



MONKEY C PROGRAMMER'S GUIDE

Garmin's Programming Language for Wearables

connect IQ™

powered by Garmin

Monkey C Programmer's Guide	1
WELCOME TO CONNECT IQ	5
DEVICES AND APIS	5
THE TAO OF CONNECT IQ	6
APP TYPES	6
WATCH FACES	6
DATA FIELDS	6
WIDGETS	6
WEARABLE APPS	6
GETTING STARTED	8
INSTALLING THE CONNECT IQ SDK	8
USING THE CONNECT IQ SDK FROM THE COMMAND LINE	8
MICROSOFT WINDOWS™	8
APPLE OS X™	8
INSTALLING THE PLUG-IN	8
CONFIGURING THE SDK	10
ADDING THE PERSPECTIVE	10
CREATING A NEW PROJECT	11
RUNNING YOUR PROGRAM	13
IMPORTING AN EXAMPLE	14
SIDE LOADING AN APP	16
HELLO MONKEY C!	17
DIFFERENCES FROM OTHER LANGUAGES	18
JAVA	18
LUA/JAVASCRIPT	19
RUBY/PYTHON/PHP	20
FUNCTIONS	21

VARIABLES, EXPRESSIONS, AND OPERATORS	21
DECLARING VARIABLES.	23
ARRAYS	23
DICTIONARIES	24
SYMBOLS	24
CALLING	24
IF STATEMENTS	25
LOOPS	25
RETURNING VALUES FROM FUNCTIONS	26
INSTANCEOF AND HAS	26
ERRORS	27
STRUCTURED EXCEPTION HANDLING	28
OBJECTS	28
CONSTRUCTORS	28
INHERITANCE	28
DATA HIDING	29
MODULES	30
USING STATEMENTS	30
APIs AND APP TYPES	31
SCOPING	31
ANNOTATIONS	32
 RESOURCE COMPILER	 34
 CONCEPTS	 34
REZ MODULE	34
BITMAPS	35
FONTS	36
OVERRIDING RESOURCES	37
 ENTRY POINTS AND THE MANIFEST FILE	 38
 APP TYPE	 38
PRODUCTS	40
PERMISSIONS	41
 USER INTERFACE	 42
 DRAWABLES, VIEWS, AND LAYOUTS	 42
DEFINING LAYOUTS AND DRAWABLES IN RESOURCES	44
USING A LAYOUT	44

DRAWABLES	46
INPUT HANDLING	49
INPUT AND APP TYPES	49
INPUT DELEGATES	49
BEHAVIORS	50
BUILT IN HANDLERS	51
MENU	51
NUMBER PICKER	52
 POSITIONING AND SENSORS	 54
 LOCATION	 54
LOCATION EVENTS	55
SENSORS	57
FIT FILE RECORDING	58
COMMUNICATING WITH ANT SENSORS	58
 COMMUNICATION	 59
 APPROACHES	 59
GARMIN CONNECT MOBILE	59
DIRECT MESSAGING	60
BLE SIMULATION OVER ANDROID DEBUG BRIDGE	61
 DYNAMIC DATA FIELDS	 63
 DATAFIELD AND SIMPLEDATAFIELD	 63
ACTIVITY INFO	64
SIMULATING A WORKOUT	66
 HOW TO PUBLISH	 67
 PUBLISHING THE FIRST VERSION	 68
APPROVAL PROCESS	69

WELCOME TO CONNECT IQ

Developing on wearable platforms is new and challenging. An app developer is asked to make dynamic user experiences that compare to what customers see on their mobile phones and web pages on small screens. If content is too dynamic, you drain the battery faster; if the user is charging the device, they are not using your app.

Connect IQ combines three W's

1. **Wear:** Garmin's experience in power management, activity tracking, and ANT+ sensors means you will spend more time wearing our product and less time charging it.
2. **Where:** Location awareness is at the center of our products. Our products don't become dead weight without a smart phone, but become more alive with one.
3. **Ware:** Garmin's new Connect IQ app system allows for developers to extend their apps into Garmin's wearable ecosystem.

Connect IQ products provide the best of what Garmin has to offer like beautiful design, location awareness, efficient power management with the Connect IQ app system. Using the Connect IQ SDK, developers can create apps for Connect IQ devices and distribute them via the Connect IQ Store.

Monkey C is designed for easy app development on wearable devices. The goal of Monkey C is to round the sharp edges of making a wearable app, letting developers focus on the customer and less on resource constraints. Monkey C is an object-oriented language built from the ground up. It uses reference counting to automatically clean up memory, freeing you from focusing on memory management. The resource compiler helps you import fonts and images and convert them between devices.

If you've worked with dynamic languages in the past like Java™, PHP, Ruby, or Python™, Monkey C should be very familiar.

DEVICES AND APIS

The problem of *API Fragmentation* is a challenge for app developers. If a developer takes advantage of new APIs, they may be targeting devices with the least customer penetration. If they only use established APIs, they may not be taking advantage of new capabilities of the system.

Unlike cell phones, Garmin devices are all wildly different from each other: round screens versus square screens, touch screens versus buttons, and different sensors on each one. While the Java philosophy of "write once run anywhere" is a notable goal, creating a universal API that crosses every Garmin device would inevitably become a lowest common denominator API.

Rather than attempting to abstract away differences in devices, the Connect IQ APIs are tailored to the devices they run on. If the device has a Magnetometer, the device should have the Magnetometer API. If two devices have a Magnetometer, the API should be the same between them. If a device does not have a Magnetometer, it will not provide that API.

THE TAO OF CONNECT IQ

There is a rhyme and reason behind Monkey C to make it easier for developers to support the Garmin ecosystem of products.

The developer chooses what devices to support.

Connect IQ apps can run across multiple devices, but the intended devices are up to the developer. Not every device will be aimed at the markets the developer wants to target or provide the experience the developer wants to provide. The developer should not be forced to support devices they don't want to.

The developer tools should help developers support multiple devices.

The developer tools lessen the weight of supporting multiple devices. The Resource Compiler hides device specific palettes and orientations from the developer. It also allows overrides per device of resources, allowing different images, fonts, and page layouts to be specified in XML. The Simulator needs to expose only the APIs a particular device supports so the developer can test device support.

Similar devices should have similar APIs.

Not all devices will be equal, but there is often commonality between them. Two different watches may have different display technologies, but they both support Bitmaps, Fonts, user events, ANT+, and BLE. A developer writing a sports app should not have to completely rewrite their app to support both devices.

At runtime, the developer can ask what the system 'has'.

Connect IQ applications are dynamically linked to the system. If an app makes a reference to an API that does not exist on a particular system, the app will fail at runtime when the app references the API, not at load time like C++. This allows an app to avoid making the call by taking advantage of the 'has' operator.

APP TYPES

There are four main use cases for third party developers to interact with the watch:

WATCH FACES

Watch Faces replace the main watch face. They are the home screen of the watch.

DATA FIELDS

Data Fields are fields that run within Garmin activities. They allow developers to compute values based off of the current workout.

WIDGETS

Widgets are full screen pages that can be embedded into the main page loop or the page loop of an activity. They must be added to the main page loop through Garmin Connect™ Mobile by the customer. They are loaded when brought on screen and shut down when the user goes to the next widget.

WEARABLE APPS

Apps are started from the action list. Apps push a page onto the UI and exit when the user backs out of the last page of the page stack. In the interim, the app has full control of the UI. These can be used to implement use cases like third party activities, games, and other user-initiated actions.

GETTING STARTED

The Connect IQ Eclipse plug-in turns Eclipse into a Monkey C development environment. Its features include:

- Syntax highlighting editor
- Build integration
- Allows running of applications in the Connect IQ simulator.

INSTALLING THE CONNECT IQ SDK

To install the Connect IQ SDK you need to unzip the connect-iq-sdk-x.y.z ZIP file into a directory.

USING THE CONNECT IQ SDK FROM THE COMMAND LINE

To use the Connect IQ SDK on all platforms you will need Oracle Java™ software development kit version 7 or higher. You can download the JDK at the following address:

www.oracle.com/technetwork/java/javase/downloads/index.html

MICROSOFT WINDOWS™

This release works from the Windows NT shell. The first step for using Monkey C is to point your PATH to the Connect IQ bin directory

```
> set PATH=%PATH%;<Connect IQ bin directory>
```

APPLE OS X™

The first step for using Monkey C is to point your PATH to the Connect IQ SDK bin directory. If you want to use Monkey C in your local terminal you can set the path locally.

```
> export PATH=$PATH:<Connect IQ bin dir>
```

Otherwise if you want Monkey C to be available from all terminal instances you can add it to your bash profile

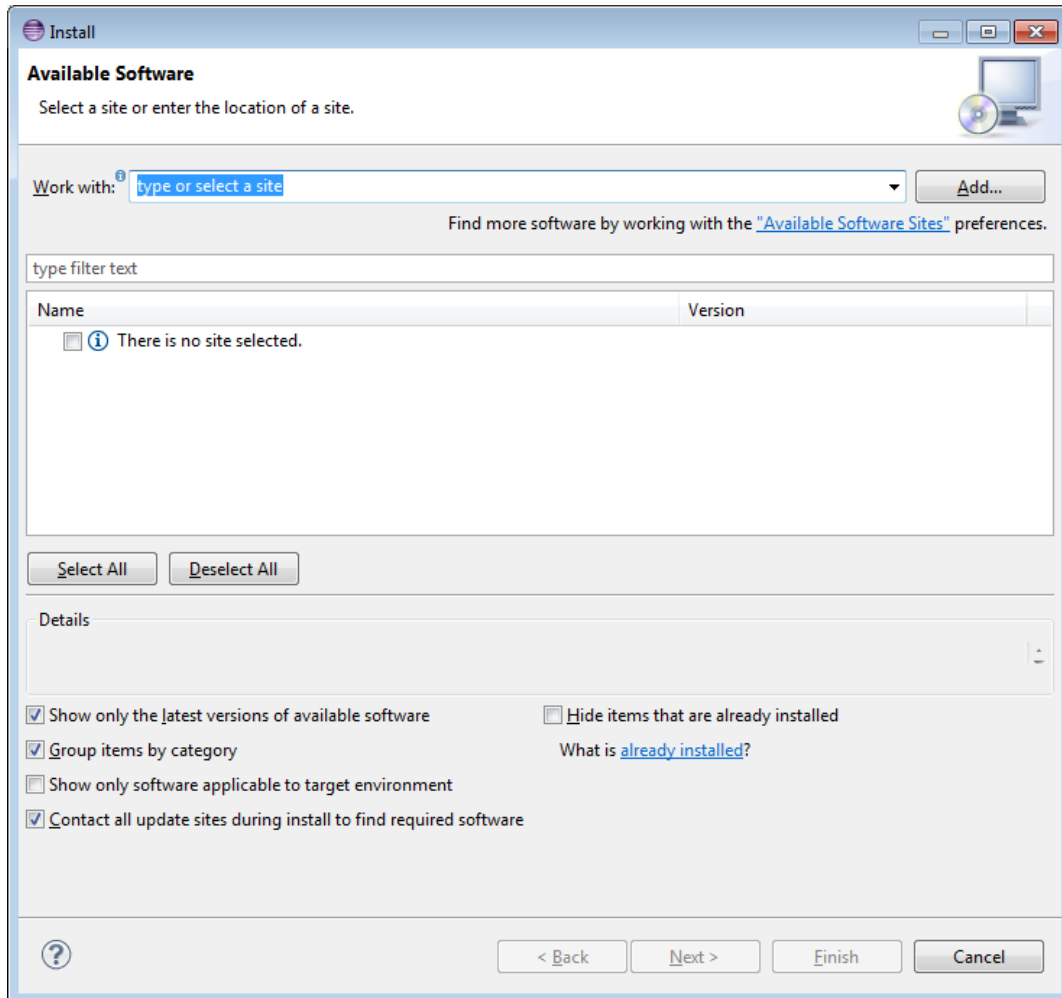
```
> touch ~/.bash_profile
> open ~/.bash_profile
```

The above opens .bash_profile in a text editor. Add the below line to the file:

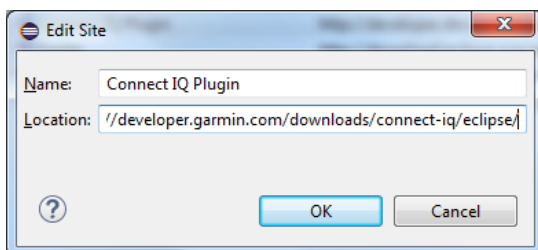
```
export PATH=$PATH:<Connect IQ bin dir>
```

INSTALLING THE PLUG-IN

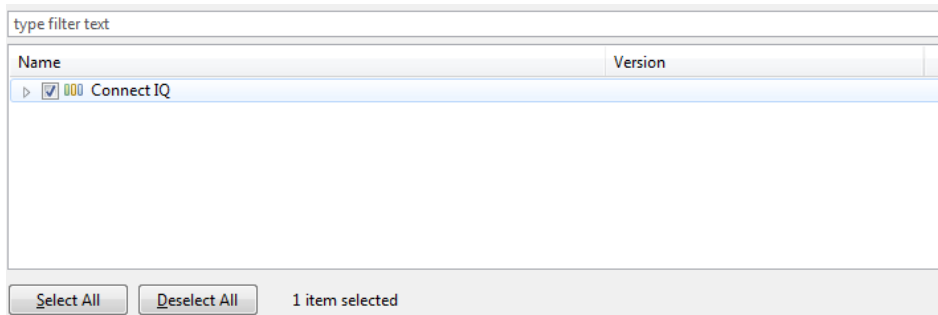
The Connect IQ plug-in requires Eclipse Luna, which is available at <http://www.eclipse.org/luna/>. You will need the Eclipse Standard edition or Eclipse IDE for Java Developers (either 32 or 64 bit). Once Eclipse is installed, you install the plug-in. In the top menu, go to Help | Install New Software.



