

HotSpot How To

! Deprecated

This page is deprecated and its content is either old or being moved to the [OpenJDK Developers' Guide](#). Please update your links.

Running tests and pushing changes

Read about the [Submit repo here](#).

! Deprecated

This page is deprecated and its content is either old or being moved to the [OpenJDK Developers' Guide](#). Please update your links.

Before pushing

In order to push a change to the HotSpot source code you need to complete all the steps in the list below. This is done in order to ensure code quality and reduce the risk of introducing bugs into the code base.

1. You must be a Committer in the [JDK project](#)
2. You need a non-JEP [JBS issue](#) for tracking
3. Your change must have been available for review at least 24 hours to accommodate for all time zones
4. Your change must have been approved by two Committers out of which at least one is also a Reviewer
5. Your change must have passed through the hs tier 1 testing provided by the [submit-hs repository](#) with zero failures
6. You must run all relevant testing to make sure your actual change is working
7. You must be available the next few hours, and the next day and ready to follow up with any fix needed in case your change causes problems in later tiers

There is a notion of *trivial changes* that can be pushed sooner than 24 hours. It should be clearly stated in the review mail that the intention is to push as a trivial change. How to actually define "trivial" is decided on a case-by-case basis but in general it would be things like fixing a comment, or moving code without changing it. Backing out a change is also considered trivial as the change itself in that case is generated by mercurial.

Relevant testing

Please note that the submit repository will only run a set of smoke-tests to ensure your change compiles and runs on a variety of platforms. It will not do any targeted testing on the particular code you have changed. Running through the submit repository is only the minimum requirement. You must also make sure your change works as expected before pushing using targeted testing. Consider writing a few JTREG tests for your change, or some unit tests using the GTest framework. Including the new tests (in the right places) with your push to the submit repository will ensure your tests will be run as part of your testing on all platforms and in the future. Look for tier1 in `test/hotspot/jtreg/TEST.groups` to see which tests and directories that are included in the submit repo testing.

The push

NOTE: This section is out of date since the move to using Git and GitHub for the main OpenJDK development project.


Pushing a change is fairly straight forward. Make sure your commit has a proper description. The JBS bug id and the Reviewed-by lines are mandatory.

```
8197844: JVMTI GetLoadedClasses should use the Access API
Summary: Make sure the holder of a class loader is accessed during iteration of CLDG
Reviewed-by: eosterlund, rkennke
```

! Always make sure there are no new changes in the repository you are pushing to before pushing

`hg in` should return an empty set of changes when you push. Always verify this the last thing you do! If there are new changes in there you must pull these changes and rebase your patch on top of these new changes. We use rebase to avoid merge changesets as we want to keep the mercurial history as linear as possible. This makes it easier to do binary searches through the history when trying to determine the cause of some changed behavior for instance (read debugging).

```
hg pull -u --rebase
```

 Look through the new changes before you push your change. Did something recently pushed conflict with your change? There may be a need to rerun some testing or at least recompile before pushing.

When you feel confident enough, push using `hg push`.


After pushing

As noted in the list above, you are expected to be around after having pushed a change in case there are any issues with it. A change that causes failures in later tiers may be backed out if a fix can not be provided fast enough, or if the developer is not responsive when noticed about the failure. Note that #7 above should be interpreted as "it is a really bad idea to push a change the last thing you do before bedtime, or the day before going on vacation".

Test analysis and bug handling


Deprecated

This page is deprecated and its content is either old or being moved to the [OpenJDK Developers' Guide](#). Please update your links.

 A critical failure is a test failure that has a high impact on the daily work for developers.


This can mean a few different things:

- **Any failure in tier 1 is considered critical.** A single failure, no matter how isolated or trivial, will cause the entire tier to signal failure. As long as there are tier 1 failures, every developer that pushes a change must dig into the results to see what caused the failure. Human nature dictates that after a while one will assume that the failure is the same as it was in the previous run, and at that point tier 1 testing is useless. Unless the results are checked every single time we will not notice when a new failure appears.
- **Any failure that is blocking integration is considered critical.** [Integration blockers](#) cause delays in propagation of other changes. People outside of the HotSpot team might be waiting for a change to propagate to master. The performance team needs HotSpot changes to get into promoted builds on a regular basis. In short, [integration blockers](#) stop others from doing their work.
- **Any failure that causes multiple unrelated tests to fail is considered critical.** Bugs with a large damage area will hide other bugs and make it harder to fix those other bugs once they are found. Tests that cause massive failures in the nightly will seriously reduce our test coverage.

