer get(String s) { return null; }

}

* Combinations of parameters are also allowed:

class MapLegendGeneric2<**V**> implements MyMap<**String**, **V**> {

public void put(**String** key, **V** value) {}

public **V** get(**String** key) { return null; }

}

* Generic classes and interfaces are collectively referred to as ***generic types*.**

**Generic Methods**

* a generic method defines its own formal type parameters.
* you can define a generic method in a generic or a nongeneric class.
* A method’s type parameter list is placed just after its access and nonaccess modifiers and before its return type. Because a type parameter could be used to define the return type, it should be known *before* the return type is used.

**WildCard**

* You can assign an instance of a subclass, say, String, to a variable of its base class, Object. But you can’t assign ArrayList<String> to a variable of type List<Object>. Inheritance doesn’t apply to the type parameters.
* When you use wildcard to declare your variables or method parameters,   
  you lose the functionality of adding objects to a collection. In this   
  case, using method add will result in compilation failure.

(example in WildCard.java)

* You can only iterate through Collection using wildcard, not add items (WildCard2.java)

**Bounded wildcards**

**Upper-bounded (extends)**

* In upper-bounded wildcards, the keyword extends is used for both a class and an interface
* In the preceding method wrapGift(), the loop variable item can be of type Gift or its subtype, Object.
* For collections defined using upper-bounded wildcards, you can’t add any objects. You can iterate and read values from such collections.
* You can use final classes in upper-bounded wildcards. Although class X extends String won’t compile, <? extends String> will compile successfully.

Lower Bounded



**Raw type**

* when a generic class is used without its type information, it’s referred to as its ***raw type*.**
* Example**,** generic class Parcel<T>, its raw type is Parcel.

Parcel parcel = new Parcel<Phone>();

* Because you lose type information when you use variable of raw type, you can pass String object to set(), instead of Phone object. Because it is type unaware it returns Object.

Terms