Use add button to add a new update site: <http://developer.garmin.com/downloads/connect-iq/eclipse/>



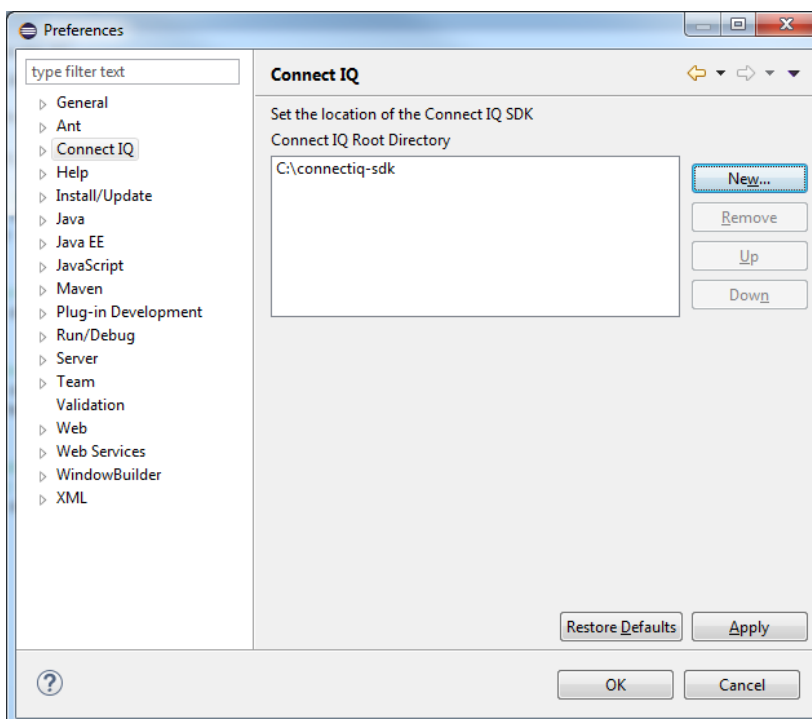
The Connect IQ plug-in will appear in the list. Select the check box and hit Next.



Dependent plug-in modules may need to be downloaded and installed. When this is complete, Eclipse will require you to restart to complete the installation.

CONFIGURING THE SDK

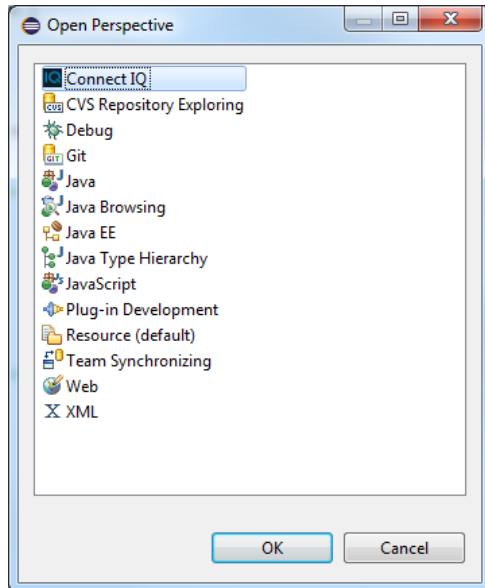
The plug-in will need to know where to find the Connect IQ SDK. To set this, go to the *Window | Preferences (Eclipse | Preferences on OSX)* menu.



Use the New button to add the path to your Connect IQ SDK. Make sure it is referencing the root directory of the SDK and not the *bin* subdirectory.

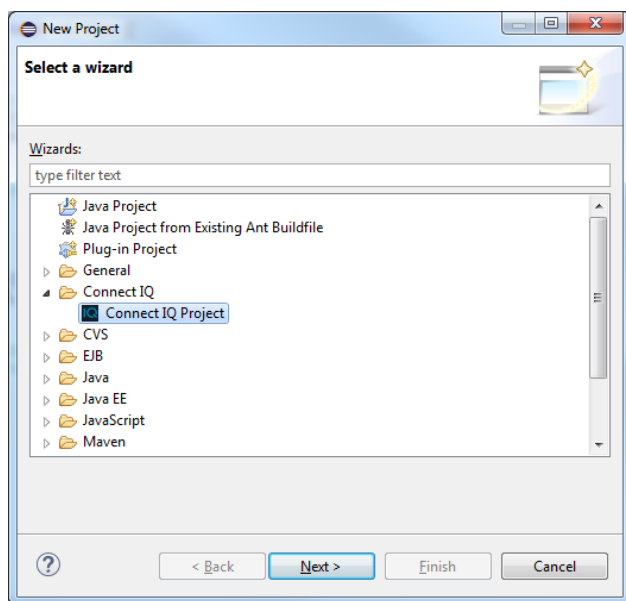
ADDING THE PERSPECTIVE

You will need to add the Connect IQ perspective to Eclipse. To do this go to *Window | Open Perspective | Other...* and select "Connect IQ"

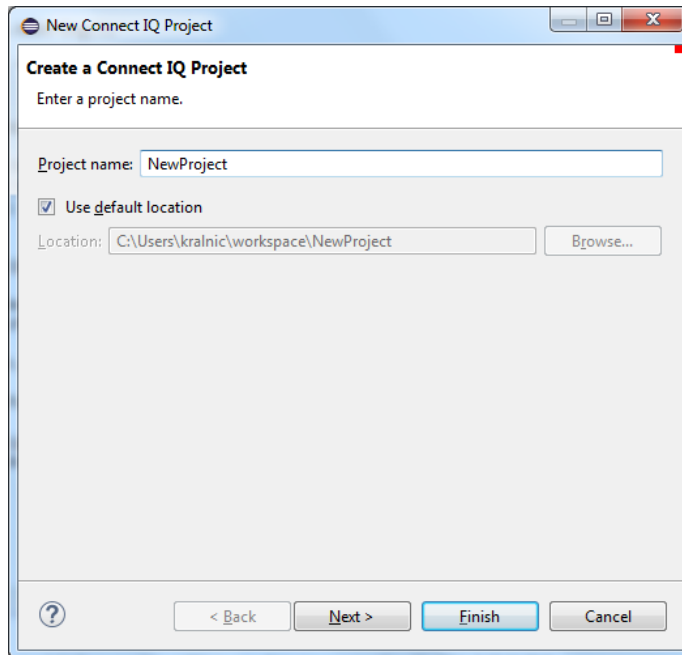


CREATING A NEW PROJECT

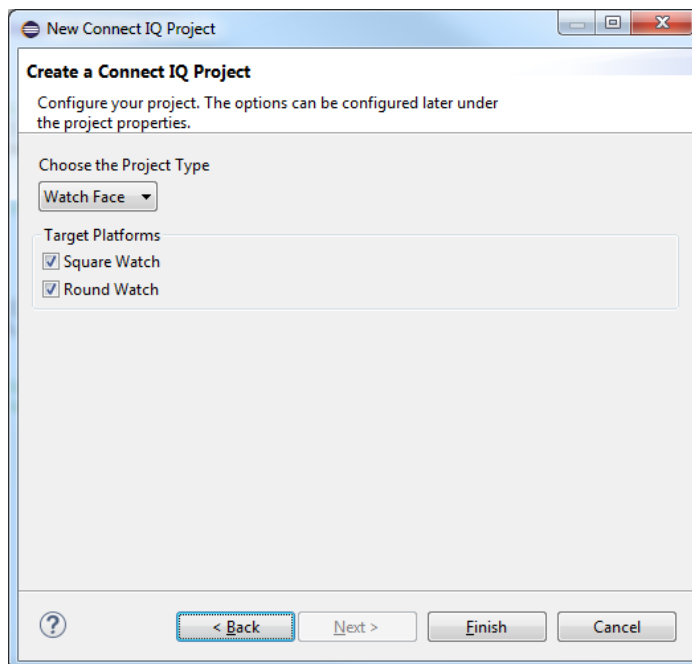
To create your first Connect IQ project, go to *File | New | Project*



Select Connect IQ Project and hit the Next button to continue.



Enter a name for your project and click next. You will need to select the type of app you want to create, and the devices you want to target.



For now, select Watch Face and click Finish. Eclipse will create the project and move you to the Connect IQ Perspective. When your project is initialized the following will be created automatically:

- Source files: The plug in will create a View.mc file and an App.mc file.


- A resource.xml file: The resource.xml is the input to the resource compiler. It will be where all images, fonts, and strings are enumerated.
- A manifest.xml file: The manifest file contains application properties like your app id, your app type, and the devices you target.

RUNNING YOUR PROGRAM

Before you can run a project, Eclipse needs a run configuration to be made for your project. This only needs to be created once. To add one, go to the *Run | Run Configurations* menu.

To create a new Connect IQ run configuration, right click on the *Connect IQ App* option and select "New".

Give your configuration a name at the top. You need to set which Connect IQ project you want to run. It should automatically select your current project, but you can use the Project button to select a different one. Click "Apply" and "Close".

Finally, before you run your program you need to start the simulator. You can do that by using the *Connect IQ | Start Simulator* menu, or clicking on the start simulator icon () on the Toolbar. An oddly blank window will appear.

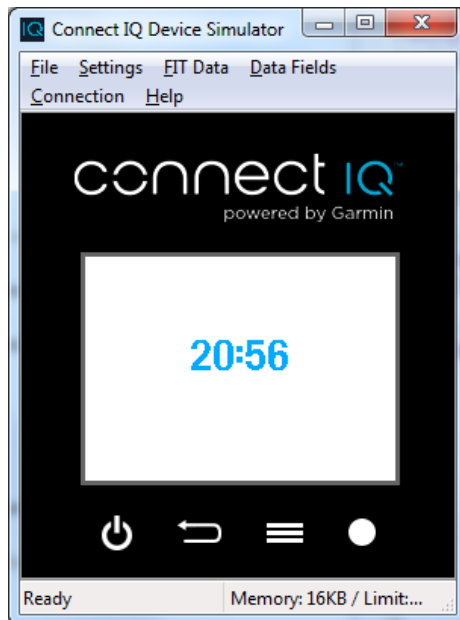


With our Run Configuration created and the simulator started, we're ready to go! Use the down arrow next to the *Run | Run Configurations* menu (or from the Run Configurations selector on the toolbar) to select your configuration.

If all goes well the following run output will appear in the console.