 Please note that there is a significant difference between high priority and high urgency. High priority issues like vulnerabilities for example will always have higher priority than regular bug fixing. Critical issues are urgent, but do not always have high priority. It could be a trivial bug in a test that has no particular significance, a test that nobody cares whether it is fixed or not. But, if it causes failures in tier 1 it is urgent to fix it. Not high priority but still urgent. It is most likely urgent enough to take 20 minutes away from fixing a vulnerability.

Handling a critical failure

It is required that anyone who pushes a change monitors the tier 1 testing and the following nightly, or designates this requirement to someone who can also fix possible regressions. It is considered exceptionally bad engineering to push a change at the end of the day, or on the last day before going on vacation or otherwise becoming unavailable. Critical failures must be handled with high urgency, thus the engineer who caused the failure is best suited to do so.

 If a failure in **tier 1** is identified and can be fixed within **two - three hours**, this is the preferred way to go. If a fix is not expected within that time the broken change **must be backed out**.

Integration blockers must be fixed before the next integration per definition. Usually this means that a fix should be available **within a few days**.

Bugs that causes unrelated tests to fail in the nightly should be fixed **before the next nightly**. If a fix can not be delivered within **two days** the causing change **must be backed out**.

Backing a change out is easy. Anyone can do it.

There can never be any ownership involved when it comes to backing out a change. It doesn't matter who made the change or why it fails, if it needs to be backed out it should be backed out. Whoever is on site first should do it. Usually there is some synchronization before doing the actual work to avoid duplicate work.

Use mercurial to create a backout changeset:


```
hg backout -r <revision_id>
```

A backout is considered a trivial change so a single Reviewer is enough and you can push immediately when reviewed. See [How to backout a change](#) for details on bug management in JBS related to backing out changes.

Deprecated


This page is deprecated and its content is either old or being moved to the [OpenJDK Developers' Guide](#). Please update your links.

The HotSpot team owns two components in JBS: hotspot and core-svc.

 Please note that all issues in the hotspot component are required to have a subcomponent set.

Team	JBS component	JBS query
Compiler	hotspot / compiler	component = hotspot AND Subcomponent = compiler
GC	hotspot / gc	component = hotspot AND Subcomponent = gc
Runtime	hotspot / runtime hotspot / jfr	component = hotspot AND Subcomponent in (runtime, jfr)
Serviceability	hotspot / jvmti hotspot / svc core-svc core-svc / debugger core-svc / tools core-svc / java.lang. instrument tools / hprof	((component = hotspot AND Subcomponent in (jvmti, svc)) OR (component = core-svc AND (Subcomponent is EMPTY OR Subcomponent in (debugger, tools, java. lang.instrument))) OR (component = tools AND Subcomponent = hprof))
Monitor and Management	hotspot / svc-agent core-svc / java.lang. management core-svc / javax. management	((component = hotspot AND Subcomponent = svc-agent) OR (component = core-svc AND (Subcomponent in (java.lang.management, javax.management))
Test	hotspot / test	component = hotspot AND Subcomponent = test

The **integration blocker** label should be used to indicate that a bug is present in a repository but not present upstreams. When filing a new bug found in hotspot testing, the default should be to add the integration_blocker label. If it is verified (when filing or later) that the new failure is already present upstreams from the repository where it was found the label should be removed and an explicit comment must be added to the bug explaining why the bug is not an integration blocker. A failure seen in a project repo is considered escaped when found in `jdk/hs`. A failure seen in `jdk/hs` is considered escaped if seen in `jdk/jdk` or if older failures are found from a date before the last integration to `jdk/jdk`.

 If an integration blocker escapes from a project repository it will become an integration blocker in `jdk/hs`.

The **testbug** label should be used for bugs in tests and test infrastructure. There is no fundamental difference between a test bug and a product bug in the eyes of the gatekeeper. Both are considered integration blockers and both should be handled asap when appearing in the testing.

There are a few ways to figure out if a problem has already escaped a repository:

- The easiest way is of course if you know which change that caused the problem. Just have a look at the hg log of the [upstream repository](#) to see if the change is in there.

- Searching JBS can give information as well. See if there is more than one bug filed for the issue. If it escaped it is likely that someone else has found it and maybe filed a new bug.

A bug that has escaped to an upstream repository should be fixed in the upstream repository

If a bug has escaped from a project repository into `jdk/hs`, the fix/backout/quarantine should be done in `jdk/hs` to avoid blocking integration for other Hotspot changes. If a bug escapes `jdk/hs` into `jdk/jdk` the fix must go into `jdk/jdk`.