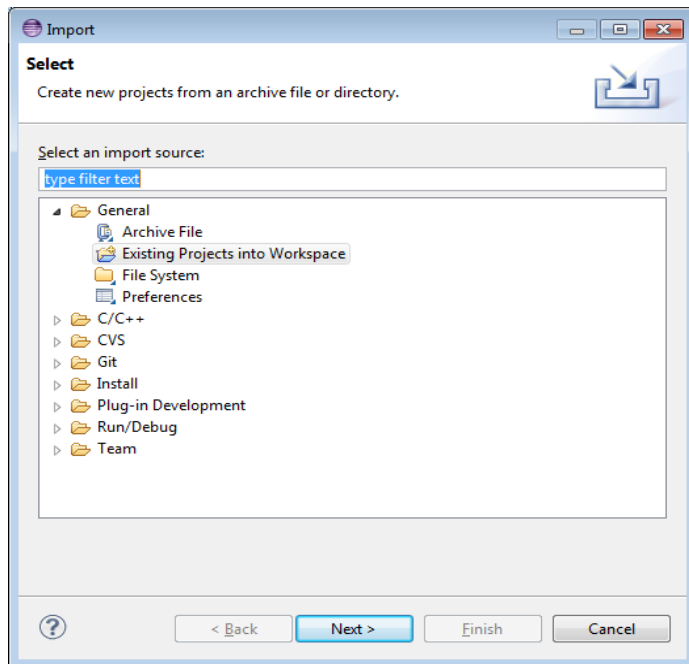
```
Found Transport: tcp
Connecting...
Connecting to device...
Device Version 0.1.0
Device id 1 name "A garmin device"
Shell Version 0.1.0
Connection Finished
Closing shell and usb
```

...and the watch will appear in the simulator.

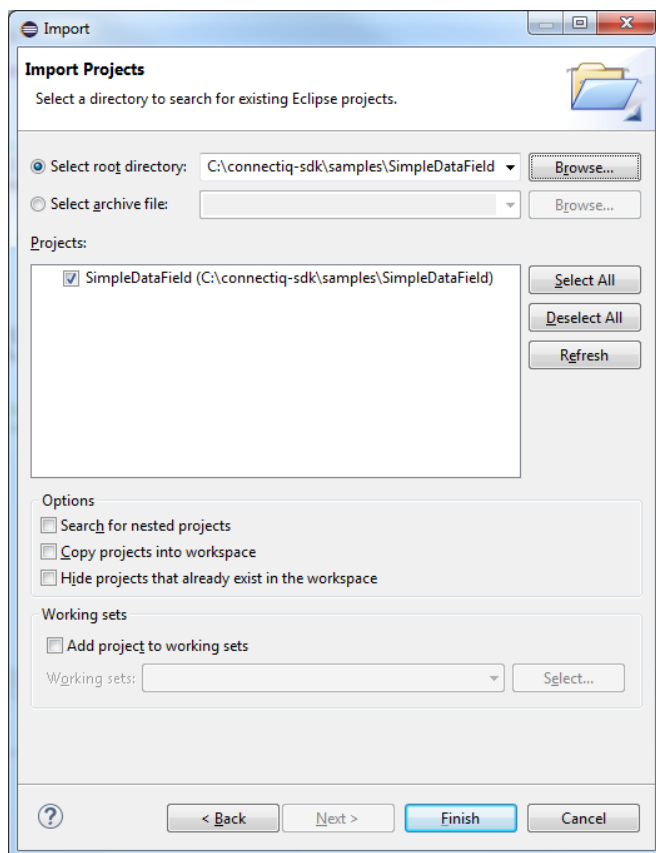


IMPORTING AN EXAMPLE

To try one of the samples you will need to import it into your Eclipse workspace. Go to *File | Import...* to go to the import dialog. You want to import an existing project into the workspace, which is under the "General" folder.

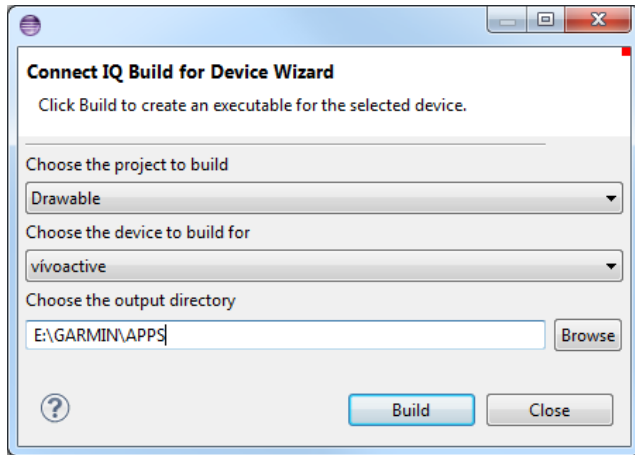


After you hit next, you'll want to pick the root directory of the sample you want to import with the browse button.

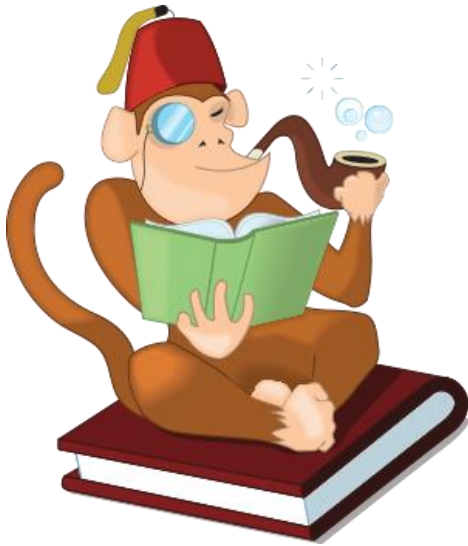


SIDE LOADING AN APP

The Eclipse plug-in provides a wizard to help developers side load an application. You can access this wizard through the Build Project for Device option in the Connect IQ menu. After opening the wizard select the project and device you wish to build for from the drop down menus. You then browse to your device's Garmin/App directory for the output directory and click the Build button. The wizard will create an executable of the selected project and save it to your device for testing.



HELLO MONKEY C!



There is no better way to learn Monkey C than by jumping right in. Let's take a look at the application object of a watch face.

```
using Toybox.Application as App;
using Toybox.System;

class MyProjectApp extends App.AppBase {

    //! onStart() is called on application start up
    function onStart() {
    }

    //! onStop() is called when your application is exiting
    function onStop() {
    }

    //! Return the initial view of your application here
    function getInitialView() {
        return [ new MyProjectView() ];
    }
}
```

If this looks familiar and non-threatening, that's the point. Monkey C is intended to be the language you didn't know you already knew.

At the top is a using statement, which is analogous to C++'s using statement, or an import in Java™, Ruby, or Python™. Using statements lexically bring modules into our name space. After a using clause, we can refer to a module by its shortened name (in this case `System`). `Toybox` is the root module for Monkey C system modules; all the cool toys are kept in there.

To print out values to the debug console, use:

```
System.println("Hello Monkey C!");
```

System is the Toybox.System module we imported with the using statement. Unlike Java namespaces, Modules are static objects that can contain functions, classes, and variables. The println function should be familiar to Java programmers - it prints a string and automatically adds a new line. The system module has a number of useful functions:

```

    ///! Use println() to print to the console with a line terminator.
    ///! @param a Object or string to display
    function println(a)
    {
        print(a.toString() + "\n");
    }

    ///! Use print() to print to the console
    ///! @param x Object or string to display
    function print(x);

    ///! Use getTimer() to get the current millisecond timer.
    ///! @return [Number] System millisecond timer
    function getTimer();

    ///! Get the current clock time with getClockTime().
    ///! @return [System.ClockTime] Current clock time
    function getClockTime();

    ///! Get the current system stats with getSystemStats().
    ///! @return [System.Stats] Current system stats
    function getSystemStats();

    ///! In future versions of the SDK trap() will break
    ///! into the debugger.
    function trap();

    ///! To end execution of the current system with an error message
    ///! of "User terminated", call exit().
    function exit();

    ///! Call error() to cause an error. This will exit the system.
    ///! @param [String] msg Error message to output
    function error(msg);

```

DIFFERENCES FROM OTHER LANGUAGES

As Italian and Spanish derive from Latin, Monkey C derives heavily from past languages. C, Java, JavaScript, Python, Lua, Ruby, and PHP all influenced the design for Monkey C. If you are familiar with any of those languages, Monkey C should be easy to pick up.

JAVA

Like Java, Monkey C compiles into byte code that is interpreted by a virtual machine. Also like Java, all objects are allocated on the heap, and the virtual machine cleans up memory (Java through garbage collection, Monkey C through reference counting). Unlike Java, Monkey C does not have primitive types; integers and floats are objects. This means primitives can have methods just like other objects.

Java is a statically typed language. The developer must declare the types for all parameters of a function, and declare the return value type. The compiler checks these at compile time to ensure type safety. Monkey C is duck typed. Duck typing is the concept that if it walks like a duck, and quacks like a duck, then it must be a duck¹.

```
function add(a, b)
{
    return a + b;
}

function thisFunctionUsesAdd()
{
    var a = add(1, 3); // Return 4
    var b = add("Hello ", "World"); // Returns "Hello World"
}
```

The Monkey C compiler does not verify type safety, and instead causes a runtime error if a function mishandles a method.

Monkey C modules serve a similar purpose as Java packages, but unlike packages, modules can contain variables and functions. It is common for static methods to exist in the module as opposed to a particular class.

LUA/JAVASCRIPT

The main difference between JavaScript or Lua and Monkey C is that functions in Monkey C are not first class. In JavaScript, you can often pass a function to handle a callback.

```
function wakeMeUpBeforeYouGoGo() {
    // Handle completion
}

// Do a long running thing, and pass callback to call when done.
doLongRunningTask( wakeMeUpBeforeYouGoGo );
```

In Lua, to create an object, you bind functions to a hash table

```
function doSomethingFunction(me) {
    // Do something here
}

// Constructor for MyObject
function newMyObject()
{
    local result = {};
    result["doSomething"] = doSomethingFunction;
}
```

Neither of these techniques works in Monkey C, because functions are bound to the object they are created in.

¹ This is different than Monkey Typing, where a thousand monkeys over infinite time write the works of Shakespeare.

If you want to create a callback, you can create a Method object. Method objects are a combination of the function and the object instance or module they originate from. You can then call the method using the invoke method.

```
// Handle completion of long
function wakeMeUpBeforeYouGoGo() {
    // Do something here
}

// Create a call back with the method method (jinx!)
doLongRunningTask(method(:wakeMeUpBeforeYouGoGo));
```

RUBY/PYTHON/PHP

Objects in Ruby/Python are hash tables, and have many of the properties of hash tables. Functions and variables can be added to objects at run time.

Monkey C objects are compiled and cannot be modified at runtime. All variables have to be declared before they can be used, either in the local function, the class instance, or in the parent module.

When importing a module, all classes inside the module have to be referenced through their parent module. You import modules, not classes, into your namespace.

FUNCTIONS



Functions are the meat² of your program. Functions define discrete callable units of code.

Monkey C functions can take arguments, but because Monkey C is a dynamically typed language you do not declare the argument type; just its name. Also, you do not have to declare the return value of a function, or even if a function returns a value, because all functions return values. You can specify the return value with a `return` statement, but if your function doesn't have a `return` statement it will return the last value on the stack.

Functions can exist in a class or module, or appear in the global module.

VARIABLES, EXPRESSIONS, AND OPERATORS

Variables are declared with the `var` keyword. All variables must be declared before they are used. The basic types that Monkey C supports are:

- **Integers** - 32 bit signed integers
- **Floats** - 32 bit floating point numbers
- **Long** – 64 bit signed integers
- **Double** – 64 bit floating point numbers
- **Booleans** - true and false
- **Strings** - Strings of characters

² Tofu for the vegetarians, BBQ for Kansans...

- **Objects** – Instantiated objects (defined with the class keyword)
- **Arrays** - Allocated with the syntax "new [X]" where X is an **expression** computing the size of the array
- **Dictionaries** - Associative arrays, allocated with the syntax "{}"

Monkey C supports the following operators

Precedence	Operator	Description
1	new ! ~ ()	Creation Logical NOT Bitwise NOT Function invocation
2	* / % & << >>	Multiplication Division Modulo Bitwise AND Left Shift Right Shift
3	+ - ^	Addition Subtraction Bitwise OR Bitwise XOR
4	< <= > >= == !=	Less Than Less Than Or Equals Greater Than Greater Than Or Equals Equals Not Equals
5	&& and	Logical And
6	 or	Logical Or

DECLARING VARIABLES.

All local variables must be declared ahead of time using the var keyword.

```
var n = null; // Null reference
var x = 5; // 32 bit signed integers
var y = 6.0; // 32 bit floating point
var l = 51; // 64 bit signed integers
var d = 4.0d; // 64 bit floating point
var bool = true; // Boolean (true or false)
var arr = new [20 + 30]; // Array of size 50
var str = "Hello"; // String
var dict = { x=>y }; // Dictionary: key is 5, value is 6.0
var z = arr[2] + x; // Null pointer waiting to happen
```

In the Monkey C language, all values (including numeric values) are objects.

ARRAYS

Arrays in Monkey C, like variables, are typeless, so you do not have to declare a data type. There are two forms for creating a new array. If you want to create an empty array of a fixed size, use this:

```
var array = new [size];
```

Size is an expression of how large of an array to create.

Monkey C does not have a direct way of creating an empty two-dimensional array, but you can initialize one with this syntax:

```
// Shout out to all the Java programmers
// in the house
var array = new [first_dimension_size];
// Initialize the sub-arrays
for(var i = 0; i < first_dimension_size; i += 1)
{
    array[i] = new [second_dimension_size];
}
```

Finally, if you want to pre-initialize an array, you can use this syntax:

```
var array = [1, 2, 3, 4, 5];
```

Elements are expressions, so you can also create multidimensional arrays using this syntax:

```
var array = [ [1,2], [3,4] ];
```

DICTIONARIES

Dictionaries, or associative arrays, are a built in data structure in Monkey C.

```
var dict = { "a" => 1, "b" => 2 }; // Creates a dictionary
System.println(dict["a"]); // Print "1"
System.println(dict["b"]); // Print "2"
System.println(dict["c"]); // Print "null"
```

If you want to initialize an empty dictionary, you can use the following syntaxes:

```
var x = {}; // Empty dictionary
```

By default, objects hash on their reference value. Classes should override the `hashCode()` method in `Toybox.Lang.Object` to change the hashing function for their type:

```
class Person
{
    ///! Return a number as the hash code. Remember that the hash code
    ///! must be the same for two objects that are equal.
    ///! @return Hash code value
    function hashCode() {
        // Using the unique person id for the hash code
        return mPersonId;
    }
}
```

Dictionaries automatically resize and rehash as the contents grow or shrink. This makes them extremely flexible, but it comes at a cost. Insert/removal of the contents can cause performance problems if you are accidentally causing resizes and rehashing. Also, because hash tables require extra space for buckets, they are not as space efficient as either objects or arrays.

SYMBOLS

Symbols are lightweight constant identifiers. When the Monkey C compiler finds a new symbol, it will assign it a new unique value. This allows you to use symbols as keys or constant without explicitly declaring a `const` or `enum`.

```
var a = :symbol_1;
var b = :symbol_1;
var c = :symbol_2;
Sys.println(a == b); // Will print true
Sys.println(a == c); // Will print false
```

Symbols can be useful when wanting to create keys without having to declare an `enum`.

```
var person = { :firstName=>"Bob", :lastName=>"Jones"};
```

CALLING

To call a method within your own class or module, you can simply use the function call syntax.


```
function foo(a)
{
    //Assume foo does something really impressive
}
function bar()
{
    foo("hello");
}
```

If you are calling on an instance of an object, you have to precede the call with the object and a "."

IF STATEMENTS

If statements allow you to have branch points in your code.

```
myInstance.methodToCall(parameter);

if (a == true)
{
    // Do something
}
else if (b == true)
{
    // Do something else
}
else
{
    // If all else fails
}

// Monkey C also supports the ternary operator
var result = a ? 1 : 2;
```

The expression inside the `if` statement is required to be an expression; assignments are not allowed. Things that will evaluate to true are

- true
- A non-zero integer
- A non-null object

LOOPS

Monkey C supports for loops, while loops, and do/while loops. While and do/while loops have a familiar syntax:

```
// do/while loop
do
{
    // Code to do in a loop
}
while( expression );

// while loop
while( expression )
{
    // Code to do in loop
}
```

Loops must have braces around them as it does not support single line loops.

```
// Monkey C does allow for variable declaration in for loops.
for(var i = 0; i < array.size(); i ++)
{
    // Loop
}
```

RETURNING VALUES FROM FUNCTIONS

All functions return values in Monkey C. You can explicitly set the return value by using the return keyword.

```
return expression;
```

The expression is optional. Functions without a return automatically return the last value operated on.

INSTANCEOF AND HAS

As a duck typed language, Monkey C gives the programmer great flexibility, but the trade off is that the compiler cannot perform the type checking you would get in a language like C, C++ or Java. Monkey C provides two tools to do runtime type checking - `instanceof` and `has`.

The `instanceof` operator offers the ability to check if an object instance inherits from a given class. The second argument is the class name you want to check against.

```
var value = 5;
// Check to see if value is a number
if (value instanceof Toybox.Lang.Number)
{
    System.println("Value is a number");
}
```

The `has` operator lets you check if a given object has a symbol, which could be a public method, instance variable, or even a class definition or module. The second argument is the symbol you want to check. It is handy for checking for supported APIs. For example, assume we have magnetometer libraries in `Toybox.Sensor.Magnetometer`, but not all products have a magnetometer. Here is an example of changing your implementation based on those criteria:

```
var impl;
// Check to see if the Magnetometer module exists in Toybox
if (Toybox has :Magnetometer)
{
    // Instantiate the magnetometer implementation
    impl = new ImplementationWithMagnetometer();
}
else
{
    impl = new ImplementationWithoutMagnetometer();
}
```

Monkey C object oriented design patterns in conjunction with the `has` and `instanceof` operator enables software that has implementations for many devices in one code base.

ERRORS

Because Monkey C uses dynamic typing, there are many errors for which the compiler cannot check. If the error is of high enough severity, it will raise a fatal API error and cause your app to terminate at runtime. These errors cannot be caught. At this time all of these errors are fatal and there is no way to trap them, though this may be addressed in future updates.

- **Symbol Not Found:** A name is being referenced that does not exist in specified object or module.
- **Invalid Value:** An argument passed to a method is invalid.
- **Null Reference:** A value is being requested from a null value.
- **Out Of Memory:** The application is out of available memory.
- **Array Out Of Bounds:** An attempt is being made to reference an array outside of its allocated bounds.
- **Unexpected Type:** An operation is being done on a type that does not support it. An example would be trying to perform a bitwise or on two strings.
- **Stack Underflow / Stack Overflow:** The application stack has crossed an illegal boundary, either going past the bottom or flowing over the top of the stack limit.
- **Circular Dependency:** There is a loop in the dependency graph of a module or object that prevents a module or object from being constructed.
- **System Error:** A generic error used by the Toybox APIs for fatal errors.
- **Too Many Timers:** The app is attempting to create too many timers.
- **Too Many Arguments:** Monkey C currently has a limit of 9 arguments for a function.
- **Illegal Frame:** The return address on the stack is corrupted.
- **File Not Found:** The app file could not be found. This is usually caused when trying to load a resource from the app file.

- **Watchdog Tripped:** Your Monkey C function has executed for too long. This prevents a Monkey C program from hanging the system via an infinite loop.
- **Permission Required:** Certain Monkey C modules require you to declare a permission in the manifest before you can use them.
- **Unhandled Exception:** An exception was thrown and was not caught by an exception handler.

STRUCTURED EXCEPTION HANDLING

Monkey C supports structured exception handling. These are for non-fatal errors that can be recovered from. The syntax should be familiar for Java and Javascript developers.

```
try {
    // Code to execute
}
catch( ex instanceof AnExceptionClass ) {
    // Code to handle the throw of AnExceptionClass
}
catch( ex ) {
    // Code to catch all execeptions
}
finally {
    // Code to execute when
}
```

You can use the `throw` keyword to throw an exception.

OBJECTS

Objects are created with the `class` keyword. Classes allow you to bind data and operations together on an object.

CONSTRUCTORS

When an object is instantiated with the `new` keyword, the memory is allocated and the "initialize" method is called.

```
class Circle
{
    hidden var mRadius;
    function initialize(aRadius)
    {
        mRadius = aRadius;
    }
}
function createCircle()
{
    var c = new Circle(1.5);
}
```

INHERITANCE

Monkey C uses the `extends` keyword to support class inheritance.

```
using Toybox.System as Sys;

class A
{
    function print()
    {
        Sys.print("Hello!");
    }
}

class B extends A
{
}

function usageSample()
{
    var inst = new B();
    inst.print(); // Will print "Hello!"
}
```

You can call super class methods by using the super class's symbol.

```
using Toybox.System;

class A
{
    function print()
    {
        System.print("Hello!");
    }
}

class B extends A
{
    function print()
    {
        // Call the super class implementation
        A.print();
        // Amend the output
        System.println(" Hola!");
    }
}

function usageSample()
{
    var inst = new B();
    inst.print(); // Will print "Hello! Hola!"
}
```

DATA HIDING

Classes have two levels of access - public and hidden. Public is the default. Hidden is the same as "protected"; it says that a variable or function is only accessible to a class or its subclasses.

```
class Foo
{
    hidden var mVar;
}
function usageSample()
{
    var v = new Foo();
    Toybox.System.println(v.mVar); // Runtime Error
}
```

MODULES

Modules in Monkey C allow for the scoping of classes and functions. Unlike Java packages, Monkey C modules have many of the same properties as classes. You can have variables, functions, and classes at the module level.

```
module MyModule
{
    class Foo
    {
        var mValue;
    }
    var moduleVariable;
}

function usageSample()
{
    MyModule.moduleVariable = new MyModule.Foo();
}
```

USING STATEMENTS

You can bring a module into your scoping level with the `using` keyword. Using allows you to import a module into another class or module by a symbol.

```
using Toybox.System;

function foo()
{
    System.print("Hello");
}
```

The `as` clause lets you give a module a different name within scope. This is useful when you want to shorten the name or simply disagree with our naming scheme³.

³ We are all about conflict avoidance here.

```
using Toybox.System as Sys;

function foo()
{
    Sys.print("Hello");
}
```

Using statements are scoped to the class or module in which they are defined.

APIs AND APP TYPES

The app type defines the user context of an app. Watch faces, for example, have many constraints because they operate in low power mode. To enforce these limits, the Monkey VM will limit your available APIs based on your app type.

Data Field	Watch Face	Widget	App
Activity ActivityMonitor Application Graphics Lang Math System Time UserProfile* WatchUi	ActivityMonitor Application Graphics Lang Math System Time Timer UserProfile* WatchUi	ActivityMonitor Application Attention Communications* Graphics Lang Math Positioning* Sensor* System Time Timer UserProfile* WatchUi	ActivityMonitor Ant Application Attention ActivityRecording* Communications* Graphics Lang Math Positioning* Sensor* System Time Timer UserProfile* WatchUi

**Requires app permission*

If you request a `Toybox` module outside this list for your app type, you will get a `SymbolNotFound` error.

SCOPING

Monkey C is a message-passed language. When you call a function, at runtime the VM does a look up operation to find the function you are calling. It has a hierarchy that it will search:

1. Instance members of the class.
2. Members of your super class.
3. Static members of this class.
4. Members of the parent module, and the parent modules up to the global namespace.
5. Members of the super class's parent module up to the global namespace.

The code below tries to clarify. If a() is called on an instance of Child(), it will be able to access b(), c(), and d(), because

- b() is a member of the parent module of the object.
- c() is a static member of the object.
- d() is the parent module of the parent module, also known as the globals module.

```
using Toybox.System as Sys;
// A globally visible function
function d()
{
    Sys.print("this is D!");
}
module Parent
{
    // A module function.
    function b()
    {
        Sys.print("This is B!");
        // Call a globally visible function
        d();
    }
    // A child class of a Parent module
    class Child
    {
        // An instance method of Child
        function a()
        {
            Sys.print("This is A!");
            // Call a function in our parent module
            b();
            // Call a static function within the class.
            c();
            // Call a globally visible function.
            d();
        }
        // A static function of Child.
        static function c()
        {
            Sys.print("This is C!");
            // Call a method in the parent module.
            // Note that static methods can't call
            // instance method but still have access
            // to parent modules.
            b();
            // Call a globally visible function
            d();
        }
    }
}
```

ANNOTATIONS

Monkey C allows associating symbols with a class's or module's methods and variables. These symbols are currently written into the debug.xml file generated by the compiler, but may be used in the future to add new features without changing the Monkey C grammar.

```
(:debugOnly) class TestMethods
{
  (:test) static function testThisClass(x)
}
```

RESOURCE COMPILER



The resource compiler compiles images, text, and static data into a resource database that the application can access at run time. The resource compiler is tied into the Monkey C compiler. Its input is an XML file.

```
<resources>
  <bitmap id="bitmap_id" filename="path/for/image" />
  <font id="font_id" filename="path/to/fnt" >
  <string id="string_id">Hello World!</string>
</resources>
```

Note: In order for the Eclipse Plug-in to recognize a resource file its path must include a folder with the word *resource* in it.