Integration

Deprecated

This page is deprecated and its content is either old or being moved to the [OpenJDK Developers' Guide](#). Please update your links.

Most HotSpot development work is done in `jdk/hs`. Larger projects are recommended to use a project repository which is synced with `jdk/hs` on a regular basis.

When is a change integrated?

Changes in `jdk/hs` are bulk integrated to `jdk/jdk` once a week. A change in HotSpot will go through these steps on the way to `jdk/jdk`:

1. After proper testing HotSpot developers push changes to the HotSpot repo.
 - Pushes are direct and done using `hg push`
2. When a push is done **Continuous Integration (CI)** will automatically start a set of test jobs on the HotSpot repo.
 - These tests include `hs-tier1`, `hs-tier2`, `jdk-tier1`, `jdk-tier2`, `jdk-tier3`, and `builds-tier1`.
 - It is obviously a good idea to have run the relevant tests before pushing to ensure that no failures happen in CI.
 - If a CI job is already running, the next job will start once the current is done. If several pushes have been made in this time, all will be tested in the same CI job.
 - CI failures are considered **critical failures** and must be dealt with immediately.
3. Every weeknight (Mon - Fri) a **nightly test** job is started at **8 pm PT**.
 - The nightly testing runs `hs-tier3`, `hs-tier4`, and `hs-tier5`.
 - New failures that are encountered in the nightly testing are always labeled as **integration blockers**.
4. Each Friday the changes included in the most recent clean nightly are merged with the most recent clean CI build of master. This merge is then sent to **PIT** (Product Integration Testing).
 - A clean nightly is one where no integration blockers were open when the nightly started, and no new ones were found in the same.
 - If all integration blockers in a nightly snapshot have been fixed before the PIT is integrated, that snapshot can be chosen provided the extra fixes are included in the integration.
5. Once the PIT result has been analysed (usually early the next week), and no integration blockers were found, the merge is **pushed to master** (using `hg push`)

All in all this means that if you need a change to be in master at a specific date, you need to make sure your change is pushed to `jdk/hs` in time to go through the PIT and be pushed to master before the actual date.

What is an integration blocker?

An integration blocker is **a bug that is present in a repository, but has not yet spread to the upstream repository**. We want to make sure not to spread problems, so if there are new failures in a repository, hold upstream integration.

There is a JBS label that is used to help keep track of bugs that are blocking integration, `integration_blocker`.

Never integrate if there are open integration blockers

Use the JBS filter [Hotspot Integration blockers](#) to find out if there are any blockers.

If a main bug is targeted to a release and the fix is pushed to a different release, then a backport bug is automatically created. Usually this is a "good thing", e.g., when you are really backporting a fix to an earlier release, but not always... If the main bug is targeted to a later release (due to schedule planning), but someone finds the time to fix that bug in the current release, then the bug should be retargeted to the current release before pushing the fix. However, sometimes we forget to do that.

Here is how to fix that:

 In this example a fix was pushed to JDK 10 (a.k.a. the current release) while the JBS bug was targeted to JDK 11 (a.k.a. a future release).

1. **Reopen the *backport* bug that was created automatically**

- Use a comment like the following (in the reopen dialog):

```
Fix was pushed while main bug was targeted to '11'. Reset the main bug to fixed in '10', reset this bug to fix in '11' and closed as 'Not An Issue' since JDK 11 will automatically get this fix from JDK 10.
```

- Change the 'Fix Version/s' from '10' to '11'.
- Close the *backport* bug as "Not an Issue".

2. **Clean up the *main* bug**

- Copy the open push notification comment from the *backport* bug to the *main* bug, e.g.:

```
HG Updates added a comment - 6 hours ago
URL: http://hg.openjdk.java.net/jdk/jdk/rev/8db54e2c453b
User: clanger
Date: 2017-12-11 07:25:20 +0000
```

- Add a comment like the following to the *main* bug:

```
Fix was pushed while main bug was targeted to '11'. Reset the main bug to fixed in '10' and copied the hgupdater entry here.
```

- Reset the *main* bug 'Fix Version/s' from '11' to '10'.
- Resolve the *main* bug as "Fixed" in build "team" or in build "master" depending on where the fix was pushed. Pushes to 'jdk/jdk' are fixed in build "master" and pushes to other repos are fixed in build "team".

There are several ways to find out if a change has been integrated to `jdk/jdk` or not. The easiest way is probably to look at the mercurial log in `jdk/jdk` and see if your change is there:

```
hg log | grep 1234567
```

Or, if you don't have a recent clone of `jdk/jdk` available, look at <http://hg.openjdk.java.net/jdk/jdk/>