CONCEPTS

REZ MODULE

The resource compiler auto-generates a Monkey C module named `Rez` that contains the resource ids for the resource file. These identifiers are used to refer to your resources.

```
module Rez
{
    module Drawables
    {
        var bitmap_id = 123;
    }
    module Strings
    {
        var hello_id = 456;
    }
    module Fonts
    {
        var font_id = 789;
    }
}
```

The code can use the `Rez` class to reference the resources at run time. For example, let's say you have a bitmap you want to use in your view. Before you can use it, you have to load it from the resource file.

```
image = Ui.loadResource(Rez.Drawables.id_monkey);
```

Now you can draw the bitmap in your update handler.

```
dc.drawBitmap(50,50,image);
```

Resources are reference counted just like other Monkey C objects. Loading a resource can be an expensive operation, so do not load resources when handling screen updates.

BITMAPS

Garmin devices have different form factors, screen sizes, and screen technologies. Bitmaps need to be explicitly converted for every device. Therefore, the resource compiler will generate resources for every intended product. This will allow the developer to have one set of resources for black and white products, one set for color products, one for larger screen sizes, etc. The resource compiler supports jpg/jpeg, bmp/wbmp, gif and png file formats.

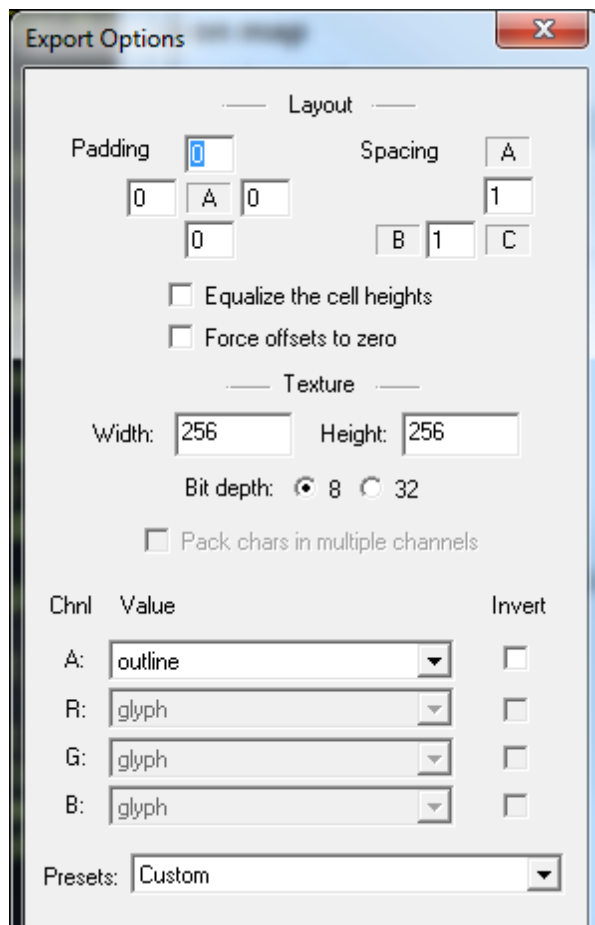
While each device has a unique palette the developer can specify a palette to use for an image. The resource compiler will map the colors that are defined in the developer's palette to their closest match in the device palette and use only those colors. A palette can be defined using the following syntax:

```
<resources>
  <bitmap id="bitmap_id" filename="path/for/image">
    <palette>
      <color>FF0000</color>
      <color>FFFFFF</color>
      <color>0000FF</color>
    </palette>
  </bitmap>
</resources>
```

A dithering attribute may also be set for a bitmap resource. The dithering attribute specifies what type of dithering should be used when compiling the bitmap. The valid values are "none" and "floyd_steinberg"; if not specified the default value is "floyd_steinberg".

FONTS

The resource compiler supports fonts from the [BMFont](http://www.angelcode.com/products/bmfont/) tool (you can download it at <http://www.angelcode.com/products/bmfont/>). Use the BMFont tool to convert a font from many formats into BMFont format. The resource compiler reads fonts in TXT/PNG format. Currently only 1-bit color fonts are supported, so the color can be set using `dc.setColor()`.



Large bitmap fonts can take a lot of run time memory. Sometimes, you only want particular glyphs (like numbers for a watch face) to be large sized. You can use the filter attribute to specify the particular glyphs to include.

```
<!-- Only include digits from this large font -->
<font id="font_id" file="big_font.fnt" filter="0123456789:"/>
```

OVERRIDING RESOURCES

Resources can be overridden using qualifiers. A resource with the same ID but a more specific qualifier will replace the resource with less specific qualifiers when the app is compiled.

Resource qualifiers are applied to the base resources folder and all resources found within that folder will inherit the qualifiers of the base folder. Qualifiers are added to a resources folder by adding a hyphen (-) followed by a valid qualifier value. Multiple qualifiers may be used on a single folder but they must be separated by hyphens and given in the order they are found in the table below. All qualifiers are case insensitive.

Qualifier Type	Qualifier Values	Description
Device	square_watch round_watch fr920xt	A device code. Resource included in a folder with a device qualifier will override the resources with the same ID that are defined in the base resource folder when building for that specific device.
Language and Region	Examples <ul style="list-style-type: none"> • fre • fre_FRA • fre_CAN 	ISO 639-2 language code followed by an optional ISO 3166-1 alpha-3 country code. If you wish to include a region code append it after the language code with an underscore (_) separating the two. You cannot specify a region alone. These qualifiers are ONLY applied to string resources.

ENTRY POINTS AND THE MANIFEST FILE



All Connect IQ apps require a manifest file. The manifest file is an XML file that specifies application properties like the application type and the supported products. The manifest file is created for you automatically by the Eclipse plug-in, but you can create it by hand as well.

```
<iq:manifest version="1">
  <iq:application entry="CommExample" id="a3421feed289106a538cb9547ab12095"
    name="AppName" launcherIcon="LauncherIcon" type="widget">

    <iq:products>
      <iq:product id="square_watch"/>
    </iq:products>

    <iq:permissions>
      <iq:uses-permission id="Communications"/>
    </iq:permissions>

  </iq:application>
</iq:manifest>
```

The application element has a number of important attributes. The id field is a 128 bit UUID identifier. You can generate your own identifier at <http://www.uuidgenerator.net/version4> or using standard tools.

APP TYPE

The type field specifies what kind of application you are developing. Currently Connect IQ supports four types of apps:

1. watchface
2. datafield
3. widget

4. watch-app

Where your app appears on the device and what API's you can call depend on the app type selected in the manifest file.

The `name` and `launcherIcon` attributes must specify a resource ID that is defined in the app resources. If a `launcherIcon` isn't specified a default icon will be compiled into the application.

The `entry` attribute must specify the `Toybox.Application.AppBase` object for your application. Every application must include an `Application` object, which serves as the entry point for your application.

```

    /// AppBase is the base class for an app. All apps shall inherit
    /// from this class. It is used to manage the lifecycle of an app.
    /// For widgets and watch-apps, the functions are called in the
    /// following order: onStart(), getInitialView() and onStop().
    /// For watchfaces and datafields, only getInitialView() is called.
    /// Every AppBase object has access to an object store to persist
    /// data.
    class AppBase
    {
        /// Before the initial WatchUi.View is retrieved, onStart() is called.
        /// This is where app level settings can be initialized or retrieved from
        /// the object store before the initial View is created.
        function onStart();

        /// To retrieve the initial WatchUi.View and WatchUi.InputDelegate
        /// of the application, call getInitialView(). Providing a
        /// WatchUi.InputDelegate is optional for widgets and watch-apps. For
        /// watchfaces and datafields, an array containing just a WatchUi.View should
        /// be returned as input is not available for these app types.
        /// @return [Array] An array containing
        ///         [ WatchUi.View, WatchUi.InputDelegate (optional) ]
        function getInitialView();

        /// When the system is going to terminate an application, onStop() is called.
        /// If the application needs to save state to the object store it should be
        /// done in this function.
        function onStop();

        /// To get the data associated with the given key from the object store,
        /// use getProperty().
        /// @param key Key of the value to retrieve from the object store
        ///         (cannot be a Symbol)
        /// @return [Object] Content associated with the key, or null if the key
        ///         is not in the object store
        function getProperty(key);

        /// Using a key, store the given data in the object by calling setProperty().
        /// @param key The key used to store and retrieve the value from the object
        ///         store (cannot be a Symbol)
        /// @param [String] value The value to put into the object store
        function setProperty(key, value);
    }

```

An application object should override `getInitialView()` to provide the view object they want to push initially. You have to return an array with either a View and a Delegate, or just a one element array with the View object.

```
return [ new MyView(), new MyDelegate() ];
```

The Eclipse plug-in will generate an Application object for you when creating a project.

PRODUCTS

Garmin makes a wide variety of products for many use cases. Monkey C makes it easy to write for our all Connect IQ devices, but it is impossible what future product might be incompatible with your app. For this reason, Monkey C asks the developer what Connect IQ devices they

choose to support. As new Connect IQ products appear on the market the simulator will be updated to support them, and you will have the option to support it or not.

You list which products you want in the Products block. The SDK currently supports two imaginary products – a round-watch and a square-watch. The round watch is button only, and the square watch is a touch screen product with less buttons.

```
<iq:products>
  <iq:product id="Product"/>
</iq:products>
```

PERMISSIONS

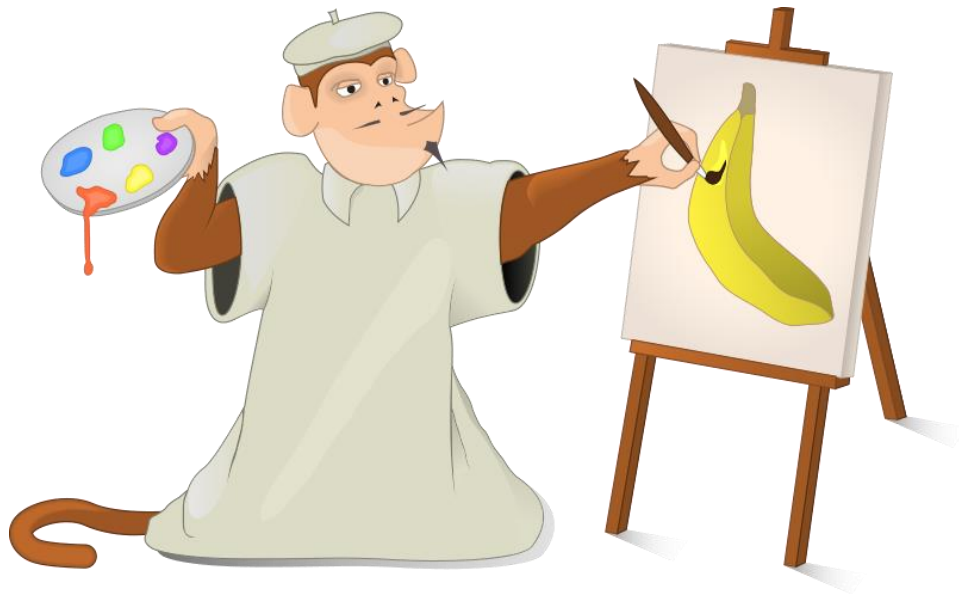
Certain modules expose personal information about the user or expose communication to the internet. To use these modules you must request permission from the user at installation time. To request permission, the module name must be added to the permissions list of the manifest file. The following modules require permission:

- Sensor
- Positioning
- Communications
- FIT

More modules may be added to this list as modules are added to the API. To request permission, use the following syntax in the manifest file:

```
<iq:permissions>
  <iq:uses-permission id="Module"/>
</iq:permissions>
```

USER INTERFACE



Making a user interface toolkit that can adapt to multiple products is hard, but it is even harder in the Garmin ecosystem. The user interfaces of Garmin devices vary by product. Some have physical buttons, some have coordinate touch screens, and some provide up/down/action buttons. The mechanical designs of these products may be very different depending on the use case the product is designed for.

Rather than design a user interface toolkit that adapts to what device the UI is running on, the Connect IQ SDK makes it easier to port the user interface to different products. At the resource compiler level, we include a way to define a page in XML as well as specify page definitions per product. These tools make it easy to support multiple devices with a single app.

DRAWABLES, VIEWS, AND LAYOUTS

Watch faces and apps have a page stack. A View is an object that represents a page. Views can be pushed onto and popped off of the page stack, or a View can replace another View on the page stack with a transition.

```

    /// View is the base class for a drawable page. It handles
    /// the lifecycle of each View in an app. For widgets and
    /// watch-apps the lifecycle is onStart(), onLayout(), onShow(),
    /// onUpdate() and onHide(). For watchfaces and datafields
    /// the lifecycle is onLayout(), onShow() and onUpdate().
    class View
    {
        /// The entry point for the View is onLayout(). This is called before the
        /// View is shown to load resources and set up the layout of the View.
        /// @param [Graphics.Dc] dc The drawing context
        /// @return [Boolean] true if handled, false otherwise
        function onLayout(dc);

        /// When the View is brought into the foreground, onShow() is called.
        /// @return [Boolean] true if handled, false otherwise
        function onShow();

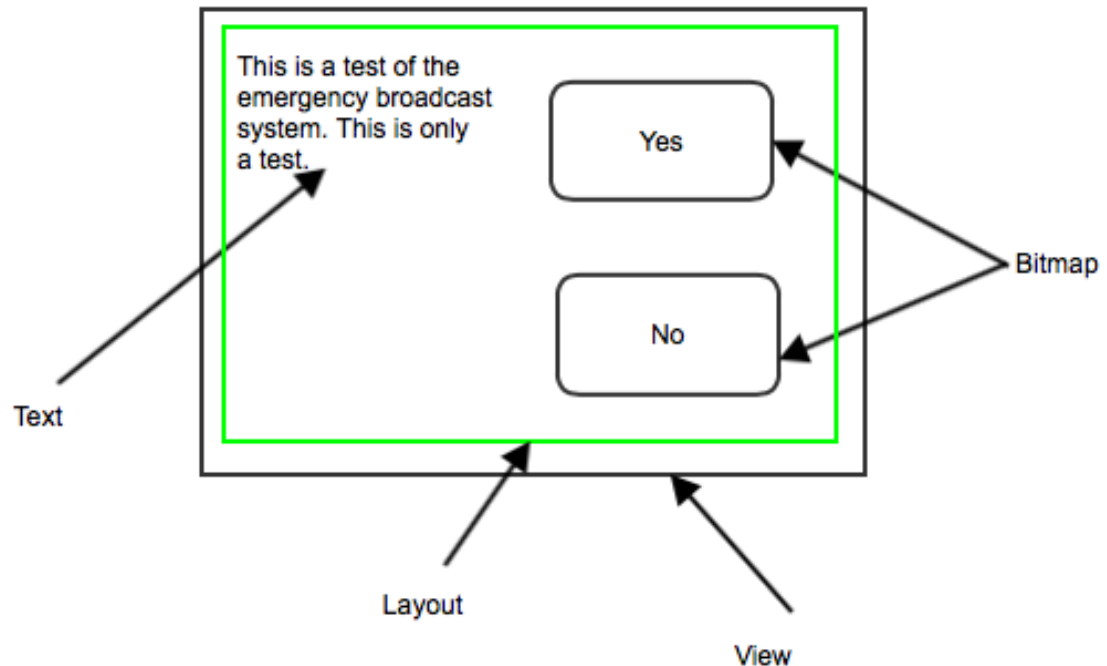
        /// When a View is active, onUpdate() is used to update
        /// dynamic content. This function is called when
        /// the View is brought to the foreground. For widgets and
        /// watch-apps it is also called when WatchUi.requestUpdate()
        /// is called. For watchfaces it is called once a minute and
        /// for datafields it is called once a second. If a class that
        /// extends View does not implement this function then any
        /// Drawables contained in the View will automatically be drawn.
        /// @param [Graphics.Dc] dc The drawing context
        /// @return [Boolean] true if handled, false otherwise
        function onUpdate(dc);

        /// Before the View is removed from the foreground, onHide() is called.
        function onHide();

        /// Use setLayout() to set the layout for the View. If the extending class does
        /// not override onUpdate(), then all Drawables contained in layout will
        /// automatically be drawn by the View.
        /// @param [Array] layout An array of Drawables
        function setLayout(layout);
    }

```

Each View contains a Layout. A Layout is an array of Drawables, where each Drawable is positioned in the View. A Drawable is an abstract object that can draw itself to a device context through the `draw(dc)` method.



Any object that extends `Drawable` is a drawable object. Monkey C provides the basic drawables `Text` and `Bitmap`, allowing `Text` and `Bitmap` resources to be included in layouts.

Drawables expose their properties publicly, which will become useful when we discuss the animation system.

DEFINING LAYOUTS AND DRAWABLES IN RESOURCES

The resource compiler allows you to customize your page layouts to a particular device without changing any Monkey C code. In addition, it allows you to define `DrawableList` objects, which are `Drawable` objects that can draw a number of graphics primitives.

USING A LAYOUT

When defining a layout in XML you will simply list the drawable objects you want included in your layout inside a set of layout tags. Each of the drawables you list will be turned into Monkey C `Drawable` objects and drawn one after the other. Because of this if two drawables in your layout overlap each other the drawable that is defined second will draw on top of the drawable that is defined first.

```
<resources>
  <layout id="MainLayout" background="Gfx.COLOR_WHITE">
    <drawable id="MainBackground" />
    <label text="Page Heading" x="10" y="5" font="Gfx.FONT_LARGE"
color="Gfx.COLOR_BLACK" />
    <label text="Your information goes here." x="10" y="25" font="Gfx.FONT_MEDIUM"
color="Gfx.COLOR_DK_GREY" />
  </layout>
</resources>
```

To use this layout in your code simply call `setLayout()` inside your `onLayout(dc)` function. You will need to call `View's onUpdate(dc)` if you plan on using the `onUpdate(dc)` function to update dynamic values on the screen.

```
class MainView {
  function onLayout(dc) {
    setLayout(Rez.Layouts.MainLayout(dc));
  }

  function onUpdate(dc) {
    // Call parent's onUpdate(dc) to redraw the layout
    View.onUpdate(dc);

    // Update anything that needs update here.
  }
}
```

The following attributes are supported by the layout tag.

Attribute	Definition	Valid Values	Default Value
id (required)	The handle for the layout. This is used to reference the layout in the Rez module.	Any value that starts with a letter.	NA

Drawables (both bitmaps and drawable XML resources) can be included in a layout using the `drawable` tag. The following attributes are supported by the `drawable` tag.

Attribute	Definition	Valid Values	Default Value
id (required)	The handle for the drawable. You must define the drawable you want to use as a drawable XML resource. The ID you provide here simply references the defined drawable.	Any value that starts with a letter. The drawable must also be defined in a resource file.	NA

Text can also be include in layouts. To include text use the label tag. The label tag supports the following attribute.

Attribute	Definition	Valid Values	Default Value
text	The text to be displayed.	NA	An empty string
font	The font to be used when drawing the text.	Gfx Font constant or the ID of a user defined font.	Gfx.FONT_MEDIUM
x	The X coordinate of the point that the text will be justified against.	Pixel value	0
y	The Y coordinate of the point that the text will be justified against.	Pixel value	0
justification	How the text should be justified in relation to the X, Y location.	Gfx text justify constant.	Gfx.TEXT_JUSTIFY_LEFT
color	The color of the text.	Gfx color constant or a 24-bit integer of the form 0xRRGGBB	The draw context's foreground color.

DRAWABLES

Drawable XML resources consist of a list of basic drawables: both bitmaps and shapes. To create an XML drawable you will define a drawable-list in an XML resource file. Inside the drawable-list you can place both the bitmap and shape tags. An example drawable-list is given below.

```
<resources>
  <drawable-list id="Smiley" background="Gfx.COLOR_YELLOW">
    <shape type="circle" x="10" y="10" radius="5" color="Gfx.COLOR_BLACK" />
    <shape type="circle" x="30" y="10" radius="5" color="Gfx.COLOR_BLACK" />
    <bitmap id="mouth" x="15" y="25" filename="../bitmaps/mouth.png" />
  </drawable-list>
</resources>
```

To use this drawable in code use the following code.

```
function onUpdate(dc) {
  var mySmiley = new Rez.Drawables.Smile();
  mySmiley.draw(dc);
}
```

The drawable-list tag supports the following attributes.

Attribute	Definition	Valid Values	Default Value
id (required)	The ID of the drawable.	Any string that starts with a character.	NA
x	The X coordinate of the top left corner in relation to the parent element.	Pixel value	0
y	The Y coordinate of the top left corner in relation to the parent element.	Pixel value	0
foreground	The color of elements (shapes and text) drawn in this layout.	Gfx Color constant or a 24-bit integer of the form 0xRRGGBB	Current draw context's foreground color.

The shape tag supports the following attributes.

Attribute	Definition	Valid Values	Default Value
type (required)	The type of the shape to be drawn.	rectangle ellipse circle polygon	NA
x	The X coordinate of the top left corner in relation to the parent element.	Pixel value	0
y	The Y coordinate of the top left corner in relation to the parent element.	Pixel value	0
points (required, only for type = polygon)	A list of points which defines the polygon.	[[x1, y1], [x2, y2], ..., [xN, yN]] (must have at least 3 points)	NA
width (only for type = ellipse and rectangle)	The width of the shape to be drawn.	Pixel value or "fill"	fill
height (only for type = ellipse and rectangle)	The height of the shape to be drawn.	Pixel value or "fill"	Fill
color	The color of the shape	Gfx Color constant or a 24-bit integer of the form 0xRRGGBB	Current draw context's foreground color.
corner_radius (only for type = rectangle)	The radius of the rounded corners of a rectangle.	Pixel value	0
radius (required, only for type = circle)	The radius of the circle.	Pixel value	NA
border_width (only for type = rectangle, ellipse and circle)	The width of the border around the shape.	Pixel value	0
border_color (only for type = rectangle, ellipse and circle)	The color of the border to be drawn around the shape.	Gfx Color constant or a 24-bit integer of the form 0xRRGGBB	Current draw context's foreground color.

The bitmap tag supports the following attributes.

Attribute	Definition	Valid Values	Default Value
id (required)	The ID of the drawable.	Any string that starts with a character.	NA
x	The X coordinate of the top left corner in relation to the parent element.	Pixel value	0
y	The Y coordinate of the top left corner in relation to the parent element.	Pixel value	0
filename (required)	The relative path to the image that should be shown.	A valid, relative path.	NA

INPUT HANDLING

As if input handling wasn't already one of the most important and complicated pieces of a UI toolkit, Garmin devices take the complication up a level. Unlike those touchable glowing rectangles that are modern smart phones, Garmin devices come in lots of shapes and sizes. Touch screens are not always ideal for all watch products, and we will have a mix of input styles and screen technologies. It's the job of the UI toolkit to make this coherent to the developer.

INPUT AND APP TYPES

Not all App Types have access to input. Watch faces and data fields cannot handle input. Widgets can receive input but it may be limited on some devices, while watch-apps will have the most input capability.

INPUT DELEGATES

The delegate object is an object that implements a certain interface specific to input handling. Monkey C provides a low level `InputDelegate` that allows handling of events at a basic level. This is good when the app needs to handle button presses and key holds in a particular way.

```
module WatchUi
{
    ///! This class implements basic input handling.
    ///! A developer needs to override
    ///! the events they want to handle.
    class InputDelegate
    {
        ///! Key event
        ///! @param evt KEY_XXX enum value
        ///! @return true if handled, false otherwise
        function onKey(evt);
        ///! Click event
        ///! @param evt Event object with click information
        ///! @return true if handled, false otherwise
        function onTap(evt);
        ///! Screen hold event. Sent if user is touching
        ///! the screen
        ///! @param evt Event object with hold information
        ///! @return true if handled, false otherwise
        function onHold(evt);
        ///! Screen release. Sent after an onHold
        ///! @param evt Event object with hold information
        ///! @return true if handled, false otherwise
        function onRelease(evt);
        ///! Screen swipe event
        ///! @param evt Event
        function onSwipe(evt);
    }
}
```

BEHAVIORS

Garmin makes products with a purpose, and that purpose can alter the design of one product line over another. The decision if a product has a touch screen or has buttons can depend on the environment the user will take it; if the product is intended to be used in water (swimming, canoeing, on a boat), it may not have a touch screen. These decisions make for superior products, but also developer frustration over device fragmentation.

Often all products will support common behaviors (next page, back), but how they are executed by the user may differ based on the available input types. To help with this dilemma, Monkey C provides exposes events at a *behavior* level. Behaviors separate high-level intentions from the actual input type - next page versus screen press. The `BehaviorDelegate` is a super class of `InputDelegate` and maps its low level inputs to common operations across multiple products. Using the `BehaviorDelegate` can lead to much more portable code.

```

    /// BehaviorDelegate handles behavior inputs. A BehaviorDelegate differs from
    /// an InputDelegate in that it acts upon device independent behaviours such as
    /// next page and previous page. On touch screen devices these behaviors might be
    /// mapped to the basic swipe left and right inputs, while on non-touch screen
    /// devices these behaviors might be mapped to actual keys.
    /// BehaviorDelegate extends InputDelegate so it can also act on basic inputs as
    /// well. If a BehaviorDelegate does return true for a function, indicating that
    /// the input was used, then the InputDelegate function that corresponds to the
    /// behavior will be called.
    /// @see InputDelegate
    class BehaviorDelegate extends InputDelegate
    {
        /// When a next page behavior occurs, onNextPage() is called.
        /// @return [Boolean] true if handled, false otherwise
        function onNextPage();

        /// When a previous page behavior occurs, onPreviousPage() is called.
        /// @return [Boolean] true if handled, false otherwise
        function onPreviousPage();

        /// When a menu behavior occurs, onMenu() is called.
        /// @return [Boolean] true if handled, false otherwise
        function onMenu();

        /// When a back behavior occurs, onBack() is called.
        /// @return [Boolean] true if handled, false otherwise
        function onBack();

        /// When a next mode behavior occurs, onNextMode() is called.
        /// @return [Boolean] true if handled, false otherwise
        function onNextMode();

        /// When a previous mode behavior occurs, onPreviousMode() is called.
        /// @return [Boolean] true if handled, false otherwise
        function onPreviousMode();
    }

```

BUILT IN HANDLERS

In general, complicated input from users should not be done on a wearable device, but should instead be done via a phone app or web page. However, sometimes the device needs to get some form of user input. `WatchUi` provides two main widgets to handle input: Menus and the Number Picker.

MENU

A menu is a list of options for the user. The options are displayed in a list that matches the device the app is running on.

A menu can be defined in XML in the resource file using the following format.

```
<menu name="MainMenu">
  <menu-item id="item_1" handlerMethod="menu_handler_method_1">Item 1</menu-item>
  <menu-item id="item_2" handlerMethod="menu_handler_method_2">Item 2</menu-item>
</menu>
```

The resource compiler will then take this XML and generate a function in the Rez module. In order to use this menu, a developer simply needs to define the handler method in their file and then call the Rez module function generated for them.

```
class MyClass extends Ui.Page {
    var menu;

    function initialize() {
        menu = new MainMenu(self);
    }

    function menu_handler_method_1() {
        System.println("item 1 clicked");
    }

    function menu_handler_method_2() {
        System.println("item 2 clicked");
    }

    function openTheMenu() {
        menu.openMenu(Ui);
    }
}
```

NUMBER PICKER

To allow the user to pick or adjust numerical data, the number picker widget is your best bet. The number picker widget allows for editing/adjusting of common numerical types.

```
enum
{
    ///! A Float in meters
    NUMBER_PICKER_DISTANCE,
    ///! A Duration
    NUMBER_PICKER_TIME,
    ///! A Duration
    NUMBER_PICKER_TIME_MIN_SEC,
    ///! A Duration representing the number of seconds since midnight
    NUMBER_PICKER_TIME_OF_DAY,
    ///! A Float in kilograms
    NUMBER_PICKER_WEIGHT,
    ///! A Float in meters
    NUMBER_PICKER_HEIGHT,
    ///! A Number
    NUMBER_PICKER_CALORIES,
    ///! A Number
    NUMBER_PICKER_BIRTH_YEAR
}

///! This is the on-screen representation of number picker.
///! The look-and-feel will be device specific. NumberPickers are pushed
///! using WatchUi.pushView(), providing a NumberPickerDelegate
///! for the input delegate. There are minimum and maximum
///! values enforced by the product for each mode. The initial
///! value will be modified to be within these bounds.
class NumberPicker
{
    ///! Constructor
    ///! @param mode An enum value of type NUMBER_PICKER_*
    ///! @param initialValue The initial value for the Number Picker, type
    ///! depends on mode
    function initialize(mode, initialValue);
}

///! NumberPickerDelegate responds to a number being chosen.
///! This class should be extended to get the chosen number.
class NumberPickerDelegate
{
    ///! When a number is chosen, onNumberPicked() is called, passing the
    ///! chosen value.
    ///! @param value The chosen number, type depends on the NumberPicker mode
    function onNumberPicked(value);
}
```

POSITIONING AND SENSORS



Monkey C provides access to the wearable's GPS sensor.

LOCATION

Location is an abstraction of a coordinate. It exposes the ability to retrieve the coordinates in radians or decimal degrees and then provides a method to convert to coordinate formats supported by the Garmin system. The Position module also exposes string parsing interface to convert from various coordinate formats to a Location object.

```

    ///! The GEO enum is used to retrieve coordinate information in various
    ///! String representations.
    enum
    {
        ///! Degree Format, ddd.ddddd: 38.278652
        GEO_DEG,
        ///! Degree/Minute Format, dddmm.mmm: 38 27.865'
        GEO_DM,
        ///! Degree/Minute/Seconds Format, dddmmss: 38 27' 8"
        GEO_DMS,
        ///! Military Grid Reference System (MGRS): 4QFJ12345678
        GEO_MGRS
    }

    ///! The Location object represents a position. It provides accessor
    ///! methods for retrieving the coordinates in various formats.
    class Location
    {
        ///! Constructor: create a coordinate based off an options hash table
        ///! @param [Dictionary] options Hash table of options
        ///! @option options [Number] :latitude The latitude
        ///! @option options [Number] :longitude The longitude
        ///! @option options [Symbol] :format The format of lat/long (possible
        ///!         values are :degrees, :radians, or :semicircles)
        function initialize(options);

        ///! Use toDegrees() to retrieve the coordinate back as an Array of
        ///! degree values.
        ///! @return [Array] An Array of the latitude and the longitude in degree format
        function toDegrees();

        ///! Use toRadians() to retrieve the coordinate back as an Array of
        ///! radian values.
        ///! @return [Array] An Array of the latitude and the longitude in radian format
        function toRadians();

        ///! Use toGeoString() to get a String representation of the coordinate.
        ///! @param format Coordinate format to which coordinate should be
        ///!         converted (GEO constant)
        ///! @return [String] Formatted coordinate String
        function toGeoString(format);
    }

```

LOCATION EVENTS

To enable the GPS call the `enableLocationEvents` method. The parameters are outlined below:

```
enum
{
    ///! One-time retrieval of Location
    LOCATION_ONE_SHOT,
    ///! Register for Location updates
    LOCATION_CONTINUOUS,
    ///! Unregister for Location updates
    LOCATION_DISABLE
}

///! Request a location event with enableLocationEvents().
///! @param type LOCATION_ONE_SHOT for a single location request,
///!         LOCATION_CONTINUOUS to enable location tracking, and
///!         LOCATION_DISABLE to turn off location tracking
///! @param [Method] listener Method object to call with location updates
function enableLocationEvents(type, listener);
```

To register a position listener, you use the `method()` call to create a Method callback.

```
function onPosition(info)
{
    Sys.println("Position " +
        info.position.toGeoString(Position.GEO_DM));
}

function initializeListener()
{
    Position.enableLocationEvents(
        Position.LOCATION_CONTINUOUS,
        method(:onPosition));
}
```

All of the location information will be sent in an `Info` object.


```

    ///! The Location.Info class contains all information necessary
    ///! for the Location. It can be passed on the update or it
    ///! can be retrieved on demand.
    class Info
    {
        ///! Lat/lon
        var position;
        ///! Speed in meters per second
        var speed;
        ///! Altitude in meters (mean sea level)
        var altitude;
        ///! Accuracy (good, usable, poor, not available)
        var accuracy;
        ///! Heading in radians
        var heading;
        ///! Moment Object: GPS time stamp of fix
        var when;
    }

    ///! Use getInfo() to retrieve the current Location.Info
    ///! @return [Location.Info] The Info object containing the current information
    function getInfo();

```

SENSORS

The Sensor module allows the app to enable and receive information from Garmin ANT sensors. To receive information, you must assign a listener method and enable the sensors.

```

function initialize()
{
    Sensor.setEnabledSensors( [Sensor.SENSOR_HEARTRATE] );
    Sensor.enableSensorEvents( method(:onSensor) );
}

function onSensor(sensorInfo)
{
    System.println("Heart Rate: " + sensorInfo.heartRate);
}

```

Sensor information is packaged in the `Sensor.Info` object.

```

    ///! The Sensor.Info class contains all information necessary
    ///! for the Sensor. It can be passed on the update or it
    ///! can be retrieved on demand.
    class Info
    {
        ///! Speed (m/s)
        var speed;
        ///! Cadence (rpm)
        var cadence;
        ///! HR (BPM)
        var heartRate;
        ///! Temperature (degC)
        var temperature;
        ///! Altitude (m)

```

```
var altitude;
//! Pressure (Pa)
var pressure;
//! Heading (Radians)
var heading;
}
```

The simulator can simulate sensor data via *Fit Data | Simulate*. This generates valid but random values that can be read in via the sensor interface. For more accurate simulation, the simulator can play back a FIT file and feed the input into the Sensor module. To do this, use *Fit Data | Run From File* and choose a FIT file from the dialog.

FIT FILE RECORDING

Monkey C allows for watch-apps to start and stop recording of FIT files. Controlling the FIT file recording takes a few steps:

1. Enable the sensors you wish to record
2. Use `Fit.createSession(options)` to create a session object.
3. Use the `start()` method of the session to begin recording. Data from the enabled sensors will be recorded into the FIT file.
4. Use `stop()` to pause the recording.
5. Use `save()` to persist the recording, or `discard()` to delete the recording.

The FIT file will sync with Garmin Connect. You can use the [Garmin Connect API](#) to process the FIT file from a web service.

COMMUNICATING WITH ANT SENSORS

Connect IQ provides a low level interface for communication with ANT sensors. With this interface you can create an ANT channel as well as send and receive ANT packets. The `MO2Display` sample provides a sample application that implements the Muscle Oxygen ANT profile.

The ANT Generic interface is not available to watch faces, widgets, or data fields.

COMMUNICATION



Widgets and apps can communicate with a mobile phone via Bluetooth Low Energy (BLE). The mobile phone may be sharing data with the device, or it may act as a bridge between the app and the Internet. This allows the mobile phone to become part of the wearable web.

There are a few complications with device to phone communication. For example, the watch app may be killed during the time communication happens, or a phone app may try to send information while the app is not active. In order to simplify these cases for the developer, Monkey C does not expose a low level interface, but instead exposes a very high level approach. Instead of using a socket metaphor, the API exposes a mailbox metaphor. Messages are constructed as a parcel of information and sent back and forth. Each app will have a mailbox where messages are received, and an event that fires when new messages arrive.

There will also be a high level interface for making JSON and image requests. This will allow developers the option of developing a wearable web app without having to write their own companion phone app.

APPROACHES

Monkey C exposes two approaches for communicating with the mobile phone.

GARMIN CONNECT MOBILE

Monkey C exposes some high level APIs to allow calls to basic web services.

```
module Communications
{
    /// To use Garmin Connect Mobile as a proxy to the web, use makeJsonRequest().
    /// The request is asynchronous; the responseCallback will be called
    /// when the request returns.
    /// @param [String] url The URL being requested
    /// @param [Dictionary] request Dictionary of keys and values, appended
    /// to the URL as a GET request
    /// @param [Dictionary] options Dictionary of options
    /// @param [Method] responseCallback This is a callback in the format
    /// function responseCallback(responseCode, data);
    /// responseCode has the server response code, and data contains
    /// a Dictionary of content if the request was successful.
    function makeJsonRequest(url, request, options, responseCallback);

    /// To request an image through Garmin Connect Mobile, call makeImageRequest(). GCM
    /// will scale and dither the image based on the capabilities of the device, but
    /// the user will be able to pass additional options (like dithering it down to a
    /// one color image)
    /// @param [String] url URL of image to request
    /// @param [Dictionary] options Additional image options (TBD)
    /// @param [Method] responseCallback This is a callback in the format
    /// function responseCallback(responseCode, data);
    /// responseCode has the server response code, and data contains
    /// a WatchUi.Bitmap if it was successful.
    function makeImageRequest(url, options, responseCallback);
}
```

These APIs expose JSON requests and image requests as very straightforward APIs for making REST API calls. The JSON calls are converted to serialized Monkey C data and sent across the BLE pipe.

DIRECT MESSAGING

Some watch apps will want to communicate directly with a phone app. For these cases Monkey C exposes a concept of a mailbox. Each app will have its own mailbox that it can retrieve messages from and it can subscribe to an event that fires when a new message comes in.

```

    ///! The MailboxIterator is used to get the messages out of the Mailbox.
    class MailboxIterator
    {
        ///! Call next() to get the next message from the mailbox.
        ///! @return Message content, or null if no messages
        function next();
    }

    ///! Call getMailbox() to get the MailboxIterator for this App's mailbox.
    ///! @return [MailboxIterator] Iterator for the mailbox
    function getMailbox();

    ///! Add a listener for mailbox events. The listener method is called whenever a
    ///! new message is received
    ///! @param [Method] listener Callback in the format function listener(iterator).
    ///! iterator is the mailbox iterator for the app.
    function setMailboxListener(listener);

    ///! To clear the contents of the mailbox, call emptyMailbox().
    function emptyMailbox();

    ///! Send data across the the BLE link.
    ///! @param [Object] The object to be sent
    ///! @param [Dictionary] options Additional transmit options (TBD)
    ///! @param [ConnectionListener] listener An extension of the
    ///! ConnectionListener class.
    function transmit(content, options, listener);

    ///! Listener class for transmit
    class ConnectionListener
    {
        ///! Callback when a communications operation error occurs.
        function onError();

        ///! Callback when a communications operation completes.
        function onComplete();
    }

```

BLE SIMULATION OVER ANDROID DEBUG BRIDGE

When using the Connect IQ Simulator, you can communicate with a companion app running on an Android device using the Android Debug Bridge (ADB). This will simulate actual BLE speeds so you can get a feel for the performance of your application. For information regarding Android Debug Bridge visit: <http://developer.android.com/tools/help/adb.html>

To enable the companion to communicate over ADB you must:

1. Connect the phone to the PC running the simulator via USB.
2. Have USB debugging enabled on the Android handset.
3. Obtain an instance of ConnectIQ using
`getInstance(IQCommProtocol.ADB_SIMULATOR)`
4. Optionally call `setAdbPort(int port)` to set a specific port to use for communication. The default port is 7381.

5. Call `initialize()`.

To allow the simulator to communicate over ADB, in a terminal or console forward the tcp port you are using to the Android device.

```
adb forward tcp:7381 tcp:7381
```

Note that this command will need to be reissued for each Android device you connect, or if you disconnect and re-connect a device.

Once your app is started on the phone, you can connect it to the simulator using the *Connection / Start* (CTRL-F1) menu item. The Connect IQ apps in the simulator will now be able to communicate with your device via the Communications APIs.

DYNAMIC DATA FIELDS



Dynamic Data Fields allow customers and third party developers to write data fields for Garmin activities. The goal is to make a system that not only makes it easy for a user to make a quick data field based off our workout data, but gives the developer the the power to customize the presentation.

DATAFIELD AND SIMPLEDATAFIELD

The base class for data fields is `WatchUi.DataField`. This class extends `WatchUi.View`, and in many ways behaves similarly to other `WatchUi View` objects. The `onUpdate()` call will be made every time the watch needs to update.

In Garmin activities, the user controls the data page layout; specifically if it displays one, two, three or more fields. The field has to handle displaying in all of those layouts, and the developer can use the simulator to test their field in all layouts supported by the device. Many developers will only want to display a single value and not want to handle all the complexity of the drawing of a data field. In those instances, they can use a `SimpleDataField` object. The simple data field handles the drawing of the field in multiple sizes, and it only requires the developer to implement a compute method. The compute method is passed an `Activity.Info` object, which contains all current workout information.

The following is an example of a "Beers Earned" data field. This data field displays how many beers you have "earned" during your workout.

```
using Toybox.Application;
using Toybox.WatchUi;

class BeerView extends WatchUi.SimpleField
{
    function initialize()
    {
        units = "beers";
    }

    function compute(info)
    {
        return info.calories / 150; // Calories in average bottle of beer
    }
}

class BeersEarned extends Application.AppBase
{
    function getInitialView()
    {
        return new BeerView();
    }
}
```

ACTIVITY INFO

The compute method will be called before update. This method allows the developer to run a computation based on the workout information. The workout information is passed in as a `Toybox.Activity.Info` object. This object contains all the info about the workout that is computed by the system.

```
module Activity
{
    class Info
    {
        ///! Starting location of the activity. Location object
        var startLocation;
        ///! Starting time of activity. Moment object
        var startTime;
        ///! Elapsed time of the activity in ms
        var elapsedTime;
        ///! Timer time in ms
        var timerTime;
        ///! Distance in meters
        var elapsedDistance;
        ///! GPS Accuracy (See location constant)
        var currentLocationAccuracy;
        ///! Current location as Location object
        var currentLocation;
        ///! Calories in kcal
        var calories;
        ///! Speed in meters per second
        var currentSpeed;
        ///! Average speed in meters per second
        var averageSpeed;
        ///! Maximum speed in meters per second
        var maxSpeed;
    }
}
```



```

    ///! Current pace in meters per second
    var currentPace;
    ///! Current power in watts
    var currentPower;
    ///! Average power in watts
    var averagePower;
    ///! Maximum power in watts
    var maxPower;
    ///! Total ascent in meters
    var totalAscent;
    ///! Total descent in meters
    var totalDescent;
    ///! Current heart rate in beats per second
    var currentHeartRate;
    ///! Average heart rate in beats per second
    var averageHeartRate;
    ///! Maximum heart rate in beats per second
    var maxHeartRate;
    ///! Current cadence in revolutions per minute
    var currentCadence;
    ///! Average cadence in revolutions per minute
    var averageCadence;
    ///! Maximum cadence in revolutions per minute
    var maximumCadence;
    ///! Swim stroke type
    var swimStrokeType;
    ///! Swim SWOLF
    var swimSwolf;
    ///! Swim effficence
    var swimEffficency;
    ///! Swim stroke average distance
    var averageDistance;
    ///! Current heading in radians
    var currentHeading;
  }
}

```

The following is the API for the data fields:

```

///! The DataField class is used for creating data fields.
///! In a DataField, the developer will implement a compute method,
///! which is called by the system once per second, with an Activity.Info
///! object. The system will call the onUpdate() method inherited
///! from View when the field is displayed by the system.
class DataField extends View
{
    ///! To retrieve Activity.Info data for DataFields, it is necessary to override compute().
    ///! This is the method that is called to update the field information.
    ///! @param [Activity.Info] info The updated Activity.Info object
    function compute(info);
}

///! The SimpleDataField class is used for creating simple data fields.
///! In a SimpleDataField, the developer is only required to implement a
///! compute method. The compute method is passed an Activity.Info
///! object, which contains all current workout information. The compute
///! method should return a value to be displayed. Allowed types are
///! Number, Float, Long, Double, and String. The SimpleDataField also
///! contains a variable "label". This variable should be assigned to
///! a String label for the field.
class SimpleDataField extends DataField
{
    ///! Constructor
    function initialize();

    ///! To retrieve Activity.Info data for DataFields, it is necessary to override compute().
    ///! This is the method that is called to update the field information.
    ///! @param [Activity.Info] info The updated Activity.Info object
    function compute(info);
}

```

SIMULATING A WORKOUT

To test your data field in the simulator, you can feed your data field simulated data. Use *FIT Data | Simulate* to generate random but valid data. You can also use *FIT Data | Run From File...* to simulate a workout by using a FIT file.

HOW TO PUBLISH

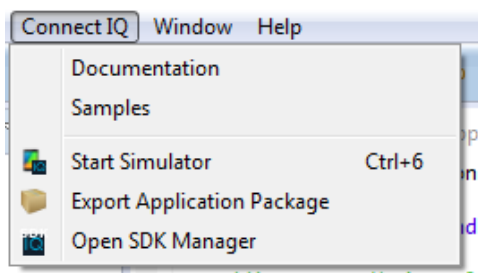


Note: The app store is scheduled for January 2015

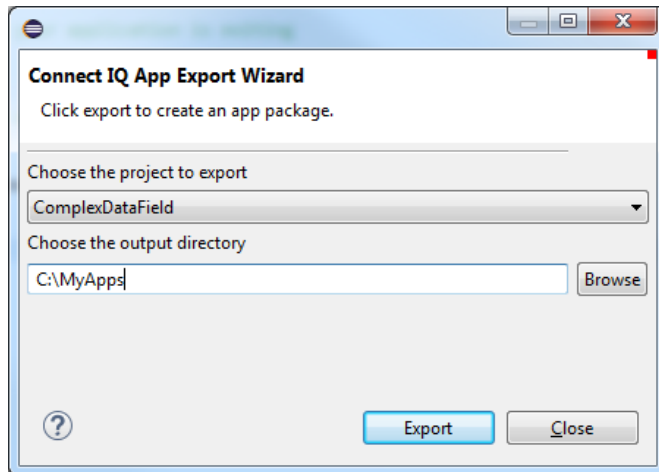
Once your apps are fully tested and ready to go, you can publish and promote your apps. Here's how to prepare your apps for submission.

- Double check your App Manifest specifies all the products you wish to support
- Use the Package Tool to generate your `.iq` file that contains the binaries for all of the products you are supporting.

You can find the Package Tool in the Connect IQ Eclipse plug-in under Export Application Package:



After you select it you will be presented a wizard to create the IQ file.



Hit Export. If there are no errors your apps are now ready to publish!

PUBLISHING THE FIRST VERSION

The first step is to upload your IQ file to your developer account.

Upload an App

[Step 1: Attach File](#) > [Step 2: Enter Details](#)

App File

File should be ZIP format, and additional requirements. You'll enter your description information once the file is verified.

File Path

 [Choose file](#)

App Version (Maximum 20 Characters)

By submitting this app, you agree to our [Developer License Agreement and Terms of Use](#).

[Continue](#)
[Cancel](#)

Once the binary has been validated you will be able to add in the description and screen shots. Be very specific in your description. You want people to download your app.

Title and Description

Select a Language
Add

English

Title (Maximum 50 Characters)

Description (Maximum 4,000 Characters)

Category

Choose the category that best fits your app. Categories are used to organize apps in the Connect IQ Store.

Category

Entertainment

Subcategory (Optional)

Select a Subcategory

RULES

We're glad you want to develop for Garmin devices, and we want you to be successful. Here are some guidelines that you should keep in mind as you develop your apps. These guidelines will help you speed through the approval process:

- We will reject submissions for any content or behavior that we believe is inappropriate.
- If your submission doesn't function or crashes frequently, it may not be accepted.
- Your submission cannot use GPS data in a way that violates our developer terms of use.
- Do not use copyrighted content without permission.
- These guidelines may change at any time.

APPROVAL PROCESS

After you upload your app successfully, we will review it. You will be able to preview your app and download it yourself for testing. While approval is pending, your app will not appear on the Garmin Connect App Library. Once it's approved, we will notify you. Then it will appear on the Connect App Library for all users to download and send to devices.

